THE EXPERT'S VOICE® IN WEB DEVELOPMENT

Practical JavaScript™, DOM Scripting, and Ajax Projects

Learn advanced JavaScript™ techniques by example to build superior web applications

Frank W. Zammetti

Apress®

PART 1

Say Hello to My Little Friend: JavaScript!

Eaten any good books lately?

Q (to Worf) in the *Star Trek: The Next Generation* episode, "Deja-Q"

The Internet? Is that thing still around?

Homer Simpson

Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning.

Rich Cook

The first 90% of the code accounts for the first 10% of the development time. The remaining 10% of the code accounts for the other 90% of the development time.

Tom Cargill

There are only two kinds of programming languages: those people always bitch about and those nobody uses.

Bjarne Stroustrup

There are only two industries that refer to their customers as 'users.'

Edward Tufte

Practical JavaScript™, DOM Scripting, and Ajax Projects

Frank W. Zammetti

Apress®

Practical JavaScript™, DOM Scripting, and Ajax Projects

ISBN-13 (pbk): 978-1-59059-816-0

ISBN-10 (pbk): 1-59059-816-4

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

Lead Editor: Matthew Moodie
Technical Reviewer: Herman van Rosmalen
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Jeff Pepper, Paul Sarknas, Dominic Shakeshaft, Jim Sumser, Matt Wade
Project Manager: Tracy Brown Collins
Copy Edit Manager: Nicole Flores
Copy Editor: Marilyn Smith
Assistant Production Director: Kari Brooks-Copony
Production Editor: Laura Esterman
Compositor: Susan Glinert
Proofreaders: Lori Bring and April Eddy
Indexer: Broccoli Information Management
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 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code/Download section.

Dedicated to all the animals I've eaten over the years, without whom I most certainly would have died a long time ago due to starvation. Well, I suppose I could have been a vegan, but then I'd have to dedicate this to all the plants I've eaten, and that would just be silly because very few plants can read.

To all my childhood friends who provided me with cool stories to tell: Joe, Thad, Meenie, Kenny, Franny, Tubby, Stubby, Kenway, JD, dVoot, Corey, and Francine.

To Denny Crane, for raising awareness of Mad Cow disease.

Hmm, who am I forgetting? Oh yeah, and to my wife and kids. You guys make life worth living.

Contents at a Glance

Contents

PART 2 The Projects

About the Author

FRANK W. ZAMMETTI is a web architect specialist for a leading worldwide financial company by day, and a PocketPC and open source developer by night. He is the founder and chief software architect of Omnytex Technologies, a PocketPC development house.

Frank has more than 13 years of "professional" experience in the IT field, and over 12 more of "amateur" experience. He began his nearly lifelong love of computers at age 7, when he became one of four students chosen to take part in the school district's pilot computer program. A year later, he was the only participant left! The first computer Frank owned was a Timex Sinclair 1000, in 1982, on which he wrote a program to look up movie times for all of Long Island (and without the 16kb expansion module!). After that, he moved on to an Atari computer, and then a Commodore 64, where he spent about four years doing nothing but assembly programming (games mostly). He finally got his first IBM-compatible PC in 1987, and began learning the finer points of programming (as they existed at that time!).

Frank has primarily developed web-based applications for about eight years. Before that, he developed Windows-based client/server applications in a variety of languages. Frank holds numerous certifications, including SCJP, MCSD, CNA, i-Net+, A+, CIW Associate, MCP, and numerous BrainBench certifications. He is a contributor to a number of open source projects, including DataVision, Struts, PocketFrog, and Jakarta Commons. In addition, Frank has started two projects: Java Web Parts and The Struts Web Services Enablement Project. He also was one of the founding members of a project that created the first fully functioning Commodore 64 emulator for PocketPC devices (PocketHobbit).

Frank has authored various articles on topics that range from integrating DataVision into web applications to using Ajax in Struts-based applications, as well as a book on Ajax for Apress. He is currently working on a new application framework specifically geared to creating next-generation web applications.

Frank lives in the United States with his wife Traci, his two kids Andrew and Ashley, and his dog Belle. And an assortment of voices in his head, but the pills are supposed to stop that.

About the Technical Reviewer

HERMAN VAN ROSMALEN works as a developer/software architect for De Nederlandsche Bank N.V., the central bank of the Netherlands. He has more than 20 years of experience in developing software applications in a variety of programming languages. Herman has been involved in building mainframe, PC, and client/server applications. For the past six years, however, he has been involved mainly in building J2EE web-based applications. After working with Struts (pre-1.0) for years, he got interested in Ajax and joined the Java Web Parts open source project in 2005.

Herman lives in a small town, Pijnacker, in the Netherlands, with his wife Liesbeth and their children, Barbara, Leonie, and Ramon.

About the Illustrator

ANTHONY VOLPE did the illustrations for this book and the K&G Arcade game. He has worked on several video games with author Frank Zammetti, including Invasion Trivia!, Io Lander, and Ajax Warrior. Anthony lives in Collegeville, Pennsylvania, and works as a graphic designer and front-end web developer. His hobbies include recording music, writing fiction, making video games, and going to karaoke bars to make a spectacle of himself.

Acknowledgments

Many people helped make this book a reality in one form or another, and some of them may not even realize it! I'll try to remember them all here, but chances are I haven't, and I apologize in advance.

First and foremost, I would like to thank everyone at Apress who made this book a reality. This is my second go-round with you folks, and it was just as pleasurable an experience this time as the first. Chris, Matt, Tracy, Marilyn, Laura, Tina, and all the rest, thank you!

A great deal of thanks goes to Herman van Rosmalen, one of my partners in crime on the Java Web Parts project (`http://javawebparts.sourceforge.net`) project, and technical reviewer for this book. I know you put in a lot of time and effort in keeping me honest, and I can't tell you how much I appreciate it! Now, let's get back to work on JWP!

A big thanks must also go to Anthony Volpe, the fine artist who did the illustrations for this book. He and I have been friends for about ten years now, and we have collaborated on a number of projects, including three PocketPC games (check 'em out: `http://www.omnytex.com`), as well as a couple of Flash games (`http://www.planetvolpe.com/crackhead`) and some web cartoons (`http://www.planetvolpe.com/du`). He is a fantastic artist, as I'm sure you can see for yourself, an incredibly creative person, and a good friend to boot.

I would also like to thank those that built some of the libraries used in this book, including all the folks working on Dojo, Sam Stephenson (Prototype), Aaron Newton, Christophe Beyls, and Valerio Proietti of the Mootools team; Bob Ippolito of MochiKit fame; all the YUI developers; and everyone working on script.aculo.us and Rico.

Last but most definitely not least, I would like to thank everyone who bought this book! I sincerely hope you have as much fun reading it as I did writing it, and I hope that you find it to be worth your hard-earned dollars and that it proves to be an educational and eye-opening experience.

As I said, I know I am almost certainly forgetting a boatload of people, so how about I just thank the entire world and be done with it?!? In fact, if I had the technology, I'd be like Wowbagger the Infinitely Prolonged, only with "Thanks!" instead of insults.

And on that note, let's get to some code!

Introduction

So there I was, just minding my own business, when along came a publisher asking me if I'd be interested in writing a book on JavaScript. It seemed like a good thing to do at the time, so I said yes.

I'm just kidding. No one asked me, I just showed up one day on the doorstep of Apress with a manuscript and some puppy-dog eyes. I'm just kidding again.

Seriously though, JavaScript is one of those kids we all knew when we were young who start out really ugly, but whom everyone wants as their beautiful date to the prom years later. Then they go on to Yale, become a district attorney, and suddenly everyone realizes that they really want to be with that person. Fortunately, unlike the DA, JavaScript doesn't involve crimes and misdemeanors, since you know you don't have a chance any other way with the DA!

JavaScript has quickly become one of the most important topics in web development, one that any self-respecting web developer can't do without. With the advent of Ajax, which I'll talk about in this book, JavaScript has very quickly gone from something that can enhance a web site a little to something used to build very serious, professional-quality applications. It's no longer a peripheral player; it's a main focus nowadays.

There are plenty of books on JavaScript and plenty of how-to articles strewn across the intrawebs, any of which can be of great help to you. Far harder to come by though are real, substantial examples. Oh, you can get a lot of simplistic, artificial examples to be sure, but it's more difficult to find full-blown, real-world applications that you can examine. Many developers learn best by tearing apart code, messing around with it a bit, and generally getting their hands dirty with real, working bits. That's why I wrote this book: to fill that gap.

In this book, you will find two chapters on some general JavaScript topics, including a brief history of JavaScript, good coding habits, debugging techniques, tools, and more. From then on, it's ten chapters of nothing but projects! Each chapter will present a different application, explain its inner workings, and offer some suggested exercises you can do to sharpen your skills and further your learning. The projects run the gamut from generally useful (an extensible calculator) to current ideas (a mashup) to just plain fun (a JavaScript game).

In the process, you will learn about a wide variety of topics, including debugging techniques, various JavaScript libraries, and a few somewhat unique and useful approaches to coding. I believe you will also find this to be an entertaining book, and in fact, one of the exercises I suggest from the beginning is to try to pick out all the pop-culture references scattered all over the place (try to place them without looking at the footnotes that accompany most, but not all!). I tried to make this book like an episode of *Gilmore Girls* in that regard (and if you aren't familiar with the show, there's your first pop-culture reference!).

So, enough babbling (for the time being anyway). You know what's coming, so let's stop dropping hints about numbers, Dharma, and bizarre connections between characters (pop-culture reference number 2!), and get on with the good stuff. Let's get on with the show!

An Overview of This Book

This book is divided into two main parts. Part 1, "Say Hello to My Little Friend: JavaScript!," contains two chapters:

- Chapter 1 is a brief history of JavaScript, from its humble beginning to its current state of acceptance.
- Chapter 2 goes into the techniques and approaches employed by modern-day "professional" JavaScript developers.

Part 2, "The Projects," contains ten chapters:

- Chapter 3 starts you off with the first project: an extensible, packaged collection of utility functions.
- Chapter 4 develops an extensible calculator and introduces the first JavaScript library, Rico.
- Chapter 5 introduces the concept of a mashup, one of the hottest topics going today, by way of a working example using the very popular script.aculo.us library.
- Chapter 6 uses the Dojo library to deal with an issue that comes up frequently in JavaScript development, that of client-side data persistence.
- Chapter 7 explores the very useful JSDigester component of the Java Web Parts project, which allows you to parse XML and create JavaScript objects from it without tedious coding on your part.
- Chapter 8 develops an extensible validation framework for doing client-side form validation in a purely declarative fashion.
- Chapter 9 introduces the Yahoo! User Interface Library and uses it to create a handy little contact manager application.
- Chapter 10 uses the MochiKit library to develop a drag-and-drop shopping cart for e-commerce applications.
- Chapter 11 is where we get into the fun stuff: a JavaScript game! And not a simple little Tetris clone or tile-matching game, but something a fair bit more substantial.
- Chapter 12 is where we have an in-depth look at Ajax, perhaps the biggest reason JavaScript has taken on a whole new level of importance in recent years, using the relatively new Mootools library.

Obtaining This Book's Source Code

All the examples in this book are freely available from the Source Code section of the Apress web site. In fact, due to the nature of this book, you will absolutely *have* to download the source code before you begin Chapter 3. To do so, visit `http://www.apress.com`, click the Source Code link, and find *Practical JavaScript, DOM Scripting, and Ajax Projects* in the list. From this book's home page, you can download the source code as a zip file. The source code is organized by chapter.

Obtaining Updates for This Book

Writing a book is a big endeavor—quite a bit bigger than many people think! Contrary to what I claim in private to my friends, I am not perfect. I make my mistakes like everyone else. Not in this book of course. Oh no, none at all.

Ahem . . .

Let me apologize in advance for any errors you may find in this book. Rest assured that everyone involved has gone to extremes to ensure there are none, but let's be real here. We've all read technical books before, and we know that the cold, sharp teeth of reality bite every now and again. I'm sorry, I'm sorry, I'm sorry!

A current errata list is available from this book's home page on the Apress web site (`http://www.apress.com`) along with information about how to notify us of any errors you may find. This will usually involve some sort of telepathy, but my understanding is that Windows Vista Service Pack 1 will include this feature, so rest easy my friends.

Contacting the Author

I very much would like to hear your questions and comments regarding this book's content and source code examples. Please do feel free to email me directly at `fzammetti@omnytex.com` (spammers *will* be hunted down by Sentinels and disposed of). I will reply to your inquiries as soon as I can, but please remember, I do have a life (no, really, I do . . . OK, no I don't), so I may not be able to reply immediately.

Lastly, and most important, thank you for buying this book! I thank you, my wife thanks you, my kids thank you, my kids' orthodontist thanks you, my dog's veterinarian thanks you, my roofing contractor thanks you . . .

PART 1

Say Hello to My Little Friend: JavaScript!

Eaten any good books lately?

Q (to Worf) in the *Star Trek: The Next Generation* episode, "Deja-Q"

The Internet? Is that thing still around?

Homer Simpson

Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning.

Rich Cook

The first 90% of the code accounts for the first 10% of the development time. The remaining 10% of the code accounts for the other 90% of the development time.

Tom Cargill

There are only two kinds of programming languages: those people always bitch about and those nobody uses.

Bjarne Stroustrup

There are only two industries that refer to their customers as 'users.'

Edward Tufte

CHAPTER 1

A Brief History of JavaScript

I can only hope Stephen Hawking doesn't mind me paraphrasing his book title as the title of this chapter![1] Just as in his book *A Brief History of Time*, we are about to begin an exploration of a universe of sorts, from its humble beginnings to its current state of being.

In this chapter, we will explore the genesis of JavaScript. More than providing a mere history lesson though, in the tradition of Mr. Hawking himself, I'll give you a deeper look and show what's below the surface. In the process, you'll gain an understanding of the problems inherent in early JavaScript development and how those flaws have largely been overcome. By the end of our journey, you'll have a good understanding of the pitfalls to avoid and start to know how to overcome them (the rest of that knowledge will be revealed in subsequent chapters). So, let's get ready for an adventure, and let's do Mr. Hawking proud!

How JavaScript Came to Exist

The year was 1995, and the Web was still very much in its infancy. It's fair to say that the vast majority of computer users couldn't tell you what a web site was at that point, and most developers couldn't build one without doing some research and learning first. Microsoft was really just beginning to realize that the Internet was going to matter. And *Google* was still just a made-up term from an old Little Rascals episode.[2]

Netscape ruled the roost at that point, with its Navigator browser as the primary method for most people to get on the Web. A new feature at the time, Java applets, was making people stand up and take notice. However, one of the things they were noticing is that Java wasn't as accessible to many developers as some (specifically, Sun Microsystems, the creator of Java) had hoped. Netscape needed something more.

---

1. *A Brief History of Time* is the title of one of the most famous books on physics and cosmology ever written, and is the obvious, ahem, inspiration, for the title of this chapter. Its author, Professor Stephen Hawking of the University of Cambridge, is considered one of the world's best theoretical physicists. His book brought many of the current theories about the universe to the layman, and those of us that pretend we actually know what we're talking about when discussing things like superstrings, supersymmetry, and quantum singularities (outside a *Star Trek* episode, that is!). For more information, see `http://en.wikipedia.org/wiki/Stephen_Hawking`.
2. The word *google* was first used in the 1927 Little Rascals silent film *Dog Heaven*, to refer to having a drink of water. See `http://experts.about.com/e/g/go/Google.htm`. Although this reference does not state it was the first use of the word, numerous other sources on the Web indicate it was. I wouldn't bet all my money on this if I ever made it to the finals of *Jeopardy*, but it should be good enough for polite party conversation!

Enter Brendan Eich, formerly of MicroUnity Systems Engineering, a new hire at Netscape. Brendan was given the task of leading development of a new, simple, lightweight language for non-Java developers to use. Many of the growing legions of web developers, who often didn't have a full programming background, found Java's object-oriented nature, compilation requirements, and package and deployment requirements a little too much to tackle. Brendan quickly realized that to make a language accessible to these developers, he would need to make certain decisions. Among them, he decided that this new language should be loosely typed and very dynamic by virtue of it being interpreted.

The language he created was initially called LiveWire, but its name was pretty quickly changed to LiveScript, owing to its dynamic nature. However, as is all too often the case, some marketing drones got hold of it and decided to call it JavaScript, to ride the coattails of Java. This change was actually implemented before the end of the Navigator 2.0 beta cycle.[3] So for all intents and purposes, JavaScript was known as JavaScript from the beginning. At least the marketing folks were smart enough to get Sun involved. On December 4, 1995, both Netscape and Sun jointly announced JavaScript, terming it "complementary" to both HTML and Java (one of the initial reasons for its creation was to help web designers manipulate Java applets easier, so this actually made some sense). The shame of all this is that for years to come, JavaScript and Java would be continually confused on mailing lists, message boards, and in general by developers and the web-surfing public alike!

It didn't take long for JavaScript to become something of a phenomenon, although tellingly on its own, rather than in the context of controlling applets. Web designers were just beginning to take the formerly static Web and make it more dynamic, more reactive to the user, and more multimedia. People were starting to try to create interactive and sophisticated (relatively speaking) user interfaces, and JavaScript was seen as a way to do that. Seemingly simple things like swapping images on mouse events, which before then would have required a bulky browser plug-in of some sort, became commonplace. In fact, this single application of JavaScript—flipping images in response to user mouse events—was probably the most popular usage of JavaScript for a long time. Manipulating forms, and, most usually, validating them, was a close second in terms of early JavaScript usage. Document Object Model (DOM) manipulation took a little bit longer to catch on for the most part, mostly because the early DOM level 0, as it came to be known, was relatively simplistic, with form, link, and anchor manipulation as the primary goals.

In early 1996, shortly after its creation, JavaScript was submitted to the European Computer Manufacturers Association (ECMA) for standardization. ECMA (`http://www.ecma-international.org`) produced the specification called ECMAScript, which covered the core JavaScript syntax, and a subset of DOM level 0. ECMAScript still exists today, and most browsers implement that specification in one form or another. However, it is rare to hear people talk about ECMAScript in place of JavaScript. The name has simply stuck in the collective consciousness for too long to be replaced. And, of course, this book itself is about *JavaScript*, not ECMAScript. But do be clear about it: they are the same thing!

What made JavaScript so popular so fast? Probably most important was the very low barrier to entry. All you had to do was open any text editor, type in some code, save it, and load that file in a browser, and it worked! You didn't need to go through a compilation cycle or package and

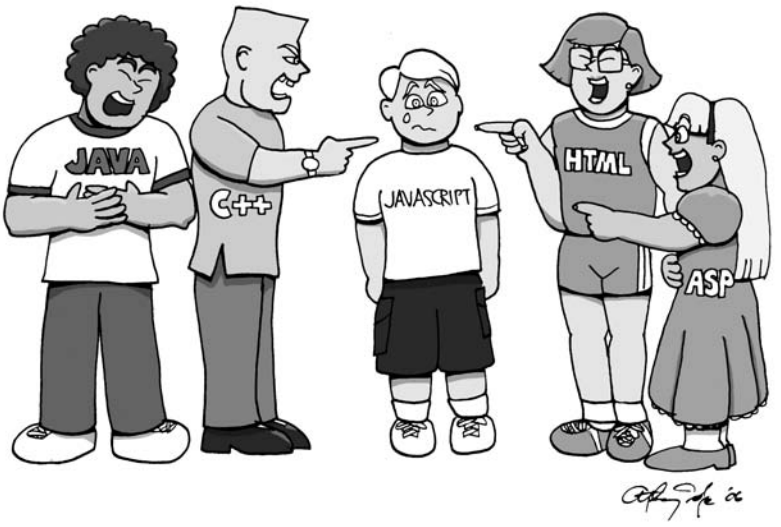
3. As a historical aside, you might be interested to know that version 2.0 of Netscape Navigator introduced not one but two noteworthy features. Aside from JavaScript, frames were also introduced. Of course, one of these has gained popularity, while the other tends to be shunned by the web developer community at large, but that's a story for another book!

deploy it—none of that complex "programming" stuff. And no complicated integrated development environment (IDE) was involved. It was really just as easy as saving a quick note to yourself.

Another important reason for JavaScript's early success was its seeming simplicity. You didn't have to worry about data types, because it was (and still is) a loosely typed language. It wasn't object-oriented, so you didn't have to think about class hierarchies and the like. In fact, you didn't even have to deal with functions if you didn't want to (and wanted your script to execute immediately upon page loading). There was no multithreading to worry about or generic collections classes to learn. In fact, the intrinsic JavaScript objects were very limited, and thus quickly picked up by anyone with even just an inkling of programming ability. It was precisely this seeming simplicity that lead to a great many of the early problems.

Unfortunately, JavaScript's infancy wasn't all roses by any stretch. A number of highly publicized security flaws hurt its early reputation considerably. A flood of books aimed squarely at nonprogrammers had the effect of getting a lot of people involved in writing code who probably shouldn't have been doing so (at least, not as publicly as a web site tends to be).

Probably the biggest problem, however, was the frankly elitist attitude of many "real" programmers. They saw JavaScript's lack of development tools (IDEs, debuggers, and so on), its inability to be developed outside a browser (in some sort of test environment), and apparent simplicity as indications that it was a "script kiddie" language—something that would be used only by amateurs, beginners, and/or hacks. For a long time, JavaScript was very much the "ugly duckling" of the programming world. It was the Christina Crawford,[4] forever being berated by her metaphorical mother, the "real" programmers of the world.



Poor javascript—other languages can be so cruel!

4. Christina Crawford was the daughter of Jane Crawford, and her story is told in the classic movie *Mommy Dearest* (`http://www.imdb.com/title/tt0082766`). Even if you don't remember the movie, you almost certainly remember the phrase "No more wire hangers!" uttered by Jane to Christina in what was probably the most memorable scene in the movie.

This attitude blinded programmers to the amazing potential that lay just below the surface, and that would become apparent as both JavaScript and the skill of those using it matured. This attitude also kept away a lot of excellent developers, who could have been helping accelerate that maturation process instead of stunting it. But JavaScript was destined for greatness, no matter what anyone else said!

The Evolution of JavaScript: Teething Pains

While it's true that JavaScript wasn't given a fair shake early on by programmers, some of their criticisms were, without question, true. JavaScript was far from perfect in its first few iterations—a fact I doubt that Netscape or Brendan Eich would dispute! As you'll see, some of it was a simple consequence of being a new technology that needed a few revisions to get right (the same problem Microsoft is so often accused of having), and some of it was, well, something else.

So, what were the issues that plagued early JavaScript? Several of them tend to stand out above the rest: browser incompatibilities, memory, and performance. Also, there was the true reason JavaScript wasn't embraced by everyone from the get-go: developers themselves! Let's explore these areas in some detail, because in order to understand where we are now, it helps to understand where we were not so very long ago.

But It's the Same Code: Browser Incompatibilities

To better understand the discussion to follow, and in the interest of those who prefer the graphical representation of information to the textual, let's look at two timelines. Figure 1-1 shows the somewhat simplified release history of Netscape's Navigator browser, and in lockstep, versions of JavaScript. Figure 1-2 shows the same basic information for Microsoft's Internet Explorer (IE) and its JScript implementation of JavaScript. While these data points are accurate, I have probably left out a point release here and there. And I haven't carried these timelines to the current day, because from the point where they end, we've been in the realm of ECMAScript and largely compatible implementations across browsers.

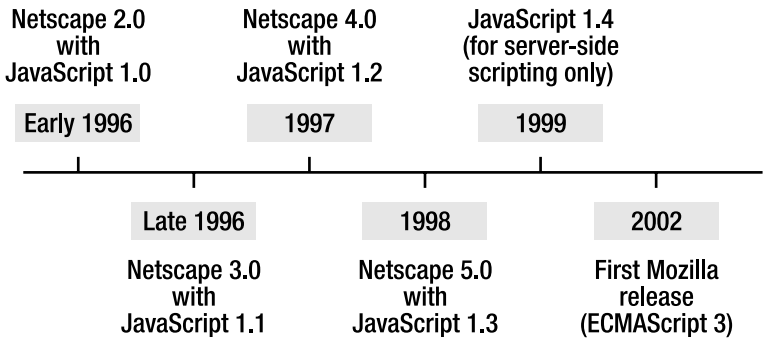


Figure 1-1. *The quick-and-dirty history of Netscape Navigator and JavaScript*

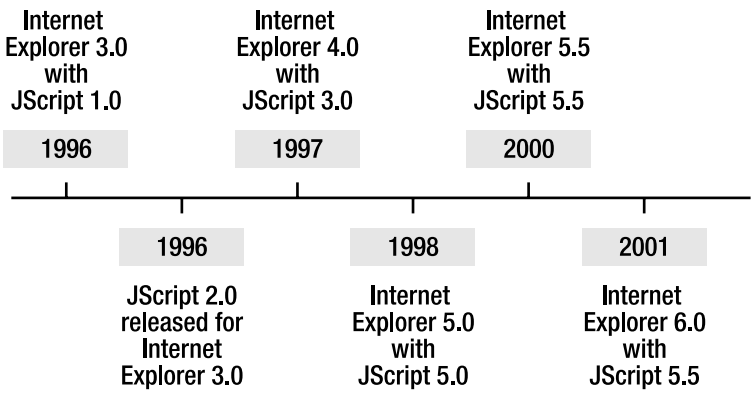


Figure 1-2. *The quick-and-dirty history of Internet Explorer and JScript*

When JavaScript came out, Microsoft developers realized they had a problem on their hands. Despite whatever issues may have existed with JavaScript early on, it was clear that this was something web developers were going to want. How could it be otherwise? For the first time, static pages could come alive.[5]

Microsoft found an answer for this situation. In fact, it had two! First, it created VBScript, which was at least syntactically modeled after its Visual Basic product. Second, and most important for the discussion in this section, Microsoft also created JScript, which was a (mostly) compatible version of JavaScript. It's that "mostly" part that caused problems.

One of the biggest perceived problems with JavaScript for a long time—really, until just two or three years ago—was incompatibilities among different browser versions. Most of this problem was caused by Microsoft's implementation coming into the picture. Logically, had Netscape remained the dominant browser, there likely would not have been any compatibility issues to speak of! On the gripping hand,[6] when Microsoft released JScript 1.0, it was actually quite compatible with JavaScript 1.0—close enough that cross-browser development could begin. It wasn't until Netscape released JavaScript 1.1 that compatibility issues really began. So, if you're a Microsoft booster, you can feel free to bash Netscape. If you're a Micro$oft hater, then it was clearly at fault!

From the point when Netscape released JavaScript 1.1 with Navigator 3.0 on, Microsoft's JScript implementation was at least one point release behind Netscape's at any given time, and

5. Well, not really the first time, but the first time without cumbersome, not to mention often buggy, plug-ins that required extra download time. Remember that this was years before broadband came into play, back in the days when a 56kbps modem that never quite performed up to spec was the predominant technology for connecting to the Internet.
6. "On the gripping hand" is a phrase used in the science-fiction book *The Mote in God's Eye*, written by Larry Niven and Jerry Ournelle, and also in *The Gripping Hand*, the sequel. It is used to describe the third choice sometimes available to us. For example, when you say, "We could do A . . . ; on the other hand, we could do B," you can also say ". . . on the gripping hand, we could do C." The phrase stems from the fact that the alien race the book deals with, the Moties, are asymmetrical in terms of their appendage layout; they have two arms on one side! It also happened to usually be the strongest of the three arms possessed by these creatures. These are excellent books, and if you are into science fiction and haven't read them yet, I highly recommend picking them up! They are considered classic works by most (so how you could call yourself a sci-fi fan without having read them?).

this condition persisted for quite some time. So, as one example, while image rollovers were becoming commonplace in Netscape browsers, this ability was not yet present in IE (around the IE 3.0 timeframe). To handle the differences, using "browser-sniffing" code to enable or disable bits of functionality became commonplace. This code would look something like that shown in Listing 1-1.

Listing 1-1. *An Old Browser-Sniffer Routine*

```
function Redirect()   {
  var WhatBrowser;
  var WhatVersion;
  WhatBrowser = navigator.appName.toUpperCase();
  WhatVersion = navigator.appVersion.toUpperCase();
  if (WhatBrowser.indexOf("MICROSOFT") >= 0) {
    if (WhatVersion.indexOf("3") >= 0) {
      top.location = "MainPage.html";
    } else {
      top.location = "BadVersion.html";
    }
  }
  if (WhatBrowser.indexOf("NETSCAPE") >= 0) {
    if (WhatVersion.indexOf("2") >= 0) {
      top.location = "MainPage.html";
    } else {
      top.location = "BadVersion.html";
    }
  }
}
```

In this code, if the browser version detected is not 3.*x* or higher for IE, or 2.*x* for Netscape, users are directed to `BadVersion.html`, which presumably tells them their browser is not compatible. They wind up at `MainPage.html` if the version meets these minimum requirements. This is obviously very flawed code for a number of reasons, which I'll leave as an exercise for you to find.

The important point here is that this "sniffing" of browser versions (and type, in some cases) was commonplace for a long time. In fact, you would often find two different versions of the same page: one designed for IE and the other for Netscape. This was clearly not an optimal situation! But for a long time, it was really the only way, because a piece of code would simply not work as expected in one browser vs. another. Often, it was more a matter of one browser supporting some feature that the other did not—sometimes because of proprietary extensions, and sometimes because one browser implemented an earlier version of JavaScript. Other times, it was outright differences in the way things worked.

It wasn't just enough to test for browser type and version though, because Microsoft had designed things such that the browser and the JScript language were separate entities. They could upgrade one without touching the other, because JScript was just a dynamic link library (DLL, a library of code linked to by another program at runtime). When IE 3.0 shipped, it did so with the first version of the JScript DLL. A short while later, when IE 3.0 was still the most current shipping version of the browser, Microsoft updated JScript to version 2.0. Microsoft did provide two functions, `ScriptEngineMajorVersion()` and `ScriptEngineMinorVersion()`, but aside from

those functions not being supported by anything other than IE, they also were not available in JScript 1.0! So dealing with them was often more trouble than they were worth. Still, they tended to be the best answer, because you sometimes needed the information to branch your code accordingly.

As an example of some of the sorts of incompatibilities you had to deal with back in the day, the `split()` method of the `String` class allowed for an optional `limitInteger` parameter, which would restrict the number of items converted into an array element. However, this parameter was recognized only by Navigator 4. As another example, Netscape did not support the `typeof` operator until Navigator 3, while Microsoft introduced it with JScript 1.0 (this is one of those proprietary extensions that proved so useful it was added to the ECMAScript 1.0 specification). For one more example, check out this simple snippet:

```
var d = new Date();
alert(d);
```

Something this simple would have been a problem early on because the `toString()` method of the `Date` object, which was intrinsically present in Netscape's implementation of the `Date` object, was not present in JScript until version 2.0!

Various problems like these would arise, and seemingly always at the most inopportune time! A tight deadline and a `substring()` function that doesn't treat negative values quite the same in IE as it does in Navigator are a sure recipe for disaster![7] That's why browser sniffing was so common for so long, even though we all knew it wasn't a good idea.

If that had been the only real problem with JavaScript though, I suspect developers would have griped and muttered under their breaths, but would have worked around it and gotten used to it. Unfortunately, it wasn't the only strike against JavaScript.

Of Snails and Elephants: JavaScript Performance and Memory Issues

JavaScript can be slow. There, I said it! Even today, you can easily write code that performs quite poorly. One trivial example is shown in Listing 1-2.

Listing 1-2. *An Example of Poor JavaScript Performance (and How to Fix It)*

```
<html>
  <head>
    <title>Listing 1-2</title>
    <script>

      function badTest() {
        var startTime = new Date().valueOf();
        var s = "";
        for (var i = 0; i < 10000; i++) {
          s += "This is a test string";
        }
```

7. I remember something like this being an issue, but I frankly couldn't pull anything out of Google to substantiate it. So, I offer it purely anecdotally, with the hope that my memory isn't failing *quite* this early in life!

```
        return new Date().valueOf() - startTime;
      }

      function goodTest() {
        var startTime = new Date().valueOf();
        var stringBuffer = new Array();
        for (var i = 0; i < 10000; i++) {
          stringBuffer.push("This is a test string");
        }
        var s = stringBuffer.join("");
        return new Date().valueOf() - startTime;
      }

      function betterTest() {
        var startTime = new Date().valueOf();
        var stringBuffer = new Array();
        for (var i = 0; i < 10000; i++) {
          stringBuffer[stringBuffer.length] = "This is a test string";
        }
        var s = stringBuffer.join("");
        return new Date().valueOf() - startTime;
      }

      function doTests() {
        var htm = "";
        htm += "Time badTest took: " + badTest() + "<br>";
        htm += "Time goodTest took: " + goodTest() + "<br>";
        htm += "Time betterTest took: " + betterTest();
        document.getElementById("result").innerHTML = htm;
      }

    </script>

  </head>

  <body>
    <a href="javascript:void(0);" onClick="doTests();">Click here to test</a>
    <br><br>
    <div id="result"> </div>
  </body>

</html>
```

As the caption for Listing 1-2 says, this example also gives you a free bonus: an optimization that you can definitely use in the real world! This example does the same (admittedly contrived) thing in three different ways:

- It constructs a string that consists of the string "This is a test string" 10,000 times ("This is a test stringThis is a test stringThis is a test string" and so on 10,000 times). It does a simple string concatenation using the + operator.
- It creates an array and uses the `push()` method to add "This is a string" to the array 10,000 times, and then finally uses the `join()` method of the `Array` class with a blank character, which returns a string formed by combining all the elements of the array together, separated by essentially nothing.
- It does this same array trick, but instead of using `push()`, it sets each element of the array explicitly, making use of the fact that if you try to set an element of an array whose index equals the length of the array, the array will grow by one.

Figure 1-3 shows how long each approach took in Firefox. You can see that none of them took an especially long time. The Mozilla developers have done an excellent job of optimizing their JavaScript engine, and this is especially evident in the simple + concatenation test case taking the least amount of time. This wasn't the case just a short while ago!

Figure 1-3. *The speed test results in Firefox (1.5.0.6, latest as of this writing)*

Now look at the same speed test results in IE, shown in Figure 1-4. The array tests are actually a little faster than in Firefox, although certainly not drastically so. But obviously string concatenation is a big no-no in IE. It's a whooping 95 times slower than Firefox!

Figure 1-4. *The speed test results in Internet Explorer (6.0.2900.2180, latest as of this writing)*

Lest anyone think something fishy is going on, these speed tests were run on the same PC, without virtual machines or anything like that. So the difference is attributable to the browsers almost entirely. It's possible that differences at runtime in the operating system itself could have had an impact. But I actually went so far as to reboot before running each test and didn't load anything else, so it was roughly as close to identical at runtime as could reasonably be expected.

■ **Note** I ran the same speed test on Maxthon, version 1.5.6 build 4.2, latest as of this writing. Maxthon tends to be my preferred browser for day-to-day browsing. It is a wrapper around IE that extends it with all sorts of features and fixes, putting it, in my opinion, on par with Firefox and most other browsers, while still using the IE rendering engine (some will say this is a bad thing, but most sites tend to work correctly in IE even if they don't in Firefox). The results were very surprising: 19141 for the bad test, 141 for the good test, and 93 for the better test. I have no explanation why it should be that much slower, especially the string concatenation approach. I don't mean this as a criticism of Maxthon, but it does illustrate the point that performance across different browsers, even where it seems that logically there should be no appreciable difference, is still something to be aware of when doing your work.

None of this is meant to persuade you that one browser is better than any other. In fact, a great many web developers will tell you that Firefox is superior, yet here we can see that in two out of three approaches to the same thing, it's a little slower than IE. The point is to illustrate the following:

- The same piece of JavaScript executed in one browser won't necessarily perform the same as in another browser, and sometimes the difference can be drastic.
- Performance of modern JavaScript engines still, in some cases, leaves a lot to be desired.

That's the situation today. It used to be much worse. As an example, Figure 1-5 shows the results of the same example in IE 4.0, which shipped with Windows 98.

Figure 1-5. *The speed test results in Internet Explorer 4.0*

Wow, the IE development team has clearly been busy! The simple bad test, using the + operator, is something on the order of 13 times faster now than it was with IE 4.0! The better test is about twice as fast. Note that the good test could not be run because the `push()` method was not available on the `Array` object in this iteration of JScript. I think we can reasonably surmise that it also would have been significantly slower back then.

The same tests on Netscape 3.01 yield even worse results. In fact, the bad test was taking so long, and was eating up so many system resources, that I had to kill the process! Suffice it to say the test more than validated my point about performance having improved markedly over the years.

Netscape 3.0 also demonstrates the other common failing of early JavaScript implementations: they were not efficient with memory. This inefficiency can largely be attributed to the simple evolution that occurs for virtually all software over time. You write something, you see what the flaws are, and you correct them for the next version. A JavaScript engine is no different.

Even just a few years ago, it was not uncommon to find that relatively simple pieces of code could cause the browser to use much more memory than it really needed. Memory leaks were not uncommon. Although they tended to be caused by developers doing things incorrectly, there were times when the engine and browser themselves caused such leaks. Remember, too, that JavaScript, like Java, is a memory-managed language with a garbage collector task running in the background. If the JavaScript interpreter may have had flaws, is it so crazy to imagine that the garbage collector implementation might have had its own set of flaws?

The speed and memory factors lent to the impression that JavaScript was slow and bloated. It was just in its early stages of development, and like all (relatively) complex pieces of software, it wasn't perfect out of the gate. That isn't to say that some problems don't exist to this day, because they do (just look at that first example). But the problems are far less frequent. In fact, I would dare say they are rare, except when caused by something the developer does. The problems also tend to not be as drastic as they once might have been. For example, unless you do something truly stupid, you won't usually kill the browser, as my test on Netscape 3.01 did.

And speaking of developers and doing something stupid . . .

The Root of All Evil: *Developers*!

As I talked about in the previous section, there were legitimate problems with early JavaScript implementations. It is also true that while you may find some problems today, they are few and far between. The one constant has been developers. Simply put, JavaScript is a tremendously powerful language, yet it is also easy to mess up. It is easy to write slow, bloated, error-prone code without trying very hard.

Like the language itself, developers had to evolve. They needed to learn what worked and what didn't, and they had to fight their own urges to take the easy way out. JavaScript is very flexible and dynamic, and this leads many developers to do things that in a more rigid language they would know not to do. For instance, consider the example in the previous section. If you were working in Java, you would almost certainly know that doing string concatenations is a Bad Thing™ and that the string buffer is your friend! But there is no string buffer in JavaScript, so many developers simply assume that string concatenation must be the way to go. In Firefox, that likely won't kill you, as the example showed, but in IE, you're just asking for trouble!

Another example is passing parameters to a function. Look at the code in Listing 1-3.

Listing 1-3. *An Example of Inefficient Coding*

```
<html>
  <head>
    <title>Listing 1-3</title>
    <script>

      function Person1(firstName, lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.toString = function() {
          return this.firstName + " " + this.lastName;
        }
      }

      function Person2(attrs) {
        this.firstName = attrs["firstName"]
        this.lastName = attrs["lastName"];
        this.toString = function() {
          return this.firstName + " " + this.lastName;
        }
      }

      function showPerson() {
        var p1 = new Person1("Frank", "Zammetti");
        var p2 = new Person2({"firstName":"Frank","lastName":"Zammetti"});
        document.getElementById("divPerson").innerHTML = p1 + "<br><br>" + p2;
      }
```

```
    </script>
  </head>
  <body onLoad="showPerson();">
    <div id="divPerson"> </div>
  </body>
</html>
```

Here, we have two different classes representing a person: `Person1` and `Person2`. `Person1`'s constructor accepts two parameters, `firstName` and `lastName`. `Person2` accepts a single parameter, `attrs`, which is an array of attributes. The `showPerson()` function creates two identical people, one using `Person1` and the other using `Person2`. What happens when we want to have other attributes to help describe a person? For `Person1`, we need to modify the constructor to accept more parameters. For `Person2`, it's just a matter of adding the appropriate field set lines. The call to the constructor has to change for both, so that's a wash. But what does the `Person1` call tell us?

```
var p1 = new Person1("Frank", "Zammetti");
```

You cannot deduce the meaning of the parameters just by looking at this call. How do we know that Zammetti isn't actually my first name? Or that Frank isn't the name of my father (which it just happens to be)? Clearly, the call syntax for `Person2` is better in terms of code clarity. The code is also a bit more easily extensible with that approach.

This is a relatively minor point, but it is an element of style that has only in the past few years come into the minds of JavaScript developers. Early on, you would rarely have seen the approach used in `Person2`. You would have instead seen function calls with oodles of arguments. But if you asked C++ developers how they would have coded this, you almost certainly would hear an answer involving some sort of collection, maybe a value object being passed in, or something along those lines.

Another problem that was prevalent for a long time was variable scoping. *Everything* was in the global scope, which is counter to most every other language out there, where variables are generally scoped only at the level they are required. Another thing that tripped up a lot of people for a long time, and sometimes still does, is the lack of block scope. Take a look at Listing 1-4.

Listing 1-4. *An Example of JavaScript's Lack of Block-Level Scoping*

```
<html>
  <head>
    <title>Listing 1-4</title>
    <script>
      function test() {
        var i = 1;
        if (1) {
          var i = 2;
          if (1) {
            var i = 3;
            alert(i);
          }
```

```
          alert(i);
        }
        alert(i);
      }
    </script>
  </head>
  <body onLoad="test();"></body>
</html>
```

In just about every other language on the planet, you would get the alerts 3, 2, 1, in that order. In JavaScript, however, you'll get the alerts 3, 3, 3. The variable i is allocated just once, the first time it is encountered, and overrides any declarations at a lower scope level.

One of the bigger changes is the drive toward more proper object-orientation. For many years, JavaScript developers—ones who seemed to know their stuff pretty well—didn't even realize that JavaScript was object-oriented! They tended to just write collections of functions, and that was that (for a long time, externalizing JavaScript wasn't even a common practice, which is another way in which developers have evolved). But if you look at most modern JavaScript libraries, such as Dojo and script.aculo.us, you will find a very clean, object-oriented design.

Another one of the early criticisms of JavaScript—something of a self-fulfilling prophecy—was that developers using JavaScript were somehow amateurs and didn't know their stuff. Unfortunately, as with most unpleasant generalizations, it started with a grain of truth. As previously discussed, the barrier to getting started with JavaScript is very low. You just need to throw together an HTML page, put some script in it, and point your browser at it. No compilation is required, and no development kit needs to be installed. Just Notepad and a reference web site somewhere would do the trick. Because of this, everyone and their mothers (literally, in some cases) started coding scripts. All of a sudden, you had forms being validated client-side, which was cool, but then the validations were not performed server-side, because the JavaScript coder didn't have the experience to know that's a Good Thing™ to do. You had image rollovers that didn't preload the images, so that each mouse event resulted in spurious network traffic, not to mention seemingly unresponsive user interfaces. You had the bane of all web surfers: pop-up ads!

All of these (except maybe pop-up ads, which are just the result of some evil marketing suits muscling their way into the technological side of the Web) are really just things that inexperienced developers do because they don't yet know any better. None were the fault of JavaScript per se, because it's likely that something else would have come along in its place anyway and caused all the same problems. Still, like our hairy ancestors before us, we had some evolving to do!

DHTML—The Devil's Buzzword

One more element to the "evil developers" story has to do with Dynamic HTML (DHTML). Although the label DHTML still correctly applies to effects used today, a certain connotation that goes along with that term makes people not want to use it any longer. The connotation is that while there was plenty of sizzle early on, there was very little steak.

Early JavaScript developers discovered that they could do all sorts of whiz-bang tricks—from fading the background color of a page when it loaded to having a colorful trail follow the cursor around the page. You could see various types of scrolling text all over the place, as well as different page-transition effects, such as wipes and the like. While some of these effects may look rather cool, they serve virtually no purpose other than as eye candy for the user. Now, don't get

me wrong here—eye candy is great! There's nothing I like more than checking out a new screen saver or a new utility that adds effects to my Windows shell. It's fun! But I always find myself removing those things later on, not only because they hurt system performance, but also because they pretty quickly become annoying and distracting.

Early JavaScript developers were huge purveyors of such muck, and it got old pretty fast. I don't think it is going too far to say that some people began to question whether the Web was worth it or not, based entirely on the perception that it was a playground and not something for serious business. A web site that annoys visitors with visual spam is not one they will likely use again. And if you're trying to make a living with that site and your company's revenues depend on it, that's going to lead to bad news real fast!

This obviously was not a failing of the technology. Just because we have nuclear weapons doesn't mean we should be flinging them all over the place! I suppose equating nuclear war to an annoying flashing thing on a web page is a bit of hyperbole, but the parallel is that just because a technology exists and allows you to do something doesn't necessarily mean you should go off and do it.[8]

Here's a quick test: if you are using Microsoft Windows, take a quick look at the Performance options for your PC (accessed by right-clicking My Computer, selecting Properties, clicking the Advanced tab, and clicking the Settings button under the Performance group). Did you turn off the expanding and collapsing of windows when minimized and maximized? Did you turn off shadows under the cursor? Did you disable the growing and shrinking of taskbar buttons when applications close? Many of us make it a habit to turn this stuff off, not only because it makes our system snappier (or at least gives that perception), but also because some of it just gets in the way. Seeing my windows fly down to the taskbar when I minimize them is pretty pointless. Now, you may argue that it depends on the implementation, because the effects on a Macintosh are better and not as annoying, and to a certain extent I would agree. But you still have to ask yourself whether the effect is helping you get work done. Is it making you more productive? I dare say the answer is no for virtually anyone. So while there may be degrees of annoyance and obtrusiveness, certain things are still generally annoying, obtrusive, and pointless. Unfortunately, this is what DHTML means to many people, and while I wish it weren't so, it isn't at all an undeserved connotation to carry.

So, part of the evolution of the JavaScript developer was in starting to recognize when the super-cool, neat-o, whiz-bang eye candy should be put aside. Developers began to realize that what they were doing was actually counterproductive, since it was distracting and annoying in many cases. Instead, a wave of responsibility has been spreading over the past few years. Some will say this is the single most important part of JavaScript's overall evolution towards acceptance.

You can still find just as many nifty-keen effects out there today as in the past—perhaps even more so. But they tend to truly enhance the experience for the user. For example, with the yellow fade effect (originated by 37signals, `http://www.37signals.com`), changes on a page are highlighted briefly upon page reload and then quickly fade to their usual state. Spotting changes after a page reload is often difficult, and so this technique helps focus the users on those changes. It enhances their ability to work effectively. This is the type of responsible eye candy that is in vogue today, and to virtually everyone, it is better than what came before.

---

8. I remember a television commercial where a bunch of web developers were showing their newly created site to their boss. The boss says there needs to be more flash, like a flaming logo. The developers look at him a little funny, and proceed to put a flaming logo on the page. It was pretty obvious to anyone watching the commercial that the flaming logo served no useful purpose, and in fact, had the opposite effect as was intended in that it made the site look amateurish. It's so easy to abuse eye candy it's not even funny!

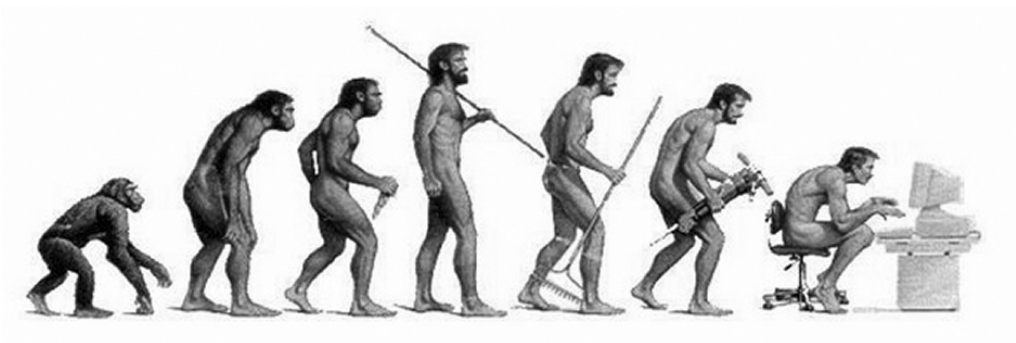
■ **Tip** To see an example of the positive usage of the yellow fade effect, take a peek at the contact form for ClearLeft at `http://clearleft.com/contact/`. Just click the submit button without entering anything and see what happens. You can also see the effect all over the place in the 37signals BaseCamp product at `http://www.basecamphq.com/` (you'll need to sign up for a free account to play around). You can get a good sense of where and why this seemingly minor (and relatively simple technically) technique has gained a great deal of attention. Other 37signals products make use of this technique, too, so by all means explore—it's always good to learn from those near the top! And if you would like to go straight to the source, check Matthew Linderman's blog entry at `http://www.37signals.com/svn/archives/000558.php`.

So, when you hear the term DHTML, don't automatically recoil in fear, as some do, because it still accurately describes what we're doing today from a purely technical definition. However, you should, at the same time, recognize that the term does have a well-earned negative connotation, brought on by the evils of early JavaScript developers.[9]

The Evolution Continues: Approaching Usability

After the initial wave of relatively inexperienced developers using JavaScript, and many times doing so poorly, the next iteration began to emerge. Certain common mistakes were recognized and began to be rectified.

Perhaps most important of all, the more experienced programmers who had initially shunned JavaScript began to see its power and brought their talents to bear on it. Those with true computer science backgrounds began to take a look and point out the mistakes and the ways to fix them. With that input came something akin to the Renaissance. Ideas began to flow, and improvements started to be made. It wasn't the final destination, but an important port of call along the way.



Javascript developers: out of the trees and onto the Web!

9. I'm not only the hair club president, but I'm also a client. I have some old web sites in my archives (thankfully, none are still live) with some really horrendous things on them! I certainly was not immune to the DHTML whiz-bang disease. I had my share of flaming logos, believe me. I like to think I've learned from my mistakes (and so would my boss).

Building a Better Widget: Code Structure

It may not sound like much, but simply structuring code in a clean, efficient way makes that code easier to follow, comprehend, and maintain months or years down the road. How many times have you run into something like the following code?

```
1:  function f(
2:  p1, p2)
3:  {
4:  p2 =
5:    p2.toUpperCase();
6:  s = ""
7:  for (i = 0; i < 10; i++) { s = s + p1;
8:      s += p2 + '-' + i
9:  }
10: if (p1 == "y") s += '<br>' + s
11:     if (p2 == 'n')
12: {
13:     s = s + "<br><br>"; }
14: }
```

Do yourself a favor and don't try to figure out what it's supposed to do. It's nonsense (I just threw some gibberish together). But it is syntactically correct and does execute, even if it does nothing intelligible. The point of the example is the structure of the code. It stinks, doesn't it? Let's try to spot the problems with it, in no particular order:

- Indentation is either nonexistent on some lines (line 2) or inconsistent between lines (two spaces on line 5 and four spaces on line 8).
- The argument names are not descriptive.
- Quotes are used inconsistently (single quotes vs. double quotes).
- Some lines end with semicolons; some do not.
- Some code blocks are surrounded by braces (the `for` loop in lines 7 through 9); some are not (the `if` on line 10).
- No checking is done before the call `toUpperCase()` on `p2`. If only one parameter were passed in, or the second parameter were passed as null, this would throw an error.
- Sometimes the code uses the Sun standard of an opening brace at the end of the line starting the block (line 7); other times it's on its own line (line 3). Sometimes the closing brace is on its own line (line 14); sometimes it's at the end of the block (line 13).
- Sometimes the += operator is used; other times the expanded `s = s +` form is used.
- The function itself doesn't have a meaningful name.
- There's not a single comment throughout the entire function, or before it.
- Characters that could cause problems, namely the < and > characters, are not escaped.

You may argue that most of this stuff, save maybe the null check of the incoming parameters, is simply sloppy coding. The problem is that this type of sloppy programming was prevalent for a long time in the JavaScript world. As more seasoned developers got involved, this problem started to go away. Anyone who programs for a living probably maintains code for a living, too (their own or someone else's), and seemingly little things like those in the example just won't fly. That isn't to say that you won't still see garbage code like this from time to time, and not just in JavaScript either, but it tends to be a lot less frequent nowadays.

Even the use of functions, as seen in the previous bad code example, isn't required in JavaScript. Indeed, early on, you could often find whole pages that didn't use functions at all, or used them only sparingly. You would find `<script>` blocks strewn throughout the page, executed as they were encountered as the page was parsed. This is still valid, and sometimes the best way to accomplish some goals, but in a whole page like this, it's not generally a good idea! So, developers started learning that functions were a good way to organize their code. The use of the `onLoad` page event to call setup functions, which previously would have just been anonymous `<script>` blocks somewhere on the page, became commonplace.

Another relatively important change was the notion of externalizing JavaScript. This is one of the tenants of unobtrusive JavaScript, which will be discussed in the next chapter. Externalizing your script tends to make your pages easier to follow, because you can concentrate on the markup and then refer to the code as required. It also leads to reusability, something else that was severely lacking early on.[10] Externalizing script tends to make you think in terms of reusability a little more. Another benefit of externalizing scripts is that it can lead to some performance gains. The browser can then cache a `.js` file, and if you happen to reuse it on another page, that's one less request the browser needs to make. Another possibly less obvious advantage is that others can easily use your scripts and see how they work. If you've ever tried to dig a couple lines of JavaScript out of a 200kb web page to see how the developers did some neat trick, you'll know exactly what I'm talking about. It can be a pain to find what you're looking for amidst all the markup and other script (and probably style sheets, too, since if they didn't externalize their scripts, they probably didn't externalize their style sheets either). Modern browser tools make this a lot less difficult, but it can still be an unpleasant experience, and it was certainly less pleasant in the not-too-distant past.

Relearning Good Habits

While a lot of the early problems with JavaScript undoubtedly did come from less experienced programmers getting into the mix, certainly that didn't account for everything. Overnight, thousands of otherwise good, experienced programmers got stupid all at once!

As I mentioned earlier, working on JavaScript was almost too easy in a sense—throw some code in a file, fire up a browser, and off you go! In most other languages, you have a compile cycle, which tends to ferret out a lot of problems. Then you often have static code analysis tools, which find even more things to fix. You may even have a code formatter involved to enforce the appropriate coding standards. None of this is (typically) present when working

10. Reusability is often hard! It's frequently—maybe even usually—easier to write code specific to the task at hand. It takes effort to think generically enough that the code can be applied to other similar situations later, but specific enough to solve the problem at hand. Programmers are often lazy beasts (I know because I am one!) and like to take the easy road. Just as anger, fear, and aggression are the path to the dark side of the Force, laziness is the path to code that can't (easily) be reused. Of course, not knowing better also has something to do with it.

with JavaScript. I put *typically* in parentheses because modern development tools now exist to give you all of this (well, generally not the compile part).

Maybe "the bubble" had something to do with it, too. I'm referring to that period when everyone thought he had the sure-fire way to make a buck off the Web, and when the public was just starting to get online and figure out how cool a place the Web was. There were 80-hour work weeks, powered by Jolt cola, jelly donuts, and the incessant chant of some flower shirt-wearing, Segway-riding (OK, Segway wasn't out then, but work with me here!) recent college grad with an MBA, who promised us all those stock options would be worth more than we could count. Maybe that caused everyone to just slap the code together so it at least *appeared* to work, in a pointless attempt to implement the business plan, and is really what caused all the trouble.

Yeah, you're right, probably not. Ahem.

The good habits that developers had learned over time—like code formatting, commenting, and logical code structure—had to essentially be relearned in the context of JavaScript. And, of course, those who hadn't done much programming before had to learn it all anew. But learn they did, and from that point, JavaScript started to become something "professional" developers didn't thumb their noses at as a reflex act. Now it could start to become a first-class citizen, with the knowledge of how to do it right.

Of course, the last step was yet to come.

The Final Evolution: Professional JavaScript at Last!

We've arrived at the present time, meaning the past two to three years. JavaScript has really come into its own.

The whole Ajax movement has certainly been the biggest catalyst for getting JavaScript on a more solid footing, but even a bit before then, things were starting to come around. The desire to build fancier, more reactive, user-friendly, and ultimately fat-client-like web applications drove the need and desire to do more on the client. Performance considerations certainly played a role, too, but I suspect a lot smaller one than many people tend to think.

The bottom line is that JavaScript has moved pretty quickly into the realm of first-class citizen, the realm of "professional" development. Perhaps the best evidence of this is that you can now find terms like *JavaScript engineer*, *JavaScript lead*, and *senior JavaScript developer* used to describe job offerings on most job search sites. And people now say them with a straight face during an interview!

So, aside from Ajax, what are the reasons for this relatively current trend toward respectability that JavaScript seems to have earned? Let's have a look.

Javascript: finally getting the respect it deserves!

The Browsers Come Around

Over the past few years, the major browsers, and even the more minor ones, have come to a point of relative equilibrium where their JavaScript implementations are mostly compatible. You can still find discrepancies here and there, but they have become what they always should have been: the exceptions to the rules. Today, it's relatively rare that you need to write branching code for different browsers, and it's virtually unheard of to do browser-sniffing to redirect to a browser-specific version of a page.

If you write ECMAScript-compliant code these days, you'll find that it works correctly in the vast majority of client browsers. That isn't to say that you won't need to do some performance tuning for a browser. The example in Listing 1-3 is a good example. The code is compatible across browsers, but you still need to accommodate the performance of IE if you initially did string concatenations.

Indeed, the major problem you find today is not at all about the JavaScript implementation, because all the big players have been based on ECMAScript for a few versions now. The problem that still crops up in terms of compatibility is actually the DOM.

DOM is the in-memory representation of the current page. Each item on the page is a node in a tree—the tree formed by the relationship between the elements on the page. JavaScript has been standardized as ECMAScript for some time now, but the DOM was not standardized for a while after JavaScript was released. This led to the major browser vendors doing things in sometimes drastically different ways.

Just as one example, let's consider handling keypresses in IE vs. Firefox. Let's say we want to hook the `keyDown` event for the current document. We can do this like so:

```
document.onkeydown=keyDown;
```

That will work just fine in IE, but in Firefox, you also have to do this:

```
document.captureEvents(Event.KEYDOWN);
```

The basic `keyDown()` function signature for either browser is this:

```
function keyDown(e) { }
```

In Firefox, the parameter `e` will be an `event` object passed in that describes the keypress event. In IE, however, this parameter is not passed in at all because IE uses an event model called *event bubbling*. To get a reference to the `event` object in IE, you need to reference the `event` property of the `window` object. This isn't a difference in JavaScript itself; this is a difference in the event-handling model in the DOM of each browser.

To take the example further, once you have a reference to the `event` object, you will quite likely want to figure out which key was actually pressed. Again, there are DOM differences to overcome. In IE, the `event` object exposes a `keyCode` property. In Firefox, the corresponding property is called `charCode`. So, you will necessarily have some branching code to obtain the key code properly. You will usually wind up with code along these lines:

```
document.onkeydown = keyDown;
if (document.layers) {
  document.captureEvents(Event.KEYDOWN);
}
function keyDown(e) {
  var ev = (e) ? e : (window.event) ? window.event : null;
  if (ev) {
    return (ev.charCode) ? ev.charCode:
      ((ev.keyCode) ? ev.keyCode : ((ev.which) ? ev.which : null));
  }
  return -1;
}
```

This code should work in any browser. The first `if` check in the second line will be true only in non-IE browsers, where the `layers` attribute of the `document` object is present. In that case, the `captureEvents()` call is made. Then, inside `keyDown()` itself, the first line will set `ev` to the passed-in argument `e`, if `e` was passed in. If it wasn't, then `window.event` is used. But if `window.event` is itself not defined, then `ev` is set to null (this can happen in certain situations, so it must be checked). Then, if `ev` is set, the code of the key is discovered by determining if `charCode` or `keyCode` is present in the `event` object. If `ev` was null, then –1 is returned.

Code like this is becoming less and less necessary, as the browsers begin to converge on their implementations of not only JavaScript, but the DOM specification as well. You may still need to do it occasionally, but it's a far better situation than it used to be!

The other improvements that the browser vendors made were in the areas of performance and memory utilization. While still interpreted, JavaScript performs much better in modern browsers than it did in earlier ones. Optimizations have come fast and furious. Each successive JavaScript release has improved optimization by leaps and bounds. Probably the biggest reason is simply that usage patterns began to emerge over time. For instance, DOM manipulation is without question the most common use of JavaScript, so a lot of work has gone into making that as efficient as possible.

Likewise, the garbage collection algorithms have improved greatly, resulting in less memory utilization over a given period of time. The JavaScript engines themselves are better written these days, so they intrinsically take up less memory. The code has been tightened up, too. Memory leaks are nearly always caused by developer mistakes nowadays; the browser and JavaScript engine are virtually never the culprits anymore.

Finally, crashes are infrequent in any current implementation. It used to be that you might occasionally see some random crashes here and there—the browser just "disappearing" and things like that. This was sometimes caused by the JavaScript engine (usually because the developer did something that wasn't too smart, but still, the engine should have been able to cope). This was never a huge problem, but even it has improved, so the browsers get credit for it anyway!

Object-Oriented JavaScript

The life of most JavaScript developers changes the day they discover the prototype. Once they realize that every JavaScript object can be extended via its prototype, and that this also allows them to create custom classes, things are never the same again. For instance, take a look at this code:

```
var answer = 0;
function addNumbers(num1, num2) {
  answer = num1 + num2;
}
function subtractNumbers(num1, num2) {
  answer = num1 - num2;
}
function multiplyNumbers(num1, num2) {
  answer = num1 * num2;
}
function divideNumbers(num1, num2) {
  if (num2 != 0) {
    answer = num1 / num2;
  } else {
    answer = 0;
  }
}
```

Now, there isn't anything *technically* wrong with that code. It will work just fine. But is it organized especially well? Not really. The `answer` variable being in global scope is a code smell, and each of the functions is just that: a stand-alone function, which also happens to be in the global scope. One of the things I'll talk about in the next chapter that contributes to a more professional style of JavaScript is not "polluting" the global scope.

Using global variables in most other languages is considered a bad practice because their nonlocality means they can be modified from any part of the program, thereby creating the potential for mutual dependencies and difficult-to-locate problems (which are often transient and therefore even more insidious). The same is true in JavaScript. Functions in the global scope are a little less bothersome, although the lack of structure means that there is no inherent relationship among the functions and no logical groups to make sense of it all at a higher level.

In contrast, let's see the code rewritten with a more object-oriented twist:

```
function NumberFunctions() {
  var answer = 0;
}
NumberFunctions.prototype.addNumbers = function(num1, num2) {
  this.answer = num1 + num2;
}
NumberFunctions.prototype.subtractNumbers = function(num1, num2) {
  this.answer = num1 - num2;
}
NumberFunctions.prototype.multiplyNumbers = function(num1, num2) {
  this.answer = num1 * num2;
}
NumberFunctions.prototype.divideNumbers = function(num1, num2) {
  if (num2 != 0) {
    this.answer = num1 / num2;
  } else {
    this.answer = 0;
  }
}
NumberFunctions.prototype.toString = function() {
  return this.answer;
}
```

To use this code, we would do something like this:

```
var nf = new NumberFunctions();
nf.addNumbers(2, 1);
alert(nf);
nf.subtractNumbers(10, 3);
alert(nf);
nf.multiplyNumbers(4, 5);
alert(nf);
nf.divideNumbers(12, 6);
alert(nf);
```

This version of the code has a few advantages:

- There is no pollution of the global scope, save the fact that `NumberFunctions` is there. This is most important in terms of the `answer` variable. Because it is declared using the `var` keyword, it is not accessible from outside the class, so only functions of `NumberFunctions` can change it.
- All the functions are clearly related by virtue of being members of the `NumberFunctions` class.
- Basic object-orientation: the data and the function that operates on it are encapsulated nicely.

None of this is anything special or unusual in most other modern languages, but it took a while to find its way into JavaScript.

Object-orientation isn't the final word in JavaScript's evolution to the modern day, however. A few others concepts come into play.

"Responsible" JavaScript: Signs and Portents

Responsibility may seem like an odd term to use with regard to a programming language. We're not dealing with handguns or nuclear weapons after all! But it is indeed a very important concept that, until recently, was severely lacking in JavaScript circles.

You probably have heard the term *graceful degradation*. This is the idea that a web page designed for a certain version of a browser should degrade gracefully in older versions, and still be usable, if not optimal. The same guiding principle can, and should, be applied to JavaScript as well.

A somewhat more recent term is *unobtrusive JavaScript*. This is graceful degradation in a more refined form, but it also covers other areas such as making pages that use JavaScript still accessible for those with handicaps, creating future-proofed code, and keeping your script separate from the markup and style of the page.

Another factor that frequently comes into play is making JavaScript an enhancement to the browsing experience, but something you barely notice (this, too, is one of the meanings of *unobtrusiveness* as applied to JavaScript). Users expect a certain level of interaction and power in modern web user interfaces, and JavaScript definitely helps enable them. But any time users notice any of this, and most especially if it gets in their way, your code has probably intruded on their experience. There is a very fine line between a whiz-bang feature that your users will feel empowers them and an annoying feature that they dread.

Proper error handling is also a tenet of responsible JavaScript. Error handling in JavaScript used to amount to not much more than letting the browser display whatever error messages it needed to! After a while, people discovered that they could hook into the error-handling mechanism and present their own error messages, but it still amounted to little more than a message to the user saying, "Sorry, something went wrong. You're boinked." Modern JavaScript implementations provide better ways to handle errors, using mechanisms built in to the language to allow your code to continue and recover in the face of exceptions. Doing so makes your code more robust and pleasant for the user.

Lastly, although not strictly speaking functions of JavaScript itself, modern development tools far exceed those available in the past. All sorts of browser plug-ins and extensions now make developing JavaScript, if not a pleasant experience, at least a far less painful one. Even more powerful commercial tools offer whole environments dedicated to JavaScript. Most modern IDEs support JavaScript natively, as a first-class citizen.

If it seems like I've glossed over these points, it's because I have! The next chapter will go into these topics in much greater detail, I promise! This last section is just my way of whetting your appetite a bit, giving you a heads-up about what is to come. Stick around—it's going to be a fun ride!

Summary

This chapter covered the genesis of JavaScript—how it evolved in terms of usage from its not so spectacular early days to the current professional-quality JavaScript. We looked at some of the problems faced by early JavaScript developers and how they began to overcome them. You then got a glimpse of the ways in which JavaScript is now being used so it is much cleaner, less intrusive, and just generally better! In the next chapter, we'll look at what goes into working with JavaScript in a more mature way than used to be the case.

CHAPTER 2

The Seven Habits of Highly Successful JavaScript Developers

In this chapter, we'll continue the discussion began in Chapter 1 and look in more detail at the art of making JavaScript a first-class language. We'll look at object-oriented techniques, as well as some of the latest buzzwords such as *unobtrusive JavaScript* and *graceful degradation*. We'll talk about how to make your web applications accessible, even with JavaScript involved (no easy task!). We'll look at error-handling and debugging techniques, since things sometimes (OK, *frequently*!) don't go right. We'll also take a look at some of the tools available to you that will make working with JavaScript a much more pleasant experience. Lastly, we'll do a quick survey of some of the most popular JavaScript libraries out there today, and discuss why you really, honestly, and truly want to be using them! That's a lot to cover, so let's get to it!

JavaScript: reach divinity in the eyes of your users by doing it right!

More on Object-Oriented JavaScript

When many JavaScript programmers start out, they often do not even realize that the language offers some object orientation. Indeed, JavaScript does not require the use of objects at all.[1]

There is more than one way to skin a cat, and likewise, there is more than one way to create objects in JavaScript.

Simple Object Creation

Perhaps the easiest way to create an object is to start with a new `Object`, and then add to it. To create a new `Object`, you simply do this:

```
var newObject = new Object();
```

The variable `newObject` now points to an instance of `Object`, which is the base class of all objects in JavaScript. To add elements to it, say a property named `firstName`, all you need to do is this:

```
newObject.firstName = "frank";
```

From that point on in the code, `newObject.firstName` will have the value `"frank"`, unless it's changed later. You can add functions just as easily:

```
newObject.sayName = function() {
 alert(this.firstName);
}
```

A call to `newObject.sayName()` now results in an alert message showing "frank." Unlike most full-blown object-oriented languages, in JavaScript, you do not necessarily need to create a class, or blueprint, for an object instance. You can instead create it on the fly, as shown here. You can do this throughout the life of the object. On a web page, that means that you can add properties and methods to the object at any time.

JavaScript actually implements all objects as nothing but associative arrays. It then puts a façade over that array to make the syntax look more like Java, or C++, using dot notation. To emphasize this point, note that you could retrieve the value of the `firstName` field of `newObject` like so:

```
var theFirstName = newObject["firstName"];
```

Likewise, the `sayName()` function could be called like so:

```
newObject["sayName"]();
```

This simple fact can be the basis for a lot of power. For instance, what if you wanted to call a method of an object based on some bit of logic? Well, you can do this:

1. Well, implicitly it does, since you use built-in objects in many cases, but your code itself doesn't have to be object-oriented.

```
var whatFunction;
if (whatVolume ==1) {
  whatFunction = "sayName";
}
if (whatVolume == 2) {
  whatFunction = "sayLoudly";
}
newObject[whatFunction]();
```

Assume that we had the function `sayLoudly()` added to `newObject`, which called `toUpperCase()` on the `firstName` field before the `alert()`. Then we could have that object saying the name loudly (all caps) or softly (all lowercase, as shown), and do this based on the value of a variable.

When adding functions to an object, you can also use existing functions. Let's go ahead and add that `sayLoudly()` function now as an example:

```
function sayLoudly() {
  alert(this.firstName.toUpperCase());
}
newObject.sayLoudly = sayLoudly;
```

Note the use of the `this` keyword here. The object it refers to will be dynamically calculated, so to speak, at runtime. Therefore, in this case, it will point to the object the `sayLoudly()` function is a member of, `newObject` in this case. What's interesting to note is that when `sayLoudly()` is part of another object entirely, the keyword `this` will then reference that other object. This runtime binding is another very powerful feature of JavaScript's object-oriented implementation, since it allows for sharing of code, and, in essence, a form of inheritance.

Object Creation with JSON

Because JavaScript Object Notation (JSON) has recently been getting a great deal of attention with its use in Ajax requests, many people are aware of it. However, many people are still not aware that JSON is actually a core part of the JavaScript specification, and it was designed even before Ajax came onto the scene. Its original goal was for quickly and easily defining complex object graphs; that is, instances where objects are nested within others. Even in its simplest form though, it allows for another way to create objects.

Recall that objects in JavaScript are just associative arrays under the covers. This fact is what allows JSON to work. Let's see how to create the previous example's `newObject` with JSON:

```
function sayLoudly() {
  alert(this.firstName.toUpperCase());
}
var newObject = {
  firstName : "frank",
  sayName : function() { alert(this.firstName); },
  sayLoudly : sayLoudly
};
```

Using JSON is very similar to defining an array, except that you use curly braces instead of square brackets. Note that functions can be defined inline or can reference external functions. (It may be a little confusing to see `sayLoudly : sayLoudly`, but JavaScript understands that the first `sayLoudly` is to be a member of the object, while the second `sayLoudly` is a reference to an existing object.)

You can nest object definitions as much as you like in JSON to create a hierarchy of objects. For instance, let's add an object into `newObject` named `LastName`:

```
function sayLoudly() {
  alert(this.firstName.toUpperCase());
}
var newObject = {
  firstName : "frank",
  sayName : function() { alert(this.firstName); },
  sayLoudly : sayLoudly,
  LastName : {
    lastName : "Zammetti",
    sayName : function() { alert(this.lastName); }
  }
};
```

You can then display the last name by calling the following:

```
newObject.LastName.sayName();
```

Class Definition

In JavaScript, virtually everything is an object. This is true with only a few exceptions, such as some built-in primitives. Most important for this discussion, functions themselves are objects! You've seen how you can create instances of `Object` and add properties and methods to it, but that means that every time you want a new instance of that object, you essentially need to construct it from scratch. Certainly there must be a better way, right? Of course there is: create a class!

A class in JavaScript is actually nothing more than a function. This function also serves as the constructor of the class. So, for example, let's write that `newObject` as a class, renamed `newClass`:

```
function newClass() {
  alert("constructor");
  this.firstName = "frank";
  this.sayName = function() {
    alert(this.firstName);
  }
}
var nc = new newClass();
nc.sayName();
```

When this code is executed, you see two alerts in sequence: first, one saying "constructor" when the line `var nc = new newClass();` executes, and then one saying "frank" when the line

`nc.sayName();` executes. You can create as many instances of `newClass` as you want, and they will have the same properties and methods. Upon creation, they will generate the same alert, and `firstName` will have the same starting value. In short, you have created a blueprint for creating `newClass` objects. You have defined a class!

However, one problem that arises from this is that each instance of `newClass` has a copy of `firstName` and a copy of the `sayName()` method, so every instance adds more memory usage. Each copy of `newClass` having its own copy of `firstName` is probably what you want, but wouldn't it be great if all instances could share the same copy of `sayName()`, thereby saving memory? Clearly, in this instance, we're not talking about a big deal in terms of memory, but you can easily imagine a more substantial piece of code where it would make a much bigger difference. Fortunately, there is a way to do that.

Prototypes

Every single object in JavaScript has a `prototype` property associated with it. There is no real equivalent to `prototype` in any other language that I am aware of, but it can be seen as a simplistic form of inheritance. Basically, the way it works is that when you construct a new instance of an object, all the properties and methods defined in the prototype of the object are attached to the new instance at runtime.

I realize this can be a bit bizarre to comprehend at first blush, but fortunately, it is simple enough to demonstrate:

```
function newClass() {
  this.firstName = "frank";
}
newClass.prototype.sayName = function() {
  alert(this.firstName);
}
var nc = new newClass();
nc.sayName();
```

When executed, this code results in the familiar alert saying "frank." What makes this different from the previous example is that no matter how many instances of `newClass` you create, only a single instance of the `sayName()` function will be in memory. This method will essentially be attached to each of those instances, and the `this` keyword will again be calculated at runtime, so that it always refers to the specific instance of `newClass` to which it belongs. For example, if you have two instances of `newClass` named `nc1` and `nc2`, then a call to `nc1.sayName()` results in `this` pointing to `nc1`, and a call to `nc2.sayName()` results in `this` pointing to `nc2`.

Which Approach Should You Use?

Each of the preceding approaches has its own pluses and minuses, and I doubt there is any real consensus anywhere about when one approach should be used over another. They are all *functionally* equivalent, so it's largely a matter of how you prefer your code to look. That being said, I think there are a few general guidelines to making your decision.

Probably the biggest one is that if you are creating a class that is rather large and you know there may be multiple instances of it, you almost certainly want to use the prototype approach. This will lead to the best memory efficiency, which is always an important goal.

If you are creating a singleton class—something that you know there will be only one instance of—I personally would opt for defining a class. To me, the code is the most logical and the most similar to the more fully object-oriented languages, and so will probably tend to be easier to comprehend for new developers on a project.

The JSON approach is probably a good choice if (a) your object hierarchy is going to be highly nested and/or (b) you need to define the object in a dynamic fashion (as the result of logic code). JSON is also pretty clearly the best choice if you need to serialize an object and transmit it over the wire. This is also true if you need to reconstitute an object sent from a server. I doubt there are many easier ways than JSON for this, and that is in no small part because that's largely what it was designed for!

Benefits of Object-Orientation

Whatever approach you choose, object-orienting your code has a lot of benefits. One important benefit is that each object is essentially a namespace. You can simulate Java and C# packaging this way, as you will see in the next chapter.

Another benefit is that you can hide data using objects. Consider the following:

```
function newClass() {
  this.firstName = "Frank";
  lastName = "Zammetti";
}
var nc = new newClass();
alert(nc.firstName);
alert(nc.lastName);
```

Executing this code results in two alerts: the first saying "Frank" and the second saying "undefined." That is because the `lastName` field is *not* accessible outside an instance of `newClass`. Note the difference in how the fields are defined. Any fields defined with the `this` keyword, as `firstName` is, will be accessible outside the class. Any defined without `this` will be accessible only inside the class. This goes for methods as well.

Also, don't forget that the built-in JavaScript objects can be extended using their prototype. In fact, the JavaScript library named Prototype does exactly this, as you will see in the "JavaScript Libraries" section later in this chapter. However, you can really mess up things if you're not careful, so extend built-in objects with caution!

You can also "borrow" functions from other objects and add them to your own. For instance, let's say you want to be able to display the `firstName` field of `newClass` simply by outputting `newClass` itself. To do this, you implement the `toString()` function. Let's further say you want to always use the `toUpperCase()` function from the `String` object on it. You can do all that easy enough:

```
function newClass() {
  this.firstName = "frank";
  this.toUC = String.toUpperCase;
  this.toString = function() {
    return this.toUC(this.firstName);
  }
}
var nc = new newClass();
alert(nc);
```

Executing this code results in an alert saying "FRANK." Note that `toString()` was called, but *not* as a method of the `firstName String` object. Instead, it was called via the reference to it included as part of `newClass` under the property named `toUC()`. This is a handy capability, especially when you create your own objects and later decide to create new ones that leverage code you have already written. You don't need to copy, cut, and paste. You just reference the existing methods of other classes, and you're all set.

Graceful Degradation and Unobtrusive JavaScript

Unobtrusive JavaScript is a term that has come onto the scene relatively recently. It is, in simplest terms, a trend where JavaScript on web pages is done in such a way that it doesn't, well, intrude on the page.

The basic tenets of unobtrusive JavaScript are pretty simple and can be easily summarized:

- Keep JavaScript separate.
- Generally, allow graceful degradation.
- Never use browser-sniffing scripts to determine the capabilities of a browser.
- Never, under any circumstances, create JavaScript that is not cross-browser, or more specifically, code that is dialect-specific.
- Properly scope variables.
- For accessibility, avoid triggering required events as the result of mouse events.

However, the term *unobtrusive JavaScript* can have different meanings to different people. Some like to extend the rules a bit further and make it more rigid. Others try to trim the rules back a bit and make it more flexible. The key point is to implement JavaScript in such a way that we learn from some of our past mistakes.

Let's now look at each of the basic tenets in a little more detail.

Keep JavaScript Separate

The idea is to treat JavaScript as a layer of your application, and try to make it as separate as possible, with well-defined interaction points. For instance, always import JavaScript from external files. None of this JavaScript embedded in HTML stuff!

This is one of those rules that can be a little flexible, in my opinion. For instance, a few configuration variables on the main page wouldn't upset me much, but others may tell you to externalize even those. But the basic idea is sound: keep scripts separate to the largest extent possible.

Think of Cascading Style Sheets (CSS). You're in the habit of externalizing style sheets, right? Look at JavaScript in the same way! This will logically break up the pieces that compose your page, making it easier to quickly home in on what you're actually interested in working on. It will also lead you down a path of reuse. Scripts that are externalized stand a much better chance of being used on other pages, other sites, and other projects. It doesn't guarantee it of course, but it tends to help.

Some people even advocate adding event handlers via scripts. The usual reasons given for doing this are to keep scripts out of markup entirely and to avoid having to modify code in many places if the function names change. I do not entirely agree with this directive, mainly because of the argument that an event handler is specific to a given element, so why shouldn't it be directly attached to the element? To me, if I need to change an event handler, it is easier to go directly to the element than to figure out which external `.js` file contains the code. An exception is if the handler will be shared (used by more than one element); in that case, I *would* externalize it.

I leave you to reach your own conclusion on what is best for you. However, I do encourage you to keep your event handlers as small as possible, regardless of where you put them. They should generally do little more than call some larger piece of code or execute one or two statements. This is an especially good idea if you do decide to have the handlers in-line with the elements.

Allow Graceful Degradation

A page should work, even if in a degraded form, without JavaScript. A good example is form validation. Don't have a plain button that calls a function that submits the form, as the form will be unsubmittable without JavaScript. For example, try the code in Listing 2-1 and see what happens if JavaScript is disabled.

Listing 2-1. *Form Submission That Doesn't Degrade*

```
<html>
  <head>
    <script>

      function doSubmit(inForm) {
        if (inForm.firstName.value == "") {
          alert("You must enter a first name");
          return false;
        }
        if (inForm.lastName.value == "") {
          alert("You must enter a last name");
          return false;
        }
        inForm.submit();
        return true;
      }

    </script>

  </head>
  <body>

    <form name="test" action="#" method="post">
      First name: <input type="test" name="firstName">
      <br>
      Last name: <input type="test" name="lastName">
```

```
      <br>
      <input type="button" onClick="doSubmit(this.form);" value="Submit">
    </form>

  </body>
</html>
```

If you run Listing 2-1 with JavaScript disabled, you'll see that nothing happens, because the form submission depends on the JavaScript executing. This is clearly bad.

Instead of this approach, validate in response to the `onSubmit` event. That way, if JavaScript is enabled, users get the benefit of the client-side validations. But if JavaScript is turned off, the form can still be submitted. Listing 2-2 shows how this works.

Listing 2-2. *Form Submission That Gracefully Degrades*

```
<html>
  <head>
    <script>

      function doSubmit(inForm) {
        if (inForm.firstName.value == "") {
          alert("You must enter a first name");
          return false;
        }
        if (inForm.lastName.value == "") {
          alert("You must enter a last name");
          return false;
        }
        inForm.submit();
        return true;
      }

    </script>

  </head>
  <body>

    <form name="test" action="#" method="post"
      onSubmit="return doSubmit(this);">
      First name: <input type="test" name="firstName">
      <br>
      Last name: <input type="test" name="lastName">
      <br>
      <input type="submit" value="Submit">
    </form>

  </body>
</html>
```

By the way, one cardinal sin is using purely client-side validation and assuming any data coming from the client is good. In fact, your systems should always be designed to assume the data coming from the client is *bad*. It's perfectly acceptable to do client-side validation. But it's virtually never acceptable for that to be the *only* validation your system performs.

Now for some opinion. Some people believe that graceful degradation, and some of the other tenets of unobtrusiveness and accessibility, should apply to *any* web application. I do not agree with that view, and moreover, I believe it is a view that is untenable.

Take a web-based game like the project in Chapter 11—can you imagine one being written without requiring JavaScript? That would be akin to saying Electronic Arts should write the next version of Madden Football using Logo, or that Bungie should write the next version of Halo in HTML, or that Microsoft should create a version of Windows based on any programming language that doesn't supply logic branching, looping, variables, or data structures.

The point is that a game requires executable code, as you'll see in Chapter 11 (go ahead, feed your curiosity and take a quick peek—I'll still be here when you return!). Do you think graceful degradation in such a project is a reasonable goal? Aside from degrading to a page that says something like, "Sorry, you can't play without JavaScript," I certainly can't. Should JavaScript be optional in such an application? I don't doubt that someone, somewhere, has written a web-based game that requires just straight HTML and/or degrades gracefully in the absence of JavaScript. But I also don't doubt that such a creation is an exceedingly rare exception.

We are in an era when Rich Internet Applications (RIAs) are beginning to rule the roost. Google, for instance, is now in the early stages of putting a full office suite on the Web for all to use.[2] Do you think the developers will attempt to make a version that adheres to all the tenets of unobtrusiveness, degrades gracefully (beyond a certain minimum level), and is fully accessible? Almost certainly not, because it is a nearly impossible task in such advanced applications. Let me be clear: *most* of what unobtrusiveness is all about as described here is still perfectly doable in an RIA world. It's just that some of it probably isn't.

This is where the distinction between a web *site* and a web *application* comes into play. A web site is something that primarily has the goal of disseminating information. It has limited requirements in terms of user interaction; usually, simple HTML forms suffice nicely. In a web site, all of the rules described here should almost certainly be followed, and more important, the rules *can* be followed. Web applications, on the other hand, are more complex and require more advanced user interactions. They are meant to replace fat clients, applications that users have become accustomed to over the years. They expect a dynamic user interface (UI) that is powerful and yet simple, bells and whistles, and features that simply can't be done without code—and some of that code will have to wind up on the client. In these situations, my opinion is that following all these rules is simply not reasonable and will lead to a lot of failed projects.

However, for anything that is for public consumption on the Web, you should without question strive for perfect accessibility, graceful degradation, and all the other unobtrusive JavaScript goals. For the places you don't achieve those goals, you should have very clear and solid reasons for not doing so, and you should be utterly convinced that you can't meet the goals of your application while at the same time adhering to these rules.

2. Google's application is called Google Docs & Spreadsheets. You can play with it by going to `http://docs.google.com`.

So, in short, I believe you should examine what you're doing and what your goals are, and decide which of these guidelines to follow. Make no mistake, I do believe you should be trying to follow them all! But that will not always be possible in my estimation. Again, this is one man's opinion. Please do form your own opinion based on your own best judgment.

Don't Use Browser-Sniffing Routines

Rather than using browser-sniffing scripts to determine the capabilities of a browser, check for object existence and capabilities. As an extension to this, JavaScript errors that occur simply because the developer was lazy, and didn't check whether a given object existed before accessing it, are obtrusive and not good.

As an example, look at the following code:

```
function setContent(inObj, inContent {
  inObject.innerHTML = inContent;
}
```

Here, if the object `inObj` does not support `innerHTML`, which is possible since `innerHTML` is not a standard part of the DOM (although, in practice, I don't know of any browser that fails to implement it), an error will occur. Rewriting this to avoid the error is trivial:

```
function setContent(inObj, inContent {
  if (inObj.innerHTML) {
    inObject.innerHTML = inContent;
  }
}
```

While checks like this are good practice in general, the real point is to determine whether a browser supports a certain capability. One of the best examples of this is basic Ajax programming, where you need an instance of the `XMLHttpRequest` object. Unfortunately, various browsers support this in different ways (Ajax will be discussed in Chapter 12, so don't worry about the details if this is new to you). However, you can check for various objects, and based on their existence or nonexistence, branch your code accordingly, like so:

```
var xhr = null;
if (window.XMLHttpRequest) {
  xhr = new XMLHttpRequest();
} else if (window.ActiveXObject) {
  xhr = new ActiveXObject("Microsoft.XMLHTTP");
}
```

It's always better to write code that doesn't need to branch at all, of course, but that just isn't always possible. Browser-sniffing is a bad idea because, as many developers have learned over the years, you always need to worry about keeping the sniffing code up-to-date and able to recognize new browsers. Object-existence checks are much less brittle and don't generally require reworking to handle new browsers, so this technique is preferred over browser-sniffing.

Don't Create Browser-Specific or Dialect-Specific JavaScript

You shouldn't ever, under any circumstances, create JavaScript that is not cross-browser, or more specifically, code that is dialect-specific—well, unless there is an exceptionally good reason!

This is one rule that should be obvious to anyone who has done even relatively trivial JavaScript coding. The simple fact is that JavaScript in modern browsers is pretty close to 100% compatible anyway. There are still exceptions here and there, but you will find that the vast majority of the differences are actually in regard to DOM differences and how to work with the DOM in a particular browser. So, while this guideline refers to cross-browser JavaScript, in reality, it probably has more to do with DOM access.

As a trivial example, you should no longer need to check for things like `document.layers` or `document.all` to determine how to properly access an element on a page. Almost all modern browsers will support `document.getElementById()`, which is the spec-compliant way to do it, and that's what you should be using. Any time you find yourself coding to a specific dialect of JavaScript, or for a specific browser, ask yourself (a) is there a spec-compliant way to accomplish this? and (b) will it work across all browsers I'm interested in supporting? When you find those exceptions, clearly note via comments in the code why you did it. That will save you a lot of mental anguish down the road, and will also remind you which parts of your code to check later to see if you can update to standards-compliancy.

Properly Scope Variables

Variables should be local unless they are truly meant as globals. In particular, be careful when working with Ajax, because global variables in an asynchronous world can be the cause of many difficult-to-debug problems.

As an example of bad scoping, take a look at the code in Listing 2-3.

Listing 2-3. *Bad Variable Scoping*

```
<html>
  <head>

    <script>

      var fauxConstant = "123";

      function badFunction() {
        fauxConstant = "456";
      }

      function goodFunction() {
       var fauxConstant = "456";
      }
```

```
      function testIt() {
        alert(fauxConstant);
        goodFunction();
        alert(fauxConstant);
        badFunction();
        alert(fauxConstant);
      }

    </script>

  </head>

  <body>
    Three alerts will follow... the first should and does say "123."
    The second should and does say "123" again. And the third should say
    "123" but instead says "456."
    <br><br>
    <input type="button" value="Click to test scoping" onClick="testIt();">

  </body>
</html>
```

In this example, notice how the last value displayed is not correct because of how the variables are scoped. The idea is that both `goodFunction()` and `badFunction()` will create a variable with the same name as the global variable, and then use it locally, but *not* change the value of the global version. `badFunction()`, as you can guess, doesn't work that way; it touches the global version. Had `fauxConstant` been declared locally in `badFunction()`, the problem would be avoided, as is the case in `goodFunction()`. If the intent were to actually have a global variable, then `goodFunction()` would be wrong, since it declares a local variable. However, the name *faux*`Constant` should be a hint that the value is not expected to be changed after it is declared and initialized. Since there are no true constants in JavaScript, we can only fake it—hence the name *faux*`Constant`. In short, scope your variables locally whenever possible.

One other point to remember is that any JavaScript variable declared inside a function without the `var` keyword will continue to exist outside that function. This can often lead to difficult-to-diagnose problems, so do yourself a favor and always use the `var` keyword unless you specifically know you have a reason not to!

Don't Use Mouse Events to Trigger Required Events

For accessibility, you should avoid triggering required events as the result of mouse events. `onChange`, while not strictly speaking a mouse event, is often misused. For instance, we've all seen sites with a `<select>` that, when changed, navigates to a new page, as in the example in Listing 2-4. This is generally bad because the page cannot properly be used without a mouse, which means it will be difficult, or even impossible, for those with certain disabilities to use your site.

Listing 2-4. *Inaccessible Page Change*

```
<html>
  <head>
  </head>
  <body>
    <select onChange="alert('Change to page ' + this.value);">
      <option value="page1.htm"></option>
      <option value="page1.htm">Page 1</option>
      <option value="page2.htm">Page 2</option>
    </select>
  </body>
</html>
```

Instead, rely on events that can be activated with the keyboard, as in Listing 2-5.

Listing 2-5. *A More Accessible Page Change*

```
<html>
  <head>
  </head>
  <body>
    <select id="theSelect">
      <option value="page1.htm"></option>
      <option value="page1.htm">Page 1</option>
      <option value="page2.htm">Page 2</option>
    </select>
    <br>
    <input type="button" value="Click to change pages"
      onClick=
      "alert('Change to page ' + document.getElementById('theSelect').value);">
  </body>
</html>
```

This example places a button beside the `<select>`, and the button is what activates the page change. This can easily be activated with the mouse as well as the keyboard, greatly enhancing the accessibility of your page.

It's Not All Just for Show: Accessibility Concerns

Accessibility for the disabled in modern RIAs, especially those using Ajax techniques, is a very difficult problem. Anyone who says differently is probably trying to sell you a solution you probably don't want! The fact is that accessibility is a growing problem, not a diminishing one, and this is due to the nature of "modern" web applications.

Accessibility generally boils down to two main concerns: helping the vision-impaired and helping those with motor dysfunctions. Those with hearing problems tend to have fewer issues with web applications, although with more multimedia-rich applications coming online each day, this may be increasingly less true. Those with motor disorders will be concerned with things

like keyboard shortcuts, since they tend to be easier to work with than mouse movements (and are generally easier than mouse movements for specialized devices to implement).

Often overlooked is another kind of vision impairment: color blindness. Web developers usually do a good job of helping the blind, but they typically don't give as much attention to those who are color-blind. It is important to understand that color-blind people do not usually see the world in only black and white.[3] Color blindness, or rather color deficiencies, is a failing of one of the three pigments that work in conjunction with the cone cells in your eyes. Each of the three pigments, as well as the cones, is sensitive to one of the three wavelengths of light: red, green, or blue. Normal eyesight, and therefore normal functioning of these pigments and cone cells, allows people to see very subtle differences in shades of the colors that can be made by mixing red, green, and blue. Someone with color blindness cannot distinguish these subtle shading differences as well as someone with normal color vision can, and sometimes cannot distinguish such differences at all. To someone with color blindness, a field of blue dots with subtle red ones mixed in will appear as a field of dots all the same color, just as one example. A page that demonstrates the effects of color deficiencies to someone with normal vision can be found at `http://colorvisiontesting.com/what%20colorblind%20people%20see.htm`.

When Life Gives You Grapes, Make Wine: Error Handling

A soberingly short time ago, error/exception handling in JavaScript amounted to little more than hoping the browser would display a not too unpleasant message to the user when something went wrong. Most "real" languages had rather sophisticated exception-handling mechanisms, but JavaScript was not one of them. Java had `try . . . catch` blocks. So did C++, even before Java did. Heck, even the much maligned Visual Basic had `On Error`, which was considered "unstructured" exception handling (as opposed to `try . . . catch`, which is considered "structured"), but even *that* was better than what JavaScript had to offer for a long time!

At some point, a clever JavaScript coder discovered that you could hook into the browser's exception-handling mechanism. So now, instead of a plain-old browser error message, like the one shown in Figure 2-1, you could put in your own (slightly) more pleasant version, as in Figure 2-2.

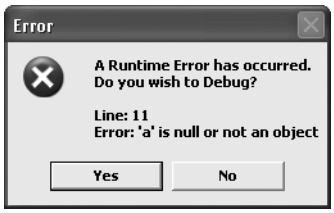


Figure 2-1. *A plain JavaScript error message from Internet Explorer*

3. Seeing only in black, gray, and white is termed *monochromasy* and is actually quite rare. Monochromasy would actually be easier to deal with than the typical forms of color blindness.

Figure 2-2. *A custom JavaScript error message*

The code for a custom error handler is pretty trivial, but it is still often a very useful capability to have in your toolbox. Listing 2-6 shows the code for the page with the handler that generated the message in Figure 2-2, along with the error-generating code to test it. In general, errors can often be handled by structured exception handling instead, but there are still times when a true last-resort error handler such as this is a good idea. In fact, it isn't too hard to convince yourself that a handler like this should *always* be present, even if you make every attempt to avoid it ever being activated, which you should, of course!

Listing 2-6. *An Example of an Error Handler and Test Code*

```
<html>
  <head>

    <script>

      window.onerror = handleError;
      var s = null;
      s.toString();

      function handleError(desc, page, line) {
        s = "An unexpected JavaScript error has occurred.  ";
        s += "We apologize unreservedly!\n\n";
        s += "Page 'test.htm', line " + line + "\n";
        s += "Description: " + desc + "\n\n";
        s += "Please contact customer support at 555-123-4567.";
        alert(s);
      }

    </script>

  </head>

  <body>
  </body>

</html>
```

■ **Note** The difference between an *error* and an *exception* is something that many developers tend to ignore, but it's a fairly important distinction when designing your code. An error is a condition that you do not expect to happen and usually will, and even arguably *should*, lead to a program crash (or, at best, an error message saying the program cannot continue). On the other hand, an exception is a situation you expect can and might happen, and the program should be able to handle it in some way and continue. You will often find that an error is really an exception in disguise; that is, if you think about it a bit and plan accordingly, you can handle it just like any other exception. It may require more work on your part, but that's one of the tickets to writing more robust code!

So, what is this structured exception handling I refer to, in the context of JavaScript? Like Java, C++, and many other languages, the `try . . . catch` construct is at the heart of it. Listing 2-7 shows a simple example of `try . . . catch` in action.

Listing 2-7. *JavaScript Exception Handling in Action*

```
<html>
  <head>

    <script>

      function test(inVal) {
        try {
          inVal = inVal.toLowerCase();
        } catch(error) {
          alert("An error has occurred.  Error was:\n\n" + error.message);
        }
      }

    </script>

  </head>

  <body>
    <input type="button" value="Test" onClick="test(null);">
  </body>

</html>
```

The exception in this code should be pretty easy to spot: it tries to call `toLowerCase()` on a string that was passed in as null. This is the type of thing that may not be caught by a developer at design time, because the conditions that call the `test()` function may be dependent on various factors (here, obviously the developer *should* catch this, since null is passed specifically, but you know what I mean!). Exceptions tend to be things that wouldn't occur until runtime based on some user-generated condition. Even still, this demonstrates how `try . . . catch` works.

In short, some condition that you, as the developer, know could throw an exception is enclosed in `try { }`. This is followed by a `catch { }` block, which will be executed if an exception occurs in the `try { }` block. In the example in Listing 2-7, the exception-handling code does nothing more than pop up an alert message, which sometimes is all you can really do anyway.

When It Doesn't Go Quite Right: Debugging Techniques

Let's face facts folks: we developers are like baseball players in that we probably get it right maybe only three out of every ten tries, and that actually makes us pretty good! What I mean is that the modern development model is quite different from the old days.

I admit I wasn't around for the period when programming was an exercise in patience, but I've heard all about it. Programmers would spend all day writing out programs on special paper, thinking every last detail through as best they could. They then sent those papers down to another department, which entered the program into a machine that spit out punch cards. The next day, the programmer (or a whole other department sometimes) would feed those punch cards into the computer (and I'm not even going to mention the times someone would trip while bringing the box of punch cards somewhere, and then frantically try to reorder hundreds or thousands of cards before his boss noticed!). Then the programmer sat around, waiting for some output somewhere to verify his program was correct, or whether he had to start all over again.

Today, things are quite considerably better. We generally get immediate feedback about the correctness of our programs. In fact, we often get earlier hints about mistakes we've made. Correcting them is a simple matter of typing in some new code and clicking a button. Yes, we definitely have it good compared to just a few (relatively speaking) years ago.

Even so, how many times do you write more than a handful of code and have it work perfectly the first run? It's a pretty rare thing. There's a reason the saying "Programmers curse a lot, but only at inanimate objects" was invented!

In the world of JavaScript, things are getting better at a breakneck speed. That being said, debugging JavaScript in a modern web application is usually not the most pleasant of experiences. It is not as bad as the punch card days, but generally not as nice as working in modern fourth-generation languages (4GLs). IDEs have only relatively recently begun to support JavaScript fully in terms of debugging capabilities and static code analysis capabilities. So, we still need to develop our own debugging techniques and learn to put them to good use. Of course, a proper debugger is no longer as rare as it was two to three years ago, and so some of the techniques are beginning to give way to using debuggers.

Perhaps the earliest debugging technique, if one can really call it that, was "alert debugging." This amounts to sprinkling `alert()` calls throughout your code to display various messages. For instance, let's say you need to debug the code in Listing 2-8.

Listing 2-8. *Using alert() Debugging*

```
<html>
  <head>

    <script>

      function test() {
        var a = 0;
        alert("checkpoint 1");
        a = a + 1;
        alert("checkpoint 2");
        a = a - 1;
        alert("checkpoint 3");
        a = a.toLowerCase();
        alert("checkpoint 4");
      }

    </script>

  </head>

  <body>
    <input type="button" value="Test" onClick="test(null);">
  </body>

</html>
```

In this example, you know you are seeing an error somewhere in the function. So, you sprinkle some `alert()` calls throughout, showing some checkpoint messages. When you view this page and click the Test button, you'll get a series of pop-ups, and eventually the error will occur. You now know that the error occurs between the pop-up showing "checkpoint 3" and "checkpoint 4." In effect, you've created a rudimentary "step-into" debugging facility of a sort. Now, this certainly can get the job done, and I often find myself doing it simply because I've gotten so quick and efficient at it. That being said, it clearly isn't the best answer. What if there were a loop involved in this code that iterated 300 times? Do I really want to be clicking through 300 alert pop-ups? Heck no!

What are the alternatives? Well, if you do your development in Firefox, you have access to a great tool call Firebug, which I'll discuss in more detail in the next section. As a preview though, Firebug offers logging to a console. So, the code from Listing 2-8 can be changed to that in Listing 2-9.

Listing 2-9. *Firebug Console Logging*

```
<html>
  <head>

    <script>

      function test() {
        var a = 0;
        console.log("checkpoint 1");
        a = a + 1;
        console.log("checkpoint 2");
        a = a - 1;
        console.log("checkpoint 3");
        a = a.toLowerCase();
        console.log("checkpoint 4");
      }

    </script>

  </head>

  <body>
    <input type="button" value="Test" onClick="test(null);">
  </body>

</html>
```

Now, instead of a bunch of pop-ups, if you look in the Firebug console, you'll see the messages displayed there. Sweet! Note, however, that trying to run this in IE will result in errors, because the `console` object isn't known to IE. If you wanted to work in IE, you could create the following code and add it to the page (as a script import most likely):

```
function Console() {
  this.log = function(inText) {
    alert(inText);
  }
}
console = new Console();
```

Of course that goes back to logging to an `alert()` popup, so you would probably instead want to write out to a `<div>` that is on the page. But the basic idea is to somehow emulate the `console` object, and this does the trick, if not perfectly.

But, what if you don't work in Firefox or don't have Firebug installed? What if you need to send the code to a client's site, and you can't assume that client has Firefox and Firebug? Well, one option is to write your own simple message logger. This is actually part of the project in Chapter 3, so I'll save it for then, but suffice it to say that it's a relatively trivial exercise.

At the end of the day though, a logger is really just a less annoying implementation of `alert()` debugging, isn't it? "What about a proper debugger?" I hear you ask. You have it in your IDE of choice when doing development in C/C++, Java, Visual Basic, or just about any other language you use, right? If JavaScript is going to play with the big boys, it has to come to the party well equipped. Well, guess what? *JavaScript debugger* used to be almost an oxymoron, but no more! There are now quite a few options—some better than others, some free, and some not, but they most certainly exist. Let's talk about those debuggers and some other tools that all JavaScript coders should have in their toolbox.

Browser Extensions That Make Life Better

The web browser isn't just for rendering markup any more! Modern web browsers are really their own runtime platform for running other bits of software. Some are better than others, but all offer some extensibility in the form of extensions. In this section, we will look at a just a few of my personal favorites that I find help me do my day-to-day development work. Of course, I can only hope to scratch the surface in terms of what is available. I believe I've covered probably the most useful in each browser (for a developer I mean), but explore on your own, because there are plenty more out there!

Firefox Extensions

Whether you use IE, Firefox, Opera, or some other browser on a day-to-day basis, I very much recommend doing your primary development in Firefox. The reason is twofold. First, Firefox tends to be a bit more standards-compliant than other browsers, most notably IE, so your code and markup developed in Firefox will tend to be more standards-compliant. Second, Firefox has some of the best client-side development tools available today, and nearly all of them are totally free!

You can find all sorts of Firefox extensions (or *add-ons*, which is another term for the same thing) by opening Firefox, clicking the Tools menu, and selecting Add-ons. A sidebar will open, and there you will see an icon that looks like a little gray gear with a black down arrow next to it. Click the gear to open a menu that lists Firefox Add-ons near the bottom. Click that item, and you'll be taken to the Firefox Add-ons page. Alternatively, you can simply navigate to `https://addons.mozilla.org/firefox/extensions`. On the Add-ons page, you can browse through all the available extensions.

Now I'll talk about a few of the extensions I personally find to be the most useful, but I very much recommend taking some time to browse for yourself, because there is plenty more where these come from!

Venkman

It is truly amazing to think that something as powerful as the Venkman debugger is 100% free! All your favorite debugging tricks are available here, including call stack navigation, the ability to watch the values of specified variables, breakpoints in code, and real-time changing of variable values to see how the code reacts. As you can see from Figure 2-3, Venkman looks a whole lot like any of the debuggers you've probably used on the server side of things.

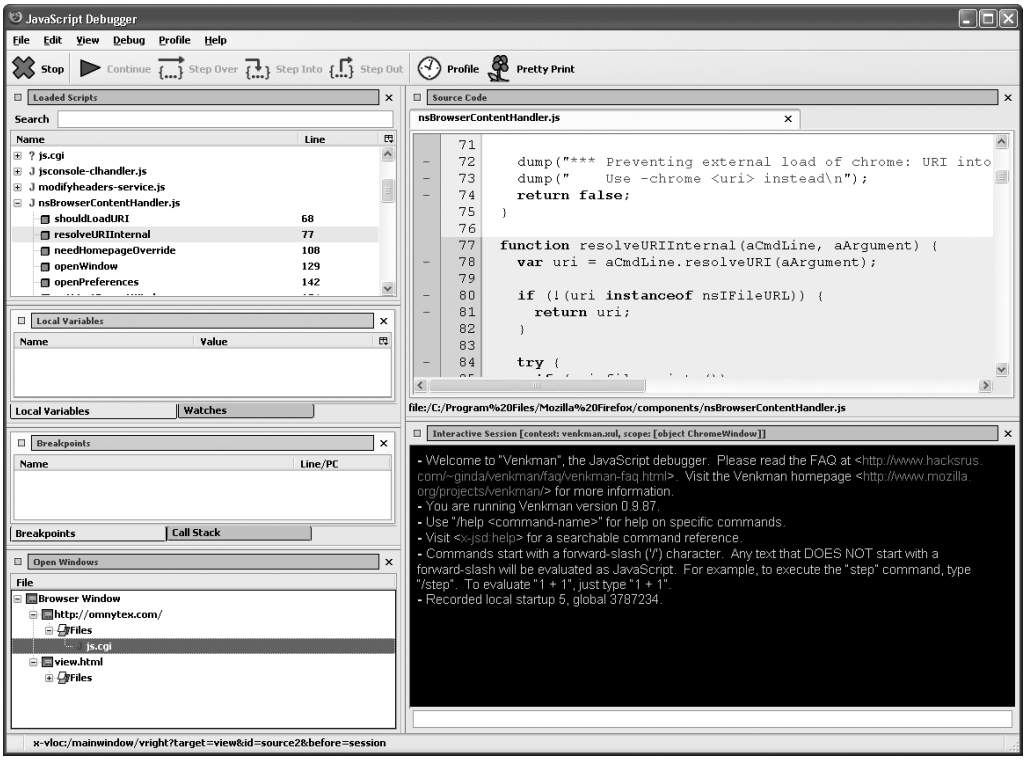


Figure 2-3. *The Venkman debugger for Firefox*

Firebug

Firebug has very quickly garnered the reputation as one of the most popular developer extensions for Firefox in existence today. In fact, a great many of us have taken to using almost nothing but Firebug for our client-side development efforts. Little else seems necessary!

Firebug offers a number of different capabilities all rolled into one nice, neat package. For instance, the Console tab, which you can see in Figure 2-4, shows errors and warning of various kinds, with filtering capabilities. A really nice thing about it is that when an error occurs, you can expand the error and see the full stack trace. Each item in that list is clickable and brings you directly to the offending line. Also, this console is accessible to your applications by simply doing this:

```
console.log("message");
```

This is exceedingly handy!

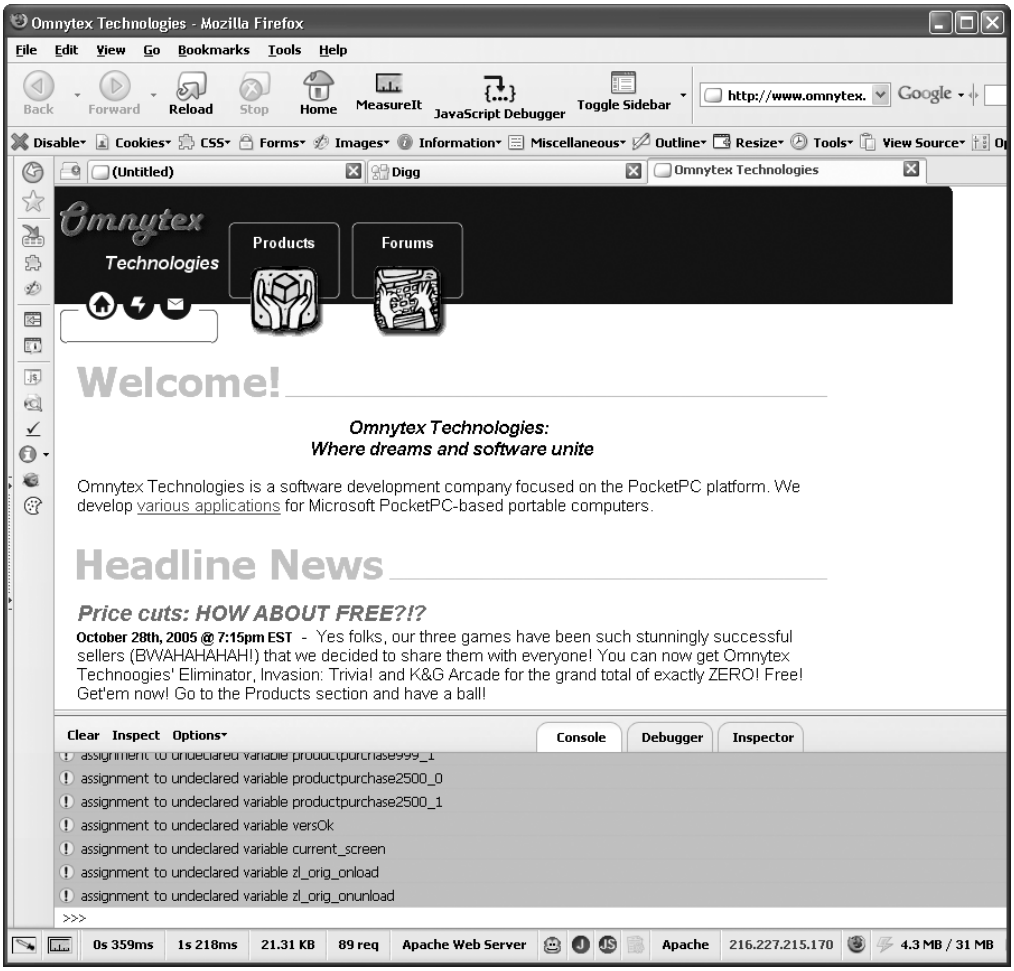


Figure 2-4. *Firebug: perhaps the single most important developer extension for Firefox to date!*

Firebug registers Ajax requests, which few other extensions I've seen do. You can expand the request and see the parameters that were passed, the POST body, the response, and so on. If you're doing Ajax work, this is absolutely invaluable.

Firebug also provides a debugger that shows full stack traces, as well as the ability to change values in real time and set breakpoints. The debugger is shown in Figure 2-5.

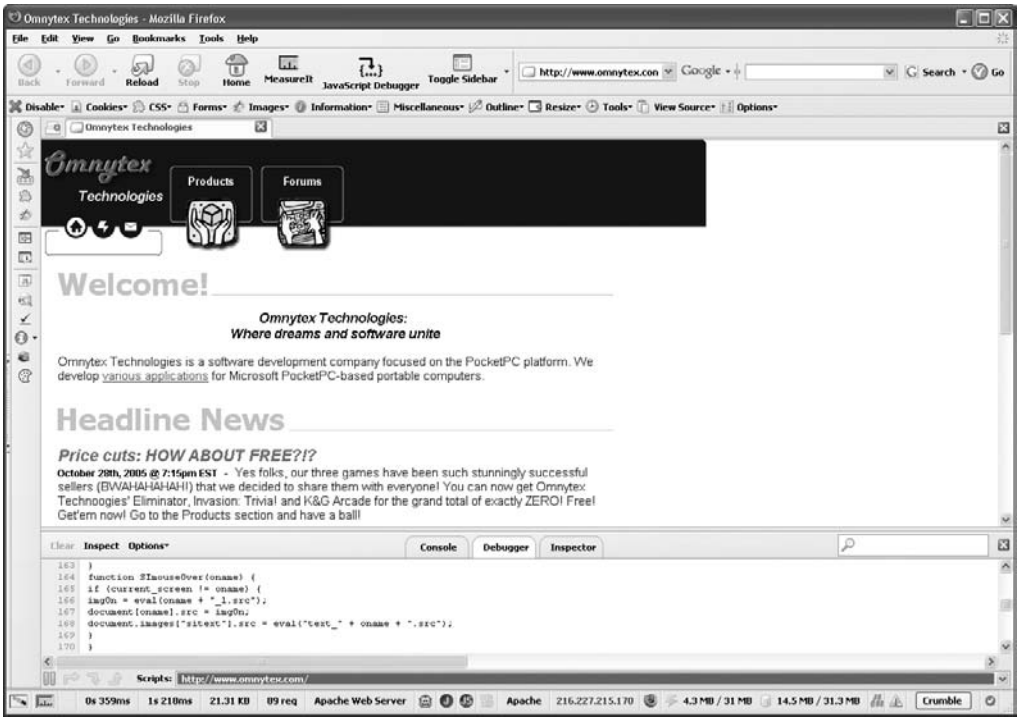


Figure 2-5. *Firebug's debugger, while extremely simple, is at the same time very powerful.*

Firebug's Inspector facility is another great feature. It allows you to hover over items on the page and see their definition, including their styles, where they were inherited from, and so on. You can explore the DOM tree at any point, digging down as far as you like.

Keep in mind that I've only scratched the surface of what Firebug can do! In short, it takes many popular features from other extensions and rolls them into one tidy, powerful package. By the way, you can put Firebug on the side rather than the bottom; I just choose to have it at the bottom, as reflected in Figures 2-4 and 2-5.

If you primarily do your development in Firefox, which I recommend, and if you install no other extension, install Firebug!

Page Info

Page Info is an immensely useful, yet immensely simple, Firefox tool, as shown in Figure 2-6. Page Info actually comes with Firefox, and is accessible from the browser's Tools menu.

Figure 2-6. *Page Info, another invaluable Firefox tool*

Page Info displays, as its name clearly states, information about the current page, such as the following:

- The render mode the page uses
- A list of all the forms on the page and all the pertinent information about them
- A list of all the links on the page
- Links to all the images and other media resources on the page (plus dimensions for images and other information about each)
- A list of cookies the page uses (with the ability to remove each one or all of them together)
- A list of scripts and style sheets used on the page (with the ability to open each separately)
- A tree view of all the page's dependencies (images, scripts, applets, and so on)
- All the HTTP headers for both the request of the page and the response, and security-related information for the page

If this sounds like an absolute treasure trove of information to you, then you've definitely gotten the right picture!

Web Developer

The Web Developer toolbar is another extension that you won't want to do without! This extension, in the form of a toolbar, offers such a tremendous wealth of features that I simply can't cover even a quarter of them. So, I'll just throw out a list of some things it lets you do, in no particular order. But keep in mind, and I can't emphasis this enough, the following is a very small portion of what it offers:

- Disable all sorts of things, like JavaScript, meta redirects, page colors, and so on.
- View cookie information, as well as clear individual cookies, or all cookies for a domain, and so on.
- Edit CSS used on the page, disable CSS entirely, or just view style sheets that were imported.
- Change the method of a form, make disabled form fields writable, change `<select>` elements to text fields, and so on.
- Manipulate images in a number of ways, as well as get all sorts of information about them.
- Get virtually any piece of information about a page you can imagine.
- Make locked frames resizable.
- Outline virtually any type of page elements you want, so you can see tables, `<div>` elements, forms . . . whatever you wish, clearly.
- Resize the window, as well as automatically resize it to any of the most common window sizes (great to see what your page looks like on a 640-by-480 monitor, for example).
- Get quick access to numerous online validators, as well as the JavaScript and Java consoles.
- View generated source, an absolutely invaluable aid!

You may by now realize that this toolbar has a great deal of overlap with the Page Info tool. However, Page Info presents it in a more well-organized manner, and makes it a little easier to access. Either one will do the trick, though.

Five minutes with this toolbar is probably enough to convince you of its merit, so I suggest taking those five minutes now and having a look. Go ahead, I'll wait.

Back already? OK, let's move on to some IE extensions.

IE Extensions

When you compare the landscape of browser extensions available in Firefox vs. IE, at least as far as developer tools go, you quickly conclude that Firefox has the edge. Heck, just putting Firebug against most of what is available for IE is a win for Firefox! However, that isn't to say that there aren't some excellent tools available for IE.

HttpWatch

HttpWatch (`http://www.httpwatch.com`), shown in Figure 2-7, offers the ability to capture requests and responses between the browser and a remote system. And when I say capture them, I mean *capture* them! Every available detail is recorded for your analysis, and better still,

it is organized very well, making it easy to get the information you need. One of the best features is that it shows you the raw HTTP data stream that was sent or retrieved. This can help a great deal in debugging some tricky issues.

Figure 2-7. *In the world of Internet Explorer, HttpWatch is unmatched.*

The unfortunate thing about HttpWatch is that it isn't free, and it isn't especially cheap for an individual. That being said, it's a truly helpful tool that will pay for itself in short order. Grab the demo and have a look!

Web Accessibility Toolbar

The Web Accessibility Toolbar (`http://www.visionaustralia.org.au/ais/toolbar`) is a great aid in making sure your site is accessible, and it offers other useful features. It is akin to the Web

Developer toolbar in Firefox, but not quite as feature-rich (which is to be expected, since this toolbar has a bit narrower focus). Here are some of the features it offers:

- The ability to resize the window to common window sizes
- Quick and easy access to a large number of online validators
- Manipulation of and information about images on the page
- Color contrast analysis
- All sorts of page structure analyses (to help ensure your page is properly readable by screen readers)
- The ability to simulate various disabilities, including color blindness and cataracts
- Tons of page information displays

Accessibility can be difficult to implement—sometimes nearly impossible with modern RIAs. This toolbar will give you a good leg up on that difficult task, and even if for some reason you have no concern about accessibility, this would still be a great tool. It's free, so I can't think of a single good reason not to add it to you repertoire.

IEDocMon

IEDocMon (`http://www.cheztabor.com/IEDocMon/index.htm`) is another one of those tools that makes you wonder how someone released it for free! This IE extension allows you to view the page's DOM tree, expanding down to the point you need, as shown in Figure 2-8. It can highlight the element on the page from the tree, so you can be sure you are looking at the right thing. It can show you the snippet of HTML representing the current element, so you can find precisely what you're looking for (a huge help when you're trying to figure out how some clever developer pulled off a specific trick!). Moreover, it can do the same thing with scripts, so you can focus in on the precise bit of script that performs a given task.

One of IEDocMon's best features is its ability to monitor events for selected elements. Say you have a `<div>`, and you have it changing colors when you mouse over it, but it doesn't seem to be working. With IEDocMon, you can select the `<div>` from the DOM tree, and you will see every event that occurs for that element. Especially in a complex RIA, this is a capability that is worth its weight in gold, and I frankly haven't found many other extensions, for any browser, that can do it.

Figure 2-8. *IEDocMon: anyone who says free isn't good hasn't seen this!*

Visual Studio Script Debugger

The Microsoft Visual Studio Script Debugger, a part of Visual Studio, is a full-fledged just-in-time debugger, as shown in Figure 2-9. It can intercept errors on the page in IE and pop up to show you the offending line and allow you to manipulate the code on the fly.

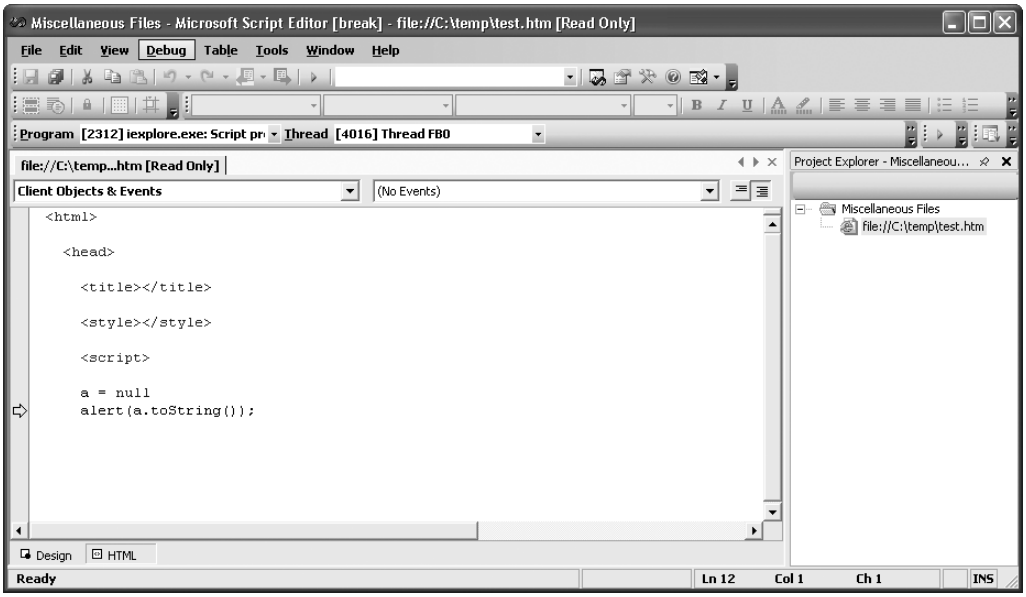


Figure 2-9. *Visual Studio Script Debugger makes debugging in Internet Explorer a tolerable experience.*

As a part of Visual Studio, this debugger is not only not free, it also is fairly heavyweight. If you are fortunate enough to have a subscription to MSDN, or already have Visual Studio, this debugger can be a big help when working in IE.

Unfortunately, there aren't many debuggers for IE in the first place, so your choices are somewhat limited. If you must use IE, this debugger will serve you pretty well, as long as you aren't looking for something particularly svelte and sprightly.

Microsoft Script Debugger

Not to be confused with the Visual Studio Script Debugger, there is also a separate Microsoft Script Debugger. This debugger is not as full-featured as the Visual Studio Script Debugger. Although it is more lightweight, it can still be quite useful. Once installed, it exposes itself via a new Script Debugger menu in IE.

If you have access to Visual Studio, you will want to use that debugger. Otherwise, have a peek at the Microsoft Script Debugger at `http://www.microsoft.com/downloads/details.aspx?FamilyID=2f465be0-94fd-4569-b3c4-dffdf19ccd99&DisplayLang=en`.

Microsoft Internet Explorer Developer Toolbar

Microsoft recently released a new developer's toolbar for IE that is very much along the lines of the Web Developer toolbar for Firefox. It isn't quite as extensive yet, but it is, as of this writing, a beta release, so it certainly could be expanded in the future. Here are a few of its features:

- The ability to outline tables, images, selected tags, and other items
- The ability to resize the browser window to a specified size

- A full-featured design ruler (to help align and measure items)
- The ability to explore the DOM of the current page in a tree view

If you would like to check this out for yourself, you can do so at this (rather long and unwieldy) address: `http://www.microsoft.com/downloads/details.aspx?familyid=e59c3964-672d-4511-bb3e-2d5e1db91038&displaylang=en`.

Maxthon Extension: DevArt

Maxthon is my day-to-day browser of choice. It is a wrapper around IE that provides many of the more advanced features IE is lacking, but keeps the underlying IE rendering engine intact. This means I rarely, if ever, have to worry about a site not working for me. It also deals with many of the security flaws IE tends to have, so it's safer than "naked" IE to boot. But we're not here to talk about which browser is better or more secure, we're talking about developer tools!

Like Firefox, Maxthon (`http://www.maxthon.com`) has a much more robust extension architecture than does IE. One of those extensions is DevArt (`http://forum.maxthon.com/index.php?showtopic=14885`), which provides a number of very handy developer features, as shown in Figure 2-10.

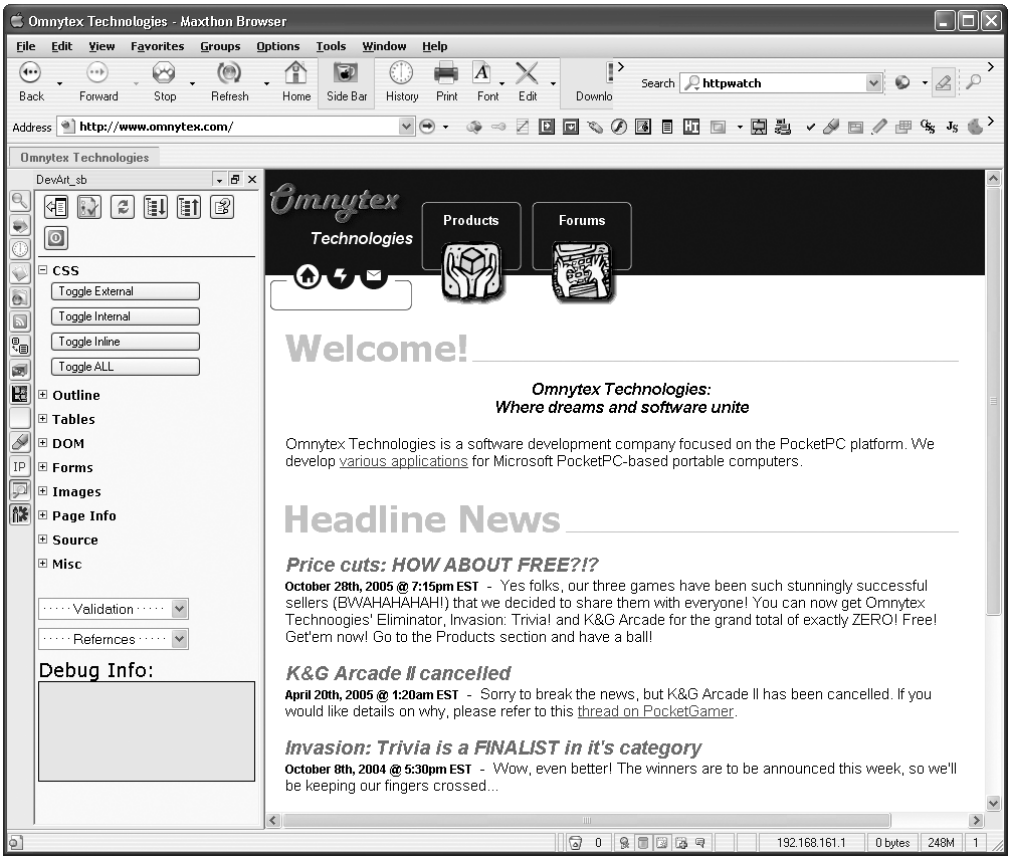


Figure 2-10. *DevArt, an excellent extension for Maxthon*

DevArt provides the following capabilities:

- Toggle style sheets on and off.
- Put outlines around tables, images, and `<div>` elements.
- Remove tables.
- Display DOM trees.
- Show hidden input fields on forms and show the values of input fields.
- Display the sources of all images on a page as well as their dimensions.
- Show response headers for the current page.
- View the generated source for the page (that is, what the browser actually used to render the page).
- Validate the current page in various ways.

All of this makes DevArt a must-have for developers if you use Maxthon. DevArt comes in two flavors: a toolbar and a sidebar.

JavaScript Libraries

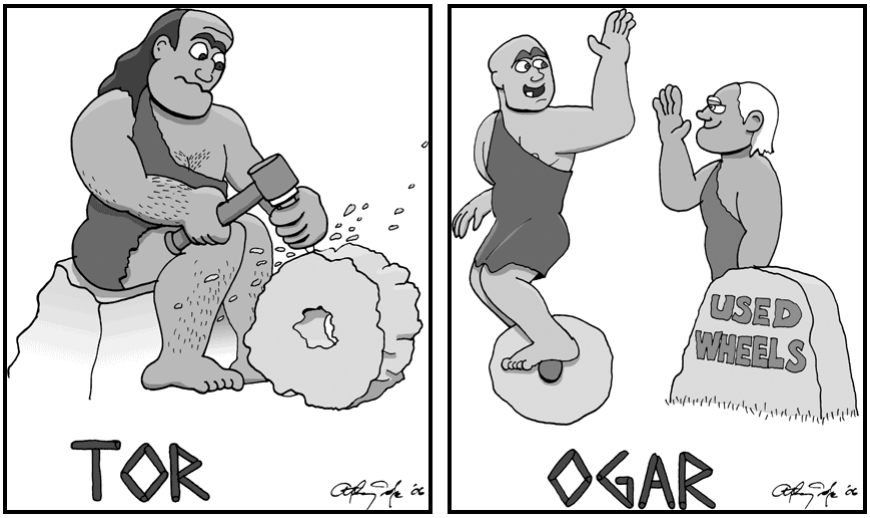
JavaScript libraries have grown leaps and bounds over just the past two to three years. It used to be that you could spend a few hours scouring the Web looking for a particular piece of code, and you would eventually find it. Often, you might have to, ahem, appropriate it from some web site. Many times, you could find it on one of a handful of "script sites" that were there expressly to supply developers with JavaScript snippets for their own use.

Larger libraries that provide all sorts of bells and whistles, as exist in the big-brother world of Java, C++, PHP, and other languages, are a more recent development in the world of JavaScript. In many ways, we are now in a golden age, and you will find almost more options than you would want!

Some libraries out there focus on one area or another: GUI widgets, Ajax, UI effects, and so on. Other libraries try to be the proverbial jack-of-all-trades, covering a wide variety of areas such as client-side storage, widgets, Ajax, collections, basic JavaScript enhancements, and security.

The one thing they all have in common is that their quality is light-years beyond what came before, and they all will make your life considerably easier! There's usually no sense in reinventing the wheel. If you are doing Ajax, unless you need absolute control over every detail, I can't think of a good reason not to use a library for it. If you know your UI design requires some more advanced widgets that the browser doesn't natively provide, these libraries can be invaluable. Do you need to store some data client side and want to use Flash scope (covered in Chapter 6)? Let a library handle the details for you!

In this section, I'll introduce you to some representative libraries, give you a basic overview of what they offer, and point you to more expansive documentation on them. These libraries will be used in the projects throughout the book, so you'll get to see some real-world examples of their usage.



No point in reinventing the wheel, so USE THOSE LIBRARIES, lest you wind up like Tor!

There are oodles of libraries out there today, and it would be impossible to cover more than just a handful here, and even these will not be covered in excruciating detail. So, please don't look at this as the definitive reference on these libraries. I'm offering a brief introduction to whet your appetite. I'm confident that once you read this section, and look at the usage of these libraries in the projects to follow, you will want to check them out yourself in much more detail! Until you explore them on your own, I guarantee you won't get the full feel for the benefits they offer.

Prototype

Some libraries focus squarely on Ajax; others are concerned with GUI widgets; still others provide all sorts of whiz-bang effects you can easily add to your pages. Prototype is something that exists at a layer below most of that, as evidenced by the fact that many libraries are actually built *on top* of Prototype.

Prototype (`http://prototype.conio.net`) can, in a sense, be viewed as an extension to JavaScript itself. On a more technical level, it works a lot of its magic by quite literally extending some of the built-in JavaScript objects.

As is the case with most of these libraries, the only way to really get your brain wrapped around Prototype is to explore it and use it. That's one of the points of the projects in this book, which begin in the next chapter. However, I want to highlight a few of the more notable things Prototype provides:

- Prototype includes a number of shorthand utility functions, including `$()`,`$()`, which is a shortcut to writing `document.getElementById()`. Likewise, `$F()` returns the value of any form field. There are others, but these are the two I find to be the most useful.

- Prototype offers relatively basic Ajax support via its Ajax object. One of its most useful features is `Ajax.Updater`, which provides a quick and easy way to fill an existing page element with the response from the server, assumed to be HTML. This is by far the most common thing to do in the realm of Ajax.

- The `PeriodicalExecuter` object provides a simple way to set up a piece of code to be executed repeatedly at a known interval. This saves you from having to set up timeouts and such.
- Prototype extends the built-in `Array` object to provide some excellent added value, such as an `Enumerable` interface so that you can iterate over an array very cleanly, without having to set up a `for` loop using the length of the array.

Some people have an aversion to Prototype because of the way it extends built-in objects, which can cause some subtle problems. There are those who will not touch Prototype, or any library that uses it, because of this. My take—from my own experience with Prototype, as well as a lot of reading on the issues—is that while the problems are not to be ignored, they are not enough to stop me from using Prototype. Still, you should be aware of the potential issues in case they do come up.

Dojo

Dojo (`http://dojotoolkit.org`) is probably one of the fastest growing and most popular libraries out there today. It has a massive scope, seeking to provide just about everything you need to effectively do modern client-side development.

In my previous book on Ajax (*Practical Ajax Projects with Java Technology*), I said that Dojo had one problem at that point: lack of documentation and good examples. I said that using Dojo means fending for yourself, often having to look over the source to figure out how to do things. While I can't yet reverse that opinion, I can back off of it a little. Dojo has improved a good bit in all these regards. The documentation is coming along nicely, and you can find more examples now. This is the logical progression you would expect from a library that is obviously run by people who know what they're doing, as is the case with Dojo.

Also, IBM, Sun, and some other vendors have pledged support for Dojo, and one of the primary areas they talk about helping with is documentation. So, there is every reason to believe that this evolution will continue, and perhaps even quicken, in short order. Until then, I still do highly suggest you sign up for the Dojo mailing list if you intend to use this library. Plenty of helpful people will do their best to answer your questions. However, please do bring a good dose of patience with you, because it often does take a day or two to get a useful response. But the responses usually *will* come, and that's what counts!

So, what does Dojo have to offer? Tons! As I mentioned, Dojo has a very wide scope, but here are a few things that I find to be of immense interest:

- The widgets! Dojo is, I think it's fair to say, best known for its many widgets. Some are definitely better than others, but that's to be expected. They all extend from the same basic widget framework, so they expose a similar set of baseline functionality. This makes working with them fairly easy for the most part. Some of the more noteworthy widgets include the following:
  - The Fisheye, which emulates the Apple launcher bar, with its expanding icons as you mouse over them
  - The tree widget, which gives you an expanding/collapsing tree interface similar to Windows Explorer's folder list

    - A very nice slideshow widget
    - A widget for inserting Google maps into your pages
    - Analogies to most of the basic HTML form elements, as well as expanded form elements, such as buttons with arrows to open a drop-down list, a rich text editor, a date picker, a color picker, and combo box
- Dojo's support of Ajax is very, very good. I can't go so far as to say it's the best, but you can't go wrong with it.
- Dojo provides most of the commonly used effects, such as wipes and fades, and makes it very easy to do them.
- Dojo provides drag-and-drop support that is drop-dead easy to use.
- The storage support Dojo provides is unique as far as I can see. It offers the ability to use durable client-side data stores such as Flash storage (this is essentially cookies on steroids, provided by the Adobe Flash plug-in).
- Dojo contains a number of collection implementations and other commonly used data structures that can make your JavaScript code much more robust and more akin to what you write on the server side of things.
- A number of "core" libraries are part of Dojo. They provide things like advanced string manipulations, simplified DOM manipulation, functions to make JavaScript itself easier and more powerful, and functions specifically for manipulating HTML.
- Dojo contains some basic cryptography functionality and some more powerful math-related functionality.

Although serious improvements have been made to Dojo since I wrote about it in my Ajax book, I still feel it necessary to put up a slight caution sign: Dojo will occasionally give you fits. I am currently using it on a very complex project, and while it cooperates and makes life better the vast majority of the time, there are still days when I have to fight with it a bit. Now, to be fair about it, some of that (maybe even most of that) is due to me not being an expert in it. I'm learning about all Dojo has to offer and how it works, right along with everyone else! However, I have found definite bugs here and there, and have found things that could probably work a little better. All that this means is that you should be prepared to take the initiative when working with Dojo. Don't just expect that it will be plug-and-play, even though you'll find more and more that is indeed the case. You will have questions, and some of them won't be answered by any existing documentation. Google will help sometimes, but often you will find you need to ask someone, which is where the mailing list comes into play.

At the end of the day, Dojo, in my opinion, offers so much that it is ultimately a no-brainer in terms of whether it's worth it or not. It is! Any problems you may encounter and have to overcome will be more than balanced by how powerful it is and by how much time and effort it ultimately saves you. Dojo has a tremendously bright future, and even in the year since I first wrote about it, I have clearly seen the improvements. Give it a shot—it's getting a lot of press for very good reason!

Java Web Parts

Java Web Parts (`http://javawebparts.sourceforge.net`) is a project that is geared toward Java developers, but provides some JavaScript functionality as well. The way it provides this functionality is a bit unique: it is part of a component called JSTags, which is a tag library that emits JavaScript. A number of useful functions can be found there, including the following:

- JSDigester, a client-side implementation of the popular Jakarta Commons Digester component
- A function to convert a form to XML
- Functions for working with cookies
- A function for validating string input
- A function for disabling right-click functionality

If you're not a Java developer, you can steal the JavaScript the tags emit and use it independently. If you're a Java developer though, the tag library will make your life easier.

Also of note is another tag library in Java Web Parts: UI Widgets, which provides a couple of good widgets, including a pop-up calendar and a swapper.

Finally, the AjaxParts Taglib (APT) is, I believe, one of the best Ajax libraries around. This is one you won't be able to use unless you are a Java developer, but if you are a Java developer, prepare to have Ajax become as easy as pie! APT allows you to add Ajax by doing nothing but adding tags to your page and configuring some XML. Every Ajax function that occurs on your page is defined in an XML configuration file—there is *zero* JavaScript to write yourself! All the most common Ajax functions are built in, which should cover your needs probably 95% of the time. For the other 5%, APT is extensible in a very simple and logical manner. So, if you need to do something more advanced, you can do so, and only have to write the basic JavaScript that your particular case needs; you still will not need to write the underlying Ajax code. All of this makes APT an excellent choice for those doing Java web development.

Script.aculo.us

Script.aculo.us (`http://script.aculo.us`) is one of those libraries built on top of Prototype. Script.aculo.us offers functionality in a couple of areas, but frankly, it's best at one thing: effects. If you're looking for fades, wipes, animations, and that sort of thing, script.aculo.us is one of the first libraries you should consider. Here are some of the items it offers:

- Five core effects: opacity, scale, moveBy, highlight, and parallel. Parallel is an effect that allows you to combine effects, which leads to the next item.
- Combination effects, which can be thought of as more advanced effects, created by combining more than one core effect. Examples of these are shakes, pulsate, slideDown, and squish.
- A few controls, such as an auto-complete input box and in-place editing of content.
- The Builder object, which makes it easier to build DOM fragments in JavaScript.

Script.aculo.us also offers capabilities in the areas of unit testing and functional testing, which is pretty unique among libraries. If you've ever used JUnit, this support will look rather familiar!

I strongly suggest cruising over to the script.aculo.us page and spending some time looking at the various demos there, especially if the effects mentioned earlier piqued your interest. Seeing them in action is the best way to appreciate what this library has to offer. Don't ignore the other stuff, though. You'll find some good features that have nothing to do with effects! But if effects are your game, than script.aculo.us is your name (uh, or something like that!).

Yahoo! User Interface Library

The Yahoo! User Interface Library (`http://developer.yahoo.com/yui`), or YUI Library, as it is often called, has garnered a lot of attention since its introduction earlier this year. It provides a collection of UI widgets and a collection of commonly needed JavaScript functions, all in a very clean, well-documented package. While the YUI Library isn't as eye-catching as some other libraries out there, the widgets it offers are simple, easy to use, and relatively lightweight, as is the entire library. The following are some of the items it provides:

- Simple cross-browser logging
- An `Event` component, which allows you to do things like attach events to elements, execute code when a DOM element is detected, and fully abstract the browser event model from your code, among a host of other event-related things
- The `ConnectionManager` object, which provides Ajax functionality in a clean and simple way
- Some utility functions for manipulating DOM, such as a handy feature that lets you get the viewport width and height in a cross-browser fashion (something that can be quite tricky!)
- Basic animation support, including motion along a curve and scrolling
- A fairly rich drag-and-drop utility (actually one of the more robust drag-and-drop implementations I've seen, providing a great number of events to hook into)
- UI widgets, including a calendar, a drop-down menu, a slider, a tree view, and a number of containers for organizing your UIs

The YUI Library is a little odd in the sense that when you first look at it, you may not see its true power. Take some time and explore the documentation, which is excellent, as well as the examples. After a few minutes, I suspect you'll begin to see how good and useful it really is.

MochiKit

"MochiKit makes JavaScript suck less." That's the site's tagline, and who am I to disagree? Indeed, some of the features offered by MochiKit (`http://www.mochikit.com`) most certainly do exactly as the tagline says.

Have you ever tried to do rounded corners on tables? Have you looked it up and seen exactly how many tricks and techniques there are to do this? There are tons, and it's surprisingly difficult to do well and in a cross-browser fashion at the same time. MochiKit does it for

you! It may not seem like much, but rounded corners can really make a page look a lot better when used properly.

MochiKit has a neat feature that displays the source of the current page in a very nicely formatted way. While I agree this may not be the most useful feature in terms of end users, it can be a great thing for developers!

MochiKit also provides a client-side sortable table widget and a cross-browser key event-handling mechanism. On the demo page of the MochiKit site, you'll also find some neat examples, such as a live regular expression evaluator and a minimal JavaScript interpreter. All of this shows that MochiKit has some very interesting capabilities to explore.

Rico

Next up on our parade of libraries is Rico (`http://openrico.org`), which bills itself simply enough as "JavaScript for Rich Internet Applications." Rico offers functionality in four key areas: Ajax, drag-and-drop, cinematic effects, and behaviors. It has a fairly limited focus, so you would expect it to cover these areas pretty well, and indeed it does. Here is a brief summation of what it provides:

- In the Ajax department, Rico offers a nifty feature to populate form elements automatically from an Ajax request. It also provides the prototypical `innerHTML` change from an Ajax request and does so in a simple way.
- In the drag-and-drop department, the basics are covered well. It also offers some more advanced features, such as customized drop zones and custom draggability for elements.
- In the cinematics department, Rico offers the ability to move and resize elements easily, as well as the ability to easily round the corners of a section.
- In the behaviors department, you'll find things like accordion, which allows you to turn a collection of `<div>` elements into an excellent accordion. Also present is the live grid, which takes an ordinary HTML table and hooks it up to an Ajax data source, allowing for real-time loading and sorting of data, among other things.

I was particularly impressed with the accordion behavior and the ability to round the corners of arbitrary sections. Both of these are top-notch implementations and things I intend to use in my own work quite a bit. That isn't to shortchange the rest of what Rico offers, but in pointing out highlights, those things stand above the rest for me.

Mootools

As the saying goes, last but not least, we have Mootools (`http://mootools.net/`). Mootools is a library I discovered late in the process of writing this book, but I thought it was definitely something I wanted to include. Mootools is a very lightweight and modular library with a number of pieces that can be included or not included at your discretion. It is a fully object-oriented library and designed to be developer-extensible. Its various modules cover a wide range of needs, including the following:

- A JavaScript chain of responsibility (CoR) pattern implementation[4]
- Tons of effects, transitions, and effects-related utility functions
- Ajax functionality (not too different from most other libraries, but a nice, simple, clean implementation makes using it a breeze)
- Functions to work with cookies, create and consume JSON, work with the browser window, and helpful string utilities

One especially cool thing about Mootools is actually outside the library itself, and that's its download page! Rather than the typical "download this and that" type page, it instead presents the list of modules offered, from which you select what you want. Your download is then generated on the fly based on your selections, including compression! You can create your own custom Mootools library quickly and easily with this handy tool.

Summary

Whew! This chapter has been quite a whirlwind of topics! We opened up by discussing object-oriented techniques in JavaScript a bit more beyond what was discussed in Chapter 1. We then looked at the relatively new term *unobtrusive JavaScript* and discussed what it means and when and why it should be applied. Tied in with that was the concept of graceful degradation, and how it still applies today as much as ever. We then discussed accessibility concerns and some ways to keep our sites accessible to those with disabilities. Next, we took a look at more robust error handling in JavaScript than used to be possible. After that came a tour of debugging techniques and tools available to us nowadays. Lastly, we looked at some browser plug-ins that make our lives easier, as well as some of the popular JavaScript libraries out there that save us time and effort in spades.

Armed with this knowledge, the next chapter will begin the barrage of projects that form the core of this book. Let's have some fun, shall we?

4. You can argue that it isn't technically a CoR implementation, and to be clear, this is my description of what it is, not the Mootools team's description. But this is basically what it does, and it's something unique to Mootools as far as I am aware. For more information on the CoR pattern in general, see `http://en.wikipedia.org/wiki/Chain-of-responsibility_pattern`.

PART 2

The Projects

The important thing is not to stop questioning.

Albert Einstein

The dumbest people I know are those who know it all.

Malcolm Forbes

Human beings, who are almost unique in having the ability to learn from the experience of others, are also remarkable for their apparent disinclination to do so.

Douglas Adams

A computer lets you make more mistakes faster than any invention in human history—with the possible exceptions of handguns and tequila.

Mitch Ratliffe

Creativity is the sudden cessation of stupidity.

Edwin Land

Never trust a computer you can't throw out a window.

Steve Wozniak

CHAPTER 3

Hodgepodge: Building an Extensible JavaScript Library

No, we aren't talking about the rabbit[1] here. Programmers who have been coding for any length of time have invariably built up their own little private library of handy bits of code that they reuse from project to project. JavaScript is certainly no different in this regard. In this chapter, you'll put together such a library for yourself—a library you'll find numerous uses for in the projects to follow.

This chapter will take the form of a question-and-answer session between an imaginary junior developer and his senior mentor who is, shall we say, a bit on the eccentric side. Not only will you put together a batch of handy functions, but you'll also see how to organize it in a pseudo-package structure to avoid namespace collisions and make it easier to find what you want, and also to make it look a bit more like a "real" language, *à la* Java. Oh yeah, the tongue-in-cheek structure of this chapter is a bit fun, too! Now it's time to do a *Wayne's World*[2] flashback fade and have at it!

Bill the n00b Starts the Day

"Geez, 11:30 a.m., and they expect me to start coding right away? I hate this place."

Gilbert had been working at Initech[3] for all of five years (he was exceptionally loyal for a developer!), and he still couldn't quite get the hang of getting up "early." Gilbert rolled out of bed at around 10:45 a.m., showered (usually), and then hopped in the car for the 15-minute drive to work. Fortunately, he was a phenomenal developer, and everyone knew it, so he was cut perhaps more than his fair share of slack.

1. Hodge-Podge was the name of the rabbit in the comic strip *Bloom County* by Berke Breathed.
2. *Wayne's World* was a 1992 movie starring Mike Myers, Dana Carvey, and Tia Carrere, in which two slacker friends (Myers and Carvey) try to promote their public-access cable television show called "Wayne's World." As you might expect, madness way above and beyond that simple plot quickly ensues. At least one scene involves a flashback by one of the characters, which is preceded by a transition effect where the screen begins to wobble as the scene dissolves to the flashback scene (and Myers and Carvey accompany the waving with hilarious hand motions and sound effects). Still not ringing a bell? Take a cruise over to YouTube (`http://www.youtube.com`) and search for it; you're bound to find a clip in no time!
3. Initech is the name of the imaginary company where the main characters in the movie *Office Space* worked.

But today he was facing a challenge he had never encountered before—a horror so grave that he could barely comprehend how he would do it, let alone at this ungodly hour of the morning.

Today was Bill's third day of work. Bill was a n00b[4] of the highest order.

Hence, Gilbert was extremely unenthusiastic. Even by his standards.

"Uh, Gilbert, sir?" Bill asked timidly.

"What offering do you have for me today?" Gilbert replied.

"Two cans of Jolt cola, some gummy worms, and a jug of Dunkin' Donuts coffee."

Gilbert was pleased with his young apprentice. "You may approach and converse."

"Well, umm . . . Jack, the senior architect . . ."

"Do not speak the beast's name in my presence!" exclaimed Gilbert.

Bill had been afraid of this. He knew Gilbert didn't think too highly of people who sat around all day just writing "thesis papers," as he put it, and never actually twiddled bits anymore. Bill had indeed been prepared, though.

"I realize the beast is not on your level my master, but he is my superior, and I must therefore acquiesce to his will."

Gilbert considered his young apprentice for a moment. "That is true. He is superior to you. Few aren't. You may proceed."

"Well," Bill continued, "He asked me to add some functions to the online accounting system, and I have some questions I was hoping you could answer for me, as only you can." That last part, Bill knew, would score him some points with Gilbert. Indeed, he looked pleased.

"You may ask your questions, n00b, so that you may receive my wisdom."

And with that, Bill began.

Whoa, back to reality for just a minute. Yes, this is the voice of your author! I just wanted to make sure I haven't lost you here. What we'll do now is build up a library of JavaScript functions and also learn how to put it in something of a pseudo-package structure reminiscent of Java or C# packages. Many (but probably not all) of these functions will be used in later projects, and should give you a nice start on your own little library. You should absolutely add your own code to build up this library, and you'll find it to be a very handy tool in your future work!

Overall Code Organization

In this section, we will look at how to organize JavaScript in a clean, logical way that will also make it a little safer in terms of avoiding naming conflicts.

Question: How can I organize JavaScript code to avoid naming conflicts and generally group related functions together cleanly?

I want to make sure I organize my code well and that I don't risk naming anything I add in such a way that it conflicts with code that's already in the system.

Answer: In JavaScript, you can, to a certain degree, emulate the packaging system you use in Java, C#, or most other modern languages. All you need to do is create a new class, and then make all your utility functions members of that class.

For example, let's say you want to create a package that will contain a bunch of functions for displaying various predefined alert messages. You can do this by writing the following code:

4. n00b is short for newbie in Leetspeak (1337). A newbie is someone who is new at something. Leetspeak is a way of speaking, or more precisely, writing words, usually associated with the computer "underground" (software pirates, hackers, crackers, hardcore gamers, and so on).

```
jscript = function() { }
jscript.ui = function() { }
jscript.ui.alerts = new function() { }
jscript.ui.alerts.showErrorAlert = function() { }
```

What happens when this executes? Simply put, you will have an object named `jscript`, which is a reference to a function. Remember that in JavaScript, a function is an object, and you can therefore have a variable that references the object (similar to how in C you can have a pointer to a function). Within this object will be a member named `ui`, which is itself a function. Then within that object is another object named `alerts`, again a function. Finally, within that object is another object named `showErrorAlert`, which is once again a function.

We are essentially building up a hierarchy of objects, nested within the parent object `jscript`. This is the root of our package. Each subsequent line adds a member to that object, representing a new subpackage.

We can then reference any member (subpackage) of `jscript` or any object down in its hierarchy of child objects (in other words, functions or fields defined in a given subpackage). If you have ever worked in Java or C#, you will recognize that this gives us the appearance of packages. I say "appearance" because, in truth, you have a series of objects that you can actually instantiate individually. For instance, you can do this:

```
var v = new jscript.ui.alerts();
```

Clearly, you can't do that with a Java package, for instance, but you can do it here. There is really no way to stop this either, because each function is, in effect, the constructor of a class, not the mechanism by which an instance is created. In other words, it is tempting to try this:

```
jscript = new function() {
  return null;
}
var v = new jscript();
```

It might seem reasonable to suspect that the value of v would be null, but, in fact, that is not its value. This is because the function that `jscript` points to is returned, not the result of that function being executed.

The function that you are instantiating in this case *will* execute (remember that it's essentially a constructor), but as in Java, there is no return type, which is why returning null doesn't do what we might expect. However, this *does* allow us to do something like this:

```
jscript = new function() {
  alert("Do not instantiate me!");
}
```

Clearly, this isn't as good as making instantiation impossible in the first place, but it's better than nothing.

Now, moving on, what if you want to have a method that displays an alert when an error occurs, and you want it to be a part of this package? That's easy enough. You just write this:

```
jscript = function() { }
jscript.ui = function() { }
jscript.ui.alerts = new function() { }
jscript.ui.alerts.showErrorAlert = function() {
  alert("An error occurred");
}
```

Now, you can call this:

```
jscript.ui.alerts.showErrorAlert();
```

This will pop up the alert "An error occurred," just as you would expect.

Bill interrupted Gilbert's explanation at this point and asked, "But what if I want to have a class in that alerts package, just like I can do in Java, to display a specific message, for instance?" Gilbert smiled at the attentiveness of his pupil and replied, "Not a problem . . ."

```
jscript.ui.alerts.MessageDisplayer = function(inMsg) {
  this.msg = inMsg;
  this.toString = function() {
    return "msg=" + this.msg;
  }
}
var v = new jscript.ui.alerts.MessageDisplayer("Hello!");
alert(v);
```

This effectively creates a class named `MessageDisplayer` in the `jscript.ui.alerts` package. When the last two lines are executed, a new instance of `MessageDisplayer` will be created, and the string `"Hello!"` is passed to the constructor. Then when we call `alert()`, passing the variable that points to that instance, the `toString()` function is called, and we get the expected alert pop-up that says "msg=Hello!."

"That's pretty cool!" exclaimed Bill, clearly excited with his new knowledge. "Let me see if I can put it all together." Bill hacked away at the keyboard for a few moments and finally produced the code shown in Listing 3-1.

Listing 3-1. *A Complete Example of Pseudo-Packaging in JavaScript*

```
<html>
  <head>
    <title>JavaScript Packaging Example</title>
    <script>
      jscript = function() { }
      jscript.ui = function() { }
      jscript.ui.alerts = new function() { }
      jscript.ui.alerts.showErrorAlert = function() {
        alert("An error occurred");
      }
```

```
      jscript.ui.alerts.MessageDisplayer = function(inMsg) {
        this.msg = inMsg;
        this.toString = function() {
          return "msg=" + this.msg;
        }
      }
      function test() {
        jscript.ui.alerts.showErrorAlert()
        var v = new jscript.ui.alerts.MessageDisplayer ("Hello!!");
        alert(v);
      }
    </script>
  </head>
  <body>
    <input type="button" value="Test Alert"
      onclick="test();">
  </body>
</html>
```

Gilbert examined Bill's code, tried it out, and saw that it worked as expected. He was pleased.

"Only one thing could make this better," Gilbert said. "Do you have any idea what it might be?" Bill thought for a moment, and then suddenly realized what Gilbert was getting at. "Yes, something along the lines of import!" Bill exclaimed. "Exactly," Gilbert replied, "And you know what? It's not even that hard."

Let's say we want to have a package named `jscript.string`, and we want to be able to import this package separately from any other that may exist in `jscript`. We'll create a file like so:

```
if (typeof jscript == 'undefined') {
  jscript = function() { }
}

jscript.string = function() { }

jscript.string.sampleFunction = function(inMsg) {
  alert(inMsg);
}
```

Now, to "import" this into a page, we simply do this:

```
<script src="jscript.string.js"></script>
```

If this is the only import, then we wind up with a `jscript` object, which contains a string function, which finally contains the `sampleFunction` function, all in a hierarchy, forming our pseudo-package. Even better, if we have other packages under `jscript` and we import them, the `if` check seen here will ensure we always have only one copy of each package object. One last benefit: if we want to extend our packages later, say, add a `jscript.string.format` package, all we need to do is add a new `jscript.string.format.js` file, use the same check, and also add one to check whether `jscript.string` is defined and instantiate if it is not defined.

If we really wanted to go nuts, we could place each function, or each object, from every single package, in its own `.js` file. That way, just like in Java or C#, you can import only the specific classes you want (here, we are essentially equating stand-alone functions to classes). As it stands now, you basically can do only the equivalent of wildcard imports, which is probably sufficient, but I wanted to point out that you could have class-specific imports as well, if you wish.

Keep in mind, however, that unlike Java imports, which don't affect the size of the final class if you wind up not using something you imported, the size of the page the user downloads *will* be affected by what you import here, whether or not you use it. So, it is important to import only those packages you actually need. Remember, too, that it isn't only a matter of size. Each `.js` file imported will require another trip to the server to retrieve the resources (ignoring caching, which should, in many cases, eliminate the request, but not the first time, for sure!).

"You have more questions, my young apprentice?" Bill half expected Gilbert to pull out a light saber. He was acting even more bizarre than usual today. Bill did indeed have many more questions though, so he pressed on.

Creating the Packages

Now that we know how the packages will be structured, we're ready to begin building the JavaScript library itself. The library will contain a diverse collection of useful functions, many of which you'll use throughout this book. Let's get to it!

Building the jscript.array Package

In this section, we will write some code that will help us work with arrays, and start to create our first package, `jscript.array`.

Question: How can I copy the contents of one array into another?

Well, here's another scenario Jack is bringing up: when the application is first accessed, it builds up an array of categories, but the user can add to that array later. I'd like to take the array of entries by the user and append them to the existing array.

Answer: Are you kidding? Is that all? Take a peek at Listing 3-2.

Listing 3-2. *The copyArray() Function*

```
jscript.array.copyArray = function(inSrcArray, inDestArray) {

  var i;
  for (i = 0; i < inSrcArray.length; i++) {
    inDestArray.push(inSrcArray[i]);
  }
  return inDestArray;

} // End copyArray().
```

It's literally nothing more than looping through `inSrcArray`, and pushing each element into `inDestArray`. The result is that `inDestArray` will be expanded by *X* elements, where *X* is the length of `inSrcArray`, and will then include the contents of `inSrcArray`.

Question: How can I search for a specific element in an array?

Let's say we let the user enter a series of values on a page. It seems reasonable that we might want to put them into an array. What if later on we need to find if a specific value is present in the array? Can JavaScript do that for us intrinsically?

Answer: No, it can't. We'll have to throw some bits together to make that happen. Listing 3-3 shows those bits.

Listing 3-3. *The findInArray() Function*

```
jscript.array.findInArray = function(inArray, inValue) {

  var i;
  for (i = 0; i < inArray.length; i++) {
    if (inArray[i] == inValue) {
      return i;
    }
  }
  return -1;

} // End findInArray().
```

Just iterate over `inArray`, and check each element to see if it matches `inValue`, the value to find. If we find it, we'll return the index we found it at because, presumably, you may want to actually do something with the value once it is found. If it isn't found, we'll return –1, which is, at this point, almost the universal return value for "nope, not found" in any type of search.

Question: Assuming I have an array of numeric values, how can I calculate the average of all the elements in the array?

Jack is also asking me to add the capability to calculate an average of all the expense items the user has entered. Naturally, he has specified this happens on the client side. How?!?

Answer: You know how to calculate an average in general, right? Well, doing it on an array of numbers is the same thing, as shown in Listing 3-4.

Listing 3-4. *The arrayAverage() Function*

```
jscript.array.arrayAverage = function(inArray) {

  var accumulator = 0;
  var i;
  for (i = 0; i < inArray.length; i++) {
    accumulator += inArray[i];
  }
  return accumulator / inArray.length;

} // End arrayAverage().
```

Begin by iterating over `inArray`, and adding up all the values you find. Then divide that accumulated result by the length of `inArray`, and you have your average!

Building the jscript.browser Package

This section will begin a package of code that deals with the web browser as a whole, independent of any specific page that might be loaded at the time.

Question: How can I get identification information on the browser accessing the application?

Currently, the accounting application supports only IE. Jack would like me to rectify this, and I believe the first step is simply to be able to display some identification information about the browser client accessing the web site. How do I do that?

Answer: Check out Listing 3-5.

Listing 3-5. *The getBrowserIdentity() Function*

```
jscript.browser.getBrowserIdentity = function() {

  return navigator.appName + " " + navigator.appVersion;

} // End getBrowserIdentity().
```

This code will return a string consisting of the browser name and version, such as:

```
Microsoft Internet Explorer 4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; Maxthon;
WebCloner ; .NET CLR 1.1.4322; .NET CLR 2.0.50727; InfoPath.1)
```

The `Microsoft Internet Explorer` portion is the result of `navigator.appName()`. The rest is the result of `navigator.appVersion()`.

Building the jscript.datetime Package

This section deals with code that helps work with dates and times, and puts it all in a new package in our ever-growing package structure.

Question: How can I easily determine how many days are in a given month without having to remember that stupid poem?

I need to validate that a user-entered date is allowed for a given month. For example, if they enter 31, I need to be sure the entered month has 31 days.

Answer: Well, most kids learn a poem[5] to remember that:

30 days has September, April, June, and November

Teaching a computer to understand poetry might prove just a tad difficult, but fortunately, the "algorithm" to match this poem, such as it is, turns out to be very simple, as Listing 3-6 shows.

5. I could frankly never remember it, but here is what most people seem to have learned early in school: `http://www.kidport.com/Grade1/TAL/G1-TAL-Rhymes.htm`.

Listing 3-6. *The getNumberDaysInMonth() Function*

```
jscript.datetime.getNumberDaysInMonth = function(inMonth, inYear) {

  inMonth = inMonth - 1;
  var leap_year = this.isLeapYear(inYear);
  if (leap_year) {
    leap_year = 1;
  } else {
    leap_year = 0;
  }
  if (inMonth == 3 || inMonth == 5 || inMonth == 8 || inMonth == 10) {
    return 30;
  } else if (inMonth == 1) {
    return 28 + leap_year;
  } else {
    return 31;
  }

} // End getNumberDaysInMonth().
```

First, we need to determine if the specified `inYear` is a leap year. To do that, we'll be writing another function called `isLeapYear()` that does that check. We need to do this because we know that in a leap year, February has 29 days, and in nonleap years, it has only 28 days. Once we do that, we check to see if the `inMonth` is April (3), June (5), September (8), or November (10), and if it is, we return 30.

Note that the first step is to subtract 1 from the incoming month. The caller uses a value of 1 for January and 12 for December, as is most logical. But in order to make the leap year calculation easy, internally we subtract 1 so that January becomes 0 and December becomes 11. That's why the values for April, June, September, and November seem off by 1. If `inMonth` is February (1), then we return 28 plus 1 if it is a leap year, resulting in 29, or 28 plus 0 if it isn't a leap year, resulting in 28. If it is any other month, we return 31.

Question: How can I determine if a given year is a leap year?

It's funny you mention leap years, Gilbert, because Jack identified a flaw in the system when February is used in a leap year. So, I need to be able to determine if a specified year is a leap year to correct that flaw.

Answer: Here, we have to play with some math, but it's still relatively simple, as Listing 3-7 shows.

Listing 3-7. *The isLeapYear() Function*

```
jscript.datetime.isLeapYear = function(inYear) {

  if ((inYear % 4 == 0 && !(inYear % 100 == 0)) || inYear % 400 == 0) {
    return true;
  } else {
    return false;
  }

} // End isLeapYear().
```

The algorithm is basically this: if the year is evenly divisible by 4, and if it isn't evenly divisible by 100, or if it's evenly divisible by 400, then it's a leap year!

Building the jscript.debug Package

Our `jscript` package continues to grow in this section as we introduce a new package that will contain code to help us debug our JavaScript.

Question: How can I display all the properties, and their values, of an arbitrary object?

You know, Gilbert, I often find that when debugging JavaScript, I have some object, and it would be helpful to see its current state—all its properties and their values. I know I can use a debugger to do this, but sometimes a simple alert would be sufficient. Is there any way to do that?

Answer: Well, of course, there is, Bill! Listing 3-8 shows how.

Listing 3-8. *The enumProps() Function*

```
jscript.debug.enumProps = function(inObj) {

  var props = "";
  var i;
  for (i in inObj) {
    props += i + " = " + inObj[i] + "\n";
  }
  alert(props);

} // End enumProps().
```

We use the `for . . . in` loop style to iterate over the properties of `inObj`. For each, we add its name (the value of `i`) and its value (using array notation to access the member) to a string. At the end, we pass this string to `alert()`, and we've achieved your goal.

Question: How can I implement a somewhat robust logging mechanism, something similar to Jakarta Commons Logging, for instance?

I often find that I'd like to put logging messages in my code. But I'm not sure how to do it in JavaScript as I would in, say, Java with the Jakarta Commons Logging package, where I can create some object, a logger for example, and pass it messages to write to a log without having to know the details of the underlying logging implementation. Got any thoughts, Gilbert?

Answer: Well, if by "somewhat robust," you mean something that gives you the ability to log or not log messages based on a severity level, and not a whole lot more, then sure, we can do that. Check out Listing 3-9.

Listing 3-9. *The DivLogger Class Function*

```
jscript.debug.DivLogger = function() {


  /**
   * The following are faux constants that define the various levels a log
   * instance can be set to output.
   */
```

```
this.LEVEL_TRACE = 1;
this.LEVEL_DEBUG = 2;
this.LEVEL_INFO  = 3;
this.LEVEL_WARN  = 4;
this.LEVEL_ERROR = 5;
this.LEVEL_FATAL = 6;


/**
 * These are the font colors for each logging level.
 */
this.LEVEL_TRACE_COLOR = "a0a000";
this.LEVEL_DEBUG_COLOR = "64c864";
this.LEVEL_INFO_COLOR  = "000000";
this.LEVEL_WARN_COLOR  = "0000ff";
this.LEVEL_ERROR_COLOR = "ff8c00";
this.LEVEL_FATAL_COLOR = "ff0000";


/**
 * logLevel determines the minimum message level the instance will show.
 */
this.logLevel = 3;


/**
 * targetDIV is the DIV object to output to.
 */
this.targetDiv = null;


/**
 * This function is used to set the minimum level a log instance will show.
 *
 * @param inLevel One of the level constants.  Any message at this level
 *                or a higher level will be displayed, others will not.
 */
this.setLevel = function(inLevel) {

  this.logLevel = inLevel;

} // End setLevel().
```

```
/**
 * This function is used to set the target DIV that all messages are
 * written to.  Note that when you call this, the DIV's existing contents
 * are cleared out.
 *
 * @param inTargetDiv The DIV object that all messages are written to.
 */
this.setTargetDiv = function(inTargetDiv) {

  this.targetDiv = inTargetDiv;
  this.targetDiv.innerHTML = "";

} // End setTargetDiv().


/**
 * This function is called to determine if a particular message meets or
 * exceeds the current level of the log instance and should therefore be
 * logged.
 *
 * @param inLevel The level of the message being checked.
 */
this.shouldBeLogged = function(inLevel) {

  if (inLevel >= this.logLevel) {
    return true;
  } else {
    return false;
  }

} // End shouldBeLogged().


/**
 * This function logs messages at TRACE level.
 *
 * @param inMessage The message to log.
 */
this.trace = function(inMessage) {

  if (this.shouldBeLogged(this.LEVEL_TRACE) && this.targetDiv) {
    this.targetDiv.innerHTML +=
    "<div style='color:#" + this.LEVEL_TRACE_COLOR + ";'>" +
    "[TRACE] " + inMessage + "</div>";
  }
```

```
  } // End trace().


} // End DivLogger().
```

Note that after the `trace()` function would actually be a couple more: `debug()`, `info()`, `warn()`, `error()`, and `fatal()`—one for each logging level. I left them out just to save some space here, but they are essentially identical to the `trace()` method, except that `trace` is replaced with `debug`, `info`, `error`, or `fatal`, as appropriate.

To use this, you instantiate a `DivLogger`, like so:

```
var log = new jscript.debug.DivLogger();
```

The one other required thing for this particular logger is to call `setTargetDiv()`, passing it a reference to the `<div>` element to which all log output should be written. From then on out, you simply call `log.`*xxxx*`(`*yyyy*`)`, where *xxxx* is the severity (`trace`, `debug`, `info`, `warn`, `error`, or `fatal`) and *yyyy* is the message to log. You can also call `log.setLevel()` to set the level of messages to be logged. If, for instance, you did this:

```
log.setLevel(log.LEVEL_ERROR);
```

then from that point on, only messages of severity `LEVEL_ERROR` or `LEVEL_FATAL` would be logged.

This is a very simple logger that just appends a message to the target `<div>`, and it also color-codes the log messages to make it easier to see messages of a certain type while browsing the log output. You could easily write another implementation to make an Ajax call to write the message to a database on the server, or whatever you like. The basic skeleton would be the same, although you may or may not need the `targetDiv` stuff or the color-coding stuff.

Building the jscript.dom Package

In this section, we add a new package of functions that will aid us in manipulating the DOM.

Question: How can I center an arbitrary DOM element?

Gilbert, at present, the application shows a "please wait" pop-up when a form is being submitted. It's just a `<div>` with a z-index set to a high number so it is on top of everything else. Unfortunately, the contractor that wrote the code didn't know how to center the `<div>`, so it's always in the upper-left corner, and Jack isn't thrilled with this. How can I center it?

Answer: I have code lying around to do that. Interestingly, it was written a long time ago, as evidenced by the fact that it uses the term *layer*, which is an old Netscape term. That doesn't really matter, because it still works. Listing 3-10 shows how to center an element horizontally.

Listing 3-10. *The layerCenterH() Function*

```
jscript.dom.layerCenterH = function(inObj) {

  var lca;
  var lcb;
  var lcx;
  var iebody;
  var dsocleft;
```

```
  if (window.innerWidth) {
    lca = window.innerWidth;
  } else {
    lca = document.body.clientWidth;
  }
  lcb = inObj.offsetWidth;
  lcx = (Math.round(lca / 2)) - (Math.round(lcb / 2));
  iebody = (document.compatMode &&
    document.compatMode != "BackCompat") ?
    document.documentElement : document.body;
  dsocleft = document.all ? iebody.scrollLeft : window.pageXOffset;
  inObj.style.left = lcx + dsocleft + "px";

} // End layerCenterH().
```

This will actually center any element that has a `left` style property exposed, which is most elements, not just `<div>` elements. It works by first getting the size of the browser window—the area that the content actually fills. This is done differently, depending on which browser it is executing in: either by getting the `innerWidth` property of the window or by getting the body's `clientWidth` property. Next, we get the width of the object by getting its `offsetWidth` property. Then it's a simple bit of math to calculate what the *X* location should be to center the object (it's basically subtracting half the element's width from half the window's size).

There is one more bit of calculation to do, and that's taking the amount that the page may be horizontally scrolled into account. Once again, there is a different method to get this value depending on which browser is in use, and what's worse, there is a different method depending on which rendering mode IE is in! When rendering in compatibility mode but *not* `BackCompat` mode, we need to go after the `document.documentElement` element; otherwise, it's the `document.body` element. Next, we see if the `document.all` element is present in the DOM. If it is, then we're running in IE, and we request the `scrollLeft` property of the object we previously determined we needed. If `document.all` is not present, we're not running in IE, so we need the window's `pageXOffset` property. In either case, we now have the amount the page is scrolled horizontally, so we add that to the value we calculated to center the element, and *voilà*, we've got it centered on the page as it is currently displayed!

You probably noticed this would only center horizontally. What about vertically? Well, the code is nearly identical. It's just a matter of making some replacements: `width` with `height`, `style.left` with `style.top`, `scrollLeft` with `scrollTop`, and `pageXOffset` with `pageYOffset`. Listing 3-11 shows what this modified code would look like, and as I said, it's basically the same.

Listing 3-11. *The layerCenterV() Function*

```
jscript.dom.layerCenterV = function(inObj) {

  var lca;
  var lcb;
  var lcy;
  var iebody;
  var dsoctop;
```

```
  if (window.innerHeight) {
    lca = window.innerHeight;
  } else {
    lca = document.body.clientHeight;
  }
  lcb = inObj.offsetHeight;
  lcy = (Math.round(lca / 2)) - (Math.round(lcb / 2));
  iebody = (document.compatMode &&
    document.compatMode != "BackCompat") ?
    document.documentElement : document.body;
  dsoctop = document.all ? iebody.scrollTop : window.pageYOffset;
  inObj.style.top = lcy + dsoctop + "px";

} // End layerCenterV().
```

I made it two separate functions because you may sometimes want to center one way, but not the other. Now you have that ability.

■ **Note** In Listings 3-10 and 3-11, you may have noticed that I broke one of my own rules from Chapter 1: self-describing variable names. `lca`, `lcb`, `lcx`, `lcy`—what exactly do those mean? Well, you're right to slap my hand with a ruler next time you see me! However, what good are rules unless you can break them occasionally? The reason it's probably acceptable here is that we're talking about very short-lived variables whose context you can easily see on a single screen. Furthermore, these are used as intermediate parts of a calculation, so they don't have any sustained meaning. Think of them as you would loop counter variables, for which people frequently use just single letters, which generally doesn't bother anyone. So, while I certainly stand by the original rule, there are times where breaking it is OK. Just use your best judgment!

Question: When I make an Ajax request, I get a chunk of text back. If that chunk of text has some `<script>` blocks in it, how can I execute them?

I'm actually a bit amazed by this, given how backwards the application seems to be, but it is actually using Ajax for updating parts of a couple of screens. Unfortunately, Jack discovered that it isn't working fully (the consultants are at it again!) because the returned text from the server contains some `<script>` blocks that are not being executed. Is there a way to execute them?

Answer: Well, of course there is, Bill, and this is a somewhat common problem when doing Ajax. Take a gander at Listing 3-12 to see how it's done.

Listing 3-12. *The execScripts() Function*

```
jscript.dom.execScripts = function (inText) {

  var si = 0;
  while (true) {
    // Finding opening script tag.
    var ss = inText.indexOf("<" + "script" + ">", si);
```

```
    if (ss == -1) {
      return;
    }
    // Find closing script tag.
    var se = inText.indexOf("<" + "/" + "script" + ">", ss);
    if (se == -1) {
      return;
    }
    // Jump ahead 9 characters, after the closing script tag.
    si = se + 9;
    // Get the content in between and execute it.
    var sc = inText.substring(ss + 8, se);
    eval(sc);
  }

} // End execScripts().
```

I'm sure there are other ways to write this code—some fancy regular expression probably—but sometimes I like the simple things in life, so I went with the straightforward approach. We take in a string of text as `inText`, and we then begin to scan through it, looking for `<script>` blocks (and note that it literally must be `<script>`—`<script type="text/javascript">`, for instance, won't be detected. Do you sense a suggested enhancement?). If we find one, we then find its corresponding closing `</script>` tag. Once we have that, we take the substring in between those tags and `eval()` it. We then take care to set the location after the closing `</script>` tag we just found, and we do it again. At whatever point no more `<script>` tags are found, our work is done.

Question: How can I get a reference to an arbitrary number of DOM elements?

I know I can use `document.getElementById()` to get a reference to a DOM element—that's no problem. But if I want to get a reference to a batch of elements, it gets a little onerous to write all those calls. Is there a simpler way?

Answer: Sure, a relatively simple wrapper function can save you a lot of time and effort. Check out Listing 3-13.

Listing 3-13. *The getDOMElements() Function*

```
jscript.dom.getDOMElements = function() {

  if (arguments.length == 0) {
    return null;
  }
  if (arguments.length == 1) {
    return document.getElementById(arguments[0]);
  }
  var elems = new Array();
  for (var i = 0; i < arguments.length; i++) {
    elems.push(document.getElementById(arguments[i]));
  }
  return elems;

} // End getDOMElements().
```

This function will accept a variable number of arguments, which are presumed to be DOM element IDs. Remember that every JavaScript function inherently has reference to an `arguments` array, which is an array of all the arguments (parameters) that were passed into it. So, if no arguments are passed in, this function returns null, which is essentially an invalid call. If just one argument is passed in, we do the typical `document.getElementById()` and return it, and that's that. When the number of arguments is more than one though, that's when it gets fun! We loop through the `arguments` array, and for each, we do `document.getElementById()`. We push the return from that call onto a new array we created, and when we're finished, we return that array. Each element in the array is now a reference to one of the DOM IDs passed in. Then you can do whatever you want with that array, and you didn't have to write all the calls to `document.getElementById()` yourself!

Building the jscript.form Package

In the `jscript.form` package, which we are about to begin building, we will introduce some code that helps us work with HTML forms and form elements.

Question: How can I generate XML from an HTML form?

Jack has asked me to create a simple web service interface to the application. To test it, I'd like to take the HTML form on one page of the application, convert it to XML, and submit it to a specified URL. The only part I'm stuck on is the conversion to XML.

Answer: This is one of those tasks that kind of sounds like it should be difficult, but it's actually pretty easy. I've put the code in Listing 3-14 together to show you.

Listing 3-14. *The formToXML() Function*

```
jscript.form.formToXML = function(inForm, inRootElement) {

  if (inForm == null) {
    return null;
  }
  if (inRootElement == null) {
    return null;
  }
  var outXML = "<" + inRootElement + ">";
  var i;
  for (i = 0; i < inForm.length; i++) {
    var ofe = inForm[i];
    var ofeType = ofe.type.toUpperCase();
    var ofeName = ofe.name;
    var ofeValue = ofe.value;
    if (ofeType == "TEXT" || ofeType == "HIDDEN" ||
      ofeType == "PASSWORD" || ofeType == "SELECT-ONE" ||
      ofeType == "TEXTAREA") {
      outXML += "<" + ofeName + ">" + ofeValue + "</" + ofeName + ">"
    }
    if (ofeType == "RADIO" && ofe.checked == true) {
      outXML += "<" + ofeName + ">" + ofeValue + "</" + ofeName + ">"
    }
```

```
    if (ofeType == "CHECKBOX") {
      if (ofe.checked == true) {
        cbval = "true";
      } else {
        cbval = "false";
      }
      outXML = outXML + "<" + ofeName + ">" + cbval + "</" + ofeName + ">"

    }
    outXML += "";
  }
  outXML += "</" + inRootElement + ">";
  return outXML;

} // End formToXML().
```

Let's say we have the following HTML form:

```
<form>
  <input type="text" name="firstName"><br>
  <input type="hidden" name="lastName"><br>
  <input type="password" name="password"><br>
  <textarea name="notes"></textarea><br>
  <select name="gender">
    <option value="male">Male</option>
    <option value="female">Female</option>
  </select>
  <input type="radio" name="married" value="yes">Yes<br>
  <input type="radio" name="married" value="no">No<br>
  <input type="checkbox" name="haveKids">Check if you have kids</input><br>
</form>
```

This function will accept a reference to a form as `inForm`, and also a string `inRootElement`, which is the name of the root element of the XML document to create. After we check the input values to make sure they are good, we begin to build up the string `outXML` by first adding the root element to it.

Then we start iterating over the children of the form. For each, we get its type, name, and value. Next, we see what its type is. If it's a `text`, `hidden`, `password`, `select-one`, or `textarea` field, we simply add an element to our XML string with the name of the element as the tag, and then the value of the field in between the opening and closing tags. `select-one`, by the way, is the value that you see in the `type` attribute for a `<select>` field, when it's not multi-enabled (this code won't handle multiple selection fields—something for you to extend, I think!). For `radio` fields, we'll actually examine each of them, even if they are all in the same group. Of course, since only one can be selected at a time, this isn't a problem. The XML string is generated similarly as for the other element types. Finally, for `checkbox` fields, we will send the value `"true"` or `"false"`, depending on whether or not it is checked.

So, assuming we passed in `Person` as the root element, and assuming the entries in the form fields are Frank, Zammetti, myPassword, Hello, Male, and Yes, and haveKids is checked, the following XML would be generated:

```
<Person>
  <firstName>Frank</firstName>
  <lastName>Zammetti</lastName>
  <password>myPassword</password>
  <notes>Hello</notes>
  <gender>male</gender>
  <married>yes</married>
  <haveKids>true</haveKids>
</Person>
```

This is returned as a string, for the caller to handle as desired, such as submit via POST body to a web service endpoint, as you suggested, Bill.

Question: How can I find, and optionally select, a specified option in a `<select>` field?

When one of the pages of the accounting application is first shown, there is a `<select>` field with some options, and initially, one of those options will be selected, depending on various criteria. How can I select a given option? Also, what if I just wanted to find it and not select it?

Answer: You can simply take the brute-force approach, Bill. Also, although you didn't mention it, I think it would be handy to be able to determine if case will matter during the search. Listing 3-15 does all of this.

Listing 3-15. *The selectLocateOption() Function*

```
jscript.form.selectLocateOption = function(inSelect, inValue, inJustFind,
  inCaseInsensitive) {

  if (inSelect == null ||
    inValue == null || invalue == "" ||
    inCaseInsensitive == null ||
    inJustFind == null) {
    return;
  }
  if (inCaseInsensitive) {
    inValue = inValue.toLowerCase();
  }
  var found = false;
  var i;
  for (i = 0; (i < inSelect.length) && !found; i++) {
    var nextVal = inSelect.options[i].value;
    if (inCaseInsensitive) {
      nextVal = nextVal.toLowerCase();
    }
    if (nextVal == inValue) {
      found = true;
```

```
      if (!inJustFind) {
        inSelect.options[i].selected = true;
      }
    }
  }
  return found;

} // End selectLocateOption().
```

After the usual trivial rejections, we start iterating over the options in the specified <select> element as passed in as inSelect. For each option, we examine its value. If it matches inValue, which is the value we are looking for, then we set the found flag to true, which will then be the return value from the function. Also, we check the value of inCaseInsensitive. If its value is true, then the match will ignore case. If it is false, then the case must match exactly. Once we find the option, or run out of options to check, we see if the caller requested the option be selected via the value of the inJustFind parameter. If the value is true, then we don't have anything further to do, but if it is false, we need to select the option as well. Lastly, we return the value of found, which will be true if the option was found (regardless of whether it was selected or not) and false if not.

Question: How can I provide the ability to select all the options in a <select>?

Gilbert, Jack has asked me to give the users the ability to select all the options in a <select> on one screen, at one time. Is there an easy way to do it?

Answer: There is, indeed. Direct your gaze to Listing 3-16.

Listing 3-16. *The selectSelectAll() Function*

```
jscript.form.selectSelectAll = function(inSelect) {

  if (inSelect == null || !inSelect.options || inSelect.options.length == 0) {
    return;
  }
  var i;
  for (i = 0; i < inSelect.options.length; i++) {
    inSelect.options[i].selected = true;
  }

} // End selectSelectAll().
```

inSelect is a reference to the <select> you want to manipulate. Then you simply iterate over the collection of options it contains, and set selected on each one to true. That's really it!

I'm sure you'll also want a selectUnselectAll() function. Well, to do that, simply change the line:

```
inSelect.options[i].selected = true;
```

to:

```
inSelect.options[i].selected = false;
```

and you've done it!

You may want to consider combining it and making it one function, where you pass in a boolean parameter. But for now you can keep them separate—no real harm.

Building the jscript.lang Package

In this section, we'll build the `jscript.lang` package, which will contain code that helps us work with JavaScript at a fundamental language level.

Question: How can I take the properties of one object and copy them into another object?

In a couple of instances, I've had a JavaScript object that I basically want to combine with another object. In other words, I want to copy all the properties of one into another. Can you show me how to do that, Gilbert?

Answer: Ask and ye shall receive. Listing 3-17 is your blessing, Bill.

Listing 3-17. *The copyProperties() Function*

```
jscript.lang.copyProperties = function(inSrcObj, inDestObj, inOverride){

  var prop;
  for (prop in inSrcObj) {
    if (inOverride || !inDestObj[prop]) {
      inDestObj[prop] = inSrcObj[prop];
    }
  }
  return inDestObj;

} // End copyProperties().
```

Using the `for . . . in` loop style, we iterate over the properties of `inSrcObj`. For each property, we check to see if it already exists in `inDestObj`, and if it does, we see if the caller told us to override existing properties by passing `true` as the value of the `inOverride` parameter. If it exists and we are overriding, or if it didn't already exist, we use array notation to set the value of the property on `inDestObj`. This has the effect of adding the property if is wasn't there, or changing the value to that found in `inSrcObj` if the property already existed. We then return `inDestObj`, and our work here is complete.

Building the jscript.math Package

Now we'll create some code to help perform some mathematical functions (well, OK, just one actually), and we'll stick it in a new package, appropriately named `jscript.math`.

Question: How can I generate a random number in a specified range?

OK, Gilbert, I admit this one isn't something Jack asked me about because, after all, random numbers in an accounting system would probably be a bad thing! But I'm working on a little JavaScript game on the side, and I'd like to know how to generate a random number in a given range.

Answer: Ah, well, far be it for me to get in the way of your slackery, Bill! As you've probably discovered, random number generation in JavaScript doesn't give you this feature for free, so I'll show you how to do it. Please review Listing 3-18 now.

Listing 3-18. *The genRandomNumber() Function*

```
jscript.math.genRandomNumber = function(inMin, inMax) {

  if (inMin > inMax) {
    return 0;
  }
  return inMin + (inMax - inMin) * Math.random();

} // End genRandomNumber().
```

First, we do a quick trivial rejection: if the `inMin` value (the start of the range) is greater than `inMax` (the end of the range), then we just return zero, content in the knowledge that the caller did something stupid and we didn't blow up because of it! Once that's out of the way, we use the basic formula seen in the `return` statement, which will always result in a number in the specified range.

Building the jscript.page Package

The `jscript.page` package will contain code that deals with the current web page as a whole. Let's go build it!

Question: How can I programmatically initiate printing of the current page?

Jack asked me to add a Print button to the final report page. I know that the user can just click the browser's Print button, but he insists!

Answer: Well, you can call `window.print()` at any time, but that's only on more recent browsers. And also note that even with the function I'm about to show you, the user will still get the usual print dialog box. There's no way to simply initiate printing without user intervention (which is probably a good thing—imagine all the trees you could kill with the right hack!). So, a little wrapping function is in order. Check out Listing 3-19.

Listing 3-19. *The printPage() Function*

```
jscript.page.printPage = function() {

  if (parseInt(navigator.appVersion) >= 4) {
    window.print()
  }

} // End printPage().
```

We just do a quick version check to ensure the browser will support the `window.print()` call, and that's it. There's really not much to say about this one, Bill.

Question: How can I access parameters that were passed to a page?

You know, Gilbert, sometimes I try to prototype something locally using just plain HTML pages. If I want to submit a form, and I want to submit it to another HTML page, is there any way I can access the parameters?

Answer: Yes, Bill, there certainly is. Listing 3-20 shows how.

Listing 3-20. *The getParameter() Function*

```
jscript.page.getParameter = function(inParamName) {

  var retVal = null;
  var varvals = unescape(location.search.substring(1));
  if (varvals) {
    var search_array = varvals.split("&");
    var temp_array = new Array();
    var j = 0;
    var i = 0;
    for (i = 0; i < search_array.length; i++) {
      temp_array = search_array[i].split("=");
      var pName = temp_array[0];
      var pVal = temp_array[1];
      if (inParamName == null) {
        if (retVal == null) {
          retVal = new Array();
        }
        retVal[j] = pName;
        retVal[j + 1] = pVal;
        j = j + 2;
      } else {
        if (pName == inParamName) {
          retVal = pVal;
          break;
        }
      }
    }
  }
  return retVal;

} // End getParameters().
```

This function actually allows you to get a specific parameter by name, or an array of all parameters. If `inParamName` is passed in, then a parameter with the specified name will be returned (or null will be returned if it isn't found). If null is passed as the value of `inParamName`, then an array of all parameters will be returned.

`location.search.substring(1)` is the way we get a reference to the query string. By starting with the second character of the URL, which is what the `(1)` parameter does, we are removing the leading question mark, leaving just the parameters themselves. After that, we simply call `split()` on that string, splitting on the ampersand character (`&`) that separates each parameter, which gives us an array. Then that array is iterated over, and each element is further split on an equal sign, since each parameter is a name=value pair.

After that, it's a simple matter to see if a specific parameter was requested or all of them will be returned. In the latter case, we instantiate a new array the first time through, and add the parameter to the array, and the value after it, so the array winds up being in the form *name*,

value, *name*, *value*, and so on. Once we run out of parameters, we return the array. In the case of a specific parameter being requested, as soon as we find it, we return its value.

Question: How can I break out of a frameset via JavaScript?

Jack didn't ask for this, but I noticed it on my own. We have that home page where all our users start, and it has links to all of our applications, including this accounting application. Unfortunately, that home page is built with frames, and when you go to any application, you are still within that frameset. This isn't good most of the time, so I'd like to provide a way for the applications to break out of the frameset.

Answer: You know how many people truly hate frames these days? It seems like no self-respecting web developer uses frames anymore. I, however, do not suffer the opinions of fools and therefore I don't mind frames, when used properly. But I digress. To answer your question, I have provided Listing 3-21.

Listing 3-21. *The breakOutOfFrames() Function*

```
jscript.page.breakOutOfFrames = function() {

  if (self != top) {
    top.location = self.location;
  }

} // End breakOutOfFrames().
```

Breaking out of frames is a simple matter of making sure that the document in the browser is also the top, which means, if it were a frameset, it would be the parent frameset document. If the document isn't the top document, then we set the location of the top document to the location of the current document, which basically causes any frameset to be overwritten with the new document (as a result of a new retrieval from the server).

Building the jscript.storage Package

Client-side storage isn't really all that complex, but we could do with some utility functions to make it that much easier, and that's exactly what we'll put together now, in the `jscript.storage` package.

Question: How do I create a cookie and store it on the client?

Jack pointed out this one part of the application where we store user preferences on the server. He thinks, and I agree, that it would be more efficient to store it on the client. I know storage on the client is a bit limited with JavaScript, but cookies would seem to be a good fit here. How do I create one?

Answer: You're right, cookies are perfect for things like this: small bits of data stored per web site on the client. Let's make Cookie Monster happy and create a cookie, as shown in Listing 3-22.

Listing 3-22. *The setCookie() Function*

```
jscript.storage.setCookie = function(inName, inValue, inExpiry) {

  if (typeof inExpiry == "Date") {
    inExpiry = inExpiry.toGMTString();
  }
  document.cookie = inName + "=" + escape(inValue) + "; expires=" + inExpiry;

} // End setCookie().
```

Each cookie has a name and a value, obviously, as well as an expiration date. Therefore, this function accepts all three as `inName`, `inValue`, and `inExpiry`. The date must be in the form of a GMT date string, so we'll allow `inExpiry` to be either an actual `Date` object or a presumably properly formatted GMT date string. If it is a `Date` object, we call its `toGMTString()` method to get the proper format. After that, we set `document.cookie` equal to a string in the form of *xxxx*`=`*yyyy*`;`*expires*`=`*zzzz*, where *xxxx* is the name of the cookie, *yyyy* is the value, and *zzzz* is the date string. It might seem a little weird to you to be setting the property of document to a cookie—after all, wouldn't that mean that if you tried to set another cookie, the first cookie would be overwritten? But don't worry, because the browser properly deals with that. It's just a bit of syntactical weirdness, probably left over from the Netscape days.

Question: How can I get the value of a specified cookie?
Setting a cookie is easy! How about getting its value later?
Answer: That's also not difficult, Bill. Examine Listing 3-23.

Listing 3-23. *The getCookie() Function*

```
jscript.storage.getCookie = function(inName) {

  var docCookies = document.cookie;
  var cIndex = docCookies.indexOf(inName + "=");
  if (cIndex == -1) {
    return null;
  }
  cIndex = docCookies.indexOf("=", cIndex) + 1;
  var endStr = docCookies.indexOf(";", cIndex);
  if (endStr == -1) {
    endStr = docCookies.length;
  }
  return unescape(docCookies.substring(cIndex, endStr));

} // End getCookie().
```

When you retrieve the value of `document.cookie`, what you get is a giant string with all the cookies applicable for that page. So, the easiest way to find the cookie you are interested in is just to look for the substring *xxxx*`=`, where *xxxx* is the name of the cookie you want.

If we get back –1 from the call to `indexOf()`,`()`, then the cookie is not present, so we just return null. If it is found, then we need to find the end of it. Since all cookies will have the `;expires=`*zzzz* string after it, we can look for the semicolon. Once we have the start and end

location of the cookie we want, we return only the substring, making sure to `unescape()` it, since it is stored as a URL-encoded string, and the caller is happy.

Question: How can I delete a cookie?

OK, Gilbert, so I can create and retrieve cookies. Only one thing remains: how do I delete them?

Answer: Well, strictly speaking, you can't actually delete a cookie outright. However, you can retrieve the cookie, change its expiration date to something that has already passed, and set it again. That will overwrite the existing cookie, and the browser will immediately see that it has already expired, and will go ahead and delete it. See Listing 3-24 for the details.

Listing 3-24. *The deleteCookie() Function*

```
jscript.storage.deleteCookie = function(inName) {

  if (this.getCookie(inName)) {
    this.setCookie(inName, null, "Thu, 01-Jan-1970 00:00:01 GMT");
  }

} // End deleteCookie().
```

We use the `getCookie()` function we built earlier, and then pass what it returns along to `setCookie(),()`, as we saw earlier. We also pass in a string with an expiration date of January 1, 1970. Unless the system clock is pretty severely messed up, the cookie will be set, overwriting the one that's there already. It will then immediately expire and be deleted by the browser. A bit of trickery I suppose, but it works!

Building the jscript.string Package

Now we come to the final package we will build for our library, the `jscript.string` package. I'm sure you can guess its purpose: to help us work with strings!

Question: How can I count how many times a substring appears in a string?

One of the features Jack is requesting is the ability to check some free-form text that the user can enter and see how many times certain keywords appear, so we can flag the input as suspicious. How can I count how many times a given substring appears in a given string?

Answer: Not a big deal really. Check out Listing 3-25.

Listing 3-25. *The substrCount() Function*

```
jscript.string.substrCount = function(inStr, inSearchStr) {

  if (inStr == null || inStr == "" ||
    inSearchStr == null || inSearchStr == "") {
    return 0;
  }
  var splitChars = inStr.split(inSearchStr);
  return splitChars.length - 1;

} // End substrCount().
```

We can call this function, and pass it the string to search (`inStr`) and the string to search for (`inSearchStr`). First, the function does some trivial rejections to be sure the input parameters are valid. It will return zero if either is not. After that, it uses a handy method of the JavaScript `String` object called `split()`. This is essentially like `StringTokenizer` in Java. It splits the string into pieces, breaking it on a specified substring. So, in other words, if we have the string `Sally sells seashells by the seashore on this dreary day`, and we want to split it on the substring `ea`, it would break like this: `Sally sells seashells by the seashore on this dreary day`. The result is four pieces: `Sally sells s`, `shells by the s`, `shore on this dr`, and `ry day`.

What actually is returned by the call to `split()` is an array, which, in this case, contains four elements, as stated. So, all we need to do is return the length of the array minus one, and we have our answer. "Why minus one?" you ask. Think about splitting the string `XYZ` on `Y`. The length of the array would be 2, with `X` and `Z` as the elements. But we want to know how many times `Y` occurs, and that is always the length of the resultant array from the call to `split()`, minus 1! And if the element we're looking for isn't found, the array returned by `split()` still has a single element: the string we were trying to split. So, subtracting 1 from the length of that array still gives the correct answer: 0.

"That makes sense?" asked Gilbert. "Oh yes, completely!" replied Bill, "But I have more," he added.

Question: How can I strip certain characters from a string, or alternatively, strip any characters *except* certain ones from a string?

Jack also wants me to modify the code on the page where the user enters expense categories. It seems that, currently, the user can enter anything, even though only numbers are valid entries. I think I should write the function to be able to strip any characters that aren't in a list of allowed characters, as well as be able to strip only characters that appear in a disallowed list, just to be sure I cover all my bases for the future.

Answer: Neither of these goals is especially difficult. Both basically boil down to scanning through a source string and examining each character. If it matches any character from another string, then either copy it or don't copy it to a new string. You could write two separate functions to do this, but a single one should do the trick. Listing 3-26 shows the function that does both types of stripping.

Listing 3-26. *The stripChars() Function*

```
jscript.string.stripChars = function(inStr, inStripOrAllow, inCharList) {

  if (inStr == null || inStr == "" ||
    inCharList == null || inCharList == "" ||
    inStripOrAllow == null || inStripOrAllow == "") {
     return "";
  }
  inStripOrAllow = inStripOrAllow.toLowerCase();
  var outStr = "";
  var i;
  var j;
  var nextChar;
  var keepChar;
```

```
  for (i = 0; i < inStr.length; i++) {
    nextChar = inStr.substr(i, 1);
    if (inStripOrAllow == "allow") {
      keepChar = false;
    } else {
      keepChar = true;
    }
    for (j = 0; j < inCharList.length; j++) {
      checkChar = inCharList.substr(j, 1);
      if (inStripOrAllow == "allow" && nextChar == checkChar) {
        keepChar = true;
      }
      if (inStripOrAllow == "strip" && nextChar == checkChar) {
        keepChar = false;
      }
    }
    if (keepChar == true) {
      outStr = outStr + nextChar;
    }
  }
  return outStr;

} // End stripChars().
```

After a quick trivial rejection is done, just to be sure we have valid input values, we start scanning through the source string `inStr`. For each character, we scan through the list of allowed (or disallowed) values, which is also passed in as `inCharList`. For each, we check the value of the `inStripOrAllow` parameter, which is either `allow` or `strip`. If it's `allow`, and the character in `inStr` we are currently checking appears in `inCharList`, then we are keeping the character. If `inStripOrAllow` is `strip`, and the current character appears in `inCharList`, then in this case we are removing that character. To understand that, note the way `keepChar` is set before the inner loop begins.

When we are checking for allowed characters, the assumption is that the character will *not* be allowed *unless* it is found in the list. Conversely, when we are stripping characters, the assumption is that the character *will* be allowed *unless* it is found in the list. Finally, every character we keep in either case is added to `outStr`, which is returned at the end. So, that returned string will be less any characters as specified.

Question: What if I don't want to actually alter the string, but I just want to test if it contains only valid characters, or alternatively, contains any invalid characters?

`stripChars()` is indeed handy, but I'm thinking I may at some point just want to do a test to see if a given string contains only valid characters, or even just to see if it contains any invalid characters.

Answer: This is also not a big deal, and is quite similar to `stripChars()`, but I think we can do it a bit more efficiently, as shown in Listing 3-27.

Listing 3-27. *The strContentValid() Function*

```
jscript.string.strContentValid = function(inString, inCharList, inFromExcept) {

  if (inString == null || inCharList == null || inFromExcept == null ||
    inString == "" || inCharList == "") {
    return false;
  }
  inFromExcept = inFromExcept.toLowerCase();
  var i;
  if (inFromExcept == "from_list") {
    for (i = 0; i < inString.length; i++) {
      if (inCharList.indexOf(inString.charAt(i)) == -1) {
        return false;
      }
    }
    return true;
  }
  if (inFromExcept == "not_from_list") {
    for (i = 0; i < inString.length; i++) {
      if (inCharList.indexOf(inString.charAt(i)) != -1) {
        return false;
      }
    }
    return true;
  }

} // End strContentValid().
```

Again, we start with a trivial rejection, which is usually a good idea, by the way! After that, we again scan the input string `inString`. This time, though, our job is a little easier because we don't really need to go through the entire string. All we need to do is determine if the current character is not present in `inCharList` in the case of `inFromExcept` being `from_list`, and if it isn't, return `false`. In the case of `inFromExcept` being `not_from_list`, we check to be sure the current character *does not* appear in `inCharList`, and if it does, we again return `false`. If we make it all the way through `inString`, we return `true`.

Question: How can I replace *all* occurrences of a substring in a string?

I know that the `String` object in JavaScript has a `replace()` method that lets you replace a substring in a string with another substring. However, what if I want to replace *all* occurrences of a substring in a string?

Answer: You are observant to notice this shortcoming of the built-in `replace()` method, Bill. Fortunately, handling all occurrences, even though it requires some work on our part, isn't a big deal either. Listing 3-28 shows how to do it.

Listing 3-28. *The replace() Function*

```
jscript.string.replace = function(inSrc, inOld, inNew) {

  if (inSrc == null || inSrc == "" || inOld == null || inOld == "" ||
    inNew == null || inNew == "") {
    return "";
  }
  while (inSrc.indexOf(inOld) > -1) {
    inSrc = inSrc.replace(inOld, inNew);
  }
  return inSrc;

} // End replace().
```

Yes, that's really it! It's a simple matter of looping, looking for `inOld` in `inSrc`, and each time, replacing it with `inNew`. Just keep doing this in a loop until `inOld` doesn't appear in `inSrc` anymore, and we're finished. Couldn't be easier!

Question: How can I trim spaces from the start of a string?

I notice that in most other languages, the string class has methods to trim spaces from the beginning of a string, usually `leftTrim()` or something like that. JavaScript doesn't. How can I do that?

Answer: Oh, come on now, Bill, can't you challenge me a bit more? Listing 3-29 is your answer.

Listing 3-29. *The leftTrim() Function*

```
jscript.string.leftTrim = function(inStr) {

  if (inStr == null || inStr == "") {
    return null;
  }
  var j;
  for (j = 0; inStr.charAt(j) == " "; j++) { }
  return inStr.substring(j, inStr.length);

} // End leftTrim().
```

It's just a matter of finding where the first nonspace character is in the string, and we do that by iterating over the characters in `inStr` as long as we encounter a space. Then we just use the built-in `substring()` function and return the string starting from the value of the loop variable at the end (which, remember, is the first nonspace character) until the end of the string.

Oh yeah, and before you ask, you can easily make a `rightTrim()` function, too, as shown in Listing 3-30.

Listing 3-30. *The rightTrim() Function*

```
jscript.string.rightTrim = function(inStr) {

  if (inStr == null || inStr == "") {
    return null;
  }
  var j;
  for (j = inStr.length - 1; inStr.charAt(j) == " "; j--) { }
  return inStr.substring(0, j + 1);

} // End rightTrim().
```

It's the same basic logic, except that, this time, we iterate backwards over the string, since we're looking for the *last* nonspace character this time. Then we just return the substring starting from the beginning of the string until that last nonspace character.

I know what you're going to ask next: what about trimming both at the same time? Certainly, you could just call both of these on the source string, but why not make it a little more convenient? Listing 3-31 shows the `fullTrim()` method, which provides that convenience.

Listing 3-31. *The fullTrim() Function*

```
jscript.string.fullTrim = function(inStr) {

  if (inStr == null || inStr == "") {
    return "";
  }
  inStr = this.leftTrim(inStr);
  inStr = this.rightTrim(inStr);
  return inStr;

} // End fullTrim().
```

Might as well use what we've already developed, right? So, just call `leftTrim()` and then `rightTrim()` on the input `inStr`, and we're good to go.

Question: How can I take a string and break it into pieces of a specified length?

One last thing with regard to strings, Gilbert. We have a free-form text entry area for notes about an expense, and Jack wants me to store it in a number of database fields, each of which is 100 characters long. How can I break up what the user enters into 100-character chunks?

Answer: While breaking up a string like that isn't hard, one thing you didn't mention was making sure you don't break up the string in the middle of words. Certainly, Jack wouldn't like that, right? So, we have to take that into account. Listing 3-32 shows one of probably many ways you can pull this off.

Listing 3-32. *The breakLine() Function*

```
jscript.string.breakLine = function(inText, inSize) {

  if (inText == null || inText == "" || inSize <= 0) {
    return inText;
  }
  if (inText.length <= inSize) {
    return inText;
  }
  var outArray = new Array();
  var str = inText;
  while (str.length > inSize) {
    var x = str.substring(0, inSize);
    var y = x.lastIndexOf(" ");
    var z = x.lastIndexOf("\n");
    if (z != -1) {
      y = z;
    }
    if (y == -1) {
      y = inSize;
    }
    outArray.push(str.substring(0, y));
    str = str.substring(y);
  }
  outArray.push(str);
  return outArray;

} // End breakLine().
```

First things first: make sure we have a string to break up, and also make sure the specified size is greater than or equal to 1, since anything else wouldn't make much sense. Also, we check to see if the size of `inText` is less than or equal to the specified size. If it is, we just return `inText`, and we're finished!

After those checks, we copy `inText` to a variable named, creatively enough, `str`, and we begin a loop that will continue until `str` is longer than the specified size. See, with each loop iteration, we're going to reduce `str`, so that eventually it will be shorter than `inSize`, and the loop will end. So, in the loop, we get a substring whose size equals `inSize`. We then find the last space and line break in the string. If either is found, we set the variable `y` to its location. If neither is found, `y` gets set to the size of the string. We then finally push the substring into our `outArray`, and cut `str` down by the chunk we just removed, and then the loop begins again. Finally, we return `outArray`, which contains `inText`, broken up into chunks of the appropriate size (or slightly smaller, depending on where the breaks wound up falling).

"Well," said Bill, "That was quite a ride! I learned a ton today. Thank you, Gilbert!"

"Eh, it's all in a day's work," came the smug reply from Gilbert. "I'll make a good c0dr out of you yet."

"Well, with all you've taught me here, I should be able to fulfill all of Jack's requests, and then some," said Bill. "I should be able to impress him greatly, and soon he'll give *me* that promotion that *you've* been after! Oh yeah, who's the C00l D00D with the M4d 5KiLL2 now, huh? I R0><0R, j00'r3 0\/\/|\|3D, ll4|\/|4!"

(For the secret decoder ring you'll need to understand that mess, visit `http://www.learnleetspeak.com`.)

Gilbert sat in stunned silence, looking at the grinning face of Bill, the realization of what just happened sinking in. Gilbert opened his mouth, trying to prepare a smart-ass comeback on the fly, but before he could utter a syllable, Bill turned and left the room, a new bounce in his step obvious to anyone who was looking.

Gilbert looked at his Dilbert mug, his wall of IT certifications, his 1:100 scale models of the Enterprise NCC-1701D and E, and his Darth Maul mask from last Halloween, and realized he would have to pack up all this stuff pretty soon.

And for the first time in the five years Gilbert worked at Initech, just as he was realizing his days were likely numbered, he was smug and arrogant no more!

Testing All the Pieces

OK, back to reality, this time to stay!

The source code presented in this book is available for download from the Source Code/Download area of the Apress web site (`http://www.apress.com`). One of the downloadable items for this chapter, which does not actually appear in the chapter itself, is a test HTML document that exercises all the functions we have built here. This also doubles as a form of documentation, since it shows basic usages of all the functions. I highly recommend grabbing that source code and taking a look to see how all of this fits together and works. Go ahead, Gilbert won't mind! In fact, you will really *need* the source in front of you in later chapters. If you grab everything now, you'll be able to move through the rest of the chapters smoothly.

Just as a quick example of what you'll see when you run this application, Figure 3-1 shows the tests available for the `jscript.array package`, and the pop-up shows the result of the `findInArray()` function specifically.

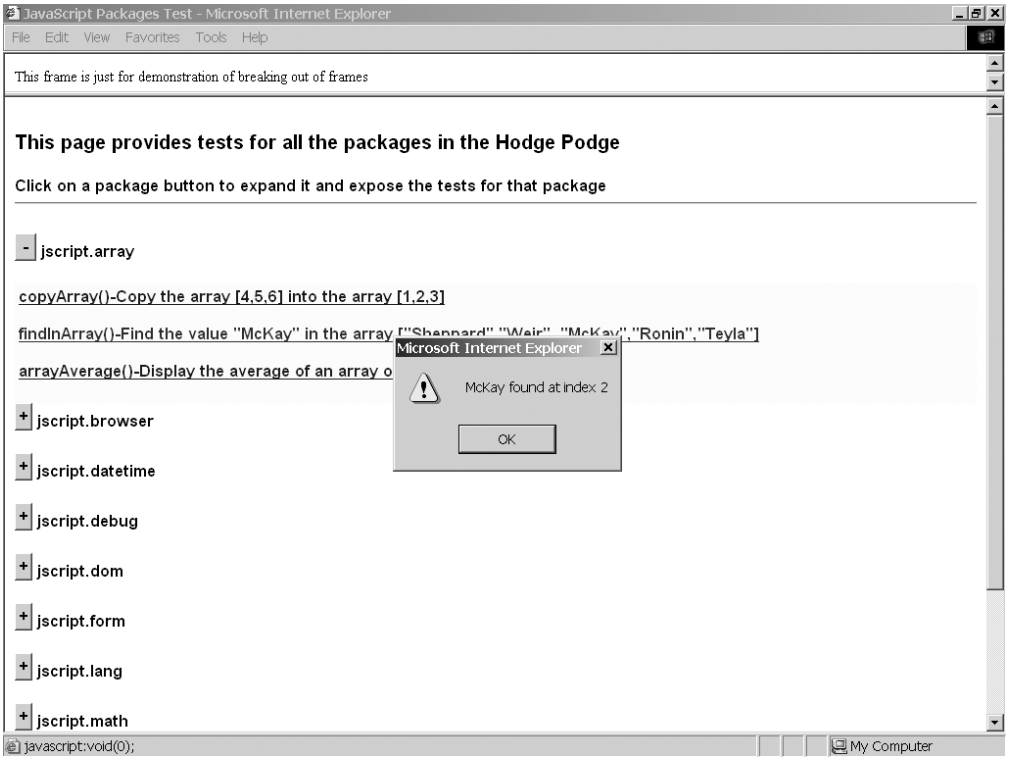


Figure 3-1. *Some tests for the jscript.array package, and the pop-up showing the result of testing the findInArray() function*

As another example to whet your appetite, Figure 3-2 shows the `jscript.debug` package test group, specifically, the `DivLogger` in action. (Unfortunately, you cannot discern the color-coding in the screenshot here, but trust me, each message is color-coded!)

One other important aspect that this test page demonstrates is how to use this as a library. All you actually need to do is have the various package source files that you intend to use available to your pages, and then "import," via the appropriate `<script>` tag, those packages. There are no packaging requirements aside from that—no building a DLL or anything along those lines. However, it is reasonable to set aside a directory for just these source files. Then when you want to include the library in another project, it's just a matter of copying that single directory.

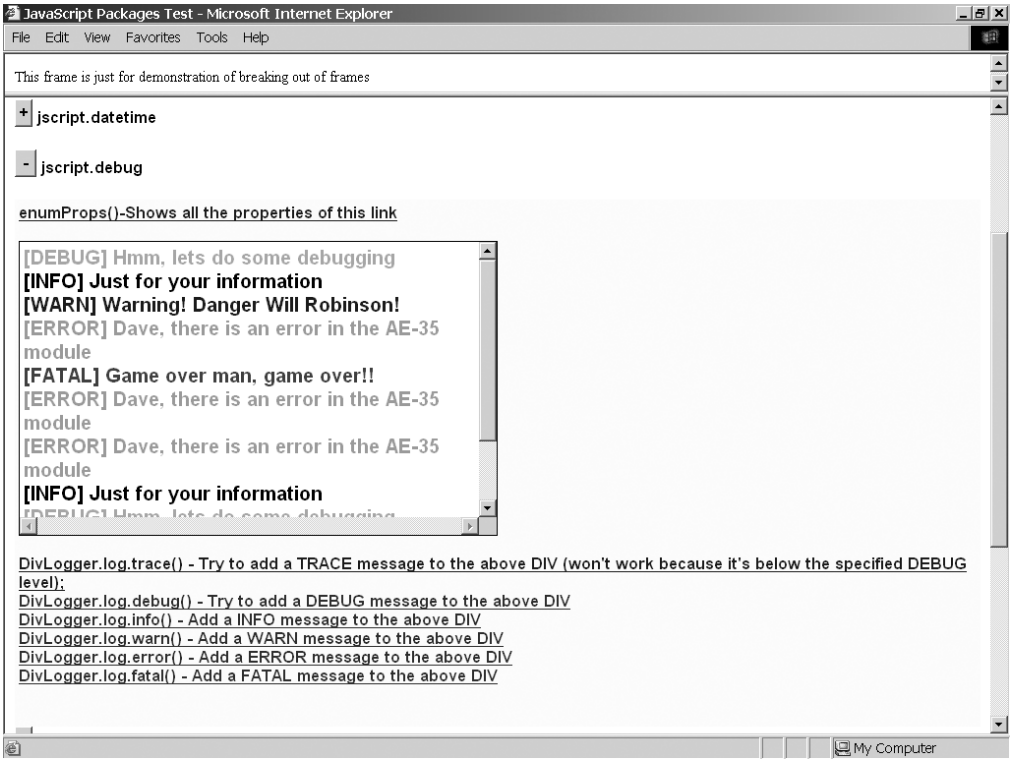


Figure 3-2. *The tests for the jscript.debug package, with some output from the DivLogger*

Suggested Exercises

A chapter like this makes is rather easy to suggest exercises because really one suggestion covers them all: go add some stuff! I suggest adding various functions to the existing packages—whatever you think will come in handy. I also suggest adding a package or two, just to see that it works. You may want to add a whole new package under `jscript`, as well as add a new subpackage to an existing package, maybe something like `jscript.dom.effects`, if you want to add some functions to do various effects. The possibilities are unlimited!

Summary

In this chapter, we put together a nice little library of functions that you should find a great deal of use for later. You have also seen how to create a rudimentary package structure that helps avoid namespace collisions and having a lot of global variables and functions all over the place.

More than likely, not all the functions will be used in this book's projects. (I wrote this chapter before writing the code for the projects to follow, so some of my guesses may have been off). That doesn't really matter though, because they are useful functions nonetheless and should serve you down the road.

CHAPTER 4

CalcTron 3000: A JavaScript Calculator

From Dustin Hoffman to Russell Crowe,[1] calculators have played an important role in the everyday lives of humans ever since the Babylonians first put stones on some lines in the sand (or was it the Chinese, as some pundits claim—I'm no historian, so I'll leave that debate to more qualified folks). Why not bring the idea into the modern age and build one in JavaScript for ourselves?

Along with the simple add, subtract, multiply, and divide functions, our calculator (dubbed CalcTron) will include some other common functions, such as percentages, square roots, and, since we're programmers, base conversions. Of course, those won't be quite enough to make a geek happy, so we'll make this a fully extensible calculator, to which we can add functions at will. We'll also do our best to make the interface a bit fancy, using some styles and cool effects. We can then see if adding enough features later allows it to gain sentience and take over the world, but one thing at a time!

Calculator Project Requirements and Goals

A calculator isn't fundamentally a complex project, as long as you don't try to include every bit of functionality possible. At the same time, it should be a good project to get some exposure to JavaScript concepts and make you think a bit. Let's throw some requirements out there that will help to fulfill that goal:

- CalcTron should present a relatively flexible interface that can morph as we add new features. Specifically, we'll allow CalcTron to be switched into a number of modes, each with its own defined layout (within some predefined constraints). Let's allow these layouts to be specified in JSON.
- A calculator isn't fundamentally the most visually exciting project, so to alleviate our boredom, we'll put some special effects and visual flair into it where possible. We'll do this with a library to save ourselves as much effort as possible.
- CalcTron should be extensible, allowing us to plug in new functions as required.

1. In the movie *Rain Man*, Dustin Hoffman played Raymond Babbitt, who was an autistic man with some startling mathematical abilities. In the movie *A Beautiful Mind*, Russell Crowe played John Nash, a brilliant mathematician.

That's a fairly short list, I admit. However, once we get into the code of things, you'll see that a project that seems minimal on the outside isn't necessarily that simplistic on the inside.

A Preview of CalcTron

We'll begin by having a look at CalcTron, and said look commences with Figure 4-1.

Figure 4-1. *CalcTron in Standard mode*

CalcTron provides two modes of operation out of the box: Standard mode, as shown in Figure 4-1, and BaseCalc (base calculations) mode, as shown in Figure 4-2.

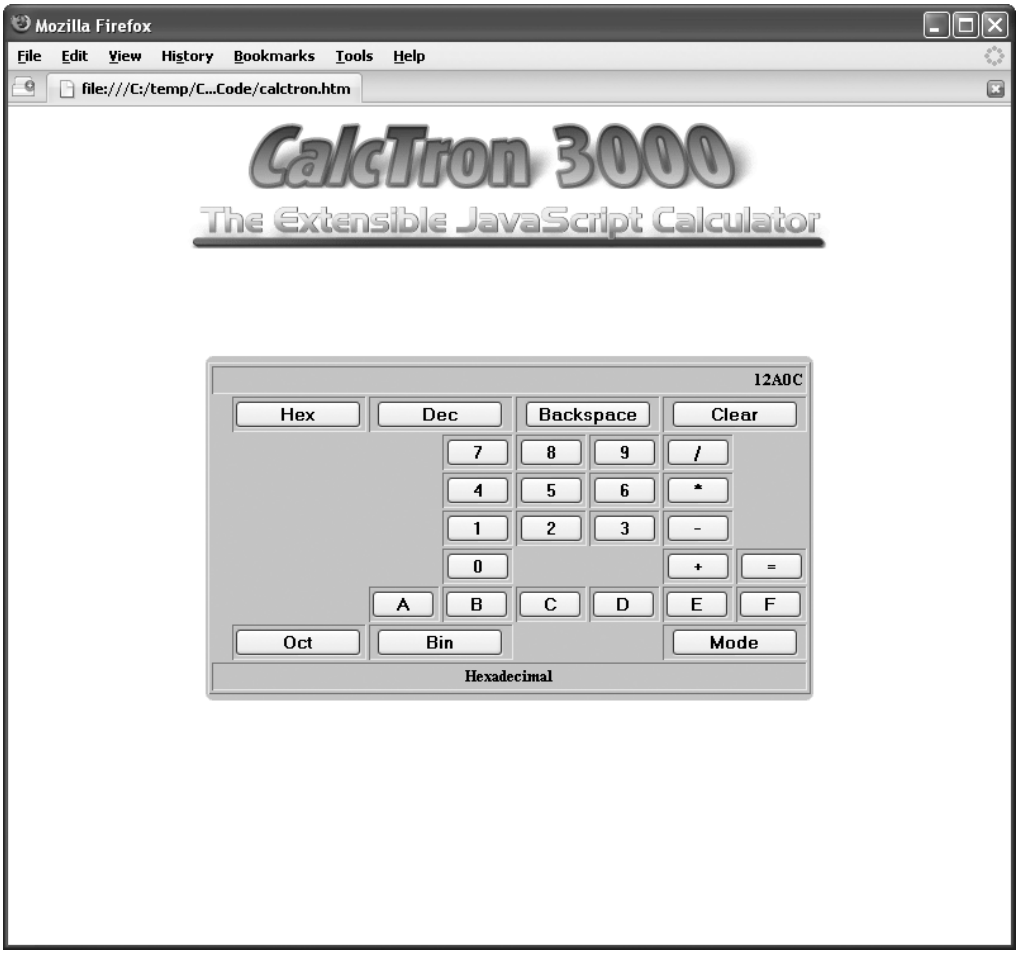


Figure 4-2. *CalcTron in BaseCalc mode*

When you click the Mode button, you are presented with a pop-up that flies onto the screen and allows you to choose a new mode of operation for the calculator, as shown in Figure 4-3. The pop-up is pretty simple, but it's an integral part of making CalcTron extensible.

What you cannot see in a screenshot is the fact that the pop-up flies to the center of the browser content area from one of the four corners randomly. This flying is accomplished with the help of a library named Rico. In addition to the flying pop-up, Rico does the rounding of the corners of the calculator itself. Let's take a quick peek at what Rico has to offer.

Figure 4-3. *Mode change pop-up*

Rico Features

I introduced Rico with some of the other representative JavaScript libraries in Chapter 2. Rico (`http://openrico.org`) is a smallish library (as compared to something like Dojo, for example) that covers relatively few topics, but does so quite well. Rico provides functionality in four key areas: Ajax, drag-and-drop management, cinematic effects, and behaviors.

Rico is one of the many libraries that is built on top of the Prototype library. Rico itself is housed in one relatively small (88kb) JavaScript file. Add the 46kb of Prototype (depending on version), and you can see it's not very big at all. As they say though, it packs a pretty good punch.

Rico provides two Ajax functions: one designed for updating the `innerHTML` of a target element, and one for updating multiple elements via an XML response. Both of these use an interesting model where you register a given Ajax request with the Ajax engine Rico provides and give it an ID. You can then reuse this request in different circumstances and at different times by referencing its ID. The latter expects an XML response from the server, and then uses

it to populate elements on the screen. If you jump over to the Rico web site and check out the demos, which you can get to by clicking the Demos link, you'll see some good examples of this (as well as everything else I'm describing here).

Rico also offers some nifty drag-and-drop support. In addition to being able to make arbitrary elements (`<div>` elements usually) draggable, it also allows you to define drop zones. This means that you can, for instance, have a `<div>` that is draggable, and have another `<div>` defined as a drop zone. When you drag the first `<div>` onto the second, it becomes a child of the drop zone. The Rico demos page shows this, as well as another example of a custom swap box that is drag-and-drop-enabled (that is, there is a list of items on the left, and you can drag those items into the list box on the right). Coding all this yourself would be a real hassle. Rico makes it very easy (a few lines of code in most cases).

In the area of cinematic effects, Rico offers functionality such as the ability to animate the position of an element, animate the size of an element, animate both the size and position of an element, fade elements in and out, and round shapes. Rounding shapes and animating size and position are two effects that you'll be seeing in action in CalcTron.

Another area of functionality Rico provides is called *behaviors*. Behaviors are what most other libraries call *widgets*; at least, that's true of what's in Rico now (it might not always be this way). Behaviors are generally combinations of cinematic effects and/or Ajax that create a unique component of functionality (a widget). Here are some of the behaviors Rico currently offers:

Accordion: Microsoft Outlook has a sidebar where you can click a category and have it expand into view. This is roughly what the accordion behavior is. Basically, you have a bunch of `<div>` elements running down the screen. You then specify that they form an accordion. When you click one of them, Rico will expand it, while shrinking any other that is showing. The effect is really quite impressive. Other libraries offer similar functionality, but I have to say, Rico's is the simplest, most straightforward, and cleanest looking implementation I've seen.

Weather: This is the typical show-me-the-weather-in-my-area widget. This makes use of some effects, the accordion behavior, and Ajax to make calls to a remote server to get the weather information. Once again, I highly recommend checking out the demos page on the Rico web site, because this is a really impressive behavior to see in action.

LiveGrid: This is the typical data grid with Ajax connectivity, buffering, and compression strategies to aid in performance. You've probably seen a number of different versions of this idea, and while Rico's is nice, it's not especially remarkable. It is definitely useful, but just OK in my opinion.

As you can see, Rico isn't about covering every last requirement a JavaScript and RIA developer might have. It's about a handful of targeted areas only, but it covers them rather well. The drag-and-drop support especially stands out to me as one of the best implementations I've seen. In Chapter 10's project, we'll use MochiKit for drag-and-drop support. While that library also offers pretty good drag-and-drop features, if I had to do it all over again, to be quite honest, I would choose Rico for this functionality. That's not meant as a slap against MochiKit at all. You'll see that it's a good library as well. I'm just emphasizing how good Rico's feature set is in this area.

I also feel that the accordion behavior is a real standout. I thought of ways I could shoehorn it into CalcTron, but decided not to force the issue.

■ **Tip** I've said it a couple of times now, but it's worth repeating: spend some time on the Rico demos page to see all of this in action. I think you'll really like what you see there.

Now that you've seen CalcTron and Rico, and understand some of our goals and expectations for this project (and I hope you've played with CalcTron a bit by now, too), we can begin our exploration of what makes it tick.

Dissecting the CalcTron Solution

To get a grasp on how CalcTron is put together, let's examine the directory structure of the application, as shown in Figure 4-4.

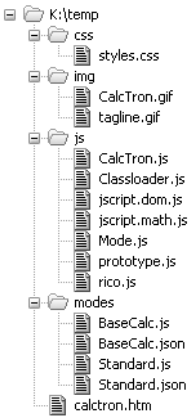


Figure 4-4. *Directory structure for the CalcTron project*

The solution consists of the following:

- `calctron.htm`: In the root directory is our starting point. The `calctron.htm` file defines the basic layout of the application, includes all the required JavaScript, and begins executing the application.
- `css`: This directory is home to the `styles.css` file, which is the style information CalcTron uses to define its display.
- `img`: This directory is where the images are stored. Only two images are used in CalcTron: the title graphic and the tag line underneath it.
- `js`: This directory contains a number of JavaScript files. These files are what literally make up CalcTron. Also found here are some support libraries that the application uses.
- `modes`: This directory houses a JavaScript file, as well as a JSON file, for each mode the calculator has.

Not really much to it, is there? With that brief overview, we can get down to brass tacks and dive straight into some code.

Writing calctron.htm

CalcTron begins by importing its style sheet, which we'll look at next, as well as all of the JavaScript source files it needs:

```
<link rel="StyleSheet" href="css/styles.css" type="text/css">

<script src="js/prototype.js" type="text/javascript"></script>
<script src="js/rico.js" type="text/javascript"></script>
<script src="js/jscript.math.js" type="text/javascript"></script>
<script src="js/Mode.js" type="text/javascript"></script>
<script src="js/Classloader.js" type="text/javascript"></script>
<script src="js/CalcTron.js" type="text/javascript"></script>
```

Prototype is imported first, and then Rico, which needs Prototype (we'll also be using a function from Prototype directly). Next is one of the packages we built in Chapter 3, the `jscript.math` package. The `math` package is needed because the code will be making a random determination of which corner the mode-change pop-up will fly in from, and the `math` package includes the random number generation function. After that is the import of the `Mode` class, the `Classloader` class, and the `CalcTron` class. We'll get to those shortly.

After these imports comes a single line of JavaScript that is actually key to making everything work:

```
<script>
  var calcTron = new CalcTron();
</script>
```

This single line of code creates an instance of the `CalcTron` class and assigns it to the variable `calcTron`. The `CalcTron` class is the core of the application, but let's not get ahead of ourselves; there's more to `calctron.htm` than this.

Since `calctron.htm` is the file the user loads, what happens on load is important:

```
<body onLoad="calcTron.init();">
```

The `init()` method of the `CalcTron` class is responsible for all application-level initialization, and hence is called in response to the page's `onLoad` event.

The body of `calctron.htm` is the basic structure of the page, as shown earlier in Figures 4-1, 4-2, and 4-3. The first element we encounter is a `<div>`:

```
<div id="divMode" class="cssDivMode">
  <br/>
  <center>
    Select mode here
    <br/><br/>
    <input type="button" value="Standard"
      onClick="calcTron.setMode('Standard');">
    <br/>
```

```
        <input type="button" value="BaseCalc"
          onClick="calcTron.setMode('BaseCalc');">
      </center>
    </div>
```

This is the mode-change pop-up. It's pretty straightforward; in fact, the onClick event handler is the only interesting thing about it. As discussed in previous chapters, you generally want to avoid in-line JavaScript like this. However, I don't view it as an egregious breach to do so when it is just a function call, as is the case here. As the name implies, the setMode() method of the calcTron object sets the mode as specified. We'll get into those details soon enough (in the "Writing CalcTron.js" section), so please allow this explanation to suffice for the time being.

After this is another one with the ID mainContainer, which is where the actual calculator structure is housed. Within it is a table, where each cell is one of the buttons of the calculator, preceded by the results, and followed by the information area at the bottom. This is a somewhat large chunk of frankly rather mundane HTML, so I won't list it all here. However, let's look at it in brief. First is the results section:

```
        <tr>
          <td nowrap colspan="10" align="right" valign="middle">
            <div style="height:16px;" id="divResults"></div>
          </td>
        </tr>
```

There's nothing unusual here. That being said, note that the inside the cell with the ID divResults is where the results (or the number the user is currently entering) are displayed by altering its innerHTML property.

Following this is the top row of command buttons (the calculator has five command buttons on top and five below, with the input buttons in between):

```
<tr>
  <td nowrap colspan="2" align="center" valign="middle"> ➥
    <input type="button" class="cssInputCommandButton" id="commandButton0" ➥
      onClick="calcTron.currentMode.commandButton0();"> ➥
  </td>
  <td nowrap colspan="2" align="center" valign="middle"> ➥
    <input type="button" class="cssInputCommandButton" id="commandButton1" ➥
    onClick="calcTron.currentMode.commandButton1();"> ➥
  </td>
  <td nowrap colspan="2" align="center" valign="middle"> ➥
    <input type="button" class="cssInputCommandButton" id="commandButton2" ➥
    onClick="calcTron.currentMode.commandButton2();"> ➥
  </td>
  <td nowrap colspan="2" align="center" valign="middle"> ➥
    <input type="button" class="cssInputCommandButton" id="commandButton3" ➥
    onClick="calcTron.currentMode.commandButton3();"> ➥
  </td>
```

```
  <td nowrap colspan="2" align="center" valign="middle"> ➥
    <input type="button" class="cssInputCommandButton" id="commandButton4" ➥
    onClick="calcTron.currentMode.commandButton4();"> ➥
  </td>
</tr>
```

I've broken the lines here so they'll fit on the page. In the actual code you execute, each of the cells is on a single line of code. Unfortunately, IE does not always ignore whitespace as it is supposed to, and so this code would not display properly if it were actually entered as it is here.

Once again, you see a single function call in the `onClick` event handler. Note that none of the buttons has a value (a label), because one is added dynamically when a calculator mode is selected, as you'll see in a bit. Also note that the bottom row of command buttons looks basically the same as this, except for the last button, which is always the mode-change button. It looks like this:

```
<td nowrap colspan="2" align="center" valign="middle">
  <input type="button" class="cssInputCommandButton" style="display:block;"
    value="Mode" onClick="calcTron.changeModePopup();">
</td>
```

As you can see, a value is given here, and the `onClick` handler is different. Otherwise, it's nothing special.

In between the top and bottom row of command buttons are five rows of input buttons. Let's take a look at a single row:

```
<tr>
  <td nowrap align="center" valign="middle">
    <input type="button" class="cssInputButton" id="button0_0"
      onClick="calcTron.currentMode.button0_0();">
  </td>
  <td nowrap align="center" valign="middle">
    <input type="button" class="cssInputButton" id="button0_1"
      onClick="calcTron.currentMode.button0_1();">
  </td>
  <td nowrap align="center" valign="middle">
    <input type="button" class="cssInputButton" id="button0_2"
      onClick="calcTron.currentMode.button0_2();">
  </td>
  <td nowrap align="center" valign="middle">
    <input type="button" class="cssInputButton" id="button0_3"
      onClick="calcTron.currentMode.button0_3();">
  </td>
  <td nowrap align="center" valign="middle">
    <input type="button" class="cssInputButton" id="button0_4"
      onClick="calcTron.currentMode.button0_4();">
  </td>
```

```
  <td nowrap align="center" valign="middle">
    <input type="button" class="cssInputButton" id="button0_5"
      onClick="calcTron.currentMode.button0_5();">
  </td>
  <td nowrap align="center" valign="middle">
    <input type="button" class="cssInputButton" id="button0_6"
      onClick="calcTron.currentMode.button0_6();">
  </td>
  <td nowrap align="center" valign="middle">
    <input type="button" class="cssInputButton" id="button0_7"
      onClick="calcTron.currentMode.button0_7();">
  </td>
  <td nowrap align="center" valign="middle">
    <input type="button" class="cssInputButton" id="button0_8"
      onClick="calcTron.currentMode.button0_8();">
  </td>
  <td nowrap align="center" valign="middle">
    <input type="button" class="cssInputButton" id="button0_9"
      onClick="calcTron.currentMode.button0_9();">
  </td>
</tr>
```

These are substantially similar to the command buttons; only the functions called in the `onClick` handlers differ, as well as the style class applied to them. The other four rows are the same, differing only in the `id` of each button.

Following the buttons is the information area:

```
<td nowrap colspan="10" align="center" valign="middle">
  <div style="height:16px;" id="divInfo"></div>
</td>
```

This is pretty much the same as the result area, with just a different style. And with that, we've examined `calctron.htm` pretty much in its entirety. I never said it was rocket science.

Writing styles.css

`styles.css` is, naturally enough, the main style sheet used by CalcTron. You've already seen a number of the styles it contains in our examination of `calctron.htm`, so let's see if there's anything else of interest. The entire style sheet is shown in Listing 4-1.

Listing 4-1. *The styles.css File for CalcTron*

```
/* Style applied to all elements */
* {
  font-family      : arial;
  font-size        : 10pt;
  font-weight      : bold;
}
```

```
/* Style applied to the outer calculator container */
.cssCalculatorOuter {
  position         : absolute;
  background-color : #c6c3de;
}

/* Style applied to the table cell where command buttons are placed */
.cssSpanCB {
  width            : 110px;
}

/* Style applied to the table cell where input buttons are placed */
.cssSpanB {
  width            : 60px;
}

/* Style applied to input buttons */
.cssInputButton {
  width            : 50px;
  display          : none;
}

/* Style applied to command buttons */
.cssInputCommandButton {
  width            : 100px;
  display          : none;
}

/* Style applied to the mode switch popup DIV */
.cssDivMode {
  display          : none;
  z-index          : 100;
  position         : absolute;
  border           : 2px solid #000000;
  background-color : #efefef;
}
```

Let's look at each selector in turn:

- The first selector is somewhat interesting. It applies to all elements on the page, and is kind of a catchall style. The nice thing about it is that it cascades down into tables and cells and such, which usually isn't the case, so it really does cover *everything*. This style sets the font to Arial, 10pt, and makes it all bold.
- The next style is applied to the outer `<div>`. We set it to a purple-blue color and position it absolutely, which is required so we can center it.

- Next, two styles are applied to the cells that contain the command buttons and input buttons, respectively. They ensure each cell is sized for a button to fit nicely, with a little padding.
- After that, two styles are applied to the command and input buttons. These ensure that all the buttons have a consistent size. They also ensure that all the buttons start out hidden, which avoids any unnecessary and ugly flickering when the application loads and the initial mode initiates.
- Finally, a style is applied to the mode-change pop-up. We set its `z-index` to ensure it floats over the calculator itself, and give it a solid color and a border. Like the outer `<div>` style, it is positioned absolutely, which is the only way we could make it fly in from a corner and center it.

Writing CalcTron.js

As mentioned earlier, `CalcTron` is the main class that powers this application—its core. It contains fields describing the overall state of the calculator, as well as initialization code, and the code that deals with switching modes. For all it does, and as central to the application as it is, it really isn't that big or complicated at all, as the UML diagram in Figure 4-5 illustrates.

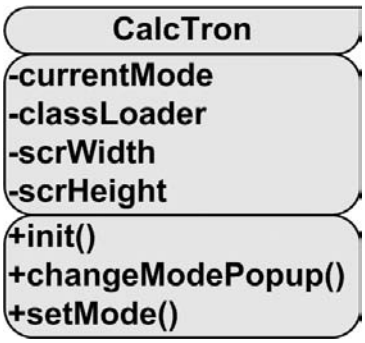


Figure 4-5. *UML diagram of the CalcTron class*

A mere four fields and three methods are all it takes. Let's begin by looking at those four fields:

- `currentMode`: This field stores the `id`, which is really the name, of CalcTron's current mode. As it stands, CalcTron has two modes: Standard and BaseCalc, and those are the only two values you would find in this field (not counting its initial null value).
- `classLoader`: This field is a reference to an instance of the `Classloader` class, which is described in the next section. In brief, this class is responsible for loading a class that contains the functions needed for a given mode. It also verifies the class meets certain interface requirements, which will generally mean the class is a valid CalcTron `Mode` class.
- `scrWidth` and `scrHeight`: These fields store the width and height of the browser window at startup. This information is used to center the calculator itself, and also for various calculations dealing with the mode-change pop-up.

Moving along to the methods of the `CalcTron` class, we first encounter `init()`. This is the method called by the `onLoad` of the `calctron.htm` file, and its task is to initialize a number of things to get CalcTron ready for user interaction. The `init()` function is as follows:

```
this.init = function() {

  // Figure out how wide the browser content area is.
  if (window.innerWidth) {
    this.scrWidth = window.innerWidth;
  } else {
    this.scrWidth = document.body.clientWidth;
  }

  // Figure out how high the browser content area is.
  if (window.innerHeight) {
    this.scrHeight = window.innerHeight;
  } else {
    this.scrHeight = document.body.clientHeight;
  }

  // Round the corners of the main content div.
  new Rico.Effect.Round(null, "cssCalculatorOuter");

  // Set initial mode to standard.
  this.setMode("Standard");

} // End init().
```

First, the function determines the width of the content area of the browser window. Some browsers present this information via the `innerWidth` attribute of the `window` object, while others present it via the `clientWidth` attribute of the `document.body` object, so a little branching action is in order, followed by virtually identical code for determining the height of the browser content area.

Following that is our first exposure to Rico in this application. As mentioned in the earlier look at Rico, one of the neat features it offers is the ability to round arbitrary elements, so we can have some nice, soft, round corners around our calculator instead of the usual sharp, square corners. The code instantiates a new `Rico.Effect.Round` object, passing it the name of a style sheet. The first argument to `Round()` is actually an element to round, and the second is a class name. I decided to use the class name because, initially, I wasn't sure if there might be other shapes to round, and using the class means that I can round any object that uses the same style class.

■ **Tip** You can pass a third argument to `Round()` to define the rounding further. For example, if for the third parameter you pass `{ corners : 'tl br' }`, you are saying that only the top-left and bottom-right corners are to be rounded. See the Rico documentation and examples for further explanations of what is possible with these options.

Once the rounding is done, one important piece of business remains, and that's to set the initial mode of CalcTron. This is done with the following statement:

```
this.setMode("Standard");
```

Standard is our starting mode, just as with the built-in Windows calculator (which CalcTron was roughly modeled after, at least as far as the Standard mode goes). Once that's complete, CalcTron is ready for the user.

The next method in CalcTron is the `changeModePopup()` method. This is called when the Mode button is clicked. Although it's a bit longer than `init()`, it's still a pretty simple animal:

```
this.changeModePopup = function() {

  // This is the width and height of the div as it should ultimately appear.
  var divWidth = 150;
  var divHeight = 150;

  // Get reference to mode change div and reset it to begin animation. It's
  // going to randomly come flying from one of the corners of the screen,
  // so first choose which corner, then set the top and left attributes
  // accordingly.
  var modeDiv = $("divMode");
  modeDiv.style.width = "0px";
  modeDiv.style.height = "0px";

  // What corner does it fly from?
  var whatCorner = jscript.math.genRandomNumber(1, 4)

  // Set the starting coordinates accordingly.
  switch (whatCorner) {
    case 1:
      modeDiv.style.left = "0px";
      modeDiv.style.top = "0px";
    break;
    case 2:
      modeDiv.style.left = this.scrWidth - divWidth;
      modeDiv.style.top = "0px";
    break;
    case 3:
      modeDiv.style.left = "0px";
      modeDiv.style.top = this.scrHeight - divHeight;
    break;
    case 4:
      modeDiv.style.left = this.scrWidth - divWidth;
      modeDiv.style.top = this.scrHeight - divHeight;
    break;
  }
```

```
    // Calculate the final left and top position for the div so it's centered
    // in the browser content area.
    var left = (this.scrWidth - divWidth) / 2;
    var top = (this.scrHeight - divHeight) / 2;

    // Show the div so the animation can begin.  Since its width and height are
    // zero, it won't actually be visible just yet.
    $("divMode").style.display = "block";

    // Ask Rico to do the animation for us.
    new Rico.Effect.SizeAndPosition("divMode", left, top, divWidth, divHeight,
      400, 25, null
    );

  } // End changeMode().
```

First, we have two variables, `divWidth` and `divHeight`, which define how wide and how tall the mode-change pop-up is. These are needed for calculations to come shortly.

We begin the real work by getting a reference to `divMode`. This is accomplished by using the `$()` function, which is actually part of Prototype, not Rico. `$()` is, in simplest terms, shorthand for the ubiquitous `document.getElementById()`, although it adds the ability to get a reference to multiple objects at the same time. Once we have a reference to the `<div>`, we set its width and height to zero. Not only does the mode-change pop-up fly in from one of the four corners of the browser content area, but it also grows as it flies in. So, it needs to start out as small as possible to make the effect work correctly.

Next, we see a call to the `jscript.math.genRandomNumber()` function introduced in Chapter 3. It generates a random number between 1 and 4, inclusive, which determines from which corner the pop-up will fly in.

After that, we see a `switch` on this value. Depending on which corner was chosen, the `left` and `top` attributes of `divMode` are set to start it out in the proper corner.

Following that are two lines that calculate the final location of the pop-up. These statements make use of the `scrWidth` and `scrHeight` we calculated in `init()`, as well as `divWidth` and `divHeight`, as set at the beginning of this function. Taking the difference between those two values and dividing by two, for both width and height, results in the proper coordinates required for the pop-up to be centered in the content area.

Finally, we see our next usage of Rico, the `Rico.Effect.SizeAndPosition` object. This object is fed the name of the `<div>` to manipulate (`divMode`), the final X and Y (`left` and `top`, respectively) location where it should end up, the width and height it should wind up, how many milliseconds the whole thing should take (400), and how many steps there should be (25). This means that Rico will take `divMode` and move it from its current location to the location specified by the `left` and `top` variables, and at the same time, will expand it from its current size to the size specified by `divWidth` and `divHeight`. It will do this over 25 steps in 400 milliseconds (which means that each step will take 16 milliseconds). Isn't it cool that we can get all that action from one function call?

The final method of CalcTron is `setMode()`, which is called when one of the buttons on the mode-change pop-up is clicked. It's actually an interesting little function because it gets called not once, but *twice* when modes are switched, and it does something a little different each time. Let's have a look at this method first:

```
this.setMode = function(inVal) {

  // First time through: should have been passed a string naming the mode
  // to switch to. We simply pass it to the classloader to load it for us.
  if (typeof inVal == "string") {

    $("divMode").style.display = "none";
    this.classloader.load(inVal);

  } else {

    // Second time through, inVal is an instance of a class descending from
    // Mode. In that case, we ask the classloader to verify it for us,
    // and assumimg it's valid, we store a reference to it and ask it to
    // initialize itself.
    if (this.classloader.verify(inVal, new Mode())) {
      this.currentMode = inVal;
      this.currentMode.init();
    } else {
      alert("Not a valid mode class");
    }

  }

} // End setMode().
```

OK, so it's fairly diminutive, but interesting nonetheless. The first time it's called, immediately when a mode-selection button is clicked, it's passed the name of the mode to switch to. So, the first check is if the incoming parameter is a string. If it is, the mode-change pop-up is hidden, and the `Classloader` instance is called with the value that was passed to `setMode()`.

The `Classloader` loads the class, and when it completes, `setMode()` is called again (you'll see exactly how in just a bit). However, at this point, what is passed in is *not* a string, but is instead a class descending from `Mode`. That's where the `else` clause comes into play. Here, we ask the `Classloader` to verify the class, and if it doesn't pass, we just pop up an alert, since there's not much we can do about it. If it is verified though, we set the `currentMode` field of CalcTron to point to this class that was passed in, and we then call the `init()` method on it.

And that sums up `CalcTron.js`. A couple of points probably are not quite clear to you yet, so let's commence clearing those things up now. We'll begin by looking at this `Classloader` class I've referenced a couple of times.

Writing Classloader.htm

As I've noted, CalcTron is designed to be extensible; that is, you can add modes to it with little difficulty. Each mode is implemented as a class extending the `Mode` class. We'll be looking at these implementation classes next, but before we do, let's talk about how those classes are loaded.

Java has a fairly complex mechanism called the *classloader*. Its job, as I'm sure you'll be unsurprised to learn, is to . . . wait for it . . . load classes. It does more than that, however. It is

also responsible for verifying classes, checking them for security violations, making sure they aren't corrupt, and so on.

JavaScript doesn't offer anything that fancy, but that has never stopped us from doing it ourselves. What would a JavaScript classloader offer? Well, security isn't really a concern, since JavaScript executes in a relatively secure sandbox to begin with, so we can skip that. Could we check for a corrupt class? Perhaps, but beyond ensuring we didn't get an error from the server the class is being loaded from, there isn't much to be done there either.

One operation we can perform is to verify that the class meets a certain public interface. Each of the mode implementation classes must implement certain functions for CalcTron to be able to use it. We can verify that a loaded class does so.

But perhaps we should discuss the idea of how exactly to load a JavaScript class in the first place, since even that simple concept isn't native to JavaScript. Before we even do *that*, however, let's take a look at the UML diagram for the `Classloader` class, as shown in Figure 4-6.

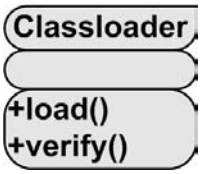


Figure 4-6. *UML diagram of the Classloader class*

Heck, we might as well look at the code, too. There isn't a whole lot to it, so the entire `Classloader` code is listed in Listing 4-2.

Listing 4-2. *The Classloader.js File*

```
function Classloader() {


  /**
   * Load a named class.
   *
   * @param inClassName The name of the class to load.  We assume it's a
   *                    calculator mode class, so it's always in the modes
   *                    subdirectory.
   */
  this.load = function(inClassName) {

    // Dynamically create a new script tag, point it at the mode source file,
    // and append it to the document's head section, thereby loading and
    // parsing it automatically.
    var scriptTag = document.createElement("script");
    scriptTag.src = "modes/" + inClassName + ".js";
    var headTag = document.getElementsByTagName("head").item(0);
    headTag.appendChild(scriptTag);
```

```
  } // End load().


  /**
   * This function verifies that a given class matches another. In other
   * words, it ensures that all the functions of inBaseClass are found in
   * inClass, which means they have the same public interface. It also
   * checks that the id field is present, which is required by code outside
   * a mode class (note that all other non-function fields are ignored, since
   * they do not contribute to the public interface).
   *
   * @param  inClass     The class to verify.
   * @param  inBaseClass The class to verify against.
   * @return             True if inClass is "valid", false if not.
   */
  this.verify = function(inClass, inBaseClass) {

    var isValid = true;
    for (i in inBaseClass) {
      if (i != "resultsCurrent" && i != "resultsPrevious" &&
        i != "resultsCurrentNegated" && i != "resultsPreviousNegated" &&
        !inClass[i]) {
        isValid = false;
      }
    }
    return isValid;

  } // End verify().


} // End Classloader class.
```

With those preliminaries out of the way, let's see how this all works.

The first step is a call to the `load()` method. This method accepts the name of the class to be loaded. This name must match exactly, including case, the name of the JavaScript source file that contains it. Since this `Classloader` is specific to loading `Mode` classes, it assumes the source file is found in the `modes` subdirectory. Here's the `load()` method:

```
  this.load = function(inClassName) {

    // Dynamically create a new script tag, point it at the mode source file,
    // and append it to the document's head section, thereby loading and
    // parsing it automatically.
    var scriptTag = document.createElement("script");
    scriptTag.src = "modes/" + inClassName + ".js";
    var headTag = document.getElementsByTagName("head").item(0);
    headTag.appendChild(scriptTag);

  } // End load().
```

LOADING REMOTE CONTENT VIA A DYNAMIC SCRIPT TAG

Tangent alert! One of the problems with most Ajax techniques is that they do not work across domains. All current browsers implement a security restriction that says you can make Ajax requests only to the domain that served the original page. This makes many sorts of things much more difficult than they need to be. A number of solutions exist to get around this restriction, and one of the most useful is the dynamic `<script>` tag trick.

Basically, if you create a new `<script>` tag and insert it into the DOM, the browser goes off and loads the source file and evaluates it, just as it does for a `<script>` tag on the page at load time. Since a `<script>` tag does not have a domain restriction, you can do cross-domain calls. Now, if the content that is the source of the `<script>` tag adheres to a special rule, you can essentially do Ajax in this manner. What is that special rule? Simply that the response must contain a JavaScript call to some callback function.

So, when the `<script>` tag is inserted, the browser loads the source file it specifies. This source file contains a call to some JavaScript function that already exists on the page. The browser evaluates this source file, which executes the call to that function. This is akin to the callback function in Ajax, except that it is called only once in this case, as opposed to many times for an Ajax call. In other words, the source of a `<script>` tag *doesn't have to be script*. Well, not in the usual sense anyway; it still is JavaScript.

What I mean by this is best illustrated with a simple example. Let's say we want to retrieve a list of URLs from a remote server that we want to display in an `alert()` box when the user clicks a button. Why we would want to do this is beyond me, but that's why they invented contrived examples. Anyway, let's say we want to do this. Let's further say that the file `js_book_ch4_url_list.txt` exists on the Omnytex web server (which it does, so yes, all of this will work if you try it, but if it should fail for some reason, just create a file with this name and the contents shown next, and place it on some web server somewhere, and change the target URL accordingly). So, when we access `http://www.omnytex.com/js_book_ch4_url_list.txt`, we get the following response:

```
showURLs("www.microsoft.com", "www.omnytex.com", "www.apress.com");
```

Here's the code to do what we want:

```
<html>
<head>
<script>
  function testIt() {
    var scriptTag = document.createElement("script");
    scriptTag.src = "http://www.omnytex.com/js_book_ch4_url_list.txt";
    var headTag = document.getElementsByTagName("head").item(0);
    headTag.appendChild(scriptTag);
  }
  function showURLs() {
    var s = "";
    for (var i = 0; i < showURLs.arguments.length; i++) {
      s += showURLs.arguments[i] + "\n";
    }
    alert(s);
  }
```

```
</script>
</head>
<body>
  <input type="button" value="Click here to display URLs" onClick="testIt();">
</body>
</html>
```

Let's walk through this. The user clicks the button, which calls `testIt()`. In `testIt()`, we create a new `<script>` tag, setting its `src` attribute to point to our remote URL file. This new `<script>` tag is appended to the document's `<head>` element. The browser then goes off and loads the remote file, and evaluates it. This evaluation causes the call to `showURLs()`, which is contained in the remote file, to execute. `showURLs()` executes, and it iterates over the collection of arguments passed to it, which can be none to as many as we want. For each, it appends it to a string, breaking the line after each one. Finally, it pops up an `alert()` showing the string. Blink and you missed it.

Now take another look at the `load()` method in the `Classloader.js` file. Doesn't that look familiar? It's the exact same code, but with a different URL for the `src` attribute.

Once a class has been loaded via the `load()` method, the `verify()` method is called. This method accepts the class that was loaded and another class that it is to be verified against. `verify()` iterates over the properties contained in the class to verify against, and for each, it ensures that the property is present in the class to be verified. If the property isn't present, `verify()` sets the `isValid` flag to `false`. This flag is the return value of the method, so `true` will be returned if all properties are found. Note that this method checks only for functions. It ignores data fields, except for `id`, which is a required part of the public interface of a `Mode`-descendant class. All the other data fields do not contribute to the public interface, so they are ignored.

It is often said that a picture is worth a thousand words, and in the spirit of that statement, Figure 4-7 is a flow diagram showing all the steps involved in loading a class in CalcTron.

This flow is slightly different for the BaseCalc mode. In that case, everything is the same up until the step "JSON includes call to `init()` of `Mode` object, JSON is passed to it." This still occurs, but in the case of the `BaseCalc` class, the `init()` method is overridden, so it does some work, then calls `init()` on the superclass (which, in case you are unfamiliar with the term, means the same thing as parent class), namely the `Mode` class. The rest of the flow after that is again the same. Don't worry if this didn't quite make sense. We haven't looked at the `Mode` class or the classes for the two calculator modes, so this is kind of jumping the gun just a bit. Let's get to that stuff now to put this all in the proper context.

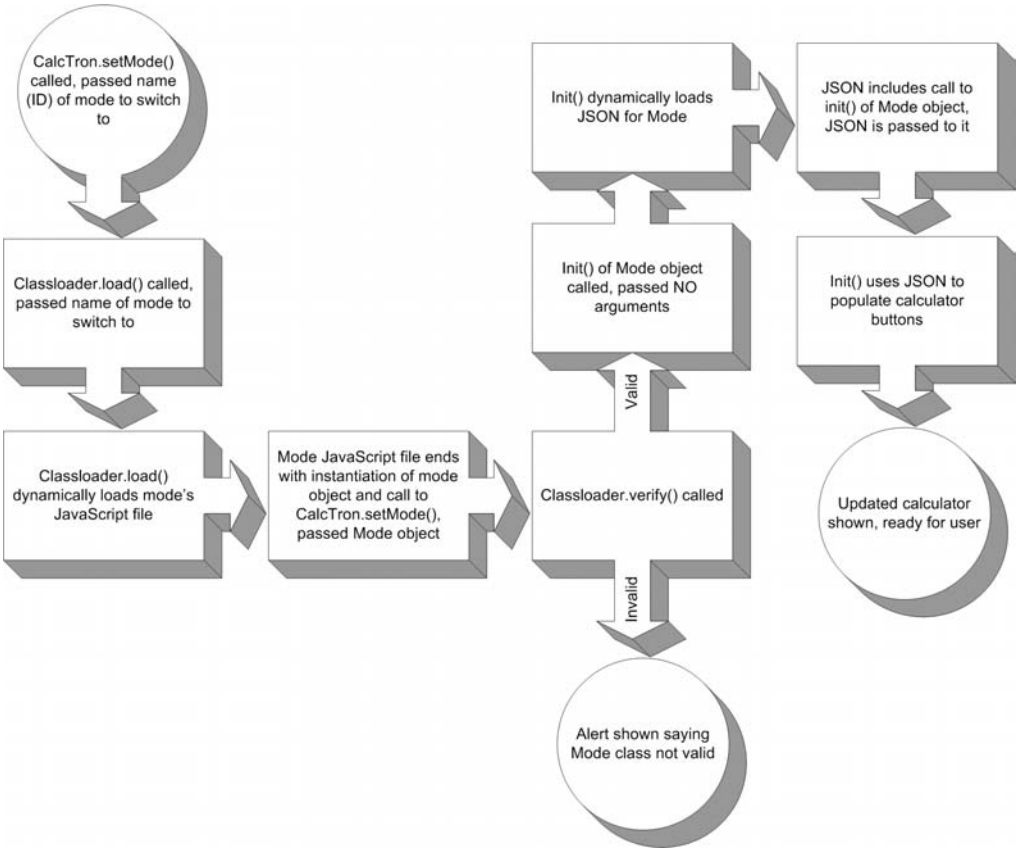


Figure 4-7. *Flow diagram of the steps involved in loading a class*

Writing Mode.js

Every mode that CalcTron supports, including the Standard and BaseCalc modes that come with it, is implemented in a class that extends from the `Mode` class. The `Mode` class itself contains some fields that are commonly needed by all calculator modes, and it also includes stub methods for all the methods any calculator mode is expected to implement. Together, these methods form the public interface a `Mode` implementation class must expose for CalcTron to be able to properly interact with it. You can see the overall structure of the `Mode` class in the UML diagram in Figure 4-8.

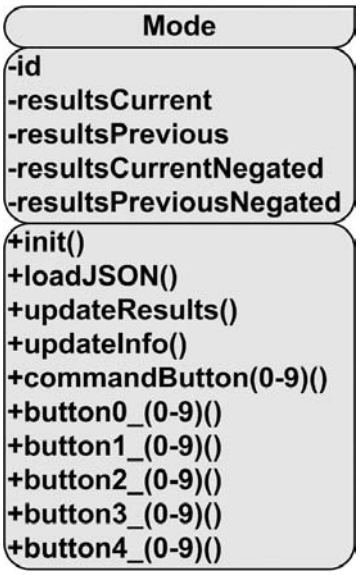


Figure 4-8. *UML diagram of the Mode class*

The first property, `id`, is actually part of the public interface as well, as this is needed by other code outside a class extending from `Mode`. However, the others—`resultsCurrent`, `resultsPrevious`, `resultsCurrentNegated`, and `resultsPreviousNegated`—are *not* considered part of the public interface, because they are needed only within the class itself.

The methods begin with `init()`, which as you saw during the discussion of the class-loading cycle, is called after the class is loaded and verified to initialize the calculator mode. Most of the time, the implementation of this method in the `Mode` class suffices, so let's look at that code now:

```
this.init = function(inVal) {

  if (inVal) {

    var mainDiv = $("mainContainer");

    // Size width and height as specified.
    mainDiv.style.width = inVal.mainWidth + "px";
    mainDiv.style.height = inVal.mainHeight + "px";

    // Center the main content div in the browser content area.
    mainDiv.style.left = (calcTron.scrWidth - parseInt(inVal.mainWidth)) / 2;
    mainDiv.style.top = (calcTron.scrHeight - parseInt(inVal.mainHeight)) / 2;

    // Command buttons (10 of them, numbered 0-8).
    for (var i = 0; i < 9; i++) {
      var btn = $("commandButton" + i);
```

```
      if (inVal.commandButtons[i].enabled == "true") {
        btn.style.display = "block";
        btn.value = inVal.commandButtons[i].caption;
      } else {
        btn.style.display = "none";
      }
    }

    // Buttons (50 of them, 10 in a row numbered 0-9 in 5 rows numbered 0-4).
    for (var y = 0; y < 5; y++) {
      for (var x = 0; x < 10; x++) {
        btn = $("button" + y + "_" + x);
        if (inVal.buttons[y][x].enabled == "true") {
          btn.style.display = "block";
          btn.value = inVal.buttons[y][x].caption;
        } else {
          btn.style.display = "none";
        }
      }
    }

  } else {

    this.loadJSON(this.id);

  }

  // Show current mode in info box and clear results div.
  this.updateResults("");
  this.updateInfo(this.id + " Mode");

} // End init().
```

As you may recall, this method is actually called twice during a mode switch. The first time it is called, nothing is passed to it. When that occurs, the `else` clause kicks in, which results in a call to `loadJSON()`, passing it the value of the `id` field, which would have been populated prior to this. `loadJSON()` is conceptually, and quite literally, just like the `load()` method of the `Classloader` class, as you can see here:

```
this.loadJSON = function(inID) {

  var scriptTag = document.createElement("script");
  scriptTag.src = "modes/" + this.id + ".json";
  var headTag = document.getElementsByTagName("head").item(0);
  headTag.appendChild(scriptTag);

} // End loadJSON().
```

A dynamic <script> tag is created, with its src attribute pointing to the JSON file in the modes subdirectory with the name matching the name (id field value) of the mode. This causes the JSON to be loaded and evaluated. When it is evaluated, init() will be called again, this time with the evaluated JSON itself as an argument. When that happens, the if block executes.

First, the if block code sets the width and height of the calculator outer to the values specified by the mainWidth and mainHeight properties in the JSON. This is done because dynamically calculating these values proves to be more difficult than you might at first think, if cross-browser display and proper display under all conditions are important to you. Instead, I decided to put the onus on the mode developer. There's little more to it than a bit of trial and error, but it's not that big a hassle when creating a new mode (I should know—I did it twice).

After that, the calculator is centered using the same basic logic you saw earlier to center the pop-up.

Next, we come to something slightly more interesting: generation of the command buttons.

```
// Command buttons (10 of them, numbered 0-8).
for (var i = 0; i < 9; i++) {
  var btn = $("commandButton" + i);
  if (inVal.commandButtons[i].enabled == "true") {
    btn.style.display = "block";
    btn.value = inVal.commandButtons[i].caption;
  } else {
    btn.style.display = "none";
  }
}
```

Ten command buttons are available in a mode layout: five on top and five at the bottom. The last one on the bottom is always the mode-switch button, so the developer has nine buttons available; hence, the 9 in the for loop. For each iteration, we first get the reference to the button, which, if you recall, already exists. We then check the enabled value in the JSON for the appropriate button. If it is false, then the button is not used, and we set the display style attribute to none to hide it. If it is enabled (true), then we set display to block instead to show it, and also change the label to that specified in the JSON. The event handler is already connected, as seen in calctron.htm, so that's all the work we have to do here.

The input buttons are handled in exactly the same way; however, the developer has a total of fifty different buttons available, ten each in five rows. So, we have two for loops: one for the row (y) and one for each button in the row (x).

Following loadJSON() is the method updateResults(). This simply displays the value of the resultsCurrent field in the results box at the top of the calculator, preceding the value with a negative sign if the resultsCurrentNegated field has a value of true. updateInfo() follows that, and it simply displays its argument in the info box at the bottom of the calculator.

After those two functions is a rather large batch of empty functions—one for each of the calculator's command buttons and input buttons. This is done so that if a calculator mode's implementation class doesn't need a given button, it still will expose an event handler function for it, but that function will do nothing. This just keeps the public interface a known constant, so there's no chance of a button's onClick handler calling a method that doesn't exist. (Well, a developer *could* set one of these methods to null in the implementation class, but he would have to go out of his way to do that, and we can't guard against a developer purposely trying to break something.)

As mentioned, the `Mode` class is the superclass for all the implementation classes for the calculator modes, so as you can probably surmise, it's now time to look at those implementation classes. We'll also look at the JSON that defines each mode as well, since they go hand in hand.

Writing Standard.json and Standard.js

Now that we've seen the basic `Mode` class, let's look at the classes that extend from it: the implementation classes that define a CalcTron mode. Let's begin with the Standard mode.

Two items make up a CalcTron mode: the implementation class extending from the `Mode` class and a JSON file, which describes the mode. For instance, the JSON for the standard mode is shown in Listing 4-3.

Listing 4-3. *The JSON Describing the Standard CalcTron Mode*

```
calcTron.currentMode.init(

  {

    "mainWidth" : "340", "mainHeight" : "248",

    "commandButtons" : [
      { "enabled"   : "false", "caption" : ""           },
      { "enabled"   : "false", "caption" : ""           },
      { "enabled"   : "false", "caption" : ""           },
      { "enabled"   : "true",  "caption" : "Backspace"  },
      { "enabled"   : "true",  "caption" : "Clear"      },
      { "enabled"   : "false", "caption" : ""           },
      { "enabled"   : "false", "caption" : ""           },
      { "enabled"   : "false", "caption" : ""           },
      { "enabled"   : "false", "caption" : ""           },
    ],

    "buttons" : [
      [
        { "enabled" : "false", "caption" : ""           },
        { "enabled" : "false", "caption" : ""           },
        { "enabled" : "false", "caption" : ""           },
        { "enabled" : "false", "caption" : ""           },
        { "enabled" : "false", "caption" : ""           },
        { "enabled" : "true",  "caption" : "7"          },
        { "enabled" : "true",  "caption" : "8"          },
        { "enabled" : "true",  "caption" : "9"          },
        { "enabled" : "true",  "caption" : "/"          },
        { "enabled" : "true",  "caption" : "sqrt"       }
      ],
```

```
[
  { "enabled" : "false",  "caption" : ""          },
  { "enabled" : "false",  "caption" : ""          },
  { "enabled" : "false",  "caption" : ""          },
  { "enabled" : "false",  "caption" : ""          },
  { "enabled" : "false",  "caption" : ""          },
  { "enabled" : "true",   "caption" : "4"         },
  { "enabled" : "true",   "caption" : "5"         },
  { "enabled" : "true",   "caption" : "6"         },
  { "enabled" : "true",   "caption" : "*"         },
  { "enabled" : "true",   "caption" : "%"         }
],
[
  { "enabled" : "false", "caption" : ""          },
  { "enabled" : "false", "caption" : ""          },
  { "enabled" : "false", "caption" : ""          },
  { "enabled" : "false", "caption" : ""          },
  { "enabled" : "false", "caption" : ""          },
  { "enabled" : "true",  "caption" : "1"         },
  { "enabled" : "true",  "caption" : "2"         },
  { "enabled" : "true",  "caption" : "3"         },
  { "enabled" : "true",  "caption" : "-"         },
  { "enabled" : "true",  "caption" : "1/x"       }
],
[
  { "enabled" : "false", "caption" : ""          },
  { "enabled" : "false", "caption" : ""          },
  { "enabled" : "false", "caption" : ""          },
  { "enabled" : "false", "caption" : ""          },
  { "enabled" : "false", "caption" : ""          },
  { "enabled" : "true",  "caption" : "0"         },
  { "enabled" : "true",  "caption" : "+/-"       },
  { "enabled" : "true",  "caption" : "."         },
  { "enabled" : "true",  "caption" : "+"         },
  { "enabled" : "true",  "caption" : "="         }
],
[
  { "enabled" : "false", "caption" : ""          },
  { "enabled" : "false", "caption" : ""          },
  { "enabled" : "false", "caption" : ""          },
  { "enabled" : "false", "caption" : ""          },
  { "enabled" : "false", "caption" : ""          },
  { "enabled" : "false", "caption" : ""          },
  { "enabled" : "false", "caption" : ""          },
  { "enabled" : "false", "caption" : ""          },
  { "enabled" : "false", "caption" : ""          },
  { "enabled" : "false", "caption" : ""          }
```

```
      ]
    ]

  }

);
```

If you've ever seen JSON before, you may be thinking, "That doesn't look quite right." Well, in fact, this is JSON wrapped in a function call. The JSON itself is the argument to the function, or more precisely, an object constructed from it. Recall that when the `init()` method of the `Mode` class is first called, by the `Classloader` class, it is passed nothing, but the second time, it is passed this JSON. So what actually calls `init()` the second time? The answer is that the JSON does. Well, sort of—the JSON is loaded dynamically using the previously discussed dynamic `<script>` tag technique.

Like any `<script>` tag on a page, the browser evaluates the contents returned from the server. In doing so, it executes any JavaScript not contained within a function, as is the case with the call to `calcTron.currentMode.init()`. During mode loading, the `currentMode` property of the `calcTron` instance is pointed to the `Standard` class loaded before the JSON. So when the JSON loads and is evaluated, the call to `init()` occurs, and the JSON is passed to it. `init()` then does its thing, as described previously.

■ **Tip** The dynamic `<script>` tag technique is becoming more and more common. Yahoo was the first to do it to a degree that people started to become aware of it. It's basically just a way to define a callback function that will be executed when the data returns. The data doesn't need to be JSON, although that is a prevalent return type. It could be an actual object (imagine a class being defined and an instance created instead of JSON, with fields populated, and then the call to `init()` passing a reference to that object). This is becoming a popular approach because it allows for cross-domain Ajax calls, in essence. As long as the client knows what callback function will be called, and the server adheres to that contract, there are no same-domain limitations, as is usually the case with Ajax. Just think, you and your friends can create CalcTron modes, host them on your own server, and your local copy of CalcTron can access them from anywhere in the world (once you inform CalcTron of these new modes by updating the mode-switch pop-up).

The meaning of the JSON itself is relatively simple. `mainWidth` and `mainHeight`, which were mentioned earlier, define the width and height of the calculator `<div>`. After that comes a group of elements, named `commandButtons`. Each element in this group defines a single command button. Every button, be it a command button or an input button, has two attributes: `enabled` and `caption`. The `enabled` attribute defines whether the button is visible (`true`) or not (`false`). The `caption` attribute is the text that appears on the button.

After the input buttons is a group named `buttons`. Within this group are five subgroups, each one corresponding to a row of buttons. Each of these groups contains ten elements, each corresponding to a button.

EVENT HANDLERS

In `calctron.htm`, each button has an `onClick` event handler defined. The `Mode` class includes stub functions, which map to the buttons. The correlation between these functions and the JSON you see here is an implied correlation. For instance, take the command button group:

```
"commandButtons" : [
  { "enabled"   : "false", "caption" : ""          },
  { "enabled"   : "false", "caption" : ""          },
  { "enabled"   : "false", "caption" : ""          },
  { "enabled"   : "true",  "caption" : "Backspace" },
  { "enabled"   : "true",  "caption" : "Clear"     },
  { "enabled"   : "false", "caption" : ""          },
  { "enabled"   : "false", "caption" : ""          },
  { "enabled"   : "false", "caption" : ""          },
  { "enabled"   : "false", "caption" : ""          },
]
```

How do we know which function will be called when the Backspace button is clicked? It isn't named here, so how do we know? Simply put, it's positional. The first button defined in this JSON would call `commandButton0()` when clicked, the second `commandButton1()`, the third `commandButton2()`, and finally, Backspace would call `commandButton3()`. For the input buttons, consider the first row as an example:

```
  [
    { "enabled" : "false", "caption" : ""          },
    { "enabled" : "false", "caption" : ""          },
    { "enabled" : "false", "caption" : ""          },
    { "enabled" : "false", "caption" : ""          },
    { "enabled" : "false", "caption" : ""          },
    { "enabled" : "true",  "caption" : "7"         },
    { "enabled" : "true",  "caption" : "8"         },
    { "enabled" : "true",  "caption" : "9"         },
    { "enabled" : "true",  "caption" : "/"         },
    { "enabled" : "true",  "caption" : "sqrt"      }
  ]
```

This works in exactly the same way, except that we need to know the row and column of the button in this case. For the first row of input buttons, the row number is 0. So, the first button here would call the function `button0_0()`. Jumping down to the number 7 button on the screen, it would call function `button0_5()`.

With the JSON out of the way, we can move on to the `Standard.js` file, the Standard mode implementation class itself. The UML diagram of the `Standard` class is shown in Figure 4-9.

In this diagram, I wrote the `commandButton` and `button` methods in shorthand because there are so many of them. The shorthand simply means that there are `commandButton0()`, `commandButton1()`, `commandButton2()`, and so on, up to `commandButton9()`. For the button methods, there are `button0_0()`, `button0_1()`, `button0_2()`, and so on, up to `button0_9()`, and then this repeats for `button1_x()`, `button2_x()`, `button3_x()`, and `button4_x()`, where *x* is 0–9.

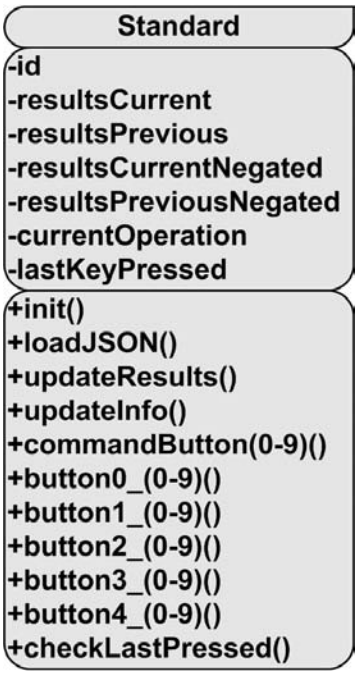


Figure 4-9. *UML diagram of the Standard class*

As you look at the source (which you should have already downloaded from the Apress web site), the first thing you see is the `id` field being set to `Standard`. Setting the `id` field is necessary for any implementation class you write, and this value must match exactly, including case, the `.js` and `.json` files for the mode. Next are two fields specific to this mode, `currentOperation` and `lastKeyPressed`. These will likely be needed by any mode you write as well, but I left them out of the `Mode` class because there could be some situations where they aren't required. Their names should make clear what they are: storage of what operation is currently being performed (the last operation button clicked, such as +, -, *, or /) and storage of what the last button clicked was (this is needed for proper operation, as you'll see).

I want to jump around just a little to cover the methods in a slightly more logical order than they appear in the code. First, let's look at the method `commandButton3()`, which correlates to the Backspace button:

```
this.commandButton3 = function() {

  if (this.resultsCurrent != "") {
    this.resultsCurrent =
      this.resultsCurrent.substr(0, this.resultsCurrent.length - 1);
    this.updateResults();
  }

} // End commandButton3().
```

It's admittedly not a complex function. After we're sure we have a current result (because if we don't, there's nothing to delete and hence we would get an error if we tried), we cut the last character off from the current result string, and call `updateResults()` to show it.

This is probably a good time to answer a question that has probably already occurred to you: why exactly are the `resultsCurrent` and `resultsPrevious` fields always strings? The answer is that it makes backspacing easier. It also makes showing negative numbers easier, and makes base conversions, which are done in the BaseCalc mode, easier. Leaving these values as strings until calculations are actually performed on them just makes the code a little cleaner and less verbose.

The Clear button is also a very simple bit of code:

```
this.commandButton4 = function() {

  this.resultsCurrent = "";
  this.updateResults();

} // End commandButton4().
```

I'm willing to bet you don't even need an explanation of that, so let's just keep moving right along.

Each of the number button handlers is pretty much the same, so let's look at just one:

```
this.button0_5 = function() {

  this.checkLastPressed();
  this.lastKeyPressed = "7";
  this.resultsCurrent += "7";
  this.updateResults();

} // End button0_5().
```

As discussed, by pure positional reckoning, the number 7 button winds up mapping to the `button0_5()` method. In it, we first call the `checkLastPressed()` method, which we'll look at next. For now, let's skip over it. We record 7 as being the last button pressed, and then add the digit 7 to the current results. We then redisplay the current results. The net result is that if 31 were showing in the results box, it would now show 317, and we would know that the last button pressed was 7.

Now let's see why knowing which button was last pressed is important by looking at that `checkLastPressed()` method:

```
this.checkLastPressed = function() {

  if (this.lastKeyPressed == "+" || this.lastKeyPressed == "-" ||
    this.lastKeyPressed == "*" || this.lastKeyPressed == "/") {
    // Time to start entering a new number, but save the current one first.
    this.resultsPrevious = this.resultsCurrent;
    this.resultsPreviousNegated = this.resultsCurrentNegated;
    this.resultsCurrent = "";
    this.resultsCurrentNegated = false;
  }
```

```
    // When equals is pressed, it's also time to start a new number, but in
    // that case we clear the previous number too.
    if (this.lastKeyPressed == "=") {
      this.lastKeyPressed = "";
      this.resultsCurrent = "";
      this.resultsCurrentNegated = false;
      this.resultsPrevious = "";
      this.resultsPreviousNegated = false;
      this.currentOperation = "";
    }

  } // End checkLastPressed().
```

This function is needed because it's only when a new button is clicked that certain things can happen. For instance, when the user clicks +, -, *, or /, it is time to start a new number. Open the standard Windows calculator and enter a number, click one of those buttons, and then click a number button again. Notice that a new number is started; the old one is cleared (although it is stored). We want to mimic that functionality, and the only way to do it, without having a lot of duplicate code all over the place, is to have this function do that work for us. So, we check if one of those four operation buttons was clicked. If it was, we copy the current number to the previous field (`resultsPrevious = resultsCurrent`) and we also copy whether the value is negative (`resultsPreviousNegated = resultsCurrentNegated`). We then clear the current number and make it positive. The net result is that it works like the standard Windows calculator.

The user clicking the equal sign is also a situation that needs to be handled here. Clicking the equal sign obviously performs some calculation, and that is done in the event handler for that button. But what happens when the next button is clicked? We need to start entering a new number, but in that case, we're not saving the current value as the previous value. Instead, we are just starting a new number. The equal sign button is almost like a reset button, so we need to clear a few extra fields when it is clicked, namely `lastKeyPressed` and `currentOperation`.

Using this as a segue, and not the kind you can ride on,[2] let's jump down now to that method that deals with the equal sign button:

```
  this.button3_9 = function() {

    if (this.currentOperation) {
      var answer = 0;
      // Negate the current value if the flag says to.
      var resCurrent = parseFloat(this.resultsCurrent);
      if (isNaN(resPrevious)) {
        resPrevious = resCurrent;
      }
      if (this.resultsCurrentNegated) {
        resCurrent = resCurent * -1;
      }
```

2. Segway, that annoying scooter all the cool kids in Silicon Valley have, was the brainchild of famous uber-genius Dean Karnen. Yes, I know, bad pun, but work with me here!

```
    // Negate the previous value if the flag says to.
    var resPrevious = parseFloat(this.resultsPrevious);
    if (this.resultsPreviousNegated) {
      resPrevious = resPrevious * -1;
    }
    // Now perform the current operation.
    switch(this.currentOperation) {
      case "+":
        answer = resPrevious + resCurrent;
      break;
      case "-":
        answer = resPrevious - resCurrent;
      break;
      case "*":
        answer = resPrevious * resCurrent;
      break;
      case "/":
        answer = resPrevious / resCurrent;
      break;
    }
    // Reset some variables so we're ready for the next operation or input
    // key, and finally, update the results to show the answer.
    this.resultsCurrent = "" + answer;
    this.resultsPrevious = "";
    this.resultsPreviousNegated = false;
    this.currentOperation = null;
    this.lastKeyPressed = "=";
    this.updateResults();
  }

} // End button3_9().
```

First, a trivial rejection: is there a current operation to perform? This means that if the user enters a number and just clicks the equal sign, nothing will happen. If there is a current operation, we start by getting the numeric form of the current value. If the `resultsCurrentNegated` flag indicates it is a negative number, multiple it by –1 to make it negative. Then we do the same for the previous number. We also do a check here: if there is no previous result, which means that `parseFloat()` resulted in the value not being a number (which we determine by using the built-in `isNaN()` function), we make the previous value the current value. This allows us to mimic the operation of the standard Windows calculator in that you can do 9+= and get 18, and then += again gives you 36, and so on.

After that, the code switches on the current operation and performs the appropriate operation. Next, we set `resultsCurrent` to the answer, appending it to an empty string to convert it to a string. We clear the previous value, reset the negated flag to `false`, clear the current operation, and record the equal sign as the last button pressed. Finally, we display the answer by calling `updateResults()` (remember that `updateResults()` shows the value of `resultsCurrent`, which we just set to the answer). It's really pretty simple.

That covers the operations involving two numbers. What about those that involve only a single number—square root and reciprocal? Well, let's look at square root first:

```
this.button0_9 = function() {

  if (this.resultsCurrent != "") {
    this.resultsCurrent = Math.sqrt(parseFloat(this.resultsCurrent)) + "";
    this.updateResults();
  }

} // End button0_9().
```

As long as there is a current value, we use the `Math` package's `sqrt()` function to do the work, appending a blank string onto the result so that the value we put into `resultsCurrent` is still a string. Then we display the new number by calling `updateResults()`, and that's that.

Reciprocal is similarly easy:

```
this.button2_9 = function() {

  if (this.resultsCurrent != "") {
    this.resultsCurrent = (1 / parseFloat(this.resultsCurrent)) + "";
    this.updateResults();
  }

} // End button2_9().
```

Only one function remains, and that's percentage. Percentage is a bit of a hybrid in that it requires two numbers like addition, subtraction, multiplication, and division, but it operates immediately when the % button is clicked, so the calculation is performed in the event handler method, as it is for square root and reciprocal. Here's the code for that method:

```
this.button1_9 = function() {

  if (this.resultsCurrent != "" && this.resultsPrevious != "") {
    var a = parseFloat(this.resultsPrevious) / 100;
    var b = a * parseFloat(this.resultsCurrent);
    this.resultsCurrent = b + "";
    this.updateResults();
  }

} // End button1_9().
```

The check to make sure we have a current value also now includes a check to be sure we have a previous value. Once we know we have both, we get the numeric version of each, and divide the previous value by 100. Then multiplying that result by the current value results in a percentage value. So, we again set `resultsCurrent` to the answer and display it with `updateResults()`.

The only thing left to discuss is how exactly this class extends the `Mode` class. The answer is found in the last two lines of code in the `Standard.js` file (not including comments):

```
// Standard inherits from Mode.
Standard.prototype = new Mode();
// Continue the sequence of events after this class is loaded.
calcTron.setMode(new Standard());
```

The prototype concept was discussed in Chapter 2, so I refer you to that chapter if the first line isn't clear. That single line implements the inheritance, and it's as simple as that.

The second line, however, is new, and it is what continues the class-loading cycle as previously discussed.

And that covers the Standard CalcTron mode. The BaseCalc mode is next.

Writing BaseCalc.json and BaseCalc.js

In the interest of saving space, I won't show the JSON for this mode because, frankly, once you've seen one mode's JSON, you've pretty much seen 'em all. You should take a look at it on your own, but don't spend more than a minute on it if you've already examined the JSON for the Standard mode—it's substantially the same.

The `BaseCalc` mode implementation class is also substantially similar to the `Standard` mode class, but there are some differences. To begin, let's examine the UML diagram for it, shown in Figure 4-10.

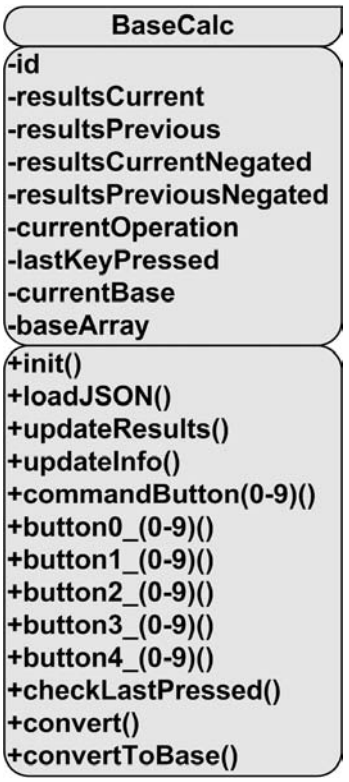


Figure 4-10. *UML diagram of the BaseCalc class*

As you would certainly expect, it contains all the same fields and methods as the `Standard` class owing to the fact that they both extend the `Mode` class. It also contains a few additional items though, as is allowed when extending a class. The `currentBase` field records the number base of the current value: decimal, hexadecimal, binary, or octal. The `baseArray` is an array of alphanumeric values that is used during base conversions, as you'll see shortly. In addition to these extra fields, you also see a couple additional methods: `convert()` and `convertToBase().()`. We'll get to those soon, but let's look at some plumbing first.

As you examine this code, you'll see that it is very similar to the `Standard` class, but one difference should jump out at you: `BaseCalc` implements an `init()` method. In fact, it overrides `init()` in the superclass (`Mode`). Let's see what bizarreness is going on there exactly:

```
this.init = function(inVal) {

  if (inVal) {

    // Initialize array for base conversions.
    this.baseArray[1] = "0";
    this.baseArray[2] = "1";
    this.baseArray[3] = "2";
    this.baseArray[4] = "3";
    this.baseArray[5] = "4";
    this.baseArray[6] = "5";
    this.baseArray[7] = "6";
    this.baseArray[8] = "7";
    this.baseArray[9] = "8";
    this.baseArray[10] = "9";
    this.baseArray[11] = "A";
    this.baseArray[12] = "B";
    this.baseArray[13] = "C";
    this.baseArray[14] = "D";
    this.baseArray[15] = "E";
    this.baseArray[16] = "F";

    // Call superclass constructor. Note that this only works if the
    // method of the superclass does not reference anything specific to the
    // subclass... see the notes about the id field on the next statement!
    BaseCalc.prototype.init(inVal);

    // Note that the call to init() of the superclass will result in the
    // information bar saying "null Mode" because the this reference points
    // to the instance of Mode that is the prototype for this BaseCalc
    // instance. So, we need to set it here using the id field of this
    // instance so what's in the info bar is correct.
    this.updateInfo(this.id + " Mode");

  } else {
```

```
      // Load the JSON for this tab.
      this.loadJSON(this.id);

    }

  } // End init().
```

Because this version of `init()` overrides that found in `Mode`, any time `calcTron.currentMode.init()` is called (because `currentMode` will be pointing to an instance of `BaseCalc` when we switch to that mode), the version of `init()` seen here will be executed, *not* the one in `Mode`, as is the case with the `Standard` class. The first time it is called, no value is passed, so the `else` clause is executed, and the JSON for the mode is loaded. This is what happens in the `Mode` class's version of `init()` as well.

The second time it is called, however, the loaded JSON will be passed in, so we wind up in the `if` block. There, the first thing you see is the `baseArray` being initialized with values. We'll skip over the purpose of that array for a little while longer; for now, it's enough to know it is populated with values here.

Next, you see an interesting thing that might be new to you in JavaScript: calling the superclass version of a function. To do so, we reference the prototype of the `BaseCalc` class, and call `init()` on it. You can think of `BaseCalc.prototype` as being equivalent to `super()` in Java, if you are familiar with Java. The difference is that you must name the method you want to call in JavaScript; hence, `init()` is tacked on. This call carries the JSON input argument with it, so the usual `init()` functionality you saw in the `Mode` base class executes now. Once that call returns, we update the info bar to state the new mode. The astute reader might wonder why this is necessary, since it is done in the `Mode` class's `init()` method. Here is the code that does so in `Mode`:

```
this.updateInfo(this.id + " Mode");
```

The problem here is that the `this` reference points not to the `BaseCalc` instance, but instead to `Mode` itself, and we therefore get the wrong value showing up in the info bar. So, we essentially override what is done in `Mode`'s `init()` method when control returns to `BaseCalc`'s `init()` method, and the problem is solved.

The `lastKeyPressed()` method of `BaseCalc` is virtually identical to the version from `Standard`, with one extra line of code in the first `if` block. This line of code takes the current value and converts it to decimal:

```
    this.convert("dec");
```

Any time we do calculations, we do so in decimal, and then just convert the result to whatever the current base is. Note that we have a field, `currentBase`, that tells us the base of the current value, but we don't have a field to indicate the base of the previous value. The reason is that we know, because of this line of code, that it's always going to be decimal. As I said, the rest of the method is identical to what you've seen already.

The Backspace and Clear command buttons are also identical to their `Standard` equivalents, so there's no need to review them here.

Next, we come to the four command buttons that allow us to switch modes. They are all similar, so we'll look at just the one for hexadecimal here:

```
this.commandButton1 = function() {

  this.updateInfo("Hexadecimal");
  if (this.resultsCurrent != "") {
    this.convert("hex");
    this.updateResults();
  }
  this.currentBase = "hex";

} // End commandButton1().
```

Here, we're just displaying the new mode in the info bar at the bottom. And if we have a current value, we convert it to the new base with a call to `convert()`, passing it the base to convert to—`hex` in this case (`dec`, `oct`, and `bin` are the other valid parameter values). We update the result display as well, which gives us the ability to convert the current number to any base. Finally, we record the new number base, and we're finished.

Like the command buttons, the input buttons are all nearly the same as those in the `Standard` class. However, there are some differences in a few buttons. Let's take the number 7 button as an example:

```
this.button0_5 = function() {

  if (this.currentBase == "bin") {
    return;
  }
  this.checkLastPressed();
  this.lastKeyPressed = "7";
  this.resultsCurrent += "7";
  this.updateResults();

} // End button0_5().
```

Here, we begin with a check to ensure that the current number base isn't binary. If it is binary, we immediately return and do nothing else. This has the effect that when binary is the number base, only the 0 and 1 buttons will be responsive; all others will do nothing (all of the buttons after 0 and 1 have this check in them). Likewise, if you look at 8 and 9, you'll see a check to return immediately if the number base is binary *or* octal. And the A–F buttons will return immediately if the number base is anything other than hexadecimal. The net result is that only buttons valid for the currently selected number base will be reactive, as you would expect.

The +, -, *, and / buttons are the same as in `Standard`, so we'll skip them here. There is no negation, percentage, square root, or reciprocal in this mode, so nothing to see there either.

Now we come to the method for the equal sign. It is again largely like its counterpart in `Standard`, so we'll just look at the differences. First, recall a while back I mentioned that all calculations are done in decimal? Well, as you may have guessed, one of the first things we see is a conversion of the current value to decimal (the previous value is already in decimal, so only the current value is of concern). After that is the same sort of `switch` on the current operation as in `Standard`. After that is something a little different:

```
    // Next, convert the new current value to the current base.  Before we
    // can do that though, we need to set the current base to decimal to
    // match the answer we have.
    var storedCurrentBase = this.currentBase;
    this.currentBase = "dec";
    this.convert(storedCurrentBase);
    this.currentBase = storedCurrentBase;
```

We need to convert the answer to the current number base. However, you'll notice when we look at the `convert()` method that the `currentBase` field is changed by `convert()` to the number base specified for the conversion. This is because `convert()` is used by the four number base command buttons to convert the current value, and also record the new number base. So what we have here is saving the current number base first, then calling `convert()`, which results in `currentBase` being set to `dec`. We then set `currentBase` to the number base stored before the call to `convert()`, which may or may not have been `dec`, and the net result is that the current number base is preserved; the change by `convert()` is effectively undone.

Next up is the `convert()` method itself:

```
this.convert = function(inNewBase) {

  var currentValue = null;
  switch (this.currentBase) {
    case "dec":
      currentValue = parseInt(this.resultsCurrent, 10);
    break;
    case "hex":
      currentValue = parseInt(this.resultsCurrent, 16);
    break;
    case "oct":
      currentValue = parseInt(this.resultsCurrent, 8);
    break;
    case "bin":
      currentValue = parseInt(this.resultsCurrent, 2);
    break;
  }
  switch (inNewBase) {
    case "dec":
      currentValue = this.convertToBase(currentValue, 10);
    break;
    case "hex":
      currentValue = this.convertToBase(currentValue, 16);
    break;
    case "oct":
      currentValue = this.convertToBase(currentValue, 8);
    break;
```

```
      case "bin":
        currentValue = this.convertToBase(currentValue, 2);
      break;
    }
    this.resultsCurrent = "" + currentValue;

  } // End convert().
```

It begins by taking the current value and getting it as a numeric value. To do this, we use the `parseInt()` function. This function takes the radix, or number base, of the number as its second argument (its first argument is the number to convert). So, we `switch` on `currentBase` to determine this. Once we have the numeric version of the current value, we then `switch` on `inNewBase` and call `convertToBase()`, passing it the current value and the radix to convert to. The current value is then set to the number in the new base, converted to a string.

You may have noticed that no real conversion per se happens in `convert()`. That function is performed by the `convertToBase()` method:

```
  this.convertToBase = function (inNumber, inNewBase) {

    var str = "";
    var calc = inNumber;
    while (calc >= inNewBase) {
      var divVal = calc % inNewBase;
      calc = Math.floor(calc / inNewBase);
      str += this.baseArray[divVal + 1];
    }
    str += this.baseArray[calc + 1];
    var len = str.length;
    var fnl = "";
    for (var j = 0; j < len; j++) {
      var a = (len - j) - 1;
      var b = len - j;
      fnl += str.substring(a, b);
    }
    return fnl;

  } // End convertToBase().
```

As you can see, this is where that `baseArray` field comes into play. In brief, this function continually divides the incoming number by the radix, and for each division, adds the appropriate elements from `baseArray` to an output string. That iteration ends when the result of the division is smaller than the number base. At that point, the remainder is used as a lookup into `baseArray` as well. The output string now needs to be reversed, because building it up results in the answer in reverse. Once that reverse is done, the string is returned, and what we have is the input number converted to the requested base.

And that concludes our look at `BaseCalc`, and in fact, CalcTron as a whole.

Suggested Exercises

I admit to never being a big fan of math growing up, but as an adult, I've come to appreciate it more, both from a practical standpoint and as a purely intellectual pursuit. It is my hope that CalcTron can at least start you down that same path. To that end, I offer a few suggestions on where you can take this project next to learn more not only about JavaScript, but mathematics in general:

Add memory functions: This is a simple one to get you started. Most calculators have memory functions, but CalcTron does not. I purposely left these out to give you a nice, easy suggestion to start with, so get to it.

Expand `Classloader`: Modify the `Classloader` class to be more generic, and allow it to load *any* class. Perhaps pass the `load()` method a fully qualified package name, such as `com.omnytex.javascript.SomeClass`, and then use the `com.omnytex.javascript` portion as a subdirectory (so it becomes `com/omnytex/javascript`). Also, change it so that the `load()` method accepts the class to verify against as an argument, and automatically call `verify()` when `load()` completes (you should probably make the verification step optional). Also, research to see if there's a way to determine if a `<script>` tag did not load successfully, and add that as a verification step (as I write this, I am not even sure this is possible, so it's going to be a fun exercise for both of us).

Add conversion capabilities: Originally, this chapter was to have been followed by one about building ConvertTron. The concept was to expand CalcTron to include conversion capabilities. Well, things change in the publishing business as a project evolves, so that chapter didn't make the cut. However, the basic idea is still a very good one. Conversions shouldn't be too difficult to add in, so go for it.

Add simple algebra functions: How about a simple algebra solver? I personally wouldn't try to create the code to solve simultaneous equations or anything like that, but $9=x+5$ shouldn't be too tough to solve. Writing an algorithm to handle such simple equations probably shouldn't be too difficult, but should be just challenging enough to be a good learning exercise.

Summary

This chapter's project seemed like a pretty simple idea: a JavaScript-based calculator. Far be it for me to leave things simple though. We took the basic calculator concept and turned it into an extensible framework that can grow as your mathematical requirements do. You saw how some common object-oriented techniques can be implemented in JavaScript, and how those techniques allow you to enhance the basic application as you see fit. You even saw a way to write a rudimentary classloader that verifies that the classes you load are valid (well, somewhat anyway) for the application's purposes.

You also saw how you can dynamically load JavaScript and JSON without reloading the page and without using the `XMLHttpRequest` object that implies Ajax techniques. Additionally, you saw a bit of what the Rico library has to offer. I hope you'll agree that this chapter enforces the "never judge a book by its cover" saying. Even a relatively simple project can expose you to some very interesting techniques.

CHAPTER 5

Doing the Monster Mash: A Mashup

A long time ago, some god-like developer came up with the concept of an Application Programming Interface (API). In short, an API is nothing but a known (to those that might use it) interface to a program or system. The developer came up with this idea, and everyone saw it, and saw that it was a Good Thing™. But, in the immortal words of Dr. Leonard H. McCoy, "... engineers, they love to change things." We couldn't just stick with the term API. No, we just *had* to come up with something new, and that something is the term *mashup*. Since it's a term that is all the rage these days, and also something that often involves JavaScript to a large degree, it's most definitely an appropriate topic for this book. So, in this chapter, I'll introduce the concept of the mashup, and then we'll put that concept to use in a handy little application.

What's a Mashup?

A mashup, as it has come to be known, is basically a web site or application that takes content from multiple sources, most usually via some sort of public programmatic interface, and integrates it all into a new experience—that is, a new application. If this sounds a bit like the promise of web services to you, you aren't too far off. In fact, web services are sometimes involved in mashups, although that setup typically involves a server infrastructure and some server-side code, *vis-à-vis*. You won't typically call on *true* web services (SOAP, UDDI, WSDL, and all that the term *web services* typically denotes) from a JavaScript client, and almost certainly not from a web browser (not without plug-ins or similar technology, generally).

No, in recent times, the term mashup has generally come to mean browser-based JavaScript clients aggregating content through public APIs from various companies and vendors to form new applications. These APIs are often referred to as web services, and even though they may not truly be web services in the sense of using the full technology stack, they fulfill the same basic goal as those types of web services. They provide services and function over a network—specifically, the Web—so calling them web services isn't really too far-fetched!

Many companies are getting into the API business, including companies you've certainly heard of: Google, Yahoo, Amazon, and eBay, just to name a few. Google and Yahoo have really led the charge, and Yahoo, in particular, originated a neat trick that will be central to the application we'll build in this chapter.

Mashups are also a part of what people often mean when they use the term Web 2.0. Web 2.0 means different things to different people, but sharing resources is usually part of what people mean by it, so mashups certainly fit right in.

One of the other things that is often lumped under Web 2.0 is effects. Take a look at a site like Digg, for instance. I was going to insert a screenshot here, but truthfully, it wouldn't get the point across because it has to be seen live. So please visit `http://www.digg.com`, if you aren't already a frequent visitor (and you should be, by the way!) and just look around a bit. As you do, take note of the various effects. For instance, assuming you aren't signed in, try to click the Digg It button next to an article. Notice how the text is faded and you get a little pop-up over it telling you about signing up? Now, if you create an account, sign in, and try clicking that button again, you'll see the Digg count fade out, then fade back in with the new value. These are all examples of the kinds of UI effects that most consider a part of Web 2.0.

Monster Mash(up) Requirements and Goals

Now, with all of that about mashups in mind, let's discuss what this chapter's application will do and what it will demonstrate.

- The basic function of the application is to be able to enter a ZIP code and to get from that a list of hotels.
- For each hotel, we should be able to click its name and see some extended information.
- In addition to extended information, we would also like to see a map of the area.
- We should be able to zoom in and out on the map.
- The UI will use various effects provided by the well-known script.aculo.us library.
- We will utilize some APIs from Google and Yahoo to get the hotel information and maps.
- This application should be purely browser-based and not require any server component to function.

All of this will result in an application that is fully buzzword-compliant and very much fits the most common definition of Web 2.0 applications. Let's start by taking a look at the Yahoo and Google APIs.

The Yahoo APIs

Yahoo did something very cool a short while ago, and it is this one cool thing that makes the application in this chapter possible. Before we can discuss that though, we have to discuss what was going on before the coolness occurred.

For a while now, many companies have been exposing public APIs for people to use, Yahoo among them. For instance, you could perform a Yahoo search remotely, or you could get a Yahoo map from your own application, and so on. These APIs—these "web services," if you will—usually used XML as their data-transport mechanism. You would post some XML to a given URL, and you would get an XML response back. It was (and still is) as simple as that. These types of services don't require all the web services like SOAP, UDDI, WSDL, and the like.

You may have heard the term *Ajax*, which stands for Asynchronous JavaScript And XML. In fact, if you read Chapter 4, you already saw Ajax in action a bit (although ironically, as we'll be doing here, not using the prototypical `XMLHttpRequest` object). We'll be getting into Ajax in detail in later chapters, but for now, it is enough to know that Ajax is a technique by which you can make a request of a server and use the results it returns in some way *without* reloading the entire page. The most common operation is simply to insert the returned result into the page somewhere, essentially performing an out-of-band partial update of the page. Ajax usually (and some may say necessarily, but I disagree) implies the use of the `XMLHttpRequest` object in the browser to make requests.

`XMLHttpRequest` is a component that makes the request to the server on your behalf and then calls a specified JavaScript callback function to process the result. For the sake of the discussion here, you need to be aware that this object presents one consistent limitation, which is known as the *same-domain* restriction. This means that the `XMLHttpRequest` object will not allow a request to a domain other than the domain from which the document it is in was served. For instance, if you have a page named `page1.htm` located at `http://www.omnytex.com`, you can make requests to any URL at `www.omnytex.com`. However, if you try to make a request to something at `www.yahoo.com`, the `XMLHttpRequest` object won't allow it. This means that the APIs Yahoo exposes aren't of much use to you if you try to access them directly from a browser.

There are ways around this same-domain restriction. Probably the most common is to write a server-side component on your own server that acts as a proxy. So you can make requests via `XMLHttpRequest` to something like `www.omnytex.com/proxy`, which makes a request to something at `www.yahoo.com` for you and returns the results. This is very cool.

However, wouldn't it be so much more useful if you could make the request directly to Yahoo from the browser and not need a server-side component? Yes, indeed it would be! And as you probably have guessed, there is a way to do it. Take a look at the following bit of JavaScript:

```
var scriptTag = document.createElement("script");
scriptTag.setAttribute("src", "www.yahoo.com/someAPI");
scriptTag.setAttribute("type", "text/javascript");
var headTag = document.getElementsByTagName("head").item(0);
headTag.appendChild(scriptTag);
```

So, what we have here is a new `<script>` tag being created. We set the `src` attribute to point to some API at Yahoo, and finally we append that new tag to the `<head>` of the document. The browser will go off and retrieve the resource at the specified URL, and then evaluate it, just as it does for any imported JavaScript file.

Now, in and of itself, that isn't very useful. As I said, the Yahoo APIs return XML, and XML being evaluated by the browser won't do much (some browsers may generate a DOM object from it, but even still, that on its own isn't of much use). Unlike with the `XMLHttpRequest` object, you don't get any events to work with, callback functions that can act upon what was returned, and so on.

Now we come to the bit of coolness I mentioned before!

Let's say we have some XML being returned by a Yahoo service like so:

```
<name>Frank</name>
```

It may not be very interesting, but it's perfectly valid XML. So what is the JSON equivalent to that XML? It's nothing more than this:

```
{ "name" : "Frank" }
```

OK, now suppose that we pass that JSON to a JavaScript function like so:

```
someFunction( { "name" : "Frank" } );
```

What is the parameter passed to `someFunction()`? As it turns out, it's an object constructed from the JSON. This means that if `someFunction()` is this:

```
function someFunction(obj) {
  alert(obj.name);
}
```

the result is an alert pop-up saying "Frank."

Are you maybe starting to see what Yahoo might have done? If you are saying that the return is something like this:

```
someFunction( { "name" : "Frank" } );
```

then you are absolutely right!

What Yahoo came up with is the idea of returning JSON in place of XML, and wrapping the JSON in a function call. When you call the API function, you tell it what the callback function is. So let's say you wanted to interact with some Yahoo API that returns a person's name, as we've been discussing as our example. Your page might look something like this:

```
<html>
  <head>
    <title>Dummy Yahoo API Test</title>
    <script>
      function makeRequest() {
        var scriptTag = document.createElement("script");
        scriptTag.setAttribute("src", "www.yahoo.com/someAPI/callback= ➥
myCallback&output=json");
        scriptTag.setAttribute("type", "text/javascript");
        var headTag = document.getElementsByTagName("head").item(0);
        headTag.appendChild(scriptTag);
      }
      function myCallback(inJSON) {
        alert(inJSON.name);
      }
    </script>
  </head>
  <body>
    <input type="button" value="Test" onClick="makeRequest();">
  </body>
</html>
```

When you click the button, `makeRequest()` is called, and it uses that dynamic `<script>` tag trick to call the Yahoo API function. Notice the URL, which specifies the name of the callback function and that we want to get back JSON, instead of the usual XML. Now, when the response

comes back, the browser evaluates what was inserted into the document via the `<script>` tag, which would be this:

```
myCallback( { "name" : "Frank" } );
```

`myCallback()` is called at that point, with the object resulting from evaluation of the JSON being passed to it. You can load this page from any domain, and it will work. Hence, we've done what the `XMLHttpRequest` object does (in a basic sense anyway), and we've gotten around the same-domain limitation. Sweet!

Yahoo was the first to use this hack (that I am aware of), but as you'll see, other companies have begun to follow suit, because what this allows is purely client-side mashups and API utilization. No longer do you need a server-side proxy. You can now make the requests across domains directly.

■ **Caution** While this technique is very useful because it allows you to make direct requests to any server you want, it also has the potential for malicious code to be introduced. Remember that what is being returned is script that winds up executing with the same privileges as any other script on the page. This means that there is the potential for scams like stealing cookies, spoofing, phishing, and so on. You therefore want to take care in your choice of services and organizations. Accessing APIs from Yahoo or Google, for instance, isn't likely to present any security issues, but less well-known companies may not be quite as safe.

Now that you know the basics of how we're going to be interacting with the Yahoo APIs, as well as the Google APIs, as it turns out, let's take a look at the Yahoo functions that we'll be using in this application.

Yahoo Maps Map Image Service

Yahoo is going to be providing the maps that you can see on the right side of the application (for a preview, see Figure 5-2). Yahoo Maps is a service that has been around for a while, even before a public interface was provided for it. It allows you to get maps for a given address, as well as access other features, such as traffic and local places of interest. The API Yahoo provides has a number of different services, but for our purposes, we'll be focusing on the Map Image service.

The Yahoo Maps Map Image API allows you to get a reference to a graphic of a map generated according to the parameters you specify in your request. You may specify latitude and longitude or address in your request (we'll be specifying address).

This service is referenced via a simple HTTP request, such as the following:

```
http://api.local.yahoo.com/MapsService/V1/mapImage?appid=YahooDemo&location=11719
```

The `location` parameter specified is just a US ZIP code, and the `appid` is an ID you get when you register for the services, as discussed in the next section. If you go ahead and paste that into the address bar of your web browser, you'll see the following response:

```
<Result>
http://img.maps.yahoo.com/mapimage?MAPDATA=ytUWRed6wXWoR2TGzwl3wROg3iyHedtGDvtw ➥
766fmR4iboSayYoDOI4llk594b5QaoMqKvZB5AdndE5FtDXv8lT8apVTrjOY5Zuhrhiugmeogq5t ➥
GHi5&mvt=m
</Result>
<!--
 ws01.search.re2.yahoo.com uncompressed/chunked Sun Dec 10 22:18:44 PST 2006
-->
```

What you've gotten back is a reference to an image now sitting on Yahoo's servers. If you pluck out the following URL:

```
http://img.maps.yahoo.com/mapimage?MAPDATA=ytUWRed6wXWoR2TGzwl3wROg3iyHedtGDvtw ➥
766fmR4iboSayYoDOI4llk594b5QaoMqKvZB5AdndE5FtDXv8lT8apVTrjOY5Zuhrhiugmeogq5t ➥
GHi5&mvt=m
```

and put that in the address bar of a web browser, you'll see an image that is a map of the Bellport/Mastic Beach area of Long Island, New York.

You can also add some parameters to the original request. For instance, you can specify that you want a GIF back (by default, you get a PNG file), and you can specify that instead of XML, you want JSON back. The URL would then look like this:

```
http://api.local.yahoo.com/MapsService/V1/mapImage?appid=YahooDemo&location=11719 ➥
&image_type=gif&output=json&callback=myCallback
```

Now the response looks like this:

```
myCallback({"ResultSet":{"Result":"http:\/\/image.maps.yahoo.com\/mapimage? ➥
MAPDATA=cgnWqud6wXUpZCKOcjzKJ3PPgRQkY6thMdXo4raWKRcxvbRSpJ67PGisuDp5YO829Zi5fd ➥
hWYTOm5mmvfBZCqHKDBG8ePGPcc8AlFAuhbWwd6rPOwZ67&mvt=m"}});
```

All you need to do now is write the `myCallback()` function and make it somehow display the images, like this:

```
function myCallback(inJSON) {
  document.getElementById("someImgTag").src = inJSON.ResultSet.Result;
}
```

And as you'll soon see, that's just about all this application is doing as far as interacting with Yahoo's services goes! A few other parameters are used in the application, as summarized in Table 5-1.

Table 5-1. *Some Yahoo Map Image Service Parameters Used in Monster Mash(up)*

Parameter	Meaning
`width`	The width of the map image.
`height`	The height of the map image.
`zoom`	The zoom factor to apply to the map. This is a value in the range 1–12, where 1 represents street level and 12 represents regional level (a little wider than state level).

Yahoo Registration

Most API services require you to register to use their APIs, and Yahoo is no exception. As you saw, in the HTTP request, an `appid` parameter is passed. This value is a unique identifier you must pass when making your requests. Not passing this value, or passing an invalid value, will result in the call failing. `YahooDemo` is the `appid` value used in the examples in Yahoo's own documentation. However, before you really play with this application a great deal, you should register and get your own `appid`. You can access the following page to do so:

```
http://api.search.yahoo.com/webservices/register_application
```

You should plug your own `appid` into the `Masher` class (in the `Masher.js` source file) before you spend time with the application, just so you are playing nice with Yahoo.

There are some limitations associated with using the APIs in terms of request volume, but the upper limit is so high as to not be a realistic concern for your adventures with this application! If you are intent on building a production-level application using these services, you will need to consult with Yahoo for other registration options that allow for high volumes. Again, for our purposes, the number of requests allowed is more than sufficient.

The Google APIs

Google Base is, in simplest terms, an online database where people can post about various items, describing them with various attributes. For instance, if you would like to list a number of events occurring in your neighborhood, you can do so at Google Base. Do you want to post your great-aunt Erma's recipe for stuffed cabbage (you wacko, you!)? Google Base is the place to do it.

As you would expect, you can search through the posted information. Would you like to find a list of hotels in a given area? You can do so at Google Base. And that's exactly what we need for this application!

The API for Google Base allows you to query for a list of items, and it also allows you to add items to the database. We care only about querying here, but both capabilities are available.

The Google Base API provides a number of "feeds" through which you can get information. Each feed corresponds to a URL that is formed by taking a base URL and appending the feed-specific path. For instance, the base URL is `http://www.google.com/base`, and the feed-specific portion for the snippets feed is `/feeds/snippets`. You simply put them together to form the final URL where you can query for items:

```
http://www.google.com/base/feeds/snippets
```

So, as a simple example, if you want to do a search for "laptop," you would use this URL:

```
http://www.google.com/base/feeds/snippets?bq=laptop&max-results=1
```

This will result in an XML response like the following:

```
<?xml version="1.0" encoding="UTF-8" ?>
<feed xmlns="http://www.w3.org/2005/Atom" xmlns:openSearch="http://a9.com/-/spec/ ➥
opensearchrss/1.0/" xmlns:g="http://base.google.com/ns/1.0" xmlns:batch= ➥
"http://schemas.google.com/gdata/batch">
  <id>http://www.google.com/base/feeds/snippets</id>
  <updated>2006-12-16T23:28:51.897Z</updated>
  <title type="text">Items matching query: laptop</title>
  <link rel="alternate" type="text/html" href="http://base.google.com" />
  <link rel="http://schemas.google.com/g/2005#feed" type="application/atom+ ➥
xml" href ="http://www.google.com/base/feeds/snippets" />
  <link rel="self" type="application/atom+xml" href="http://www.google.com/base/ ➥
feeds/snippets?max-results=1&bq=laptop" />
  <link rel="next" type="application/atom+xml" href="http://www.google.com/base/ ➥
feeds/snippets?start-index=2&max-results=1&bq=laptop" />
  <generator version="1.0" uri="http://base.google.com">GoogleBase</generator>
  <openSearch:totalResults>3612575</openSearch:totalResults>
  <openSearch:startIndex>1</openSearch:startIndex>
  <openSearch:itemsPerPage>1</openSearch:itemsPerPage>
  <entry>
    <id>http://www.google.com/base/feeds/snippets/18343852209328178501</id>
    <published>2006-11-10T03:41:07.000Z</published>
    <updated>2006-12-08T03:24:30.000Z</updated>
    <title type="text">Portable Laptop Desk</title>
    <content type="html">Looking for portable laptop desk? See our portable ➥
laptop desk guide.</content>
    <link rel="alternate" type="text/html" href="http://portable-laptop- ➥
desk.info" />
    <link rel="self" type="application/atom+xml" href="http://www.google.com/base ➥
/feeds/snippets/18343852209328178501" />
    <author />
    <g:item_language type="text" />
    <g:customer_id type="int">7048781</g:customer_id>
    <g:target_country type="text" />
  </entry>
</feed>
```

The `bq` parameter is the query you wish to perform; in this case, simply the word `laptop`. The parameter `max-results`, which is optional, indicates the maximum number of items to return. For the sake of a short result set being printed here, I requested only a single item be returned.

If you want to query for a particular item type, such as hotels, you can append `-/hotels` to the following URL:

```
http://www.google.com/base/feeds/snippets/-/hotels?bq=laptop&max-results=1
```

Now obviously, that's a little bit of a nonsensical query,[1] but it's a valid query nonetheless.

As a more practical example, suppose that you want to search for a hotel in a given area, say a list of hotels in a given ZIP code (signs and portents here!). To perform that query, you need to use the `location` parameter. The `location` parameter can take a location in just about any form you can imagine—a full street address, just a ZIP code, or even longitude and latitude! The service generally takes care of understanding what you've sent in, so you don't even have to tell it!

What would the URL look like to search for hotels in the 90210 area code? Just like this:

```
http://www.google.com/base/feeds/snippets/-/hotels?bq=%5blocation:@%2290210%22%2 ➥
b50mi%5d&max-results=1
```

Go ahead and try it in your browser—this one will work, too. To understand this more fully, I need to point out that the value of the `bq` parameter, the query you want performed, is actually this:

```
[location: @"90210" + 50mi]
```

This says you want hotels in the 90210 ZIP code, at most 50 miles from roughly the center of the ZIP code. The reason it looks a little funky in the URL is because of the URL-encoding that I already did on the URL: left and right brackets, at sign, plus sign, and quotes. You will also see this is the case in the application, but it's possible to encode the entire thing in one step. Either way gets the job done. As long as the URL is ultimately URL-encoded, that's all that matters.

As with the Yahoo APIs, you can specify that you want JSON returned wrapped in a JavaScript function call, and you can specify the function to call. This is done via the `alt` and `callback` parameters. When you pass the value `json-in-script` for the `alt` parameter, you get exactly that: JSON wrapped in a JavaScript call. The `callback` parameter is naturally the name of the function to which to send the JSON.

And while it doesn't seem like much, this is essentially all the information you'll need to work with this API in this application!

■ **Note** The snippets feed is a read-only public feed, which means you do not need to get an API key to access it. If you want to deal with many other feeds, or if you intend to do write or update operations with the API, you will indeed need a key, just as you do for Yahoo. You can obtain said key at `http://code.google.com/apis/base/signup.html`.

Script.aculo.us Effects

Script.aculo.us is all about the effects! Components that fly onto the page, elements that shrink and expand, parts that fade out of existence, text that color-cycles into existence—all of this can be done with script.aculo.us.

---

1. Note that you *will* get results, because hotels may list laptop wireless support in their description, and the query will see that.

ARE EFFECTS JUST EYE CANDY?

Let's tackle one question that often comes to mind first: why do we need effects at all? Isn't it just a bunch of superfluous eye candy that doesn't serve much purpose other than to make people go "ooh" and "aah"? Well, first off, if you've ever designed an application for someone else, you know that presentation is an important part of the mix. The more people like how your application looks, the more they'll like how it works—whether it works well or not. It's a relative measure. That's the lesser reason though, although one which should not be quickly dismissed.

The much more important reason has do with how we perceive things. Look around you right now. Pick up any object you want and move it somewhere else. Did the object just pop out from the starting point and appear at the new location? No, of course not! It moved smoothly and deliberately from one place to another. Guess what? This is how the world works! And furthermore, this is how our brains are wired to expect things to work. When things don't work that way, it's jarring, it's confusing, and it's frustrating.

People use movement as a visual cue as to what's going on. This is why modern operating systems are beginning to add all sorts of whiz-bang features, like windows collapsing and expanding. They aren't just eye candy. They do, in fact, serve a purpose: they help our brains maintain their focus where it should be and on what interests us.

In a web application, the same is true. If you can slide something out of view and something else into view, it tends to be more pleasant for the users, and more important, helps them be more productive by not making them lose focus for even a small measure of time.

Using script.aculo.us boils down to three simple steps:

1. Import the required JavaScript files.
2. Create a new `Effect` object, passing it the ID of the element to perform the effect on, and optionally, parameters for the effect.
3. Sit back and enjoy!

The required files—`prototype.js`, `scriptaculous.js`, `builder.js`, `effects.js`, `dragdrop.js`, `slider.js`, and `controls.js`—are simply imported like any other external JavaScript files, via `<script>` tags. Once they are present on the page, you initiate an effect like this:

```
new Effect.Appear("div1");
```

This will begin an `Appear` effect, which makes an element fade in over some time period. Assuming we had a `<div>` on the page with the ID `div1` that's what would be faded in. What's happening here is a new object is being instantiated, namely the `Effect.Appear` object. The first argument to the constructor for an effect is always the ID of the element to operate on or a DOM object reference itself, the second is a required parameter (although most effects do not have required parameters), and the third is a collection of options. The options are, well, optional! You'll get some set of default values if you don't pass in any options.

Most effects share some common options, as summarized in Table 5-2.

Table 5-2. *Some Common Script.aculo.us Effect Options*

Option	Description
`duration`	Sets the duration of the effect in seconds, given as a float. Default value is 1.0.
`fps`	Targets this many frames per second. Default value is 25. This cannot be set higher than 100.
`transition`	Sets a function that modifies the current point of the animation, which is between 0 and 1. The following transitions are supplied: `Effect.Transitions.sinoidal` (default), `Effect.Transitions.linear`, `Effect.Transitions.reverse`, `Effect.Transitions.wobble`, and `Effect.Transitions.flicker`.
`from`	Sets the starting point of the transition, a floating-point value between 0.0 and 1.0. Default value is 0.0.
`to`	Sets the end point of the transition, a floating-point value between 0.0 and 1.0. Default value is 1.0.
`sync`	Sets whether the effect should render new frames automatically, which it does by default. If `true`, you can render frames manually by calling the `render()` method of an effect.
`queue`	Sets queuing options. When used with a string, this can be `front` or `end` to queue the effect in the global effects queue at the beginning or end, or a queue parameter object that can have `{position:"front/end", scope:"scope", limit:1}`.
`direction`	Sets the direction of the transition. Values can be `top-left`, `top-right`, `bottom-left`, `bottom-right` or `center` (`center` is the default value). This is applicable only on Grow and Shrink effects.

All the effects also support supplying callback functions in the options. This allows you to perform some function when certain events in the life cycle of an effect occur. Table 5-3 summarizes the possible callbacks.

Table 5-3. *Possible Script.aculo.us Callbacks for Effects*

Callback Event	Description
`beforeStart`	Called before the main effects rendering loop is started
`beforeUpdate`	Called on each iteration of the effects rendering loop, before the redraw takes place
`afterUpdate`	Called on each iteration of the effects rendering loop, after the redraw takes place
`afterFinish`	Called after the last redraw of the effect was made

In this chapter's project, we're going to use the following effects:

- `BlindUp` and `BlindDown`: These function like blinds in your window, basically rolling an element up or down correspondingly. They are used to collapse and expand the search results.
- `Shrink`: This effect reduces an element to its top-left corner, essentially collapsing it to that corner. When extended information for a hotel is showing, and you click a new hotel, we'll use the `Shrink` effect to hide the information that is currently showing.
- `Grow`: This effect expands an element into view. We'll use it when showing extended hotel information and hiding any information that is currently showing.
- `Puff`: This effect gives the illusion of the element puffing away like a cloud of smoke. We'll use it to remove the map when a new search is performed.

So, taking all this into account, let's look at some sample effects and how you code them. All of these assume we have a `<div>` on the page with the ID `div1` and some text in it. First, let's see the `BlindUp` effect:

```
new Effect.BlindUp("div1",
  {
    afterFinish : function() {
      alert("All done!");;
    }
  }
);
```

Here, we've specified a callback function, so that once the element is completely rolled up, we'll see an alert message pop-up.

Here's another example:

```
new Effect.Shrink("div1", {duration : 4.0, fps : 60 } );
```

This will shrink our `<div>` out of view over a period of four seconds and will try to do so at a rate of 60 frames per second. Assuming the computer and browser can achieve this frame rate, it will appear super-smooth to the user. Remember that generally speaking, the human eye begins to perceive animation as being smooth somewhere between 24 and 30 frames per second, so the default value of 25fps is pretty reasonable.[2]

2. For reference, most movies have been shot at a rate of 24 frames per second.

Here are a few more notes about script.aculo.us effects:

- All of the effects are time-based, which means that if you want an element to expand into view using the `Grow` effect, and you want it to take two seconds to do so, the effect will take two seconds, regardless of how fast the browser renders each frame.
- In general, the effects are ignorant of the type of element to which you apply them. You should, generally, be able to apply effects to just about anything. Now, you'll likely find some exceptions, and that's to be expected due to the variations in CSS interpretation by various browsers. But, for the most part, you'll find it to be true.
- For most of these effects to work, you must specify at least some style attributes in-line with the element; they will *not* work if specified in an external style sheet. For instance, many of the effects you'll see in this chapter's application won't work if the `display` attribute isn't specified in-line. This is not exactly a big deal, but the first time you try an effect and find that it doesn't do anything, and you wonder why, remember this point. You'll likely save yourself some time!

With the review of the APIs and script.aculo.us out of the way, let's begin to look at the application itself and see how it all comes together,

A Preview of the Monster Mash(up)

The Monster Mash(up) application is relatively simple in appearance, until you actually play with it. The page consists essentially of the following four main sections:

- The top, which contains the title graphic and a place to enter a ZIP code
- A section below that and to the left where hotel search results are shown (as well as hotel information)
- A section to the right where a map of the area surrounding the hotel will be shown
- Between the results and map sections, a section containing the map zoom buttons

Figure 5-1 shows this page as it appears when you have performed a search and are now viewing the list of matching hotels.

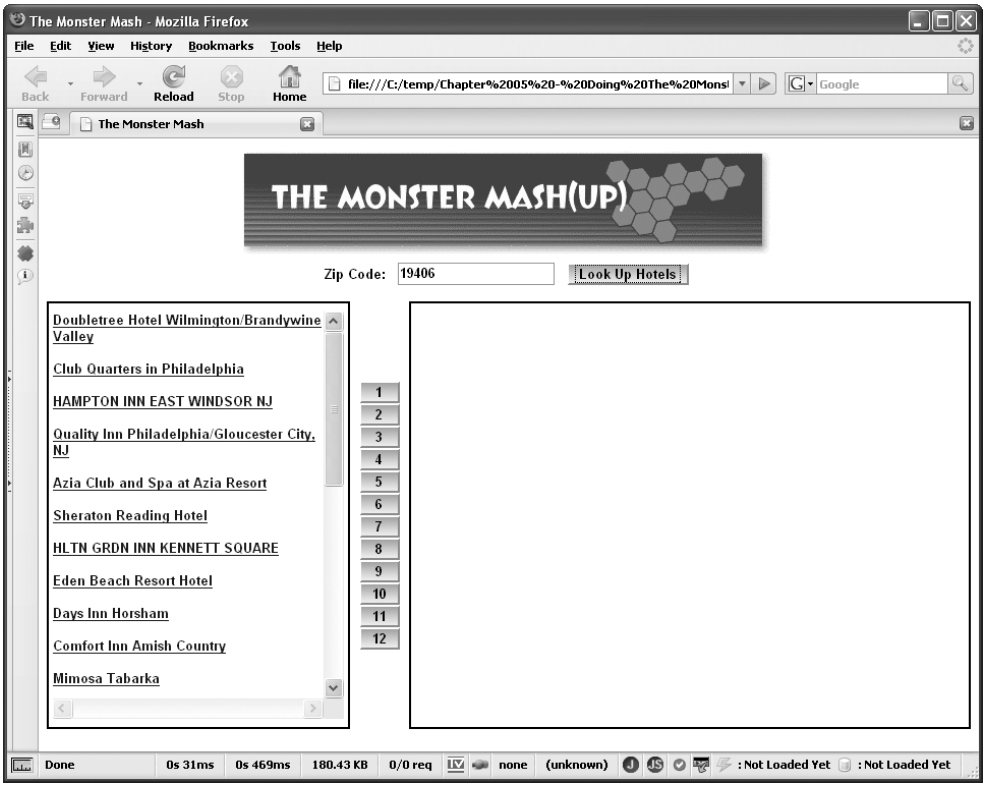


Figure 5-1. *Monster Mash(up) showing a list of hotels matching the requested ZIP code*

When you search for results, the list of hotels returned expands into view, which obviously you can't see in a screenshot here! If results are already showing and you do another search, the existing results will shrink out of view.

When you click a hotel name, you get extended information about that hotel just below the clicked item. This information "flies" into view, and conversely, it flies out of view when you click another hotel name. Figure 5-2 shows the screen when information is being viewed.

Along with the extended information, you'll also get a map of the area around the hotel, also visible in Figure 5-2. You can then use the zoom buttons to zoom in for a closer look or zoom out for a wider look. The smaller the zoom number, the closer you go. Level one is street level; level 12 is regional/state level.

One of the neatest effects to see is when you perform a new search while a map is showing. You'll see that the map "puffs" out of existence—that is, it flies toward *you* and fades out at the same time. Everyone say, "Thank you, script.aculo.us!"

And now, in the immortal words of Bugs Bunny and Daffy Duck (you *do* remember the theme song from Saturday mornings, don't you?): "On with the show!"

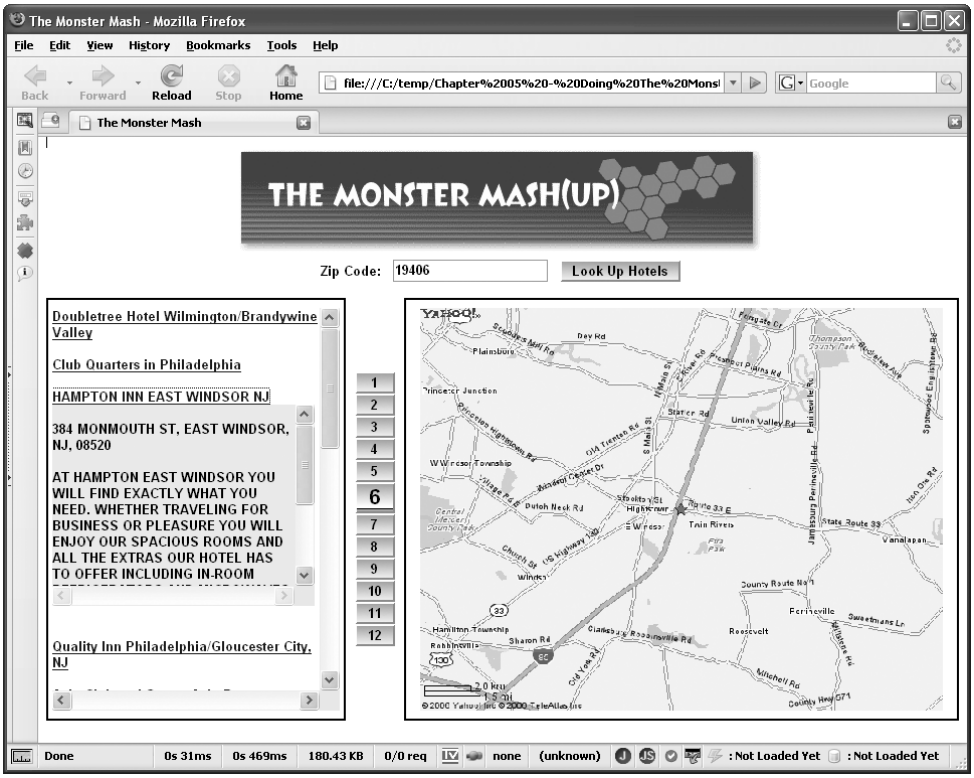


Figure 5-2. *Monster Mash(up) showing a map for a selected hotel, along with extended information*

Dissecting the Monster Mash(up) Solution

Understanding any application tends to begin with a high-level look at the files—whether source or executable (or both)—that make up the application, and we'll not make an exception to that here! In Figure 5-3, you can see the directory structure of our mashup application.

The root directory holds the single HTML file that constitutes the mashup, quite unimaginatively named `mashup.htm`! The `css` directory contains a single `styles.css` style sheet. In the `img` directory are four images: `buttonBG.gif`, which is the background image used to give the metallic look to all the buttons; `pixel_of_destiny.gif`, otherwise known as a single-pixel transparent image (Google for "pixel of destiny," to get the joke!); `retrieving_map.gif`, which is the image containing the message seen in the map area while a map is retrieved; and `title.gif`, which is simply the title banner at the top of the page.

The `js` directory is where all the JavaScript files that make up the application are found. Some of them, namely `builder.js`, `controls.js`, `dragdrop.js`, `effects.js`, `prototype.js`, `scriptaculous.js`, and `slider.js`, are all components of the script.aculo.us library, and we will not be reviewing them here.[3]

3. It is always a valuable and worthwhile exercise to look at the code of the pros, so I certainly suggest you spend at least a few minutes looking at how script.aculo.us works. It's in no way required to understand the mashup, but you'll likely pick up some tricks.

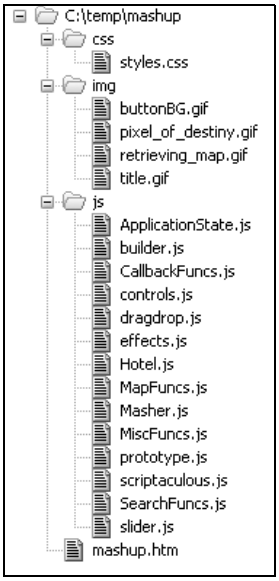


Figure 5-3. *Directory structure of the mashup application*

The remaining files make up the mashup application itself, starting with `ApplicationState.js`, which defines a class named `ApplicationState` (and I'm willing to bet you can surmise its purpose!). `CallbackFuncs.js` contains the functions that will be "called back" by the Yahoo and Google web services. `Hotel.js` defines a `Hotel` class that is used to describe a hotel. `MapFuncs.js` contains functions for working with the map for a given hotel. `Masher.js` contains the functions dealing with communicating with the Google and Yahoo web services. `MiscFuncs.js` contains, well, miscellaneous functions, what else? Finally, `SearchFuncs.js` contains functions for performing a search for hotels.

Now that we've completed the preliminaries, let's get to looking at some actual code!

Writing styles.css

The `styles.css` file, shown in Listing 5-1, is the single style sheet used to define styling for the application.

Listing 5-1. *The styles.css File*

```
/* Style applied to everything on the page. */
* {
  font-size        : 10pt;
  font-weight      : bold;
  font-family      : arial;
}
```

```
/* Style applied to the body of the document. */
.cssBody {
  background-color : #ffffff;
}


/* Style applied to the left and right sections. */
.cssSectionBorder {
  border           : 2px solid #000000;
}


/* Style applied to the search results div. */
.cssSearchResults {
  width            : 100%;
  height           : 400px;
  overflow         : scroll;
}


/* Style applied to the popup area where hotel info is displayed. */
.hotelInfo {
  width            : 98%;
  height           : 200px;
  overflow         : scroll;
  background-color : #eaeaea;
}


/* Style applied to buttons. */
.cssButton {
  width            : 40px;
  border-color     : #ffffff;
  background       : url(../img/buttonBG.gif);
  color            : #000000;
}


/* Style applied to buttons when they are hovered over. */
.cssButtonOver {
  width            : 40px;
  border-color     : #ff0000;
  background       : url(../img/buttonBG.gif);
  color            : #ff0000;
}
```

The first style uses the wildcard selector to apply those styles to all elements within the document. This is a handy trick to cover *everything*, even things that typically can be problematic to apply styles to globally, such as table cells, which don't get style information cascaded down properly in some browsers.

For the body, we apply a background color of white using the `cssBody` class. The `cssSectionBorder` is applied to the cells that house the search results on the left and the map on the right. The `cssSearchResults` class styles the search results, ensuring that they fit nicely within the confines of the left side of the page, and also ensuring that the results will scroll when bigger than that area (`overflow:scroll`).

The `hotelInfo` class styles the section below a particular hotel where its information appears. With this class, we define that it doesn't quite stretch across the entire content area it's in, so that its scrollbar doesn't touch the scrollbar of the search results.

The `cssButton` class is used to style our buttons. With it, we are specifying a background image and making sure all buttons have a consistent size. The background image is designed so that it will fill up just about any size button in the proper fashion, giving the button a brushed-metal look. This is purely an aesthetic touch, but it gives the buttons a nice, unique look. The `cssButtonOver` class is essentially the same, but with a red text color and border, which is used when you hover over the button to give it an active look.

All in all, this is a pretty simple style sheet. Not too much of interest is going on here, except perhaps for the buttons, which I feel are a neat little touch.

Writing mashup.htm

The `mashup.htm` file, shown in Listing 5-2, is the starting point for this application, and the file that defines the basic page layout.

Listing 5-2. *The mashup.htm File*

```
<html>
  <head>

    <title>The Monster Mash</title>

    <link rel="StyleSheet" href="css/styles.css" type="text/css">

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

    <script src="js/ApplicationState.js" type="text/javascript"></script>
    <script src="js/Masher.js" type="text/javascript"></script>
    <script src="js/Hotel.js" type="text/javascript"></script>
    <script src="js/CallbackFuncs.js" type="text/javascript"></script>
    <script src="js/SearchFuncs.js" type="text/javascript"></script>
    <script src="js/MapFuncs.js" type="text/javascript"></script>
    <script src="js/MiscFuncs.js" type="text/javascript"></script>
```

```
  <script>
    masher = new Masher();
    appState = new ApplicationState();
  </script>

</head>

<body class="cssBody">

  <center>
    <img src="img/title.gif" vspace="5">
    <br>
    <form onSubmit="search();return false;">
    Zip Code:
     
    <input type="text" id="zipCodeField" value="">
     
    <input type="submit" value="Look Up Hotels"
      class="cssButton" style="width:120px;"
      onMouseOver="this.className='cssButtonOver';"
      onMouseOut="this.className='cssButton';">
    </form>
    <br>
  </center>

  <table border="0" width="100%" cellpadding="4" cellspacing="0">
    <tr>

      <td align="left" height="420" class="cssSectionBorder">
        <img width="1" height="1" src="img/pixel_of_destiny.gif">
        <center>
          <div id="pleaseWait" style="display:none;">
            Please Wait, Retrieving Data...
          </div>
        </center>
        <div id="searchResults" class="cssSearchResults" style="display:none;">
        </div>
      </td>

      <td width="50" align="center">
        <input type="button" value="1" onClick="zoomMap(1);" id="zoomButton1"
          class="cssButton"
          onMouseOver="this.className='cssButtonOver';"
          onMouseOut="this.className='cssButton';"><br>
        <input type="button" value="2" onClick="zoomMap(2);" id="zoomButton2"
          class="cssButton"
          onMouseOver="this.className='cssButtonOver';"
          onMouseOut="this.className='cssButton';"><br>
```

```
    <input type="button" value="3" onClick="zoomMap(3);" id="zoomButton3"
      class="cssButton"
      onMouseOver="this.className='cssButtonOver';"
      onMouseOut="this.className='cssButton';"><br>
    <input type="button" value="4" onClick="zoomMap(4);" id="zoomButton4"
      class="cssButton"
      onMouseOver="this.className='cssButtonOver';"
      onMouseOut="this.className='cssButton';"><br>
    <input type="button" value="5" onClick="zoomMap(5);" id="zoomButton5"
      class="cssButton"
      onMouseOver="this.className='cssButtonOver';"
      onMouseOut="this.className='cssButton';"><br>
    <input type="button" value="6" onClick="zoomMap(6);" id="zoomButton6"
      class="cssButton"
      onMouseOver="this.className='cssButtonOver';"
      onMouseOut="this.className='cssButton';"><br>
    <input type="button" value="7" onClick="zoomMap(7);" id="zoomButton7"
      class="cssButton"
      onMouseOver="this.className='cssButtonOver';"
      onMouseOut="this.className='cssButton';"><br>
    <input type="button" value="8" onClick="zoomMap(8);" id="zoomButton8"
      class="cssButton"
      onMouseOver="this.className='cssButtonOver';"
      onMouseOut="this.className='cssButton';"><br>
    <input type="button" value="9" onClick="zoomMap(9);" id="zoomButton9"
      class="cssButton"
      onMouseOver="this.className='cssButtonOver';"
      onMouseOut="this.className='cssButton';"><br>
    <input type="button" value="10" onClick="zoomMap(10);" id="zoomButton10"
      class="cssButton"
      onMouseOver="this.className='cssButtonOver';"
      onMouseOut="this.className='cssButton';"><br>
    <input type="button" value="11" onClick="zoomMap(11);" id="zoomButton11"
      class="cssButton"
      onMouseOver="this.className='cssButtonOver';"
      onMouseOut="this.className='cssButton';"><br>
    <input type="button" value="12" onClick="zoomMap(12);" id="zoomButton12"
      class="cssButton"
      onMouseOver="this.className='cssButtonOver';"
      onMouseOut="this.className='cssButton';">
  </td>

  <td align="center" width="540" class="cssSectionBorder">
    <img id="mapFiller" style="display:block;" width="520" height="400"
      src="img/pixel_of_destiny.gif">
    <img id="map" style="display:none;" width="520" height="400"
      src="img/pixel_of_destiny.gif">
```

```
        </td>

      </tr>
    </table>

  </body>

</html>
```

As you can see, `mashup.htm` begins with a series of JavaScript file imports. The first two, `prototype.js` and `scriptaculous.js`, are the two source files required for script.aculo.us to work. The `scriptaculous.js` file takes care of including all the other JavaScript files it depends on, such as `builder.js` and `effects.js`. The remainder of the imports are the source files that make up the application itself.

Following the JavaScript imports is one very small section of JavaScript that is responsible for creating an instance of the `Masher` and `ApplicationState` objects, which we'll discuss shortly. That small block of script concludes the `<head>` of the document.

Moving along to the `<body>`, we begin with the title graphic, a text field where you enter the ZIP code you want to search, and then the search button. This is all part of a form, but ironically, it only needs to be in order to deal with allowing the Enter key to be used in Firefox (in IE, you can drop the `<form>` and just use a plain-old `<input type="button">`). That's why we make a call to `search()` `onSubmit` of the form, and also return `false`, which stops the form from actually being submitted. This is one of those times Firefox actually makes things *more* difficult than IE!

Next up we find the start of a table that is used to define the two halves of the screen: the search results on the left and the map on the right, with the map zoom buttons separating them.

The first column of the table contains two `<div>` elements. The first contains the "Please Wait" message you see while a search is in progress. This is initially hidden. The next `<div>` will contain the search results. Note that it, too, is initially hidden. But more important, that style is specified in-line and not in the external style sheet. This is no oversight! In fact, this `<div>` will be involved in some script.aculo.us effects, and for those to work, some style information must be specified in-line. This is true of one or two other areas of the page, for the same reason. For elements that will be hidden and shown via script.aculo.us, you need to define their initial visibility in-line in the elements by setting the display `style` attribute. It's as simple as that.

■ **Note** The reason that script.aculo.us effects require the initial visibility specified in-line is that script.aculo.us is based on the Prototype library, and it uses the `show()` function that Prototype supplies to make a specified element visible. This function works by setting the element's `display` style attribute to an empty string (undefined, in other words). The idea here is to set it to its default value—the empty string for that particular attribute—which means the element would be visible. A problem can occur, however, if that attribute is defined "higher up" in the CSS than at the element level. Remember that Prototype is overwriting what's at the element level. In this situation, it will look at the undefined value at the element level, even though your style sheet may have specified `display:none`, and the net result is it looks like nothing happens.

Following all of that is the second column in the table where the map zoom buttons are located. Within it is a series of 12 buttons. They all share the same basic structure in that the `onClick` event handler attached calls the `zoomMap()` function, passing it the zoom level corresponding to the button. There are also the same `onMouseOver` and `onMouseOut` event handlers we saw earlier on the search button. It's nothing too fancy really.

Lastly, we come to the table column that contains the map, or more precisely, *will* contain the map, once you select a hotel to view. This column contains two `<img>` tags. The one with the ID `mapFiller` is needed so that the table cell doesn't collapse and remains a constant size in IE, which makes for a rather ugly display. The tag with the ID `map` displays the map or the "Please Wait" message, which is itself an image. The `src` of this tag is updated to point to either that Please Wait image or to the map that Yahoo generates. These two `<img>` tags are alternatively hidden and shown as appropriate.

And that wraps it up for the markup!

Writing ApplicationState.js

The `ApplicationState.js` file contains, not surprisingly, the `ApplicationState` class. While this class is simple, it plays a very important role in that it stores values that are used throughout the application. Figure 5-4 is the UML class diagram for the `ApplicationState` class.

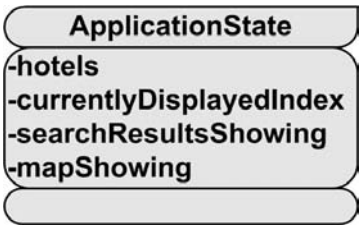


Figure 5-4. *UML diagram of the ApplicationState class*

As you can see in Listing 5-3, the `ApplicationState` class contains a grand total of four fields.

Listing 5-3. *The ApplicationState.js File*

```
/**
 * This class stores information about the state of the application.
 */
function ApplicationState() {


  /**
   * This is an array of hotels returned from a search. */
  this.hotels = new Array();
```

```
/**
 * This is the index into the hotels array of the hotel that is currently
 * being viewed, i.e., that has info showing and that has a map showing.
 */
this.currentlyDisplayedIndex = -1;


/**
 * This is a flag that indicates whether search results are currently
 * showing or not.
 */
this.searchResultsShowing = false;


/**
 * This is a flag that indicates whether a map is currently showing or not.
 */
this.mapShowing = false;


} // End ApplicationState.
```

The `hotels` array is a collection of `Hotel` objects that is populated when a search is performed. The field `currentlyDisplayedIndex` stores the index into the `hotels` array of the hotel currently being viewed. The `searchResultsShowing` field is a simple flag that tells us whether search results are currently visible. Finally, the `mapShowing` field is another flag that tells us whether a map is showing.

You will see these fields used throughout the rest of the code in determining what to do at any given point, but by and large, `ApplicationState` is a very simple class.

Writing Hotel.js

As you just saw, the `ApplicationState` class contains an array named `hotels`, which is information for the hotels from the current search results. The array contains `Hotel` objects, one for each hotel. In Figure 5-5, you can see the UML class diagram for the `Hotel` class.

Figure 5-5. *UML diagram of the Hotel class*

Like ApplicationState, the Hotel class does not have a whole lot to it; in fact, less than there was to the ApplicationState class! A grand total of three fields gives us all the information about a hotel that we need for this application, as shown in Listing 5-4.

Listing 5-4. *The Hotel.js File*

```
/**
 * This class represents a hotel as returned by the Google web service.
 */
function Hotel() {


  /**
   * The name of the hotel.
   */
  this.name = "";


  /**
   * The location (address) of the hotel.
   */
  this.location = "";


  /**
   * The description of the hotel.
   */
  this.description = "";


} // End Hotel.
```

The name field is the name of the hotel, as seen in the results list. The location field is the address of the hotel as returned by the web services. The description field is the entire description for the hotel. This description can, and in many cases does, contain the name and location as well, so there tends to be some redundancy in the display.

Writing SearchFuncs.js

Everything in this application starts with a search, so it seems reasonable to start looking at the first of the code where the real action takes place by starting with searching. When the search button is clicked, the search() function is called. This function is shown in Listing 5-5.

Listing 5-5. *The SearchFuncs.js File*

```
/**
 * Start execution of a search by ZIP code with Google Base.
 */
```

```
function search() {

  // Reset things that need to be reset.
  resetZoomButtons();
  appState.currentlyDisplayedIndex = -1;

  // Hide information for all hotels.
  for (var i = 0; i < appState.hotels.length; i++) {
    $("hotelInfo" + i).style.display = "none";
  }

  // If there are search results showing, hide them.
  if (appState.searchResultsShowing) {

    new Effect.BlindUp("searchResults",
      {
        afterFinish : function() {
          // Now do the actual search.
          searchPart2();
        }
      }
    );
    // If a map and hotel info are showing, hide them too.
    if (appState.mapShowing) {
      new Effect.Puff("map",
        {
          afterFinish : function() {
            $("map").style.display = "none";
            $("mapFiller").style.display = "block";
          }
        }
      );
    }

  } else {

    // No results currently showing, so just do the search.
    searchPart2();

  }

} // End search().


/**
 * Continue search after effect.
 */
```

```
function searchPart2() {

  // Show Please Wait.
  $("searchResults").style.display = "none";
  $("pleaseWait").style.display = "block";

  // Do search.
  var zipCode = $("zipCodeField").value;
  masher.doRequest(masher.googleURL,
    {
      "bq" : "%5blocation:@%22" + escape(zipCode) + "%22%2b50mi%5d",
      "alt" : "json-in-script",
      "callback" : "googleCallback"
    }
  );

} // End searchPart2().
```

First, before we go any further, notice the use of the `$()` function (`$` is a valid name for a JavaScript function, even if it looks a little unusual on its own). As you may recall from the previous chapter, this function comes from Prototype, which underlies script.aculo.us. This is a shorthand for `document.getElementById()`, although `$()` provides a few added capabilities, such as the possibility of getting multiple elements back. You'll see this throughout the code we're examining.

Moving on, this code does some resetting that puts the application into the proper state for a search. Most important, this means resetting the zoom buttons so none of them are selected, and changing the `currentlyDisplayedIndex` variable to –1, indicating no hotel information is currently being viewed. The next reset is to make sure no hotel information is showing. To do this, the list of hotels currently seen on the screen, if any, is iterated over. For each, the information `<div>` below the hotel in the list is hidden by setting its `display` style attribute to `none`.

After that, a check is done to see if any search results are currently showing. If they are, we have to hide both the search results and the map. To hide the search results, we collapse the area. Script.aculo.us provides a `BlindUp` effect that collapses, or shrinks, an element, just like rolling up window blinds. This code tells script.aculo.us to go ahead and collapse our search results:

```
new Effect.BlindUp("searchResults",
  {
    afterFinish : function() {
      // Now do the actual search.
      searchPart2();
    }
  }
);
```

We are also telling it that, upon completion, the `searchPart2()` function (which you will see shortly) should be called.

After that, we also have to hide the map if it is showing, and again, script.aculo.us provides a nice effect for us. The `Puff` effect expands an element and fades it at the same time, like a puff of smoke. Here's all that's required for the image with the ID `map` to be "puffed" out of existence:

```
new Effect.Puff("map",
  {
    afterFinish : function() {
      $("map").style.display = "none";
      $("mapFiller").style.display = "block";
    }
  }
);
```

When the effect completes, we hide the map image and show the filler image, just to be sure the cell doesn't collapse and we get a border back (which vanishes during the effect).

Now, if no search results where showing in the first place, then `searchPart2()` is simply called, and nothing needs to be hidden.

So what is the `searchPart2()` I've been talking about? It's nothing but a continuation of the `search()` function. This function first hides the search results `<div>`, and shows the Please Wait `<div>` in its place. Next, it grabs the ZIP code the user entered, and finally, it calls the `doRequest()` function of the `Masher` object, passing it the URL for the Google service and the parameters required to use that service. These parameters are `bq`, which defines the query we want the service to perform; `alt`, which tells the service we want JSON wrapped in a JavaScript call returned to us; and finally, `callback`, which specifies the name of the callback function that will be called. Note that the `bq` value is encoded as per the rules for the search service, and the ZIP code is escaped to make it safe for inclusion in the URL later.

The next step, naturally enough, is seeing what this `Masher` object is all about!

Writing Masher.js

In short, the `Masher` object, or more precisely, the `Masher` class, of which an instance is created at page load, is responsible for communicating with a JSON-based web API. It is written in a somewhat generic fashion so that other services can be accessed using it with a minimum of change. Figure 5-6 shows the rather simple UML diagram for this class.

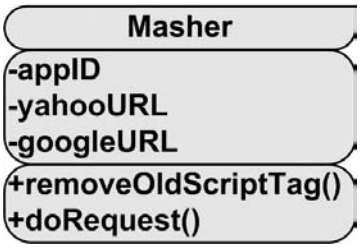


Figure 5-6. *UML diagram of the Masher class*

The appID field is the application ID needed to access Yahoo's APIs. The yahooURL field is the URL for the Yahoo Map Image service. Lastly, the googleURL field is the URL for the Google Base hotel search service.

The first method, removeOldScriptTag(), is a simple function that deals with removing a dynamically added <script> tag. This is done so that as we make calls to the services, we aren't building up memory usage as we add new tags. Listing 5-6 shows this method's code, as well as the rest of the Masher class.

Listing 5-6. *The Masher.js File*

```
/**
 * This class allows us to interact with JSON-based web services in the style
 * of Yahoo, that is, JSON wrapped in a function call.
 */
function Masher() {


  // Parameters for Yahoo! services.
  this.appID = "xxxxx";
  this.yahooURL = "http://api.local.yahoo.com/MapsService/V1/mapImage";

  // Parameters for Google services.
  this.googleURL = "http://www.google.com/base/feeds/snippets/-/hotels";


  /**
   * Removes an old script tag used to retrieve JSON.
   */
  this.removeOldScriptTag = function() {

    var scriptTag = $("jsonScriptTag");
      if(scriptTag) {
        scriptTag.parentNode.removeChild(scriptTag);
    }

  } // End removeOldScriptTag().


  /**
   * Perform an Ajax request using the dynamic script tag approach.
   *
   * @param inURL    The URL of the service.
   * @param inParams The parameters for the call.
   */
  this.doRequest = function(inURL, inParams) {
```

```
    // First, to avoid continually building up memory, remove any old script
    // tag out there.
    this.removeOldScriptTag();

    // Now build up a query string using the passed in parameters
    var queryString = "";
    for (param in inParams) {
      var paramVal = inParams[param];
      if (queryString == "") {
        queryString += "?";
      } else {
        queryString += "&";
      }
      queryString += param + "=" + paramVal;
    }

    // Now add a new script tag with the appropriate URL.
    var scriptTag = document.createElement("script");
    scriptTag.setAttribute("id", "jsonScriptTag");
    scriptTag.setAttribute("src", inURL + queryString);
    scriptTag.setAttribute("type", "text/javascript");
    var headTag = document.getElementsByTagName("head").item(0);
    headTag.appendChild(scriptTag);

  } // End doRequest().


} // End Masher.
```

As you saw in the `SearchFuncs.js` code, when an API is needed, it is called by using the `doRequest()` method in the `Masher` class. This function begins by making a call to `removeOldScriptTag()`, as mentioned, to be sure memory isn't continually being consumed. Next, a query string is built up based on the parameters passed in. For each, we add it to the query string. Note that the values are *not* escaped at this point because it is assumed any parameters passed in have already been escaped (this avoids potentially double-escaping something that shouldn't be). The ZIP code is the only user-entered value being passed, and you will recall that it is escaped in the `SearchFuncs.js` code, so no problem there.

Once the query string is built up, we have essentially the same code you've seen already for adding a `<script>` tag:

```
    var scriptTag = document.createElement("script");
    scriptTag.setAttribute("id", "jsonScriptTag");
    scriptTag.setAttribute("src", inURL + queryString);
    scriptTag.setAttribute("type", "text/javascript");
    var headTag = document.getElementsByTagName("head").item(0);
    headTag.appendChild(scriptTag);
```

Note that the ID set on this tag is the same as the one removed in `removeOldScriptTag()`. Therefore, we have only one dynamic `<script>` tag on the page at any given time. Otherwise, this code is the same as what you saw before, and I trust you remember how it works!

There isn't a whole lot to interacting with these services, as this class clearly illustrates. Only one piece remains: the callback functions that will be called when the response from the services returns. That's precisely where we're headed next!

Writing CallbackFuncs.js

As you've seen, when the Google or Yahoo APIs return their responses, they are in the form of a JavaScript function call with JSON passed as the argument to it (well, more precisely, the object resulting from the evaluation of the JSON). The callbacks, `googleCallback()` and `yahooCallback()`, are shown in Listing 5-7.

Listing 5-7. *The CallbackFuncs.js File*

```
/**
 * This is the function that is called when a Google service returns.
 *
 * @param inJSON The JSON object returned by the service.
 */
function googleCallback(inJSON) {

  var htmlOut = "";
  appState.hotels = new Array();

  // Iterate over the list of hotels.
  for (var i = 0; i < inJSON.feed.openSearch$itemsPerPage.$t; i++) {

    // Construct markup for the list for each hotel, including its information.
    var entry = inJSON.feed.entry[i];
    var hotel = new Hotel();
    hotel.name = entry.title.$t;
    hotel.location = entry.g$location.$t;
    hotel.description = entry.content.$t;
    appState.hotels.push(hotel);
    htmlOut += "<span onClick=\"" +
      "getMap('" + entry.g$location.$t + "');" +
      "showInfo(" + i + ");\" " +
      "onMouseOver=\"this.style.backgroundColor='#ffff00';" +
      "this.style.cursor='pointer';\" " +
      "onMouseOut=\"this.style.backgroundColor='';" +
      "this.style.cursor='';\"" +
      ">";
    htmlOut += entry.title.$t;
    htmlOut += "</span>";    htmlOut += "<div id=\"hotelInfo" + i + "\" style=\" ➥
display:none;\" ";
    htmlOut += "class=\"hotelInfo\">";
```

```
    htmlOut += "</div>";
    htmlOut += "<br><br>";

  }

  // Put the generated markup in the search results list div.
  $("searchResults").innerHTML = htmlOut;
  $("pleaseWait").style.display = "none";

  // Ask script.aculo.us to show the search results.
  new Effect.BlindDown("searchResults");

  // Set flags so we know what state the application is in.
  appState.searchResultsShowing = true;
  appState.mapShowing = false;

} // End googleCallback().


/**
 * This is the function that is called when a Yahoo service returns.
 *
 * @param inJSON The JSON object returned by the service.
 */
function yahooCallback(inJSON) {

  if (inJSON.Error) {
    var msg = "An error occurred retrieving map: ";
    if (inJSON.Error.Message) {
      msg += inJSON.Error.Message;
    }
    appState.mapShowing = false;
    $("map").src = "img/pixel_of_destiny.gif";
    alert(msg);
  } else {
    $("map").src = inJSON.ResultSet.Result;
  }

} // End yahooCallback().
```

The first important task the `googleCallback()` function handles is to clear any existing hotels that may be stored in the `ApplicationState` instance by assigning the `hotels` field to a new array. Once that's done, it's time to iterate over the hotels returned.

To do this iteration, we need to know how many items were returned, and that is found in the `feed.openSearch$itemsPerPage.$t` attribute of the JSON object. Once we have that value, we begin the loop. For each iteration, we grab the entry, which is a hotel, by accessing the `feed.entry` array, where we append an index value to the end based on the loop counter, so `feed.entry[i]`.

Once we have the entry, it's just a matter of constructing the HTML for the results list. Each entry is surrounded by a `<span>`, and it has some event handlers for when the user hovers over it (the yellow background with the pointer) as well as an `onClick` handler that loads the map for the entry and its extended information. The contents of the `<span>` are the hotel's name (accessed via the `entry.title.$t` attribute) and a `<div>` below it, where the extended information will go.

At the same time, we are populating a newly instantiated `Hotel` object with the hotel name (`entry.title.$t`), its location (`entry.g$location.$t`), and its description (`entry.content.$t`). This object is pushed onto the `hotels` array in `ApplicationState`.

Two things of interest are worth noting here. First, the description itself is not included in the markup, but is added dynamically later. I admit that I have no real reason for having done it this way other than to show that it's possible. Second, notice the `onClick` handler attached to the entry's `<span>`, which is responsible for retrieving the map for the hotel, via the `getMap()`call, and for showing its information, via the `showInfo()` call.

The Yahoo callback does even less than the Google one. Its sole job is to update the `src` attribute of the `<img>` with the ID of `map` to point to the URL returned by the Yahoo service. That's all there is to showing a map! However, there is a nasty downside to using the dynamic `<script>` approach to Ajax, and that's error handling.

Often, errors can be handled fairly easily, as is the case in the `yahooCallback()` function. If an `Error` attribute is found in the returned JSON, we can get the message and display it to the user. We also need to reset the `mapShowing` flag and be sure we show the blank "pixel of destiny" image again. However, this covers only one type of potential error. If a network failure occurs, for instance, and the response doesn't come back, there is no error handling you can do. The application will simply appear to not work.

Also, if the service returns an invalid data structure, that cannot be handled either. This is the case with some of the Yahoo functions. For instance, try the ZIP code 94505 and select any of the first few hotels. You'll notice the "Please wait, retrieving map" message appears, but no map ever does appear. If you are running in Firefox and open Firebug, you'll notice that the response from the service is invalid and causes a JavaScript error when it is evaluated. Unfortunately, we can do nothing about this. It's much like including a `.js` file that has a syntax error, but it's worse here, because you can't just correct it yourself. This is definitely a problem to be aware of with mashups, and more specifically, the dynamic `<script>` tag technique. There's no free lunch!

Writing MapFuncs.js

The `MapFuncs.js` file contains four functions that deal with the map or operations performed on the map, such as zooming in and out. Listing 5-8 shows the code in this file.

Listing 5-8. *The MapFuncs.js File*

```
/**
 * Retrieve map for address of selected hotel with Yahoo Maps.
 *
 * @param inLocation The address of the hotel to get a map for.
 * @param inZoom     The zoom level, 1-12, of the map.
 */
```

```
function getMap(inLocation, inZoom) {

  // The default zoom level is 6.
  if (!inZoom) {
    inZoom = 6;
  }

  // Show the Please Wait message while we request the map.
  $("map").src = "img/retrieving_map.gif";
  $("map").style.display = "block";
  $("mapFiller").style.display = "none";

  // Ask the masher to make the request for us.
  masher.doRequest(masher.yahooURL,
    {
      "appid" : masher.appID,
      "location" : escape(inLocation),
      "image_type" : "gif",
      "output" : "json",
      "width" : "520",
      "height" : "400",
      "zoom" : inZoom,
      "callback" : "yahooCallback"
    }
  );

  // Set state and reset the zoom buttons, and highlight the current zoom
  // level.
  appState.mapShowing = true;
  resetZoomButtons();
  highlightZoomButton(inZoom);

} // End getMap().


/**
 * Zoom the map according to the zoom button clicked.
 *
 * @param inZoom The zoom level, 1-12, to zoom the map to.
 */
function zoomMap(inZoom) {

  // Obviously this only does something if a map is showing.
  if (appState.mapShowing) {
    var hotel = appState.hotels[appState.currentlyDisplayedIndex];
    getMap(hotel.location, inZoom);
  }
```

```
} // End zoomMap().


/**
 * Reset all the zoom buttons so none are highlighted.
 */
function resetZoomButtons() {

  for (var i = 1; i < 13; i++) {
    $("zoomButton" + i).style.fontSize = "10pt"
  }

} // End resetZoomButtons().


/**
 * Highlight the specified zoom button.
 *
 * @param inZoom The zoom level, 1-12, of the button to highlight.
 */
function highlightZoomButton(inZoom) {

  $("zoomButton" + inZoom).style.fontSize = "16pt"

} // End highlightZoomButton().
```

The first function, `getMap()`, is called when the user clicks one of the hotel links in the search results. It is also called when one of the zoom buttons is clicked. Because of this dual purpose, it accepts two arguments: `inLocation` and `inZoom`. `inLocation`, as you would expect, is the location for which we want a map. This correlates to the `location` field of a `Hotel` object. `inZoom` is the zoom level. Since Yahoo is generating the map, we need to tell it what level of zoom we want. However, when the map is first retrieved in response to a hotel being clicked, the `inZoom` argument is not passed in. So, the first thing you see in this code is a check of whether `inZoom` was passed in. If not, that's where the default zoom level of 6 is set.

After that are three lines of code that are responsible for showing the "Please Wait" message in the map area. It's just a matter of pointing the `map <img>` tag to the Please Wait image, showing it, and hiding the `mapFiller`.

Then we ask the `Masher` instance to get the map. For this particular API call, we need to pass the `appid` that we obtained when we registered for Yahoo's service. Obviously we need to pass the `location` as well, and we also need to specify that we want a GIF back (the `image_type` parameter). We specify that we want our output to be JSON (via the `output` parameter) wrapped in a JavaScript function call, which is specified with the `callback` parameter. We want a map that fits in our map area, which is 520-by-400 pixels in size, so we specify the appropriate `width` and `height` parameters. Lastly, we pass the zoom level that we want via the appropriately named `zoom` parameter.

A few last details need to be taken care of, namely setting the flag to indicate the map is showing, resetting the zoom buttons to the default states (no current zoom button is bigger than the rest), and then setting the button for the zoom level to current.

Once the API call returns, we'll wind up in `yahooCallback()`, which you've already seen.

The `zoomMap()` function is called when a map zoom button is clicked. It does a simple check to be sure the map is showing; otherwise, nothing should happen when a button is clicked. If a map is showing, `zoomMap()` gets the `Hotel` object corresponding to the hotel that is currently being viewed and uses that to pass the location and the new zoom level, which is the value passed in, to the `getMap()` function.

The `resetZoomButtons()` function that follows performs the simple task of resetting all the zoom buttons to the default font size, effectively making it so that none of the buttons indicates the current zoom level. This is just a loop through the 12 buttons, constructing the appropriate DOM ID for each, and setting the `fontSize` style attribute on it.[4]

Last is the `highlightZoomButton()` function. This is called when a user clicks one of the buttons, and its task is to make the button "current," which means to have a bigger font size. This is again just a straight manipulation of the `fontSize` style attribute.

Writing MiscFuncs.js

Now we come to the final source file in this project, `MiscFuncs.js`. This file contains a single function, but it's a pretty important one: `showInfo()` is used to display the extended information for a clicked hotel. Listing 5-9 shows this code.

Listing 5-9. *The MiscFuncs.js File*

```
/**
 * Function to show extended hotel information.
 *
 * @param inIndex Index into the array of hotels in appState.
 */
function showInfo(inIndex) {

  // Trivial rejection: are we already showing the requested hotel?
  if (inIndex == appState.currentlyDisplayedIndex) {
    return;
  }

  // Shrink the information for the currently showing hotel.
  if (appState.currentlyDisplayedIndex != -1) {
    new Effect.Shrink("hotelInfo" +
      appState.currentlyDisplayedIndex,{duration:1.0});
  }
```

4. This certainly could have been done by switching the button to a different style class, and in many ways that is a better approach. But I like to show some of the alternate ways you can accomplish the same thing, and certainly direct manipulation of the style attributes is one way.

```
  // Update application state and insert the new hotel information.
  appState.currentlyDisplayedIndex = inIndex;
  var hotel = appState.hotels[inIndex];
  var htmlOut = "<br>" + hotel.location + "<br><br>";
  htmlOut += hotel.description;
  $("hotelInfo" + inIndex).innerHTML = htmlOut;

  // And finally, have the new info "grow" into view.
  new Effect.Grow("hotelInfo" + inIndex,{duration:1.0});

} // End showInfo().
```

This function takes as an argument the index into the `hotels` array of the hotel for which to get information. So, first is a quick check to see if the hotel that was just clicked already has information showing. If it does, we're finished—bug outta here!

Assuming it's a different hotel, it next checks to see if *any* hotel has information showing, which would mean the `currentDisplayedIndex` field in `appState` has a value other than –1. In that case, we ask script.aculo.us to shrink that information out of view using the `Shrink` effect.

The code then records the index that is now going to be shown, grabs the appropriate `Hotel` object, and builds markup using its `location` and `description`. Once the markup is built, it is inserted into the `<div>` directly below the clicked hotel, the ID of which is `hotelInfo`, followed by the index. Lastly, we ask script.aculo.us to expand this new information into view using the `Grow` effect.

Note that the `Shrink` and `Grow` effects will occur simultaneously because, when the new `Shrink` effect is instantiated, a timer is started to perform that effect, but the code between that and the new `Grow` effect instantiation will occur before the `Shrink` effect finishes. There, we'll have two timers running, one for each effect, and they'll appear to happen at roughly the same times. This is by design, as it makes the whole transition look a lot cooler. Of course, script.aculo.us handles all those messy details for us.

And, at long last, we've come to the end! I hope you've enjoyed seeing a mashup in action, and appreciate the APIs Yahoo and Google, among many others, are providing for us to use as building blocks in our own applications.

Suggested Exercises

With mashups, you can just continue to add features to your heart's content, as long as you can find a suitable API! While I leave the hunt for APIs to you, here are a small handful of suggestions you could play around with:

- Add the ability to get a weather report for a particular day, or range of days. This would be useful if you're trying to book a vacation!
- Add the ability to look for restaurants instead of hotels (hint: Google provides this capability as well).
- Extend the map functionality. You can simply look at the features Yahoo Maps provides and duplicate them, since most, if not all, of the functionality is exposed through the APIs.

Summary

In this chapter, we got into one of the most popular buzzwords being tossed around today: mashups. You saw how to get around the same-domain restriction typical of most Ajax implementations. We covered using a number of public APIs, how they present themselves to the developer, and how an application can interact with them. You also saw how the script.aculo.us library can provide some pretty nifty little UI eye candy with a minimum of code and fuss.

All in all, we managed to create a pretty useful little application without much effort, which is, of course, the goal of a mashup. Along the way, you saw a few neat JavaScript tricks in the process, expanding your mental toolbox just a bit further!

CHAPTER 6

Don't Just Live in the Moment: Client-Side Persistence

For applications, two types of data storage are available: *persistent* and *nonpersistent*, or transient. Persistent storage is any storage mechanism that provides a place for data to be saved between program executions, and often for an indefinite period of time (until explicitly removed from storage). Transient storage is any storage mechanism where the data lives only as long as the program is actually executing (or for some short time thereafter). A database is generally considered a persistent storage mechanism, whereas RAM clearly is not. Writing to a hard drive is usually persistent as well, while session memory generally is not. The term *durable* is also often used to describe persistent storage media.

In web applications, storing state on the server—whether in a database or in a session—is pretty much expected of most applications. But what happens when you don't have a persistence mechanism on the server, or possibly don't want one for some reason? What's the alternative?

When discussing persistence in a pure JavaScript-based client-side application, there is a very short list of possible storage mechanisms.[1] The ubiquitous cookie—small bits of information stored on the client on a per-domain basis—is one. Another is a facility now available on most browsers, dubbed *local shared objects* (or Flash shared objects, or even Flash cookies, depending on the documentation you read). This is a storage mechanism provided by the Adobe Flash browser plug-in.

Local shared objects are gaining quite a bit of popularity, and we'll put that approach to use in this chapter's project, which is a simple contact manager. Perhaps most interesting though is the Dojo library we will use to implement this client-side persistence, which makes our lives so much easier. So, on with the show.

Contact Manager Requirements and Goals

In this chapter, we will build a Contact Manager application that runs entirely on the client side. This is handy because it means it can run on virtually any PC, without requiring a network connection (although, conversely, it means that the contact data can't easily be shared among

1. Java applets or ActiveX controls are other options, but they are generally thought of as a whole other class of possibilities, because they can, in a sense, be seen as extensions to the browser itself (certainly, ActiveX controls are meant to be that, but applets can also be viewed in that light). In either case, they require something more than HTML and JavaScript, and that in and of itself places them in a different category.

multiple machines). Clearly, persistence will come into play here, since it would be quite a useless application if we couldn't actually store contacts.

Let's list some of the key things this project will accomplish and some of the features we'll seek to implement.

- Each contact should include a good amount of data. Along with the basics of name, phone number, and email address, we'll also allow for a fair amount of extended information, such as birthday, spouse's name, children's names, and so on. However, we want to make these items as free-form as possible, so users can use the different data fields however they like.
- We'll implement the typical alphabet selector tabs to make it easier to find contacts.
- We'll store our contacts with local shared objects.
- We'll use Dojo to provide some widgets coolness to make the interface fun and attractive. We'll also use Dojo to deal with the underlying details of working with local shared objects.

Now that we have some goals in mind, let's take a look at the library we'll use for this project.

Dojo Features

To help build the Contact Manager application, we'll use the very popular Dojo toolkit. Chapter 2 briefly introduced Dojo, but let's look at it in just a little more detail.

On its front page, Dojo (`http://dojotoolkit.org`) explains what it is, and I see no need to try to paraphrase, so here's a direct quote:

> *Dojo is the Open Source JavaScript toolkit that helps you build serious applications in less time. It fills in the gaps where JavaScript and browsers don't go quite far enough, and gives you powerful, portable, lightweight, and tested tools for constructing dynamic interfaces. Dojo lets you prototype interactive widgets quickly, animate transitions, and build Ajax requests with the most powerful and easiest to use abstractions available. These capabilities are built on top of a lightweight packaging system, so you never have to figure out which order to request script files in again. Dojo's package system and optional build tools help you develop quickly and optimize transparently.*
>
> *Dojo also packs an easy to use widget system. From prototype to deployment, Dojo widgets are HTML and CSS all the way. Best of all, since Dojo is portable JavaScript to the core, your widgets can be portable between HTML, SVG, and whatever else comes down the pike. The web is changing, and Dojo can help you stay ahead.*
>
> *Dojo makes professional web development better, easier, and faster. In that order.*

Yes, that does about sum it up! Dojo has been gaining a following of late, and has even been integrated into some popular frameworks, such as WebWork from OpenSymphony (now Struts 2 from Apache, `http://www.opensymphony.com/webwork` or `http://struts.apache.org`).

Dojo is a rather large library, containing a myriad of packages and features. Dojo is not just about Ajax, like some other libraries out there. It provides some functions that extend JavaScript itself, as well as general utility code for JavaScript applications.

However, Dojo does have a downside: it is still really in its infancy. One look at the online documentation, and you'll realize that using Dojo means that, to a large degree, you'll be fending for yourself. Many of the packages do not, as of the time of this writing, appear to have any documentation at all. Examples are a bit thin at this point, and the planned features are not fully baked just yet. However, Dojo has improved leaps and bounds in this department from just a few months ago. You can begin to have a certain comfort level with Dojo in this regard if you are thinking of using it. And the Dojo mailing list is *very* active, with a large number of truly helpful people. Any good open source project is defined by the nature and activity level of its community, and Dojo gets high marks in this regard.

In this chapter, we'll be using only two (a UI widget and the storage capabilities) of Dojo's many packages. Table 6-1 shows some of the packages Dojo offers. Note that this is just a small sampling, so you will want to explore what else it offers as your time allows.

Table 6-1. *Some Dojo Packages*

Package	Description
`dojo.lang`	Utility routines to make JavaScript easier to use. Contains a number of functions for manipulating JavaScript objects, testing data types, and so on.
`dojo.string`	String manipulation functions, including `trim()`, `trimStart()`, `escape()`, and so on.
`dojo.logging`	JavaScript logging.
`dojo.profile`	JavaScript code profiling.
`dojo.validate`	Data validation functions, such as `isNumber()`, `isText()`, `isValidDate()`, and so on.
`dojo.crypto`	Cryptographic routines.
`dojo.storage`	Code that implements a durable client-side cache using Flash's cookie mechanism. This effectively gives you a client-side analogy to the `HttpSession` object on the server.
`dojo.widget`	Various highly cool GUI widgets (such as buttons, dialog windows, photo slideshow, and so on).
`dojo.Collections`	Various data structures like `Dictionary`, `ArrayList`, `Set`, and so on.

To begin using Dojo, you have a couple of options. Dojo comes in a number of "editions." So, if you're interested only in the Ajax functionality, you can download just the IO edition. If you are interested only in GUI widgets, you can download the widget edition. There is also a so-called "kitchen sink" edition that contains everything Dojo offers. For this chapter's project, I used the minimal edition, but we'll discuss that further in just a bit.

After you have downloaded the proper edition, all you need to do is a typical JavaScript import of the `dojo.js` file, like so:

```
<script src="js/dojo.js"></script>
```

After that, you're all set. Dojo also offers an "import" feature. So, for instance, if you have downloaded the IO edition and later decide you want to use the event system, you can do this:

```
<script type="text/javascript">
    dojo.require("dojo.event.*");
</script>
```

Dojo will then take care of loading all the dependencies for you, and you will be good to go. You can do this for any of the features you want, at any time. In other words, you don't need to have a bunch of imports at the top of the page for each package you want to use; you can instead treat the Dojo import just like a Java import, and let Dojo handle all the details (although the main import of `dojo.js` *is* still required).

Dojo and Cookies

Working with cookies is quite simple, as you saw in Chapter 3 when we built the `jscript.storage` package of functions. Dojo offers very similar functionality, with perhaps a bit more capability. To see some of the functions Dojo provides, check out the online API reference at `http://dojotoolkit.org/api`, which has come a long way in a short time and has begun to make Dojo quite a bit nicer to use. Figure 6-1 shows this page.

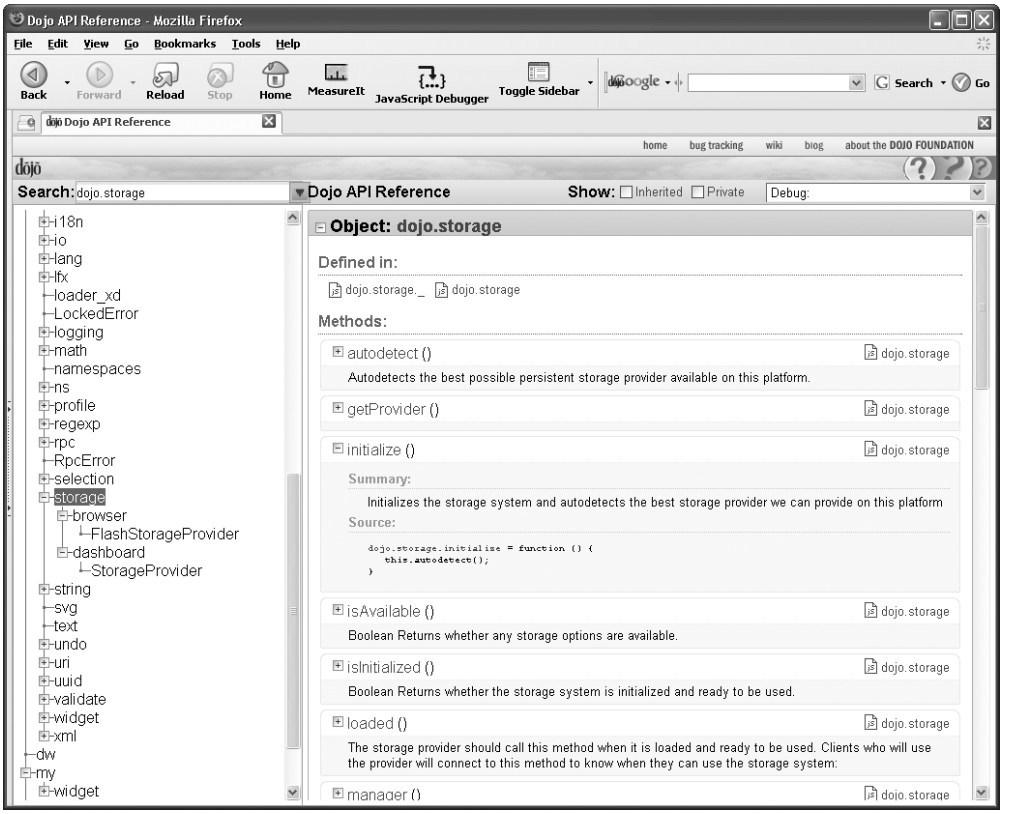


Figure 6-1. *The Dojo API online reference*

You'll find the cookie functions in the `dojo.io.cookie` package (which might seem a little odd, since there is a `dojo.storage` package).

As an example, to set a cookie in Dojo, just do the following:

```
dojo.io.cookie.setCookie("cookieName", "cookieValue");
```

That doesn't look a whole lot different than the `setCookie()` function we developed in Chapter 3. However, Dojo adds four more (optional) parameters to the end of this function:

- `days`: Determines how many days this cookie will live before it expires.
- `path`: Sets the path the cookie is for (what path within the domain the cookie is valid and will be returned for).
- `domain`: Sets the domain the cookie is for (what domain the cookie will be returned for).
- `secure`: Determines if the cookie is secure (when `true`, the cookie will be returned only over an SSL channel).

As you might expect, Dojo also provides both `getCookie(name)` and `deleteCookie(name)` functions. It also provides two slightly more advanced functions: `setObjectCookie()` and `getObjectCookie()`. These functions deal with cookies as object/value pairs as the cookie value instead of a simple data type. These are nice shortcuts that save you from having to write code to parse your objects before writing them out.

Cookies are rather ubiquitous and are used day in and day out in applications all over the place. They are simple, quick, and quite sufficient for a great many tasks. They are, however, not without their limitations:

- Each domain is limited to a maximum of 20 cookies. You may find some browsers that allow for more, but the HTTP spec states 20 as the required minimum. So it is best to assume that is actually the maximum to ensure your code won't break in some browsers, because most treat 20 as the upper limit.
- Each cookie is limited to 4kb maximum size. Some quick math tells us we have a maximum of 80kb per domain on the client in cookies. Aside from 80kb not being enough for many tasks, the fact that it must be divided among 20 cookies, and you'll have to write that code yourself, makes cookies less than desirable for many purposes.

Fortunately, the folks at Adobe have a ready solution for us, and as you might expect, Dojo is there to take advantage of it and make our lives as developers better. So, we won't be using cookies in this chapter's project. We'll get to the storage mechanism that we'll use shortly, but first, let's look at a few other Dojo features.

Dojo Widgets and Event System

The Contact Manager application will make use of one of the UI widgets Dojo provides, called the fisheye list. If you've played with the application already, you'll recognize the fisheye list as the icons across the top that expand and contract as you mouse over them, similar to the Mac launch bar effect, if you are familiar with Apple's operating system. The nice thing about the Dojo widgets is that, because they are built with a common widget framework, they all, by and large, are used in the same way, so seeing one gives you a pretty good idea of how to use the

others. The details of using Dojo widgets can get fairly verbose, so I'll explain them as we dissect the application. Seeing their use in context is clearer in this instance.

We'll also use the event system in Dojo, which allows us to attach functions to virtually any event that can occur, not just UI events like mouse clicks and such. Dojo offers an aspect-oriented event system, whereby you can have a function called any time another function is called, for instance. The event system Dojo offers is extensive, and we'll only be scratching the surface in this application, but you'll start to get an idea of what's possible. Once again, I'll explain the details in the dissection process to come.

The real star of the show though, and frankly the main focus of this chapter, is the Dojo storage system and local shared objects, so let's dive into that now!

Local Shared Objects and the Dojo Storage System

Local shared objects (also called Flash shared objects) are somewhat akin to cookies, but are a mechanism of the Adobe Flash plug-in. They were introduced with Flash MX, so earlier versions (prior to version 6) will not support them. You work with them in much the same way you do cookies, but the size and number-per-domain limitations that exist for cookies do not apply with local shared objects. You can, for all intents and purposes, store as much data as you like in local shared objects, at least until your users' hard drives fill up.

Flash actually has a larger installed base than even IE: about 97% at the time of this writing. This means that any concerns you may have had about needing the Flash plug-in for client-side persistence in the past can probably be thrown out the door—the plug-in is more likely to be available than many other things, perhaps even JavaScript itself. Flash is even available for many traditionally limited devices, such as PDAs and cell phones (although, unfortunately, you may find that local shared objects are not available on some of those devices).

However, there would appear to be a speed bump in our way, and that is the simple fact that local shared objects are for use in Flash movies, not in JavaScript. How do we overcome that? By having Dojo handle it for us, of course.

We could write the code ourselves, but that would require writing a Flash movie or two that exposes a scripting interface, which we could then interact with from JavaScript. If this makes your brain hurt, join the club. While I've worked with Flash a bit, I'm far from any sort of expert, and it would certainly take a fair amount of time and effort to write these components myself. But why go through all that trouble anyway, when the folks working on Dojo have already handled all the difficult bits and given us a simple way to utilize it? Sometimes, being lazy is actually a *good* thing!

Dojo provides a storage package that is billed as a pluggable mechanism for client-side persistence. Its architecture is based on the concept of a storage manager and any number of storage providers. Each provider can persist data via any method it wants, but the client application writes to a common interface that all providers implement, thereby allowing the developer to swap between various storage mechanisms at the drop of a hat without any change to their code. It's very cool.

At the time of this writing, the `dojo.storage` package provides only a single storage provider,[2] and that is one that deals with shared objects. The `dojo.storage` package is a wonderful creation

2. Cookie functions would obviously be a logical fit here as well. That's why there has been talk of a cookie provider (there may even be one by the time you read this). One could conceive of an ActiveX storage provider that writes directly to a SQL Server database as one example, and yet the application that utilized `dojo.storage` wouldn't need to know about the details of that at all.

that offers a great deal of power to developers. It's essentially a simple architecture, as shown in Figure 6-2, which again proves that simplicity is usually the way to go.

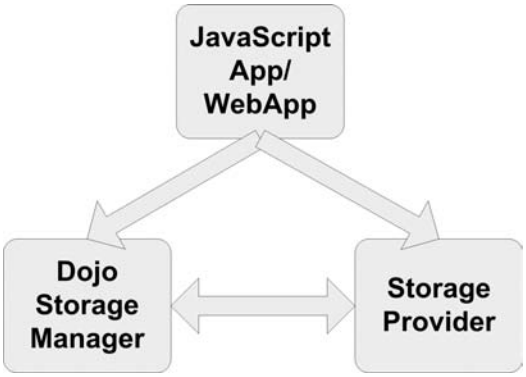


Figure 6-2. *The dojo.storage package architecture*

As you'll see when we examine the code behind the application in this chapter, the primary interaction your code will have with `dojo.storage` is via the storage manager and storage provider interfaces, which are shown in Figure 6-3 and Figure 6-4, respectively. Once again, there isn't a whole lot to them, and, in fact, for the purposes of the application in this chapter, we won't use more than about half of what these interfaces offer.

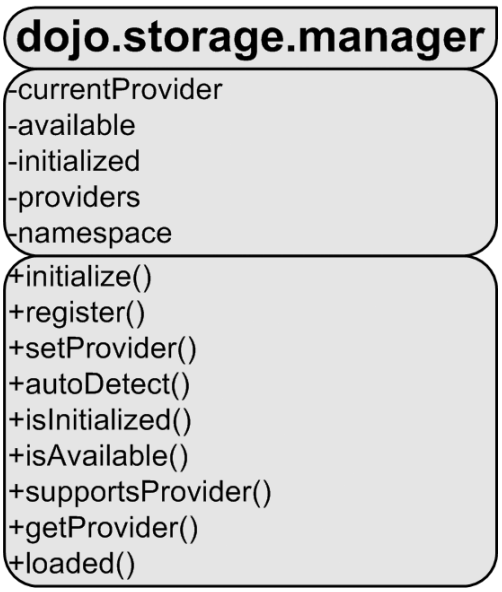


Figure 6-3. *The Dojo storage manager interface*

Figure 6-4. *The Dojo storage provider interface*

As mentioned, the Dojo developers have dealt with the details of implementing the Flash movies that can be interacted with via JavaScript. Think of those movies as something like a Data Access Object (DAO), responsible for the actual storage implementation. The API it exposes is wrapped by the `dojo.storage` API in a sense. So, when you call the `put()` method of the `StorageProvider` class (well, the class implementing that API anyway), it calls some function in the Flash movie that does the actual work of saving to the stored objects. It's really a rather elegant solution.

So, now that you have an idea how this storage mechanism works in Dojo, let's take a look at the application that will use it.

A Preview of the Contact Manager

Figure 6-5 shows the application we will build in this chapter. I certainly recommend spending a few minutes playing with it before proceeding. I think you'll find it to be a fairly clean and useful little contact list. It won't make anyone on the Microsoft Outlook team lose any sleep, but it's not bad by any stretch.

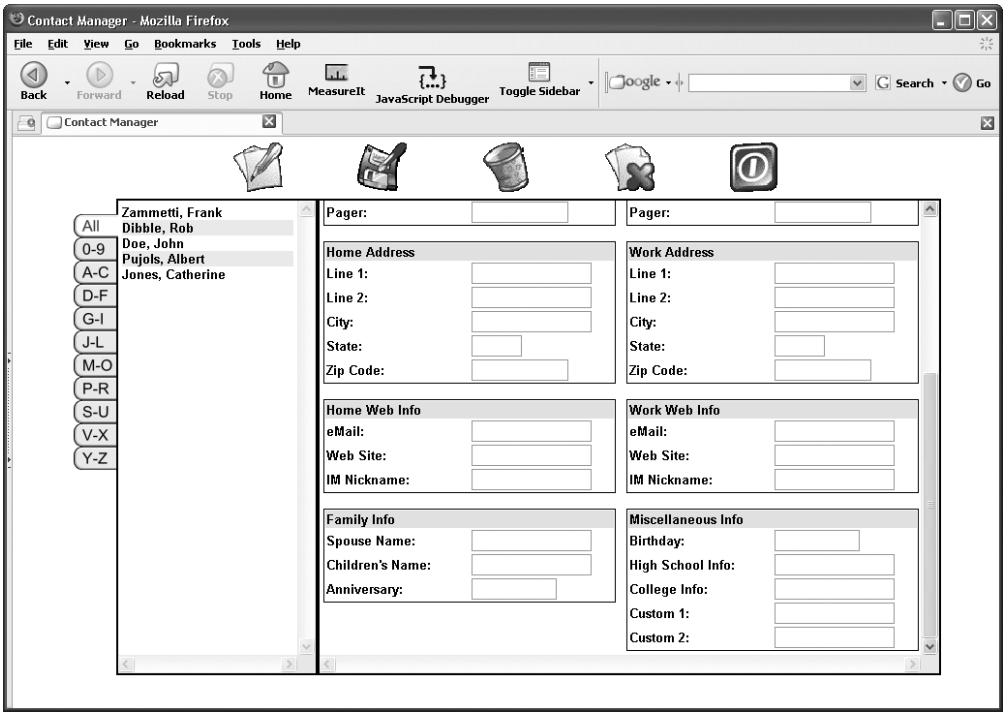


Figure 6-5. *JS Contact Manager: Outlook may not have to worry, but it ain't bad!*

There is a good chance when you first run the application that you will see a pop-up dialog box generated by Flash that looks like Figure 6-6.

Figure 6-6. *Security warning seen when you first run the application*

If you don't see this pop-up, then great; count yourself lucky and just move on. If you see it (which is likely), this is a result of some security precautions in place in the Flash plug-in. You need to tell the plug-in that the folder on your local file system in which you are running the application is allowed and you shouldn't be asked about it. The vexing thing about this is that you will need Internet connectivity to change the setting. Believe it or not, the setting dialog box that you need is a web page on the Adobe web site! Figure 6-7 shows this page.

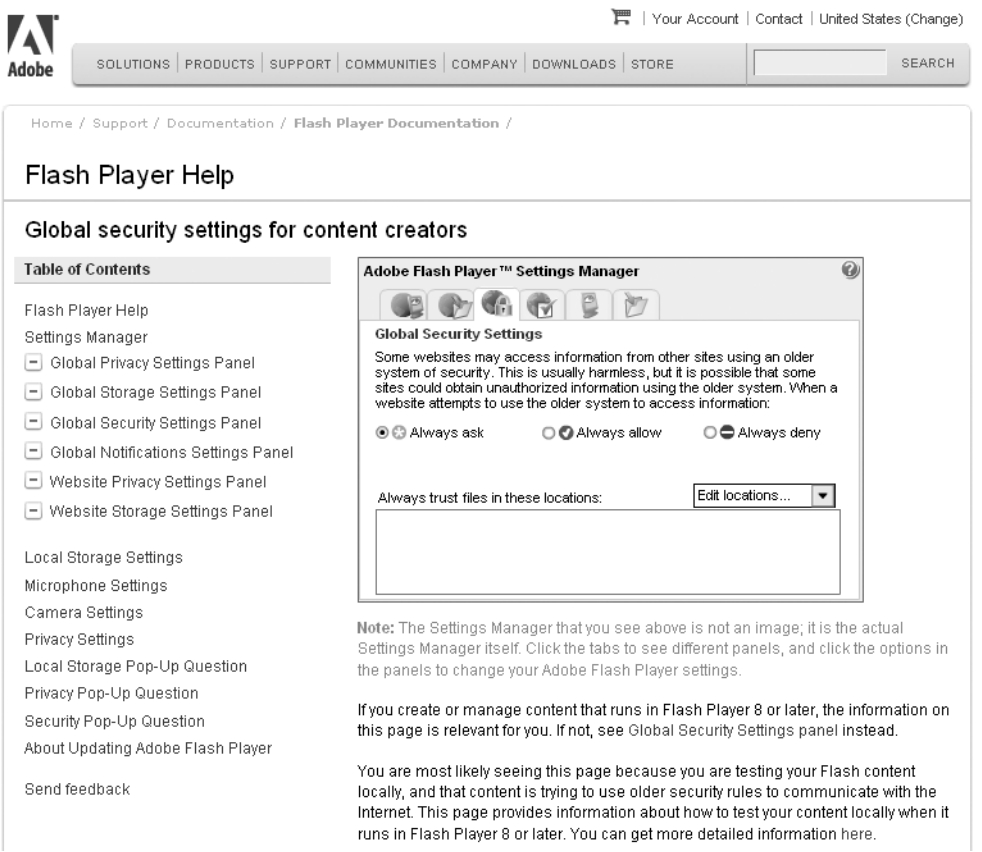


Figure 6-7. *The Flash security setting web page*

I can only assume there is some good reason for doing it this way and that Adobe knows that reason, but I sure don't! In any case, when you click the Settings button in the security warning dialog box, you will, assuming you are connected to the Internet, be directed to this page. Here, you will need to click the Edit Locations box and select Add Location. Then browse for the folder from which you are running the application. That should do the trick. I also select the radio button next to Always Allow. This probably isn't necessary, but you may want to do it as well, just to be sure. Once you make these changes, close that page, reload the application, and you should be good to go.

Dissecting the Contact Manager Solution

As we typically do with the projects in this book, let's first get a feel for the lay of the land, so to speak, and see what the directory structure looks like. Take a look at Figure 6-8.

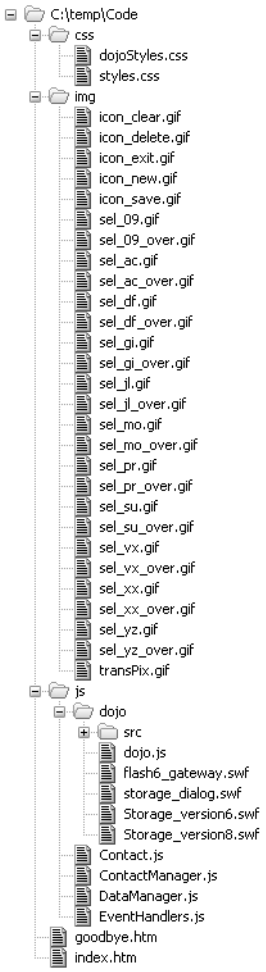


Figure 6-8. *JS Contact Manager's directory structure*

In the root directory are two files:

- `index.htm`, really the heart and soul of the application, as you'll soon see
- `goodbye.htm`, a simple page that is shown when the user exits the application

Under the root are a couple of directories, starting with the `css` directory. As is typical, the `css` directory is where we keep our style sheets. In this case, we have two:

- `styles.css`, which is the main style sheet for the application
- `dojoStyles.css`, which contains styles used specifically for Dojo widgets

The next directory is `img`, which, obviously enough, contains images used throughout the application. The images are named in a pretty obvious way, so there's no need to run through them all.

Last is the `js` directory, which, of course, is where the JavaScript lives. Four `.js` files are present:

- `Contact.js`, which defines a class representing a contact
- `ContactManager.js`, which contains the main application code; it's kind of a traffic cop, more or less, calling the other classes as required
- `DataManager.js`, which contains the code that actually deals with storing and retrieving our contacts
- `EventHandlers.js`, which contains the UI event handlers that make our UI work

Also in the `js` directory you'll find a subdirectory named `dojo`. This is where the Dojo library lives. Dojo comes in a number of editions, each basically baking certain parts of Dojo into a single `.js` file. Alternatively, you can use the minimal edition, which means that only the core Dojo code will be in `dojo.js`, and any other parts of Dojo you use will be imported as needed. This is the edition we'll use because it reduces the up-front loading time required and loads the various parts of Dojo as needed. This also, to my way of thinking, makes it less likely that we'll run into any problems for no other reason than less code in memory means less that can break.

Within the `dojo` directory, you'll find a number of subdirectories containing other Dojo package source files, as well as HTML and CSS resources, and anything else Dojo needs to function. While this application uses only a small portion of Dojo, having it all available means that as you want to extend the application and use more of Dojo, it is all there, ready for you.

And now, without further delay, let's get to some code.

Writing styles.css

Listing 6-1 shows the `styles.css` file, which is the main style sheet that defines, for the most part, all of the visual styling of the application.

Listing 6-1. *The styles.css File*

```
/* Generic style applied to all elements */
* {
  font-family      : arial;
  font-size        : 10pt;
  font-weight      : bold;
}

/* Style for body of document */
body {
  margin           : 0px;
}

/* Style for spacer between data boxes */
.cssSpacer {
  height           : 64px;
}
```

```
/* Non-hover state of contacts in the contact list */
.cssContactListNormal {
  background-color : #ffffff;
}

/* Non-hover state of contacts in the contact list, alternate row striping */
.cssContactListAlternate {
  background-color : #eaeaea;
}

/* Hover state of contacts in the contact list */
.cssContactListOver {
  background-color : #ffffa0;
  cursor           : pointer;
}

/* Style for main container box */
.cssMain {
  width            : 100%;
  height           : 580px;
  z-index          : 0;
}

/* Style of the initializing message seen at startup */
.cssInitializing {
  width            : 920px;
  height           : 2000px;
  z-index          : 1000;
}

/* Style of selector tabs */
.cssTab {
  position         : relative;
  left             : 2px;
  _left            : 4px; /* Style for IE */
}

/* Style of the contact list container */
.cssContactList {
  padding          : 4px;
  height           : 480px;
  border           : 2px solid #000000;
  overflow         : scroll;
}
```

```
/* Style of the box surrounding the data boxes */
.cssMainOuter {
  padding          : 4px;
  height           : 480px;
  border           : 2px solid #000000;
  overflow         : scroll;
}

/* Style of a data box */
.cssDataBox {
  width            : 100%;
  border           : 1px solid #000000;
}

/* Style for the header of a data box */
.cssDataBoxHeader {
  background-color : #e0e0f0;
}

/* Style for textboxes */
.cssTextbox {
  border : 1px solid #7f9db9;
}
```

The asterisk selector, as you've seen in other projects throughout this book, is here again employed to allow us to style everything on the page in one fell swoop.

Most of the style sheet is pretty self-explanatory, but one trick to point out is in the `cssTab` selector. This style is used to position the alphabetic selector tabs on the left side. To make the effect of one of them being the current tab work properly required that the tab overlap the border of the contact list box by a few pixels. Unfortunately, the number of pixels required didn't seem be consistent between IE and Firefox, so we use a little trick to allow for the difference.

When Firefox encounters an attribute name that begins with an underscore character, it ignores it. IE, on the other hand, ignores just the underscore, acting as if it were not there. So, what essentially happens here is that in both browsers, the first value of `2px` for the `left` attribute is set. Then when the `_left` attribute is encountered, Firefox ignores it, but IE strips the underscore and treats it as if the `left` attribute were set again, overriding the `2px` value with `4px`. In this way, you can deal with the style sheet differences that sometimes come up between IE and Firefox, without having to write branching code to deal with it or use alternate style sheets for each browser.

■ **Note** If you find yourself using the attribute name with a preceding underscore trick a lot, you probably want to rethink the way you're styling elements, because you may be doing things in a way that is too browser-specific. But here and there and every now and again, this is a good trick to know. Also be aware that in the future, one or both browsers could change this behavior, effectively breaking any page that uses it. This is just something to keep in mind.

Writing dojoStyles.css

`dojoStyles.css` is a style sheet containing just a small number of selectors that override default styles in Dojo. Listing 6-2 shows this rather diminutive source. Whoever said size doesn't matter must have been talking about this style sheet.

Listing 6-2. *The dojoStyles.css File*

```
/* Style for the fisheye listbar */
.dojoHtmlFisheyeListBar {
  margin           : 0 auto;
  text-align       : center;
}

/* Style for the fisheye container */
.outerbar {
  text-align       : center;
  position         : absolute;
  left             : 0px;
  top              : 0px;
  width            : 100%;
}
```

Although small, this style sheet provides some important styling. Without it, the way-cool fisheye icons wouldn't look quite right.

Writing index.htm

`index.htm` is where we find the bulk of the source that makes up the Contact Manager application. It is mostly just markup, with a sprinkle of script. Let's begin by taking a look at part of the `<head>` section where the style sheet and JavaScript imports are, as shown in Listing 6-3.

Listing 6-3. *Style Sheet and JavaScript Imports in index.htm*

```
<link rel="StyleSheet" href="css/dojoStyles.css" type="text/css">
<link rel="StyleSheet" href="css/styles.css" type="text/css">

<script type="text/javascript">
  var djConfig = {
    baseScriptUri : "js/dojo/",
    isDebug : true
  };
</script>
<script type="text/javascript" src="js/dojo/dojo.js"></script>
<script language="JavaScript" type="text/javascript">
  dojo.require("dojo.widget.FisheyeList");
  dojo.require("dojo.storage.*");
</script>
```

```
<script type="text/javascript" src="js/Contact.js"></script>
<script type="text/javascript" src="js/EventHandlers.js"></script>
<script type="text/javascript" src="js/DataManager.js"></script>
<script type="text/javascript" src="js/ContactManager.js"></script>
```

First, we import our two style sheets. Next comes a small block of JavaScript that sets up some Dojo properties. The `djConfig` variable is an associative array that Dojo looks for when it starts up (which means this code must come *before* the Dojo imports), and it contains a number of options for various Dojo settings. In this case, two are important:

- `baseScriptUri` defines the beginning of the path where all Dojo resources can be found. These include items such as source files to be imported, images for widgets, style sheets, and so on. In this case, we have Dojo installed in `js/dojo`, so that is the appropriate value for this attribute.
- `isDebug` determines if Dojo should output error messages.

A number of other options are available in `djConfig` (a few minutes with Google will reveal them), but for our purposes here, only these two are important.

Following that block is the import of the main part of Dojo itself. Immediately following that is a section with a series of `dojo.require()` function calls. Dojo is organized in a package structure, just as we built in Chapter 3. Dojo also implements the idea of importing small portions of it as required, and even offers wildcard (`.*`) import capabilities. What's more, if you import something that requires something else that you haven't imported, Dojo will import the dependencies for you.

Following the Dojo imports are four more plain JavaScript imports, which bring in the code that makes up the Contact Manager application. We will be looking at each in detail shortly.

Adding Bootstrap Code

At the end of the `<head>` section is a `<script>` block that contains what is essentially bootstrap code to get the application going. This code is shown in Listing 6-4.

Listing 6-4. *Bootstrap JavaScript in <head> of index.htm*

```
<script>

  // Shorthand function to get a reference to a DOM element.
  function $(inID) {
    return document.getElementById(inID);
  } // End $().

  // The contactManager instance that is the core of this application.
  var contactManager = new ContactManager();

  // Connect init() function in ContactManager to onLoad event.
  dojo.event.connect(window, "onload", contactManager.init);

</script>
```

The first thing you see is a function named `$`, which you've also seen in the projects in previous chapters. As you know by now, this is a function that wraps the ubiquitous `document.getElementById()` function call. This saves us some typing as we develop and makes for slightly cleaner looking code.

Following that is the instantiation of the `ContactManager` class, which is basically the core code of the application (you'll see this in just a bit).

Finally, we find the line:

```
dojo.event.connect(window, "onload", contactManager.init);
```

One thing that bites new Dojo developers (including me) is the fact that Dojo takes over the `onLoad` event and overwrites anything you may have put there yourself. This means that we can't simply call some function `onLoad` to initialize the application as we otherwise would. Instead, we use the `dojo.event` package, which is an aspect-oriented programming (AOP) implementation that allows you to hook up JavaScript to various events. This sounds simple, but `dojo.event` is an amazingly powerful package. You can hook up not only the usual UI events, such as `onLoad`, `onClick`, `onMouseOver`, and so on, but also hook up an event to any function call. So, if you want to have function A execute any time function B is called, for instance, without function B having to explicitly call function A (to avoid them having to know about each other), you can do so with the `dojo.event` package.

Here, we are saying that we want the function `init()` on the `contactManager` object to be called whenever the `onLoad` event of the `window` object fires. This gives us the same functionality as the usual use of `onLoad`, but using the Dojo event system.

In keeping with the idea of unobtrusive JavaScript, you'll notice there is precious little on the page so far. Some would argue that even what I have here should be in external `.js` files, but I think you can take that exercise a little too far sometimes. I don't feel it necessary to externalize every last bit of script, but certainly it should be kept to a minimum, and I think you'll agree that is the case here.

Initializing the Application

Now we come to the `<body>` of the page, which begins like this:

```
<div id="divInitializing" class="cssInitializing">
  <br><br><br><br><br><br><br><br><br><br>
  <center>...Initializing Contact Manager, please wait...</center>
</div>
```

When the page is first loaded, we don't want the user to be able to fire UI events before the UI has fully loaded (for example, clicking the save button before the persisted contacts have been fully restored might not be such a good thing). So, to start out, the user sees a message saying the application is initializing. Once everything is set to go, this `<div>` is hidden, and the main content is shown.

Adding the Fisheye List

Now we come to some UI fun. One of the things Dojo does really well—what it's probably most famous for—is widgets. One of the singularly most impressive widgets it contains (*the* most impressive for my money) is the fisheye list. Have you ever seen the launch bar in Mac OS?

Do you like how the icons expand and shrink as you mouse over them? Well, Dojo's fisheye list lets you have the same feature in your own web applications, as you can see in Figure 6-9. Of course, seeing it statically in print doesn't quite do it justice, so fire up the application and just mouse over the icons a bit. I think you'll have fun doing nothing but that for a few minutes.

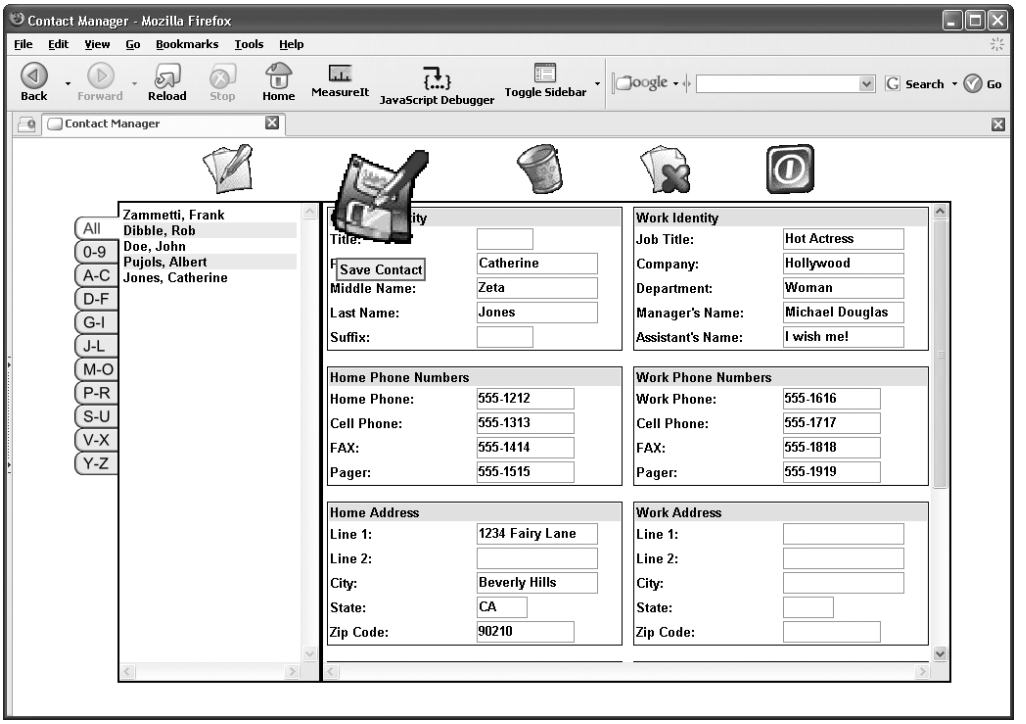


Figure 6-9. *The Dojo fisheye widget coolness—'nough said*

So is it difficult to make use of that widget? Heck no! Dojo allows you to define most, if not all, of its widgets with nothing but HTML. Listing 6-5 shows the markup responsible for the fisheye list in this application.

Listing 6-5. *The Fisheye Definition Markup*

```
<div class="outerbar">
  <div class="dojo-FisheyeList"
    dojo:itemWidth="64" dojo:itemHeight="64"
    dojo:itemMaxWidth="128" dojo:itemMaxHeight="128"
    dojo:orientation="horizontal" dojo:effectUnits="1"
    dojo:itemPadding="10" dojo:attachEdge="top"
    dojo:labelEdge="bottom" dojo:enableCrappySvgSupport="false">
    <div class="dojo-FisheyeListItem"
      onClick="contactManager.doNewContact();"
      dojo:iconsrc="img/icon_new.gif" caption="New Contact">
    </div>
```

```
      <div class="dojo-FisheyeListItem"
        dojo:iconsrc="img/transPix.gif" caption="">
      </div>
      <div class="dojo-FisheyeListItem"
        onClick="contactManager.doSaveContact();"
        dojo:iconsrc="img/icon_save.gif" caption="Save Contact">
      </div>
      <div class="dojo-FisheyeListItem"
        dojo:iconsrc="img/transPix.gif" caption="">
      </div>
      <div class="dojo-FisheyeListItem"
        onClick="contactManager.doDeleteContact()"
        dojo:iconsrc="img/icon_delete.gif" caption="Delete Contact">
      </div>
      <div class="dojo-FisheyeListItem"
        dojo:iconsrc="img/transPix.gif" caption="">
      </div>
      <div class="dojo-FisheyeListItem"
        onClick="contactManager.doClearContacts()"
        dojo:iconsrc="img/icon_clear.gif" caption="Clear Contacts">
      </div>
      <div class="dojo-FisheyeListItem"
        dojo:iconsrc="img/transPix.gif" caption="">
      </div>
      <div class="dojo-FisheyeListItem"
        onClick="contactManager.doExit();"
        dojo:iconsrc="img/icon_exit.gif" caption="Exit Contact Manager">
      </div>
    </div>
  </div>
```

When the page loads, Dojo will parse it, looking for tags that it recognizes as defining widgets. It then replaces them with the appropriate markup to form the widget. In this case, setting the class of a `<div>` to `dojo-FisheyeList` gets the job done. You'll notice that this `<div>` also contains a number of custom attributes. This is also a hallmark of Dojo and how you configure options for the widgets. Each of the icons is simply another `<div>`, this time with a class of `dojo-FisheyeListItem`. Notice that we put essentially blank icons between each real icon. There may be another way that I am unaware of, but this is how I was able to space the icons out a bit; otherwise, they looked a little cramped to me.

Feel free to play around with the attributes on the second `<div>`. By manipulating them, you can alter the fisheye list, such as making the expanded icons bigger, changing how far away from an icon you need to be to activate, and so on. Experimentation is a great way to learn Dojo (in fact, it's often the *only* way to really figure things out). Dojo is frequently worth the effort, so I certainly encourage you to put a little time into fiddling with it, and messing with the fisheye attributes is a good, gentle start.

Continuing on, for each icon, we define an `onClick` handler to do whatever it is the icon represents. This is again a situation where strict unobtrusive JavaScript practice would be to not put the event handlers in-line. I'd like to make two points here:

- If the event handler is just a call to some JavaScript function that does the actual work, I see no real problem having the handlers in-line. In fact, I think it makes more sense for them to be there, as they are attributes of the element and therefore should probably be defined with the element. Also, this will aid performance since no script has to process to hook up the event handlers.

- Because Dojo is generating markup in place of these `<div>` elements, trying to hook up the events after the fact would have been a bit more difficult than usual. The `dojo.event` package may have helped, but it would still not have been straightforward.

So, I won't lose any sleep tonight because of in-line event handlers.

As always, don't blindly follow any guideline, no matter how reasonable it generally seems. Instead, think about each situation and make an appropriate choice.

OK, putting philosophy aside and getting back to concrete code, let's move to the contact list section.

Adding the Contact List

Following the fisheye markup is a chunk of markup that defines the selector tabs on the left. In the interest of brevity, let's just look at the first one and note that the rest are virtually identical:

```
<img src="img/sel_xx_over.gif" id="sel_XX" class="cssTab"
  onClick="contactManager.eventHandlers.stClick(this);"
  onMouseOver="contactManager.eventHandlers.stOver(this);"
  onMouseOut="contactManager.eventHandlers.stOut(this);"><br>
```

This is the All tab, meaning it shows all contacts. The ID of the tab is used to determine the image it should have, based on its current state (that is, if it's the selected tab or if it's being hovered over). For instance, in this case, the ID is `sel_XX`, and as you can see, that is the beginning of the name of the image that is the initial source for this tab. When we look at the event handlers (which are again called from in-line handlers here), I think you'll find that this all makes more sense. As I mentioned, all the tabs that follow are the same, except that the ID is different. For instance, the next tab's ID is `sel_09` because it is the tab for showing contacts beginning with any character from 0 to 9.

Following this section of markup is a very small bit:

```
<td width="200" valign="top">
  <div class="cssContactList" id="contactList">
  </div>
</td>
```

The `<div> contactList` is, not surprisingly, where our list of contacts will be inserted. The list is filtered by the selector tabs, and this `<div>`'s contents will be rewritten any time a new tab is selected, or when a contact is added or deleted.

Following this is the remainder of the markup for the data-entry boxes where we fill in the information for our contacts. This is a big bit of markup that all looks pretty similar really, so let's just review a representative snippet of it. The section for entering contact identity (both personal and business) is shown in Listing 6-6.

Listing 6-6. *Section of the Data-Entry Markup*

```
<tr>
  <!-- Contact Identity -->
  <td width="49%" valign="top">
    <div class="cssDataBox">
      <table border="0" cellpadding="1" cellspacing="1"
        width="100%">
        <tr>
          <td colspan="2" class="cssDataBoxHeader">
            Contact Identity
          </td>
        </tr>
        <tr>
          <td width="50%" valign="middle">Title:</td>
          <td width="50%" valign="middle">
            <input type="text" id="title"
              maxlength="3" size="4" class="cssTextbox">
          </td>
        </tr>
        <tr>
          <td valign="middle">First Name:</td>
          <td valign="middle">
            <input type="text" id="firstName"
              maxlength="15" size="15" class="cssTextbox">
          </td>
        </tr>
        <tr>
          <td valign="middle">Middle Name:</td>
          <td valign="middle">
            <input type="text" id="middleName"
              maxlength="15" size="15" class="cssTextbox">
          </td>
        </tr>
        <tr>
          <td valign="middle">Last Name:</td>
          <td valign="middle">
            <input type="text" id="lastName"
              maxlength="20" size="15" class="cssTextbox">
          </td>
        </tr>
        <tr>
          <td valign="middle">Suffix:</td>
          <td valign="middle">
            <input type="text" id="suffix"
              maxlength="3" size="4" class="cssTextbox">
          </td>
        </tr>
```

```
    </table>
  </div>
</td>
<!-- Divider -->
<td width="2%"> </td>
<!-- Work Identity -->
<td width="49%" valign="top">
  <div class="cssDataBox">
    <table border="0" cellpadding="1" cellspacing="1"
      width="100%">
      <tr>
        <td colspan="2" class="cssDataBoxHeader">
          Work Identity
        </td>
      </tr>
      <tr>
        <td width="50%" valign="middle">Job Title:</td>
        <td width="50%" valign="middle">
          <input type="text" id="jobTitle"
            maxlength="24" size="15" class="cssTextbox">
        </td>
      </tr>
      <tr>
        <td valign="middle">Company:</td>
        <td valign="middle">
          <input type="text" id="company"
            maxlength="25" size="15" class="cssTextbox">
        </td>
      </tr>
      <tr>
        <td valign="middle">Department:</td>
        <td valign="middle">
          <input type="text" id="department"
            maxlength="25" size="15" class="cssTextbox">
        </td>
      </tr>
      <tr>
        <td valign="middle">Manager's Name:</td>
        <td valign="middle">
          <input type="text" id="managerName"
            maxlength="30" size="15" class="cssTextbox">
        </td>
      </tr>
```

```
            <tr>
              <td valign="middle">Assistant's Name:</td>
              <td valign="middle">
                <input type="text" id="assistantName"
                  maxlength="30" size="15" class="cssTextbox">
              </td>
            </tr>
          </table>
        </div>
      </td>
    </tr>
```

This is perfectly typical HTML. Note that the sizes of the fields have been limited such that each contact takes up just a hair under 1024 bytes. This is done so that if you wanted to modify the code to store contacts with plain-old cookies instead (hint, hint), you could do so and fit four contacts per cookie (remember that each cookie is limited to 4kb).

Also note the `cssTextbox` style class being applied. This is to deal with a situation where the border can change to inset when the field gets the focus, but not change back. This appears to have been a browser quirk, but specifically setting the border to what we want in the `cssTextbox` class takes care of it.

And with that, we have only one bit of markup left to look at, and that's the `goodbye.htm` page.

Writing goodbye.htm

There isn't a whole lot to the `goodbye.htm` file, as Listing 6-7 clearly indicates.

Listing 6-7. *The goodbye.htm File (Not Much to See Here)*

```
<html>

  <head>

    <title>Contact Manager</title>

  </head>

  <body>
    Thanks for using the contact manager... goodbye!
  </body>

</html>
```

This is a simple landing page that you see when you click the Exit icon. It's just always good form to end on a polite, "thanks for stopping by" kind of note.

Let's begin our review of the JavaScript files with the `EventHandlers` class, since this is something of a stand-alone piece of code.

Writing EventHandlers.js

In `index.htm`, you saw that the selector tabs called event-handler functions in this class. Also, as you shall soon see, all of the input fields actually do the same thing ("Huh?" you're thinking, I don't remember seeing any event handlers on the input fields," and you're right). Also, the contact list you see on the left side of the screen makes use of some functions here as well, but these are all simply UI-related functions. In other words, if you took these functions out, things would still basically work, although the UI wouldn't be as reactive as it is: the selectors wouldn't turn red when you hovered over them, the input fields wouldn't be highlighted when they gained focus, and the contact list wouldn't have a hover effect at all. All of these reactions are within the `EventHandlers` class, the class diagram of which is shown in Figure 6-10.

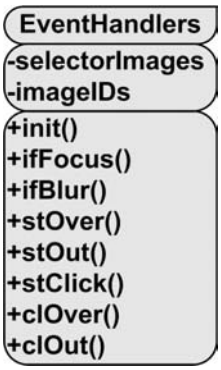


Figure 6-10. *UML diagram of the EventHandlers class*

First, the `EventHandlers` class is instantiated in the `ContactManager` class, which we will look at later, and the reference to it is one of the fields on `ContactManager` (the reason all the event handlers in `index.htm` are in the form `contactManager.xxxx()`).

The first item in the `EventHandlers` class is the `selectorImages` field, which is an array that will hold references to the preloaded images for the selector tabs. Next is another array field, `imageIDs`. This is a list of the selector tab IDs. The filenames of the graphics for each tab can be formed using these IDs, and so can the ID of the elements on the page, and we'll need both.

As I mentioned, `ContactManager` will instantiate `EventHandlers` and also initialize it by calling `init()` on `EventHandlers`. This `init()` function is as follows:

```
this.init = function() {

  this.selectorImages = new Array();

  // Load images from the above array and store them in selectorImages.
  for (var i = 0; i < this.imageIDs.length; i++) {
    var sid = this.imageIDs[i];
    this.selectorImages[sid] = new Image();
    this.selectorImages[sid].src = "img/" +
      sid + ".gif";
    this.selectorImages[sid + "_over"] = new Image();
    this.selectorImages[sid + "_over"].src = "img/" +
      sid + "_over.gif";
  }

  // Get all input fields and attach onFocus and onBlur handlers.
  var inputFields = document.getElementsByTagName("input");
  for (i = 0; i < inputFields.length; i++) {
    inputFields[i].onfocus = this.ifFocus;
    inputFields[i].onblur = this.ifBlur;
  }

} // End init().
```

The first task this function performs is preloading the images for the selector tabs. To do so, it iterates over the elements of the `imageIDs` array. For each, it instantiates an `Image` object and sets its `src` attribute to the filename of the image. Two images for each tab are loaded: one in its nonhover state and the other in its hover state. These images are added to the array, keyed by the ID taken from the `imageIDs` array.

What we wind up with is an associative array `selectorImages`, which contains all the preloaded images for the two states for each tab, and we can get at each image by using the ID as the key (for the hover images, it's the ID plus the string `_over` appended). This all saves us from writing explicit code to load each image. If we want to add more tabs later, as long as they follow the same naming scheme, we will need to add only the ID to the `imageIDs` array.

The next task this function performs is hooking up the event handlers to the input fields. Ah yes, there we go—that's how it works. We get the collection of `<input>` fields on the page using the handy-dandy `document.getElementsByTagName()` function. Then, for each of them, we attach `onFocus` and `onBlur` events, pointing to the `ifFocus()` and `ifBlur()` functions of the `EventHandlers` class. Nice to not need to specify these handlers on each input element, huh?

The idea of attaching event handlers to plain-old markup is another tenet of unobtrusiveness. While I'm not sure I like the idea of doing so for every event handler on a page, in cases like this, where a rather large number of elements need the same event handlers attached, this strikes me as better than having to put the handlers in-line with each element in the markup.

I suppose we should look at those `ifFocus()` and `ifBlur()` functions, shouldn't we? Well, here you go:

```
// ********** Input Field focus.
this.ifFocus = function() {

  this.style.backgroundColor = "#ffffa0";

} // End ifFocus();


// ********** Input Field blur.
this.ifBlur = function() {

  this.style.backgroundColor = "#ffffff";

} // End ifBlur().
```

Certainly, these are nothing special. They just change the background color of the input field: yellow when it has focus or white when it loses focus. Highlighting the current field is something that goes over well with most users, so it's a good thing to implement.

In the `ifFocus()` and `ifBlur()` functions, note the use of the keyword `this`, which can be a bit confusing. Recall that these functions are attached as event handlers to elements on the page. When they are called, the keyword `this` in the line with `this.style.backgroundColor` refers to the element firing the event because, at runtime, the keyword `this` is always evaluated in the context in which it executes. Contrast this to the usage of `this` in the line with `this.ifBlur`. In that case, `this` refers to the `EventHandlers` class because it is defined within that class. You can view this as static vs. dynamic interpretation of the `this` keyword; static being the usage to attach the method to the `EventHandler` class, and dynamic being the usage with the event handler. This is commonly referred to as *early binding* vs. *late binding*. Late binding occurs at runtime, while early binding occurs at compile time. Of course, there is no compile time with JavaScript, but it still means before the code actually runs.

Following those functions are three that deal with the selector tabs:

```
// ********** Selector Tab mouseOver.
this.stOver = function(inTab) {

  inTab.src = this.selectorImages[inTab.id + "_over"].src;

} // End stOver().


// ********** Selector Tab mouseOut.
this.stOut = function(inTab) {
```

```
    // Only switch state if not the current tab.
    if (contactManager.currentTab != inTab.id.substr(4, 2)) {
      inTab.src = this.selectorImages[inTab.id].src;
    }

  } // End stOut().


  // ********** Selector Tab click.
  this.stClick = function(inTab) {

    // Reset all tabs before setting the current one.
    for (var i = 0; i < this.imageIDs.length; i++) {
      var sid = this.imageIDs[i];
      $(sid).src = this.selectorImages[sid].src;
    }

    inTab.src = this.selectorImages[inTab.id + "_over"].src;

    // Record the current tab, and redisplay the contact list.
    contactManager.currentTab = inTab.id.substr(4, 2);
    contactManager.displayContactList();

  } // End stClick().
```

`stOver()` and `stOut()` handle the `onMouseOver` and `onMouseOut` events, respectively. They make use of the preloaded images stored in the `selectorImages` array discussed previously. It's a simple matter of changing the `src` attribute on the tab firing the event to the `src` of the appropriate image in the array. Of course, if a user hovers over the current selected tab and then mouses off it, we don't want to reset it to the nonhover state, hence the check in `stOut()` to be sure the tab that fired the event isn't the currently selected tab.

`stClick()` is just a little more interesting. When the user clicks a tab, it becomes the current tab, which means it remains in the hover state until another one is clicked. To accomplish this, we first have to reset the currently selected tab to its nonhover state. I decided to do this by resetting all the tabs, and then setting up the new current tab.[3] After the tabs are taken care of, we call `displayContactList()` on the `ContactManager` object to update the contact list to correspond to only those contacts that should show up on the new current tab.

Last up in the `EventHandler` class are the two functions that deal with mouse events on the items in the contact list on the left side of the screen:

3. I could have just as easily reset only the current tab, rather than all of them. You can view it as simply seeing an alternative approach in action.

```
// ********** Contact List mouseOver.
this.clOver = function(inContact) {

  inContact.className = "cssContactListOver";

} // End clOver().


// ********** Contact List mouseOut.
this.clOut = function(inContact) {

  if (inContact.getAttribute("altRow") == "true") {
    inContact.className = "cssContactListAlternate";
  } else {
    inContact.className = "cssContactListNormal";
  }

} // End clOut().
```

For the sake of demonstrating a slightly different technique, I decided that, unlike the handlers for the input fields, which access style attributes of the target element directly, here I would set the style class for the target element as appropriate. It is generally better I think to do it this way, since the styles are abstracted out into style sheets as they probably should be, but now you've seen that you can go the other way, too, if you feel it is more appropriate.

Here, the only real complexity is in the `clOut()` function. The style to switch the element to when the mouse leaves it can be one of two because the contacts in the contact list are displayed with alternate row striping, typical of many display lists. In order to determine which should be set, we interrogate the custom `altRow` attribute that each contact in the list carries. When that attribute is set to `true`, we know it is an element with a gray background (meaning the `cssContactListAlternate` style selector); otherwise, it is a white background (using the `cssContactListNormal` selector). Other than that, the `clOut()` function is pretty straightforward.

Writing Contact.js

If you are familiar with the concept of a Data Transfer Object (DTO) or Value Object (VO), the `Contact.js` source will be nothing at all special to you. That's because it simply defines a DTO representing a contact. Its class diagram is shown in Figure 6-11.

Figure 6-11. *UML diagram of the Contact class*

First up is a list of properties that represents a contact:

```
this.title = "";
this.firstName = "";
this.middleName = "";
this.lastName = "";
this.suffix = "";
this.jobTitle = "";
this.company = "";
this.department = "";
this.managerName = "";
this.assistantName = "";
this.homePhone = "";
this.homeCellPhone = "";
this.homeFAX = "";
this.homePager = "";
this.workPhone = "";
this.workCellPhone = "";
this.workFAX = "";
this.workPager = "";
this.homeEMail = "";
this.homeWebSite = "";
this.homeIMNickname = "";
this.workEMail = "";
this.workWebSite = "";
this.workIMNickname = "";
this.spouseName = "";
this.childrenName = "";
this.anniversary = "";
this.birthday = "";
this.highSchoolInfo = "";
this.collegeInfo = "";
this.custom1 = "";
this.custom2 = "";
this.homeAddressLine1 = "";
this.homeAddressLine2 = "";
this.homeCity = "";
this.homeState = "";
this.homeZipCode = "";
this.workAddressLine1 = "";
this.workAddressLine2 = "";
this.workCity = "";
this.workState = "";
this.workZipCode = "";

this.arrayIndex = -1;
```

Notice that straggler at the end, `arrayIndex`. This is a field that you can essentially think of as transient and is the index into the array of contacts stored in the `DataManager` where this contact is stored. This value will change as contacts are deleted, and will be dynamically calculated when contacts are restored at application initialization. Note that it *does not* appear in the `fieldsArray` we will discuss next, which is essentially the reason it is transient (the reason it isn't in the list should become apparent as we discuss `fieldsArray`).

After this, we find an interesting bit of code that certainly requires some explanation. It is the `fieldsArray` variable, which is an array containing the names of all the fields of this class, like so:

```
this.fieldsArray = [
  "title", "firstName", "middleName", "lastName", "suffix", "jobTitle",
  "company", "department", "managerName", "assistantName", "homePhone",
  "homeCellPhone", "homeFAX", "homePager", "workPhone", "workCellPhone",
  "workFAX", "workPager", "homeEMail", "homeWebSite", "homeIMNickname",
  "workEMail", "workWebSite", "workIMNickname", "spouseName", "childrenName",
  "anniversary", "birthday", "highSchoolInfo", "collegeInfo", "custom1",
  "custom2", "homeAddressLine1", "homeAddressLine2", "homeCity", "homeState",
  "homeZipCode", "workAddressLine1", "workAddressLine2", "workCity",
  "workState", "workZipCode"
];
```

If you look at these data fields, and then examine `index.htm` and look at the IDs of the input fields, you'll see that they match. This should be a clue as to what this array is for, but don't worry, we're about to figure it out together!

At various points in the application, we will need to populate an instance of the `Contact` class from the input fields, or populate the screen from the data fields within the `Contact` instance. One could certainly imagine writing code along these lines:

```
this.firstName = $("firstName").value;
this.lastName = $("lastName").value;
```

One could also imagine jumping off the Brooklyn Bridge on a hot day, surviving, and being cooled off. But just like jumping off the bridge, writing code like that isn't the best way to achieve the desired goal. Instead, it would be great if we could write some generic code to populate the object, or the input fields, without that code actually knowing precisely which fields are available. This is all the better when we want to add elements to a contact. That's exactly the kind of code that is present in the `Contact` class—for instance, in the `populateContact()` function:

```
this.populateContact = function() {

  for (var i = 0; i < this.fieldsArray.length; i++) {
    var fieldValue = $(this.fieldsArray[i]).value;
    this[this.fieldsArray[i]] = fieldValue;
  }

} // End populateContact();
```

Now the purpose of that `fieldsArray` member is probably starting to make sense. We iterate over the array, and because the members of the `Contact` class instance to populate are the same as the IDs of the input fields, we can use the values of the array to access both, thereby making this code agnostic about which fields are actually present in the class and on the screen. If we want to add a field to record a contact's blood type, we could do that by simply adding it to the `fieldsArray` member, and none of the population code would need to change.

Populating the screen is even simpler, but uses precisely the same concept:

```
this.populateScreen = function() {

  for (var i = 0; i < this.fieldsArray.length; i++) {
    $(this.fieldsArray[i]).value = this[this.fieldsArray[i]];
  }

} // End populateScreen().
```

The next function we come to is `toString()`. Recall that, as in Java, all JavaScript objects implement a `toString()` function. The basic version inherited from the base `Object` class (which, again as in Java, all objects in JavaScript inherit from) may or may not be very useful. However, we can override it simply and provide some output that is more useful. In this case, the output is JSON representing the contact. Here's the `toString()` function:

```
this.toString = function() {

  var json = "";
  json += "{ ";
  // For each field in the fieldsArray, get the value and add it to the JSON.
  for (var i = 0; i < this.fieldsArray.length; i++) {
    if (json != "{ ") {
      json += ", ";
    }
    json += "\"" + this.fieldsArray[i] + "\":\"" +
      this[this.fieldsArray[i]] + "\"";
  }
  json += " }";
  return json;

} // End toString().
```

You may be wondering why I didn't instead have something like a `toJSON()` function. That would have worked perfectly well, except that overriding `toString()` instead makes the code to get the contact as JSON just a hair cleaner, as you'll see later in the `DataManager` class. Also, it makes debugging a little better because, if you want to display a given `Contact` instance—for example, in an `alert()` pop-up—you'll get something that is a bit more helpful than the default `toString()` provides. And that's always a good thing.

The last function found in the `Contact` class is `restoreFromJSON()`:

```
this.restoreFromJSON = function(inJSON) {

  eval("json = (" + inJSON + ")");
  for (var i = 0; i < this.fieldsArray.length; i++) {
    this[this.fieldsArray[i]] = json[this.fieldsArray[i]];
  }

} // End restoreFromJSON().
```

The name says it all. This function populates the `Contact` class instance it is executed on from an incoming string of JSON. I think the beauty of storing the contact as JSON should be pretty apparent here: this code is amazingly simple and compact. We simply iterate over that handy `fieldsArray` again, and for each element, we set the appropriate field in the class from the parsed JSON object. It's very simple, which is also always a good thing.

By the way, remember the `arrayIndex` field that I mentioned was transient? Do you see now why that is? By virtue of not being listed in `fieldsArray`, it is neither included in the JSON generated by `toString()` nor is it reconstituted by `restoreFromJSON()`.

Writing ContactManager.js

The `ContactManager` class, defined in the `ContactManager.js` file, is the main code behind this application. Its class diagram is shown in Figure 6-12.

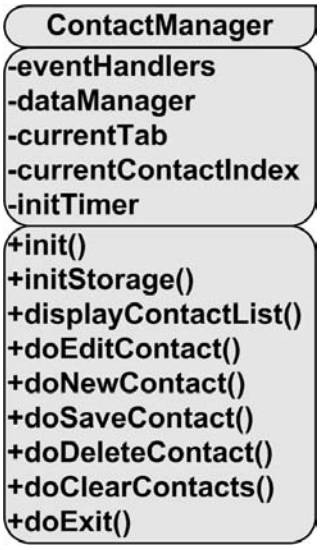


Figure 6-12. *UML diagram of the ContactManager class*

This class starts off with five data fields:

- `eventHandlers`: A reference to an instance of the `EventHandlers` class
- `dataManager`: A reference to an instance of the `DataManager` class

- `currentTab`: The ID of the currently selected tab
- `currentContactIndex`: The index into the array of contacts (stored in the `DataManager` class) of the contact currently being edited (or –1 if creating a new contact)
- `initTimer`: A reference to the timer used during application initialization

Initializing

Recall that in `index.htm`, we use the Dojo event system to hook up an `onLoad` event that calls the `init()` function of the `ContactManager` class. Well, now it's time to see what's in that function:

```
this.init = function() {

  contactManager.eventHandlers = new EventHandlers();
  contactManager.eventHandlers.init();
  contactManager.dataManager = new DataManager();
  this.initTimer  = setTimeout("contactManager.initStorage()", 500);

} // End init().
```

First, the `EventHandlers` class is instantiated and the reference to it stored in the `eventHandlers` field. Next, `init()` is called on that class.

Then we do the same thing for the `DataManager` class, but we don't call `init()` on it right away, as we do with the `EventHandlers` instance. Instead, we start a timer that every 500 milliseconds calls the `initStorage()` function of the `ContactManager`. That function is as follows:

```
this.initStorage = function() {

  if (dojo.storage.manager.isInitialized()) {
    clearTimeout(this.initTimer);
    contactManager.dataManager.init();
    contactManager.displayContactList();
    $("divInitializing").style.display = "none";
    this.initTimer = null;
  } else {
    this.initTimer  = setTimeout("contactManager.initStorage()", 500);
  }

} // End initStorage().
```

What's this all about you ask? Simply, Bad Things™ will happen if you try to use Dojo's functions for working with shared objects before the storage system has properly initialized. When we look at the `DataManager` class, you'll see that one of the things done in its `init()` function is to restore saved contacts from persistent storage (read local shared objects). Therefore, we can't call that function immediately as we did with the `EventHandlers` instance. In fact, we can't call it until the storage system has fully initialized. And that's the reason for the timer. Every 500 milliseconds, we check with Dojo to see if the storage system has initialized yet. If not, we just keep firing the timer until it does.

As soon as the storage system initializes, we stop the timer and call `init()` on the `DataManager` instance. At that point, we also display the contact list so that any restored contacts will be available to the user, and finally we hide the initializing message discussed earlier. Once the `DataManager` has been initialized, the application is then ready for user interaction.

Generating the Contacts

Recall that the list of contacts restored from persistent storage is displayed during this initialization cycle. Here is the code that generates the list of contacts:

```
this.displayContactList = function() {

  // Get a list of contacts for the current tab.
  var contacts = this.dataManager.listContacts(this.currentTab);

  // Generate the markup for the list.
  var html = "";
  var alt = false;
  for (var i = 0; i < contacts.length; i++) {
    html += "<div indexNum=\"" + contacts[i].arrayIndex + "\" ";
    html += "onMouseOver=\"contactManager.eventHandlers.clOver(this);\" ";
    html += "onMouseOut=\"contactManager.eventHandlers.clOut(this);\" ";
    html += "onClick=\"contactManager.doEditContact(" +
      "this.getAttribute('indexNum'));\" ";
    if (alt) {
      html += "class=\"cssContactListAlternate\" altRow=\"true\">";
      alt = false;
    } else {
      html += "class=\"cssContactListNormal\" altRow=\"false\">";
      alt = true;
    }
    html += contacts[i].lastName + ", " + contacts[i].firstName;
    html += "</div>";
  }

  // Display it.
  $("contactList").innerHTML = html;

} // End displayContactList().
```

First, the list of contacts is retrieved from the `DataManager`. We pass the `listContacts()` function of the `DataManager` class the currently selected tab so that the list can be filtered accordingly. Next, we cycle through the returned contacts (each element of the returned array is a `Contact` object) and construct the appropriate markup for the list. Each element in the list has mouse events attached to highlight the contact when the user hovers over it. And each element also contains an `onClick` handler that calls `doEditContact()` in the `ContactManager` class, which loads the contact into the input fields on the screen for editing. Once the markup is fully constructed, it is inserted into the `contactList` `<div>`, which shows it to the user.

Editing Contacts

Speaking of the doEditContact() function, let's see that now, shall we?

```
this.doEditContact = function(inIndex) {

  // Record contact index, retrieve contact and populate screen.
  this.currentContactIndex = inIndex;
  var contact = this.dataManager.getContact(inIndex);
  contact.populateScreen();

}
```

Wow, really, that's it? Yes indeed. Note in the displayContactList() function that each contact listed has an indexNum custom attribute attached to it. This value corresponds to the value of the arrayIndex member of the Contact class. Since we are viewing a subset of the contacts array when a particular tab is set, using this value ensures that a contact's indexNum attribute is the correct index into the array. This way, when the user clicks a contact to edit it, we pull the correct data to display.

For instance, if you click the A-C tab, and you see three of ten stored contacts, the array returned by listContacts() in the DataManager class will simply return an array with three elements. But the index in that array (0, 1, or 2) may not match the index into the contacts array for a given contact. In other words, the first contact returned by listContacts() is index 0 in that returned array, but may actually be the contact at index 9 in the main contacts array. Therefore, if we used the index number of the returned array as the value for indexNum, we wouldn't go after the correct element in the contacts array (in this example, 0 instead of 9). We instead need the arrayIndex field of the Contact object.

Adding Button Functions

The ContactManager class contains five more functions, and each corresponds to one of the five buttons at the top of the page. First up is doNewContact():

```
this.doNewContact = function() {

  if (this.initTimer == null) {

    if (confirm("Create New Contact\n\nYou will lose any unsaved changes.  " +
      "Are you sure?")) {
      document.forms[0].reset();
      this.currentContactIndex = -1;
    }

  }

} // End doNewContact().
```

Not really much going on here, I admit. One thing you'll notice in this and the next functions is the check of initTimer being null. When the application starts, initTimer has a value of –1. When the initialization cycle completes, it is set to null. Since we don't want the user to be able

to do anything until the application initializes fully, and since Dojo produces the fisheye icons before initialization completes, we need to ensure initialization is finished before we process any user events. That's why we need the check that allows the code to execute only after `initTimer` is null, meaning everything is ready to go.

Once that is the case, we simply make sure the user wants to create a new contact, because if she had any edits on the page, they would be lost. Then we reset the form, which clears all our input fields, and set the `currentContactIndex` to –1, which indicates to the rest of the code that a new contact is being created.

Next up is the function called when the save icon is clicked, appropriately named `doSaveContact()`:

```
this.doSaveContact = function() {

  if (this.initTimer == null) {

    // Make sure required fields are filled in.
    if ($("firstName").value == "" || $("lastName").value == "") {
      alert("First Name and Last Name are required fields");
      return false;
    }

    // Create a new contact and populate it from the entry fields.
    var contact = new Contact();
    contact.arrayIndex = this.currentContactIndex;
    contact.populateContact();

    // Save the contact.
    this.dataManager.saveContact(contact, this.currentContactIndex);

    // Redisplay the updated contact list.
    this.displayContactList();

    // Reset the entry fields and currentContactIndex.
    document.forms[0].reset();
    this.currentContactIndex = -1;

  }

} // End doSaveContact().
```

This function handles two different save situations: saving a new contact or saving edits to an existing contact. So, after our check of `initTimer` as in `doNewContact()`, we first do a quick edit check to ensure a first name and last name have been entered. These two fields are the only required fields for a contact because they are used to generate the contact list, and this is true whether it is a new contact or an existing one.

Once that is done, we instantiate a new `Contact` object and tell it to populate itself from the input fields by calling its `populateContact()` function. We also set the `arrayIndex` field based on

the currentContactIndex value, which will be –1, as set in doNewContact() if this is a new contact, or to the appropriate array index if editing an existing contact.

Next, we call saveContact() on the DataManager, passing it the contact and the index (we'll be looking at that code shortly). After that, we regenerate the contact list and show it because we may have just added a contact that should be immediately visible (we're on the tab the contact would appear on naturally or on the All tab). Lastly, we reset the input fields and the currentContactIndex value (which is actually a little redundant, but does no harm), and that's how a contact is saved.

The next function to look at is doDeleteContact(), and I'll give you just one guess what it does:

```
this.doDeleteContact = function() {

  if (this.initTimer == null) {

    if (this.currentContactIndex != -1 &&
      confirm("Are you sure you want to delete this contact?")) {

      // Ask the data manager to do the deletion.
      this.dataManager.deleteContact(this.currentContactIndex);

      // Redisplay the updated contact list.
      this.displayContactList();

      // Reset the entry fields and currentContactIndex.
      document.forms[0].reset();
      this.currentContactIndex = -1;

    }

  }

} // End doDeleteContact().
```

After the usual check of initTimer, and a confirmation that the user really wants to delete the current contact (which includes a check to be sure there is a contact selected to be deleted), we ask the DataManager to do the deletion for us, passing it the index of the contact to delete. Once again, we regenerate and display the contact list, and reset the input fields and the currentContactIndex value (and yes, it's still a bit redundant here, but it still does no harm, and I prefer variables that aren't based on user input being in known states at any given time because they make for easier debugging sessions).

The next function is doClearContacts(), which is a giant, shiny "push to destroy the universe" button. Well, maybe not quite, but it does delete *all* contacts from persistent storage, so it isn't something that you want the user clicking willy-nilly. For that reason, there is a double verification required, as you can plainly see:

```
this.doClearContacts = function() {

  if (this.initTimer == null) {

    if (confirm("This will PERMANENTLY delete ALL contacts from " +
      "persistent storage\n\nAre you sure??")) {
      if (confirm("Sorry to be a nudge, but are you REALLY, REALLY SURE " +
        "you want to lose ALL your contacts FOREVER??")) {
        this.dataManager.clearContacts();
        // Redisplay now empty contact list.
        this.displayContactList();
        // Reset form for good measure.
        document.forms[0].reset();
        this.currentContactIndex = -1;
        alert("Ok, it's done.  Don't come cryin' to me later.");
      }
    }

  }

} // End doClearContacts().
```

Only one function remains, `doExit()`, and it's a pretty trivial piece of code:

```
this.doExit = function() {

  if (this.initTimer == null) {

    if (confirm("Exit Contact Manager\n\nAre you sure?")) {
      window.location = "goodbye.htm";
    }

  }

} // End doExit().
```

Nothing fancy here—just a quick confirmation, and then the browser is redirected to the `goodbye.htm` page we looked at earlier.

Writing DataManager.js

Throughout the `ContactManager` code, you saw a number of calls to the `DataManager`. Now we come to the point in our show where we need to take a look at that class and see what's going on under the covers there. Its class diagram is shown in Figure 6-13.

Figure 6-13. *UML diagram of the DataManager class*

I wrote the `DataManager` class with the idea in mind that you could swap it out for another implementation, perhaps one that made use of some ActiveX control (as evil as those are) to persist the contacts and deal with the underlying storage mechanism. As such, its public API is pretty generic and conducive to such a swap.

The first item in the `DataManager` class is the `contacts` array. This is, not surprisingly, the array in which our contacts are stored. This array is loaded from shared objects when the application initializes, and until the user exits the application, the array is essentially kept synchronized with the persistent store. In other words, when we add a contact, it is added to this array, and then the array is persisted to shared objects. When we delete a contact, it is deleted from the array, and then the array is persisted to shared objects. When we edit an existing contact, it is updated in the array, and then the array is persisted to shared objects. Are you seeing a pattern here?

When the `ContactManager` class is instantiated during application initialization, it instantiates the instance of `DataManager` and keeps a reference to it, as you saw earlier. Also as you saw earlier, it initialized the `DataManager` by calling its `init()` function, which we can now look at:

```
this.init = function() {

  // Read in existing contacts from the applicable storage mechanism.
  this.contacts = new Array();
  this.restoreContacts();

} // End init().
```

Once the `contacts` array is initialized, we ask the `DataManager` to restore any contacts from persistent storage by calling the `restoreContacts()` function, which is the following:

```
this.restoreContacts = function() {

  // Retrieve stored contacts.
  var storedContacts = dojo.storage.get("js_contact_manager_contacts");

  // Only do work if there actually were any contacts stored.
  if (storedContacts) {
    // Tokenize the string that was stored.
    var splitContacts = storedContacts.split("~>!<~");
    // Each element in splitContacts is a contact.
    for (var i = 0; i < splitContacts.length; i++) {
      // Instantiate a new Contact instance and populate it.
      var contact = new Contact();
      contact.restoreFromJSON(splitContacts[i]);
      contact.arrayIndex = i;
      // Add it to the array of contacts.
      this.contacts.push(contact);
    }
  }

} // End restoreContacts().
```

Restoring contacts is a pretty simple process. First, we ask Dojo to get our contacts from local shared objects. The contacts are stored under the name `js_contact_manager_contacts`. The code then checks to be sure we actually got something back. If this is the first time the application is run on this machine, for instance, there will be no object under that name, and hence no contacts to restore.

Assuming there are contacts, what we get back from the `dojo.storage.get()` call is basically a giant string consisting of contacts in JSON form separated by a sequence of characters: `~>!<~`. We can't just use a single character, such as a comma, because it could appear naturally in the data entered by the user, and therefore we would not tokenize the string properly. So we need a delimiter to separate contacts that isn't likely to be entered by the user. The `~>!<~` sequence is a reasonably safe combination, in that it isn't likely to naturally occur in real user input. However, entering it in any field for a contact will, in fact, break the code.[4]

After the string is tokenized, we start iterating over the tokens, which I remind you are each a contact in JSON form. For each, all we need to do is instantiate a new `Contact` object, and then pass the JSON string to the `restoreFromJSON()` function of the `Contact`, which we looked at earlier. It uses the evaluated JSON and populates the `Contact` instance, effectively restoring it.

Only two things remain to do: set the `arrayIndex` field of the contact and add it to the `contacts` array. Once all the tokens (contacts) have been processed in this way, `restoreContacts()` has completed its work, and we now have a `contacts` array that is identical to how it was when it was last persisted.

The next function we encounter in our exploration of the `DataManager` class is the `saveContact()` function, which is used to rotate the ad banner at the top of the page.

4. To be really bullet proof, the application should check all inputs to be sure this sequence doesn't appear. But it seems pretty unlikely that it would be entered *except* by someone deliberately trying to break the program, so I can live with the risk.

You were paying attention there I hope and noticed something amiss? Obviously, the `saveContact()` function, in fact, is called to save a contact, and it looks like this:

```
this.saveContact = function(inContact, inIndex) {

  // Save new contact.
  if (inIndex == -1) {
    inContact.arrayIndex = this.contacts.length;
    this.contacts.push(inContact);
  } else {
    // Update existing contact.
    this.contacts[inIndex] = inContact;
  }
  this.persistContacts();

} // End saveContact().
```

We first do some simple branching based on whether we are saving a new contact (`inIndex == -1`) or updating an existing one (`inIndex != -1`). In the case of adding a new index, all we really need to do is set the `arrayIndex` field of the `inContact` object, and push it onto the `contacts` array. When we are updating a contact, we just set the appropriate element of the `contacts` array to the `inContact` object. After one of those things happens, we call `persistContacts()` to save the `contacts` array to shared objects.

So, what of this `persistContacts()` function? Let's get to that now. Actually, with `persistContacts()` goes `saveHandler()`, which works hand in hand with `persistContacts()` to do the job of saving to shared objects:

```
this.persistContacts = function() {

  // First, construct a giant string from our contact list, where each
  // contact is separated by ~>!<~ (that delimiter isn't too likely to
  // naturally appear in our data I figure!)
  var contactsString = "";
  for (var i = 0; i < this.contacts.length; i++) {
    if (contactsString != "") {
      contactsString += "~>!<~";
    }
    contactsString += this.contacts[i];
  }

  try {
    dojo.storage.put("js_contact_manager_contacts", contactsString,
      this.saveHandler);
  } catch(e) {
    alert(e);
  }

} // End persistContacts().
```

```
// ********** Callback function for Flash storage system save.
this.saveHandler = function(status, keyName){

  if (status == dojo.storage.FAILED) {
    alert("A failure occurred saving contact to Flash storage");
  }

} // End saveHandler().
```

First, we construct that giant string from the `contacts` array that I talked about earlier when discussing the `restoreContacts()` function. To do so, we simply iterate over the `contacts` array and add each contact to a string, which fires its `toString()` function as described earlier when we looked at the `Contact` class.

We then append our special delimiter character sequence, and continue that until the whole `contacts` array has been processed into this string. Once that's done, we pass this string to the `dojo.storage.put()` function, telling it to store the string under the name `js_contact_manager_contacts`. We also pass it a reference to the `saveHandler()` function. This function is a callback that will be called by Dojo when the operation completes. We can examine the outcome of the operation and act accordingly. Here, all we really care about is a failure; in which case, we alert the user. There isn't a whole lot to be done if a failure occurs, so that's the end of things. We could also alert users if the operation succeeds, but I think they can surmise that if no error message is shown.

The `getContact()` function comes next, and it's definitely a trivial piece of code. In fact, it's so trivial that I'm not even going to show it. All it does is take in an index number and return that element from the `contacts` array. A single line of code, that's it.

Following `getContact()` is `deleteContact()`, which has a little more meat to it (although I admit, not a *lot* of meat):

```
// ********** Delete a contact.
this.deleteContact = function(inIndex) {

  // Delete from contacts array.
  this.contacts.splice(inIndex, 1);

  // Store the updated contact list.
  this.persistContacts();

  // Finally, renumber all the remaining contacts.
  for (var i = 0; i < this.contacts.length; i++) {
    this.contacts[i].arrayIndex = i;
  }

} // End deleteContact().
```

JavaScript arrays expose the `splice()` method, which allows us to remove elements from an array easily. We simply specify from which index to start removing elements, and then specify how many elements to remove.

Once the contact has been removed, we ask the DataManager to persist our contacts, effectively updating the shared objects.

One last bit of work remains at this point, and that is to renumber the contacts. Recall that the arrayIndex field, which is not persisted with the contact, is the element in the contacts array where the contact is located. It is important that this be accurate, because if the user clicks a selector tab and we return only a subset of the contacts array, each contact needs to know what index it's at so we can edit and/or delete the appropriate contact if the user requests it.

However, let's say we have three contacts in the contacts array, and we delete the second one. Now, the first contact has an arrayIndex value of 0, and the second one has a value of 3, because that's where it was previously (we're assuming they were numbered correctly to begin with). So, if the user clicks that second contact to edit, we'll get an error as we try to access index 3 of the array, which no longer exists. As you can see, the arrayIndex values need to be updated when we delete a contact. Fortunately, this is a simple procedure: we just need to iterate over the array and set the arrayIndex for each contact as we do so. This will remove any gaps in the order left by a deletion, and everything will be set up properly again.

Following deleteContact() is listContacts(), which is called to get some subset of the contacts array (or the entire array in the case of the All tab). While it serves an important purpose, there isn't really much to it:

```
this.listContacts = function(inCurrentTab) {

  if (inCurrentTab == "XX") {
    // ALL tab selected, return ALL contact.
    return this.contacts;
  } else {
    // Filter contacts based on current tab.
    var retArray = new Array();
    var start = inCurrentTab.substr(0, 1).toUpperCase();
    var end = inCurrentTab.substr(1, 1).toUpperCase();
    for (var i = 0; i < this.contacts.length; i++) {
      var firstLetter = this.contacts[i].lastName.substr(0, 1).toUpperCase();
      if (firstLetter >= start && firstLetter <= end) {
        retArray.push(this.contacts[i]);
      }
    }
    return retArray;
  }

} // End listContacts().
```

First, we check to see if the inCurrentTab value is XX, which indicates the All tab has been selected. In that case, we just return the contacts array, and that's that. For any other tab, we have a little more work to do. First, we create a new array to hold the subset of contacts we'll be returning. Next, we take the first character of inCurrentTab, converted to uppercase, which is the start of the range of characters for which we want to return contacts. We do the same for the second character, which is the end of the range.

So, let's say the inCurrentTab value is AC. In that case, we want to return any contact whose last name begins with *A*, *B*, or *C*. So, we iterate over the contacts and, for each, we grab the first

letter of the last name. After converting it to uppercase, we see if it falls within the range defined by `start` and `end`, and if so, we add it to the array. Once we go through all the contacts, we return the array, which is now some subset of the contacts appropriate for the currently selected tab.

We're just about finished with the `DataManager` class now. Here's the final function left to examine:

```
this.clearContacts = function() {

  dojo.storage.clear();
  this.contacts = new Array();

} // End clearContacts().
```

If the user becomes depressed and wants to cut off all contact with the outside world, he may decide he no longer wants any contacts, and he may click the Clear Contacts icon. In that case, the `clearContacts`) function is called. Two things need to occur to clear contacts. First, we need to clear our persistent storage. Dojo provides the `dojo.storage.clear()` function for this. After that, we have to clear the `contacts` array in `DataManager`, which is a simple matter of setting it to a new, empty array. After this, the user can go seek professional psychiatric help to deal with his problems.

Suggested Exercises

While the primary goal of this chapter is to highlight the persistence aspect of the project, that's no reason not to make suggestions that tackle other areas as well. Here are just a few ideas you could explore that would certainly prove to be good learning exercises:

- Allow for searching on *any* field. For bonus points, use some Dojo transition effects to have a search panel slide into view.
- Allow for sending email by clicking a contact's email address. I purposely left this out because I wanted to make this relatively simple suggestion here. This addition shouldn't take much effort.
- Sort the contacts listed on a given tab.
- Implement persistence to cookies. I purposely limited the maximum size that a single contact could take up to just a hair under 1024 bytes. This should allow you to store 4 contacts per cookie, so with the limit of 20 cookies per domain you can store 80 contacts.

Summary

In this chapter, we looked at a couple different mechanisms for storing data in a persistent manner on the client. We focused on using the local shared objects mechanism provided by Adobe's Flash plug-in. You saw how the Dojo library helps make all of this a bit easier by taking care of most of the details for us. We built a small Contact Manager application to demonstrate these techniques, and in the process saw some Dojo widget magic as well.

CHAPTER 7

JSDigester: Taking the Pain Out of Client-Side XML

I'll just come out and say it: parsing XML in a browser is not a particularly pleasant experience. Actually, if you think about it, parsing XML *anywhere* can be a bit of a hassle. However, one library that does make it a bearable experience is the Jakarta Commons Digester component (`http://jakarta.apache.org/commons/digester`). Digester allows you to specify a series of rules that will be triggered by various elements in an XML document. These rules may handle the parsing in a number of ways, including creating and populating objects from the XML. Wouldn't it be great if we could do the same thing in JavaScript? Well, we're going to make that dream a reality in this chapter, and in the process, make working with XML on the client a much less painful experience.

Parsing XML in JavaScript

Parsing XML on the client is about as much fun as a gum scraping is for most people. Believe me, I don't make the dental analogy lightly (well, while I've never had my gums scraped, I *am* married with children, so I figure I know what it would feel like). If you've ever done much in the way of parsing XML in JavaScript, then the code shown in Listing 7-1 will probably look both familiar *and* painful.

Listing 7-1. *Parsing XML in JavaScript in a Browser*

```
<html>

  <head>

    <link rel="StyleSheet" href="styles.css" type="text/css">

    <title>Simple JavaScript XML Parsing Example</title>

    <script>

      function doParsing() {
```

```
        // This is the XML we will parse.
        var xml = "<messages>";
        xml += "<msg poster=\"Frank\">Hello!</msg>";
        xml += "<msg poster=\"Traci\">I hope all is well with you!</msg>";
        xml += "<msg poster=\"Andrew\">Well, I guess that's it.</msg>";
        xml += "<msg poster=\"Ashley\">Have a good day!</msg>";
        xml += "</messages>";

        // Instantiate an XML parser (or DOM, depending on browser).
        var xmlDoc = null;
        if (window.XMLHttpRequest){
          var parser = new DOMParser();
          xmlDoc = parser.parseFromString(xml, "application/xml");
        } else {
          xmlDoc = new ActiveXObject("Microsoft.XMLDOM");
          xmlDoc.async = false;
          xmlDoc.loadXML(xml);
        }

        // Now iterate over the DOM created above, and construct an output
        // string to display.
        var strOut = "Root node = " + xmlDoc.documentElement.nodeName + "<br>";
        for (var i = 0; i < xmlDoc.documentElement.childNodes.length; i++) {
          strOut += "nodeName = " +
            xmlDoc.documentElement.childNodes[i].nodeName + ", poster = " +
            xmlDoc.documentElement.childNodes[i].getAttribute("poster") +
            ", text = " +
            xmlDoc.documentElement.childNodes[i].firstChild.nodeValue + "<br>";
        }
        document.getElementById("divOut").innerHTML = strOut;

      }

    </script>

  </head>

  <body class="cssBody">

    <div class="cssTitle">JavaScript XML Parsing Example</div>
    <br><br>
    <input type="button" value="Click me to parse XML"
      onClick="doParsing();" class="cssBody">
    <br><br>
    Info will appear here:
```

```
    <br><br>
    <div id="divOut" class="cssLog"></div>

  </body>

</html>
```

While I have to admit it probably isn't much to look at, Figure 7-1 shows the output of the code in Listing 7-1.

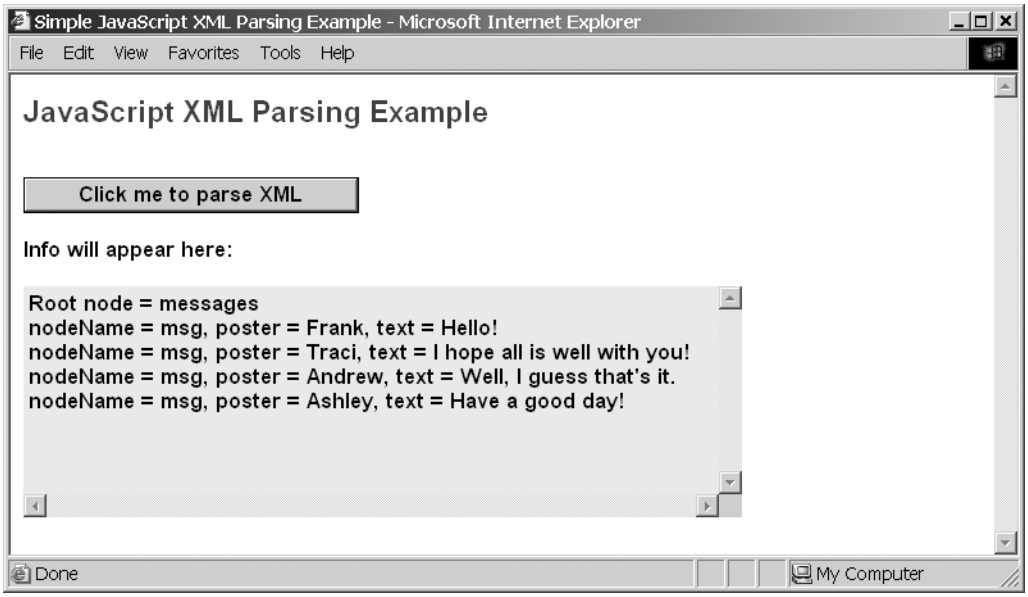


Figure 7-1. *The result of the simple parsing example*

Keep in mind that this is a very trivial example. Also keep in mind that there is more than one way to write XML parsing code in JavaScript. However, I think it's safe to say that all approaches suffer from the same flaw: they are simply too much work. Not only that, but did you notice that the code is aware of the form and structure of the XML it is parsing? This isn't always a bad thing, of course, and is often outright necessary. But it's still something to try to limit as much as possible, so that our code is flexible and reusable. In this case, referencing specific element names probably isn't ideal.

Another important consideration is the question of what you are actually trying to do with the XML you are parsing. Are you simply scanning through it and acting on each element, perhaps displaying it? Or is your goal to populate some objects from the XML—something of an object-to-XML mapping? If the former is the goal, then code like that in Listing 7-1 maybe is not so bad, and could ultimately be more efficient. If the goal is object creation and population, however, then you'll need to write far more code to make that happen.

JSDigester Requirements and Goals

The aim of this chapter's project is to make our lives a lot simpler. We will take a cue from the Apache Jakarta Commons Digester project and build ourselves a JavaScript implementation of Digester that we'll call JSDigester.

Here are our goals and requirements:

- Rather than try to implement the entire breadth of Digester's capabilities, we'll include just a bare minimum to get us by. However, like Digester, JSDigester should be extensible, so that if we need other rules down the road, we can implement them and use them without any trouble.
- In short, we should be able to create an instance of JSDigester, configure some rules on it, and pass it some XML to parse. The result should be an object that presumably contains other objects populated from the parsed XML.
- JSDigester should look and work as much like its big brother Digester as possible, with the understanding that there will be some things that just don't translate very well to JavaScript or are outright not possible, and hence will not be attempted here.
- Naturally, JSDigester should be fully cross browser–compatible.

With those goals in mind, let's get down to business and try to extricate ourselves from the relative masochistic experience of parsing XML in JavaScript.

How Digester Works

To try to duplicate Commons Digester, we need look at that project itself. Although Digester is a Java library, I think a basic example will be understandable to most any developer with experience in a C-like language.

The Digester home page does a good job of describing what Digester is (and it had better, right?):

> *Basically, the Digester package lets you configure an XML -> Java object mapping module, which triggers certain actions called rules whenever a particular pattern of nested XML elements is recognized. A rich set of predefined rules is available for your use, or you can also create your own. Advanced features of Digester include:*
>
> - *Ability to plug in your own pattern matching engine, if the standard one is not sufficient for your requirements.*
> - *Optional namespace-aware processing, so that you can define rules that are relevant only to a particular XML namespace.*
> - *Encapsulation of Rules into RuleSets that can be easily and conveniently reused in more than one application that requires the same type of processing.*

The basic idea behind Digester is that you can map certain XML elements to rules that define how they will be treated. Then, as an XML document is parsed and Digester encounters the mapped elements, it creates objects for these elements, sets properties of a given object, or calls methods of a given object—all according to the rules you define. When the XML document is completely parsed, you wind up with an object graph containing new objects that represent various elements in the XML document.

Digester can seem a bit overwhelming at first, but once you get the hang of it, you won't want to use anything else! Let's look at a simple example now.

Imagine we are writing a simple shopping cart web application, as seen on virtually any e-commerce web site like Amazon.com. Suppose that our application has the two classes shown in Listings 7-2 and 7-3, which represent the shopping cart and any items in the shopping cart.

Listing 7-2. *The ShoppingCart Class*

```
package myApp;
public class ShoppingCart {
    public void addItem(Item item);
    public item getItem(int id);
    public Iterator getItems();
    public String getShopperName();
    public void setShopperName(String shopperName);
  }
```

Listing 7-3. *The Item Class*

```
  package myApp;
  public class Item {
    public int getId();
    public void setId(int id);
    public String getDescription();
    public void setDescription(String description);
  }
```

Now let's assume that we have previously had a user who started shopping, maybe buying Christmas presents for his friends, dropped some items in his cart, and then left. Let's further assume that we wanted the user to have a good shopping experience, so we saved the state of his shopping cart for later. Lastly, let's assume that we saved that state in the form of the XML document shown in Listing 7-4.

Listing 7-4. *The Saved XML Document Representing the User's Shopping Cart*

```
<ShoppingCart shopperName="Rick Wakeman">
  <Item id="10" description="Child's bike (boys)" />
  <Item id="11" description="Red Blouse" />
  <Item id="12" description="Reciprocating Saw" />
</ShoppingCart>
```

Now, when the user comes back to our site, we want to read in this XML document and create a `ShoppingCart` object that contains three `Item` objects, all with their properties set appropriately to match the data in the XML document. We could do so with the following Digester code:

```
Digester digester = new Digester();
digester.setValidating(false);
digester.addObjectCreate("ShoppingCart", "myApp.ShoppingCart");
digester.addSetProperties("ShoppingCart");
digester.addObjectCreate("ShoppingCart/Item", "myApp.Item");
digester.addSetProperties("ShoppingCart/Item");
digester.addSetNext("ShoppingCart/Item", "addItem", "myApp.Item");
ShoppingCart shoppingCart = (ShoppingCart)digester.parse();
```

Let's break this down and look at each line of code. The first line instantiates a `Digester` object. The second line of code tells Digester that we do not want the XML document validated against a Document Type Definition (DTD).

After that comes a series of `add`*`XXX`*`()` method calls, which each adds a particular rule to Digester. A number of built-in rules are available, and you can write your own as required.

All of the rules share the first method call parameter in common: the path to the element for which the rule will fire. Recall that an XML document is a hierarchical tree structure, so to get to any particular element in the document, you form a path to it that starts at the document root and proceeds through all the ancestors of the element. In other words, looking at the `<Item>` elements, the parent of all of the `<Item>` elements is the `<ShoppingCart>` element. Therefore, the full path to any of the `<Item>` elements is `ShoppingCart/Item`. In the same way, if the `<Item>` element had an element nested beneath it, say `<Price>`, then the path to that element would be `ShoppingCart/Item/Price`.

A Digester rule is attached to a given path and will fire any time an element with that path is encountered. You can have multiple rules attached to a given path, and multiple rules can fire for any given path.

In this example, our first rule—an `ObjectCreate` rule—is defined to fire for the path `ShoppingCart`. This means that when the `<ShoppingCart>` element is encountered, an instance of the class `myApp.ShoppingCart` will be created.

Digester uses a stack implementation to deal with the objects it creates. For instance, when the `ObjectCreate` rule fires and instantiates that `ShoppingCart` object, it is pushed onto the stack. All subsequent rules will work against that object, until it is popped off the stack (either explicitly, as a result of another rule, or because parsing is completed). So, when the next rule—the `SetProperties` rule—fires, it will set all the properties of the object on the top of the stack—in this case, our `ShoppingCart` object—using the attributes of the `<ShoppingCart>` element in the document.

Next comes another `ObjectCreate` rule set up for the `<Item>` elements, and also another `SetProperties` rule for that same element. So, when the first `<Item>` element is encountered, the object is created and pushed onto the stack, meaning it is now on top of the `ShoppingCart` object.

The last rule is the `SetNext` rule. This rule calls a given method—`addItem()` in this case—on the *next* object on the stack, which would be the `ShoppingCart` object, passing it the object on the top of the stack, which is the `Item` object. At the end of this, the `Item` object on the top of the stack is popped off, revealing the `ShoppingCart` object, which is again the top object on the stack. This process repeats three times for each `<Item>` element.

At the end, the object on the top of the stack, which would be our `ShoppingCart` object at that point, is popped off and returned by Digester. We catch that return into the `shoppingCart` variable, and we now have a reconstituted shopping cart in the same state the user left it.

Interestingly, Digester uses Simple API for XML (SAX) under the covers. SAX is a Java-based, event-driven API for parsing XML, like Digester (which stands to reason!), but functions at a lower level and tends to be quite a bit more work than Digester to use. Since this book strives to be language-neutral as much as possible (with the exception of JavaScript, of course!), I won't go into an example of using SAX directly. However, it is important to know that with SAX, as an XML document is parsed, various events occur. Some of the events are when the document begins or ends, when a new tag is encountered, when text that a tag pair wraps is parsed, and when a tag is closed—to name a few. As the developer, you write what is called a document handler class, which is the class that will be called when these events occur. Digester is essentially a document handler class. It builds on the SAX parsing events, extending them in a sense.

While it would certainly be feasible to build our own JavaScript implementation of SAX, why bother when someone else has already done so? JSLib from the Mozilla Foundation (`http://jslib.mozdev.org`) provides such an implementation. JSLib, as the name suggests, is a library of JavaScript functions covering a range of needs, one of which is SAX, because, of course, everyone needs a little SAX every now and again (come on, how could I possibly not make that joke at *some* point in this chapter?).

You will shortly see how the SAX component of JSLib is used as we dissect JSDigester. In fact, why put it off any longer?

Dissecting the JSDigester Solution

Let's begin by executing the test code to see what JSDigester actually does. Figure 7-2 shows the screen that you will see when you load the test page in a browser and click the button to initiate the test.

Before we look at the code of JSDigester itself, it would, naturally enough, be useful to look at the code that will test JSDigester.

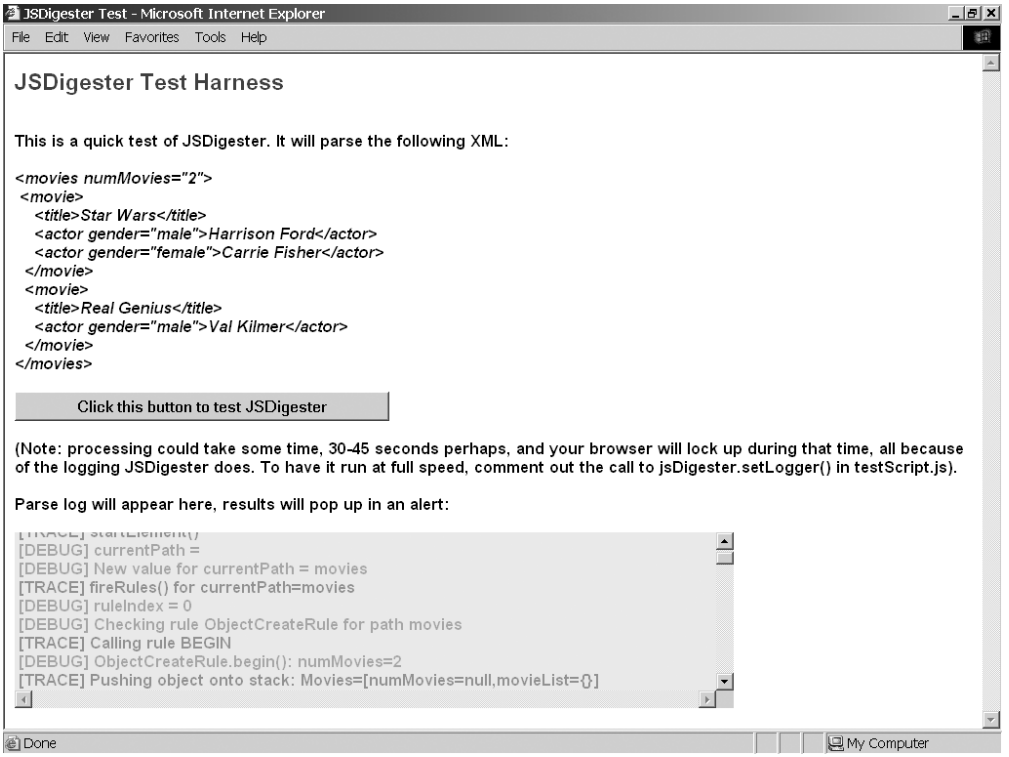


Figure 7-2. *The result of testing JSDigester*

Writing the Test Code

The `JSDigesterTest.htm` file is where the tests begin, and it is this file that you will load in your browser. I'm not going to show this code here in the interest of saving space, but do take a look. It is really just some straightforward HTML with some JavaScript imports. The real interesting stuff happens in the JavaScript files.

The first import, `sax.js`, is the SAX parser from JSLib. I will not be going into the details of how this code works because it is not code I wrote. I do recommend going through it on your own though, and getting a feel for it. Generally, it is not terribly difficult to understand, and at just over 200 lines of code, it isn't anything too overwhelming. Still, it is a library we are making use of, and as is usually the case with libraries, we're less concerned with what goes on inside it and how it works its magic than we are with how we use it in our own code, so that will be our focus here.

However, one detail to be aware of however is that when you use JSLib, you provide something called a `DocumentHandler` callback. This is essentially an object that will react to well-defined life-cycle events, which are encountered as an XML stream is parsed. This object must adhere to a known interface; that is, must implement a number of functions. You'll see how this interface is implemented in short order, but as far as using JSLib goes, this is about the extent of the knowledge you need.

Getting on with the code, take a look at Listing 7-5. It shows three JavaScript classes: one class represents a collection of movies, another represents an individual movie, and the third represents an actor. This code is found in the `testClasses.js` file. These classes are what JSDigester will instantiate and populate as it does its work.

Listing 7-5. *Three JavaScript Classes JSDigester Will Create and Populate*

```
// This class represents an Actor in a Movie.
function Actor() {
  this.gender = null;
  this.name = null;
}
Actor.prototype.setGender = function(inGender) {
  this.gender = inGender;
}
Actor.prototype.getGender = function() {
  return this.gender;
}
Actor.prototype.setName = function(inName) {
  this.name = inName;
}
Actor.prototype.getName = function() {
  return this.name;
}
Actor.prototype.toString = function() {
  return "Actor=[name=" + this.name + ",gender=" + this.gender + "]";
}


// This class represents a Movie.
function Movie() {
  this.title = null;
  this.actors = new Array();
}
```

```
Movie.prototype.setTitle = function(inTitle) {
  this.title = inTitle;
}
Movie.prototype.getTitle = function() {
  return this.title;
}
Movie.prototype.addActor = function(inActor) {
  this.actors.push(inActor);
}
Movie.prototype.getActors = function() {
  return this.actors;
}
Movie.prototype.toString = function() {
  return "Movie=[title=" + this.title + ",actors={" + this.actors + "}]";
}


// This class stores a collection of Movies.
function Movies() {
  this.movieList = new Array();
  this.numMovies = null;
}
Movies.prototype.setNumMovies = function(inNumMovies) {
  this.numMovies = inNumMovies;
}
Movies.prototype.getNumMovies = function() {
  return this.numMovies;
}
Movies.prototype.addMovie = function(inMovie) {
  this.movieList.push(inMovie);
}
Movies.prototype.getMovieList = function() {
  return this.movieList;
}
Movies.prototype.toString = function() {
  return "Movies=[numMovies=" + this.numMovies + ",movieList={" +
  this.movieList + "}]";
}
```

Figure 7-3 shows a quick bit of UML so you can visualize this small class hierarchy.

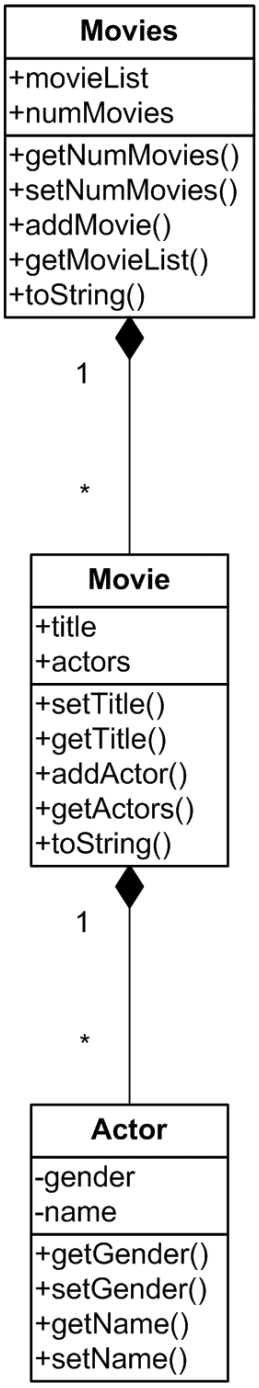


Figure 7-3. *The test class hierarchy in UML*

These three classes don't have much in the way of behaviors; they are more or less just containers in which to store data.

The `Actor` class has two attributes: `gender` and `name`, along with the associated getter and setter methods for each (this is typical of JavaBeans, if you are familiar with Java). The `Actor` class also provides an overridden `toString()` method, which is a method every object in JavaScript has by default. This version renders a slightly more meaningful string representation of the `Actor` object.

The `Movie` class has the two attributes `title` and `actors`. `actors` is an array of `Actor` objects associated with the `Movie`. This class also has an overridden `toString()` method.

Lastly, the `Movies` class also has two attributes: `movieList` and `numMovies`. `movieList` is an array of `Movie` objects, and `numMovies` is the number of movies contained in that array. While it is true that this information is intrinsic in the array, within the `length` property, I did it this way to demonstrate some of the parsing capabilities of JSDigester (so don't think too hard on it, because it isn't meant to be the best implementation). Just like the `Actor` and `Movie` classes, `Movies` has its own `toString()` method, the output of which you will see when you fire up the test code.

Now, let's look at the test code that will exercise JSDigester. This code is shown in Listing 7-6 and is found in the `testScript.js` file.

Listing 7-6. *JavaScript Code That Tests JSDigester*

```
function testJSDigester() {

  // Create a string of test XML to have JSDigester parse.
  var sampleXML = "";
  sampleXML += "<movies numMovies=\"2\">\n";
  sampleXML += "  <movie>\n";
  sampleXML += "    <title>Star Wars</title>\n";
  sampleXML += "    <actor gender=\"male\">Harrison Ford</actor>\n";
  sampleXML += "    <actor gender=\"female\">Carrie Fisher</actor>\n";
  sampleXML += "  </movie>\n";
  sampleXML += "  <movie>\n";
  sampleXML += "    <title>Real Genius</title>\n";
  sampleXML += "    <actor gender=\"male\">Val Kilmer</actor>\n";
  sampleXML += "  </movie>\n";
  sampleXML += "</movies>";

  // Create a logger for JSDigester to use, and set its level to TRACE, and tell
  // it where to log to.
  var log = new jscript.debug.DivLogger();
  log.setLevel(log.LEVEL_TRACE);
  log.setTargetDiv(document.getElementById("divLog"));

  // Instantiate a JSDigester instance and set up the rules, and logger.
  var jsDigester = new JSDigester();
  jsDigester.setLogger(log);
  jsDigester.addObjectCreate("movies", "Movies");
```

```
  jsDigester.addSetProperties("movies");
  jsDigester.addObjectCreate("movies/movie", "Movie");
  jsDigester.addBeanPropertySetter("movies/movie/title", "setTitle");
  jsDigester.addObjectCreate("movies/movie/actor", "Actor");
  jsDigester.addSetProperties("movies/movie/actor");
  jsDigester.addBeanPropertySetter("movies/movie/actor", "setName");
  jsDigester.addSetNext("movies/movie/actor", "addActor");
  jsDigester.addSetNext("movies/movie", "addMovie");

  // Parse the XML, resulting in an instance of the Movies class.
  var myMovies = jsDigester.parse(sampleXML);

  // Construct result string.
  var outStr = "JSDigester processed the specified XML." +
    "\n\nIt created an object graph consisting of a Movies object, " +
    "with a numMovies property, and containing a collection of " +
    "Movie objects." +
    "\n\nEach Movie object has a title property, and " +
    "contains a collection of Actor objects.\n\n" +
    "Each Actor object has two fields, name and gender.\n\n" +
    "Here's the final Movies object JSDigester returned: \n\n" +
    myMovies;

  // Display results.
  alert(outStr);

}
```

The `onClick` event of the button in `JSDigesterTest.htm` calls the `testJSDigester()` function seen here. The first thing it does is build up a string that is the XML JSDigester will parse for us. I believe that is pretty self-explanatory.

After that comes the instantiation and configuration of a `DivLogger` instance (`DivLogger` was introduced in Chapter 3). We set the logging level to trace, so that we can see absolutely everything JSDigester does, and we also give it a reference to the `<div>` where our log output will go.

Next, we instantiate JSDigester itself, and pass it the logger we just configured. Note that if you do not pass a logger to JSDigester, it will still work. In fact, this is exactly what I recommend you do in a production environment. As you will see when you actually try out the test page, the logging slows JSDigester significantly. Therefore, you should generally pass a logger instance in only when you are debugging and can accept the delay logging causes. JSDigester actually performs pretty well when logging is not enabled, as long as the input XML isn't too large. With logging on, it definitely doesn't perform as well, to say the least.

After that comes the part that really makes JSDigester work: the rules. The first rule configured is an `ObjectCreateRule` that will fire when the `<movies>` element is encountered in the XML. This will result, not surprisingly, in an instance of the `Movies` class being created. Related to that is the next rule, the `SetProperties` rule. This will call setter methods on that newly created `Movies` object for each attribute of the `<movies>` tag—in this case, just `numMovies`.

Next, we find another `ObjectCreateRule`, this time mapped to the path `movies/movie`. This will create an instance of the `Movie` class when any `<movie>` element that is a child of the `<movies>` element is encountered. Working in tandem with that is the `BeanPropertySetter` rule that follows. This fires when the `movies/movie/title` path, which correlates to a `<title>` element as a child of a `<movie>`, is encountered. It will call the `setTitle()` method of the `Movie` object just created, passing it the value of the `<title>` element.

After that comes a group of three rules that handles the `<actor>` elements. First, as you've probably come to expect by now, is an `ObjectCreateRule`. After that is a `SetPropertiesRule`. So at this point, JSDigester knows how and when to create instances of the `Actor` class, and it also knows to take any of the `<actor>` elements' attributes and call the corresponding setter methods for each to set those properties.

Lastly, we have another `BeanPropertySetter` rule mapped to the path `movies/movie/actor`. The text contained between the opening and closing `<actor>` tags is the actor's name. So, this rule will take that text and call the `setName()` method on the `Actor` instance, as the rule specifies.

By this point, JSDigester has enough information to create objects for us and populate them from the XML. The last step then is to let JSDigester know about the hierarchy of objects; that is, how to assign `Actor` objects to `Movie` objects and how to add `Movie` objects to the `Movies` object, which is ultimately the object that will be returned to the caller.

Whoa, hold up—let's not gloss over that. Why exactly is it that JSDigester, at the end of its processing, will return the `Movies` instance to us? Let's take a look at the overall flow to see how that happens.

Understanding the Overall JSDigester Flow

Recall earlier in the discussion of Digester when I mentioned that it uses a stack implementation to do its work? Well, this is how we end up getting the `Movies` instance at the end of the test. Every time Digester creates a new object, it pushes it onto the stack. It's a first in, last out (FILO) stack, so as objects are created and pushed on, the first object created will always be on the bottom. Therefore, as objects are popped off the stack, which is what happens as the XML is parsed, the last object will ultimately be uncovered, which is the very first object created, and this is what JSDigester returns.

Remember the first rule we configured? It was mapped to the `<movies>` element, which happens to be the root of the XML document. It kind of makes sense that the last object returned would be the root object, doesn't it? All the other objects get rolled up into the root after all, just as the structure of the XML document dictates.

So, how exactly does an `Actor` object get added to a `Movie` object, and a `Movie` object get added to the `Movies` object? Both of those things happen as a result of the `SetNext` rules, which are the last two rules added to JSDigester. A `SetNext` rule uses the nature of the stack to its advantage by calling a specified setter method on the *next* object on the stack. To understand this, let's walk through the sequence of events when parsing our test XML. Figure 7-4 shows the sequence of events in flowchart form.

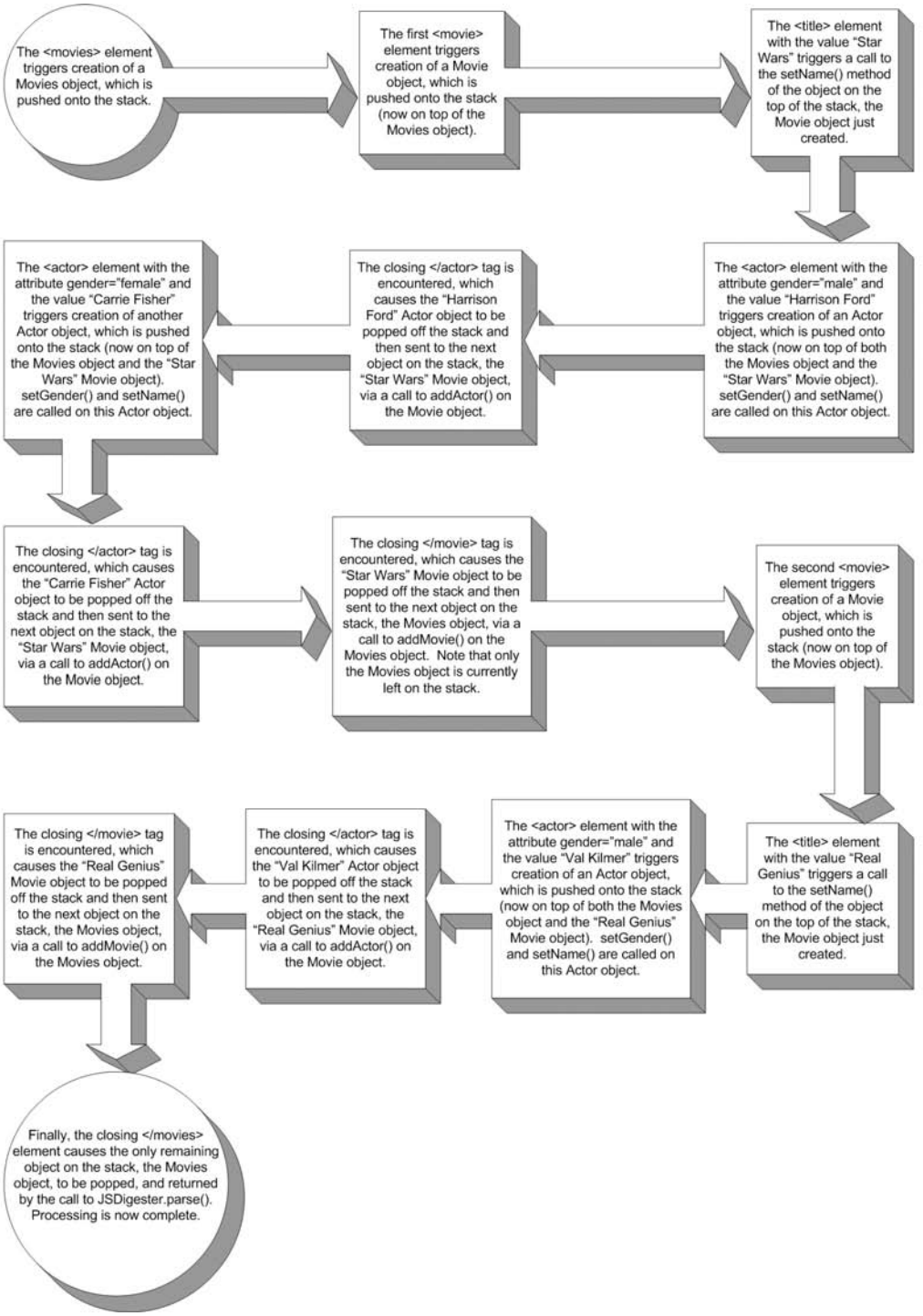


Figure 7-4. *JSDigester parsing sequence of events*

At this point, you should have a pretty good idea of how JSDigester works and an insight into the test code that makes sure it works. Only one small, minor, tiny detail remains: exploring JSDigester itself.

Writing the JSDigester Code

JSDigester checks in at about 678 lines of code,[1] so I won't be listing it here in its entirety. I will be calling out the parts of interest as we proceed, so you'll find it helpful to have the full listing in front of you.

Ready (Preparing to Parse) . . .

The first 15 or so lines of actual code in the `JSDigester.js` file, which is where the `JSDigester` class (and rules classes as well) is found, is a batch of class members. The first three are some constants that are used internally to determine which event is occurring for a given element in the incoming XML: the start of the element (`EVENT_BEGIN`), the body text of the element (`EVENT_BODY`), or the closing of an element (`EVENT_END`). The word *constant* is, of course, a bit of a misnomer in JavaScript because there are no true constants; code can come along and change these values as desired. Still, conceptually, this is what they are and how they are treated throughout.

After that is a variable that appears rather often: `currentPath`. Recall that as we parse through our XML, each element can be referenced by a path, like `movies/movie/actor` in our previous example, to reference an `<actor>` element below a `<movie>` element, which is itself a child of the root `<movies>` element. `currentPath` is how JSDigester keeps track of where it is as the XML is parsed.

Next up we find an array named `rules`. This is simply a collection of all the rules that are added to this JSDigester instance to process.

After that is our `objectStack`, which will, of course, be at the center of all the JSDigester activity.

`rootObject` is the next item we encounter, and it is a reference to the root object on the stack. This is done so that it is easy to return the root object when the time comes. This will change as the XML is processed. As objects are popped from the stack, each one essentially becomes the root object, at least temporarily. By the end though, the *true* root object will be referenced, and it's a simple matter of returning that object.

Following `rootObject` is the variable `log`. This, as I'm sure you expect, is a reference to the logger instance that JSDigester is to use during its processing. As mentioned during our discussion of the test code, logging is a very expensive operation, primarily in this implementation because of the constant rewrites of the `innerHTML` property of some `<div>` elements. Therefore, by default, `log` is set to null, which means no logging will occur. Therefore, by default, a JSDigester instance is properly configured for maximum performance.

The next class member encountered is `saxParser`, which is a reference to the SAX parser instance from JSLib that will process the XML for us. Notice that a new instance is instantiated and assigned to this variable right then and there—no sense putting it off. However, you should see an obvious flaw with that: if we wanted to use another SAX parser, there's no ready way to do it. The code using JSDigester could cheat and set `saxParser` to another object with the same

1. Interestingly, less than half of that number, 316 to be precise, is the executable code. The rest is whitespace and comments—not all that much for what it does, I think.

interface as `SAXParser`, but that's not the best answer. Just like the first appearance of the Shadows,[2] you should have a sense of what is to come: you'll see this as a suggested exercise later.

Following all these class variable declarations and initializations comes the last line of code in the JSDigester constructor function: a call to `init()`. The `init()` function itself is very simple but also very important:

```
JSDigester.prototype.init = function() {

  // Tell the SAX parser that this instance of JSDigester is the document
  // handler.
  this.saxParser.setDocumentHandler(this);

} // End init().
```

In JavaScript, one way to construct a class is to use the `prototype`. Every object, of which a function is one kind, has a `prototype` associated with it. By assigning a field—`init` in this case—to the `prototype` of the existing JSDigester function and giving it the value of a function (I know that sounds weird), we are essentially adding a function named `init()` to JSDigester. But because we are adding it to the `prototype` of JSDigester, that means that every time you instantiate a new JSDigester object, it will contain that `init` member, which happens to be a function. This is a common approach for creating objects and then extending them in JavaScript.

What's more bizarre here though is that even before you instantiate a JSDigester object, you have one already, by virtue of the function definition—the one containing all the member variables. However, if you were to examine it before the preceding code executes, you would find that it does not contain the `init()` function. Only JSDigester objects instantiated after this code executes will have that function. Remember that JavaScript is a very dynamic language, and somewhat unusual situations like this occur frequently.

Moving right along, recall that the SAX parser is responsible for actually parsing the XML. However, what happens as each element is encountered is still our responsibility. In SAX parlance, you need to tell the parser what class will act as the document handler. That class will receive callbacks throughout the parsing process when various events occur, such as an element first being encountered, text between two tags being encountered, or a closing tag being encountered. JSDigester itself is the document handler here because it will be dealing with those events to work its magic, so we pass the implicit `this` reference to `setDocumentHandler()` of the `saxParser` instance. That's about all the initialization required for JSDigester.

After the `init()` method comes the `setLogger()` method. This is a typical property setter method that accepts a reference to a logger instance for JSDigester to use.

Set (Kicking Off the Main Process) . . .

Next up we have the `parse()` method, the main entry point to JSDigester for all intents and purposes. Let's have a look at it, shall we? Listing 7-7 shows that code.

---

2. The Shadows were the ancient, technologically advanced race that was the main antagonist on the television series *Babylon 5*. In the first season episode "Signs and Portents," we got our first glimpse of the Shadows, and saw just how powerful they were. This was the episode that truly hooked many fans, yours truly included. It was a great hint of things to come, very true to its title. Have a look: `http://www.imdb.com/title/tt0517690`.

Listing 7-7. *Where All the Action in JSDigester Begins: The parse() Method*

```
JSDigester.prototype.parse = function(inXMLString) {

  if (this.log) {
    this.log.trace("JSDigester.parse()...");
    this.log.debug("inXMLString = " + inXMLString);
  }
  // Remove all items from the object stack, just in case this isn't the
  // first parse this instance of JSDigester has performed.
  this.objectStack.splice(0);
  // Clear current path and root object.
  this.currentPath = "";
  this.rootObject = null;
  // Ask the SAX parser to parse the incoming XML.
  if (this.log) {
    this.log.debug("Calling SAX parser...");
  }
  this.saxParser.parse(inXMLString);
  if (this.log) {
    this.log.debug("SAX parser returned");
  }
  // Return the root object on the stack.
  if (this.log) {
    this.log.debug("Returning root object: " + this.rootObject);
  }
  return this.rootObject;

} // End parse().
```

The first line is something you will see repeated throughout the code. Remember that I said that JSDigester would continue to work, and in fact would work optimally, when no logger is supplied? This line is what makes that statement true. This is akin to a code guard in Java, where you check if a certain log level is enabled before attempting to log a message, usually one that is expensive, which generally means involving string concatenations. This avoids overhead in the logging call, and the same is true here. Aside from making it efficient, it obviously avoids errors, too. If no logger instance were passed in, we would get an error trying to reference a null object—the `this.log` reference variable in this case.

After the opening salvo of logging messages, we find the line:

```
this.objectStack.splice(0);
```

The `splice()` method is used to remove elements from an array. The special case of passing a zero to it results in the array being cleared. We do this here because if this JSDigester instance is being reused, the stack needs to be clear before we begin. In the same vein, the next two lines reset the `currentPath` field and the `rootObject` reference to their initial states. Reusing a JSDigester instance can save a bit of overhead, if the rules are the same, so we definitely want to allow for it, and these three lines take care of that.

The next line is what kicks off the actual work: the call to the SAX parser's `parse()` method. The XML to be parsed is passed in, and the parser goes to work. From that point on, JSDigester will be called back to handle various events, as you'll soon see. The last line in `parse()` (JSDigester's `parse()` that is, not the SAX parser's `parse()`) returns the root element, just as we expect from looking at the test code.

Two quick utility functions follow: `pop()` and `push()`. These are the methods that handle pushing objects onto and popping objects off the stack. Every array in JavaScript supplies a `push()` and `pop()` method, so we can therefore treat any array as a stack. Wrapping these methods in our own methods in JSDigester allows us to handle any extra requirements around these operations, such as logging. And, aside from `pop()`, which always sets the `rootObject` field to point to the object last popped off the stack, that's about all these do.

After those functions, we find the `startDocument()` method. This is the callback that the SAX parser will call on when the XML document begins to be parsed. There isn't anything for JSDigester to do for this event, so there is just a log message to say the event fired. This method must be present, however, for JSDigester to fulfill the `DocumentHandler` callback contract.

The `startElement()` method, shown in Listing 7-8, is where some real work begins to occur.

Listing 7-8. *The startElement() Method of JSDigester*

```
JSDigester.prototype.startElement = function(inName, inAttributes) {

  if (this.log) {
    this.log.trace("startElement()");
  }
  // If this is not the first element encountered, start by adding a forward
  // slash (the path separator).
  if (this.currentPath != "") {
    this.currentPath += "/";
  }
  if (this.log) {
    this.log.debug("currentPath = " + this.currentPath);
  }
  // Build up the path of the current element.
  this.currentPath += inName;
  if (this.log) {
    this.log.debug("New value for currentPath = " + this.currentPath);
  }
  // Fire all the rules associated with this element.
  this.fireRules(this.EVENT_BEGIN, inName, inAttributes, null);

} // End startElement().
```

First, after the typical log message, we determine if this is the first element—the root element—of the XML document. If it isn't, then we need to add a forward slash to the current path, which allows us to build up the path. So, with our sample XML, `currentPath` begins blank, of course. The first element encountered will be `<movies>`, so `currentPath` would be set to `movies`. When the first `<movie>` element is encountered, since `currentPath` is no longer blank when `startElement()` is called, it will append a forward slash, so `currentPath` would be `movies/`. After that, the name of

the element that has begun, which is passed in to `startElement()`, is appended, so that `currentPath` would at that point be `movies/movie`. This building up of `currentPath` continues any time `startElement()` is called (and as you'll see later, it is cut down, so to speak, when `endElement()` is called).

The last line of `startElement()` is the most important. It calls the `fireRules()` method of JSDigester. It passes `fireRules()` the constant `EVENT_BEGIN`, since, of course, `startElement()` is called when an element begins. It also passes it the name of the element and the collection of attributes that element contains, as an associative array. Hold that thought for just a moment; we'll be looking at `fireRules()` in detail shortly.

In the same vein as `startElement()` is the next method, `characters()`. This is called when the text contained between two tags is parsed. So, for example, `<title>Star Wars</title>` would result in a call to `characters()` with the `inText` parameter having the value `"Star Wars"`. It too calls `fireRules()`, but this time passing the constant `EVENT_BODY`, and just the text that was passed in to `startElement()`.

Following `characters()` comes the `endElement()` function, as alluded to earlier. Its job, firstly, is to call `fireRules()`, this time passing the `EVENT_END` constant to indicate that the rules applicable to the closing of the tag should now be executed. Its next task is to remove the last element from `currentPath`, which corresponds to the element that just ended (remember that the elements form a hierarchy, so the closing of one element will always happen before the close of its parent). Here's the code that literally removes the element from `currentPath`:

```
var i = this.currentPath.lastIndexOf("/");
this.currentPath = this.currentPath.substr(0, i);
```

It simply finds the last occurrence of the forward slash character, which will always precede the current element's name, and then sets `currentPath` to the substring of `currentPath` right up to, but not including, that forward slash. It's neat and clean.

After that we find the `endDocument()` method which, just like `startDocument()`, is required by the SAX parser's `DocumentHandler` interface contract, but serves no real purpose as far as JSDigester goes. So, enough said.

Go (the Real Bulk of the Work)!

Now we finally come to `fireRules()`. But before that can be discussed, I need to point out something about rules, even though I'd be willing to bet you've figured it out already.

Some rules do something when an element begins; `ObjectCreateRule` is a good example of this. Other rules do something when text is encountered; `BeanPropertySetter` is one example. Still other rules do something only when an element ends; `SetNextRule`, for example, works that way. However, all of the rules are called three times per element: once when it starts, once when its body text, if any, is encountered, and once when it closes. We'll be taking a look at rules when we're finished with JSDigester, but as a preview, you will find that every rule has three methods corresponding to these events: `begin()`, `body()`, and `end()`. You will also find that in general, two of the three will do nothing (there's nothing to stop a rule from doing something in response to more than one event, but none of the existing rules do). This information will be important to understanding `fireRules()`. Speaking of which, you can see `fireRules()` in Listing 7-9.

Listing 7-9. *The True Core of JSDigester: The fireRules() Method*

```
JSDigester.prototype.fireRules = function(inEvent, inName, inAttributes,
  inText) {

  if (this.log) {
    this.log.trace("fireRules() for currentPath=" + this.currentPath);
  }
  var ruleIndex = 0;
  if (inEvent != this.EVENT_BEGIN) {
    ruleIndex = this.rules.length - 1;
  }
  if (this.log) {
    this.log.debug("ruleIndex = " + ruleIndex);
  }
  for (var ruleCounter = 0; ruleCounter < this.rules.length; ruleCounter++) {
    var rule = this.rules[ruleIndex];
    if (this.log) {
      this.log.debug("Checking rule " + rule.getRuleType() +
        " for path " + rule.getPath());
    }
    if (rule.getPath() == this.currentPath) {
      switch (inEvent) {
        case this.EVENT_BEGIN:
          if (this.log) {
            this.log.trace("Calling rule BEGIN");
          }
          rule.begin(inAttributes);
        break;
        case this.EVENT_BODY:
          if (this.log) {
            this.log.trace("Calling rule BODY");
          }
          rule.body(inText);
        break;
        case this.EVENT_END:
          if (this.log) {
            this.log.trace("Calling rule END");
          }
          rule.end(inName);
        break;
      }
    } else {
      if (this.log) {
        this.log.debug("Rule not applicable to this path, so not firing");
      }
    }
```

```
    if (inEvent == this.EVENT_BEGIN) {
      ruleIndex++;
    } else {
      ruleIndex--;
    }
  }

} // End fireRules().
```

So, what's going on here? Well, first we check to see what kind of event it is. We have a variable named `ruleIndex`, which is the index into our array of rules. If the event isn't for an element opening, then we need to go through the rules in reverse order to ensure that they fire in the order they should, so we set `ruleIndex` to point to the last rule in the array. For an event corresponding to the start of an element, we need to run through the rules in forward order, so `ruleIndex` is zero in that case. The reason for this reversal of order is that the rules must fire against the order the objects they would work against appear on the stack. So, executing a `SetNext` rule should work against the next object in the stack, and this wouldn't be the case going through the stack in forward order. For the opening of a tag, however, the stack needs to be built up initially, and going through them in forward order is what accomplishes that.

Then we begin a loop, a single iteration per rule in our array of rules. We first pull out the next rule during the iteration, using `ruleIndex` to get the appropriate next rule. We then check to see if the rule's path corresponds to the current path of the element being processed. If it does, we check what type of event we're processing, and call the appropriate method of the rule: `begin()`, passing it the attributes of the current element; `body()`, passing it the text the current element contains; or `end()`, passing it the name of the element.

Lastly, we either increment or decrement `ruleIndex` as appropriate. We increment it if the event is a begin event, or decrement it for any other event.

As simple as it seems, and it *is* pretty simple, this is the method that really makes JSDigester go, as stated previously. I never said JSDigester was technically all that impressive, but like the wheel, it really makes life quite a bit better (well, "life" as defined by a developer, that is).

The last four methods in JSDigester are all pretty similar, so I will cover them together. `addObjectCreate()`, `addSetProperties()`, `addBeanPropertySetter()`, and `addSetNext()` are the methods that a user of JSDigester calls to add a processing rule, as you saw in the sample code in Listing 7-6. Each one has a different set of parameters, since each rule requires different information. They all share `inPath` in common though, and this is the path to the element in the XML to process. The `addObjectCreate()` rule also needs the name of the class to instantiate, and is provided that with the `inClassName` parameter. The `addSetProperties()` method requires only the path. `addBeanPropertySetter()`, in addition to the path, requires the name of the property to call on the object whose property is being set, and `inMethod` provides that. Lastly, `addSetNext()` requires `inMethod`, the name of the method to call on the next object on the stack, in addition to the path.

Each of these methods instantiates an object of the appropriate rule class—`ObjectCreateRule`, `SetPropertiesRule`, `BeanPropertySetterRule`, or `SetNextRule`—and passes in the pertinent information upon construction. It then pushes this new rule instance into the array of rules configured for this instance of JSDigester. Aside from some log output, that's the full extent of what these methods do.

And with that, we've seen everything JSDigester has to offer in the way of code. Only one thing remains, and that's to look at the four rule classes themselves. I think you'll be surprised how little code is actually involved in them, much like JSDigester itself.

Writing the Rules Classes Code

Each of the four existing JSDigester rule classes share the same basic structure, so in the interest of brevity, I will show a full one here, and after that show only where the other three differ. For no real reason other than random chance, I've chosen `ObjectCreateRule` as our example. Its code is shown in Listing 7-10.

Listing 7-10. *The JSDigester ObjectCreateRule Class*

```
function ObjectCreateRule(inPath, inClassName, inJSDigester) {

  // Set rule type and path to fire for.
  this.ruleType = "ObjectCreateRule";
  this.path = inPath;
  // Set the JavaScript class to instantiate.
  this.className = inClassName;
  // Record the JSDigester instance that the instance of this class belongs to.
  this.jsDigester = inJSDigester;

} // End ObjectCreateRule().


/**
 * Return the ruleType.
 */
ObjectCreateRule.prototype.getRuleType = function() {

  return this.ruleType;

} // End getRuleType().


/**
 * Return the path.
 */
ObjectCreateRule.prototype.getPath = function() {

  return this.path;

} // End getPath().
```

```
/**
 * Begin an element.
 */
ObjectCreateRule.prototype.begin = function(inAttributes) {

  if (this.jsDigester.log) {
    this.jsDigester.log.debug("ObjectCreateRule.begin(): " + inAttributes);
  }
  var protoObj = eval(this.className);
  this.jsDigester.push(new protoObj());

} // End begin().


/**
 * Process body text.
 */
ObjectCreateRule.prototype.body = function(inText) {

  if (this.jsDigester.log) {
    this.jsDigester.log.debug("ObjectCreateRule.body(): " + inText);
  }

}


/**
 * Process closing tag.
 */
ObjectCreateRule.prototype.end = function(inName) {

  if (this.jsDigester.log) {
    this.jsDigester.log.debug("ObjectCreateRule.end(): " + inName);
  }
  this.jsDigester.pop();

} // End end().
```

A JSDigester rule always contains three fields:

- `ruleType`: The name of the rule. The only real use of this value is in logging.
- `path`: The path in the XML structure for which this rule should fire.
- `jsDigester`: The instance of JSDigester, which is passed in to the constructor when the rule is instantiated.

All rule classes always provide a `getRuleType()` method, which just returns the `ruleType` value, and `getPath()`,which returns the `path` value.

After those fields come three methods: `begin()`, `body()`, and `end()`. As described when we looked at JSDigester's `fireRules()` method, the `begin()` method is called when an element is first encountered; that is, the opening tag. `body()` is called when the text between two tags is fully parsed. `end()` is called when a closing tag is encountered.

Recall that a rule, generally speaking (and specifically for the four existing rules), handles only one of these three events. `ObjectCreateRule` does work only in response to the begin event. Well, I suppose that isn't *technically* accurate. It also pops the created object off the stack when the end event occurs, but that's a relatively minor thing. Still, I suppose in the interest of complete accuracy, it should be stated that `ObjectCreateRule` does actually process *two* events, even if one is a minor thing.

If you look in the other three rules (`ObjectCreateRule` is the exception), you will see that two of the three methods are nothing but log statements. The remaining one does some actual work though. In the case of `ObjectCreateRule`, it's pretty simple: instantiate a new instance of the class named and push it on the stack. The instantiation is done by simply `eval()`'ing the `className` value, which results in a new instance of that class being created.

But how did it know the class to create? By its `className`, obviously. But where did that come from? Well, remember that there are three fields that all rules share in common: `ruleType`, `path`, and `jsDigester`. While those fields are sufficient for some rules, such as the `SetPropertiesRule`, other ones, including our `ObjectCreateRule` example, need more fields. This is not a problem. We simply add the field and add it to the list of parameters for the constructor, and the problem is solved. In the case of `ObjectCreateRule`, we find a `className` field, and we see that it is the second parameter passed in to the constructor, `inClassName`.

For the `SetPropertiesRule` class, the work is done in the `begin()` method, and it's a fair bit more substantial than the `ObjectCreateRule`:

```
SetPropertiesRule.prototype.begin = function(inAttributes) {

  if (this.jsDigester.log) {
    this.jsDigester.log.debug("SetPropertiesRule.begin(): " + inAttributes);
  }
  var obj = this.jsDigester.pop();
  for (var i = 0; i < inAttributes.length; i++) {
    var nextAttribute = inAttributes[i];
    var keyVal = nextAttribute.split("=");
    var key = keyVal[0];
    var val = keyVal[1];
    key = "set" + key.substring(0, 1).toUpperCase() + key.substring(1);
    if (this.jsDigester.log) {
      this.jsDigester.log.debug("SetPropertiesRule.begin() - key=" + key +
        "val=" + val);
    }
    obj[key](val);
  }
  this.jsDigester.push(obj);

} // End begin().
```

So, what this code accomplishes is taking in an array representing the attributes of the element being parsed in the form of *name=value* pairs, and setting each property on the object on the top of the stack. To do this, it first pops the object off the top of the stack. It then iterates over the array of attributes passed in. For each, it splits the array contents on the equal sign character. The first element of the array resulting from the call to `nextAttribute.split()` is what was on the left of the equal sign, which is the name of the attribute, and the second element is what was on the right, which is the value of the attribute. We then see this line of code:

```
key = "set" + key.substring(0, 1).toUpperCase() + key.substring(1);
```

Its job is to construct the name of the method to call, using standard JavaBean format, which is `get`*XXXX*, where *XXXX* is the name of the property to set (the first character is capitalized, and the rest are in whatever case they are in the property's name). So, for the element `<actor>`, which has a `gender` attribute, the name of the method formed would be `setGender()`. Once the method name is formed, it is executed with this line:

```
obj[key](val);
```

This is actually a really nifty feature of JavaScript. You see, you can treat any JavaScript object like an associative array, where the names of the elements in the array are the members of the object. So, let's say `obj` is a reference to an `Actor` object. If we then do `obj["setGender"]`, that gets us a reference to the member `setGender()` function of that `Actor` object. Again, we are treating the `Actor` object like an associative array, and `setGender` is the name of the element to which we want a reference. If we then append `(val)` to that, we are creating a function call dynamically. We can use a variable in place of that hard-coded `setGender` value, which is precisely what the code in the rule class does. Think about how tricky that would be in something like Java, using all sorts of introspection and reflection to do the equivalent of that one simple line of code, and then tell me JavaScript isn't cool.

After all the properties have been set, the object is pushed back onto the stack, returning it to the state it was in before this rule fired, and at that point, the rule has finished its work.

Moving right along to the `BeanPropertySetter` rule, we find another additional field is needed: the `setMethod` field, which names the method to call to set the value on the object. In this rule, it's in the `body()` method where the action happens, and again we see something very similar to the dynamic method call technique used in the `SetPropertiesRule` class. The object on the top of the stack is again popped off, and virtually the same single line of code sets the property:

```
obj[this.setMethod](inText);
```

In this case, just a single property is being set, and its name is stored in the `setMethod` field. So, there is no loop as in `BeanPropertySetter`, and no incoming associative array to rip apart. There is just one simple call, passing it the text that was passed to the `body()` method, and we're finished, simple as that.

The last rule to look at is the `SetNextRule`. Just like `BeanPropertySetter`, it requires the name of the method to call, so we see a `setMethod` field here as well, and the corresponding constructor parameter `inMethod`. In this case, the work is done in the `end()` method, and it's actually a little bit tricky (well, *interesting* is probably the better description).

```
SetNextRule.prototype.end = function(inName) {

  if (this.jsDigester.log) {
    this.jsDigester.log.debug("SetNextRule.end(): " + inName);
  }
  var childObj  = this.jsDigester.pop();
  var parentObj = this.jsDigester.pop();
  if (this.jsDigester.log) {
    this.jsDigester.log.debug("SetNextRule.end() - childObj=" + childObj +
      "parentObj=" + parentObj);
  }
  parentObj[this.setMethod](childObj);
  this.jsDigester.push(parentObj);
  this.jsDigester.push(childObj);

} // End end().
```

So, after a bit of logging, we pop two objects off the stack: the first is the child object, and the second is the parent object. Think about this for a minute: the `SetNextRule` is used to set a property of the *next* object on the stack, passing it the object on the top of the stack. This is used to create a hierarchy of objects, where one object is the parent of another. The object on the top of the stack is always the child because of the structure of XML:

```
<movies>
  <movie>
    <actor></actor>
  </movie>
</movies>
```

Let's pretend this is all there is to the XML being parsed, and let's pretend we have configured only an `ObjectCreateRule` for each. At the point the `<actor>` element is encountered (but not yet closed), what does the object stack in JSDigester look like? They say a picture is worth a thousand words, so take a gander at Figure 7-5.

Figure 7-5. *The state of the JSDigester stack when the <actor> element begins*

Now, what if we have a `SetNextRule` configured for the path `movies/movie/actor` using the method `setActor()`? Well, the first object popped off the stack is the top object, the `Actor` object. That's the child. The next object popped off is again the top object, at this point, the `Movie` object. This is the parent. The `setActor()` method of the `Movie` object, since it is the parent, is then called, with this line:

```
parentObj[this.setMethod](childObj);
```

And *voilà*, we've created a parent-child hierarchical relationship. The `Movie` object now contains the `Actor` object, just as we wanted. There is only one last step to perform, and that is to push both of those objects back on the stack, because there still could be other rules yet to fire for them. But remember that to ensure the order of the stack is unchanged after this, we need to push them on in the opposite order they were popped off, so that the `Actor` object is on top of the `Movie` object, and you can see that done in the last two lines of the `end()` method.

See, wasn't that an interesting bit of code?

Well, that was fun, and it didn't take that long. As I've said a few times, JSDigester and its associated rules classes really aren't much to look at in terms of code length. In fact, you could go so far as to call them paltry. But simplicity that provides this much power is a beautiful thing, and it is a testament to the original creators of the Commons Digester that it makes parsing XML almost a fun experience. I hope you find JSDigester useful in your day-to-day work, as I have.

Suggested Exercises

Here are a few suggested exercises that you could pursue to make JSDigester that much more useful:

Implement more rules. Digester has a boatload of rules that could be implemented in JSDigester. Some may well not be possible, but others almost certainly are. Since JSDigester is written to be extensible, this would be a great exercise to become even more familiar with it.

Add validation capabilities. Digester can validate XML against a DTD, and this would be a very nice additional capability to give to JSDigester.

Allow for an alternate SAX parser. Provide a `setSAXParser()` function so that an alternate SAX parser could be used by JSDigester. As an added bonus, go write one yourself.

Optimize the code. I purposely left one bad design decision in the code specifically so I could make this suggestion. Did you notice that each of the rule classes has a different constructor signature? Surely, this can't be optimal, and, in fact, it isn't. I therefore suggest correcting that. There is more than one way you could do it, but I suggest passing in a single associative array as an argument, and use the values it contains to set the fields. However you choose to do it though, the goal is to normalize the method signature of the rule classes.

Improve the logging capabilities. This may actually wind up being a suggestion of something to add to the JavaScript library (introduced in Chapter 3), rather than JSDigester. It might be nice if you had a logger implementation that could log messages in a more effi-

cient manner so that having logging enabled isn't the performance-killer it is with `DivLogger`. The obvious problem is how to do that in such a way that logging will still always work (for example, if you try to cache all messages until some `flush()` method was called on the logger, you run the risk of losing all the messages if a hard error occurs). One suggestion, if you develop in Firefox frequently, is to log messages to the Firebug extension, which I would expect to work much more efficiently than constantly writing to a `<div>`'s `innerHTML` property.

Summary

In this chapter, we looked very briefly at how XML is typically parsed in JavaScript without the aid of any library or toolkit. We then built ourselves a piece of code modeled after the Jakarta Commons Digester project that should make our lives a lot easier when we need to parse XML in a browser. In the process, you became aware of the SAX parser created by the Mozilla project, which wound up as the basis for JSDigester.

CHAPTER 8

Get It Right, Bub: A JavaScript Validation Framework

When you hear the word *validation* in the context of a web application, you generally think of validating user input on a form. This usually evokes thoughts of writing event-handler code to perform various checks on form input before submitting it. All of this can quickly become rather messy. No matter how well you externalize your scripts and set up basic event-handler code that just calls functions, the simple fact is that these validations are scattered throughout your code—it's just a question of to what degree that's true.

Wouldn't it be great if we could truly externalize those validation checks to the point where we don't have to write any code at all? Wouldn't it be great if we could write a simple configuration file, say in XML, that defines what validations we want performed on each field and when? This is precisely the goal of the project in this chapter!

JSValidator Requirements and Goals

Building a JavaScript form validation framework isn't really that complex an undertaking, but making it truly useful requires a bit of forethought. To that end, let's lay out some of the goals we want to accomplish for our project, which we'll call JSValidator:

- The framework should be driven by an external configuration file written in XML. This will define what validations occur for which field, the parameters a given validation may need, and what to do if the validation fails.
- The framework should be extensible. We should be able to add validators—that is, classes that perform a given validation function—any time we want. We should be able to do this without modifying code, just the configuration file.
- We should create a couple of common validator types and cook them into the framework, so developers know they can always use these types without doing anything extra.
- This framework should be unobtrusive, meaning we shouldn't need to sprinkle code all over the place. Moreover, the form on which validations are defined should still work if JavaScript is turned off.

- Using the framework should be about as simple as can be for developers, requiring only a single JavaScript import, some minimal configuration parameters, and the external configuration file.
- The framework should offer error reporting in three ways: `alert()` messages, messages inserted into a specified `<div>`, and highlighting of fields (with developer-defined styles). Moreover, the messages should allow for reuse by understanding a token system, where the tokens can be replaced with values defined in the configuration file when a validation failure occurs.

If that sounds like a lot of work, I think you'll be surprised to see that it isn't quite as much as you might think. You'll also find that the end result is something useful and expandable that can be applied in your projects immediately. But I've left the door open for a couple of enhancements that will make it even more useful, and applying them will give you some valuable exercises to work through in order to expand your JavaScript chops.

With our goals and requirements set, let's see how we'll make it work!

How We Will Pull It Off

As mentioned in the requirements, we want JSValidator to use an external configuration file to define all the validations that will be performed for a given HTML form. More specifically, we want this configuration file to be in the form of XML.

There are always choices to be made when deciding on a format for a configuration file, but I feel that while XML isn't perfect for everything, for configuration files, it probably is. That is because it tends to be self-describing (unless the developer does a bad job laying it out). Also, size and parse time don't generally matter, because you tend to parse them once at startup and not frequently after that. You can take a slight performance hit at startup usually, which also means you don't care so much how large or verbose the file is. In fact, the more verbose, the better, most likely, as long as that verbosity adds to the clearness of the configuration.

So, having decided XML is the way to go for JSValidator, on to the next question: how are we going to deal with parsing it? If you've ever done XML parsing in JavaScript, you know that it isn't necessarily a pleasant experience. I discussed this a bit in Chapter 7, so no need to rehash the pain. In Chapter 7, we also find the answer to making it a bearable situation: JSDigester. With JSValidator, you'll see a real-world usage of JSDigester. You'll see how it allows you to take a somewhat complex XML document, describe it with a few simple rules, and have it parsed into JavaScript object quickly and easily. (If you skipped the details on JSDigester in Chapter 7, I suggest going back and reading that now.)

Now that you know how we're going to parse the configuration file, let's answer another important question: how are we going to load it? Remember one of the other goals for JSValidator is to make it as simple and unobtrusive for a developer to use as possible. As a matter of fact, here's all the developer has to do on any given page to use the framework:

```
<script>
  var JSVConfig = {
    pathPrefix : "jsvalidator/",
    configFile : "jsv_config.xml",
    manualInit : false
  };
```

```
</script>
<script src="jsvalidator/JSValidator.js"></script>
```

All that is required is a single JavaScript file to import, and some configuration parameters defined before that. Beyond that, the developer just has to write the configuration file. That's it! No mucking around with the form fields, no special attributes to attach to anything, and *no code to write*! Better still, if JavaScript is disabled, the page won't break—the form will still be workable (assuming it's not broken in some other way, of course), and the user won't know the difference. Of course, the validations won't occur, but that's as expected.

Speaking of that configuration file, how exactly does it get loaded? We can see in the preceding example that the file is named in the parameters, but that doesn't explain how it is loaded. The answer lies in a JavaScript library that serves as the foundation for a number of other libraries, including Rico (which we used in Chapter 4) and script.aculo.us (which we used in Chapter 5). In those chapters, I didn't go into detail about Prototype, but the time has come to do exactly that.

The Prototype Library

Prototype has gained a great deal of popularity because it is simple, lightweight, clean, and generally very helpful. It basically provides extensions to JavaScript that, once you use them, seem like they should have been there from the start. Prototype adds new methods to basic JavaScript objects, as well as provides new functions for things like Ajax, DOM manipulations, and looping constructs.

One of the simplest and yet most useful things Prototype offers is the `$()` function. As you've seen in previous chapters, this is essentially shorthand for writing `document.getElementById()`. The `$()` function allows for referencing a single object or a batch at once, like so:

```
$("id1", "id2", "id3");
```

This example will actually retrieve references to three elements and return an array of those references. That's much better than three separate calls to `document.getElementById()`!

Prototype also provides some simple Ajax support. When I say "simple," I don't mean that as a negative at all. On the contrary, it is very easy to understand and use, which to me is a good thing. The first item Prototype offers in the realm of Ajax is this:

```
Ajax.Request(url, { method: 'get', parameters: pars, onComplete: showResponse });
```

Yes, that's all it takes to make an Ajax request with Prototype! Just supply the URL, set the method to use, pass in any parameters you want to send with the request, and tell Prototype which JavaScript function to call when the response returns.

But wait, there's more!

Probably the most common Ajax function is inserting some markup returned by the server into a `<div>` or other element. Prototype makes this very simple:

```
new Ajax.Updater( "targetID", url, { method: 'get', parameters: pars });
```

This looks very much like the previous line of code, except that now we're passing in the ID of the element to update, and leaving off the callback, because Prototype handles the callback functionality for us.

As I mentioned earlier, Prototype also extends some basic JavaScript objects. For example, Prototype adds some of the methods to the objects shown in Table 8-1.

Table 8-1. *Some of the Methods Prototype Adds to Intrinsic JavaScript Objects*

Method	Object	Description
`extend`	`Object`	Provides a way to implement inheritance by copying all properties and methods from source to destination.
`bind`	`Function`	Returns an instance of the function previously bound to the function(=method) owner object. The returned function will have the same arguments as the original one.
`stripTags`	`String`	Returns the string with any HTML or XML tags removed.
`escapeHTML`	`String`	Returns the string with any HTML markup characters properly escaped.
`toArray`	`String`	Splits the string into an array of its characters.
`clear`	`Array`	Empties the array and returns itself.
`flatten`	`Array`	Returns a flat, one-dimensional version of the array. This flattening happens by finding each of the array's elements that are also arrays and including their elements in the returned array, recursively.
`getElementsByClassName`	`document`	Returns all the elements that are associated with the given CSS class name. If no parent element ID is given, the entire document body will be searched.
`element`	`Event`	Returns the element that originated the event.
`pointerX/pointerY`	`Event`	Returns the x/y coordinate of the mouse pointer on the page.

■ **Note** As I mentioned in Chapter 2, some people view Prototype's extension of intrinsic JavaScript objects as a bad thing. There have been instances where these extensions didn't "play nice" with other JavaScript code. I have personally never been burned by these types of problems, but some people have, so this is something to keep in mind when using Prototype (and by extension, any library that is built on top of Prototype). While my feeling is that it shouldn't dissuade you from using Prototype, or other libraries built on it, you should be aware of the issue.

As you can see, Prototype makes life a little simpler, and I've just scratched the surface here! I suggest taking a longer look at Prototype yourself. Also, in terms of documentation, have a look at the "unofficial" guide to Prototype at `http://www.sergiopereira.com/articles/prototype.js.html`.

Why do I feel as though I've forgotten something? Oh yes, how about a look at JSValidator?

A Preview of JSValidator

Because JSValidator is really a nonvisual tool, there isn't going to be a whole lot to look at, but that didn't stop me from adding some screenshots here. What you're actually looking at is the demonstration application that shows JSValidator in use.

Figure 8-1 shows the demo application when you first start it up. It describes what validations have been applied to the five form fields.

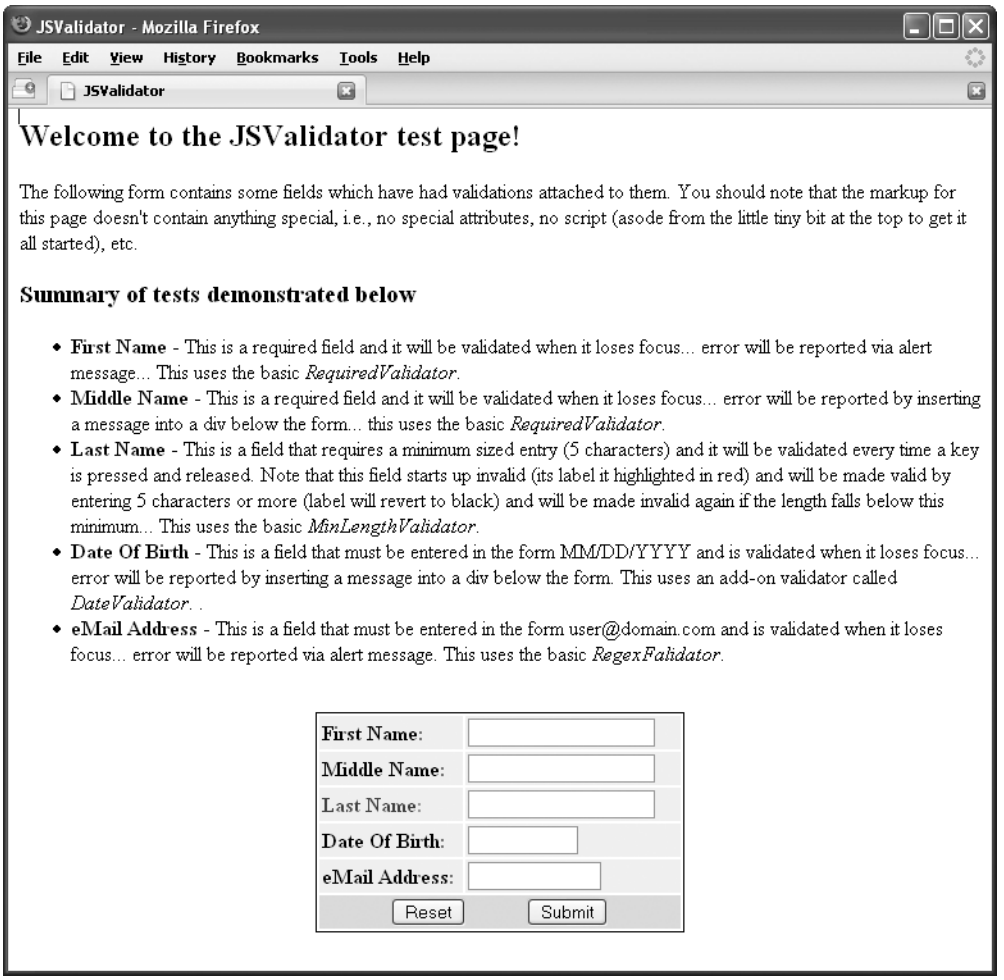


Figure 8-1. *The JSValidator framework test application*

Figure 8-2 shows one way a validation failure can be presented to the user: an alert message. Here, when users tab or click away from the First Name field without having entered something (it is configured to be a required field), they get an `alert()` message. Note that the message is configurable and reusable, including allowing for tokens in it to be replaced.

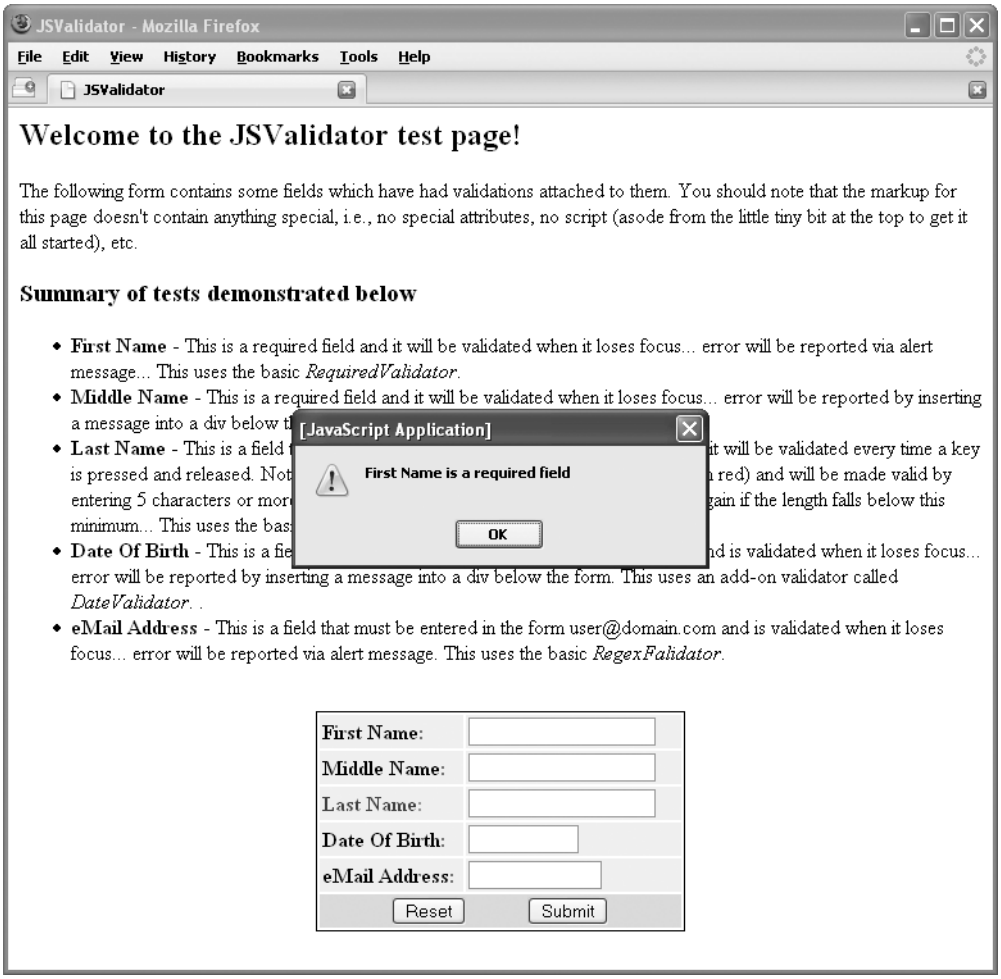


Figure 8-2. *Showing a validation error via an alert message*

Figure 8-3 demonstrates another way to show a validation error. The Last Name field is configured to be highlighted when an error occurs. This field is highlighted in red (which may be nearly impossible to see in a black-and-white version, but it's obvious when you try the application), indicating that it is marked as invalid. In fact, you can configure any element on the page to have a particular style applied to it. If you prefer to change the background color of the text box itself to red, you can do that. Or if you want to make a big "ERROR" appear over the page, you could do that, too!

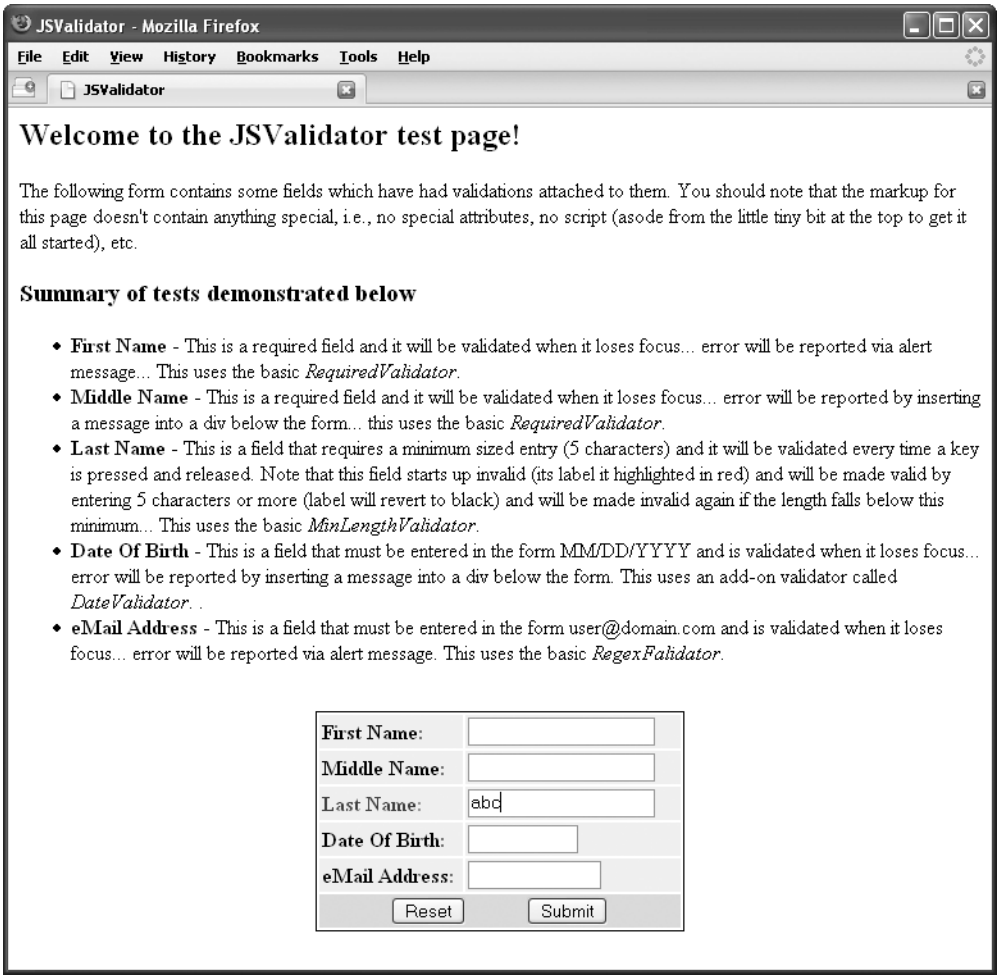


Figure 8-3. *Showing a validation error via field highlighting*

Figure 8-4 shows a third way a validation error can be reported: by inserting a message into an element on the page, usually a `<div>`. The message is constructed the same way as the `alert()` display, but in this case, the `innerHTML` attribute of some page element is updated with the message.

Clicking the Submit button brings up a new page, which is nothing more than a message to indicate the submission was OK. We aren't actually submitting to a server obviously, so this page is something of a *faux* server. I didn't see much sense in showing that page here, but you'll see it when you try out the application, so I wanted to be sure to mention it.

Now that the formalities are out of the way, we can move on to examining this solution!

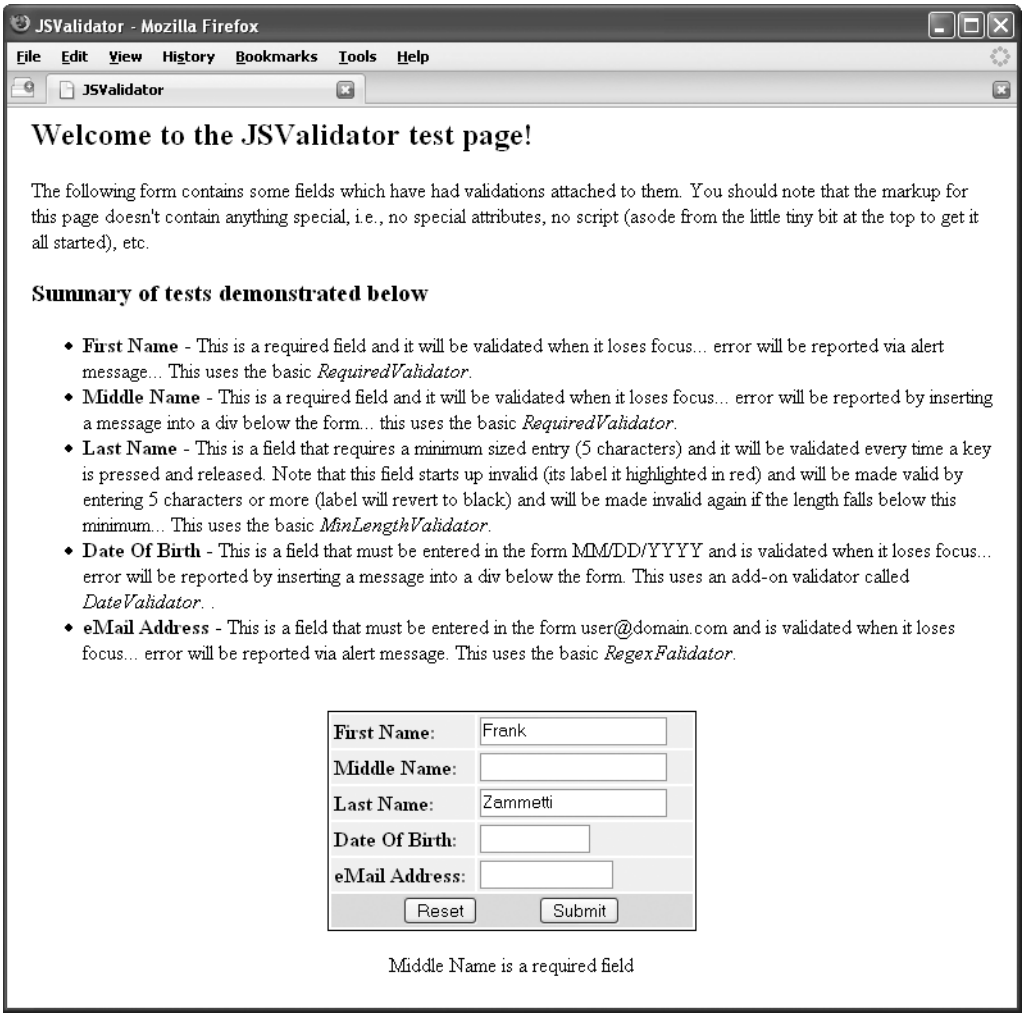


Figure 8-4. *Showing a validation error via a message inserted into a element*

Dissecting the JSValidator Solution

What we are about to dissect is a combination of the JSValidator framework and a test application for it. To begin, let's see how this all lays out on our hard drives, as shown in Figure 8-5.

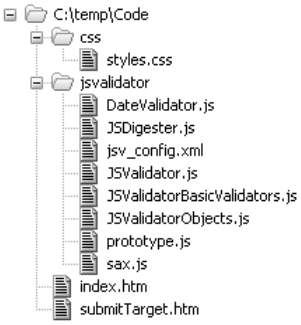


Figure 8-5. *The directory structure of the JSValidator test application*

Writing index.htm

`index.htm` is easily the most boring of files to look at in this project because it is, quite literally, nothing but run-of-the-mill HTML. It's just some text, a very vanilla HTML form, a style sheet import, and a sprinkle of JavaScript. In light of this, it won't be listed here, and we won't be going over it in any real detail. However, you should probably take the five seconds you'll need to fully review it.

The JavaScript is the only thing worth mentioning, and you've actually seen it already. But let's look at it one more time and go into a little more detail this time:

```
<script>
  var JSVConfig = {
    pathPrefix : "jsvalidator/",
    configFile : "jsv_config.xml",
    manualInit : false
  };
</script>
<script src="jsvalidator/JSValidator.js"></script>
```

First, you see the definition of the `JSVConfig` structure. This is just an associative array that has three elements in it, which happen to be the only ones JSValidator currently understands.

The `pathPrefix` member specifies the beginning of all the paths that JSValidator will need to construct to load validators and rule files. Here, we are saying that all those dynamically created paths will begin with `jsvalidator/`, which makes sense if you look back at the directory structure (Figure 8-5). Any reference to any JSValidator resource should be relative to the root of the web application, hence this value.

The second member, `configFile`, should be obvious. It's the name of the XML configuration file JSValidator will use.

Finally, `manualInit` tells JSValidator whether it should initialize itself automatically or the developer is taking responsibility for doing so. As you will see, initializing JSValidator amounts to nothing more than a call to `jsValidator.init()`. Normally, this happens automatically when JSValidator loads. However, in order for that to work, JSValidator needs to overwrite the `onLoad` event handler for the page. Since this is often utilized by developers for their own purposes, simply overwriting the existing handler wouldn't be a great idea. Setting `manualInit` to `true`

means that the developers will make the call to `init()` themselves, and the `onLoad` handler will *not* be overwritten.

Writing styles.css

`styles.css`, like `index.htm`, is rather mundane. However, let's have a look at it in its entirety, in Listing 8-1.

Listing 8-1. *The styles.css File (Blink and You'll Miss It!)*

```
/* Style for labels on fields that have a validation error. */
.cssErrorField {
  color            : #ff0000;
  background-color : #d0ffd0;
  font-weight      : bold;
}

/* Style for labels on fields that pass validations. */
.cssOKField {
  color            : #000000;
  background-color : #d0ffd0;
  font-weight      : bold;
}

/* Background for the table cells with the entry fields. */
.cssEntryCell {
  background-color : #f0f0f0;
}

/* Style for the table row with the reset and submit buttons. */
.cssButtons {
  background-color : #ffd0d0;
}
```

As you saw in Figure 8-3, one of the ways JSValidator can report a validation error is to highlight a given field. It can highlight any element on the page. Actually, it doesn't have to "highlight" anything!

All that really happens is that when a validation error occurs, the element with a given ID has its class attribute changed to the style class configured. So really, you could do any wacky thing you wanted with this. Want to show an image that is in a hidden `<div>`? No problem. Want to make some text blink for anyone using old Netscape browsers? Check. Anything you can accomplish by applying a style class, you can do with the "highlighting" capability. All of this is a roundabout way of explaining that the `cssErrorField` class seen here is the class that will be applied to highlight a field's label in red in the sample application.

Going hand in hand with `cssErrorField` is the `cssOKField` class, which will be applied to the label for a field that passes its validations. This means that JSValidator will take care of setting the style to indicate no error in addition to indicting an error. For example, with the Last Name field validation, which checks for a minimum length every time a key is pressed and

released, you get the automatic valid/invalid highlighting as you type. There is no extra work for you to do—just point the framework at those two classes, and it's handled for you automatically.

The `cssEntryCell` class just makes the background behind the entry boxes a light gray. Finally, `cssButton` is applied to the row where the Reset and Submit buttons are located.

Writing jsv_config.xml

`jsv_config.xml` is what actually drives this whole thing, so we definitely want to understand it. Figure 8-6 shows a graphical representation of it, which may help to make all the relationships crystal clear in your mind.

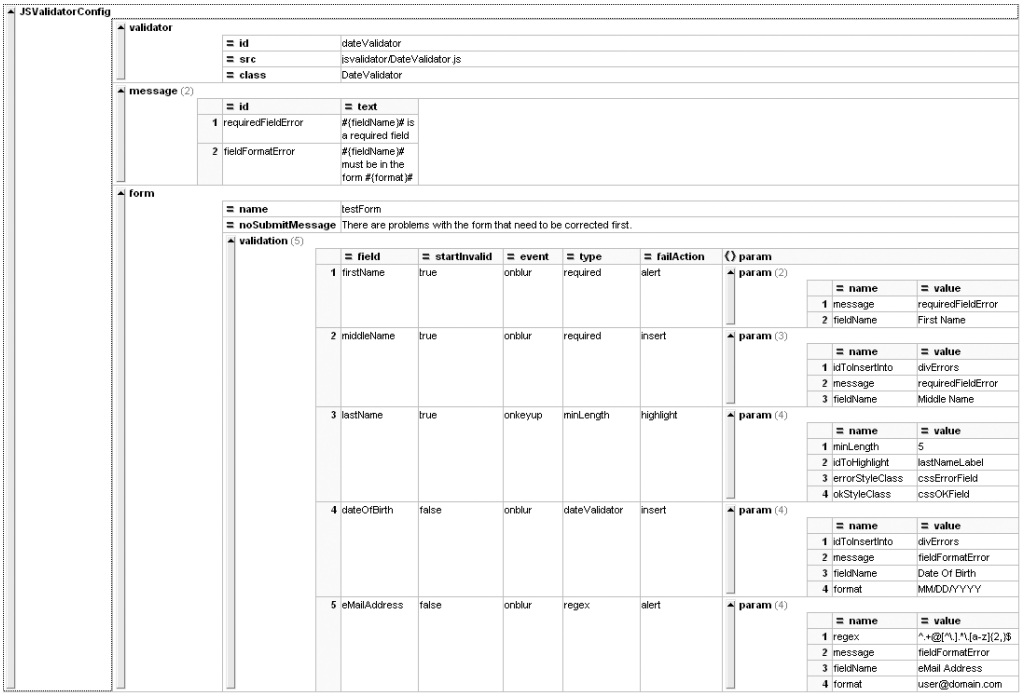


Figure 8-6. *A visual grid representation of the jsv_config.xml file*

And if Figure 8-6 didn't quite do it for you, Listing 8-2 shows the configuration file in all its textual glory!

Listing 8-2. *The JSValidator Configuration File in Plain Text Form*

```
<JSValidatorConfig>

  <validator id="dateValidator" src="jsvalidator/DateValidator.js" ➥
class="DateValidator"/>
```

```
  <message id="requiredFieldError" text="#{fieldName}# is a required field"/>
  <message id="fieldFormatError"  ➥
text="#{fieldName}# must be in the form #{format}#"/>

  <form name="testForm"  ➥
noSubmitMessage="There are problems with the form that need to be corrected first.">
    <validation field="firstName" startInvalid="true" event="onblur"  ➥
type="required" failAction="alert">
      <param name="message" value="requiredFieldError"/>
      <param name="fieldName" value="First Name"/>
    </validation>
    <validation field="middleName" startInvalid="true" event="onblur"  ➥
type="required" failAction="insert">
      <param name="idToInsertInto" value="divErrors"/>
      <param name="message" value="requiredFieldError"/>
      <param name="fieldName" value="Middle Name"/>
    </validation>
    <validation field="lastName" startInvalid="true" event="onkeyup"  ➥
type="minLength" failAction="highlight">
      <param name="minLength" value="5"/>
      <param name="idToHighlight" value="lastNameLabel"/>
      <param name="errorStyleClass" value="cssErrorField"/>
      <param name="okStyleClass" value="cssOKField"/>
    </validation>
    <validation field="dateOfBirth" startInvalid="false" event="onblur"  ➥
type="dateValidator" failAction="insert">
      <param name="idToInsertInto" value="divErrors"/>
      <param name="message" value="fieldFormatError"/>
      <param name="fieldName" value="Date Of Birth"/>
      <param name="format" value="MM/DD/YYYY"/>
    </validation>
    <validation field="eMailAddress" startInvalid="false" event="onblur"  ➥
type="regex" failAction="alert">
      <param name="regex" value="^.+@[^\.].*\.[a-z]{2,}$"/>
      <param name="message" value="fieldFormatError"/>
      <param name="fieldName" value="eMail Address"/>
      <param name="format" value="user@domain.com"/>
    </validation>
  </form>

</JSValidatorConfig>
```

Let's dissect this file in detail and go over each element and attribute. First, everything is nested under the root `JSValidatorConfig` element, to make it valid XML.

Defining Validators

After the root element are any number of `validator` elements (and there could be none). These define any validator (class that performs some validation logic) that you add. There are a couple of built-in validators, as you'll see shortly, but any others you add will need to be defined with a `validator` element. The `validator` element has three required attributes:

- `id`: How you will reference the validator for a given field validation.
- `src`: The JavaScript source file where the class that implements the validator is found. It can be absolute or relative—basically any value that would be valid for a `<script>` tag's `src` attribute.
- `class`: The name of the JavaScript class that implements the validator logic.

Defining Messages

Direct children of `JSValidatorConfig` are the `message` elements. Like the `validator` elements, these are completely optional and you can define as many as you wish. Even though they are technically optional, you will likely always have at least one, because any validator that displays a message in response to a validation failure will reference one of these elements.

The `message` elements have two required attributes: `id`, which serves the same purpose as on the `validator` element, and `text`, which is the text of the message. This text can contain any number of replacement tokens. These tokens are in the form `#{`*xxx*`}#`, where *xxx* is any string. These tokens will be replaced by the values present for a given field validation. So, if you had the text "`#{fieldName}# is a required field`," as seen in the first `message` element in Listing 8-2, assuming a particular field validation defines a `fieldName` parameter, the value of that parameter will be inserted for `#{fieldname}#` when the message is shown.

Defining Forms

Next up is the `form` element. This corresponds to a single physical HTML form on the page. It has two attributes: `name` and `noSubmitMessage`. The `name` attribute needs to match exactly the `name` attribute of the `<form>` tag on the page to which the validations apply. The `noSubmitMessage` attribute is the message that will be displayed to users when they try to submit the form and any of the fields are currently invalid.

Defining Field Validations

The `form` element has as its children any number of `validation` elements, and this is where the real action occurs. The `validation` element has five required attributes:

- `field`: The name of the element on the form to which this validation applies. It must match exactly the `name` attribute of the `form` element.
- `startInvalid`: The value `true` or `false`. When set to `true`, the field will initially be marked as invalid, and if it is configured to report its error via highlighting, at startup it will be highlighted. This is useful for something like the Last Name field, which is configured to require a minimum length, and which starts out empty, so is clearly not going to meet the minimum length initially!

- `event`: The event on which the field will be validated. It can be any of the usual DOM events, but `onblur` will likely be the most usual.
- `type`: The ID of a validator, either one you configure yourself or one of the built-in basic validators.
- `failAction`: The action that will be performed when the field fails validation. This can be one of three values: `alert`, which means a message will be shown to the user via an `alert()` pop-up; `insert`, which means a message will be inserted into a specified page element via its `innerHTML` property; or `highlight`, which means a specified field will be highlighted.

Defining Validation Parameters

Nested beneath a `validation` element are the `param` elements. Each `param` element has two attributes: `name`, which is simply the name under which this parameter will be stored, and `value`, which is the actual value of the parameter.

The `param` elements are a flexible mechanism by which information can be passed to the validator configured for a given element, and to JSValidator to use during a failure action. Because of this flexibility, the parameters can really have any meaning you wish, and each validator and failure action can require anything they need.

For example, the built-in `MinLengthValidator`, which validates that a field has a specified minimum length, gets what the minimum length for the field is by looking for a parameter with the name `minLength`. Similarly, the `RegexValidator`, which allows a field to be validated against an arbitrary regular expression, gets that expression via a parameter named `regex`.

For both the `alert` and `insert` failure actions, the ID of the message element to use is found via the `message` parameter. If there are any replacement tokens in the string—`fieldName` for instance, as seen in the messages in Listing 8-2—they are also found in the parameter list here. For the `insert` failure action, the ID of the element to highlight is in a parameter named `idToHighlight`, and the style classes for valid and invalid entries are found in `errorStyleClass` and `okStyleClass`, respectively.

As mentioned, when you write your own validators, you can require any parameters you need—there are no real limitations. However, one thing to note is that replacement tokens are used only to replace tokens in message strings during processing of the `alert` or `insert` failure actions. You can, however, use the same `replaceTokens()` function in JSValidator that these use, as you'll see shortly. For example, if you wanted a parameter that could have a value inserted into it dynamically, you could do so using the same token mechanism as the messages.

Writing JSValidatorObjects.js

The `JSValidatorObjects.js` file contains the class definitions for seven different classes. All but one are objects that are populated when the configuration file is parsed. These seven classes do not provide any functionality per se. They are basically just Value Objects (VOs) that store some data and provide a public interface for getting at that data. As such, I'm sure you will find them to be quick and easy to understand. Nonetheless, you need to know their purposes, so we'll look at each in turn, and then at the end see the big picture of how they all fit together.

The JSValidatorValidatorImpl Class

To begin with, take a look at Figure 8-7, which shows the UML diagram for the only one of the seven classes that doesn't represent configuration information. The `JSValidatorValidatorImpl` class is the base class for all validator implementation classes. A validator will generally need to inherit only from this class and override the `validate()` method to become a validator that the framework can use.

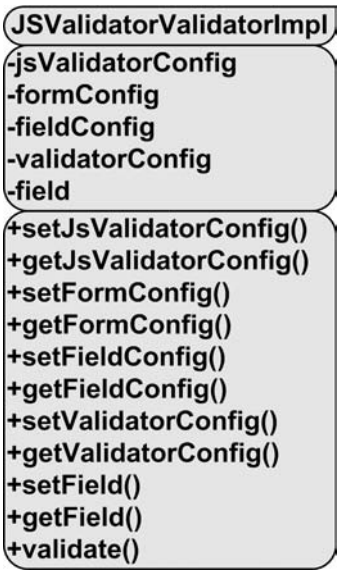


Figure 8-7. *UML diagram of the JSValidatorValidatorImpl class*

The five fields in this class are populated by the framework (using the corresponding setter methods) before the validator is asked to validate the field:

- The `jsValidatorConfig` field holds a reference to the `JSValidatorConfig` instance that is the parent of all the configuration data parsed from the configuration file.
- The `formConfig` field holds a reference to the `JSValidatorForm` object that represents the `<form>` element in the configuration file of which the element firing the validation is a part.
- The `fieldConfig` field holds a reference to a `JSValidatorFormValidation` object, which is the JavaScript representation of a `<validation>` element from the configuration file. This object describes the validation to be performed on a given field.
- The `validatorConfig` field holds a reference to a `JSValidatorValidatorConfig` object, which is the JavaScript representation of a `<validator>` element from the configuration file. This object describes the validation that will be used to validate a given field.
- Finally, the `field` field, if you'll excuse the alliteration, is a reference to the form field itself that fired the validation.

As mentioned, a validator will typically just supply its own implementation of `validate()` and call it a day. This method returns `true` if the field passes validation; `false` if not. Of course, there is nothing that says there can't be other methods and fields as necessary, but strictly speaking, this is all that's required.[1]

Just to prove that I'm not imagining all this, here is the code for the `JSValidatorValidatorImpl` class. As you can see, it really is empty, and yet is itself a complete implementation of a validator, albeit a rather pointless one!

```
function JSValidatorValidatorImpl() {

  this.jsValidatorConfig = null;
  this.formConfig = null;
  this.fieldConfig = null;
  this.validatorConfig = null;
  this.field = null;

  this.setJsValidatorConfig = function(inJsValidatorConfig) {
    this.jsValidatorConfig = inJsValidatorConfig;
  }
  this.getJsValidatorConfig = function() {
    return this.jsValidatorConfig;
  }

  this.setFormConfig = function(inFormConfig) {
    this.formConfig = inFormConfig;
  }
  this.getFormConfig = function() {
    return this.formConfig;
  }

  this.setFieldConfig = function(inFieldConfig) {
    this.fieldConfig = inFieldConfig;
  }
  this.getFieldConfig = function() {
    return this.fieldConfig;
  }

  this.setValidatorConfig = function(inValidatorConfig) {
    this.validatorConfig = inValidatorConfig;
  }
  this.getValidatorConfig = function() {
    return this.validatorConfig;
  }
```

1. In fact, there are empty implementations of all methods in the `JSValidatorValidatorImpl` class. So even if you wrote an empty class that extends `JSValidatorValidatorImpl`, it wouldn't break JSValidator—it just wouldn't do much of anything.

```
  this.setField = function(inField) {
    this.field = inField;
  }
  this.getField = function() {
    return this.field;
  }

  this.validate = function() { }

} // End JSValidatorValidatorImpl class.
```

The JSValidatorConfig Class

The `JSValidatorConfig` class, shown in Figure 8-8, is the top of the object entity relationship hierarchy in which the configuration information parsed from the configuration file is found. The three fields represent the three elements that can be immediate children of the `<JSValidatorConfig>` element in the configuration file: the `<validator>`, `<message>`, and `<form>` elements. Each is a collection of all the corresponding elements parsed from the configuration file.

Figure 8-8. *UML diagram of the JSValidatorConfig class*

Note that the `toString()` method has been overridden in order to display a more meaningful representation of this class for debugging purposes. This is true of all the remaining configuration file objects, so I won't mention it again.

Let's have a look at the actual code now, shall we?

```
function JSValidatorConfig() {

  var validators = new Object();
  var messages = new Object();
  var forms = new Object();
```

```
  this.addValidator = function(inValidatorConfig) {
    validators[inValidatorConfig.getId()] = inValidatorConfig;
  }
  this.getValidators = function() {
    return validators;
  }
  this.getValidator = function(inID) {
    return validators[inID];
  }

  this.addMessage = function(inMessage) {
    messages[inMessage.getId()] = inMessage;
  }
  this.getMessages = function() {
    return messages;
  }
  this.getMessage = function(inID) {
    return messages[inID];
  }

  this.addForm = function(inForm) {
    forms[inForm.getName()] = inForm;
  }
  this.getForms = function() {
    return forms;
  }
  this.getForm = function(inName) {
    return forms[inName];
  }

  this.toString = function() {
    return "JSValidatorConfig=[" +
    "validators=" + validators + "," +
    "messages=" + messages + "," +
    "forms=" + forms + "]";
  }

} // End JSValidatorConfig class.
```

Aside from the three fields and `toString()`, all this class contains are getters and setters for each field, as well as an `add`*`XXX`*`()` method for each of the collections. These methods are used by JSDigester to add `JSValidatorForm`, `JSValidatorMessage`, and `JSValidatorValidatorConfig` instances to the corresponding collection. Note that in addition to getters to get any one of the collections, there is a getter for each to get a specific element from the collection by ID or name, depending on which element we're going after (ID for validators and messages; name for forms).

The JSValidatorValidatorConfig Class

Figure 8-9 shows the `JSValidatorValidatorConfig` class, which is implemented with the following code:

```
function JSValidatorValidatorConfig() {

  var id = null;
  var src = null;
  var clazz = null;

  this.getId = function() {
    return id;
  }
  this.setId = function(inID) {
    id = inID;
  }

  this.getSrc = function() {
    return src;
  }
  this.setSrc = function(inSRC) {
    src = inSRC;
  }

  this.getClass = function() {
    return clazz;
  }
  this.setClass = function(inClass) {
    clazz = inClass;
  }

  this.toString = function() {
    return "JSValidatorValidatorConfig=[" +
    "id=" + id + "," +
    "src=" + src + "," +
    "clazz=" + clazz + "]";
  }

} // End JSValidatorValidatorConfig class.
```

An instance of this class will be created and populated for each `<validator>` element encountered in the configuration file. This information is necessary for JSValidator to be able to work with a validator you define. The meaning of the fields should be obvious at this point, based on our dissection of the configuration file.

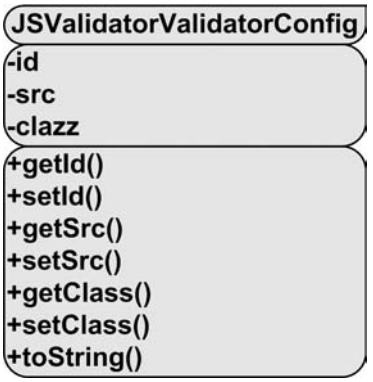


Figure 8-9. *UML diagram of the JSValidatorValidatorConfig class*

You probably noticed the use of `clazz` as opposed to `class`. IE doesn't take too kindly to using `class`, which seems to be a reserved word to it. (Surprisingly, Firefox does not seem to have a problem with `class`!) This actually mimics Java, as you may know, where `class` is a reserved word. It is typical in Java code, especially Java code dealing with reflection, to use `clazz` in place of `class` to get around this, and that's the case here as well. So, while it's not exactly a big deal, I wanted to make you aware that I didn't just make a silly typo—there's a reason for it!

The JSValidatorMessage Class

Continuing our Napoleonic march through these configuration classes, we next encounter the class shown in Figure 8-10: `JSValidatorMessage`. This class represents the `<message>` elements from the configuration file. Once again, I think the fields are pretty obvious, as they simply echo the attributes of the `<message>` element, and the methods are nothing but accessors and mutators for said fields. But, in the interest of completeness, let's have a look at the code for this class now, and then move on.

```
function JSValidatorMessage() {

  var id = null;
  var text = null;

  this.getId = function() {
    return id;
  }
  this.setId = function(inID) {
    id = inID;
  }

  this.getText = function() {
    return text;
  }
```

```
  this.setText = function(inText) {
    text = inText;
  }

  this.toString = function() {
    return "JSValidatorMessage=[" +
    "id=" + id + "," +
    "text=" + text + "]";
  }

} // End JSValidatorMessage class.
```

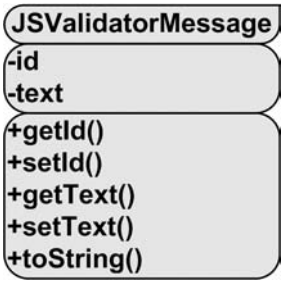


Figure 8-10. *UML diagram of the JSValidatorMessage class*

The JSValidatorForm Class

In Figure 8-11, we meet up with the `JSValidatorForm` class. Here's the code that matches up with that UML diagram:

```
function JSValidatorForm() {

  var name = null;
  var noSubmitMessage = null;
  var validations = new Object;

  this.getName = function() {
    return name;
  }
  this.setName = function(inName) {
    name = inName;
  }

  this.getNoSubmitMessage = function() {
    return noSubmitMessage;
  }
  this.setNoSubmitMessage = function(inNoSubmitMessage) {
    noSubmitMessage = inNoSubmitMessage;
  }
```

```
  this.addValidation = function(inValidation) {
    validations[inValidation.getField()] = inValidation;
  }
  this.getValidations = function() {
    return validations;
  }
  this.getValidation = function(inField) {
    return validations[inField];
  }

  this.toString = function() {
    return "JSValidatorForm=[" +
    "name=" + name + ", " +
    "validations=" + validations + "]";
  }

} // End JSValidatorForm class.
```

Figure 8-11. *UML diagram of the JSValidatorForm class*

Yes, again, this class simply matches up and represents the `<form>` element in the configuration file, and yes, what the fields are should probably be pretty obvious from the earlier discussion. However, we do have a couple interesting things to talk about.

One interesting point is the fact that the `validations` field is not an array. In fact, if you go back and look at the `JSValidatorConfig` class, you'll notice that the three collections there are not arrays either. Why is this interesting? Well, because a "collection" in most languages, assuming you aren't using some additional library, generally means an array to most people. So why would I use an `Object` here instead?

You may have heard that objects in JavaScript are essentially associative arrays? "Aha!" I can hear you say, "that's the answer right there!" I *am*, in fact, using an array; it's just not an outright `Array` object as you may have expected. Using an object allows you to reference the elements of the array by name, which happens to be exactly what we need throughout `JSValidator`. For instance, a `<form>` element can have any number of `<validation>` elements nested under it.

We want to be able to pull them up by name, and making them members of an `Object` allows for that. We can still iterate over them, as you'll see when we examine the code in `JSValidator.js`, but pulling up by name is the primary reason.

The other point to bring to your attention is that with only a few exceptions, all the fields in all these classes are private. That's the reason there are getters and setters for them (if they were public, those methods would be redundant). This is just good, everyday object-oriented design, but it's a somewhat unknown concept to a great many JavaScript programmers (and if I'm being honest, there was a time that I was doing object-oriented JavaScript and didn't know it either!).

The JSValidatorFormValidation Class

With the UML diagram in Figure 8-12, we come to the single most important class in terms of the information it provides. `JSValidatorFormValidation` is the class from which we create objects that describe a validation to occur for a given field and what happens when a validation failure occurs. The code for the `JSValidatorFormValidation` class is as follows:

```
function JSValidatorFormValidation() {

  var field = null;
  var event = null;
  var type = null;
  var failAction = null;
  var startInvalid = null;
  var params = new Object();

  this.getField = function() {
    return field;
  }
  this.setField = function(inField) {
    field = inField;
  }

  this.getEvent = function() {
    return event;
  }
  this.setEvent = function(inEvent) {
    event = inEvent;
  }

  this.getType = function() {
    return type;
  }
  this.setType = function(inType) {
    type = inType;
  }
```

```
  this.getFailAction = function() {
    return failAction;
  }
  this.setFailAction = function(inFailAction) {
    failAction = inFailAction;
  }

  this.getStartInvalid = function() {
    return startInvalid;
  }
  this.setStartInvalid = function(inStartInvalid) {
    startInvalid = inStartInvalid;
  }

  this.addParam = function(inParam) {
    params[inParam.getName()] = inParam;
  }
  this.getParams = function() {
    return params;
  }
  this.getParam = function(inName) {
    return params[inName];
  }

  this.toString = function() {
    return "JSValidatorFormValidation=[" +
    "field=" + field + "," +
    "event=" + event + "," +
    "type=" + type + "," +
    "failAction=" + failAction + "," +
    "startInvalid=" + startInvalid + "," +
    "params=" + params + "]";
  }

} // End JSValidatorFormValidation class.
```

Once again, the `JSValidatorForm` is nothing but a unit of storage for the corresponding configuration information. Any number of `JSValidatorForm` objects can be nested within a `JSValidatorForm` object, and by extension, the next class, `JSValidatorFormValidationParam`, can be nested within a `JSValidatorForm` object via the `params` field.

Figure 8-12. *UML diagram of the JSValidatorFormValidation class*

The JSValidatorFormValidationParam Class

Figure 8-13 shows the UML diagram for the `JSValidatorFormValidationParam` class. As you would expect, the code for this class is simple:

```
function JSValidatorFormValidationParam() {

  var name = null;
  var value = null;

  this.getName = function() {
    return name;
  }
  this.setName = function(inName) {
    name = inName;
  }

  this.getValue = function() {
    return value;
  }
```

```
  this.setValue = function(inValue) {
    value = inValue;
  }

  this.toString = function() {
    return "JSValidatorFormValidation=[" +
    "name=" + name + "," +
    "value=" + value + "]";
  }

} // End JSValidatorFormValidationParam class.
```

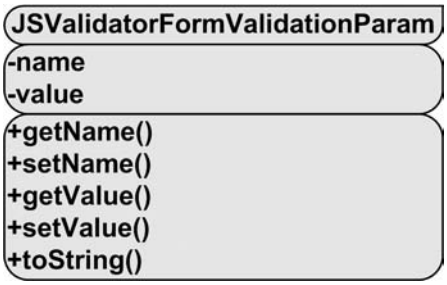


Figure 8-13. *UML diagram of the JSValidatorFormValidationParam class*

How the Classes Fit Together

Now that you've seen the UML diagram and code for each of the `JSValidatorObjects.js` classes, and also how they relate to the elements in the XML configuration file, the next step is to understand how they all fit together. I've more or less described the relationships textually as we walked through these classes, and the relationships mimic what is seen in the configuration file, but I would like to show you a graphical representation as well. An entity relationship diagram should do the trick nicely, and that's what's shown in Figure 8-14.

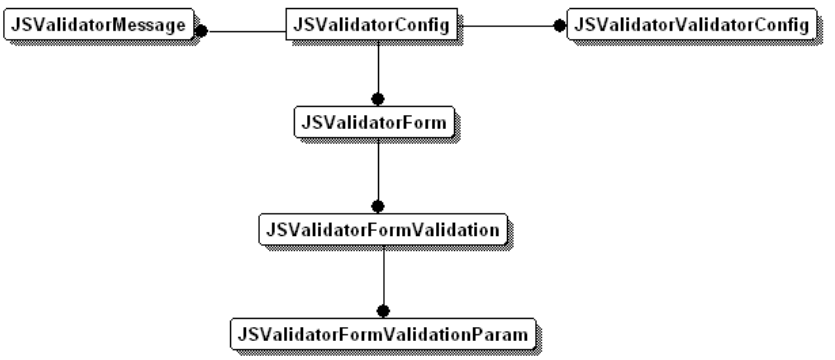


Figure 8-14. *Entity relationship hierarchy diagram showing how all the JSValidatorObject classes fit together*

Writing JSValidator.js

Now we come to the portion of our show where we examine what can rightly be called the JSValidator framework itself, and coincidence of coincidences, it's found in the file `JSValidator.js`!

Before we review the code itself, let's take a look at the UML diagram for this class. I think you'll be surprised when you look at Figure 8-15 just how little there really is to it.

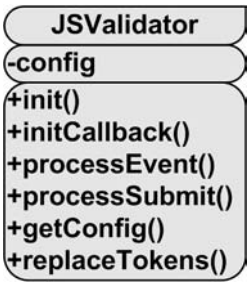


Figure 8-15. *UML diagram of the JSValidator class*

Because there is more code to this class than all the others we have looked at (by a good margin), I won't be listing the whole thing, but we'll be looking at the pieces we need to as we go along.

The first thing we see is a single field defined: `config`. The `config` field is a reference to a single `JSValidatorConfig` object, which as we saw earlier, is the object that holds all the configuration information parsed from the configuration file.

Let's skip ahead a bit, over all the methods in this class, and instead look at the bit of code that we find in this file *after* the definition of JSValidator:

```
// Include dependencies.
document.write('<script src="' + JSVConfig.pathPrefix +
  'prototype.js"></script>');
document.write('<script src="' + JSVConfig.pathPrefix +
  'sax.js"></script>');
document.write('<script src="' + JSVConfig.pathPrefix +
  'JSDigester.js"></script>');
document.write('<script src="' + JSVConfig.pathPrefix +
  'JSValidatorObjects.js"></script>');
document.write('<script src="' + JSVConfig.pathPrefix +
  'JSValidatorBasicValidators.js"></script>');

// Instantiate JSValidator.
jsValidator = new JSValidator();

// Set onload event to configure JSValidator, unless told not to.
if (!JSVConfig.manualInit) {
  window.onload = function() {
    jsValidator.init();
  };
}
```

The five `document.write()` calls can be thought of as imports of the elements required for JSValidator to function. Notice that they use the `pathPrefix` element defined in `JSVConfig` to construct the URL to the imported script file. Here are the imports:

- We need the Prototype library, of course, so we see that first.
- We know that JSDigester is built on top of the SAX parser from the Mozilla project, so that import is next.
- Next follows JSDigester itself.
- Next comes the import of the `JSValidatorObjects.js` file, which contains all of the configuration classes that we looked at a short while ago.
- Last is an import of the basic, built-in validators that JSValidator provides, which we will be looking at after JSValidator itself.

These five statements are executed at page load, and so writing `<script>` tags into the page dynamically like this will cause the browser to download those resources as well. This is how it is possible for a developer to import only a single `.js` file in order to use JSValidator. The rest are includes in `JSValidator.js` itself and are therefore "automatic," as far as the developer is concerned.

JavaScript files importing others like this is a very handy technique that allows you to keep code separate but still easily get everything you need. The alternative is to merge all these JavaScript files into one, which might be better from the perspective of not having to make extra calls to the server. But the concept of keeping all these components in separate files, and therefore allowing them to easily be updated separately, is a powerful one, and frankly one taken for granted in more full-featured languages. So, I am of the opinion that unless you know you have an issue due to these extra requests, keeping things clean and separated is more useful.

After those five statements is the one statement that gives us the instance of the `JSValidator` class that we'll be using throughout the rest of the code. Immediately following that is where JSValidator is initialized automatically, if configured to do so. Remember that the developer can set the `manualInit` element in `JSVConfig` to `false` and take control of initializing the framework. This is done by setting the `onLoad` event handler for the page to point to the `init()` method of the `JSValidator` class. Note that this is necessary, as opposed to simply calling `init()` right there, because the page has to actually be fully loaded; otherwise, the code that executes and accesses the DOM can fail because the script can execute before the page fully loads.

The init() Method

And what does this `init()` method look like? Well, that's our next stop. Here it is:

```
this.init = function() {

  // Use Prototype to load the configuration file.
  new Ajax.Request(
    JSVConfig.pathPrefix + JSVConfig.configFile,
    { method: "get", onComplete: this.initCallback }
  );

} // End init().
```

Was it all you had hoped it would be? Yeah, me neither! All that's being done here is the configuration file specified in the `JSVConfig` structure is being loaded via Prototype's Ajax support. We are declaring that when the response is received from the server—that is, the configuration file itself—the `initCallback()` method of the `JSValidator` class should be called. So, as you might expect, that method is what we'll look at now.

The initCallback() Method

Because the `initCallback()` method is somewhat long, we'll look at it in a couple of pieces to better comprehend each task it performs. These chunks will be described in the order they appear, so putting the following chunks together gives you the complete method.

Configuring the JSDigester Rules

Let's begin with this bit of code:

```
var jsDigester = new JSDigester();

// Create new object when JSValidatorConfig tag encountered.
jsDigester.addObjectCreate("JSValidatorConfig",
  "JSValidatorConfig");

// Create new object when JSValidatorConfig/validator tag encountered,
// populate its properties and add it to the JSValidatorConfig object
// on the top of the stack.
jsDigester.addObjectCreate("JSValidatorConfig/validator",
  "JSValidatorValidatorConfig");
jsDigester.addSetProperties("JSValidatorConfig/validator");
jsDigester.addSetNext("JSValidatorConfig/validator", "addValidator");

// Create new object when JSValidatorConfig/message tag encountered,
// populate its properties and add it to the JSValidatorConfig object
// on the top of the stack.
jsDigester.addObjectCreate("JSValidatorConfig/message",
  "JSValidatorMessage");
jsDigester.addSetProperties("JSValidatorConfig/message");
jsDigester.addSetNext("JSValidatorConfig/message", "addMessage");

// Create new object when JSValidatorConfig/form tag encountered,
// populate its properties and add it to the JSValidatorConfig object
// on the top of the stack.
jsDigester.addObjectCreate("JSValidatorConfig/form",
  "JSValidatorForm");
jsDigester.addSetProperties("JSValidatorConfig/form");
jsDigester.addSetNext("JSValidatorConfig/form", "addForm");
```

```
// Create new object when JSValidatorConfig/form/validation tag encountered,
// populate its properties and add it to the JSValidatorForm object
// on the top of the stack.
jsDigester.addObjectCreate("JSValidatorConfig/form/validation",
  "JSValidatorFormValidation");
jsDigester.addSetProperties("JSValidatorConfig/form/validation");
jsDigester.addSetNext("JSValidatorConfig/form/validation", "addValidation");

// Create new object when JSValidatorConfig/form/validation/param tag
// encountered, populate its properties and add it to the
// JSValidatorFormValidation object on the top of the stack.
jsDigester.addObjectCreate("JSValidatorConfig/form/validation/param",
  "JSValidatorFormValidationParam");
jsDigester.addSetProperties("JSValidatorConfig/form/validation/param");
jsDigester.addSetNext("JSValidatorConfig/form/validation/param",
  "addParam");

// Parse config.
config = jsDigester.parse(inRequest.responseText);
```

All of this, in a nutshell, is the configuration of the JSDigester rules that correspond to the JSValidator configuration file. Because JSDigester was already covered (in the previous chapter), going over all the rules in detail would be a bit of a waste. However, if any of this code doesn't make sense to you, I highly recommend rereading Chapter 7.

The outcome of these rules being executed on the configuration file is that the `config` field in the `JSValidator` class is an instance of `JSValidatorConfig`, with all its children populated from the configuration information. This code exists in the `initCallback()` function, which is passed the request object by Prototype for the Ajax request made to retrieve the configuration file. Therefore, the XML to be parsed can be accessed via the `responseText` attribute of the `inRequest` object, as you can see in the call to `jsDigester.parser()`.

Adding the Built-in Validators

Once the configuration file has been parsed, the next task we need to perform is to add in the basic built-in validators: the `Required`, `Regex`, and `MinLength` validators. The code that accomplishes this is as follows:

```
// Add in the basic validators.
var requiredValidatorConfig = new JSValidatorValidatorConfig();
requiredValidatorConfig.setId("required");
requiredValidatorConfig.setSrc("");
requiredValidatorConfig.setClass("RequiredValidator");
config.addValidator(requiredValidatorConfig);
var regexValidatorConfig = new JSValidatorValidatorConfig();
regexValidatorConfig.setId("regex");
regexValidatorConfig.setSrc("");
```

```
regexValidatorConfig.setClass("RegexValidator");
config.addValidator(regexValidatorConfig);
var minLengthValidatorConfig = new JSValidatorValidatorConfig();
minLengthValidatorConfig.setId("minLength");
minLengthValidatorConfig.setSrc("");
minLengthValidatorConfig.setClass("MinLengthValidator");
config.addValidator(minLengthValidatorConfig);
```

For each of those three validators, the code instantiates a new `JSValidatorValidatorConfig` object. We then populate the three attributes it contains: `id`, `src`, and `class`. Note that the `src` attribute is set to an empty string because there is no file to import. As you will recall, the code for these validators was imported already as part of the global code executed when this `.js` file was loaded. In fact, there is no real reason to set the `src` attribute at all for these validators, but I prefer code to be as deterministic as possible, even when it doesn't really have to be.

Loading Custom Validators

We still have some work to do for validators. We now need to import the JavaScript files that may have been named for any custom add-on validators, such as the `DateValidator`. When this code executes, the page is already loaded, so we can't simply use the `document.write()` trick as we did earlier. Instead, we need to get just slightly fancier:

```
// Add includes for external validators.
var configuredValidators = config.getValidators();
for (var validatorID in configuredValidators) {
  var nextValidatorConfig = configuredValidators[validatorID];
  // Only non-basic validators will have a src value specified.
  if (nextValidatorConfig.getSrc() != "") {
    var scriptTag = document.createElement("script");
    scriptTag.src = nextValidatorConfig.getSrc();
    var headTag = document.getElementsByTagName("head").item(0);
    headTag.appendChild(scriptTag);
  }
}
```

First, we get the collection of validators that were parsed from the configuration file and iterate over that collection. For each, we get the appropriate `JSValidatorValidatorConfig` object, and we see if its `src` attribute has a value (because, as you'll remember, we just added the basic validators, which all had the `src` attribute set to an empty string—see, it's good to be deterministic!). Once we have the `src` attribute value, we use the DOM `createElement()` method to create a new `<script>` tag. We set the `src` attribute of that tag to the value specified for the validator. Then a reference to the `<head>` of the document is retrieved, and the new `<script>` tag is appended. The browser will then immediately load the source JavaScript file and evaluate it. We now have available to the rest of the code the class for the validator configured.

Attaching Event Handlers

The next and final step to complete initialization is to actually hook up the configured events to the form fields that were configured to have validations attached to them. That is accomplished with the next bit of code, which may look a bit daunting, but really if you were to trim out the comments, you would find it's a bit simpler than you might first perceive.

```
// Attach event handlers to fields as defined in config file.
var configuredForms = config.getForms();
// Iterate over forms configured.
for (var formName in configuredForms) {
  var nextFormConfig = configuredForms[formName];
  // Get reference to form being configured.
  var targetForm = document.forms[nextFormConfig.getName()];
  // Attach an onSubmit handler to check if it can submit or not.
  targetForm.onsubmit = jsValidator.processSubmit;
  // Get reference for all validations configured for this form.
  var formValidations = nextFormConfig.getValidations();
  // Iterate over validations defined for this form.
  for (var fieldName in formValidations) {
    // Get the field validation being hooked.
    var nextValidationConfig = formValidations[fieldName];
    // Get the validator definition.
    var validator = config.getValidator(nextValidationConfig.getType());
    // Get the field to hook event to.
    var targetField = targetForm[nextValidationConfig.getField()];
    // Set attribute if this field is initially invalid. and if the field
    // is configured for the highlight action, then highlight it.
    if (nextValidationConfig.getStartInvalid() &&
      nextValidationConfig.getStartInvalid() == "true") {
      targetField.setAttribute("JSValidator_INVALID", "true");
      if (nextValidationConfig.getFailAction() == "highlight") {
        var idToHighlight =
          nextValidationConfig.getParam("idToHighlight").getValue();
        var errorStyleClass =
          nextValidationConfig.getParam("errorStyleClass").getValue();
        $(idToHighlight).className = errorStyleClass;
      }
    }
    // Set event handler.
    targetField[nextValidationConfig.getEvent()] = jsValidator.processEvent;
  }
}
```

First, the collection of forms configured is retrieved. The code then begins to iterate over that collection. For each, we get a reference to the HTML for it via the `document.forms[]` collection. An `onSubmit` event handler is attached to the form. This event handler is the `processSubmit()` function in the `JSValidator` object, as will be discussed shortly.

After that, the collection of validations for that form is retrieved from the `JSValidatorConfig` object, and we begin to iterate over it. For each, we get a reference to the field in the HTML form. Next, we see if either the `startInvalid` attribute was not specified at all or if it was specified with the value `true`. If either condition is true, we add an attribute to the field named `JSValidator_INVALID` with a value of `true` (the value is actually irrelevant; just the presence or absence of the attribute determines if the field is invalid). We also see if the `failAction` for that field is `highlight`. If so, we go ahead and get the ID of the page element to highlight and the style class used for highlighting, and we apply it using Prototype's `$()` shorthand for `document.getElementById()`.

Finally, we set the appropriate event handler for the field to point to the `processEvent()` method of the `JSValidator` class.

All of this is really a long-winded way of saying that for every field configured to have a validation applied to it in the configuration file, we're setting the appropriate event handler(s) on the field and setting each field's initial state. In this way, if JavaScript is turned off, the form will still work because it isn't dependent on JSValidator at all; the framework is strictly an add-on capability. This is good, nonintrusive coding.

With `init()` and `initCallback()` out of the way, we now turn our attention to the method that is called in response to an event on a field that triggers a validation.

The processEvent() Method

Recall that for each field that has a validation assigned to it, in `initCallback()`, we attached the appropriate event handler, as described in the configuration file, but it always calls `processEvent()` in the `JSValidator` class. This allows the framework to handle all these events in a common way. Let's now see exactly what this event-handler function does:

```
this.processEvent = function() {

  // Get reference to form, field and validator config objects for the
  // form element that fired the event that called this callback.
  var formConfig = config.getForm(this.form.name);
  var fieldConfig = formConfig.getValidation(this.name);
  var validatorConfig = config.getValidator(fieldConfig.getType());

  // Get a reference to the class that implements the validator defined
  // for this field.  Then, get a new instance of it and call its validate()
  // method.
  var clazz = eval(validatorConfig.getClass());
  clazz = new clazz;
  clazz.setJsValidatorConfig(config);
  clazz.setFieldConfig(fieldConfig);
  clazz.setValidatorConfig(validatorConfig);
  clazz.setField(this);
  // Perform the appropriate action for pass and fail.
  var isValid = clazz.validate();
  if (isValid) {
    // When field was valid, might be some cleanup to do in some cases.
    this.removeAttribute("JSValidator_INVALID");
```

```
      if (fieldConfig.getFailAction() == "highlight") {
        var idToHighlight = fieldConfig.getParam("idToHighlight").getValue();
        var okStyleClass =
          fieldConfig.getParam("okStyleClass").getValue();
        $(idToHighlight).className = okStyleClass;
      }
      if (fieldConfig.getFailAction() == "insert") {
        var targetID = fieldConfig.getParam("idToInsertInto").getValue();
        $(targetID).innerHTML = "";
      }
    } else {
      // Field NOT valid, so act according to config.
      this.setAttribute("JSValidator_INVALID", "true");
      switch (fieldConfig.getFailAction()) {
        case "alert":
          var whatMessage = fieldConfig.getParam("message");
          var messageConfig =
            jsValidator.getConfig().getMessage(whatMessage.getValue());
          var message = messageConfig.getText();
          message = jsValidator.replaceTokens(message, fieldConfig.getParams());
          alert(message);
        break;
        case "highlight":
          var idToHighlight = fieldConfig.getParam("idToHighlight").getValue();
          var errorStyleClass =
            fieldConfig.getParam("errorStyleClass").getValue();
          $(idToHighlight).className = errorStyleClass;
        break;
        case "insert":
          var whatMessage = fieldConfig.getParam("message");
          var targetID = fieldConfig.getParam("idToInsertInto").getValue();
          var messageConfig =
            jsValidator.getConfig().getMessage(whatMessage.getValue());
          var message = messageConfig.getText();
          message = jsValidator.replaceTokens(message, fieldConfig.getParams());
          $(targetID).innerHTML = message;
        break;
      }
    }

    return isValid;

  } // End processEvent().
```

First, we get a reference to the `config` objects for the form, field, and validator the field uses. Next, the specified validator is instantiated. This is done by first `eval()`'ing the name of the class returned by the call to `validatorConfig.getClass()`, which gives us a reference to the class, then creating a new instance of it by using the `new` keyword. We set on that validator

instance the JSValidatorConfig instance by passing it to setJsValidatorConfig(), the JSValidatorFormValidation for the field by passing it to setFieldConfig(), and the JSValidatorValidatorConfig by passing it to setValidatorConfig(). We also pass the field itself that triggered the event by passing it to setField().

The validator should now have access to all the pieces of information it might need, so we then call validate() on the validator. validate() returns true if the field passes validation; false if not. If validate() returns true, we essentially need to undo any failure indications that the field might have had. That means that first we need to remove that JSValidator_INVALID indicator, if it was present, and if the failAction for the field is highlight, we need to unhighlight the target element.

Lastly, if the failAction for the field is insert, then we want to clear the innerHTML property of the target element. You can see all this in action with the Last Name field. That field starts out invalid, and as you type and enter five characters or more, the field label reverts to a valid entry condition. This is due to this code in this block.

What if the field fails validation? Well, then we find ourselves in the else clause seen in this code. In that case, we examine the failAction value, and then switch on it. For action alert, we get the message referenced in the configuration, and then call replaceTokens() to do token replacement. Finally, we display the resultant message via alert().

For highlight, we just need to get the ID of the element to highlight, as well as the style class to use. Then we set the className attribute accordingly, again using Prototype's $() shortcut.

Finally, for the insert action, we do essentially the same thing as for alert, but updating the innerHTML property of the target element instead of calling alert() at the end.

The processSubmit() Method

Another function that is of concern to us is the onSubmit event for the form itself. Quite obviously, if any of the fields on the form are invalid, we should not allow the form to submit. But how do we determine whether the form should submit or not? What are the actual mechanics behind it? The answer to that is found in the processSubmit() method of JSValidator, shown here:

```
this.processSubmit = function() {

  var formValidity = true;
  // If any element of the form has the JSValidator_INVALID attribute, then
  // the form cannot be submitted.
  for (var i = 0; i < this.elements.length; i++) {
    if (this.elements[i].getAttribute("JSValidator_INVALID")) {
      formValidity = false;
    }
  }
  if (!formValidity) {
    // Can't be submitted, show configured message.
    var config = jsValidator.getConfig();
    var formConfig = config.getForm(this.name);
    alert(formConfig.getNoSubmitMessage());
  }
  return formValidity;

} // End processSubmit().
```

As you can see, all we really need to do is iterate over all the elements in the form and check if any have the `JSValidator_INVALID` attribute. If not, then the form is OK and submission can continue. If any elements have that attribute, then we get the value of the `noSubmitMessage` attribute for the form and display it via `alert()` to indicate an error is present on the form.

Note here the use of the `this` reference (you'll also note that the same thing is used in the `processEvent()` method). You may find this interesting because you may expect that `this` points to the `JSValidator` instance, since these methods are part of that instance. But recall that these two methods were attached to elements on the page: the form and the form elements. When they are called, it is in the context of those elements, and thus the `this` keyword actually points to those elements. Remember that the `this` reference always refers to the object the method is attached to *at run time*, not necessarily design time. It's easy to get confused by this, and it will most likely burn you once or twice at late hours of the night!

The replaceTokens() Method

The last bit of functionality provided by JSValidator is the `replaceTokens()` method, which is used when generating the messages the user sees when a field fails validation, if configured to do so. Let's have a look at that method now, shall we?

```
this.replaceTokens = function(inString, inParams) {

  // We're going to scan the text looking for tokens, all the while
  // constructing a new string in a StringBuffer from it, with the
  // data replacing the tokens.
  var finalText = "";
  var i = 0;
  while (i < inString.length) {
    // See if the next character is a hash sign, and if the next
    // character after that is an opening brace, as long as
    // that check doesn't put us beyond the end of the string, then we've
    // found the start of a token.
    if (inString.charAt(i) == '#' && inString.charAt(i + 1) == '{') {
      // Now we get the location of the closing token delimiter.  Note that
      // if the developer forgot to close the token, this will probably
      // blow up with a JS error, and at best it just won't work as
      // expected.  We're going to live with that!
      var lIndex = inString.indexOf("}#", i);
      // Now it's a simple matter to get the token name.
      var tokenName = inString.substring(i + 2, lIndex);
      // Look up the replacement value with that name from inParams.
      var tokenValue = "";
      var param = inParams[tokenName];
      if (param) {
        tokenValue = param.getValue();
      }
```

```
        finalText += tokenValue;
        // Set i to take us just past the closing token delimiter, and
        // we're done with this token
        i = lIndex + 1;
      } else {
        // The current character being checked was NOT part of a token
        // opening delimiter, so just append the character
        finalText += inString.charAt(i);
      }
      i++;
    }
    return finalText;

  } // End replaceTokens().
```

`replaceTokens()` is a fairly straightforward bit of string manipulation. It begins by iterating over each character in the input string, which will be one of the message strings, possibly containing replacement tokens in the form #{*xxx*}#.

For each character, we see if it's a hash mark (#). If it is, we then see if the *next* character is a brace ({). If either of these conditions is false, the code in the `else` block executes, and the character is simply added to a string we began constructing at the start. However, if both conditions are true, we've identified a replacement token. Now we need to find the location of the closing delimiter (}#). Once we have that, we can get the name of the token easily enough by using the `substring()` method of the `String` object.

With the token name in hand, we can look it up in the collection of parameters passed in. Note that in this instance, *collection of parameters* actually refers to the `Object` to which all the configured parameters for the validation being processed are attached. We can then do a simple lookup for the token. That's because, as you'll remember, the whole point of using `Object` rather than `Array` was so that we could easily look up these values using the [] paradigm. Once we find the value, we simply append the value of the parameter to the output string, and move the iteration index past the closing token delimiter. Once the iteration completes, we return the constructed string, and *voilà*, a string with all tokens replaced with the appropriate parameter values!

And with that, we've seen the guts of `JSValidator` in detail! Only two last bits of code remain to be examined, and those are the validator implementation classes—both the basic built-in ones and the one add-on validator.

Writing JSValidatorBasicValidators.js

The `JSValidatorBasicValidators.js` file, as the name clearly implies, is where you'll find the three basic built-in validators that JSValidator always makes available automatically: `RequiredValidator`, `RegexValidator`, and `MinLengthValidator`.

The RequiredValidator Class

`RequiredValidator`, shown in Figure 8-16, is just about as simple as a validator can get, as you can see here:

```
function RequiredValidator() {

  this.validate = function() {

    var retVal = true;
    if (this.field.value == "") {
      retVal = false;
    }
    return retVal;

  } // End validate().

} // End RequiredValidator().

// RequiredValidator extends JSValidatorValidatorImpl.
RequiredValidator.prototype = new JSValidatorValidatorImpl;
```

Figure 8-16. *UML diagram of the RequiredValidator class*

This class contains just a quick check of the value of the field that fired the validation to make sure something was entered, and nothing more. Note the setting of the `prototype` of the validator at the end. This is where the inheritance from the `JSValidatorValidatorImpl` class happens, and you'll see this same basic line of code for each validator. Likewise, for your own validator, you would need to include that line of code.

The RegexValidator Class

Next up is `RegexValidator`, which isn't much more complex at all. First, have a look at Figure 8-17, the UML diagram for the `RegexValidator` class.

Figure 8-17. *UML diagram of the RegexValidator class*

Moving on to the code, we see that, in this case, we need to get the regular expression from the parameters defined for this validation. To do so, we call `getParam()` on the `fieldConfig` object for this validation, and subsequently we call `getValue()` on the object returned from that call, since it is a `JSValidatorFormValidationParam`, not the value itself. With that done, we have only to use the typical JavaScript regular expression `match()` function to determine if the field's value is valid.

```
function RegexValidator() {

  this.validate = function() {

    var retVal = true;
    var parm = this.fieldConfig.getParam("regex");
    var regx = parm.getValue();
    if (!this.field.value.match(regx)) {
      retVal = false;
    }
    return retVal;

  } // End validate().

} // End RegexValidator().
```

```
// RegexValidator extends JSValidatorValidatorImpl.
RegexValidator.prototype = new JSValidatorValidatorImpl;
```

RegexValidator is one of those cases where something seemingly simple masks a great deal of power. You have the full capability of the JavaScript regular expression engine at your fingertips, without writing a single bit of code.

The MinLengthValidator Class

The last basic validator to look at is MinLengthValidator, which is another very simple bit of code. As Figure 8-18 indicates, this is again just a typical validator class.

Figure 8-18. *UML diagram of the MinLengthValidator class*

Take a look at the code:

```
function MinLengthValidator() {

  this.validate = function() {

    var retVal = true;
    if (this.field.value.length <
      this.fieldConfig.getParam("minLength").getValue()) {
      retVal = false;
    }
    return retVal;

  } // End validate().
```

```
} // End MinLengthValidator().

// MinLengthValidator extends JSValidatorValidatorImpl.
MinLengthValidator.prototype = new JSValidatorValidatorImpl;
```

As you can see, as with `RegexValidator`, all we're doing is grabbing the minimum allowed length for the field from the parameters for the validation, comparing the value of the field to it, and returning the appropriate Boolean outcome.

Writing DateValidator.js

Only one piece of code stands between us and a complete examination of this application, and that's the `DateValidator` class, as shown in Figure 8-19.

Figure 8-19. *UML diagram of the DateValidator class*

Let's get the code out of the way, and then see what makes it tick.

```
function DateValidator() {

  this.validate = function() {

    // Get the configured format of the field.
    var format = this.fieldConfig.getParam("format").getValue();
```

```
    // Make sure the value is the same length as the format.
    if (this.field.value.length != format.length) {
      return false;
    }

    // Now iterate over the value.  If any character doesn't match the format,
    // it's a reject.  Note that M, D and Y characters in the format
    // correlate to any numeric character.
    for (var i = 0; i < format.length; i++) {
      if (format.charAt(i).toUpperCase() == "M" ||
        format.charAt(i).toUpperCase() == "D" ||
        format.charAt(i).toUpperCase() == "Y") {
        // Character at this position should be a numeric.
        if (this.field.value.charAt(i) < '0' ||
          this.field.value.charAt(i) > '9') {
          return false;
        }
      } else {
        // Format doesn't specify a numeric value at this position, so just
        // be sure the character matches exactly.
        if (format.charAt(i) != this.field.value.charAt(i)) {
          return false;
        }
      }
    }

    return true;

  } // End validate().

} // End DateValidator().

// DateValidator extends JSValidatorValidatorImpl.
DateValidator.prototype = new JSValidatorValidatorImpl;
```

The first thing we see in the `validate()` method is grabbing the format to use from the configuration. We then see a trivial rejection: if the value of the field doesn't exactly match that of the format string, we know the field is invalid.

Assuming it passes that little test though, we then iterate over the value of the field. For each character, we compare it to what is expected in that position in the format string. Any `M`, `D`, or `Y` character in the format string corresponds to any digit 0 through 9. If the character in the format string is something else, typically a slash or dash, then we make sure the character in the value matches exactly. If we ever hit a situation where the character in the input string doesn't match that in the format string, then the field fails validation and we return `false`.

And that's that—another application in the books, so to speak!

Suggested Exercises

I've left the door open for a number of enhancements for you to do that should very much help further your understanding of JSValidator and, of course, JavaScript in general. Here are just a few suggestions:

- A fairly simple one first: when a validation failure occurs, the field that had the error should get focus automatically. This is definitely an obvious one, and one I left for you to start with because it shouldn't take very long or be very difficult, so it's a good way to get your feet wet.
- Another obvious one is to implement more validators. How about one that validates credit card numbers? How about one to check that an entered date falls within a given range? How about one that does greater-than or less-than comparisons of two fields (which would require you to modify the framework to allow a validation configuration to name a second field to operate on)? Whatever your imagination can come up with, go for it!
- Allow for multiple validations on a single field. This may not be quite as simple as it sounds, but would definitely be a worthy goal.
- Add internationalization (I18n) support. This could be accomplished a number of ways. One way might be to add a locale identifier to the `<message>` element—maybe it has `en` for English, `de` for German, and so on. Then just get the appropriate message based on the locale of the client.
- Allow for adding error messages as pop-up tooltips to highlighted error fields. That way, when a field is highlighted to show an error, the user can hover over it to see what's wrong.

Summary

In this chapter, we developed an extensible framework that can completely externalize our form validations, requiring virtually no code be added to our pages. You saw how you can add reusable validators to your framework that you can find uses for in other applications. And you can accomplish all this in a nice, neat, object-oriented way that lends itself to extension.

In this chapter's project, you saw how the JSDigester project from the previous chapter helps us deal with XML easily. You also got a glimpse of the Prototype library in action a bit, and had another brush with Ajax, albeit just a quick one. We'll get to a project that specifically deals with Ajax specifically in Chapter 12.

CHAPTER 9

Widget Mania: Using a GUI Widget Framework

In web development, widgets are all the rage these days. No longer are we content with regular form fields, buttons, drop-down lists, and such. Just plain-old tables aren't sufficient for many developers! But this trend isn't just due to a desire to create cooler interfaces. There are UI metaphors in a modern operating system that don't have an analogy in the web world, at least not intrinsically.

Take the tree view as an example. I'm sure you've seen a tree view before. Indeed, if you work in Windows, you can barely avoid it (it's the list of folders on the left in Windows Explorer). Have you seen one on the Web? Chances are you have, but unless it was relatively recently, the web developer likely wrote it himself, or else lifted some code from somewhere else. What it wasn't though, in all likelihood, was a piece of code that was self-contained and part of a larger framework of UI components, which is what a UI widget is.

Widgets make adding this "advanced" functionality to your web application very easy, and more important, consistent. You'll see how this works as we build a handy little application for posting notes to ourselves. In this project, we'll use a lot of widgets supplied by a library that is rapidly growing in popularity: the Yahoo! User Interface (YUI) Library.

JSNotes Requirements and Goals

For this chapter's project, which we'll call JSNotes from here on out, we're going to build something along the lines of that pad of sticky yellow notes you very possibly having sitting around your desk somewhere. In this case though, it will be a web-based application that allows us to post digital notes to ourselves. The main purpose of this application, aside from being useful in helping to keep us from losing little pieces of information we obtain throughout the day, is meant to demonstrate the use of the YUI Library.

So, what specifically is JSNotes going to do? Let's spell it out now:

- JSNotes will allow us to create notes, including a subject for each, and add a date and time (generally, the current date and time, but you never know!). We'll also be able to categorize each note as either personal or business.
- The notes will be presented in a Windows Explorer-like view, with a tree view on the left and note details on the right. The two primary branches in the tree will be our two categories: Personal and Business.

- For each note, we'll store the note text, a subject, and a date and time, as well as the note's category.
- We'll be able to delete a note, and we'll also be able to "export" a note, which really just means put it in a form suitable for copying and pasting into another program.
- When adding a note, we want to present it in a pop-up dialog box.
- We'll also have Help and About boxes, but they will be done in a different style than the add note pop-up. These will appear as an overlay over the main JSNotes display.

The best part is, with the YUI Library in the mix, all of this becomes a relatively trivial exercise!

The YUI Library

The YUI Library includes a set of UI widgets, as well as a number of utility-type classes for Ajax, drag-and-drop, DOM manipulation, and more. YUI is one of the best documented libraries I've seen recently and also one of the best demonstrated. You can find a good example of virtually every part of the library, usually along with a number of variations for guidance.

Let's talk about the widgets, since they are primarily what we'll be working with in this application, and we'll be using quite a few of them. YUI provides a number of widgets, including AutoComplete, Calendar, Container (including Module, Overlay, Panel, Tooltip, Dialog, and SimpleDialog), Logger, Menu, Slider, TabView, and TreeView. Because the widgets are designed around a common framework, they all present a fairly consistent programming interface.

Using YUI is as simple as importing some JavaScript files, and in many cases, some CSS files as well. Generally speaking, each widget is a single JavaScript file, but there also may be some dependencies on other files, and those need to be manually imported, too. Once that's done, you're ready to rock and roll.

For instance, let's say you want to create a menu. All you need to do is define your menu in the form of some simple markup, like so:

```
<div id="basicmenu" class="yuimenu">
  <div class="bd">
    <ul class="first-of-type">
      <li class="yuimenuitem"><a href="page1.htm">Page1</a></li>
      <li class="yuimenuitem"><a href="page2.htm">Page2</a></li>
    </ul>
  </div>
</div>
```

This provides the basic structure of the menu. The last step is to tell YUI to create the menu for you, based on this markup. To do so is as simple as this:

```
var oMenu = new YAHOO.widget.Menu("basicmenu");
oMenu.render();
```

YUI will find the element on the page with the ID `basicmenu`. It will then parse the contents, specifically looking for a list, and will use it to generate the menu. The markup will be replaced with the markup for the menu itself, and the menu will be all set. The first line of code instantiates the menu and stores a reference to it in `oMenu`, but it only renders it in memory. To actually get it on the screen is the job of the call to its `render()` method.

You'll find that most widgets follow the same general pattern: instantiate an instance of the widget, give it a reference to some markup, and then make a call to its `render()` method to put it on the screen. Some widgets support options passed into the constructor as well, as you'll see in the application code. A few of them are created a little differently, and you'll see that as well when we build the application.

You can also programmatically create the widgets from scratch. For instance, to create a menu strictly with code, you could do this:

```
var oMenu = new YAHOO.widget.Menu("basicmenu");
oMenu.addItem(new YAHOO.widget.MenuItem("Page1", { url : "page1.htm" } ));
oMenu.addItem(new YAHOO.widget.MenuItem("Page2", { url : "page2.htm" } ));
oMenu.render(document.body);
```

In this case, presuming an element with the ID `basicmenu` is not already on the page, the DOM structure of the menu will be created and appended under that ID to the document's `body` element.

As I mentioned, YUI also provides some utility-type functions. Here's one we'll use in the JSNotes application:

```
var obj = YAHOO.util.Dom.get("xxx");
```

This is equivalent to the following ubiquitous function:

```
var obj = document.getElementById("xxx");
```

You may wonder how these functions differ. The answer is that the YUI version accepts either a single ID or an array of IDs, and will return a reference either to a single element or an array of elements. If nothing else, that will save you a lot of typing and lines of code if you need to get a reference to more than one element at a time.

The YUI Library has gained quite a following, and for good reason. It works very well in all the major browsers, is well designed, and is fantastically documented. I wholeheartedly recommend its use, and I encourage you to check it out in more detail at `http://developer.yahoo.com/yui`. Although we'll use quite a few of the widgets, we won't touch much of the rest of the library. I encourage you to check it out in more detail at `http://developer.yahoo.com/yui`. Spend five minutes clicking around the examples on the site, and you'll have a good feel for what is has to offer.

A Preview of JSNotes

If you haven't yet fired up JSNotes, I suggest you do so now. Let's take a quick look at it. Figure 9-1 shows the application as it is at startup, and already you can see two YUI widgets: the menu and the tree view.

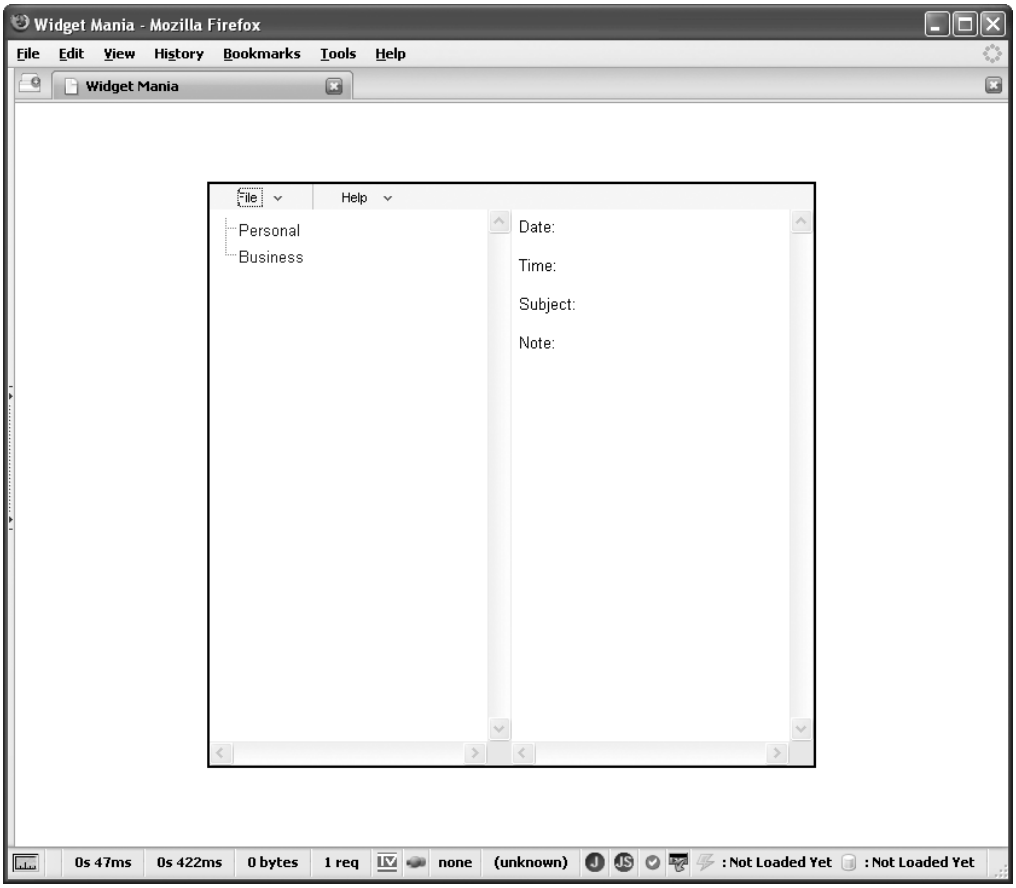


Figure 9-1. *JSNotes at startup*

When you want to add a note, you click the Add Note option on the File menu. This brings up a dialog box, where you can enter the note. The dialog box demonstrates another bunch of widgets, including the dialog box itself, the calendar, and the slider, as you can see in Figure 9-2.

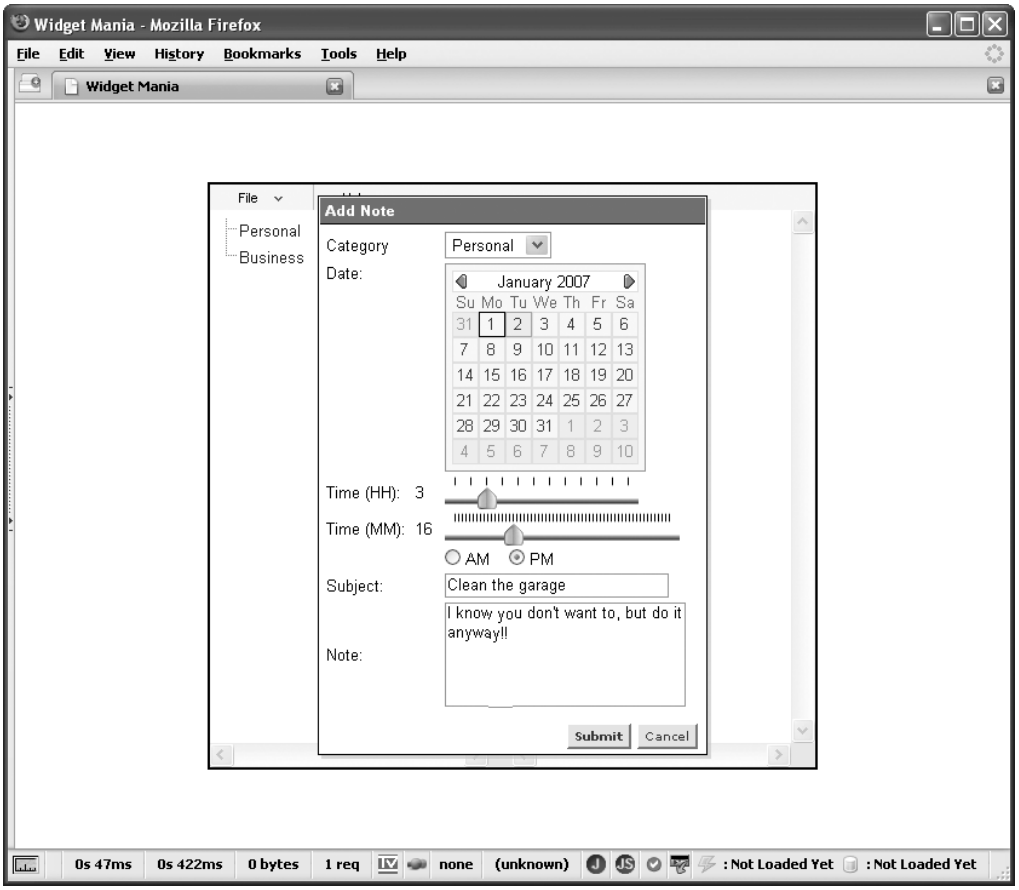


Figure 9-2. *JSNotes when adding a note*

In Figure 9-3, you can see one more widget: the overlay. Admittedly, it doesn't look like anything special. In fact, you really can't even tell that you're looking at a widget.

You'll see one or two other screenshots as we progress through the dissection of the application. See, now you have something to look forward to! And with that little tease, let's get on with the show.

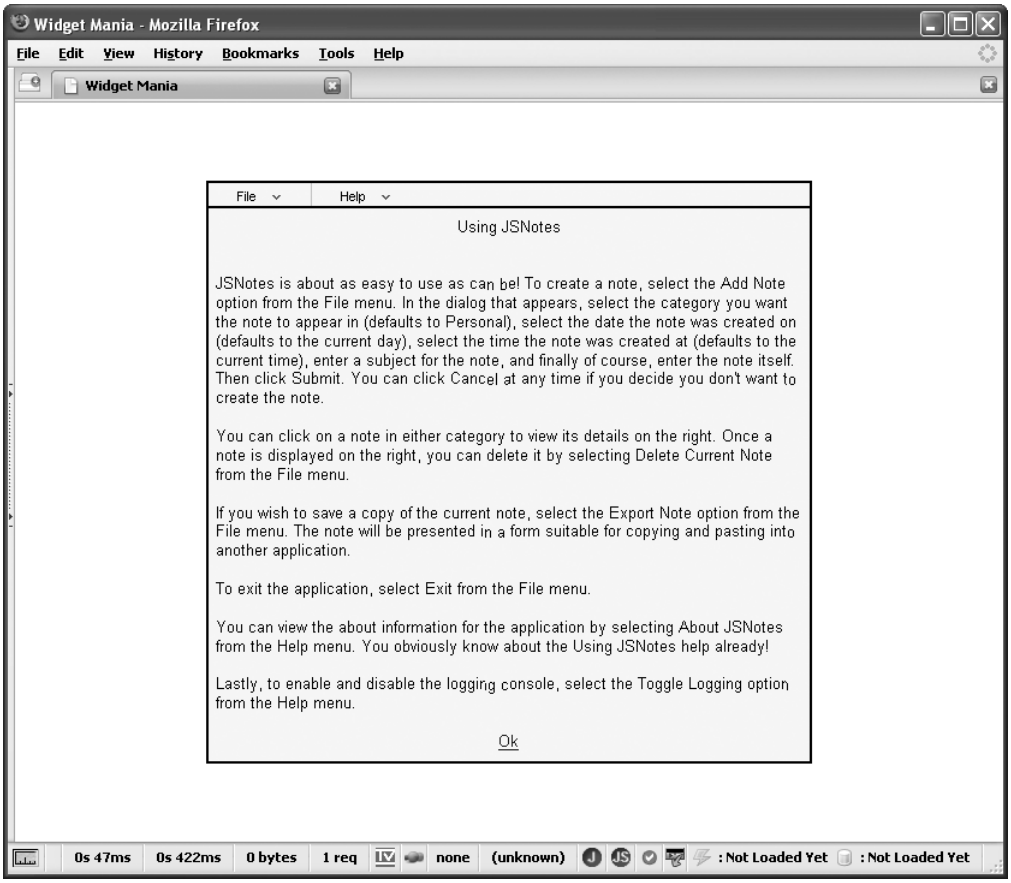


Figure 9-3. *The Using JSNotes overlay*

Dissecting the JSNotes Solution

Let's begin, as we always do, by looking at the general file layout of the application. If you've read the previous chapters, the structure shown in Figure 9-4 should be familiar by now.

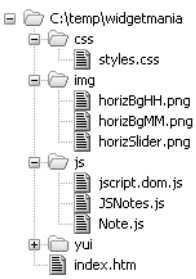


Figure 9-4. *JSNotes directory structure*

In the root directory resides a single file, `index.htm`, which is the starting point for the application. Below the root are the following four subdirectories:

- `css`: As usual, the `css` directory contains `styles.css`, which is the style sheet for the application.
- `img`: The `img` directory holds the three files `horizBgHH.png`, `horizBgMM.png`, and `horizSlider.png`. The first two are the background for the hours and minutes sliders, respectively, in the Add Note dialog box. They are the tick marks you see. The `horizSlider.png` file contains the slider handle that you drag and slide.
- `js`: In the `js` directory, you'll find three files. `jscript.dom.js` is the DOM package from Chapter 3. `JSNotes.js` is the main JavaScript code for the application. `Note.js` is the `Note` class, which contains the data of a note created by the user.
- `yui`: The `yui` directory is where the Yahoo library is housed. Within it, you'll find a number of directories, each corresponding to a part of YUI functionality. It also includes any resources YUI needs, such as style sheets and images.

Not a lot of files are involved (unless you count YUI itself, but for the purposes of this book, we won't). So, let's dive right in and check it out!

Writing index.htm

`index.htm` is the file you load to bring up the application, and it is what brings form to the function, so to speak. It is a fairly large file, so we will just look at some important pieces of it here.

At the top, in the `<head>` of the document, we first see a batch of style sheet imports:

```
<link rel="stylesheet" type="text/css" href="css/styles.css">
<link rel="stylesheet" type="text/css" href="yui/logger/assets/logger.css">
<link rel="stylesheet" type="text/css" href="yui/fonts/fonts.css">
<link rel="stylesheet" type="text/css" href="yui/reset/reset.css">
<link rel="stylesheet" type="text/css" href="yui/menu/assets/menu.css">
<link rel="stylesheet" type="text/css"
  href="yui/container/assets/container.css">
<link rel="stylesheet" type="text/css"
  href="yui/calendar/assets/calendar.css" />
<link rel="stylesheet" type="text/css"
  href="yui/treeview/assets/tree.css" />
```

Except for the first, which is the style sheet for the application itself, all of these are needed for YUI widgets. (Feel free to tear them apart if you're interested.)

Following that is a batch of JavaScript source file imports:

```
<script type="text/javascript" src="js/jscript.dom.js"></script>
<script type="text/javascript" src="js/JSNotes.js"></script>
<script type="text/javascript" src="js/Note.js"></script>
<script type="text/javascript" src="yui/yahoo/yahoo.js"></script>
<script type="text/javascript" src="yui/event/event.js"></script>
<script type="text/javascript" src="yui/dom/dom.js"></script>
<script type="text/javascript" src="yui/dragdrop/dragdrop.js" ></script>
```

```
<script type="text/javascript" src="yui/connection/connection.js" ></script>
<script type="text/javascript" src="yui/logger/logger.js"></script>
<script type="text/javascript" src="yui/container/container.js"></script>
<script type="text/javascript" src="yui/menu/menu.js"></script>
<script type="text/javascript" src="yui/animation/animation.js"></script>
<script type="text/javascript" src="yui/calendar/calendar.js"></script>
<script type="text/javascript" src="yui/slider/slider.js"></script>
<script type="text/javascript" src="yui/treeview/treeview.js"></script>
```

Once again, except for the first three, which are part of JSNotes itself, the rest are parts of YUI. Generally speaking, each widget is encapsulated in a `.js` file named for the widget, and may also require some supporting files, such as `event.js` and `dom.js`.

Following that is the instantiation of the `JSNotes` object, which is the real code behind the application. The instantiated object is referenced by the variable `jsNotes`, which you'll be seeing plenty of later.

After that, the `<body>` begins, starting with the `<div>` named `divMain`. This is the container that is centered on the page. Everything else, except the Add Note dialog box, is a child of this container.

Next up is some markup that makes up our menu bar. This is also our first YUI widget—well, sort of! As you'll see, the menu bar will be generated programmatically, but it does so based on the unordered list in this markup. Basically, we'll be feeding YUI the ID of the `<div>` containing an ordered list—`divMainMenu` in this case. YUI will then parse the list and generate the menu based on it. The classes you see are the default style classes that should be applied to the top-level menu items, as well as the submenu items. Each menu item is a link, but we don't want to navigate anywhere, which is the purpose of the `javascript:void(0)` as the `href` value. The `onClick` event handler takes care of the real functionality behind each menu item.

■ **Tip** You can have nested lists, which will be properly rendered as nested menu items. So, if you wanted a menu item on the File menu to itself be a submenu, you simply start a new list under the `<li>` item under the File `<li>`. Neat!

Next is a `<div>` with the ID `divContent`. This is a fairly mundane affair: two `<span>` elements (instead of `<div>` elements, to avoid the line break between them), one floating left and one floating right (as per the style classes, as you'll see). The one on the left is for the tree view, and the one on the right is for the details of the currently selected note. On the right, we have a series of `<td>` elements with IDs essentially matching the fields stored for each note. It's pretty straightforward markup.

After that, we find a `<div>` where our logging console appears. YUI will take care of creating the content of this `<div>`, so no need to put anything there to start.

Then we get to three `<div>` elements that look quite similar. Each one is the content for a YUI overlay, which is very similar to a `<div>` actually, but exposes some handy extra features, such as methods for centering, custom event monitoring, and a built-in solution for the common problem of `<select>` elements in IE bleeding through elements with a higher Z index.[1] Since all these `<div>`'s look very similar, let's take a peek at just one to get the idea:

1. Basically, the YUI solution puts an `iFrame` behind the overlay, preventing the `<select>` from showing through.

```
<div id="divAbout" class="cssOverlay">
  <table border="0" cellpadding="0" cellspacing="0" class="cssOverlayTable">
    <tr>
      <td align="center" valign="middle" class="cssPadded">
        JSNotes 1.0
        <br><br>
        Frank W. Zammetti
        <br><br><br><br>
        From the book "Practical JavaScript Projects"
        <br>
        Published by Apress in 2007
        <br><br><br><br>
        <a href="javascript:void(0);" onClick="jsNotes.hideAbout();">Ok</a>
      </td>
    </tr>
  </table>
</div>
```

There are no restrictions on what you can place in an overlay (even other YUI widgets!), and you can style it any way you wish. Here, we have the simpler About display, and as you can see, it's pretty typical markup. But when you feed the ID to YUI in the correct way, you get an overlay.

The final piece of `index.htm` to look at is the markup for the Add Note dialog box. You may have been expecting to see something special, based on playing with the application a little and seeing a nice floating dialog box. Well, surprise, it's just more boring markup! What we have is a simple HTML form, set up within a table for alignment purposes. In this case, we won't actually be submitting the form anywhere, so there's no `method` or `action` attributes, and for `onSubmit`, we actually return `false` to be sure no submit occurs. One interesting point is that no submit or cancel buttons are defined here. That's because YUI will create these for us when we create the dialog box. As with overlays and the menu, we'll feed the ID of the containing `<div>` to YUI, and it will do the rest.

The dialog box is a good example of how you can have YUI widgets inside other widgets. We have (or, more precisely, *will* have) a calendar and two sliders in this dialog box. You can see what are essentially the placeholders for these items here: the `addNoteCalendar` `<div>`, the `divHHSliderBG` `<div>`, and the `divMMSliderBG` `<div>`.

As you may have guessed, we aren't simply telling YUI, "Hey, make a menu out of this ID," or "Turn the contents of this `<div>` into a dialog box for me." A bit more work is involved, but not that much! You'll see that when we get to examining `JSNotes.js`. But before we get there, we have one or two stops to make along the way, so let's pull the train out of this station and move on.

Writing styles.css

The style sheet for JSNotes is really fairly vanilla. The highlights are the `cssContentLeft`, `cssContentRight`, and `cssOverlay` classes. Nonetheless, I'll briefly describe what each class is for and point out any attributes of particular interest. So let's start with the style sheet itself, shown in Listing 9-1.

Listing 9-1. *The styles.css File*

```
/* Style for the main DIV. */
.cssMain {
  position         : absolute;
  border           : 2px solid #000000;
  width            : 500px;
  background-color : #ffffff;
}


/* Style for the content container DIV. */
.cssContent {
  background-color : #ffffff;
  width            : 100%;
  height           : 460px;
}


/* Style for the left-hand side content. */
.cssContentLeft {
  width            : 50%;
  height           : 100%;
  float            : left;
  overflow         : scroll;
}


/* Style for the right-hand side content. */
.cssContentRight {
  width            : 50%;
  height           : 100%;
  float            : right;
  overflow         : scroll;
}


/* Style for elements that have padding (overlay contents, etc). */
.cssPadded {
  padding          : 6px;
}
```

```
/* Style for the tables that contain overlay contents. */
.cssOverlayTable {
  width            : 100%;
  height           : 100%;
}


/* Style for the background of the hour slider. */
.cssSliderBGHH {
  position         : relative;
  left             : 0px;
  top              : 0px;
  background       : url(../img/horizBgHH.png) no-repeat;
  height           : 26px;
  width            : 160px;
  zindex           : 5
}


/* Style for the background of the minutes slider. */
.cssSliderBGMM {
  position         : relative;
  left             : 0px;
  top              : 0px;
  background       : url(../img/horizBgMM.png) no-repeat;
  height           : 26px;
  width            : 196px;
  zindex           : 5
}


/* Style for the sliders' handles. */
.cssSliderHandle {
  position         : absolute;
  left             : 0px;
  top              : 8px;
  cursor           : default;
  width            : 18px;
  height           : 18px;
}


/* Style for cells of the table containing the fields of the new note form. */
.cssTDNewNote {
  padding          : 2px;
}
```

```
/* Style for overlays. */
.cssOverlay {
  border           : 2px solid #000000;
  position         : relative;
  left             : 0px;
  _left            : -2px; /* IE */
  background-color : #f7f7ef;
  width            : 100%;
  height           : 460px;
}


/* Style for the elements where hour and minutes are displayed. */
.cssTimeSpan {
  width            : 30px;
}
```

`cssMain` is the class applied to the `<div>` that surrounds all the contents on the page. It is positioned absolutely so that it can be centered. Its width determines the width of the JSNotes display, but note that its height is not set, as this will be determined by its content.

Speaking of its content, the `cssContent` class is applied to the `<div>` that encapsulates the content below the menu bar, namely the tree view and note details. This is where the height is determined. The height here, and the width in `cssMain`, are both specified so that JSNotes should fit on a 640-by-480 screen, and will definitely fit on an 800-by-600 screen.

Following `cssContent` are the `cssContentLeft` and `cssContentRight` classes. These are the two halves of the content area: `cssContentLeft` for the tree view and `cssContentRight` for the note details. Each is specified to fill half of the horizontal area taken up by the `<div>` with the `cssContent` style, and its entire height.

The `float` attributes of the `cssContentLeft` and `cssContentRight` classes allow you to specify that an element should float to the left or right of its surrounding text. The `float` attribute was initially meant to allow images to float to either side of a paragraph of text, but it isn't limited to that role. What's interesting is how these two elements work when they are side by side. Any element with a `float` value of `left` or `right` is treated as a block-level element, meaning its `display` attribute will be ignored. You can also use this to have two paragraphs side by side on a page, for example.

Perhaps most important for us here is the fact that an element following a floating element will render in relation to the first floating element. This just means, as you can see in JSNotes, that two `<span>` elements with `float:left` and `float:right` correspondingly will be right next to each other. The two floating elements are pushed left or right until they reach the border of the containing element or margin of another block-level element (that is, each `<span>` gets pushed left or right until it meets the edges of the containing `<div>`). Note that I used a `<span>` to avoid the inherent line break with a `<div>`, which would break the layout. Note, too, that `<div>` and `<span>` elements must have a `width` set for them to render when the `float` attribute is used, as do these.

Moving on, `cssPadded` is a simple class applied to a `<div>` inside the overlays so that there is a little bit of padding around the edges.

The `cssOverlayTable` class is applied to the table used to lay out the overlays. This is done to ensure the browser honors the 100% height for the table we want, since some will not honor the `height` attribute of the `<table>` tag.

The `cssSliderBGMM` and `cssSliderBGHH` classes define the "ticker markers" for the minute and hour sliders, respectively, in the Add Note dialog box. You can see that the image itself is loaded as the background image, so this style will be applied to the `<div>` elements where the sliders will appear. They must be positioned relatively in order for YUI to work with them properly. In this case, we want them to render where they naturally would anyway, hence the `0px` values for both `left` and `top`.

The `cssSliderHandle` is shared by both of the slider knobs, or handles, and must be positioned absolutely in order for YUI to work with them. The `left` attribute will be altered, but the `top` will not, so it's safe to set the top to `8px` to offset it from the tick marks properly.

`cssTDNewNote` is the style class applied to the cells of the table used to lay out the Add Note dialog box. Because of some mucking around YUI does with the markup, setting `cellpadding` and/or `cellspacing` on the `<table>` tag gets ignored, but doing essentially the same thing through CSS works, hence this class.

The `cssOverlay` class is the style applied to the overlay containers. One interesting point here is the use of a browser-specific hack, which you'll recall I said is something to avoid. But, sometimes you just *have* to do it to get around some browser peculiarity, and such is the case here. The `left` attribute with the value `0px` will be used for all non-IE browsers, but for IE, the value `-2px` in the second `left` attribute beginning with the underscore character will override the previous value and be used. This is needed so that the overlay lines up properly on the display and we don't get a double-border glitch (remove that `_left` attribute and fire up IE to see what happens).

`cssTimeSpan`, last but not least, is the style applied to the `<span>` where the hour and minute values are shown next to the corresponding slider. This needs to take up a known amount of space, regardless of the current value, in order for everything to line up properly, and that's the sole purpose of this class.

With the CSS and markup in `index.htm` out of the way, let's move on to what we're all here for: JavaScript!

Writing Note.js

The `Note.js` file contains the `Note` class, which is nothing more than a Value Object (VO), or struct if you're more familiar with C, which contains the data describing a note the user entered. It just contains some private fields and getter/setter methods for each. Before we look at the code, take a gander at the UML diagram for the class in Figure 9-5.

I've chosen not to list the entire class here because it is, by and large, boring! It is little more than a series of getters and setters for the fields it contains. In fact, that's precisely what it is. And besides, in general, the comments themselves should tell you all you really need to know, so if you glance at it quickly on your own, you'll pretty well have the full picture in no time.

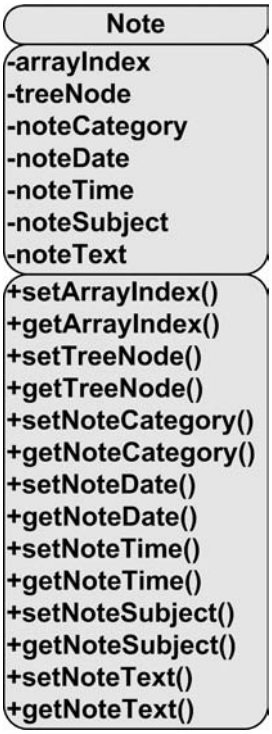


Figure 9-5. *UML diagram for the Notes class*

The `toString()` method overrides the default `toString()` that we get from the base `Object` class, and it outputs our note in a more meaningful way to aid in debugging.

Take note of the `arrayIndex` and `treeNode` fields and the comments accompanying them. These two fields are used when the user clicks a note in the tree view, but until that point, their values are meaningless. Once clicked, they are filled in and available for other actions that require them, such as deletions.

Other than that, it's an absolutely simple and straightforward piece of code. Now, if you want to see something just a little bit juicier, it's coming right up.

Writing JSNotes.js

`JSNotes.js` contains the `JSNotes` class, which is the heart and soul of the application. It's where all the functionality lives, and it's a fairly sizable piece of code. But as you'll see, a lot of it is pretty simple, and much of it is arguably boilerplate; the actual functionality doesn't require as much as you might think.

Before we get to that though, let's take a high-level look by examining the UML diagram in Figure 9-6.

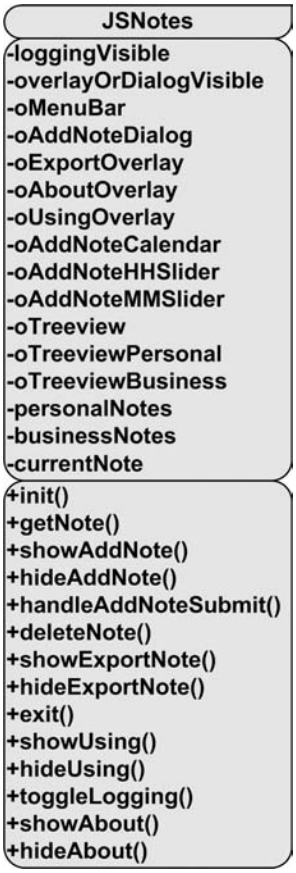


Figure 9-6. *UML diagram for the JSNotes class*

Listing 9-2 is the entire code listing, which is fairly lengthy. It checks in at a hair under 600 lines, and while a lot of it is comments and whitespace, there's plenty of meat on them bones!

Listing 9-2. *The JSNotes Class*

```
/**
 * The JSNotes class is the main class constituting the application.
 */
function JSNotes() {


  /**
   * Flag: Is the logging div currently visible?
   */
  var loggingVisible = false;
```

```
/**
 * Flag: Is an overlay or dialog currently visible?
 */
var overlayOrDialogVisible = false;


/**
 * Reference to the menubar object.
 */
var oMenuBar = null;


/**
 * Reference to the add notes dialog object.
 */
var oAddNoteDialog = null;


/**
 * Reference to the Export overlay object.
 */
var oExportOverlay = null;


/**
 * Reference to the About overlay object.
 */
var oAboutOverlay = null;


/**
 * Reference to the Using overlay object.
 */
var oUsingOverlay = null;


/**
 * Reference to the date calendar for adding a new note.
 */
var oAddNoteCalendar = null;


/**
 * Reference to the hour slider for adding a new note.
 */
var oAddNoteHHSlider = null;
```

```
/**
 * Reference to the minutes slider for adding a new note.
 */
var oAddNoteMMSlider = null;


/**
 * Reference to the treeview for listing categories/notes.
 */
var oTreeview = null;


/**
 * Reference to the treeview Personal Notes category node.
 */
var oTreeviewPersonal = null;


/**
 * Reference to the treeview Business Notes category node.
 */
var oTreeviewBusiness = null;


/**
 * The collection of Personal notes.
 */
var personalNotes = new Array();


/**
 * The collection of Business notes.
 */
var businessNotes = new Array();


/**
 * This is a reference to the Note object currently being viewed.
 */
var currentNote = null;


/**
 * Call on page load to initialize the application.
 */
this.init = function() {
```

```
// Start logging, show logging div if flag set to do so initially.
new YAHOO.widget.LogReader(YAHOO.util.Dom.get("divLog"));
YAHOO.log("init()");
if (loggingVisible) {
  YAHOO.util.Dom.get("divLog").style.display = "block";
}

// Start by centering the main DIV.
jscript.dom.layerCenterH(YAHOO.util.Dom.get("divMain"));
jscript.dom.layerCenterV(YAHOO.util.Dom.get("divMain"));

// Create menubar.
oMenuBar = new YAHOO.widget.MenuBar("divMainMenu");
oMenuBar.render();

// Create About overlay.
oAboutOverlay = new YAHOO.widget.Overlay("aboutOverlay",
  {
    context : [ "divContent", "tl", "tl" ],
    width : "500px", height : "456px", visible : false
  }
);
oAboutOverlay.setBody(YAHOO.util.Dom.get("divAbout"));
oAboutOverlay.render(document.body);

// Create Export overlay.
oExportOverlay = new YAHOO.widget.Overlay("exportOverlay",
  {
    context : [ "divContent", "tl", "tl" ],
    width : "500px", height : "456px", visible : false
  }
);
oExportOverlay.setBody(YAHOO.util.Dom.get("divExport"));
oExportOverlay.render(document.body);

// Create Using overlay.
oUsingOverlay = new YAHOO.widget.Overlay("usingOverlay",
  {
    context : [ "divContent", "tl", "tl" ],
    width : "500px", height : "456px", visible : false
  }
);
oUsingOverlay.setBody(YAHOO.util.Dom.get("divUsing"));
oUsingOverlay.render(document.body);
```

```
// Create Add Note dialog.
oAddNoteDialog = new YAHOO.widget.Dialog("divAddNote",
  {
    close : false,
    width : "320px",
    height : "460px",
    visible : false,
    constraintoviewport : true,
    buttons : [
      {
        text : "Submit",
        handler : jsNotes.handleAddNoteSubmit,
        isDefault : true
      },
      { text : "Cancel", handler : jsNotes.hideAddNote }
    ]
  }
);
oAddNoteDialog.render(document.body);

// Create Add Note calendar.
oAddNoteCalendar = new YAHOO.widget.Calendar("cal1", "addNoteCalendar");
oAddNoteCalendar.render();

// Create Add Note hour slider.
var bgHH = "divHHSliderBG";
var thumbHH = "divHHSliderThumb";
oAddNoteHHSlider = YAHOO.widget.Slider.getHorizSlider(
  bgHH, thumbHH, 0, 150, 13);
oAddNoteHHSlider.subscribe("change",
  function() {
    YAHOO.util.Dom.get("divHHValue").innerHTML =
      Math.round(oAddNoteHHSlider.getValue() / 13) + 1;
  }
);

// Create Add Note minutes slider.
var bgMM = "divMMSliderBG";
var thumbMM = "divMMSliderThumb";
oAddNoteMMSlider = YAHOO.widget.Slider.getHorizSlider(
  bgMM, thumbMM, 0, 178, 3);
oAddNoteMMSlider.subscribe("change",
  function() {
    var minute = Math.round(oAddNoteMMSlider.getValue() / 3);
    var s = "";
```

```
        if (minute < 10) {
          s += "0";
        }
        s += minute;
        YAHOO.util.Dom.get("divMMValue").innerHTML = s;
      }
    );

    // Create treeview for category/note listing.
    oTreeview = new YAHOO.widget.TreeView("divTreeview");
    var oRoot = oTreeview.getRoot();
    oTreeviewPersonal = new YAHOO.widget.TextNode("Personal",
      oRoot, false);
    oTreeviewBusiness = new YAHOO.widget.TextNode("Business",
      oRoot, false);
    oTreeview.subscribe("labelClick",
      function(node) {
        var noteSubject = node.data.subject;
        // Only do something when a note is clicked, not a category (only a
        // note would have a subject attribute).
        if (noteSubject) {
          var noteCategory = node.parent.data;
          currentNote = jsNotes.getNote(noteCategory, noteSubject);
          var noteDate = currentNote.getNoteDate();
          YAHOO.util.Dom.get("currentNoteDate").innerHTML =
            noteDate.getMonth() + "/" +
            noteDate.getDate() + "/" +
            noteDate.getFullYear();
          YAHOO.util.Dom.get("currentNoteTime").innerHTML =
            currentNote.getNoteTime();
          YAHOO.util.Dom.get("currentNoteSubject").innerHTML =
            currentNote.getNoteSubject();
          YAHOO.util.Dom.get("currentNoteText").innerHTML =
            currentNote.getNoteText();
        }
      }
    );
    oTreeview.draw();

    YAHOO.log("init() done");

  } // End init().
```

```
/**
 * Returns a Note object based on requested category and subject.
 *
 * @param inCategory The category the note belongs to.
 * @param inSubject  The subject of the note to retrieve.
 */
this.getNote = function(inCategory, inSubject) {

  var note = null;

  // Determine which array to search based on current category.
  var arrayToSearch = null;
  if (inCategory == "Personal") {
    arrayToSearch = personalNotes;
  } else {
    arrayToSearch = businessNotes;
  }

  // Search the array and find the match, if any, and return it.
  for (var i = 0; i < arrayToSearch.length; i++) {
    var n = arrayToSearch[i];
    if (n.getNoteSubject() == inSubject) {
      note = n;
      note.setArrayIndex(i);
      break;
    }
  }

  // Now find the note in the treeview for the note.
  note.setTreeNode(oTreeview.getNodeByProperty("subject",
    note.getNoteSubject()));

  // Not found.
  return note;

} // End getNote();


/**
 * Show the dialog for adding a note.
 */
this.showAddNote = function() {

  YAHOO.log("showAddNote()");
```

```
    if (overlayOrDialogVisible) { return; }
    oMenuBar.clearActiveItem();

    overlayOrDialogVisible = true;

    // Reset all form fields.
    var now = new Date();
    var hours = now.getHours();
    var minutes = now.getMinutes();
    YAHOO.util.Dom.get("frmNewNote").reset();
    oAddNoteCalendar.clear();
    oAddNoteCalendar.select(now);
    oAddNoteCalendar.render();
    oAddNoteHHSlider.setValue((hours * 13) - 13, true, true);
    oAddNoteMMSlider.setValue(minutes * 3, true, true);
    YAHOO.util.Dom.get("divHHValue").innerHTML = hours;
    if (minutes < 10) {
      minutes = "0" + minutes;
    }
    YAHOO.util.Dom.get("divMMValue").innerHTML = minutes;
    YAHOO.util.Dom.get("newNotePM").checked = true;

    // Show the dialog amd center it.
    oAddNoteDialog.center();
    oAddNoteDialog.show();

    YAHOO.log("showAddNote() done");

  } // End showAddNote().


  /**
   * Hide the dialog for adding a note.
   */
  this.hideAddNote = function() {

    YAHOO.log("hideAddNote()");

    oAddNoteDialog.hide();
    overlayOrDialogVisible = false;

    YAHOO.log("hideAddNote() done");

  } // End hideAddNote().
```

```
/**
 * Handle submit of the add new note form.
 */
this.handleAddNoteSubmit = function() {

  YAHOO.log("handleAddNoteSubmit()");

  // Get entered values.
  var noteCategory = YAHOO.util.Dom.get("newNoteCategorySelect").value;
  var noteDate = oAddNoteCalendar.getSelectedDates()[0];
  var noteHour = YAHOO.util.Dom.get("divHHValue").innerHTML;
  var noteMinute = YAHOO.util.Dom.get("divMMValue").innerHTML;
  var noteMeridian = null;
  if (YAHOO.util.Dom.get("newNoteAM").checked) {
    noteMeridian = "am";
  } else {
    noteMeridian = "pm";
  }
  var noteSubject = YAHOO.util.Dom.get("newNoteSubject").value;
  var noteText = YAHOO.util.Dom.get("newNoteText").value;

  // Now some simple validations.
  if (noteSubject == "") {
    alert("Please enter a subject for this note");
    YAHOO.util.Dom.get("newNoteSubject").focus();
    return false;
  }
  if (noteText == "") {
    alert("Please enter some text for this note");
    YAHOO.util.Dom.get("newNoteText").focus();
    return false;
  }

  // Instantiate a Note object and populate it.
  var note = new Note();
  note.setNoteCategory(noteCategory);
  note.setNoteDate(noteDate);
  note.setNoteTime(noteHour + ":" + noteMinute + noteMeridian);
  note.setNoteSubject(noteSubject);
  note.setNoteText(noteText);

  // Add the note to the appropriate treeview category and storage array.
  if (noteCategory == "Personal") {
    personalNotes.push(note);
    new YAHOO.widget.TextNode({label:noteSubject,subject:noteSubject},
      oTreeviewPersonal, false);
```

```
    } else {
      businessNotes.push(note);
      new YAHOO.widget.TextNode({label:noteSubject,subject:noteSubject},
        oTreeviewBusiness, false);
    }

    // Redraw treeview so it'll show up.
    oTreeview.draw();

    // Hide dialog and we're done!
    jsNotes.hideAddNote();
    YAHOO.log("handleAddNoteSubmit() done");
    return true;

  } // End handleAddNoteSubmit().


  /**
   * Delete the note currently being viewed.
   */
  this.deleteNote = function() {

    YAHOO.log("deleteNote()");

    if (overlayOrDialogVisible) { return; }
    oMenuBar.clearActiveItem();

    if (currentNote &&
      confirm("Are you sure you want to delete the current note?")) {
      // Delete from storage array.
      if (currentNote.getNoteCategory() == "Personal") {
        personalNotes.splice(currentNote.getArrayIndex(), 1);
      } else {
        businessNotes.splice(currentNote.getArrayIndex(), 1);
      }
      // Delete from treeview and redraw.
      oTreeview.removeNode(currentNote.getTreeNode());
      oTreeview.draw();
      // Clear display fields.
      YAHOO.util.Dom.get("currentNoteDate").innerHTML = "";
      YAHOO.util.Dom.get("currentNoteTime").innerHTML = "";
      YAHOO.util.Dom.get("currentNoteSubject").innerHTML = "";
      YAHOO.util.Dom.get("currentNoteText").innerHTML = "";
      // Finally, no more current note.
      currentNote = null;
    }
```

```
  YAHOO.log("deleteNote() done");

} // End deleteNote().


/**
 * Show the overlay for exporting the current note.
 */
this.showExportNote = function() {

  YAHOO.log("showExportNote()");

  if (overlayOrDialogVisible) { return; }
  oMenuBar.clearActiveItem();

  if (currentNote) {
    var s = "";
    var noteDate = currentNote.getNoteDate();
    s += "Category: " + currentNote.getNoteCategory() + "\n";
    s += "Date: " + noteDate.getMonth() + "/" +
      noteDate.getDate() + "/" +
      noteDate.getFullYear() + "\n";
    s += "Time: " + currentNote.getNoteTime() + "\n";
    s += "Subject: " + currentNote.getNoteSubject() + "\n";
    s += "Note: " + currentNote.getNoteText();
    YAHOO.util.Dom.get("taExport").value = s;
    YAHOO.util.Dom.get("taExport").select();
    overlayOrDialogVisible = true;
    oExportOverlay.show();
  }

  YAHOO.log("showExportNote() done");

} // End showExportNote().


/**
 * Hide the Export Note overlay.
 */
this.hideExportNote = function() {

  YAHOO.log("hideExportNote()");
  oMenuBar.clearActiveItem();

  oExportOverlay.hide();
  overlayOrDialogVisible = false;
```

```
    YAHOO.log("hideExportNote() done");

  } // End hideExportNote().


  /**
   * Exit the application
   */
  this.exit = function() {

    YAHOO.log("exit()");
    if (overlayOrDialogVisible) { return; }

    if (confirm(
      "All notes will be lost!  Are you sure you want to exit?")) {
      window.close();
    }

  } // End exit().


  /**
   * Toggle the logging div on and off.
   */
  this.toggleLogging = function() {

    YAHOO.log("toggleLogging()");
    if (overlayOrDialogVisible) { return; }

    oMenuBar.clearActiveItem();
    if (loggingVisible) {
      YAHOO.util.Dom.get("divLog").style.display = "none";
      loggingVisible = false;
    } else {
      YAHOO.util.Dom.get("divLog").style.display = "block";
      loggingVisible = true;
    }

    YAHOO.log("toggleLogging() done");

  } // End toggleLogging().


  /**
   * Show the Using (help) overlay.
   */
  this.showUsing = function() {
```

```
  YAHOO.log("showUsing()");
  if (overlayOrDialogVisible) { return; }
  oMenuBar.clearActiveItem();

  overlayOrDialogVisible = true;
  oUsingOverlay.show();

  YAHOO.log("showUsing() done");

} // End showUsing().


/**
 * Hide the Using (help) overlay.
 */
this.hideUsing = function() {

  YAHOO.log("showUsing()");
  oMenuBar.clearActiveItem();

  oUsingOverlay.hide();
  overlayOrDialogVisible = false;

  YAHOO.log("showUsing() done");

} // End hideUsing().


/**
 * Show the About overlay.
 */
this.showAbout = function() {

  YAHOO.log("showAbout()");
  if (overlayOrDialogVisible) { return; }

  oMenuBar.clearActiveItem();
  overlayOrDialogVisible = true;
  oAboutOverlay.show();

  YAHOO.log("showAbout() done");

} // End showAbout().
```

```
  /**
   * Hide the About overlay.
   */
  this.hideAbout = function() {

    YAHOO.log("hideAbout()");

    oAboutOverlay.hide();
    overlayOrDialogVisible = false;

    YAHOO.log("hideAbout() done");

  } // End hideAbout().


} // End JSNotes class.
```

As you can see, we have a number of data members and a batch of methods. Let's look at the data fields first:

- `loggingVisible`: A Boolean that determines whether the logging console is currently visible. It starts out invisible, until and unless the user turns it on via the Toggle Logging option under the Help menu.
- `overlayOrDialogVisible`: When any of the overlays or the Add Note dialog box are visible, the menus should not do anything. This flag determines when they shouldn't do anything (when this field is `true`).
- `oMenuBar`: A reference to the menu bar object as created by YUI. Later on, when a menu item is clicked, we need to clear the selected item, and to do that, we need a reference to the menu bar. It's better to cache the reference than incur the overhead of getting it each time.
- `oAddNoteDialog`: A reference to the Add Note dialog box as created by YUI. We'll again need this in various places, so keeping the reference is a good idea. In fact, the next few fields all serve the same purpose, which is to avoid object lookup overhead, so I'll skip repeating that fact from here on out!
- `oExportOverlay`: A reference to the Export Note overlay as created by YUI. I'm also going to stop saying " . . . as created by YUI" from now on, if you don't mind! It, too, applies to the next few items.
- `oAbourOverlay`: A reference to the About JSNotes overlay (are you starting to sense a pattern here?).
- `oUsingOverlay`: Yep, you guessed it—a reference to the Using JSNotes overlay.
- `oAddNoteCalendar`: Ah, something slightly different. Still a reference, but this time to the calendar in the Add Note dialog box.

- `oAddNoteMMSlider`: Reference. Minutes slider. Check.
- `oTreeView`: Oh tree view, oh tree view, where for art thou, tree view? Sorry, couldn't resist. Yes, another object reference, this time to the tree view listing the notes on the left side of the display.
- `oTreeviewPersonal`: A reference to the personal notes node object in the tree view.
- `oTreeviewBusiness`: I'll give you just one guess what this is! Now we're finished with the object references.
- `personalNotes`: An array containing all the `Note` objects categorized as personal notes.
- `businessNotes`: An array of all the `Note` objects in the business category.
- `currentNote`: Last but not least, a reference to the `Note` object the user is currently viewing.

OK, wise guy mode disengaged! Now that we've gotten through all the fields, let's see how they are used in the code, starting with the `init()` method.

The init() Method

The `init()` method is easily the lengthiest (about one-quarter of the total code size) and the most complex method in `JSNotes`. Even so, it's not all that monstrous! It is called on page load of `index.htm`, as you saw earlier. Its primary task is to construct the UI using YUI components.

Creating the Logging Console

To begin with, `init()` creates the logging console with this code:

```
// Start logging, show logging div if flag set to do so initially.
new YAHOO.widget.LogReader(YAHOO.util.Dom.get("divLog"));
YAHOO.log("init()");
if (loggingVisible) {
  YAHOO.util.Dom.get("divLog").style.display = "block";
}
```

All we need to do is instantiate `YAHOO.widget.LogReader`, passing it a reference to the DOM object that will house it. We get that reference using `YAHOO.util.Dom.get()`, which is basically a wrapper around `document.getElementById()`. We then quickly write out a message to the log with the line `YAHOO.log("init()");`, just to show where we are.

Recall from our look at `index.htm` that the logging `<div>`, `divLog`, is hidden to begin with. That's the reason for the `if` check that follows. If you were to change the default value of `loggingVisible` to `true`, then the console would be shown at this point.

When the logging console is visible, the display looks like what you see in Figure 9-7.

After that, the main `<div>` that houses what constitutes the display of JSNotes needs to be centered. To do that, we use two functions from the DOM package described in Chapter 3:

```
jscript.dom.layerCenterH(YAHOO.util.Dom.get("divMain"));
jscript.dom.layerCenterV(YAHOO.util.Dom.get("divMain"));
```

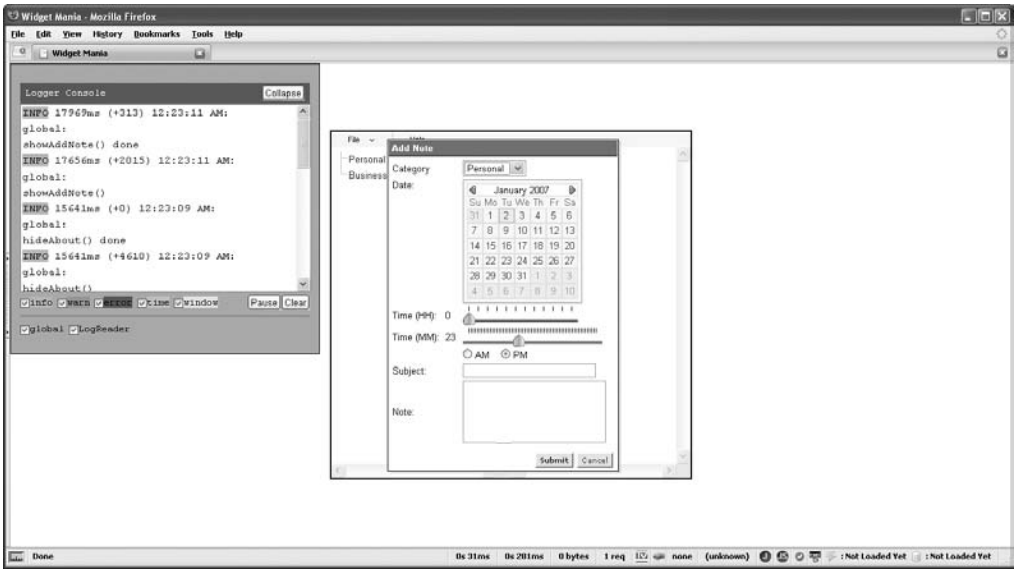


Figure 9-7. *The logging console*

Creating the Menu Bar

The next task is the creation of the menu bar, and it truly couldn't be easier:

```
oMenuBar = new YAHOO.widget.MenuBar("divMainMenu");
oMenuBar.render();
```

You already saw the markup contained in `divMainMenu`. YUI uses that to generate the menu you see on the screen. However, the first line simply creates the DOM snippet representing the menu in memory. To actually put it on the screen requires a call to its `render()` method, as you see in the second line. When both lines complete, the menu is visible and ready to be used.

Creating the Overlays

After that comes the creation of the three overlays: Export Note, Using JSNotes, and About JSNotes. They are virtually identical, so we'll pick just one of them, the Export Note overlay:

```
oExportOverlay = new YAHOO.widget.Overlay("exportOverlay",
  {
    context : [ "divContent", "tl", "tl" ],
    width : "500px", height : "456px", visible : false
  }
);
oExportOverlay.setBody(YAHOO.util.Dom.get("divExport"));
oExportOverlay.render(document.body);
```

The `YAHOO.widget.overlay` is the class we need to instantiate. The first argument to the constructor is the name the overlay will be known by—`exportOverlay` in this case. The next argument is an object containing a series of options. The first option you see, `context`, is used

to align the overlay to some other page element. In this case, we are aligning it to the `divContent` `<div>`, so that it will appear where that `<div>` does, directly below the menu. We also specify that the top-left corner of the overlay should align with the top-left corner of that `<div>`. You can align it other ways, such as the top-right corner of the overlay to the bottom-left corner of the `<div>` by passing `context : [ "divContent", "tr", "bl" ]`.

The `width` and `height` options should be self-explanatory. They are literally the width and height of the overlay. The `visible` option too is self-evident. When `false`, the overlay is initially not shown; when `true`, it is shown.

The next step is to fill in the content of the overlay. We do that by calling `setBody()` and handing it a reference to some existing element—in this case, the `divExport` `<div>` you saw in `index.htm`. Lastly, as with the menu, we need to render the overlay. We pass the `render()` method the object under which the overlay will be nested in the DOM, which is simply the document's `body` in this case.

The Export Note overlay is shown in Figure 9-8. The other two overlays look similar (see Figure 9-3 for a screenshot of the Using JSNotes overlay).

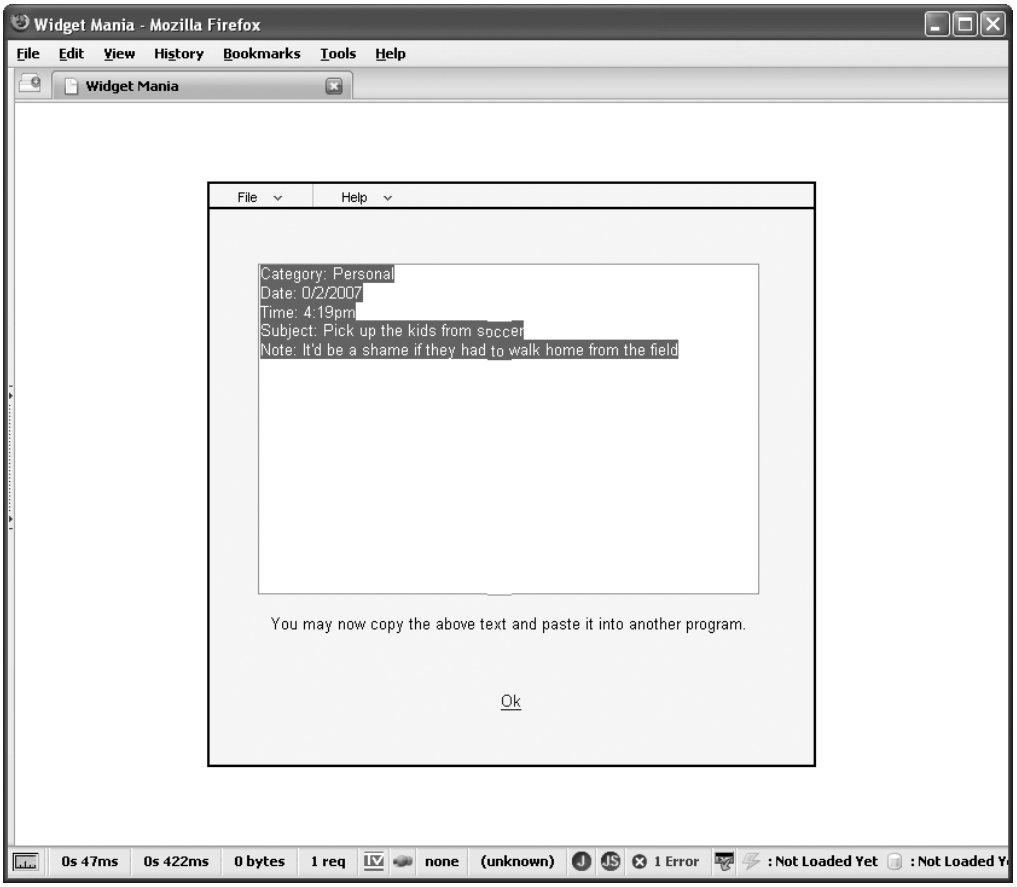


Figure 9-8. *Exporting a note from JSNotes*

Creating the Add Note Dialog Box

Next up is the creation of the Add Note dialog box. This is different from an overlay, but not a whole lot. A dialog box is generally meant to house a form to be filled out and submitted to the server. In our case, however, we won't be submitting it, but I'm getting ahead of myself. Let's see how the dialog box is created first:

```
oAddNoteDialog = new YAHOO.widget.Dialog("divAddNote",
  {
    close : false,
    width : "320px",
    height : "460px",
    visible : false,
    constraintoviewport : true,
    buttons : [
      {
        text : "Submit",
        handler : jsNotes.handleAddNoteSubmit,
        isDefault : true
      },
      { text : "Cancel", handler : jsNotes.hideAddNote }
    ]
  }
);
oAddNoteDialog.render(document.body);
```

This time, `YAHOO.widget.Dialog` is the class we need, and the first argument is the element containing the content for the dialog box—the `divAddNote <div>` here. The second argument is, as with the overlay, an object containing a series of options. Let's examine them one by one:

- `close`: This option indicates whether we want a close button, like the X button on a typical window. Here, we don't want that, so we set it to `false`.
- `width`, `height`, and `visible`: These options are exactly what you think they are, just as with the overlay widget.
- `constraintoviewport`: This option indicates whether the dialog box can be dragged off the page. We don't want that to be possible here, because we want it to stop at the edges, so `true` is the setting to use.
- `button`: This option is actually an array where each element describes a single button. Each element of the array is an object containing the options for each button.
- `text`: This option is what is displayed on the button—Submit and Cancel in this case.
- `handler`: This option defines which JavaScript function will be called when the button is clicked. Here, these are the `handlerAddNoteSubmit()` and `hideAddNote()` methods of the `JSNotes` class, respectively, pointed to by the reference `jsNotes`.
- `isDefault`: This option, on the Submit button, defines whether pressing Enter will activate that button, which is the case here.

Just as with the overlay widget, we need to tell the dialog box which element to render as a child of, and once again, the body of the document is the answer.

Next, we create the widgets that are in the Add Note dialog box. Note that there is no problem nesting widgets like this—YUI can handle it!

To begin with, we need to create the calendar used to select the note date, and it is accomplished via the following code:

```
oAddNoteCalendar = new YAHOO.widget.Calendar("cal1", "addNoteCalendar");
oAddNoteCalendar.render();
```

The first argument is the name the calendar will go by, and the second is the element that will be the parent of the calendar when it is rendered by the second line of code.

After the calendar, we need to render the two sliders, starting with the hour slider:

```
var bgHH = "divHHSliderBG";
var thumbHH = "divHHSliderThumb";
oAddNoteHHSlider = YAHOO.widget.Slider.getHorizSlider(
  bgHH, thumbHH, 0, 150, 13);
oAddNoteHHSlider.subscribe("change",
  function() {
    YAHOO.util.Dom.get("divHHValue").innerHTML =
      Math.round(oAddNoteHHSlider.getValue() / 13) + 1;
  }
);
YAHOO.util.Event.on(bgHH, "keydown",
  function(e) {
    console.log("keydown (HH)");
  }
);
YAHOO.util.Event.on(bgHH, "keypress", YAHOO.util.Event.preventDefault);
```

Well, there certainly is a little more going on here! First, we set up some variables we'll need. We start with bgHH, which names the containing the background image. Next is the name of the element containing the slider handle, divHHSliderThumb, stored in the variable thumbHH.

Once that's done, we instantiate a YAHOO.widget.Slider object, but we do so using the getHorizSlider() method, which takes as arguments the names of the background element and handle element, as well as three numbers. The first number is how many pixels to the left the handle can be moved, the second is how many to the right (which basically defines a maximum and minimum number of pixels the handle can move), and the third is the number of pixels to move for each tick mark.

It's a little tricky to set up the slider because you need to create a background image suitable for the range you want to allow. For instance, for the hours, we have 12 tick marks, one for each hour. The tick marks are 1 pixel wide, and separated by 12 pixels, so each tick movement is 13 pixels (the third number in the getHorizSlider() method). Multiplying 12 by 13 gives us 156, but we actually use 150 as the second number in getHorizSlider(). That's because the tick marks are essentially zero-based in terms of their positions; that is, the first tick mark representing hour 1 is at pixel location 0 (well, the tick mark is actually a few pixels from the edge, but for the handle to line up, the handle must be at pixel position 0). Therefore, we never want

the handle to make it to that 156 pixels position. So, we use the value of 150, which essentially constrains it to the last tick mark.

Perhaps the best way to understand this is to take the background image and the handle image and bring them into your favorite paint program (Photoshop, Paint Shop Pro, GIMP, or whatever you prefer). Enlarge the canvas of the background image so you have some room to play, then copy the handle image into it as a floating layer so you can move it around. Line it up with the first tick mark and note the X location. It should be zero. Now move it to the next tick mark. Note that it moved 13 pixels. Continue on until the last tick mark, where you should see the location is less than 150. If you were to move it to the next increment of 13, 156, it would be beyond the background tick marks, proving you need a value less than that to constrain it.

As soon as your brain stops hurting (and mine, too!), we can look at how we hook up some events to the slider. First, we want to update the hour value when the slider changes (is dragged). We do this by calling the `subscribe()` method on the slider. The first method is the event we want to subscribe to—`change` in this case. The second argument is the function to execute when the event fires, and here we've used an in-line function. When the value changes, we first need to get the new value of the slider. The value is basically the pixel location, which isn't by itself of any real use to us. So, we first need to divide that by 13, which gives us a value from 0 to 11. So, add one, and you now have the hour value corresponding to the pixel location. We get a reference to the `<div>` where the hour number is displayed, and we set its `innerHTML` attribute to that calculated value. The effect is the hour number changing as we drag the slider—sweet!

Creating the minutes slider is the same, except that since there are more tick marks, the upper constraint value is different—178 in this case. And the tick marks are closer together, so the increment value now is 3. Also, some minor differences are present in the handler for the `change` event:

```
oAddNoteMMSlider.subscribe("change",
  function() {
    var minute = Math.round(oAddNoteMMSlider.getValue() / 3);
    var s = "";
    if (minute < 10) {
      s += "0";
    }
    s += minute;
    YAHOO.util.Dom.get("divMMValue").innerHTML = s;
  }
);
```

This is very similar to the hour slider, but here the divisor in the calculation is 3, since that's the increment value. Also, there's no need to add one at the end, since the range here is zero-based (0 to 59 for minutes). Lastly, when the value is less than 10—that is, only a single digit—it should be displayed with a leading zero, so we see some logic to deal with that. Other than those differences, the minutes slider is more or less the same as the hour slider conceptually.

Creating the Tree View

The last piece of the `init()` method is the creation of the tree view that will list the notes the user creates. The code that accomplishes this is as follows:

```
oTreeview = new YAHOO.widget.TreeView("divTreeview");
var oRoot = oTreeview.getRoot();
oTreeviewPersonal = new YAHOO.widget.TextNode("Personal",
  oRoot, false);
oTreeviewBusiness = new YAHOO.widget.TextNode("Business",
  oRoot, false);
oTreeview.subscribe("labelClick",
  function(node) {
    var noteSubject = node.data.subject;
    // Only do something when a note is clicked, not a category (only a
    // note would have a subject attribute).
    if (noteSubject) {
      var noteCategory = node.parent.data;
      currentNote = jsNotes.getNote(noteCategory, noteSubject);
      var noteDate = currentNote.getNoteDate();
      YAHOO.util.Dom.get("currentNoteDate").innerHTML =
        noteDate.getMonth() + "/" +
        noteDate.getDate() + "/" +
        noteDate.getFullYear();
      YAHOO.util.Dom.get("currentNoteTime").innerHTML =
        currentNote.getNoteTime();
      YAHOO.util.Dom.get("currentNoteSubject").innerHTML =
        currentNote.getNoteSubject();
      YAHOO.util.Dom.get("currentNoteText").innerHTML =
        currentNote.getNoteText();
    }
  }
);
oTreeview.draw();
```

Well, there's quite a bit here, so let's break it down, shall we?

First, we have a typical widget instantiation line. We feed it the `<div>` that is to host the tree view. After that, we get a reference to the root node of the tree, which is created automatically when the widget is instantiated. Once we have that, we append two nodes to it: one for personal notes and one for business notes. After that, we have another event subscription line, requesting that we be notified when any label is clicked, which is the same as saying when any node is clicked.

YUI will take care of expanding and collapsing any node that has children, which will be only the two nodes we just created. However, we need to take care of what happens beyond that. In this case, that means showing the detail of a note that is clicked.

So, when we request the event subscription, we pass it a reference to a function to act as the callback for the event—in this case, an in-line anonymous function. The first thing done in this callback is to get the subject of the node. This will become clearer when you see how notes are added to the tree. For now, just be aware that when you add a node to the tree, you can attach to it arbitrary pieces of information, in addition to its text label. Here, we will be adding a `subject` attribute to it.

If you've played with the application, you will probably realize that the label of the note *is* the subject, so you may be asking, "Isn't it redundant to add the subject as well?" The answer is no, and the reason is the next line of code. You see, when *any* node is clicked, including our two

category nodes, this callback will be called. However, we need to do something only if the clicked node was *not* one of the category nodes. How do we determine that? Well, we could examine the label directly, but if we ever changed or added to the categories, that would mean one more thing we would need to change. Instead, for nodes representing actual notes only, the `subject` attribute is attached. Therefore, we can simply check if that attribute is defined for the clicked node. If it isn't, it was one of the category nodes and the callback should do nothing.

Otherwise, we find ourselves inside the `if` block. In that situation, we need to get the category of the note. To do this, we use the `node.parent` reference, `node` being a reference to the clicked node. Through that, we can get the label of the node, which is the `data` attribute by default.

Once we have the category and the subject, we can call the `getNote()` function, which you'll see in a bit. This will return a reference to the `Note` object for the clicked note. From there, it's a simple matter of populating the four data display elements on the page from the fields in the `Note` object. We again use the `YAHOO.util.Dom.get()` function to get references to those fields. The date is the only interesting field here. We have a JavaScript `Date` object, but we want to display the note's date in *MM*/*DD*/*YYYY* format, so we need to get the individual components of the date and construct the string ourselves.

Whew, that was a fair amount of code! The rest of this class will seem pretty paltry by comparison.

The getNote() Method

Next is the `getNote()` that I just mentioned:

```
this.getNote = function(inCategory, inSubject) {

  var note = null;

  // Determine which array to search based on current category.
  var arrayToSearch = null;
  if (inCategory == "Personal") {
    arrayToSearch = personalNotes;
  } else {
    arrayToSearch = businessNotes;
  }

  // Search the array and find the match, if any, and return it.
  for (var i = 0; i < arrayToSearch.length; i++) {
    var n = arrayToSearch[i];
    if (n.getNoteSubject() == inSubject) {
      note = n;
      note.setArrayIndex(i);
      break;
    }
  }
```

```
    // Now find the note in the treeview for the note.
    note.setTreeNode(oTreeview.getNodeByProperty("subject",
      note.getNoteSubject()));

    // Not found.
    return note;

  } // End getNote();
```

First, we need to determine which array we're searching, `businessNotes` or `personalNotes`, based on the category passed in. After that, it's a simple matter of checking each `Note` object in the array and seeing if its subject matches the subject passed in.

One last item of work remains, and that's to get a reference to the node in the tree view corresponding to the note, and set that reference on the `Note` object. (Remember the comments about the `treeNode` field? If not, go back and look, because they are now very relevant.) The tree view widget provides a couple of different ways to get a reference to a node, one of which is the `getNodeByProperty()` method. Recall that I mentioned that when a note is added, we add a custom property named `subject` to it. Well, that's exactly the property we specify to search here! After that, the reference to the note is returned, or null is returned if no match was found, and that's that.

The showAddNote() Method

The next method to check out is what is called when the user clicks the Add Note menu item, and it is responsible for showing the dialog box and setting up for user data entry:

```
  this.showAddNote = function() {

    YAHOO.log("showAddNote()");

    if (overlayOrDialogVisible) { return; }
    oMenuBar.clearActiveItem();

    overlayOrDialogVisible = true;

    // Reset all form fields.
    var now = new Date();
    var hours = now.getHours();
    var minutes = now.getMinutes();
    YAHOO.util.Dom.get("frmNewNote").reset();
    oAddNoteCalendar.clear();
    oAddNoteCalendar.select(now);
    oAddNoteCalendar.render();
    oAddNoteHHSlider.setValue((hours * 13) - 13, true, true);
    oAddNoteMMSlider.setValue(minutes * 3, true, true);
    YAHOO.util.Dom.get("divHHValue").innerHTML = hours;
```

```
    if (minutes < 10) {
      minutes = "0" + minutes;
    }
    YAHOO.util.Dom.get("divMMValue").innerHTML = minutes;
    YAHOO.util.Dom.get("newNotePM").checked = true;

    // Show the dialog and center it.
    oAddNoteDialog.center();
    oAddNoteDialog.show();

    YAHOO.log("showAddNote() done");

  } // End showAddNote().
```

As you can see, there's not a whole lot to it. Note the `YAHOO.log()` call as the first statement. This is the way you output a log message to the logging console. You'll see this in most of the methods in this class, so I won't mention it again.

You'll also see the next two statements in most methods. Recall that I previously mentioned that when a dialog box or overlay is showing, the menu shouldn't do anything. This is accomplished by a check of the `overlayOrDialogVisible` variable. When it's `true`, we simply return immediately. However, if it's not `true`, the first thing we need to do is dismiss the submenu, which is accomplished by a call to the `clearActiveItem()` method of the menu bar widget. These three lines of code won't be mentioned again, so don't be surprised when they pop up all over the place!

Once that's done, the `overlayOrDialogVisible` variable is immediately set so that the menu will not do anything until further notice. After that, it's time to clear the form fields in the Add Note dialog box. Before even that though, we get a reference to the current time, since we'll need that later. To begin the actual clearing, we start with a call to the `reset()` method of the form, which takes care of the category, subject, and note text fields.

Next, we deal with the calendar, first by clearing the current selection via the call to `clear()` on it, and then by calling `select()`, passing it the `Date` object we instantiated earlier, and then calling `render()` on it. This has the effect of setting the calendar to the current month and year and selecting the current date, which is the reasonable default value.

Then we handle the hour and minutes sliders. To do this, we call the `setValue()` method on them. Recall that the values of the sliders are actually pixel values. To translate the current time to pixel values, we need to multiple the hours by 13 (recall that's the pixel increment per tick mark for the hours) and 3 for the minutes (the tick mark increment for the minutes). We also set the textual representation of the hour and minutes by updating the `innerHTML` of the two `<div>` elements, making sure to append a leading zero to minute values less than 10.

Finally, we have only to center the dialog box by calling its `center()` method, and showing it via the call to its `show()` method. And now we have a pristine dialog box, ready for the user to create a new note! We add a quick log message to indicate the method completed, and it's a wrap!

The hideAddNote() Method

After that, we find the `hideAddNote()` method, which as its name implies, hides the dialog box after the user clicks Submit. It's a trivial piece of code:

```
this.hideAddNote = function() {

  YAHOO.log("hideAddNote()");

  oAddNoteDialog.hide();
  overlayOrDialogVisible = false;

  YAHOO.log("hideAddNote() done");

} // End hideAddNote().
```

I'm going to go out on a limb here and assume you don't need me to explain how it works! Besides, much bigger things loom just over the horizon.

The handleAddNoteSubmit() Method

The next method we find is `handleAddNoteSubmit()`. As you will recall, when the Add Note dialog box was created in `init()`, we passed a reference to this function into the constructor indicating it should be the callback when the Submit button is clicked. Its job is simply to save the note the user entered, or alternatively reject it if it doesn't pass some simple validation checks. Here is the code for this all-important method:

```
this.handleAddNoteSubmit = function() {

  YAHOO.log("handleAddNoteSubmit()");

  // Get entered values.
  var noteCategory = YAHOO.util.Dom.get("newNoteCategorySelect").value;
  var noteDate = oAddNoteCalendar.getSelectedDates()[0];
  var noteHour = YAHOO.util.Dom.get("divHHValue").innerHTML;
  var noteMinute = YAHOO.util.Dom.get("divMMValue").innerHTML;
  var noteMeridian = null;
  if (YAHOO.util.Dom.get("newNoteAM").checked) {
    noteMeridian = "am";
  } else {
    noteMeridian = "pm";
  }
  var noteSubject = YAHOO.util.Dom.get("newNoteSubject").value;
  var noteText = YAHOO.util.Dom.get("newNoteText").value;

  // Now some simple validations.
  if (noteSubject == "") {
    alert("Please enter a subject for this note");
    YAHOO.util.Dom.get("newNoteSubject").focus();
    return false;
  }
```

```
  if (noteText == "") {
    alert("Please enter some text for this note");
    YAHOO.util.Dom.get("newNoteText").focus();
    return false;
  }

  // Instantiate a Note object and populate it.
  var note = new Note();
  note.setNoteCategory(noteCategory);
  note.setNoteDate(noteDate);
  note.setNoteTime(noteHour + ":" + noteMinute + noteMeridian);
  note.setNoteSubject(noteSubject);
  note.setNoteText(noteText);

  // Add the note to the appropriate treeview category and storage array.
  if (noteCategory == "Personal") {
    personalNotes.push(note);
    new YAHOO.widget.TextNode({label:noteSubject,subject:noteSubject},
      oTreeviewPersonal, false);
  } else {
    businessNotes.push(note);
    new YAHOO.widget.TextNode({label:noteSubject,subject:noteSubject},
      oTreeviewBusiness, false);
  }

  // Redraw treeview so it'll show up.
  oTreeview.draw();

  // Hide dialog and we're done!
  jsNotes.hideAddNote();
  YAHOO.log("handleAddNoteSubmit() done");
  return true;

} // End handleAddNoteSubmit().
```

First, we need to get the values entered by the user:

- For the category, subject, and note fields, we just grab their `value` attributes.
- For the date, we need to call the calendar's `getSelectedDates()` method. This method returns an array of `Date` objects, but in this case, we only care about the first element in the array, hence the `[0]` subscript. Note that, by default, the calendar will allow only a single date to be selected, but `getSelectedDates()` still returns an array, so it doesn't change the way we get the value.

- For the time's hour and minutes components, we get the `innerHTML` value of the `<div>` displaying the value (no sense messing with the pixel-to-real-value conversion we've seen before, since we already have the final value in the `<div>`s). For the time meridian, we see if the AM radio button is checked, and if so, then the meridian is AM; otherwise, it is PM.

Once all the values have been captured, we do some simple validations. These validations amount to nothing more than ensuring the user entered both a subject and some note text, and if not, we set the focus to the offending field and return immediately.

Once validations have been passed, it's time to save the note. The first step is to instantiate a `Note` object and populate each of the fields. The only interesting thing here is the time, which, as you can see, is stored in a format ready to be displayed in the details section when the user clicks it. Otherwise, this is just a series of setter calls.

The next step is to add the note to both the correct storage array and the tree view. First, a logic branch occurs based on the category of the note. Once we know whether it's a personal note or a business note, we `push()` it onto the appropriate array. To add it to the tree view requires that we instantiate a new TextNode widget, just as we did when creating the nodes for the categories. This takes the following three arguments:

- The first argument to the constructor of the widget is an object containing data elements. The `label` attribute is what you see in the tree view, and the `subject` is obviously the subject of the note. This completes the puzzle we began seeing earlier. Remember how we got the subject of the note when the user clicked on it so that we could look it up in the arrays for display? Well, here is where that value gets set on the node.

- The second argument is the node in the tree that this new node will be a child of: either the personal notes node or the business notes node.

- The last argument determines whether the node is expanded (`true`) or not (`false`) initially. There really is no meaning to expanding a note node though, since there will never be any children. Just the same, passing `false` is the safer bet.

Only a few relatively minor steps remain in this method. First, somewhat important is to update the tree view. This is done by calling `draw()` on it, which results in it being repainted, including the new note. Finally, we hide the Add Note dialog box, write a log message, and return—and that's adding a new note.

The deleteNote() Method

The method that follows `handlerAddNoteSubmit()` is `deleteNote()`. There isn't much to deleting a note, as you can see:

```
this.deleteNote = function() {

  YAHOO.log("deleteNote()");

  if (overlayOrDialogVisible) { return; }
  oMenuBar.clearActiveItem();
```

```
    if (currentNote &&
      confirm("Are you sure you want to delete the current note?")) {
      // Delete from storage array.
      if (currentNote.getNoteCategory() == "Personal") {
        personalNotes.splice(currentNote.getArrayIndex(), 1);
      } else {
        businessNotes.splice(currentNote.getArrayIndex(), 1);
      }
      // Delete from treeview and redraw.
      oTreeview.removeNode(currentNote.getTreeNode());
      oTreeview.draw();
      // Clear display fields.
      YAHOO.util.Dom.get("currentNoteDate").innerHTML = "";
      YAHOO.util.Dom.get("currentNoteTime").innerHTML = "";
      YAHOO.util.Dom.get("currentNoteSubject").innerHTML = "";
      YAHOO.util.Dom.get("currentNoteText").innerHTML = "";
      // Finally, no more current note.
      currentNote = null;
    }

    YAHOO.log("deleteNote() done");

  } // End deleteNote().
```

The first step is to ensure a note is currently being displayed by seeing if the `currentNote` field is null. If no note is being displayed, we obviously don't need to do anything, so that effectively ends the method.

Assuming a note is displayed though, we first confirm the user wants to delete it (especially since there is no persistence per se in JSNotes, this is a nice thing to do!). Once it's confirmed, we determine which category the note belongs to, so that we know from which array to remove it. Once we know that, we use the standard `splice()` method on the array, which JavaScript provides to remove elements from the array. The first argument is the array index to begin deletion—in this case, it's the value stored in the `arrayIndex` field of the note. The second argument is how many elements to delete, which is just one in this case.

That takes care of deleting the note from the data array, but now we need to delete it from the tree view as well. To do that, we simply call the `removeNode()` method, passing it a reference to the node to remove, which you'll recall we stored in the `treeNode` field in the `Note` object when the note was clicked to display. So all that remains is to call `draw()` to refresh the tree view, and we're set.

Lastly, we are sure to clear the display fields, since what's there is obviously no longer valid, and set the `currentNote` field to null since there is no longer a note being displayed. Now the note is officially, completely, and utterly gone!

The showExportNote() Method

The next bit of functionality to check out is displaying the Export Note overlay. Here is the method that accomplishes that:

```
this.showExportNote = function() {

  YAHOO.log("showExportNote()");

  if (overlayOrDialogVisible) { return; }
  oMenuBar.clearActiveItem();

  if (currentNote) {
    var s = "";
    var noteDate = currentNote.getNoteDate();
    s += "Category: " + currentNote.getNoteCategory() + "\n";
    s += "Date: " + noteDate.getMonth() + "/" +
      noteDate.getDate() + "/" +
      noteDate.getFullYear() + "\n";
    s += "Time: " + currentNote.getNoteTime() + "\n";
    s += "Subject: " + currentNote.getNoteSubject() + "\n";
    s += "Note: " + currentNote.getNoteText();
    YAHOO.util.Dom.get("taExport").value = s;
    YAHOO.util.Dom.get("taExport").select();
    overlayOrDialogVisible = true;
    oExportOverlay.show();
  }

  YAHOO.log("showExportNote() done");

} // End showExportNote().
```

Assuming a note is currently being displayed, a string is constructed. This string more or less mimics the details display you see when a note is clicked, but does so in plain text. The string is then inserted into the `<textarea>` that is a part of the overlay, and the text is selected via a call to the `select()` method of the text area. Then the overlay is shown via the call to its `show()` method, and that's really all there is to it.

The hideExportNote() Method

Once the user clicks OK in the Export Notes overlay, it is dismissed via a call to `hideExportNote()`, shown here:

```
this.hideExportNote = function() {

  YAHOO.log("hideExportNote()");
  oMenuBar.clearActiveItem();

  oExportOverlay.hide();
  overlayOrDialogVisible = false;

  YAHOO.log("hideExportNote() done");

} // End hideExportNote().
```

This method has nothing more than a call to its `hide()` method, and setting `overlayOrDialogVisible` to `false` so that our menu works again. And that's it.

The exit() Method

We're nearing the end! The next method to look at is called when the Exit option is clicked on the File menu:

```
this.exit = function() {

  YAHOO.log("exit()");
  if (overlayOrDialogVisible) { return; }

  if (confirm(
    "All notes will be lost!  Are you sure you want to exit?")) {
    window.close();
  }

} // End exit().
```

There's nothing fancy here. Just a confirmation to be sure users want to leave, since all their notes will be gone at that point, and a call to the window object's `close()` method are all it takes.

The toggleLogging() Method

The `toggleLogging()` method is next, and here it is:

```
this.toggleLogging = function() {

  YAHOO.log("toggleLogging()");
  if (overlayOrDialogVisible) { return; }

  oMenuBar.clearActiveItem();
  if (loggingVisible) {
    YAHOO.util.Dom.get("divLog").style.display = "none";
    loggingVisible = false;
  } else {
    YAHOO.util.Dom.get("divLog").style.display = "block";
    loggingVisible = true;
  }

  YAHOO.log("toggleLogging() done");

} // End toggleLogging().
```

This boils down to nothing more than showing `divLog` if `loggingVisible` is `false`, and then reversing `loggingVisible` to `true`, or hiding `divLog` if `loggingVisible` if `true`, and then reversing `loggingVisible` to `false`.

The Rest

Only four methods remain, but they are really so simple that I don't see much point in showing them here. As I'm sure you can guess, `showUsing()`, `hideUsing()`, `showAbout()`, and `hdeAbout()` deal with showing and hiding the Using JSNotes and About JSNotes overlays. You've already seen the basic code in the `showExportNote()` method. The `show`*`XXX`*`()` methods just set `overlayOrDialogVisible` to `true` and call `show()` on the appropriate overlay object. For the `hide`*`XXX`*`()` methods, `overlayOrDialogVisible` is set to `false`, and the `hide()` method is called. There quite literally is nothing more to these methods!

And with that last little bit, we've reached the end of our journey! I hope you'll agree that with YUI in the mix, the volume of code is not very great, and it's pretty simple code to boot.

Suggested Exercises

When you look at that stack of sticky notes, not a whole lot of possible enhancements come to mind, aside from perhaps different colors and more writing surface. Similarly, with such a simple application as JSNotes, there's not a whole lot of really advanced functionality to add. Of course, I wouldn't leave you without any exercises, so here are a few that should extend your knowledge of YUI, and, of course, JavaScript in general:

- Add the ability to clean an entire category at once. Likewise, add the ability to export an entire category.
- Present the interface in a tabbed fashion. YUI provides a tabbed dialog widget, and it should be possible to see two tabs, Personal and Business, with just a straight list of notes on the left. I was actually going to do this in the example, but I decided it would be an excellent exercise to help familiarize you with YUI.
- For the tabbed interface, add a View menu with two options: Tabbed View and Tree View, so you can switch between the two tabs. I suggest creating two layers in `index.htm`: one that contains what you see now, and one that includes the tab view. That should make it very easy to switch between the two.
- Add some effects. YUI provides other effects that you can probably add without too much trouble. How about the Using and About overlays expanding into view perhaps?

Summary

In this chapter, we put together a handy little note-taking application. In the process, you were introduced to a top-notch JavaScript library, YUI. You saw how it makes creating usable user interfaces with custom widgets a breeze, and you also saw some of the utility functionality it provides. Examining the application revealed how, with just a relatively small amount of code, you can create a decent amount of functionality with the help of this excellent library.

CHAPTER 10

Shopping in Style: A Drag-and-Drop Shopping Cart

Most people who have shopped on the Internet are familiar with the shopping cart metaphor. You see a list of items for sale, you click some button, and some quantity of the item is added to your shopping cart. You then check out, and your order is processed based on the content of your cart. This is all well and good, but we can do a little better than that, can't we?

In this chapter, we will build a shopping cart that lets you drag items into it, rather than needing to click a button to add to your cart. We will use some special effects to make this look cooler than it might sound. At the same time though, we will respect the fact that some people may not have JavaScript available. For them, we will ensure our shopping cart degrades gracefully and still works, even under those "arcane" conditions.

We'll use a new library, MochiKit, for the drag-and-drop action. Also, you'll see a new technique that can come in very handy: a mock server, to handle your server needs without having to write actual server code.

So, break out your wallet and credits cards, and let's go shopping!

Shopping Cart Requirements and Goals

A shopping cart is a well-known paradigm used on most e-commerce web sites. Sites such as Amazon (`http://www.amazon.com`), Best Buy (`http://www.bestbuy.com`), Newegg (`http://www.newegg.com`), CD Now (`http://www.cdnow.com`), and TigerDirect (`http://www.tigerdirect.com`)—to name just a few—use a shopping cart to allow their users to purchase their products. It's conceptually not a difficult beast, but with a little added pizzazz, it can actually be more fun to use, and certainly a bit more Web 2.0-ish. Let's enumerate our goals for our Shopping Cart application:

- Users should be able to select an item, select a quantity of that item, and have it added to their cart. They should be able to do this by dragging items into the cart, or in a more typical manual fashion (part of graceful degradation, as you'll see shortly).
- Users should be able to view the contents of their cart at any time, including the current dollar total, and also be able to modify it from that view: add and remove items, as well as change quantities.

- Users should be able to check out; that is, complete their purchase. However, since this is a book focused on JavaScript and the client in general, we're not going to actually write that part.[1] Moreover, we're going to write a "mock" server on the client. It would be a simple exercise to replace this mock server with the real McCoy.
- The shopping cart should degrade gracefully; that is, work whether or not JavaScript is enabled. When it is enabled, the full drag-and-drop experience is available. When it is disabled, the cart still functions just fine, but in a more manual fashion, typical of most shopping carts. Because we're faking the server side though, we can't literally disable JavaScript, because we need it to do that fakery. Therefore, we'll provide a switch to turn JavaScript on and off in a fake way, so we can see how things react in either circumstance.
- We'll use a JavaScript library to help us with the more complex pieces, specifically the drag-and-drop functionality. That library is MochiKit.

With this application, we will have three primary concerns. First, we want the user to have the ability to drag items onto the cart, so we'll need to deal with the code to enable drag-and-drop. This isn't the most difficult thing to implement in a web application, but it involves a fair number of details, so we'll be using the MochiKit library to handle the complexity for us.

Second, we want the application to degrade gracefully, so it will still work when JavaScript is disabled. This book, being about JavaScript, hasn't focused on how this can be accomplished, so this will be a good chance to demonstrate how to do it. As you can imagine, a drag-and-drop shopping cart just isn't viable without JavaScript, so it truly will be a degraded experience, if one considers the drag-and-drop version to be the pinnacle. Still, we can continue to provide a perfectly usable shopping experience, even without JavaScript.

The third consideration is that we aren't going to mess with the server side of the equation here, yet we will need a server in the mix to do it right. How can we accomplish that? We can pull off this trick by utilizing a technique I like to call a *mock server*.[2]

Let's start with how we will pull off that graceful degradation.

Graceful Degradation, or Working in the Stone Age

These days, writing a web application without JavaScript is tantamount to trying to start a fire by rubbing two sticks together. There's no question it can be done that way, but why would you want to, when you can grab a BIC from Wal-Mart and do it with the flick of a finger?

In years gone by, JavaScript had a very poor reputation on a number of fronts: security, performance, annoyance (which is really a failing of those using it), and so on. In those days, people often would disable JavaScript entirely to make their browsing experience more enjoyable. Some people still do that today, even though it's far less common. Also, we should consider

1. You will be seeing some server-side code in the chapter on Ajax (Chapter 12), but that's because there isn't a particularly good way to fake it in a purely client-side manner. Here, we can do that with a mock server.
2. While I won't claim to have made up the term, I can honestly say I've never heard it referred to as such anywhere else, so, if you use it, send the royalty check to my publisher for forwarding along to me. Just kidding!

alternate browsing devices and limited devices like cell phone-based browsers, which often do not provide JavaScript capabilities. Therefore, making an application that can operate just as well (or nearly as well) in the absence of JavaScript as it does with it is a very good thing indeed.

How do we actually do it though? It can be boiled down to this: write the application to work without JavaScript, and then add the script that "enables" the features that can be present only when JavaScript is.

As a simple example, let's say we want to have a form like this:

```
<html>
  <head></head>
  <body>
    <form name="myForm" method="post" action="someAction.do">
      Your name: <input type="text" name="yourName" size="20">
      <br>
      <input type="submit" value="Submit">
    </form>
  </body>
</html>
```

Clearly, that will work just fine whether or not JavaScript is enabled. Now, let's think of a problem we might encounter: what if the user doesn't enter a name? Will the server we submit this to blow up and return some error? It very well may. So, we're going to have to check for that on the server. But for this simple check, why do we even need to involve the server? With JavaScript, we can do this instead:

```
<html>
  <head>
    <script>
      function validate(inForm) {
      if (inForm.yourName.value == "") {
        alert("Please enter your name");
      } else {
        inForm.submit();
      }
    </script>
  </head>
  <body>
    <form name="myForm" method="post" action="someAction.do">
      Your name: <input type="text" name="yourName" size="20">
      <br>
      <input type="button" value="Submit" onClick="validate(this.form);">
    </form>
  </body>
</html>
```

Now, when JavaScript is enabled, the server won't be involved to perform this simple check, cutting down on server utilization and network utilization. And this also makes for a better user experience, because there won't be even a little delay between clicking Submit and seeing the error message.

The trick now is to make it work in both cases of JavaScript enabled and disabled. First, we need to define what exactly *work* means in this context. When JavaScript is enabled, *work* obviously means do the check on the client, pop up the error message if required, and cancel the form submission. When JavaScript is disabled, *work* means the form should still submit to the server, but without the benefit of the client-side check. Now, you may argue that this example is flawed because if the button were a `type="submit"`, and the check were done `onSubmit` of the form instead, we would have what we want, and you would be correct. This is an example to illustrate a point, however, so bear with me a bit.

Aside from that solution, how else could we make this work in both cases? Take a look at this:

```
<html>
  <head>
    <script>
      function setup() {
        document.getElementById("btnSubmit").style.display = "none";
        document.getElementById("btnButton").style.display = "block";
      }
      function validate(inForm) {
      if (inForm.yourName.value == "") {
        alert("Please enter your name");
      } else {
        inForm.submit();
      }
    </script>
  </head>
  <body onLoad="setup();">
    <form name="myForm" method="post" action="someAction.do">
      Your name: <input type="text" name="yourName" size="20">
      <br>
      <input type="submit" value="Submit" id="btnSubmit">
      <input type="button" value="Submit" id="btnButton"
        onClick="validate(this.form);">
    </form>
  </body>
</html>
```

Here, we have the best of both worlds. When JavaScript is disabled, the only button that is visible is the Submit button. Yes, it's true we don't get the client-side check, but that's why this is degraded rather than broken; it will still work, just in a degraded way. When JavaScript is enabled, the Submit button it hidden in favor of the regular button, which includes the client-side check.

We'll be doing a very similar thing in this chapter's application. At some point before reading too much further, you should play with it a little bit. In `main.js`, you'll find a variable `javaScriptEnabled`. I'll discuss this later when we dissect the application, but for now, you can simply set this to `true` to see the application in JavaScript-enabled mode, and set it to `false` to simulate JavaScript being disabled. When you set this variable to `false`, notice that you have Description links below the items you can purchase. But when JavaScript is enabled, the links are not present. By default, these links are there, and they will be removed by JavaScript later to

give a more rich experience (hovering over the items gives you the descriptions that the links otherwise would).

The MochiKit Library

Now let's turn our attention to MochiKit (`http://www.mochikit.com`). Although I mentioned it in Chapter 2, MochiKit's slogan certainly deserves repeating:

MochiKit makes JavaScript suck less

Gold, I tell ya—pure comedy gold!

Seriously though, I'm not sure I would say that JavaScript "sucks" anyway (one would certainly hope not, based on the fact that I'm here writing this book!), but the sentiment is still valid. MochiKit definitely makes many things much more pleasant.

Take drag-and-drop, for instance. When you're developing a fat client—be it Java Swing, Windows, Linux, or whatever—drag-and-drop is pretty simple, usually requiring nothing more than setting some attributes and implementing some minor code. In a browser though, it's a fair bit more work:

- Track when the mouse button is pressed down, and see if it was pressed down on an element that is draggable (as determined by your own criteria, by the way).
- Track the mouse movement and move the draggable object accordingly.
- Recognize when the mouse button is lifted and determine which object the draggable object was on when the button was released. Was it an object where the dragged item can be dropped?
- If the drop target is valid for the dragged item, process that event. And deal with what happens to the draggable object—does it get cloned perhaps, or just return to its starting point (which you remembered to record, right?).
- Along the way, deal with the differences in browser event models, CSS differences, and so on.
- And how about the possibility of some effects, to make the whole deal look a bit cooler?

It's not that this can't be done; of course, it can. In fact, I've seen some rather elegant implementations[3] that don't make my head hurt all that much—at least as far as the dragging part goes. However, those implementations don't deal with the dropping part, other than literally ending the process of dragging. Determining if the object was dropped on something else is out of scope.

Since drag-and-drop isn't exactly a trivial exercise in the browser, it behooves us to find a good implementation that can save us that time, effort, and trouble. MochiKit provides just such a beast.

3. Check out `http://www.javascriptkit.com/howto/drag.shtml`, which is a pretty simple browser drag-and-drop implementation.

With MochiKit, making something draggable is as simple as this:

```
new MochiKit.DragAndDrop.Draggable("myObject", { revert : resetIt} );
```

With this line of code, the object on the page with the ID `myObject` is now able to be dragged all over the place! When the user releases the mouse button, the `resetIt()` function will be called. And in that function, we can do whatever we want. Can it get any easier?

"What about dropping?" you ask. That's just as drop-dead easy (pun intended):

```
new MochiKit.DragAndDrop.Droppable("dropHere", { ondrop : doOnDrop } );
```

We've now made it so that the object on the page with the ID `dropHere` can have other objects dropped onto it, and when that happens, the `doOnDrop()` function will be called. Once again, can you imagine it being any simpler?

MochiKit offers a slew of options to go along with this, such as the ability to have the dragged object return to its starting point, having a "ghost" of the object created so that you're dragging a clone, and so on. It also offers effects, such as having the dragged object fade out slightly when being dragged. And that option to have the object return to its starting point, well, it doesn't have to just jump back there; it can actually glide back gracefully!

MochiKit's drag-and-drop support is excellent in my opinion, and is one of the simplest and quickest to get up and running. You'll see it in action as we dissect the Shopping Cart application, but I hope you are already salivating with the possibilities it offers!

Another area of interest in MochiKit for this application is its Signal package. Signals are basically events, but the neat thing about them is they aren't necessarily *user* events; they can be virtually anything. For example, if you would like to call a function when some object on the page is clicked, you can do this:

```
connect('myID', 'onclick', myClicked);
```

When the object with the ID `myID` is clicked, `myClicked()` will be called.

What else does MochiKit have to offer? Oh, just a little bit—let Table 10-1 tell the story!

Table 10-1. *Some of the Many MochiKit Packages*

Package	Description
`MochiKit.Async`	Management of asynchronous tasks
`MochiKit.Base`	Functional programming and useful comparisons
`MochiKit.DOM`	Painless DOM manipulation functions
`MochiKit.Color`	Color abstraction with CSS3 support
`MochiKit.DateTime`	Time-related functions
`MochiKit.Format`	String formatting functions
`MochiKit.Iter`	Iterations
`MochiKit.Logging`	More robust logging capabilities than simple alerts
`MochiKit.LoggingPane`	Interactive logging pane
`MochiKit.Selector`	Element selection by CSS selector syntax

Table 10-1. *Some of the Many MochiKit Packages*

Package	Description
`MochiKit.Signal`	Simple universal event handling support
`MochiKit.Style`	Painless CSS manipulation functions
`MochiKit.Sortable`	A sortable object to make drag-and-drop lists easy
`MochiKit.Visual`	Visual effects

As is true with most of the libraries covered in this book, MochiKit has a lot more to offer than we can cover here, and your best bet is to spend some time on the MochiKit web site. Check out the documentation, try the examples, and get a feel for what it offers. It is quite well documented with some useful examples to play with, and I know you won't be disappointed.

■ **Note** The drag-and-drop features used in this application are not available in the most currently released version of MochiKit as of this writing, which is version 1.3.1. This is a fact I discovered rather painfully, because the documentation on the web site is for unreleased version 1.4 (although it might possibly be released by the time you read this). Therefore, in order to build this application. I had to get that unreleased version from source control, which uses the Subversion source control system. The URL for this code is `http://svn.mochikit.com/mochikit/trunk/`. You can find further information at the MochiKit web site's download page. This includes suggestions for Subversion clients (if you're using Windows as I generally do, I echo the suggestion of TortoiseSVN). Of course, when you download the source for this application from the Apress site, which I hope you've done already, you'll get the 1.4 version of MochiKit, all ready to go.

The Mock Server Technique

Well, calling the mock server approach a *technique* might be a tad grandiose, but it is a handy way to develop nonetheless.

When you write full-blown server code, more effort tends to go into it. You obviously need a server, plus a web and/or application server running. You may also need extensions installed, such as PHP, if that's your technology of choice. In some cases, such as Java and C#, aside from JSPs and ASPs that is, a compiler step is involved, which more times than not means you need to restart some server component for the changes to take effect. That's not even counting any additional development tools you may need, such as IDEs and the like.

Wouldn't it be nice it you could do this all on the server? Now, you're probably thinking "Hey, I have Tomcat or IIS or something running on my laptop, so I can do that." And indeed you can. I certainly do! But there is still more involved—compiling, restarting, and all that. What I'm talking about is truly serverless development, with nothing but client code involved.

Yes, you can do this! You can, in fact, have HTML and JavaScript playing the part of the server if you do things just right. This can increase your development speed quite a bit by eliminating the additional server code development steps. It can remove some potential points of failure, like network latencies and such. It also tends to simplify things because you don't have

to worry about getting server configurations right, dealing with operating system permissions, and so on. I hope you're convinced this might be a useful trick to know.

But how does one actually accomplish this feat of superhuman coding? It's amazingly simple. In pseudo-code, it looks like this:

```
<html>
  <head>
    <script>
      function process() {
        var function = get_request_parameter;
        switch (function) {
          case "some_operation":
            some_function();
          break;
        }
      }
    </script>
  </head>
  <body onLoad="process();"></body>
</html>
```

This is the mock server itself. It is nothing but an HTML page (you literally save it with an `.htm` or `.html` extension). Let's say you name it `mockServer.htm`. Then any time you have a form on another page, you do this:

```
<form name="myForm" method="get" action="mockServer.htm">
  Your Name: <input type="text" name="yourName">
  <input type="submit" value="Submit">
</form>
```

One important note is that the method *must* be GET, because that's the only way you'll be able to get access to the request parameters. You won't be able to access any parameters passed via POST.

You'll also see in the `mockServer.htm` file that I have `get_request_parameter`. This is a pseudo-code representation of some JavaScript that gets a request parameter for you. The function we're actually going to be using is `jscript.page.getParameter()` from Chapter 3, but we'll get to that in due course. The point here is that you get some parameter value, and then switch on that value to determine what code to execute. This is akin to the server reacting to some path and doing something different for each. The functions called can do anything you like, including render content or forward to another page. Again, you'll see this in action when we dissect the application. It's the basic concept that I'm hoping to get across right now.

I've already pointed out the benefits of this technique, but, of course, there are some negatives, too. Since this isn't a real server, you lose all the capabilities the server offers. You also need to be careful you don't do anything that you can't actually do on a real server. Still, even with the negatives, the simplicity and speed you gain are usually worth it, in my experience.

With those preliminaries out of the way, let's have a look at the application. If you haven't already done so, I suggest grabbing the code from the Apress web site and playing with it a bit.

A Preview of the Shopping Cart Application

Let's take a quick look at this chapter's project. First up, in Figure 10-1, you can see the application as it looks in the beginning. This is how it appears in non-JavaScript mode. I've added a few items to the cart already, as you can see down by the cart graphic.

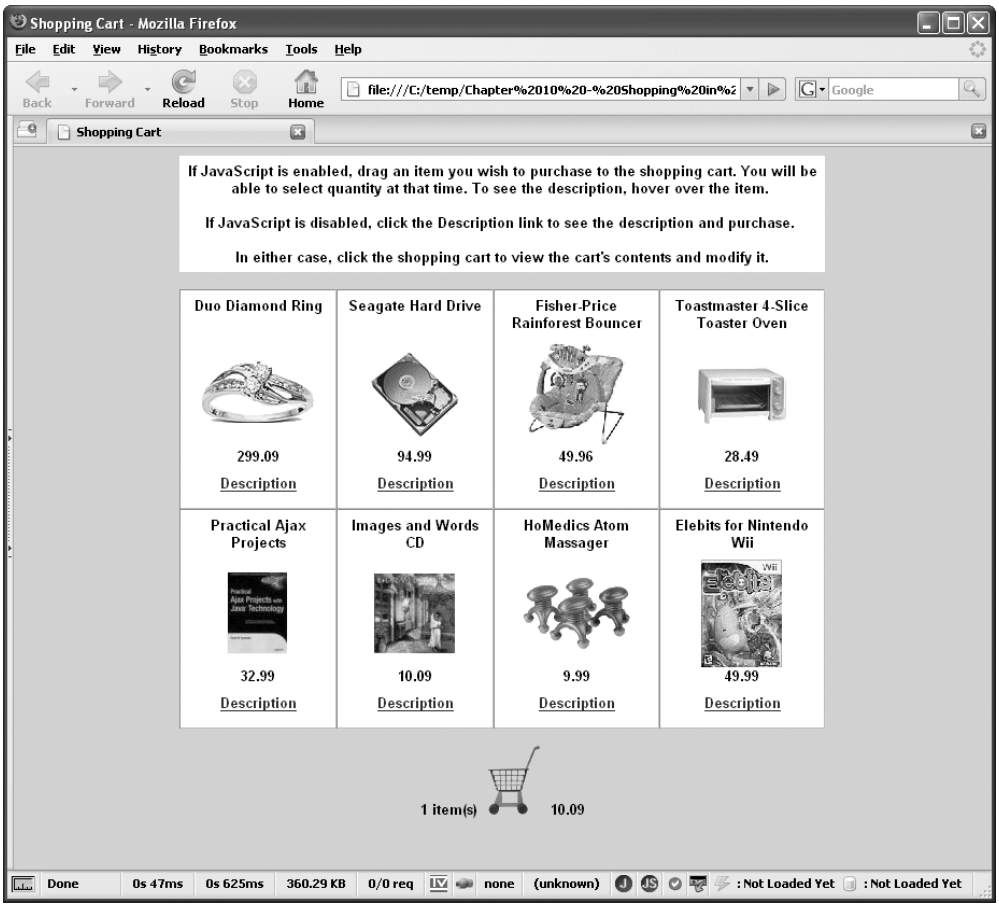


Figure 10-1. *The catalog view page when JavaScript is disabled*

Contrast Figure 10-1 with Figure 10-2, which is the catalog view when JavaScript is enabled. As you can see, the real difference is just the removal of the Description link. The description of the item is now seen when the user hovers the mouse over the item. In addition, there is no need to go to the description page you see when the Description link is clicked, because you can purchase from here as well, by simply dragging an item into the cart.

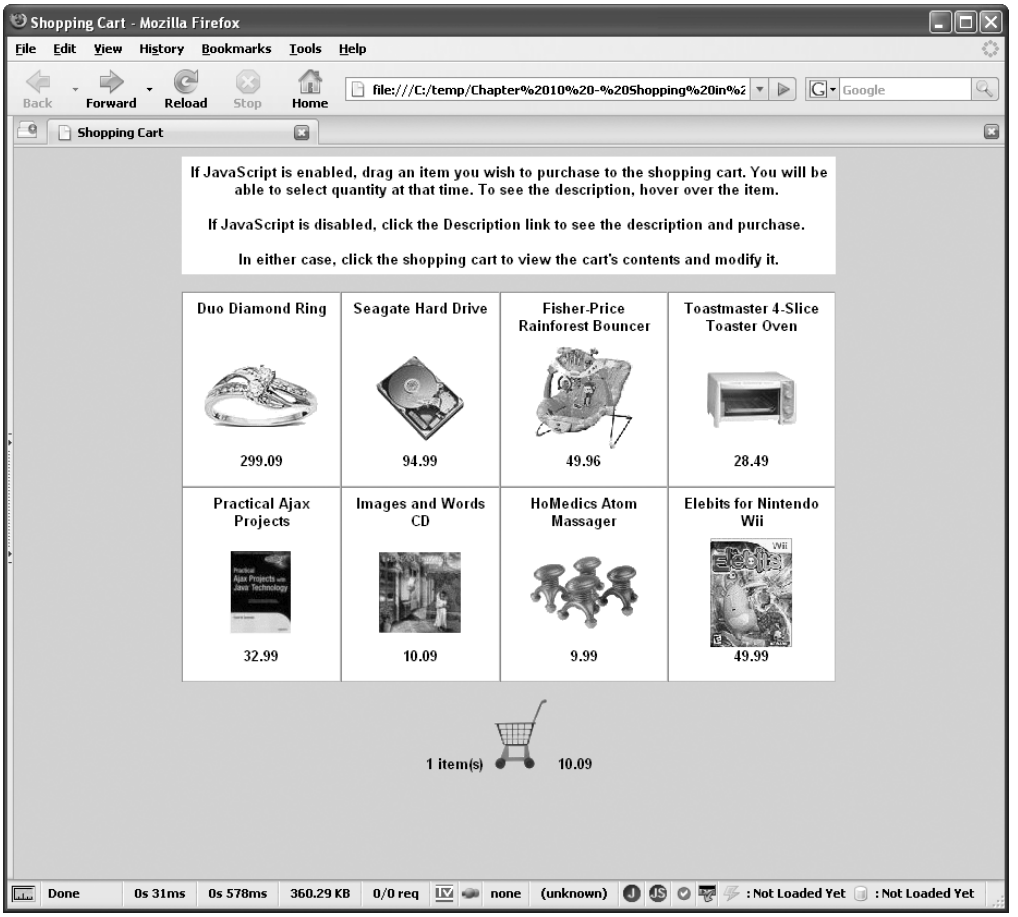


Figure 10-2. *The catalog view page when JavaScript is enabled*

In Figure 10-3, you can see the description pop-up. It will pop up at the current mouse location when you hover over an item. It will go away when you mouse off the image or if you start dragging the item.

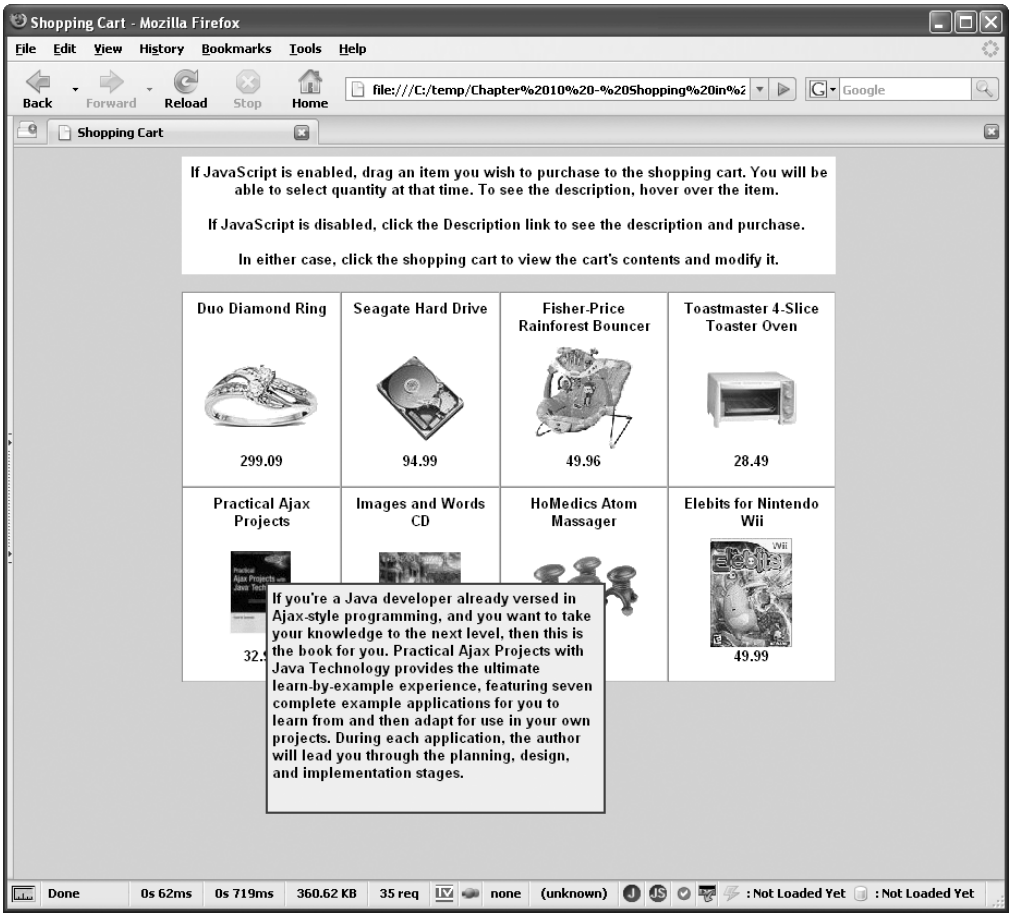


Figure 10-3. *The description pop-up seen when JavaScript is enabled and the user hovers the mouse over the item*

Although it's really hard to see unless you actually play with the application, I've tried to get a snapshot of an item being dragged in Figure 10-4. Notice that MochiKit is nice enough to fade out the image a little to give a nice effect to indicate it is being dragged.

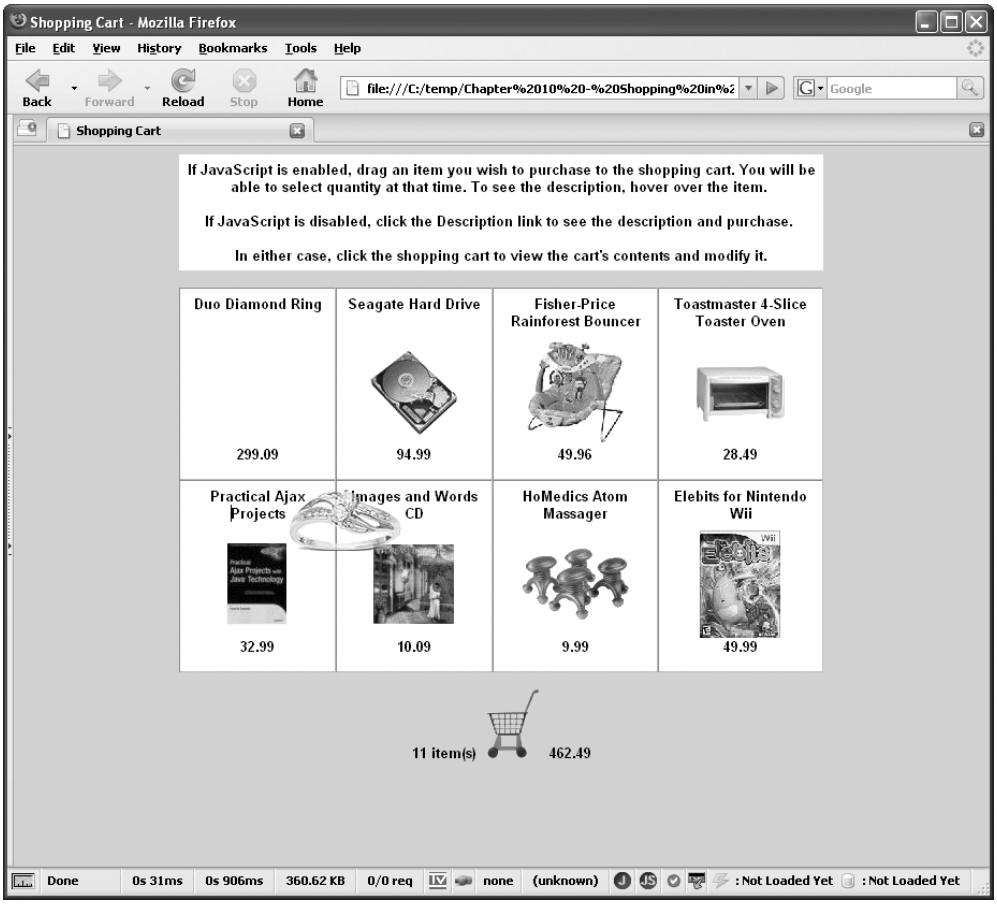


Figure 10-4. *An example of dragging an item (you really have to see it in action though)*

When you drag the item to the shopping cart, the next step is specify how many of the item you want. To accomplish this, a simple JavaScript `prompt()` gets the value, as shown in Figure 10-5.

A few other screens show up in the application, but I want to keep the anticipation going a bit longer!

Now we're ready to jump in and see what makes this application tick.

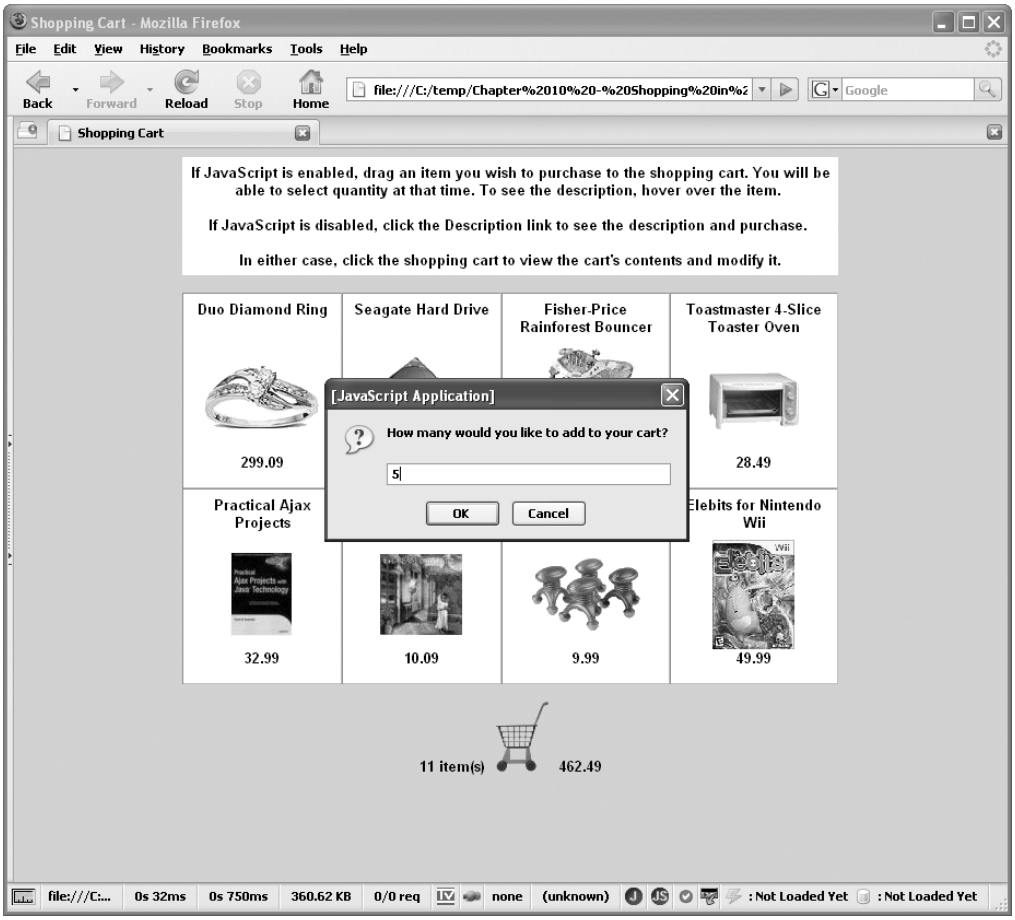


Figure 10-5. *The pop-up the user gets after dragging an item into the cart*

Dissecting the Shopping Cart Solution

As usual, we'll begin the dissection by taking a look at the layout of the application—the files that are part of it and all that. We begin this look with Figure 10-6, which shows the directory structure and file list.

Figure 10-6. *Directory layout of the Shopping Cart application*

This is the typical structure you've seen in most of the projects in this book, but we'll go over it anyway to be sure there are no surprises.

- `css`: This directory contains our single style sheet file, `styles.css`, which encapsulates all our style information.
- `img`: Here, we have all our image resources. In this case, the directory contains eight images: one for each of our purchasable items, and one for the shopping cart.
- `descs`: This directory contains the pages that show the item description and allow for purchasing an item when in non-JavaScript mode.
- `js`: This directory contains all (well, *almost* all, as you'll see) of the JavaScript for the application, including the MochiKit files in, not surprisingly, the `MochiKit` subdirectory.

- Finally, in the root are four HTML files:
  - `index.htm` is the catalog of items to purchase (which I refer to as the catalog view page).
  - `mockServer.htm` is, well, our mock server.
  - `viewCart.htm` is seen when viewing the contents of our shopping cart (which I refer to as the cart view page).
  - `checkout.htm` is what appears when the user tries to check out (which isn't much in this case!).

Let's not delay any longer. It's time to get to some code!

Writing styles.css

The first file we're going to explore is the style sheet for the application, `styles.css`. It's a pretty mundane style sheet frankly, but we should still have a look, if just a brief one. You can see the entire file in Listing 10-1.

Listing 10-1. *The styles.css File*

```
/* Style applied to all elements. */
* {
  font-family      : arial;
  font-size        : 10pt;
  font-weight      : bold;
}


/* Style for bodies of pages. */
.cssBody {
  background-color : #d0d0ff;
}


/* Style for instructions, and any other static text display areas. */
.cssInstructionsTable {
  background-color : #ffffff;
}


/* Style for the table used to display the catalog items. */
.cssCatalogTable {
  background-color : #ffffff;
  border           : 0px none #d0d0ff;
}
```

```
/* Style for the text on the checkout page. */
.cssCheckoutText {
  font-size        : 12pt;
}


/* Style for the header and footer of the view cart page. */
.cssHeaderFooter {
  background-color : #ffd0d0;
}


/* Row for the alternate row on the view cart page. */
.cssStripRow {
  background-color  : #efefef;
}


/* Style for description in cart view. */
.cssSmallDescription {
  font-size         : 8pt;
  font-weight       : normal;
}
```

As you can see, there is really nothing of any note here. The first style, as you've seen in other applications in this book, is a kind of catchall style that will effectively apply to everything on the page. The other styles work as follows:

- The `cssBody` class is applied to the `<body>` of all pages.
- The `cssInstructionsTable` class is for the instructions at the top of the page, and some other areas that need to be a white square, like the fake checkout page.
- The `cssCatalogTable` class is applied to the table that lays out the catalog. Primarily, its job is to remove the border from the table while leaving the cell borders, which isn't possible without CSS.
- The `cssCheckoutText` class is used to style the fake checkout page, to make that text a bit bigger.
- The `cssHeaderFooter` class is the red top and bottom you see on the view cart page.
- The `cssStripRow` class is used for the alternate gray rows on the view cart page, to make it easier to differentiate the rows of items.
- The `cssSmallDescription` class is used to style the description of the item on the view cart page.

Now let's move on to something just a littler meatier.

Writing index.htm

The `index.htm` file is the starting point of the application, and is what I call the catalog view. It is just some simple markup, with some JavaScript that may or may not actually do anything. First, let's see the code, as shown in Listing 10-2.

Listing 10-2. *The index.htm File*

```
<html>

  <head>

    <title>Shopping Cart</title>

    <link rel="StyleSheet" href="css/styles.css" type="text/css">

    <script type="text/javascript" src="js/MochiKit/MochiKit.js"></script>
    <script type="text/javascript" src="js/MochiKit/DragAndDrop.js"></script>

    <script type="text/javascript" src="js/jscript.page.js"></script>
    <script type="text/javascript" src="js/jscript.storage.js"></script>
    <script type="text/javascript" src="js/CatalogItem.js"></script>
    <script type="text/javascript" src="js/Catalog.js"></script>
    <script type="text/javascript" src="js/CartItem.js"></script>
    <script type="text/javascript" src="js/Cart.js"></script>
    <script type="text/javascript" src="js/main.js"></script>

  </head>

  <body class="cssBody" onLoad="init();">

    <div id="divMain">

      <table cellpadding="6" cellspacing="0" width="600" border="0"
        align="center" class="cssInstructionsTable">
        <tr>
          <td align="center" valign="middle">
            If JavaScript is enabled, drag an item you wish to purchase
            to the shopping cart.  You will be able to select quantity at that
            time.  To see the description, hover over the item.
            <br><br>
            If JavaScript is disabled, click the Description link to see the
            description and purchase.
            <br><br>
            In either case, click the shopping cart to view the cart's contents
            and modify it.
          </tr>
        </td>
      </table>
```

```
      <br>

      <table cellpadding="6" cellspacing="0" width="600" border="1"
        align="center" class="cssCatalogTable">

        <tr>

          <td align="center" valign="middle">
            <table border="0" cellpadding="0" cellspacing="0" width="100%">
              <tr><td height="40" align="center" valign="top">
                Duo Diamond Ring
              </td></tr>
              <tr><td height="100" align="center" valign="top" id="td_img_1">
                <img src="img/item1.gif" id="img_1">
              </td></tr>
              <tr><td height="25" align="center" valign="top">
                299.09
              </td></tr>
              <tr id="desc1"><td height="25" align="center" valign="top">
                <a href="mockServer.htm?function=viewDescription&itemID=1">
                  Description
                </a>
              </td></tr>
            </table>
          </td>

.... MARKUP FOR THREE OTHER ITEMS REMOVED ....

        </tr>

        <tr>

          <td align="center" valign="middle">
            <table border="0" cellpadding="0" cellspacing="0" width="100%">
              <tr><td height="40" align="center" valign="top">
                Practical Ajax Projects
              </td></tr>
              <tr><td height="100" align="center" valign="top" id="td_img_5">
                <img src="img/item5.gif" id="img_5">
              </td></tr>
              <tr><td height="25" align="center" valign="top">
                32.99
              </td></tr>
              <tr id="desc5"><td height="25" align="center" valign="top">
                <a href="mockServer.htm?function=viewDescription&itemID=5">
                  Description
                </a>
```

```
            </td></tr>
          </table>
        </td>

        .... MARKUP FOR THREE OTHER ITEMS REMOVED ....

      </tr>

    </table>

    <br>
    <center>
      <span id="spnItemCount"></span>
      <a href="mockServer.htm?function=viewCart"><img
        src="img/shoppingCart.gif" border="0"
        id="shoppingCart" hspace="6" alt="Click to view cart"></a>
      <span id="spnCartTotal"></span>
    </center>

  </div>

</body>

</html>
```

Note that two large chunks of HTML have been removed because they are very similar to the previous sections. The sections I refer to are a particular item. For example, the first item you can see in Listing 10-2 is this markup:

```
        <td align="center" valign="middle">
          <table border="0" cellpadding="0" cellspacing="0" width="100%">
            <tr><td height="40" align="center" valign="top">
              Duo Diamond Ring
            </td></tr>
            <tr><td height="100" align="center" valign="top" id="td_img_1">
              <img src="img/item1.gif" id="img_1">
            </td></tr>
            <tr><td height="25" align="center" valign="top">
              299.09
            </td></tr>
            <tr id="desc1"><td height="25" align="center" valign="top">
              <a href="mockServer.htm?function=viewDescription&itemID=1">
                Description
              </a>
            </td></tr>
          </table>
        </td>
```

As you can see, it's just straight HTML. Note the link for the description, which targets the `mockServer.htm` file. This is again what will take the place of a real server, and we'll get to that soon. Another fact that you need to be aware of is that the table row this markup is in has an ID defined. This matters quite a bit!

Let's back up a step though. Notice that most of our JavaScript files are imported onto this page, and also notice the call to `init()` `onLoad` of the document. This function is found in `main.js`, which we'll look at next. For now, the important thing to know is that when `init()` executes, if the application is running in non-JavaScript mode, it will do nothing. This also means that the fact that the table rows have IDs won't really matter in that mode.

However, the story is different when in JavaScript mode. In that case, the links in these rows are disabled, and that's why the rows have IDs—we need to be able to address them directly to remove their contents.

This page also has the shopping cart icon, which you can drag items onto when in JavaScript mode, and you can always click it to view the contents of the cart. Surrounding the graphic are some `<span>` elements, where the item count and cart total amount appear. Note that they are `<span>` elements, rather than `<div>` elements, so that they can be right next to the cart. Remember that `<div>` elements have a line break following them, so they would not appear next to the cart. `<span>` elements are not followed by a line break automatically, so they work well in this case.

OK, I've done a bit of foreshadowing, and now it's time to reveal the `main.js` file.

Writing main.js

As you just saw, `main.js` is imported into `index.htm`, and it contains the `init()` function that is called `onLoad` of the `index.htm` page. The content of `main.js` is shown in Listing 10-3.

Listing 10-3. *The main.js File*

```
/**
 * Set this to true to see the fancy version, false for the plain-jane version.
 */
var javaScriptEnabled = false;


/**
 * Called when the index.htm page loads.
 */
function init() {

  if (javaScriptEnabled) {

    // For each item...
    for (var i = 1; i < 9; i++) {

      // Remove the description link.
      document.getElementById("desc" + i).style.display = "none";

      // Hook up the description hover to it.
      var imgObject = document.getElementById("img_" + i)
```

```
      imgObject.onmouseover = cart.hoverDescriptionShow;
      imgObject.onmouseout = cart.hoverDescriptionHide;

      // Make the image draggable.
      new MochiKit.DragAndDrop.Draggable("img_" + i, { revert : true });

      // Event handler from drag starting (hides description popups).
      connect(Draggables, 'start', cart.onDragStart);

      // Create a description popup for the item.
      var descPopup = document.createElement("div");
      descPopup.setAttribute("id", "desc_" + i);
      descPopup.innerHTML =
        catalog.getItem(i).getItemDescription();
      descPopup.style.width = "300px";
      descPopup.style.height = "200px";
      descPopup.style.position = "absolute";
      descPopup.style.display = "none";
      descPopup.style.border = "2px solid #ff0000";
      descPopup.style.padding = "4px";
      descPopup.style.backgroundColor = "#efefef";
      document.getElementById("divMain").appendChild(descPopup);

    }

    // Make the shopping cart a drop target.
    new MochiKit.DragAndDrop.Droppable("shoppingCart",
      { ondrop : cart.doOnDrop }
    );

  }

  // Show the cart item count and dollar total.  This only really matters
  // if the user goes to the view cart or checkout pages and then goes back to
  // the catalog... when the cart is empty this basically has no effect.
  // Also note that what this function renders would be done by a server-side
  // component if JavaScript was disabled, but we're faking it here.
  cart.updateCartStats();

} // End init().
```

First, let's talk about the `javaScriptEnabled` variable. This global variable is the key to the concept of JavaScript-enabled and JavaScript-disabled mode. When set to `true`, the application is in JavaScript-enabled mode. This means it is acting as if JavaScript were available in the browser. When set to `false`, this is the equivalent to the user having disabled JavaScript. Of course, we need JavaScript to be enabled for real, because it is emulating the server. But this simple variable allows us to do the equivalent of turning JavaScript on and off.

What would happen if JavaScript were disabled for real? Well, `init()` would never be called `onLoad` of the document. If you look at the `init()` function, the first thing you see is this:

```
if (javaScriptEnabled) {
```

If `javaScriptEnabled` is set to `false`, then `init()` won't execute either. So, it truly is equivalent, and allows you to see the application in both situations just by changing the value of this variable, which is something I encourage you to do now.

Moving on, the next thing in `init()` is an iteration. The idea here is that we're going to modify all eight of the items in the catalog in some way. First, it gets a reference to the `<tr>` with the ID I mentioned previously and hides it. That's all there is to "removing" the Description link, which we don't need when JavaScript is enabled. Next, we need to enable the ability to hover over an item and see the description, and all that takes is this:

```
var imgObject = document.getElementById("img_" + i)
imgObject.onmouseover = cart.hoverDescriptionShow;
imgObject.onmouseout = cart.hoverDescriptionHide;
```

After we get a reference to the appropriate image, we set the `onMouseOver` and `onMouseOut` handlers to point to the appropriate methods in the `cart` object, which is an instance of the `Cart` class that you'll see later. In short, this is the shopping cart itself, and all the functionality it encapsulates.

After that comes the step of making the image draggable, which is what MochiKit does for us. It's very simple:

```
new MochiKit.DragAndDrop.Draggable("img_" + i, { revert : true });
connect(Draggables, 'start', cart.onDragStart);
```

We are instantiating a `MochiKit.DragAndDrop.Draggable` object and giving it a reference to the image of the item. We are also passing in some options; well, one option to be precise. The `revert` option can be an effect, a function, or a simple `true/false` value—in this case, it is the latter. This tells MochiKit that when the draggable object is dropped, whether or not it's dropped on the shopping cart (a droppable object), we want the object to revert back to its starting position. MochiKit does this with some flair by default, using a `Move` effect to have the object—our item's image in this case—fly back to its starting position. You can use this in conjunction with the `reverteffect` attribute, which is another option you can pass here, to determine which effect to use, but in this case, the default does rather nicely.

The second line that begins with `connect()`, is part of the MochiKit event system. It allows you to attach handlers to numerous and varied events on the page. In this case, we are saying that any `Draggable` object, which is what `Draggables` means because all `Draggable` objects are contained in the `Draggables` collection, should trigger execution of the `onDragStart()` method of the `cart` object whenever dragging starts. This is needed so that we can hide the pop-up description when dragging begins.

The next thing `init()` does is creates the pop-up descriptions for our items, and adds them to the DOM (because they aren't there initially). This is pretty typical DOM manipulation code: create an instance of `<div>`, populate its attributes, give it some content via setting `innerHTML`, and append it to the DOM as a child of some element—in this case, the `divMain` `<div>`, which surrounds all the contents on the page.

The last step of `init()` is to make the shopping cart something an image can be dropped on. We do this with a single line of code:

```
new MochiKit.DragAndDrop.Droppable("shoppingCart",
  { ondrop : cart.doOnDrop }
);
```

The `MochiKit.DragAndDrop.Droppable` class is the complement to its `Draggable` class, and its call signature is very similar. Here, we are giving it the ID of our shopping cart image, and again passing some options—in this case, just the function to call when something is dropped on the object. You'll see this function very soon, but as you can imagine, it is responsible for actually adding the dropped item to the cart.

Writing id*X*.htm

When the application is in non-JavaScript mode, the user clicks the Description link underneath an item to both view the description and purchase it. The page that the user sees at this point is shown in Figure 10-7.

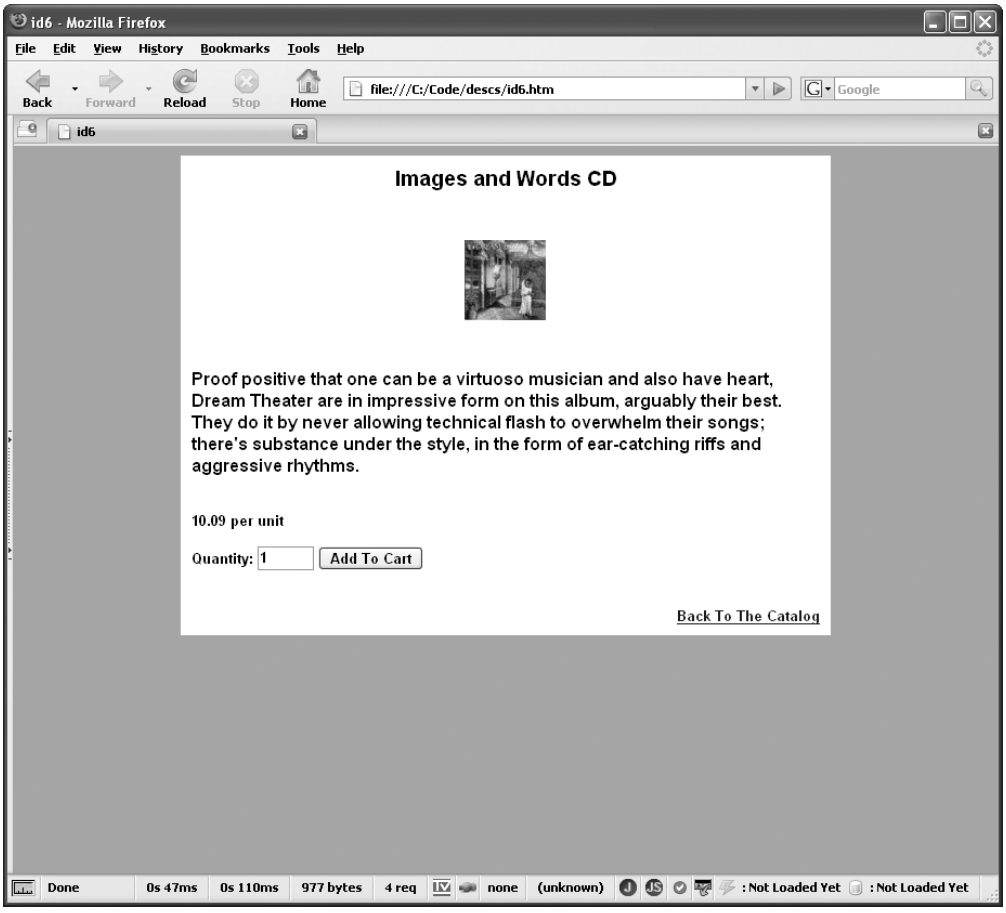


Figure 10-7. *An example of the description/purchase page in the non-JavaScript version*

Each of the items has its own HTML document, named `idX.htm`, where *X* is the ID number of the item (1 through 8). These files are found in the `/descs` directory. Because they are all identical except for the information about the item, I've shown the listing for only one here, in Listing 10-4.

Listing 10-4. *The id1.htm File (Other Files in /descs Are Virtually Identical)*

```
<html>
  <head>
    <title>id1</title>

    <link rel="StyleSheet" href="../css/styles.css" type="text/css">

    <script>
    </script>

  </head>

  <body style="background-color:#a0a0ff;">
    <table border="0" cellpadding="10" cellspacing="0" width="600"
      align="center" style="background-color:#ffffff;"><tr><td>
      <center>
        <div style="font-size:14pt;">Duo Diamond Ring</div>
        <br><br>
        <img src="../img/item1.gif">
      </center>
      <br><br>
      <div style="font-size:12pt;">
        This 10K gold Duo ring features two round diamonds in prong settings with
        round diamond accents.  Duo Jewelry is designed to celebrate a couple's
        love.
      </div>
      <br><br>
      299.09 per unit
      <br><br>
      <form name="purchase" method="get" action="../mockServer.htm">
        <input type="hidden" name="function" value="purchase">
        <input type="hidden" name="itemID" value="1">
        Quantity:
        <input type="text" size="3" maxlength="2" name="quantity" value="1">
        <input type="submit" value="Add To Cart">
      </form>
    </tr></tr>
    <tr><td align="right">
      <a href="../mockServer.htm?function=viewCatalog">Back To The Catalog</a>
    </td></tr></table>
```

```
  </body>

</html>
```

As you can see, it is perfectly straightforward HTML; no JavaScript to speak of. In fact, I dare say the only items of interest here are the target of the form submission and the Back To The Catalog link. Notice that both of them target the `mockServer.htm` file, which we'll be looking at in detail shortly. The important point now is that, conceptually, this document takes the place of a real server. Notice the `function` parameter is passed as part of the query string in the link (also notice that the method of the form is GET) and as a form field in the form submission. The `mockServer.htm` file uses the `getParameter()` function in the `jscript.page` package from Chapter 3 to get the parameters passed to it. For this to work, however, the parameters must have been passed as a query string, Parameters passed through POST cannot be read, and that's the reason for the form's method.

The `mockServer.htm` file will look for that `function` parameter and use it to determine which operation it should perform. That's a good enough description of it for now; as I said, you'll see it in more detail later in the chapter.

Writing CatalogItem.js

The `CatalogItem` class represents a single item in the catalog that the user can purchase. It has a handful of data elements to describe it, and the class also has the typical getters and setters to access it in a good object-oriented way. Figure 10-8 shows the UML diagram of the class.

Figure 10-8. *UML diagram for the CatalogItem class*

The `CatalogItem` class has the following five fields:

- `itemID`: The ID of the item. For this application, I chose to make it just a simple number, 1–8. There's nothing that says it has to be. I just figured KISS: Keep It Simple, Stupid!
- `itemTitle`: The short title of the item as seen on the catalog view page.
- `itemDescription`: The lengthier description you see when you hover over the image in JavaScript mode, or when you click the Description link in non-JavaScript mode.
- `itemImageURL`: Stores the URL to the item's image.
- `itemPrice`: The price of the item.

All of the methods in the class are simple getters and setters for the various fields. There is also an overridden `toString()` method, so we can get a more meaningful representation of an instance of this class, which is especially good when trying to debug the application.

Even though this is a simple class—really just a Data Transfer Object (DTO)—we should still look at the code, which is shown in Listing 10-5.

Listing 10-5. *The CatalogItem Class in CatalogItem.js*

```
/**
 * This class represents one item in the catalog.
 */
function CatalogItem() {


  /**
   * The ID of the item.
   */
  var itemID = "";


  /**
   * The title of the item.
   */
  var itemTitle = "";


  /**
   * The description of the item.
   */
  var itemDescription = "";
```

```
/**
 * The URL to the image of the item.
 */
var itemImageURL = "";


/**
 * The price for one of the items.
 */
var itemPrice = 0;


/**
 * Setter.
 *
 * @param inItemID New value.
 */
this.setItemID = function(inItemID) {

  itemID = inItemID;

} // End setItemID().


/**
 * Getter.
 *
 * @return The current value of the field.
 */
this.getItemID = function() {

  return itemID;

} // End getItemID().


/**
 * Setter.
 *
 * @param inItemTitle New value.
 */
this.setItemTitle = function(inItemTitle) {

  itemTitle = inItemTitle;

} // End setItemTitle().
```

```
/**
 * Getter.
 *
 * @return The current value of the field.
 */
this.getItemTitle = function() {

  return itemTitle;

} // End getItemTitle().


/**
 * Setter.
 *
 * @param inItemDescription New value.
 */
this.setItemDescription = function(inItemDescription) {

  itemDescription = inItemDescription;

} // End setItemDescription().


/**
 * Getter.
 *
 * @return The current value of the field.
 */
this.getItemDescription = function() {

  return itemDescription;

} // End getItemDescription().


/**
 * Setter.
 *
 * @param inItemImageURL New value.
 */
this.setItemImageURL = function(inItemImageURL) {

  itemImageURL = inItemImageURL;

} // End setItemImageURL().
```

```
/**
 * Getter.
 *
 * @return The current value of the field.
 */
this.getItemImageURL = function() {

  return itemImageURL;

} // End getItemImageURL().


/**
 * Setter.
 *
 * @param inItemPrice New value.
 */
this.setItemPrice = function(inItemPrice) {

  itemPrice = inItemPrice;

} // End setItemPrice().


/**
 * Getter.
 *
 * @return The current value of the field.
 */
this.getItemPrice = function() {

  return itemPrice;

} // End getItemPrice().


/**
 * Overriden toString() method.
 *
 * @return A meaningful string representation of the object.
 */
this.toString = function() {
```

```
    return "CatalogItem : [ " +
      "itemID='" + itemID + "', " +
      "itemTitle='" + itemTitle + "', " +
      "itemDescription='" + itemDescription + "', " +
      "itemImageURL='" + itemImageURL + "', " +
      "itemPrice='" + itemPrice + "' ]";

  } // End toString().


} // End CatalogItem class.
```

It's not going to win any awards as a complex piece of coding, but it gets the job done. Now, of course, a `CatalogItem` instance wouldn't be a ton of good on its own; it must be part of a catalog that knows how to deal with it. Such a beast exists, and it is not surprisingly the `Catalog` class!

Writing Catalog.js

The `Catalog` class is where all the `CatalogItem` instances that are part of the catalog of items the user can purchase are stored. This is a very simple class, as you can see in its UML diagram in Figure 10-9.

Figure 10-9. *UML diagram of the Catalog class*

The `catalogItems` field is the collection of `CatalogItems`, one for each item the user can purchase. The `getItem()` method will return a given item if you pass it the item's ID. This is used in a number of places, as you might imagine.

Wait, didn't I skip something? Oh yes, those `itemX` fields, where *X* is 1 through 8. I wonder what's going on there? Let's take a look at the code and see, as shown in Listing 10-6.

Listing 10-6. *The Code Behind the Catalog Class, in Catalog.js*

```
/**
 * This class represents a catalog of items.
 */
function Catalog() {


  /**
   * The collection of items for sale in the catalog.
   */
  var catalogItems = new Object;


  /**
   * Load some items so the user can play, assuming JavaScript is enabled.
   */
  var item1 = new CatalogItem();
  item1.setItemID("1");
  item1.setItemTitle("Duo Diamond Ring");
  item1.setItemDescription("This 10K gold Duo ring features two round diamonds in ➥
prong settings with round diamond accents.  Duo Jewelry is designed ➥
to celebrate a couple's love.");
  item1.setItemImageURL("img/item1.gif");
  item1.setItemPrice(299.09);
  catalogItems[item1.getItemID()] = item1;

.... CODE FOR SEVEN OTHER ITEMS REMOVED ....

  /**
   * Returns a CatalogItem by ID.
   *
   * @param  inItemID The ID of the item to return.
   * @return          The corresponding item, or null if not found.
   */
  this.getItem = function(inItemID) {

    return catalogItems[inItemID];

  } // End getItem().


} // End Catalog class.


// The one and only instance of the items catalog.
var catalog = new Catalog();
```

Well, I did cut it down a bit for print!

Notice the `.... CODE FOR SEVEN OTHER ITEMS REMOVED ....`? As it says, there are seven other blocks of code very similar to the block right before that notation, namely this one:

```
  var item1 = new CatalogItem();
  item1.setItemID("1");
  item1.setItemTitle("Duo Diamond Ring");
  item1.setItemDescription("This 10K gold Duo ring features two round diamonds in ➥
prong settings with round diamond accents.  Duo Jewelry is designed ➥
to celebrate a couple's love.");
  item1.setItemImageURL("img/item1.gif");
  item1.setItemPrice(299.09);
  catalogItems[item1.getItemID()] = item1;
```

Here, we're creating a `CatalogItem` instance, populating it for a particular item, and adding it to the `catalogItems` collection. This is where the items in our catalog come from. Although I've listed the `item`*X* fields in the UML diagram, in practice, they are used during construction of the `Catalog` object, and never again. Still, they should be listed for completeness. Note that these eight blocks of code are not within a method of the `Catalog` class; hence, they will be executed when `Catalog` is instantiated. They are in the constructor, in other words, and this is precisely what we want. There is no need to make the user of this class call some setup method explicitly.

Now that we know about the item catalog, let's talk about the shopping cart and the items that go into it.

Writing CartItem.js

The `CartItem` class represents a single item in the shopping cart. Figure 10-10 shows the UML diagram of this small class.

Figure 10-10. *UML diagram of the CartItem class*

A `CartItem` has only two important pieces of information stored within it: the `itemID` field, which matches the `itemID` field of a particular `CatalogItem` instance, and `quantity`, which is obviously the quantity of this particular item in the cart. You see the usual getters and setters for these two fields, as well as an overridden `toString()`, as in the `CatalogItem` class.

You also see a `serialize()` method. To make a long story short, we are going to be storing the contents of the cart in a cookie, and to do so, we need to have a string representation of each `CartItem` in the cart. The `serialize()` method returns this representation. It is nothing but the `itemID` and `quantity`, separated by a tilde (~) character.

Let's now look at Listing 10-7, which is the complete listing of the `CartItem` class.

Listing 10-7. *The CartItem Class in CartItem.js*

```
/**
 * This class represents one item in the shopping cart.
 */
function CartItem() {


  /**
   * The ID of the item.
   */
  var itemID = "";


  /**
   * The quantity of the item in the cart.
   */
  var quantity = 0;


  /**
   * Setter.
   *
   * @param inItemID New value.
   */
  this.setItemID = function(inItemID) {

    itemID = inItemID;

  } // End setItemID().


  /**
   * Getter.
   *
   * @return The current value of the field.
   */
  this.getItemID = function() {

    return itemID;
```

```
} // End getItemID().


/**
 * Setter.
 *
 * @param inQuantity New value.
 */
this.setQuantity = function(inQuantity) {

  quantity = inQuantity;

} // End setQuantity().


/**
 * Getter.
 *
 * @return The current value of the field.
 */
this.getQuantity = function() {

  return quantity;

} // End getQuantity().


/**
 * Returns a serialized version of the item suitable for writing out to the
 * cookie.
 */
this.serialize = function() {

  return itemID + "~" + quantity;

} // End serialize().


/**
 * Overriden toString() method.
 *
 * @return A meaningful string representation of the object.
 */
this.toString = function() {
```

```
    return "CartItem : [ " +
      "itemID='" + itemID + "', " +
      "quantity='" + quantity + "' ]";

  } // End toString().


} // End CartItem class.
```

There shouldn't be any surprises here at all. It's pretty boring code, to put it bluntly! Still, without it, none of this would work, so it's an important piece of boring code!

Next, we come to some code that is decidedly less boring: the `Cart` class.

Writing Cart.js

The `Cart` class, contained in the `Cart.js` source file, is truly where most of the application resides—where all the real functionality behind it is. As you saw when we looked at `main.js`, all the functions that handle the various events, such as dragging and dropping images, can be found in this class. Let's first take a look at its UML diagram, shown in Figure 10-11.

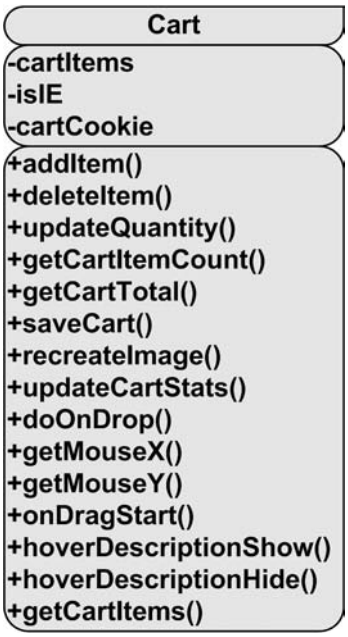


Figure 10-11. *UML diagram of the Cart class*

Because `Cart.js` is a fairly lengthy file, I won't list all of it here, but will show sections of it as required. First, we should look at the three fields it contains: `cartItems`, `isIE`, and `cartCookie`.

The `cartItems` field is an array that contains `CartItem` instances. These are the items that are currently in the cart.

The `isIE` field is needed later to deal with mouse events. Let's take a quick look at this:

```
var isIE = window.ActiveXObject ? true : false;
```

Since IE is the only browser that supports ActiveX controls (ignoring plug-ins that may exist for other browsers), it is the only browser that will return `true` when we check for the `ActiveXObject` attribute of the `window` object. Hence, it's an easy way to check if we're running in IE.

The `cartCookie` field is needed only temporarily, but I listed it anyway for completeness. It is a string that is the value of the cookie used to store the cart's contents.

Restoring the Cart's Contents

After the fields is some code that will execute when the class is instantiated. The job of this code is to read the value of the cookie where the cart's contents are stored, create `CartItem` objects from it, and store them in the `cartItems` field. Here is the code that accomplishes all that:

```
var cartCookie = jscript.storage.getCookie("js_shopping_cart");
if (cartCookie) {
  var itemsInCart = cartCookie.split("~~");
  for (var i = 0; i < itemsInCart.length; i++) {
    var nextItem = itemsInCart[i];
    var nextItemID = nextItem.split("~")[0];
    var nextItemQuantity = nextItem.split("~")[1];
    var cartItem = new CartItem();
    cartItem.setItemID(nextItemID);
    cartItem.setQuantity(nextItemQuantity);
    cartItems.push(cartItem);
  }
}
```

As you can see, the `jscript.storage.getCookie()` function that we built in Chapter 3 is used to get the value of the cookie, which is named `js_shopping_cart`. Assuming the cookie is found (which it wouldn't be the first time the user uses the application, of course), we then need to parse it.

The way the cart is stored is in the form *AA~BB~~CC~DD*. *AA* and *CC* are item IDs; *BB* and *DD* are the quantity of that item. As you can see, the ID and quantity are separated by a single tilde (`~`) character, and items are separated by two tildes. This makes it very easy to parse. We just need to use the `split()` method of the JavaScript `String` class to split on the double-tilde sequence, which gives us an array of items. Then we iterate over that array, and for each item, we use `split()` again, this time on the single tilde. The first element of the resultant array is the ID, and the second is the quantity. It's very easy!

All we need to do then is instantiate a `CartItem` object and fill in its details, and finally `push()` it onto the `cartItems` array. When we're finished going through all the items in the first array, the `cartItems` field contains a `CartItem` object for each item in the cart.

The first method you find in the source is `getCartItems()`. This simply returns the `cartItems` field—nothing more. This will be needed by our mock server code, as you'll see later.

Adding and Removing Cart Items

Next up is the addItem() method:

```
this.addItem = function(inItemToAdd) {

  cartItems.push(inItemToAdd);
  saveCart();

} // End getCartItems().
```

Obviously, this is called to add an item to the cart. It takes as an argument an instance of the CartItem class, which is presumed to be populated correctly. All addItem() does is push the incoming CartItem into the cartItems array, and then calls the saveCart() method, which is responsible for actually saving the cart as a cookie. We'll get to that method in just a bit.

Before that though, we find the deleteItem() method:

```
this.deleteItem = function(inItemIndex) {

  cartItems.splice(inItemIndex, 1);
  saveCart();

} // End deleteItem().
```

This method is equally as simple as addItem(). All it does it use the splice() method of the cartItems array to delete the item and the index that is passed in, and then calls saveCart() as well, so that the cookie is updated.

Updating Item Quantities in the Cart

The next method, updateQuantity(), is called from the view cart page to change the quantity of a given item in the cart:

```
this.updateQuantity = function(inItemIndex, inNewQuantity) {

  var cartItem = cartItems[inItemIndex];
  cartItem.setQuantity(inNewQuantity);
  saveCart();

} // End updateQuantity().
```

It first gets a reference to the CartItem object corresponding to the index passed into it. It then calls the setQuantity() method of that object, passing it the new quantity passed into updateQuantity(). Finally, it calls saveCart().

After updateQuantity() is the getCartItemCount() method, which is only marginally more complex:

```
this.getCartItemCount = function() {

  var cartItemCount = 0;
  for (var i = 0; i < cartItems.length; i++) {
    cartItemCount += parseInt(cartItems[i].getQuantity());
  }
  return cartItemCount;

} // End getCartItemCount().
```

This is a simple iteration over the `cartItems` array. For each element in the array, which is a `CartItem` object, we call the `getQuantity()` method on it, and accumulate this value. Remember that we want the total number of items in the cart to be purchased, which can be different from the total number of `CartItems` in the cart. Once the iteration is complete, the final tally is returned.

`getCartTotal()` is the next method here:

```
this.getCartTotal = function() {

  var cartTotal = 0;
  for (var i = 0; i < cartItems.length; i++) {
    var nextItem = cartItems[i];
    var nextItemQuantity = nextItem.getQuantity();
    var nextItemID = nextItem.getItemID();
    var catalogItem = catalog.getItem(nextItemID);
    cartTotal += nextItemQuantity * catalogItem.getItemPrice();
  }
  return cartTotal;

} // End getCartTotal().
```

This isn't too much different from `getCartItemCount()`. We again are iterating over the `cartItems` array. For each `CartItem` we find, we get the quantity of it, as well as its ID via a call to `getIemID()`. With this ID in hand, we then call the `getItem()` method of the `catalog` object, which returns the `CatalogItem` corresponding to the item being purchased. From that, we can get the price for one unit. It's a simple matter to multiply that price by the quantity of the item in the cart retrieved earlier. We keep a running total of this calculated value, and at the end, we have the total dollar amount for the cart.[4]

4. For the demo, I've simplified things and didn't concern myself with tax, shipping charges, and so on. Shh, don't tell the federal government, UPS, or FedEx!

Saving the Cart

Finally, we come to the `saveCart()` method, which I've mentioned a bunch of times:

```
var saveCart = function() {

  // Construct shopping cart string for cookie and store it.
  var shoppingCart = "";
  for (var i = 0; i < cartItems.length; i++) {
    nextItem = cartItems[i];
    if (shoppingCart != "") {
      shoppingCart += "~~";
    }
    shoppingCart += nextItem.serialize();
  }
  var expireDate = new Date();
  expireDate.setDate(expireDate.getDate()+7)
  jscript.storage.setCookie("js_shopping_cart", shoppingCart, expireDate);
} // End saveCart().
```

I bet you thought there would be more to it. Nope, that's it! Like the other methods, `saveCart()` iterates over the `cartItems` array. For each element, we simply call the `serialiaze()` method on the `CartItem`, which returns a string in the form *X~Y*, where *X* is the item ID and *Y* is the quantity. We are ultimately building up a string in the form *X~Y~~X~Y*. After we have that string constructed, we just need to use the `jscript.storage.setCookie()` function from Chapter 3, and the shopping cart is then saved as a cookie.

Notice that the expiration date of the cookie is set seven days in the future. So if you leave the shopping cart and come back, for up to seven days, your content will still be present. That's obviously longer than you would use on a real shopping site, but it demonstrates the point well here.

Updating Stats

After the `saveCart()` method comes the `updateCartStats()` method. This is used to display the number of items and total dollar amount of the cart next to the shopping cart graphic at the bottom. It is called when an item is dropped on to the cart:

```
this.updateCartStats = function() {

  // Put the total item count and dollar amount of the cart on the screen,
  // if and only if there are items in the cart already.
  var spnCartCountValue = "";
  var spnCartTotalValue = "";
  var cartItemCount = cart.getCartItemCount();
  if (cartItemCount != 0) {
    spnCartCountValue = cartItemCount + " item(s)";
    spnCartTotalValue = cart.getCartTotal();
```

```
    // Now some math: the total dollar amount has to be rounded for proper
    // display.  The basic logic harkens back to pre-algebra:
    // * Multiply the number by 10^x
    // * Apply Math.round() to the result
    // * Divide the result by 10^x
    spnCartTotalValue = Math.round(spnCartTotalValue * 100) / 100;
  }
  document.getElementById("spnItemCount").innerHTML = spnCartCountValue;
  document.getElementById("spnCartTotal").innerHTML = spnCartTotalValue;

} // End updateCartStats().
```

It begins by creating two variables, `spnCartCountValue` and `spnCartTotalValue`. The first is the number of items in the cart, and the second is the total dollar amount of the cart. Note that these are string values, which might seem a little odd, since we know conceptually these are numeric values. But these will be inserted as the `innerHTML` of some `<span>` tags, so it's more logical that they be strings. (And, in fact, if we have some variable with a value of zero as a number, and try to insert it into the `<span>`, it will be converted to a string. But do we really want to show a zero, or do we want to show nothing at all—that is, an empty string?) It then calls the `getCartItemCount()` method of the `cart` object to get the total number of items. If this value is anything other than zero, we take the value and append the string `" item(s)"` to it, turning it back into a string. We also call the `getCartTotal()` method on the cart object to get the dollar amount.

After that comes inserting these values into the appropriate `<span>` tags that you saw in `index.htm`. There is some funkiness here to be performed because the dollar value can have decimal points, but because it is truly a numeric value, the number of decimal points could be rather large. For a dollar amount, however, we want only two decimal places, so we need to do some rounding. The comments describe the basic algorithm of this rounding, which is just some basic math.

One final check is then performed to give us an empty string if the value is zero (which is what you would get the first time through when there are no items in the cart). Then `innerHTML` of the `<span>` elements is set, and we have updated statistics next to the shopping cart!

Handling Dropped Items

The `doOnDrop()` method, as you'll recall from looking at the code in `main.js`, is called when an item is dropped onto the shopping cart:

```
this.doOnDrop = function(element) {

  // Get the ID of the item dropped in the cart.
  var itemID = element.id.split("_")[1];
```

```
    // Find out how many the user wants.
    var quantity =
      parseInt(prompt("How many would you like to add to your cart?"));
    if (!isNaN(quantity) && quantity != 0) {
      // Create a cart item and add it to the cart.
      var cartItem = new CartItem();
      cartItem.setItemID(itemID);
      cartItem.setQuantity(quantity);
      cart.addItem(cartItem);
    }

    // Show the cart item count and dollar total.
    cart.updateCartStats();

  } // End doOnDrop().
```

First, we get the `itemID` of the item, as in previous methods. After that, we pop up a prompt for users using the JavaScript `prompt()` function, so they can enter the quantity they want. The return value from this call could be something other than a number, or it could be zero, both of which would abort adding the item to the cart, so we have a check for both of those conditions. Assuming a number was entered, however, we go ahead and instantiate a new `CartItem` object, populate its attributes, which are just `itemID` and quantity, and send it to the `addItem()` method of the `cart` object. Lastly, we call `updateCartStats()` so the newly added item is reflected in the statistics next to the shopping cart.

Showing and Hiding the Hover Description

Next are two functions that are used when showing the hover description of the item: `getMouseX()` and `getMouseY()`. As their names imply, they get the X and Y location of a given mouse event. Because IE and Firefox (as well as other browsers) provide this information in different ways, this is where we need that `isIE` field that you saw earlier. Because these methods are very similar, I'll just show one for brevity:

```
  this.getMouseX = function(inEvent) {

    var x;
    if (isIE) {
      x = (parseInt(event.clientX ) +
        parseInt(document.body.scrollLeft));
    } else {
      x = parseInt(inEvent.pageX);
    }
    return x;

  } // End getMouseX().
```

When running in IE, the event object is provided at page scope, which has as one of its members the clientX attribute. When we take this value and add the document.body.srollLeft attribute, we get the absolute X coordinate of the mouse event.

When running in Firefox, or other browsers, the inEvent object is passed to this method, which contains the pageX attribute. This is the X coordinate of the mouse event. Notice that we don't have to take into consideration how far the page is scrolled horizontally, as we do in IE, because the pageX value already has that taken into account. This is what accounts for the difference in the branches of this code, aside from the difference in which attribute we go after and to which object it belongs. The getMouseY() method is again identical, with a few exceptions: instead of clientX, it's clientY; it's scrollTop instead of scrollLeft; and it's pageY instead of pageX.

And now we can see the code that makes use of those two functions, namely hoverDescriptionShow():

```
this.hoverDescriptionShow = function(inEvent) {

  var itemID = this.id.split("_")[1];
  var mouseX = cart.getMouseX(inEvent);
  var mouseY = cart.getMouseY(inEvent);
  var descObj = document.getElementById("desc_" + itemID);
  descObj.style.left = mouseX;
  descObj.style.top = mouseY;
  descObj.style.display = "block";

} // End hoverDescriptionShow().
```

Note the use of the this keyword. In the context of this method, which as you'll remember is attached to a given item's image, the this keyword is a reference to the image. That should explain why we do this.id to get the itemID: it's the DOM ID of the <img> element. We again split() this to get the second element of the resultant array, which is the itemID we want. We then call those mouse methods to get the current coordinates of the mouse on the page. Once we have that, we get a reference to the that contains the description of the item being hovered over. We then set the left and top style attributes of this to the mouse coordinates we just got, and finally show the by setting its display style attribute to block. The description is then visible to the user at the place where the mouse cursor hovered over the image.

The last method in the Cart class is the hoverDescriptionHide() method. It just gets the itemID in the same way as in hoverDescriptionShow() (because this method is the event handler attached to the onMouseOut event of the image) and sets its display style attribute to none. That's it.

Now, let's look at the page that shows the contents of our shopping cart, viewCart.htm (if you can believe it!).

Writing viewCart.htm

The viewCart.htm file takes a bit of a leap of faith. We have some JavaScript in it that is emulating what the server would do. There is no JavaScript that would actually run on the client if this were a full-blown e-commerce site.

Before we get too far into that though, let's have a look at this page, shown in Figure 10-12.

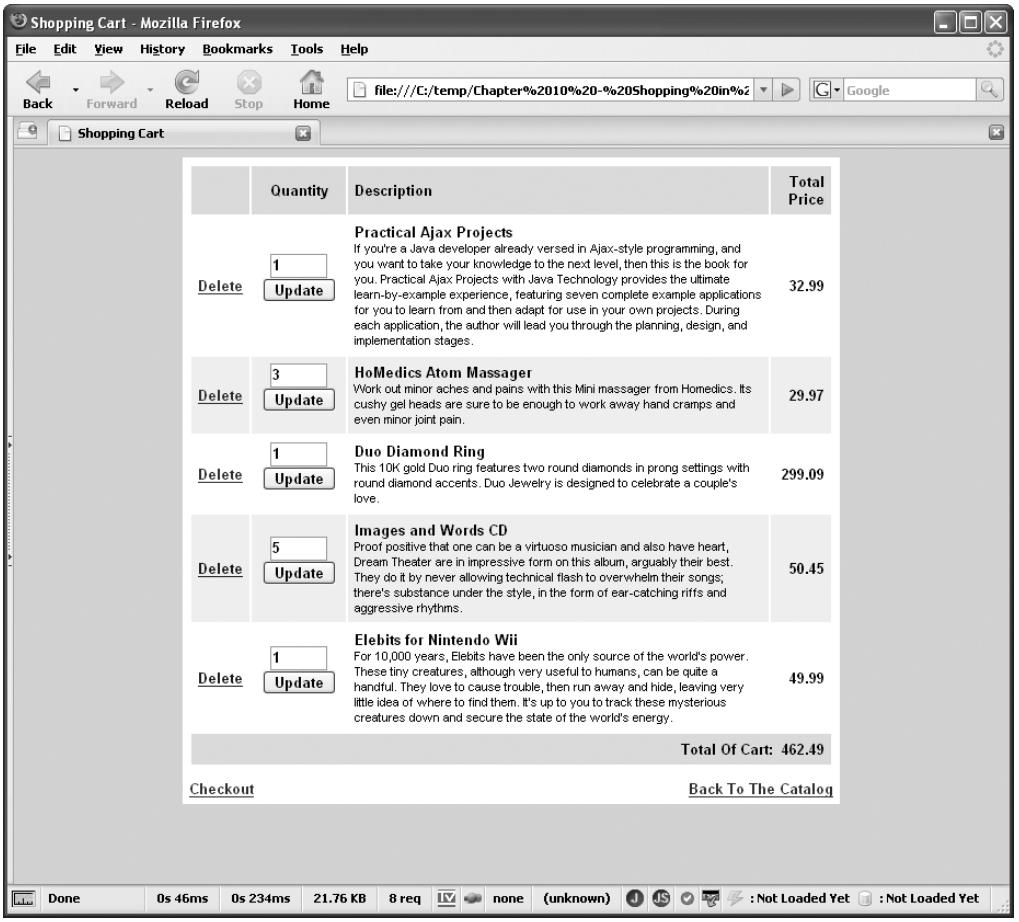


Figure 10-12. *Viewing the contents of the cart as rendered by the viewCart.htm page*

I'll call out bits of the code as needed here. As you take a look at the full source, you'll first see the style sheet import and the JavaScript imports you've seen elsewhere.

Showing the Cart's Contents

Next is some JavaScript encapsulated in a single page-scope function: `viewCart()`. This function is called `onLoad` of the page and it is where that leap of faith I mentioned comes into play. You need to pretend that this isn't really here as you conceptualize this code. This function is actually something that would be done server side, but since we have no server to work with, it's here.

This function is responsible for rendering the markup based on the contents of the cart. Although it's a fair volume of code, it's also fairly simple, consisting of just a series of string concatenations by and large.

`viewCart()` begins with a call to `getCartItems()` on the `cart` object. Remember that this returns the collection of `CartItems` in the cart. It then checks to be sure this array has a length other than zero. If it doesn't, it just renders a quick little message:

```
// No items in cart, that's easy!
document.getElementById("divCartContents").innerHTML =
  "<center><br>Your cart is empty<br></center>";
```

As you can see, this message is inserted into the `divCartContents` `<div>`, and that's all the user sees in this case.

If the cart isn't empty, then `viewCart()` begins to construct the markup for the cart contents display:

```
var htmlOut = "<table width=\"100%\" border=\"0\" ";
htmlOut += "cellpadding=\"6\" cellspacing=\"2\"";
htmlOut += "<tr class=\"cssHeaderFooter\">";
htmlOut += "<td align=\"center\"> </td>";
htmlOut += "<td align=\"center\">Quantity</td>";
htmlOut += "<td>Description</td>";
htmlOut += "<td align=\"right\">Total Price</td></tr>";
var rowStrip = false;
var cartTotal = 0;
```

Also notice the two variables at the end: `rowStrip` and `cartTotal`. `rowStrip` is used to have alternate rows in the table a different color, giving the striped effect that is common in tabular displays. `cartTotal` is the accumulation of the dollar amount of all the items in the cart.

Constructing the Markup That Displays the Cart Contents

Next, we begin to iterate over the array of `CartItem` objects. For each, we get its `itemID` and quantity. We then get the corresponding `CartItem` for it via a call to `catalog.getItem(nextItemID)`, where `nextItemID` is the `itemID` of the `CartItem`. With those two objects in hand, it's a simple matter of constructing some HTML:

```
htmlOut += "<tr";
// If this row should be striped, apply the appropriate class.
if (rowStrip) {
  htmlOut += " class=\"cssStripRow\"";
}
rowStrip = !rowStrip;
htmlOut += ">";
// Now just generate some straightforward markup.
htmlOut += "<td align=\"center\">";
htmlOut += "<a href=\"mockServer.htm?function=delete&" +
  "itemIndex=" + i + "\">Delete</a>";
htmlOut += "</td>";
htmlOut += "<td align=\"center\">";
htmlOut += "<form name=\"updateQuantity\" method=\"get\" " +
  "action=\"mockServer.htm\">";
htmlOut += "<input type=\"hidden\" name=\"function\" " +
  "value=\"updateQuantity\">";
htmlOut += "<input type=\"hidden\" name=\"itemIndex\" value=\"" +
  i + "\">";
```

```
htmlOut += "<input type=\"text\" size=\"3\" maxlength=\"2\" " +
  "name=\"quantity\" value=\"" + nextItemQuantity + "\">";
htmlOut += "<input type=\"submit\" value=\"Update\">";
htmlOut += "</form>";
htmlOut += "</td>";
htmlOut += "<td>" + catalogItem.getItemTitle() + "<br>";
htmlOut += "<div class=\"cssSmallDescription\">";
htmlOut += catalogItem.getItemDescription() + "</div></td>";
// Now some math: the total dollar amount has to be rounded for
// proper display.  The basic logic harkens back to pre-algebra:
// * Multiply the number by 10^x
// * Apply Math.round() to the result
// * Divide the result by 10^x
var itemTotalAmount = nextItemQuantity * catalogItem.getItemPrice();
itemTotalAmount = Math.round(itemTotalAmount * 100) / 100;
htmlOut += "<td align=\"right\">" + itemTotalAmount + "</td>";
htmlOut += "</tr>";
// Add cart amount to cart total.
cartTotal += nextItemQuantity * catalogItem.getItemPrice();
```

You can see the usage of the `rowStrip` variable here. Its value is inverted with each iteration, to alternate the style class applied to each row. The other interesting thing here is the calculation of the dollar amount for each item. This is just some simple multiplication: the quantity of the item times the price per unit. However, we need to do the same rounding you saw in the `Cart` class; otherwise, we might have a wild number of decimal places! You can also see where the value is added to `cartTotal` at the end.

Showing the Cart Total

Speaking of `cartTotal`, once this loop completes, we have one thing left to do, and that is to render the footer of the table where we can see the total of the cart:

```
htmlOut += "<tr class=\"cssHeaderFooter\">";
// Now some math: the total dollar amount has to be rounded for proper
// display.  The basic logic harkens back to pre-algebra:
// * Multiply the number by 10^x
// * Apply Math.round() to the result
// * Divide the result by 10^x
cartTotal = Math.round(cartTotal * 100) / 100;
htmlOut += "<td align=\"right\" colspan=\"4\">Total Of Cart: " +
  " " + cartTotal + "</td>";
htmlOut += "</tr>";
htmlOut += "</table>";
document.getElementById("divCartContents").innerHTML = htmlOut;
```

Once again we have that math, which is becoming far too familiar. (Hint: wouldn't it be nice to have an external function you could call to round a number to a specified number of decimal places? A natural fit for `jscript.math`, don't you think?)

We have only to insert the `htmlOut` string's value into the `divCartContents` `<div>`, and we've constructed the markup for the page—lock, stock, and barrel.

As I said, it's pretty straightforward code, but again, you have to pretend it isn't really here, even though it is. It's sort of not here, since it would be done by the server, but there it is, and you should understand it, even though you have to pretend it isn't there sort of . . . sorry, too much coffee today.

Writing checkout.htm

I'm almost embarrassed to be dissecting this particular file, because it is an absolutely trivial bit of code. Still, I like completeness, so it shall be done anyway. In Figure 10-13, you can see the result of the `checkout.htm` file.

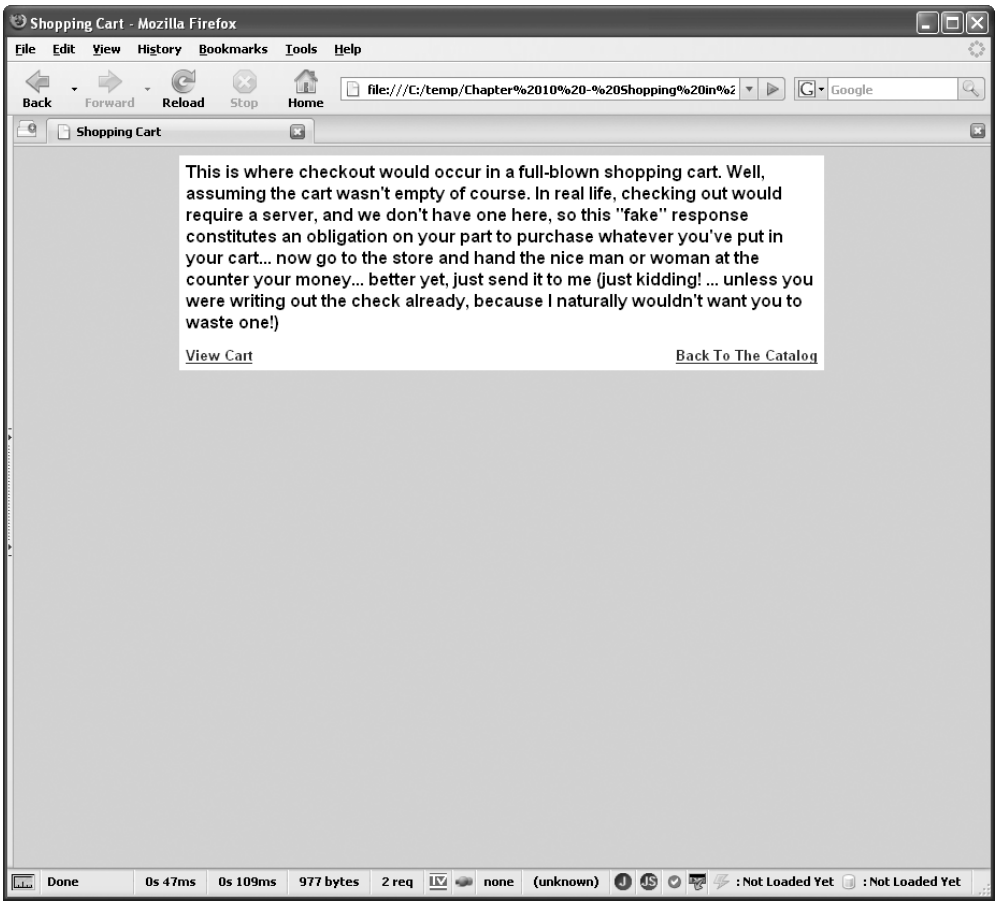


Figure 10-13. *The checkout page (not much to see I admit, but presented for the sake of completeness)*

The `checkout.htm` file is basically supposed to represent the final purchase step of the shopping cart experience. When the user clicks the Check Out link on the view cart page, the server

would be accessed and the purchase would complete, probably by getting things like credit card information, shipping instructions, and so forth from the user. Since we're not dealing with the server side, in its place, we have this page. It's nothing but my lame attempt at humor and a placeholder for a full-blown server process.

The code for this page is shown in Listing 10-8.

Listing 10-8. *The checkout.htm File—To Say It's Not Rocket Science Would Be an Understatement!*

```
<html>

  <head>

    <title>Shopping Cart</title>

    <link rel="StyleSheet" href="css/styles.css" type="text/css">

  </head>

  <body class="cssBody">

    <table border="0" cellpadding="6" cellspacing="0" width="600"
      align="center" class="cssInstructionsTable">
      <tr><td colspan="2">
        <div class="cssCheckoutText">
          This is where checkout would occur in a full-blown shopping cart.
          Well, assuming the cart wasn't empty of course.  In real life,
          checking out would require a server, and we don't have one here, so
          this "fake" response constitutes an obligation on your part to
          purchase whatever you've put in your cart... now go to the store and
          hand the nice man or woman at the counter your money... better yet,
          just send it to me (just kidding! ... unless you were writing out the
          check already, because I naturally wouldn't want you to waste one!)
        </div>
      </td></tr>
      <tr>
        <td>
          <a href="mockServer.htm?function=viewCart">View Cart</a>
        </td>
        <td align="right">
          <a href="mockServer.htm?function=viewCatalog">Back To The Catalog</a>
        </td>
      </tr>
    </table>

  </body>

</html>
```

It is quite literally nothing but straight HTML. At the bottom, you see two links that reference the mock server: one to return to the view cart page and one to return to the catalog view page. Aside from those links, there's nothing special about this page.

Only one part of the application remains to explore, but it is a key piece: our mock server. Don't change the channel now!

Writing mockServer.htm

After our long journey, we finally arrive at the final piece of the puzzle: our mock server, contained in `mockServer.htm`. As mentioned earlier, the idea of a mock server is to have a simple HTML page that all your application's requests target, and have this page pretend to be a server. You saw a simple example of how to accomplish this, and in reality, the full-blown `mockServer.htm` doesn't do a whole lot more.

After some initial JavaScript imports, we hit upon the key to making it all work, which is the `process()` function:

```
function process() {

  var func = jscript.page.getParameter("function");
  if (func) {
    func = "process" + func.substr(0, 1).toUpperCase() + func.substr(1);
    if (eval("window." + func)) {
      eval(func + "();");
    } else {
      alert("Unimplemented function received")
    }
  }

} // End process().
```

This function is called `onLoad` of the page. It is what drives everything, and you can view this as you would the component on the server that decides which operation to perform (a FrontServlet in Java parlance, for instance, or a switching ASP page in the Microsoft world). It grabs the `function` request parameter and switches on it, calling the appropriate function to service the function that was requested by the call. It just takes in the name of the function via the `"function"` request parameter, and then forms the function name to call by prepending the string `"process"` to the function, with the first letter of the function converted to uppercase. So, for the function `viewDescription`, for instance, we would wind up with `processViewDescription`, which as you're about to see, is one of the functions present on this page. Once we have the name of the function, we check to see if it exists as a child of the `window` object, and if so, we use the `eval()` function to execute it. If an unknown function is received, we just pop up an alert saying so. It's not the most robust mechanism imaginable to be sure, but sufficient for our purposes.

The functions that `process()` calls serve to process a particular service request, beginning with `processViewDescription()`. This is called when the Description link is clicked when in non-JavaScript mode. It does a simple redirect to the appropriate page in `/descs`:

```
function processViewDescription() {

  var itemID = jscript.page.getParameter("itemID");
  window.location = "descs/id" + itemID + ".htm";

} // End processViewDescription().
```

In this case, the called function would have passed an `itemID` parameter as well, which `jscript.page.getParameter()` pleasantly gets for us with no fuss. Since the pages in `/descs` are named `id`*X*`.htm`, where *X* is the `itemID`, it's a simple matter to construct the appropriate URL and redirect to it via setting `window.location`.

Next up is `processPurchase()`, which is only marginally more complex and is what is called when the user clicks the Add To Cart button on the item description page when in non-JavaScript mode:

```
function processPurchase() {

  // Add new item.
  var newItemID = jscript.page.getParameter("itemID");
  var newItemQuantity = jscript.page.getParameter("quantity");
  var itemToAdd = new CartItem();
  itemToAdd.setItemID(newItemID);
  itemToAdd.setQuantity(parseInt(newItemQuantity));
  cart.addItem(itemToAdd);

  window.location = "viewCart.htm";

} // End processPurchase().
```

Once again, we get the `itemID` parameter, and also the `quantity` parameter in this case. We then instantiate a new `CartItem` object and set the `itemID` and `quantity` on it. Then it's a simple matter of calling `cart.addItem()` and passing it the `CartItem`. That takes care of rewriting the cookie as well. Then we redirect back to `viewCart.htm` so the added item will be reflected to the user.

`processUpdateQuanitity()` is found next, and it is used when the user updates the quantity of an item from the view cart page while in non-JavaScript mode:

```
function processUpdateQuantity() {

  var itemIndex = jscript.page.getParameter("itemIndex");
  var newQuantity = jscript.page.getParameter("quantity");
  if (newQuantity == 0) {
    processDelete();
  } else if (newQuantity > 0) {
    cart.updateQuantity(itemIndex, newQuantity);
  }

  window.location = "viewCart.htm";

} // End processUpdateQuantity().
```

It begins similarly to `processPurchaseItem()`, but then gets a little different. First, it checks if the user entered zero for the quantity. If so, `processDelete()` is called to remove the item from the cart. If the value is greater than zero though, all we need to do is call the `updateQuantity()` method of the `cart` object, passing it the index of the item in the `cartItems` array (which is one of the hidden form fields rendered in the `viewPage.htm` file) and the new quantity requested, and we're all set. Note that if the user enters a nonnumeric value, the quantity will not be updated, but no error will occur either. Entering something like 12A is a little more interesting: it will register as 12, which is better than an error!

Speaking of `processDelete()`:

```
function processDelete() {

  var itemIndex = jscript.page.getParameter("itemIndex");
  cart.deleteItem(itemIndex);

  window.location = "viewCart.htm";

} // End processDelete().
```

All we need here is the `itemIndex` parameter, and we hand that off to `cart.deleteItem()`, and thy will be done! A quick redirect to `viewCart.htm`, and we're finished.

Only three functions remain, and they are basically the same, so I will discuss them as one here. These three functions—`processViewCart()`, `processViewCatalog()`, and `processCheckout()`—are nothing but simple redirects to HTML pages, like so:

```
function processViewCart() {

  window.location = "viewCart.htm";

} // End processViewCart().
```

The reason these exit, rather than simply directing to the final HTML pages, is that again, this is all meant to emulate a server. If we wanted to view the contents of the cart, we may be able to jump directly to some page—JSP, ASP, PHP, and so on. But many times, we need to hit some server component first, because JSP, ASP, and PHP are view components, and there is typically more to a modern web application. For example, the application will have Control and Model layers if it uses the Model-View-Controller architecture.[5] So, if we're going to play the part of the server, we need to do it fully. That means that, even though its just a simple redirect, the server would be doing more, such as rendering the contents of the page in the case of `viewCart.htm`.

Whew, that's it. We've finished! I hope you've enjoyed the ride. I think the whole mock server technique is something really valuable that can save you a lot of time and headaches

5. The Model-View-Controller (MVC) architecture is a way to implement an application where there is a separation, or decoupling, of the components that render the view a user sees (the View layer) and the data it uses (the Model layer), and also the business logic that operates on it (typically, part of the Model as well). This is accomplished by the view never directly interacting with the model, instead going through an intermediary (Control) layer. The goal is to allow changing any of these components without necessarily affecting the others.

during development. I hope you've also enjoyed seeing just a little of MochiKit and how truly easy it can make things.

And by the way, you have a useful little shopping cart to boot! Implement the final checkout stage, and you should be good to go.

Suggested Exercises

The application presented in this chapter isn't likely to be the subject of anyone's doctoral thesis any time soon, since it's just not all that complex, but it does most of the things a shopping cart should. That being said, here are a few improvements you could make to gain some more experience:

Round those corners: MochiKit offers an effect that rounds the corners of elements, and it would be great to apply this to all the square boxes (the white areas) on all the pages. I purposely left this for you as a suggestion, so you would need to delve into the MochiKit docs a bit. I will tell you that it isn't quite as easy as it seems and may require further changes to the application to implement. (Hint: does that effect work on tables, I wonder?).

Implement the server side: Yes, this book is focused on the client side, and yes, this particular application goes out of its way to avoid the server part of the equation. But it couldn't hurt to set up the server side in your technology of choice.

Add some effects: This is again a good way to get more familiar with MochiKit. How about the item you drop on the cart shrinking into it? How about when you delete an item from your cart, it first fades out of view and collapses the table before submitting to the server?

I'm sure you can think of plenty more, but those should give you a good start. Have fun!

Summary

In this chapter, you got your first taste of MochiKit, a very nice little JavaScript library. You saw how its drag-and-drop support is very powerful but yet still very simple to use and allows you to have very little code that does quite a bit. You also saw how to create a mock server that allows you to do all your development strictly in the browser while not having to change your methodology or code from what you would do with a server in the mix. Finally, you also saw how you can take a fairly mundane application like a shopping cart and spiff it up just a bit for those who want the next-generation web experience.

CHAPTER 11

Time for a Break: A JavaScript Game

In the first book I ever had published (*Practical Ajax Projects with Java Technology*, Apress, 2006), the final project—the apex of the book—was an adventure game named Ajax Warrior. It may well be the start of a trend, where every book I write includes a game, because that's precisely what we're going to build in this chapter! No, it won't be Ajax Warrior again. It will be a more arcade-style game, since there is no network latency to bother us.

You'll see many neat tricks here, a lot of JavaScript and DOM scripting, and even some basic game theory along the way. At the end, you'll have something that you can use to slack off at work any time you wish, or anywhere else you have a browser, for that matter! We all know the saying . . . all work and no play makes Homer . . . something . . . something,[1] so let's stop the axe from falling, shall we?

K&G Arcade Requirements and Goals

The game we'll build is a port of a PocketPC game that I wrote entitled K&G Arcade. The K&G stands for Krelmac and Gentoo, who are two wisecracking aliens bent on the destruction of the Earth. Unfortunately, they are like idiot teenagers, who just happen to have quantum destructo beams!

In K&G Arcade, which you can see at `http://www.omnytex.com/kgarcade`, you play the part of Henry, a mild-mannered farmer from jolly-old seventeenth-century England. Krelmac and Gentoo abduct you one night, and force you to try to escape their spaceship, which consists of five maze-like levels inhabited by teleporting robots that kill on contact. On each of those five levels, you find five mini-games each, which you need to play and beat (by achieving a given score in 60 seconds) in order to escape. You also meet up with other abductees, who you talk to and try to gain their trust so that they will give you clues about certain mini-games that are impossible to beat without a particular trick.

The full-blown version of K&G Arcade features cinematic cut scenes with Krelmac and Gentoo cracking wise and generally making pests of themselves. It includes an all-original soundtrack and hand-drawn cartoon graphics. K&G Arcade is actually the second game featuring

1. If you are a Simpsons fan, you almost certainly know the reference and are laughing right now. If you aren't, it's a line from the episode "Treehouse of Horror V" in the segment entitled "The Shinning," a parody of *The Shining*. I suggest grabbing a copy—it's a riot!

these characters, the first being Invasion: Trivia! (`http://www.omnytex.com/products_invasion_info.shtml`). Going to that site will also lead you to a Flash cartoon introducing these characters.

Now, our goal isn't to port the entire full-blown K&G Arcade to JavaScript. Indeed, that would be considerably more difficult, if possible at all, and would take up a book this size on its own! Instead, we'll scale it back quite a bit and implement just the mini-game portion. In fact, we'll build only 3 of the 25 mini-games. Let's get into some details:

- We'll implement three mini-games—Cosmic Squirrel, which is similar conceptually to the classic Frogger; Deathtrap, which is inspired by the Indiana Jones movies; and Refluxive, which is similar to Arkanoid, Breakout, and games in that mold (but without actually breaking anything, as you'll see!).
- We'll implement a mini-game selection screen that includes a screenshot of the mini-game.
- We should reuse existing code wherever possible. However, we will *not* be using any libraries for this game. That's because in writing games, you frequently want to be "as close to the metal" as possible, and that's the case here as well.
- Each mini-game should be its own class, and should inherit common code from a base class.
- Extensibility should be a priority so that more mini-games can be added later with little difficulty.
- In general, we want to keep global scope as clean as possible, and use good object-oriented design techniques throughout.

When doing game programming, you often try to get as low-level as you can—as close to the hardware as you can. The reason for this is simple: performance. In a game, a lot has to happen in very short time periods, so there can't be a lot of superfluous code executing or extra work being done. One of the best ways to ensure this is to not entrust things to libraries. Now, this isn't an absolute. It is often true that you can get better performance with a well-written library than without. It's also true that in the modern era, you typically don't get as low-level as you used to in general, with or without a library. In the past, it wasn't unusual to write important portions of a game in assembly language so that it could be as optimized and tight as possible. These days, that isn't as prevalent (it's still done, but not as often). So, in this particular application, we won't be using any libraries. We'll be doing all "naked" JavaScript.

Programming a game in JavaScript isn't fundamentally different from programming a game in any other environment. Some of the details are different, of course, but the overall concept is roughly the same. Rather than espouse those concepts here in one place, I'll talk about them as we progress through the code.

And with that statement, let's begin our exploration of K&G Arcade by taking a look at the game itself.

A Preview of the K&G Arcade

Figure 11-1 shows the game title screen, which is what you see when you first load K&G Arcade into your browser. The original K&G Arcade was a joint production of Omnytex Technologies, which is my own little PocketPC software company, and Crackhead Creations (`http://www.planetvolpe.com/crackhead`), which is the company of Anthony Volpe, the artist responsible for the illustrations in this book,[2] hence the logos on this screen. Anthony is also the artist who did the graphics for K&G Arcade, as well as Invasion: Trivia!.

Figure 11-1. *K&G Arcade title screen*

The game selection screen, shown in Figure 11-2 is what you see after the title screen. It is where you can select a mini-game to play. It also presents a few instructions, as well as a preview of the mini-game and a brief description of it.

Cosmic Squirrel, shown in Figure 11-3, is one of the mini-games available in K&G Arcade. This game is inspired by the classic Frogger, but with a twist: you play the part of a giant space squirrel trying to get an intergalactic acorn (not that I know what an "intergalactic acorn" actually is!). The player needs to avoid aliens, asteroids, spaceships, and comets to get the acorn.

2. If you would like to see some more of Anthony Volpe's work, and some other things I have done with him with Krelmac and Gentoo, as well as some other characters, have a look at the Downtown Uptown site: `http://www.planetvolpe.com/du`. There, you'll find some more adventures of Krelmac and Gentoo in the form of a Flash cartoon and some comics, as well as a host of other characters from this universe. Don't let the strangeness of it all scare you away! Embrace the weirdness!

Figure 11-2. *Game selection screen*

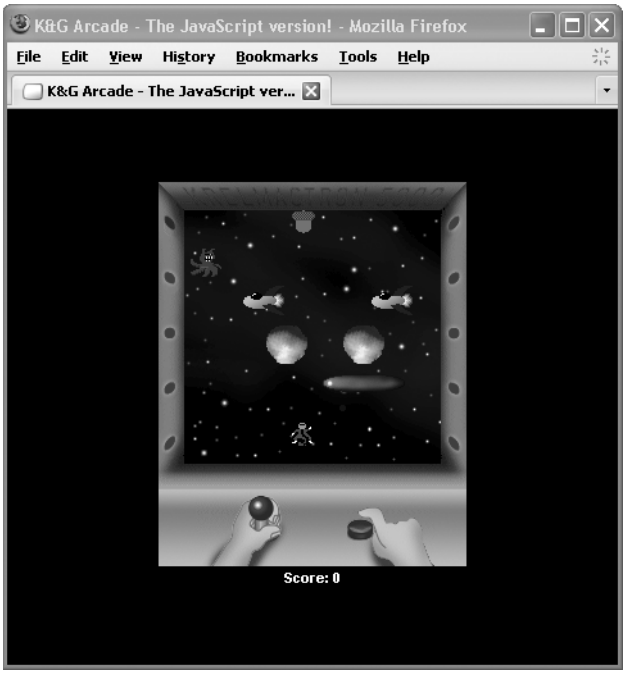


Figure 11-3. *Cosmic Squirrel*

Figure 11-4 shows the mini-game Deathtrap, which is inspired by Indiana Jones movies. Your goal is to get to the door on the top of the screen by hopping from tile to tile. The problem is that some of the tiles are electrified, and you will get zapped if you pick the wrong one.

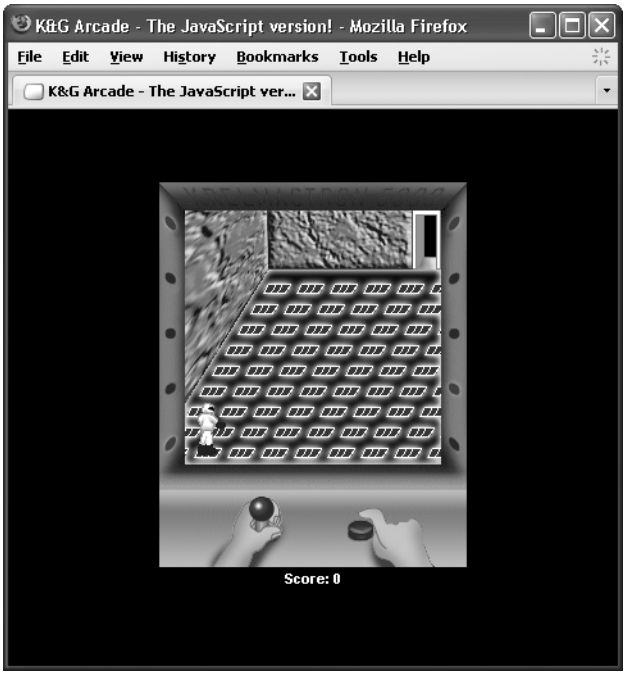


Figure 11-4. *Deathtrap*

Finally, in Figure 11-5, you see the third mini-game, Refluxive. This is similar in concept to Arkanoid or Breakout, but without the element of actually breaking through anything! Actually, come to think of it, this game is much more like the movie *Speed*. Remember that Sandra Bullock and Keanu Reeves mess, where they couldn't let the bus go below a certain speed lest it be blown to kingdom come? Well, this is similar. Someone told you to keep these bouncy things going, and you do it—no questions asked!

Now that you're familiar with what the game looks like, let's get into seeing what makes it tick. Buckle up, because it's going to be quite a ride!

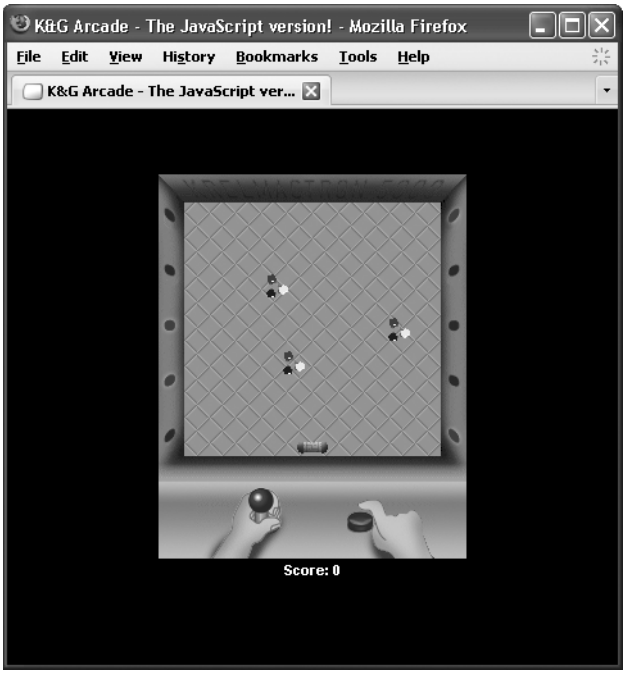


Figure 11-5. *Refluxive*

Dissecting the K&G Arcade Solution

As usual, we begin our exploration of this project by looking at its directory structure, shown in Figure 11-6.

Beginning with the root directory, we find the `index.htm` file, which is the page loaded to start the application. It contains the basic markup for the screen that the player sees, as well as all the JavaScript imports required.

The `img` directory contains images not specific to any one mini-game, such as the graphics for the title screen, game selection screen, and game console.

The `js` directory contains all our JavaScript source files, 11 of them in all. Two of them—`jscript.math.js` and `jscript.dom.js`—are packages we created in Chapter 3. Four of them—`MiniGame.js`, `Title.js`, `GameSelection.js`, and `GameState.js`—define classes that will be used. The remaining five—`main.js`, `keyHandlers.js`, `globals.js`, `gameFuncs.js`, and `consoleFuncs.js`—contain the code that makes use of those classes.

Each of our mini-games is stored in its own subdirectory, which has the same name as the mini-game itself. Within each of those subdirectories is a single `.js` file, such as `CosmicSquirrel.js`, with the code for that particular mini-game. Each of those subdirectories also contains an `img` subdirectory, which houses the images specific to that mini-game.

Now, let's get to looking at that code, shall we?

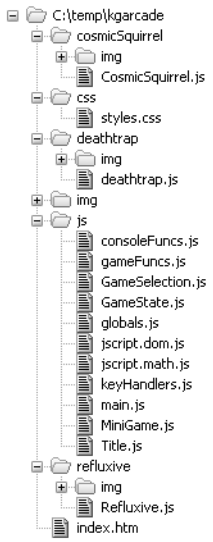


Figure 11-6. *K&G Arcade directory structure*

Writing index.htm

`index.htm` is the first page loaded when we access the game, and it defines the overall layout of things. It also "imports" all the other resources we need throughout the game. Let's begin by looking at the `<head>` of the page:

```
<head>

  <title>K&G Arcade - The JavaScript version!</title>

  <link rel="StyleSheet" href="css/styles.css" type="text/css">

  <script src="js/jscript.dom.js"></script>
  <script src="js/jscript.math.js"></script>
  <script src="js/gameFuncs.js"></script>
  <script src="js/consoleFuncs.js"></script>
  <script src="js/keyHandlers.js"></script>
  <script src="js/globals.js"></script>
  <script src="js/GameState.js"></script>
  <script src="js/MiniGame.js"></script>
  <script src="js/Title.js"></script>
  <script src="js/GameSelection.js"></script>
  <script src="js/main.js"></script>
  <script src="cosmicSquirrel/CosmicSquirrel.js"></script>
  <script src="deathtrap/deathtrap.js"></script>
  <script src="refluxive/refluxive.js"></script>

</head>
```

You see that first our style sheet (we'll look at that next) is linked in. After that comes a whole batch of JavaScript references. Since the process of dissecting this application will lead us to explore each of these in turn, it would be a bit redundant to state what each is at this juncture. Suffice it to say they are required to make everything work.

The body of the document then begins, and `onLoad` we see a call to a JavaScript function named `init()`. This will initialize the application and get everything set up for us to play. This function can be found in `main.js`, so we'll get to that shortly.

After the opening `<body>` tag is the following section of markup:

```
<!-- The div the title screen is contained in. -->
<div id="divTitle" class="cssTitle">
  <table border="0" cellpadding="0" cellspacing="0" width="98%"
    height="100%" align="center">
    <tr>
      <td align="center" valign="middle">
        <img src="img/title.gif">
        <br/><br/>
        The JavaScript Version, v1.0
        <br/><br/>
        Ported from the original PocketPC version, presented by:
        <br/><br/>
        <img src="img/logoOmnytex.gif"><img src="img/logoCrackhead.gif">
        <br/><br/>
        Press Any Key To Play
      </td>
    </tr>
  </table>
</div>
```

This is pretty much straightforward HTML markup, which renders the title screen shown earlier in Figure 11-1. The `divTitle <div>` will be hidden once the user presses a key to move on to the game selection screen, which brings us to the block of markup that comes next:

```
<!-- The div the game selection screen is contained in. -->
<div id="divGameSelection" class="cssTitleGameSelection">
  <table border="0" cellpadding="0" cellspacing="0" width="98%"
    height="100%" align="center">
    <tr>
      <td align="center" valign="middle">
        Press the LEFT and RIGHT arrow keys to cycle through the
        available games, then press SPACE to play the one you want.
        Once playing a game, press the ENTER key to return here.
        <br/><br/><br/>
        <img src="img/ssCosmicSquirrel.gif" id="ssCosmicSquirrel"
          style="display:none;">
        <img src="img/ssDeathtrap.gif" id="ssDeathtrap"
          style="display:none;">
        <img src="img/ssRefluxive.gif" id="ssRefluxive"
```

```
          style="display:none;">
        <br/>
        <div id="mgsDesc"></div>
        <br/><br/>
        To check out the full PocketPC version of K&G Arcade,
        visit the <a href="http://www.omnytex.com">Omnytex Technologies</a>
        web site.
      </td>
    </tr>
  </table>
</div>
```

Once again, this is simple, almost boring HTML. Note the screenshot images in the middle, which are initially hidden. When the user is cycling through the screenshots to pick a game, these images will be made visible with each arrow keypress. The `mgsDesc <div>` is where the description of the game will be shown.

When the `init()` function I mentioned earlier fires, one of the things it does is to center both the `divTitle <div>` and the `divGameSelection <div>`. It also centers the `<div>` that contains the actual mini-games, which is named `divMiniGame`, and can be seen here:

```
<!-- The div the game is contained in. -->
<div id="divMiniGame" class="cssMiniGame">

  <div id="divGameArea" class="cssGameArea">
  </div>

  <div id="divStatusArea" class="cssStatusArea">
    Score: 0
  </div>

  <!-- Game frame -->
  <img src="img/gameFrame.gif" id="imgGameFrame"
    class="cssConsoleImage">

  <!-- Console left, middle and right -->
  <img src="img/consoleLeft.gif" id="imgConsoleLeft"
    class="cssConsoleImage">
  <img src="img/consoleMiddle.gif" id="imgConsoleMiddle"
    class="cssConsoleImage">
  <img src="img/consoleRight.gif" id="imgConsoleRight"
    class="cssConsoleImage">

  <!-- Left hand images -->
  <img src="img/leftHandNormal.gif" id="imgLeftHandNormal"
    class="cssConsoleImage">
  <img src="img/leftHandUp.gif" id="imgLeftHandUp"
    class="cssConsoleImage">
```

```
    <img src="img/leftHandDown.gif" id="imgLeftHandDown"
      class="cssConsoleImage">
    <img src="img/leftHandLeft.gif" id="imgLeftHandLeft"
      class="cssConsoleImage">
    <img src="img/leftHandRight.gif" id="imgLeftHandRight"
      class="cssConsoleImage">
    <img src="img/leftHandDL.gif" id="imgLeftHandDL"
      class="cssConsoleImage">
    <img src="img/leftHandDR.gif" id="imgLeftHandDR"
      class="cssConsoleImage">
    <img src="img/leftHandUL.gif" id="imgLeftHandUL"
      class="cssConsoleImage">
    <img src="img/leftHandUR.gif" id="imgLeftHandUR"
      class="cssConsoleImage">

    <!-- Right hand images -->
    <img src="img/rightHandUp.gif" id="imgRightHandUp"
      class="cssConsoleImage">
    <img src="img/rightHandDown.gif" id="imgRightHandDown"
      class="cssConsoleImage">

    <!-- Console light images -->
    <img src="img/gameFrameLeftLight1.gif" id="imgGameFrameLeftLight1"
      class="cssConsoleImage">
    <img src="img/gameFrameLeftLight2.gif" id="imgGameFrameLeftLight2"
      class="cssConsoleImage">
    <img src="img/gameFrameLeftLight3.gif" id="imgGameFrameLeftLight3"
      class="cssConsoleImage">
    <img src="img/gameFrameLeftLight4.gif" id="imgGameFrameLeftLight4"
      class="cssConsoleImage">
    <img src="img/gameFrameLeftLight5.gif" id="imgGameFrameLeftLight5"
      class="cssConsoleImage">
    <img src="img/gameFrameRightLight1.gif" id="imgGameFrameRightLight1"
      class="cssConsoleImage">
    <img src="img/gameFrameRightLight2.gif" id="imgGameFrameRightLight2"
      class="cssConsoleImage">
    <img src="img/gameFrameRightLight3.gif" id="imgGameFrameRightLight3"
      class="cssConsoleImage">
    <img src="img/gameFrameRightLight4.gif" id="imgGameFrameRightLight4"
      class="cssConsoleImage">
    <img src="img/gameFrameRightLight5.gif" id="imgGameFrameRightLight5"
      class="cssConsoleImage">

  </div>
```

Perhaps the most important element here is `divGameArea`. This is the screen portion of the game console where the mini-games take place. A bit of explanation may help here.

All of these images are the ones that make up the game console—the joystick, frame around the mini-game, lights, and so on. These images do not change from mini-game to mini-game, and that's why they are static here. When the player moves while playing a mini-game, the left hand on the joystick moves accordingly. When the user clicks the action button, the right hand presses the button, too. Every half second, the lights on the frame randomly change. All this animation is accomplished by hiding the images that change, and then showing the appropriate new image.

For instance, let's talk about the action button. When we start, the image of the button not being pushed—the one with the ID `imgRightHandUp`—is showing, and the image of the button being pushed—the one with the ID `imgRightHandDown`—is not. When the user clicks the button, `imgRightHandUp` is first hidden, and then `imgRightHandDown` is shown. This is a simple case, but it is how *all* the mini-games work as well, as you will see.

Writing styles.css

All of the markup in `index.htm` wouldn't amount to a hill of beans without the style sheet in `styles.css` to back it up, so let's get familiar with that. The complete file is shown in Listing 11-1.

Listing 11-1. *The styles.css File—The Main Style Sheet for the Game*

```
/* Generic style applied to all elements. */
* {
  color            : #ffffff;
  font-family      : arial;
  font-size        : 8pt;
  font-weight      : bold;
}


/* Entire page (body element). */
.cssPage {
  background-color : #000000;
}


/* Style for div the title screen and game selection screens are */
/* contained in. */
.cssTitleGameSelection {
  border           : 1px solid #ffffff;
  position         : absolute;
  width            : 240px;
  height           : 320px;
}
```

```
/* Style for div the mini-games are contained in. */
.cssMiniGame {
  border           : 1px solid #000000;
  position         : absolute;
  width            : 240px;
  height           : 320px;
}


/* Style for the area where a mini-game takes place. */
.cssGameArea {
  position         : absolute;
  left             : 20px;
  top              : 20px;
  width            : 200px;
  height           : 200px;
  overflow         : hidden;
}

/* Style for the status area below the game console. */
.cssStatusArea {
  position         : absolute;
  left             : 0px;
  top              : 302px;
  width            : 240px;
  height           : 20px;
  text-align       : center;
}


/* Style applied to all game console images. */
.cssConsoleImage {
  position         : absolute;
  display          : none;
}
```

There isn't really much to this style sheet when you get right down to it. First, you see that generic "cover everything" selector that you've encountered in the previous projects. It deals with just font styles, but it's nice to cover everything in one clean move.

After that is `cssPage`, whose only purpose in life is to give the page a black background.

The next style, `cssTitleGameSelection`, is applied to the title screen `div` (`divTitle`) and the game selection screen `div` (`divGameSelection`). While the name of the selector is perhaps a little long, it pretty clearly describes what it's for, no? It ensures that these `<div>` elements can be centered by virtue of the `position` attribute being set to `absolute`. It also draws a border around the `<div>` elements. Lastly, it sets the size. Note that the size 240-by-320 pixels is not arbitrary;

this is Quarter VGA (QVGA) resolution, which was, at the time the full-blown version of K&G Arcade was written, the default resolution of most PocketPC devices. Since all the graphics were scaled for that resolution, that's the resolution used in the JavaScript version here as well.

The `cssMiniGame` style is applied to the `divMiniGame <div>` element. Notice it is identical to the `cssTitleGameSelection` style, except for the border color. Setting the border color to black means it will be invisible on the black page background, so, in effect, the game console won't have the white border, but everything will take up the same amount of space, thereby avoiding the game console seeming to move, or jump, relative to the game selection screen.

The `cssGameArea` style is probably the most important. One of the things the mini-games need to be able to do, as can be seen in the Cosmic Squirrel mini-game, is *clip*. In other words, when an object moves to the edges, it should appear to move off screen, not overlap the frame or anything. To accomplish this, we set the `overflow` attribute to `hidden`. This means that any content that is positioned out of the bounds of the `<div>` will be hidden, or clipped, precisely as we want. As you can see, the actual game area is 200-by-200 pixels, and is positioned 20 pixels from the `left` and `top`, placing it inside the frame, as expected, completing the illusion of clipping when objects move out of its bounds.

After that is the `cssStatusArea` style. I'm sure you can guess this is applied to the `divStatusArea <div>` element, which is where you see the score below the game console. All text is horizontally centered within this `<div>` by setting the `text-align` attribute to `center`.

Lastly, you see the `cssConsoleImage` style, which is applied to all of the images that make up the game console. Its main purpose is again to ensure that the image it is applied to can be positioned absolutely and that it initially is hidden. The point about absolute positioning will become clear when we look at the `blit()` function later on.

Writing GameState.js

You will see the `GameState` class used quite a bit throughout the code, so it makes sense to look at it first. Its purpose is . . . wait for it . . . to store information about the current state of the game. Figure 11-7 shows the UML diagram of this class.

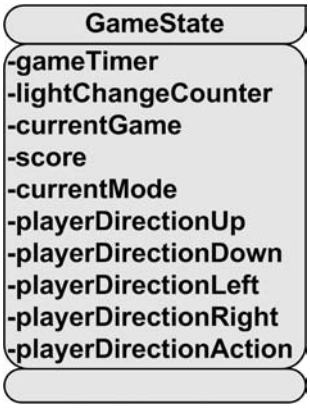


Figure 11-7. *UML diagram of the GameState class*

The GameState class includes the following fields:

- gameTimer: A reference to the JavaScript timer used as the "heartbeat" of the game.
- lightChangeCounter: Used to determine when it's time to update the lights on the game console frame.
- currentGame: A reference to the current mini-game object (as well as the title screen and game selection screens, which are essentially mini-games as far as the rest of the code is concerned).
- score: This field's purpose is abundantly obvious, I think!
- currentMode: Determines if a mini-game is in play.
- playerDirection*XXX*: Five fields—playerDirectionUp, playerDirectionDown, playerDirectionLeft, playerDirectionRight, and playerAction—used to determine in which direction the player is currently moving, and if the action button is clicked.

Listing 11-2 shows this class in its entirety.

Listing 11-2. *The GameState Class*

```
function GameState() {

  // The main timer, 24 frames a second.
  this.gameTimer = null;

  // Count of how many frames have elapsed since the lights last changed.
  this.lightChangeCounter = null;

  // This is essentially a pointer to the current game.  Note that the term
  // "game" is a little loose here because the title screen and the game
  // selection screen are also "games" as far as the code is concerned.
  this.currentGame = new Title();
  this.score = 0;

  // Mode the game is currently in: "title", "gameSelection" or "miniGame".
  this.currentMode = null;

  // Flag variables for player movement.
  this.playerDirectionUp = false;
  this.playerDirectionDown = false;
  this.playerDirectionRight = false;
  this.playerDirectionLeft = false;
  this.playerAction = false;

} // End GameState class.
```

Writing globals.js

In keeping with the theme throughout this book of not polluting the global namespace, you'll see just a small handful of values in the `globals.js` file, shown in Listing 11-3.

Listing 11-3. *Not a Whole Lot of Globals in This Application, But They Count!*

```
// Counter, reset to 0 to start each frame, used to set the z-index of
// each element blit()'d to the screen.
var frameZIndexCounter = 0;

// Key code constants.
var KEY_UP = 38;
var KEY_DOWN = 40;
var KEY_LEFT = 37;
var KEY_RIGHT = 39;
var KEY_SPACE = 32;
var KEY_ENTER = 13;

// Structure that stores all game state-related variables.
var gameState = null;

// This is an associative collection of all the images in the game.
// This saves us from having to go to the DOM every time to update one.
var consoleImages = new Object();
```

The `frameZIndexCounter` variable is used to ensure proper z-ordering when images are `blit()`'d, which will be discussed in the next section. Next you see a batch of pseudo-constants (remember that there are no real constants in JavaScript), which define various keys that can be pressed. We also find the `gameState` variable, which will be the reference to the one and only `GameState` object used throughout the code. Lastly, the `consoleImages` array will store references to the images making up the game console, which also will be discussed shortly.

Writing main.js

`main.js` is essentially the heart and soul of K&G Arcade. You will find that it makes use of functions found in the other JavaScript files, so you'll read "we'll get to this soon" fairly often. Rest assured, I'm not lying—we *will* get to those things soon! But understanding the basic core is what looking at `main.js` is all about, so let's get to it.

The init() Function

As you will recall from looking at `index.htm`, when the page loads, in response to the `onLoad` event, the `init()` function is called. Now it's time to see what that function is all about:

```
function init() {

  gameState = new GameState();

  // Get references to all existing images.  This is mainly for the console
  // images so that we don't have to go against the DOM to manipulate them.
  var imgs = document.getElementsByTagName("img");
  for (var i = 0; i < imgs.length; i++){
    consoleImages[imgs[i].id] = imgs[i];
  }

  // Center the three main layers.
  jscript.dom.layerCenterH(document.getElementById("divTitle"));
  jscript.dom.layerCenterV(document.getElementById("divTitle"));
  jscript.dom.layerCenterH(document.getElementById("divGameSelection"));
  jscript.dom.layerCenterV(document.getElementById("divGameSelection"));
  jscript.dom.layerCenterH(document.getElementById("divMiniGame"));
  jscript.dom.layerCenterV(document.getElementById("divMiniGame"));

  // Now hide what we don't need.
  document.getElementById("divGameSelection").style.display = "none";
  document.getElementById("divMiniGame").style.display = "none";

  // Hook event handlers.
  document.onkeydown = keyDownHandler;
  if (document.layers) {
    document.captureEvents(Event.KEYDOWN);
  }
  document.onkeyup = keyUpHandler;
  if (document.layers) {
    document.captureEvents(Event.KEYUP);
  }

  gameState.currentGame.init();

  gameState.gameTimer = setTimeout("mainGameLoop()", 42);

} // End init().
```

First you see a new `GameState` object being instantiated. Next is a hook that, as the comments state, gets a reference to all the `<img>` tags, those present in `index.htm`. We store a reference to each in the `consoleImages` arrays. This is because, when you do game programming, it is especially important (most of the time) to write code that is as efficient as possible.

KEEPING UP THE FRAMES-PER-SECOND (FPS) RATE

As you will see, a game usually (and certainly here) consists of a continuous loop. This loop calls on some code to update the display some number of times per second. Each of these redraws is called a frame, as in frame of animation. As I'm sure you know, there are 1000 milliseconds in one second. Game loops are usually measured in frames per second (fps); that is, how many times per second the display is updated. For smooth animation and game play, you want the frames per second to be as high as possible. Generally, you want it to be no lower than around 24 fps, which is the approximate speed at which the human eye cannot easily discern each frame. In other words, if you update the display ten times a second, your eye can rather easily track each frame, and the animation will appear slow and choppy. At 24 fps and higher, your eye is fooled into thinking there is continuous motion, which makes things look considerably smoother. The higher the better, but 24 fps is kind of the magic number.

So, let's do some simple math. If there are 1000 milliseconds in a second, and realizing that each frame will take some amount of time to process and draw, we can determine how many milliseconds each frame can take for a target frames per second. For 24 fps, we divide 1000 by 24, and we discover that each frame can take no more than about 42 milliseconds to fully process. If a frame takes longer to deal with, then our frames per second drop, and our game gets choppy and not visually pleasing. So, it becomes of paramount importance to not exceed 42 milliseconds, and that's why we have to think about optimization.

One of the killers, not just in game programming but in any browser-based DOM scripting, is accessing elements in the DOM. It takes time to traverse the DOM tree, find the element requested, and return a reference to it. If you do this too many times per frame in a game application, you'll quickly drop your frames-per-second rate. One of the best ways to optimize here is simply to get references to any images (or other elements) that you will need to access that you can up front and store those references. Getting an element in an array that happens to be a reference to a DOM element is considerably faster than getting the DOM element directly. A simple test can prove this:

```
<html>

  <head>

    <title>DOM/Array Access Test</title>

    <script>

      function testit() {

        // Time 1000 direct DOM accesses
        var timeStart = new Date();
        for (var i = 0; i < 5000; i++) {
          var elem = document.getElementById("myDiv");
          elem.innerHTML = i;
        }
        var directDOMTime = new Date() - timeStart;
```

```
        // Time 1000 accesses via array lookup
        var a = new Array();
        a[0] = document.getElementById("myDiv");
        timeStart = new Date();
        for (var i = 0; i < 5000; i++) {
          var elem = a[0];
          elem.innerHTML = i;
        }
        var arrayTime = new Date() - timeStart;

        // Display results
        document.getElementById("myDiv").innerHTML =
          "Time for direct DOM access: " + directDOMTime + "<br>" +
          "Time for array access: " + arrayTime;

      }

    </script>

  </head>

  <body>

    <input type="button" onClick="testit();" value="Test">
    <br/>
    <div id="myDiv"></div>

  </body>

</html>
```

Running this test, you'll find that the array access method is always faster than the direct DOM access, although I was surprised to find the difference isn't as drastic as I had expected. Still, it was generally in the 200 to 300 millisecond range each time I ran it, which is a pretty significant amount for game programming, as the math we went through earlier indicates.

Moving right along in our review of `init()`, we find six lines used to center the three main `<div>` elements: `divTitle`, `divGameSelection`, and `divMiniGame`, corresponding to the title screen, game selection screen, and the actual mini-games. To do this, we use the `jscript.dom.layerCenterH()` and `jscript.dom.layerCenterV()` functions that we built in Chapter 3. See, that code comes in handy, doesn't it? Immediately after they are centered, the game selection `<div>` and the mini-game `<div>` are hidden. This may seem a little bizarre. Why not just set `display:none` in the style applied to those `<div>` elements? The answer is that the centering will not work properly if the elements are hidden, because certain values those functions need are not set by the browser if the element is hidden.

Next are four lines of code that hook the keyDown and keyUp events so that our custom handlers will fire. Note the need to have two statements per event handler because of the difference in the event handling model of IE vs. Mozilla-based browsers. Just setting document.keydown and document.keyup is sufficient in IE. But in Firefox and its ilk, this requires the additional captureEvents() call. By checking if document.layers is defined, which it would be only in a non-IE browser, we can call captureEvents() only on browsers where it is applicable. As mentioned in previous chapters, using object-existence checks to conditionally execute code based on browser type is preferable to browser-sniffing code.

Finally, we have a call to gameState.currentGame.init(). This asks whatever the current mini-game is to initialize itself. Interestingly, the title screen and game selection screens are treated just like mini-games, even though they really aren't. The very last thing done in init() is to set a timeout to fire 42 milliseconds later (again, corresponding to our desired 24 fps) and set to call the mainGameLoop() function when it does. And with that statement, let's pause a moment to discuss the overall structure of K&G Arcade.

The Main Game Loop Flow

Figure 11-8 shows a flow diagram that depicts how it all works in terms of the main game flow, and also keyUp and keyDown event handling.

As you can see, the timeout set in init() fires 42 milliseconds later, calling mainGameLoop(). mainGameLoop() then updates the lights on the game console frame, as well as the hands, *if* a mini-game is in progress. Then it calls processFrame() on the object pointed to by gameState.currentGame. Once the mini-game's processFrame() function returns, the timeout is set again and this entire process repeats itself.

All of the mini-games, as well as the title and game selection screens, implement an interface by virtue of "extending" the MiniGame class. A mini-game can override five functions: init(), processFrame(), keyDownHandler(), keyUpHandler(), and destroy(). These represent the life cycle of a mini-game. In addition, three fields are present: gameName, gameImages, and fullKeyControl.

The init() function is called when the user decides to play that mini-game. Its job is to set up the mini-game, which primarily involves loading graphics, but can be other tasks as well.

The processFrame() function is called once per frame for the game to do its work. It is responsible for handling any game logic, as well as updating the screen.

keyDownHandler() and keyUpHandler() are called to deal with keypress and key release events.

destroy() is called when the user presses Enter to exit the mini-game. Its main job is to delete the images loaded in init(), but it can do other tasks as well.

The gameName field must match the directory in which the mini-game resources are found (each mini-game is presumed to be in its own directory off the root of the web application). The gameImages is an associative array of images loaded in init(). This serves the same purpose as the consoleImages array we briefly touched on earlier, namely to avoid direct DOM access where possible. Lastly, the fullKeyControl, when set to true, means that the mini-game is in full control of key events and will need to deal with everything.

All of these functions are implemented, but empty, in the base MiniGame class. Therefore, a mini-game needs to override only those it is interested in. Likewise, except for the gameName field, which *must* be set in init(), the fields have default values as well (fullKeyControl defaults to false, and gameImages is an empty array).

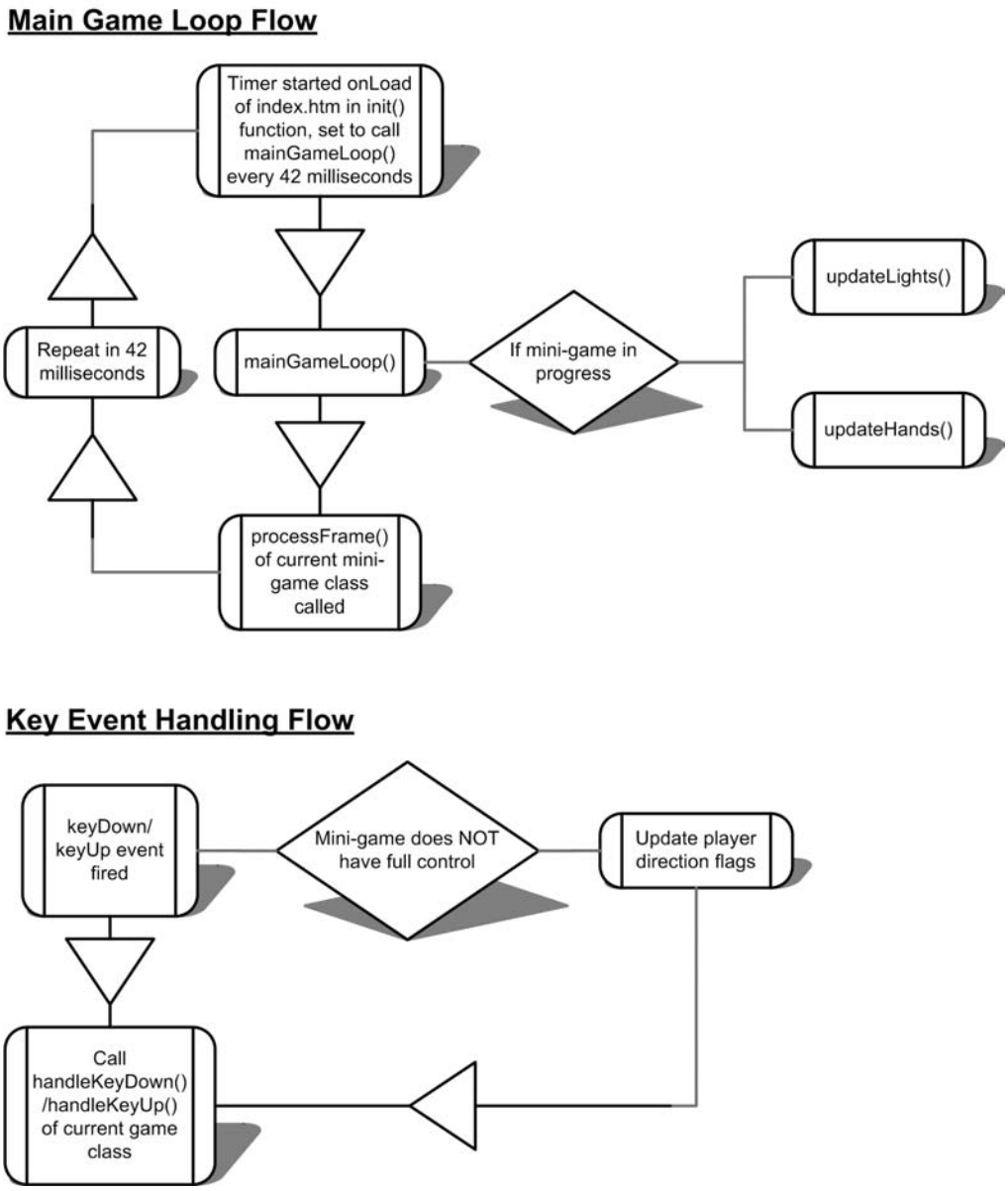


Figure 11-8. *Basic flow diagram of how K&G Arcade works, at a high level*

Starting a Mini-Game

Back in `main.js`, we find the `startMiniGame()` function:

```
function startMiniGame(inName) {

  // Reset generic game-related variables.
  gameState.playerDirectionUp = false;
  gameState.playerDirectionDown = false;
  gameState.playerDirectionRight = false;
  gameState.playerDirectionLeft = false;
  gameState.playerAction = false;
  gameState.score = 0;
  document.getElementById("divStatusArea").innerHTML = "Score: " +
    gameState.score;

  // Instantiate mini-game.
  if (inName == "cosmicSquirrel") {
    gameState.currentGame = new CosmicSquirrel();
    gameState.currentGame.init();
  } else if (inName == "deathtrap") {
    gameState.currentGame = new Deathtrap();
    gameState.currentGame.init();
  } else if (inName == "refluxive") {
    gameState.currentGame = new Refluxive();
    gameState.currentGame.init();
  }

  // Show the game and hide the game selection screen.  Set the mode to indicate
  // mini-game in progress, and draw the console.
  gameState.currentMode = "miniGame";
  drawConsole();
  document.getElementById("divGameSelection").style.display = "none";
  document.getElementById("divMiniGame").style.display = "block";

} // End startMiniGame().
```

`startMiniGame()` is called when the user presses the spacebar at the game selection screen to start the selected mini-game. It begins by resetting the five fields in `gameState` that indicate in which direction the player is currently moving, and the one that indicates whether the action button is pressed. It also resets the score to zero and updates it on the screen. Next, based on the mini-game name that was passed in, it instantiates the class for that game and calls `init()` on it.

Lastly, this function sets the current mode to indicate a mini-game is in progress, draws the console, hides the game selection screen, and shows the mini-game. Remember that the game loop is constantly firing at this point, so the very next iteration will result in the mini-game beginning.

The blit() Function—Putting Stuff on the Screen

The very last function in `main.js` is the ubiquitous `blit()` function:

```
function blit(inImage, inX, inY) {

  inImage.style.left = inX + "px";
  inImage.style.top = inY + "px";
  inImage.style.zIndex = frameZIndexCounter;
  inImage.style.display = "block";
  frameZIndexCounter++;

} // End blit().
```

`blit()` is used to place an image on the screen at some specified coordinates. The first argument to this function is a reference to the image object to place. Setting the `left` and `top` style properties of this element places the image where it needs to be. The `zIndex` property is set, and the `frameZIndexCounter` variable is reset at the start of every frame and incremented every time an image is placed. This has the effect that each subsequent `blit()` is on top of anything `blit()`'d before. This is generally how blit works. When the `display` property is set to `block`, the image is actually shown on the screen, and the image is now visible where it was specified to be.

■ **Note** The term *blit* is a common one in graphics and game programming. Without getting into too much detail, blit usually refers to drawing an image on the screen. In the case of a browser-based game, you won't be literally drawing an image; instead, you'll be placing one somewhere, as is the case here (the difference being that a blit in the classic sense draws the image pixel by pixel, whereas placing it in this case doesn't).

Writing consoleFuncs.js

`consoleFuncs.js` contains the code that deals with the game console, including the border with lights around a mini-game, the hands below it, and the status area. It contains a whopping three functions.

The drawConsole() Function

The first function in consoleFuncs.js is drawConsole():

```
function drawConsole() {

  // These are the parts of the game console that do not need to be redrawn
  // with each frame.
  blit(consoleImages["imgGameFrame"], 0, 0);
  blit(consoleImages["imgConsoleLeft"], 0, 240);
  blit(consoleImages["imgConsoleMiddle"], 108, 240);
  blit(consoleImages["imgConsoleRight"], 215, 240);

} // End drawConsole().
```

As you can see by the comments, these four images are the ones that never change, unlike the hands and lights, for instance, which do. Therefore, drawConsole() doesn't need to be called every frame, as the code for the mini-games does. If you are completely new to game design, talking about drawing something every frame may be a bit foreign to you, but please bear with me! When we look at main.js, I'll explain further. For now, we need to keep going through what you'll see are essentially support functions for the main processing code, which is what this file (and some other files) contains. So let's keep chugging along here.

The updateLights() Function

The next function we come to is updateLights(), which in a nutshell is responsible for making the lights flash around the game console frame. Here is its code:

```
function updateLights() {

  // Every half a second we are going to light some lights and restore others
  gameState.lightChangeCounter++;

  if (gameState.lightChangeCounter > 12) {

    gameState.lightChangeCounter = 0;

    // Hide the frame and lights so we start fresh.
    consoleImages["imgGameFrame"].style.display = "none";
    consoleImages["imgGameFrameLeftLight1"].style.display = "none";
    consoleImages["imgGameFrameLeftLight2"].style.display = "none";
    consoleImages["imgGameFrameLeftLight3"].style.display = "none";
    consoleImages["imgGameFrameLeftLight4"].style.display = "none";
    consoleImages["imgGameFrameLeftLight5"].style.display = "none";
    consoleImages["imgGameFrameRightLight1"].style.display = "none";
    consoleImages["imgGameFrameRightLight2"].style.display = "none";
    consoleImages["imgGameFrameRightLight3"].style.display = "none";
    consoleImages["imgGameFrameRightLight4"].style.display = "none";
    consoleImages["imgGameFrameRightLight5"].style.display = "none";
```

```
    // Draw mini-game area frame
    blit(consoleImages["imgGameFrame"], 0, 0);

    // Turn each light on or off randomly.
    if (jscript.math.genRandomNumber(0, 1) == 1) {
      blit(consoleImages["imgGameFrameLeftLight1"], 0, 22);
    }
    if (jscript.math.genRandomNumber(0, 1) == 1) {
      blit(consoleImages["imgGameFrameLeftLight2"], 0, 64);
    }
    if (jscript.math.genRandomNumber(0, 1) == 1) {
      blit(consoleImages["imgGameFrameLeftLight3"], 0, 107);
    }
    if (jscript.math.genRandomNumber(0, 1) == 1) {
      blit(consoleImages["imgGameFrameLeftLight4"], 0, 150);
    }
    if (jscript.math.genRandomNumber(0, 1) == 1) {
      blit(consoleImages["imgGameFrameLeftLight5"], 0, 193);
    }
    if (jscript.math.genRandomNumber(0, 1) == 1) {
      blit(consoleImages["imgGameFrameRightLight1"], 220, 20);
    }
    if (jscript.math.genRandomNumber(0, 1) == 1) {
      blit(consoleImages["imgGameFrameRightLight2"], 220, 62);
    }
    if (jscript.math.genRandomNumber(0, 1) == 1) {
      blit(consoleImages["imgGameFrameRightLight3"], 220, 107);
    }
    if (jscript.math.genRandomNumber(0, 1) == 1) {
      blit(consoleImages["imgGameFrameRightLight4"], 220, 150);
    }
    if (jscript.math.genRandomNumber(0, 1) == 1) {
      blit(consoleImages["imgGameFrameRightLight5"], 220, 193);
    }
  }

} // End updateLights().
```

First of all, lest we cause seizures in small children,[3] we don't want the lights to flash too wildly; every half-second seems reasonable. Since we know we get 24 fps, we want to update the lights only every twelfth frame, hence `gameState.lightChangeCounter`.

3. In December 1997, there were a few hundred incidents of Japanese children being thrown into seizures while watching the popular cartoon Pokemon. Further details can be found at `http://www.cnn.com/WORLD/9712/17/video.seizures.update`.

Once we determine it's safe (!) to change the lights, we begin by hiding all of them. Usually, when you write a game, you begin each frame by clearing the screen. Since we aren't dealing with a big grid of pixels, we can't really clear anything. But the equivalent operation is to hide the images. Again, for optimization reasons, it isn't really necessary to hide images that either don't change or can never have other images placed over them. However, in the case of the lights, they need to be cleared, as does the frame; otherwise, you would see the lights turn on but never turn off.

Once everything is cleared, we decide whether each of the ten lights is lit. To do this, we use the `jscript.math.genRandomNumber()` function we developed in Chapter 3. Then, for whichever lights are on, we blit them, and we're finished.

The updateHands() Function

Only one function remains now, and that's `updateHands()`, shown here:

```
function updateHands() {

  // Clear all images to prepare for proper display.
  consoleImages["imgLeftHandUp"].style.display = "none";
  consoleImages["imgLeftHandDown"].style.display = "none";
  consoleImages["imgLeftHandLeft"].style.display = "none";
  consoleImages["imgLeftHandRight"].style.display = "none";
  consoleImages["imgLeftHandUL"].style.display = "none";
  consoleImages["imgLeftHandUR"].style.display = "none";
  consoleImages["imgLeftHandDL"].style.display = "none";
  consoleImages["imgLeftHandDR"].style.display = "none";
  consoleImages["imgRightHandDown"].style.display = "none";

  // Display appropriate left-hand image.
  if (gameState.playerDirectionUp && !gameState.playerDirectionDown &&
    !gameState.playerDirectionLeft && !gameState.playerDirectionRight) {
    blit(consoleImages["imgLeftHandUp"], 29, 240);
  } else if (!gameState.playerDirectionUp && gameState.playerDirectionDown &&
    !gameState.playerDirectionLeft && !gameState.playerDirectionRight) {
    blit(consoleImages["imgLeftHandDown"], 29, 240);
  } else if (!gameState.playerDirectionUp && !gameState.playerDirectionDown &&
    gameState.playerDirectionLeft && !gameState.playerDirectionRight) {
    blit(consoleImages["imgLeftHandLeft"], 29, 240);
  } else if (!gameState.playerDirectionUp && !gameState.playerDirectionDown &&
    !gameState.playerDirectionLeft && gameState.playerDirectionRight) {
    blit(consoleImages["imgLeftHandRight"], 29, 240);
  } else if (gameState.playerDirectionUp && !gameState.playerDirectionDown &&
    gameState.playerDirectionLeft && !gameState.playerDirectionRight) {
    blit(consoleImages["imgLeftHandUL"], 29, 240);
```

```
  } else if (gameState.playerDirectionUp && !gameState.playerDirectionDown &&
    !gameState.playerDirectionLeft && gameState.playerDirectionRight) {
    blit(consoleImages["imgLeftHandUR"], 29, 240);
  } else if (!gameState.playerDirectionUp && gameState.playerDirectionDown &&
    gameState.playerDirectionLeft && !gameState.playerDirectionRight) {
    blit(consoleImages["imgLeftHandDL"], 29, 240);
  } else if (!gameState.playerDirectionUp && gameState.playerDirectionDown &&
    !gameState.playerDirectionLeft && gameState.playerDirectionRight) {
    blit(consoleImages["imgLeftHandDR"], 29, 240);
  } else {
    blit(consoleImages["imgLeftHandNormal"], 29, 240);
  }

  // Display appropriate left-hand image.
  if (gameState.playerAction) {
    blit(consoleImages["imgRightHandDown"], 145, 240);
  } else {
    blit(consoleImages["imgRightHandUp"], 145, 240);
  }

} // End updateHands().
```

Just as in the `updateLights()` function, we begin by hiding all the images for both hands. Then we enter a giant `if . . . else` block to determine which left hand image should be shown. Four variables help us make this determination, and those are the player direction fields in `GameState`: `playerDirectionUp`, `playerDirectionDown`, `playerDirectionLeft`, and `playerDirectionRight`. You'll note that we need to cover eight cases: four cardinal directions plus the four combinatorial directions (up/left, up/right, down/left, and down/right). The `else` block handles when the player isn't currently moving.

The same decision is made about the right hand image, but since there are only two states there—either the button is pressed or it isn't—the situation, and the code, is much more compact.

This is all it takes to make the hands on the bottom work. Well, this and the setting of the four fields in the key handlers, as discussed next.

Writing keyHandlers.js

You've already met the two functions contained in the `keyHandlers.js` file in a sense, because they are the functions that are called whenever a key is pressed or released: `keyDownHandler()` and `keyUpHandler()`. Listing 11-4 shows the `keyHandlers.js` file.

Listing 11-4. *The keyHandlers.js File*

```
/**
 * ========================================================================
 * Return the keycode of the key firing an event.
 * ========================================================================
 */
function getKeyCode(e) {

  var ev = (e) ? e : (window.event) ? window.event : null;
  if (ev) {
    return (ev.charCode) ? ev.charCode:
      ((ev.keyCode) ? ev.keyCode : ((ev.which) ? ev.which : null));
  }

} // End getKeyCode().


/**
 * ========================================================================
 * Handle key down events.
 * ========================================================================
 */
function keyDownHandler(e) {

  var keyCode = getKeyCode(e);

  if (!gameState.currentGame.fullKeyControl) {
    switch (keyCode) {
      case KEY_SPACE:
        gameState.playerAction = true;
      break;
      case KEY_UP:
        gameState.playerDirectionUp = true;
      break;
      case KEY_DOWN:
        gameState.playerDirectionDown = true;
      break;
```

```
      case KEY_LEFT:
        gameState.playerDirectionLeft = true;
      break;
      case KEY_RIGHT:
        gameState.playerDirectionRight = true;
      break;
    }
  }

  gameState.currentGame.keyDownHandler(keyCode);

} // End keyDownHandler().


/**
 * ========================================================================
 * Handle key up events.
 * ========================================================================
 */
function keyUpHandler(e) {

  var keyCode = getKeyCode(e);

  // Always handle exiting a mini-game, even if the mini-game has full control
  // over key events.
  if (keyCode == 13) {
    if (gameState.currentMode == "miniGame") {
      gameState.currentGame.destroy();
      gameState.currentGame = null;
      document.getElementById("divMiniGame").style.display = "none";
      gameState.currentGame = new GameSelection();
      gameState.currentGame.init();
    }
  }

  if (!gameState.currentGame.fullKeyControl) {
    switch (keyCode) {
      case KEY_SPACE:
        gameState.playerAction = false;
      break;
      case KEY_UP:
        gameState.playerDirectionUp = false;
      break;
      case KEY_DOWN:
        gameState.playerDirectionDown = false;
      break;
```

```
      case KEY_LEFT:
        gameState.playerDirectionLeft = false;
      break;
      case KEY_RIGHT:
        gameState.playerDirectionRight = false;
      break;
    }
  }

  gameState.currentGame.keyUpHandler(keyCode);

} // End keyUpHandler().
```

Whenever a key is pressed, `keyDownHandler()` is called. This, in turn, calls they `getKeyCode()` function. The reason for this is that the way you get the code for the key that was pressed is different in IE than it is in other browsers, because their event model is fundamentally different. IE works by having a page-scoped `event` object generated for the event, while Firefox and other browsers pass that object directly to the handler function. So, in order to abstract away these differences, `getKeyCode()` deals with it, while the two handler functions do not. Essentially, all this function does is get the key code that was pressed. The first line contains some logic, the end result of which is that the variable `ev` contains the relevant `event` object, regardless of in which browser the application is running.

The line inside the `if` block looks a bit complex (and I usually frown on trinary logic statements like this, especially strung together as this is, but it was actually cleaner to write it this way than as a series of nested `if` statements), but it boils down to the fact that the browser in use determines which property of the `event` object you need to go after to get the key code. In some, it is `charCode`; in others, is it `keyCode`; and in still others, it is `which`. In any case, the relevant key code is returned and the event handler itself continues.

Once the key code is determined, it's a simple matter of a `switch` block to determine which key was pressed, and then the appropriate flag is set in `gameState`. However, this `switch` block will be hit only if the mini-game in play doesn't have full control over key events. Some mini-games will need this, as is the case with Deathtrap.

Lastly, the `keyDownHandler()` of the current mini-game is called so that it can do whatever work needs to be done specific to that game (which may be none, as is the case with Cosmic Squirrel).

The `onKeyUp()` handler is only slightly more complex. There, we first check if Enter is pressed. If it is, we need to exit the current mini-game, which means calling `destroy()` on the current mini-game class, hiding the `divMiniGame` `<div>`, and setting the game selection screen as the current screen. Beyond that, it's essentially the same as `onKeyDown()`, except that the player direction flags get unset (set to `false`, in other words).

Writing gameFuncs.js

The gameFuncs.js file contains a couple of functions that are essentially "helper" functions for mini-games. The first one we encounter is loadGameImage():

```
function loadGameImage(inName) {

  // Create an img object and set the relevant properties on it.
  var img = document.createElement("img");
  img.src = gameState.currentGame.gameName + "/img/" + inName + ".gif";
  img.style.position = "absolute";
  img.style.display = "none";

  // Add it to the array of images for the current game to avoid DOM access
  // later, and append it to the game area.
  gameState.currentGame.gameImages[inName] = img;
  document.getElementById("divGameArea").appendChild(img);

} // End loadGameImage().
```

Recall that each mini-game class, by virtue of extending the MiniGame base class (which we'll look at next) has a gameImages array that stores references to the images the mini-game uses. This array gets populated by calls to the loadGameImage() function. It creates a new <img> element, sets its src attribute to the specified image (which loads it into memory), and sets it up to be positional (position:absolute). It then appends it to the DOM as a child of the divGameArea element, which again is the viewport inside the game console frame where the mini-games take place. This function also adds the reference to the gameImages array of the current mini-game class.

As a corollary to the loadGameImage() function, there is the destroyGameImage() function:

```
function destroyGameImage(inName) {

  // Remove it from the DOM.
  var gameArea = document.getElementById("divGameArea");
  gameArea.removeChild(gameState.currentGame.gameImages[inName]);

  // Set element in array in null to complete the destruction.
  gameState.currentGame.gameImages[inName] = null;

} // End destroyGameImage().
```

When a mini-game's destroy() function is called, it is expected to use the destroyGameImage() function to destroy any images it loaded in init(). Not doing so would cause a memory leak, since every time the game was started, the images would be created as new, but the previous copies would still remain, unused. To destroy an image, we need to first remove it from the DOM, and then set the reference in the array to null. The JavaScript engine's garbage collector will take care of the rest.

The next function in the gameFuncs.js file is detectCollision():

```
function detectCollision(inObj1, inObj2) {

  var left1 = inObj1.x;
  var left2 = inObj2.x;
  var right1 = left1 + inObj1.width;
  var right2 = left2 + inObj2.width;
  var top1 = inObj1.y;
  var top2 = inObj2.y;
  var bottom1 = top1 + inObj1.height;
  var bottom2 = top2 + inObj2.height;

  if (bottom1 < top2) {
    return false;
  }
  if (top1 > bottom2) {
    return false;
  }
  if (right1 < left2) {
    return false;
  }
  if (left1 > right2) {
    return false;
  }

  return true;

} // End detectCollision().
```

Most video games, such as Cosmic Squirrel, require the ability to detect when two images—two objects in the game (usually termed *sprites*)—collide. For instance, we need to know when our player's squirrel is squished by an asteroid. There are numerous collision-detection algorithms, but many of them are not available to us in a browser setting. For instance, checking each pixel of one image against each pixel of another, while giving 100% accurate detection, isn't possible in a browser. The method used here is called *bounding boxes*. It is a very simple method that basically just checks the four corners of the objects. If the corner of one object is within the bounds of the other, a collision has occurred.

As illustrated in the example in Figure 11-9, each sprite has a square (or rectangular) area around it, called its bounding box, which defines the boundaries of the area the sprite occupies. Note in the diagram how the upper-left corner of object 1's bounding box is within the bounding box of object 2. This represents a collision. We can detect a collision by running through a series of tests comparing the bounds of each object. If any of the conditions are untrue, then a collision cannot possibly have occurred. For instance, if the bottom of object 1 is above the top of object 2, there's no way a collision could have occurred. In fact, since we're dealing with a square or rectangular object, we have only four conditions to check, any one of which being false precludes the possibility of a collision.

This algorithm does not yield perfect results. For example, in Cosmic Squirrel, you will sometimes see the squirrel hitting an object when they clearly did not touch. This is because the *bounding boxes* can collide *without the object itself* actually colliding. This could only be fixed with pixel-level detection, which again, is not available to us. But the bounding boxes approach gives an approximation that yields "good enough" results, so all is right with the world.

The last two functions, `addToScore()` and `subtractFromScore()` are pretty self-explanatory. Note that `subtractFromScore()` must do a check to be sure the score doesn't go below zero. Some games will actually go into negative scores, but I saw no need to inflict more psychological damage on the player! Zero is embarrassing enough, I figure. Both of them simply update the `score` field in `gameState`, and update the status area as well.

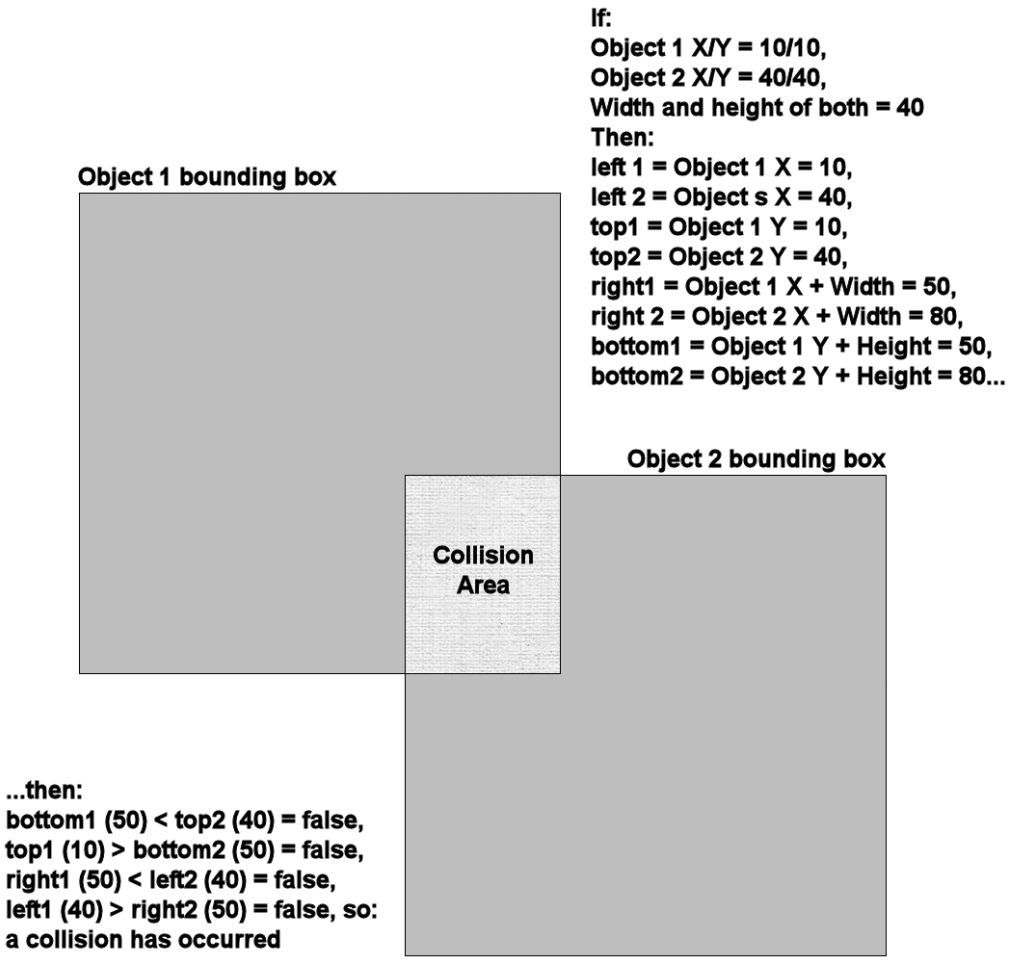


Figure 11-9. *An example of a basic bounding box collision detection algorithm*

Writing MiniGame.js

The `MiniGame` class is the base class for all mini-games, as well as the title screen and the game selection screen. Figure 11-10 shows the UML diagram for this class. It contains no real executable code, but it defines, in a sense, the interface that all mini-games must implement. The mini-games, title screen, and game selection screen essentially extend this class, overriding the default implementations of the methods it provides. Any that aren't needed don't have to be overridden; the default do-nothing version of the method will be used.

Figure 11-10. *UML diagram of the MiniGame class*

In addition to the methods that are members of the `MiniGame` class, three fields are present:

- `gameName`: This field *must* be set by the subclass during initialization, and it must match the directory in which the mini-game exists. This value is used to construct URLs for images that a mini-game may load. The default value in the `MiniGame` class is null.
- `gameImages`: This array is initialized to an empty array in `MiniGame`, so no errors will occur later if the mini-game doesn't load any images. A mini-game without images would be pretty pointless, however, so this array won't stay empty for long.
- `fullKeyControl`: This field is essentially a flag that determines whether the mini-game is in complete control of keyboard events, and none of the default behavior for these events occurs (as seen in `keyHandlers.js` previously). The default value here is `false`, so the default events will occur if the mini-game class does not override this value.

Writing Title.js

I did not put a UML diagram here for `Title.js` because it would look exactly like that of the `MiniGame` class, since the `Title` class extends the `MiniGame` class.

How does a class extend another in JavaScript exactly? By using the prototype, of course! As you saw in Chapter 1, every object in JavaScript has a prototype associated with it. This is essentially a prototype for the structure of the class. When you set the prototype of class B to class A, for instance, it means that class B will look like class A, plus whatever additional things class B defines.

In the case of the Title class, we find this line of code at the end of Title.js:

```
Title.prototype = new MiniGame;
```

That is, conceptually, the same thing as saying the Title class extends the MiniGame class. Let's say the definition of the Title class was nothing but this:

```
function Title() { }
```

If you were to do this:

```
var t = new Title();
```

you would find that the object referenced by the variable t had five methods: init(), destroy(), processFrame(), keyUpHandler(), and keyDownHandler(). You would also find that it had three properties: gameName, gameImages, and fullKeyControl. This is by virtue of it extending the MiniGame class, where those members are defined. Now, if the Title class contains an init() method itself (which it does, in this case), then the object pointed to by the variable t would have an init() method as defined in the Title class, *not* the empty version of that function found in the MiniGame class. And if the Title class defines a function named doSomething(), then the object pointed to by the variable t would contain a function doSomething(), even though the MiniGame class does not. None of this is unusual in terms of class inheritance and is what we would expect to be the case.

The Title class, shown in Listing 11-5, overrides three of the MiniGame class methods: init(), destroy(), and keyUpHandler().

Listing 11-5. *The Title Class*

```
function Title() {


  /**
   * ========================================================================
   * Game initialization.
   * ========================================================================
   */
  this.init = function() {

    document.getElementById("divTitle").style.display = "block";

  } // End init().


  /**
   * ========================================================================
   * Handle key up events.
   * ========================================================================
   */
  this.keyUpHandler = function(e) {
```

```
    gameState.currentGame.destroy();
    gameState.currentGame = null;
    gameState.currentGame = new GameSelection();
    gameState.currentGame.init();

  } // End keyUpHandler().


  /**
   * ======================================================================
   * Destroy resources.
   * ======================================================================
   */
  this.destroy = function() {

    document.getElementById("divTitle").style.display = "none";

  } // End destroy().


} // End Title class.


// Title class "inherits" from MiniGame class (even though, strictly speaking,
// it isn't a mini-game).
Title.prototype = new MiniGame;
```

Recall that I said that the `Title` class, while obviously not actually a mini-game, is treated like one just the same. As such, when the application starts, it begins by instantiating a `Title` class, and then calling `init()` on it. Here, the only job `init()` has is to make visible the `<div>` containing the markup for the title screen. Then, when a key is pressed and released, `keyUpHandler()` is called. It calls the `destroy()` method of the object pointed to by the `gameState.currentGame` field, which is itself! `destroy()` simply hides the `<div>` for the title screen again. Control then returns to `keyUpHandler()`, which instantiates a new instance of the `GameSelection` class, sets `gameState.currentGame` to point to it, and calls `init()` on it. Remember that all this time, the main game loop is firing via timeout. So, when the next iteration occurs, it will be calling `processFrame()` on the `GameSelection` instance, hence essentially switching to that screen.

Note that the `Title` class does not override the `processFrame()` function. It has no work to do there, so there's no need to include that function. The default do-nothing implementation in the `MiniGame` class will be fired once per frame, so no problem there.

Writing GameSelection.js

The `GameSelection` class is the class that deals with the game selection screen, not surprisingly! It is again, like the `Title` screen class, treated just like a mini-game. Figure 11-11 shows its UML diagram.

Figure 11-11. *UML diagram of the GameSelection class*

As in the `Title` class, when the user presses and releases a key from the game selection screen, the `GameSelection` class is instantiated, and `init()` is called. There isn't much to do there, but let's see it anyway:

```
this.init = function() {

  gameState.currentMode = null;
  document.getElementById("divGameSelection").style.display = "block";

} // End init().
```

First, we set `gameState.currentMode` to null, indicating a mini-game is not currently in progress. Remember that when the user exits a mini-game, a new `GameSelection` instance will be created, and `init()` will be called on it, hence it is a good place to set that value. After that, it's just a simple matter of showing the game selection `<div>`, and we're all set.

For each frame, `GameSelection` has some work to do:

```
this.processFrame = function() {

  document.getElementById("ssCosmicSquirrel").style.display = "none";
  document.getElementById("ssDeathtrap").style.display = "none";
  switch (this.showingGame) {
    case 1:
      document.getElementById("ssCosmicSquirrel").style.display = "block";
      document.getElementById("mgsDesc").innerHTML =
        "In space, no one can hear a giant space squirrel buy it";
    break;
    case 2:
      document.getElementById("ssDeathtrap").style.display = "block";
      document.getElementById("mgsDesc").innerHTML =
        "Hop on the tiles to escape the chasm without getting cooked";
    break;
  }

} // End processFrame().
```

It begins with hiding the screenshot preview images for our mini-games. Since there aren't too many here, and speed isn't really of the essence as it is in a mini-game, we don't load references to these images into an array to avoid the DOM lookups as previously discussed. Instead, we get a reference to the images each time through. After they are hidden, we determine which mini-game preview is currently showing, and we show that appropriate image object, and also display the appropriate description. That's all there is to it.

The `keyUpHandler()` is where the majority of the work for the `GameSelection` class actually is, as you can see for yourself:

```
this.keyUpHandler = function(e) {

  switch (e) {
    case KEY_LEFT:
      this.showingGame--;
      if (this.showingGame < 1) {
        this.showingGame = this.numGames;
      }
    break;
    case KEY_RIGHT:
      this.showingGame++;
      if (this.showingGame > this.numGames) {
        this.showingGame = 1;
      }
    break;
    case KEY_SPACE:
      gameState.currentGame.destroy();
      gameState.currentGame = null;
      switch (this.showingGame) {
        case 1:
          startMiniGame("cosmicSquirrel");
        break;
        case 2:
          startMiniGame("deathtrap");
        break;
        case 3:
          startMiniGame("refluxive");
        break;
      }
    break;
  }

} // End keyUpHandler().
```

When the user presses the left or right arrow key, we decrement or increment the value of the `showingGame` field correspondingly. When decrementing, when we get below 1 (which represents the first mini-game), we jump to the number of mini-games as defined by the `numGames` field. Likewise, when we get above the number of mini-games, we jump back to 1. This makes it so that no matter how many mini-games we have, the list will cycle back to the

other end when the bounds are reached on either end. When the user presses the spacebar, it's time to begin the currently selected mini-game. To do so, we call the `startMiniGame()` function, passing the name of the game to start. That function takes care of all the details of starting a game, as you saw earlier.

Finally, the `destroy()` function is very simple:

```
this.destroy = function() {

  document.getElementById("divGameSelection").style.display = "none";

} // End destroy().
```

We just hide the `<div>` containing the game selection screen, and our work here is done!

Writing CosmicSquirrel.js

OK, now we get to the really good stuff! We've gone through all the code that, in effect, makes up the plumbing of K&G Arcade. At the end of the day though, it's all about the mini-games! This is where the bulk of the action is, and also where the nongeneric code is found. Before we check out the code though, let's get the lay of the land, so to speak.

In Figure 11-12, you see the `CosmicSquirrel` class itself. You will by now notice that it extends the `MiniGame` class, and because of that, it exposes a known interface with our five by now well-known methods, as well as our three common attributes. As you can also see, it contains several unique fields: `ObstacleDesc`, `PlayerDesc`, `AcornDesc`, `player`, `acorn`, and `obstacles`. The first three of these are themselves classes that are defined inside the `CosmicSquirrel` class. This is akin to an inner class in Java. They could just as easily have been defined outside the `CosmicSquirrel` class, but I felt that since they are used by only that class, it made sense to define them inside, to make the relationship somewhat more explicit. The other three fields are instances of those inner classes (well, `player` and `acorn` are; `obstacles` is actually an array of `ObstacleDesc` objects).

Figure 11-12. *UML diagram of the CosmicSquirrel class*

Setting Up the Obstacle, the Player, and the Acorn

Figure 11-13 shows the UML diagram of the `ObstacleDesc` class. This class, whose name is short for Obstacle Descriptor, is used to represent an obstacle on the screen, which could be an alien, spaceship, asteroid, or comet. Each of these objects defines the ID of the object, its X/Y location, the width and height of the image that represents it on the screen, the direction it is currently moving in, and its speed.

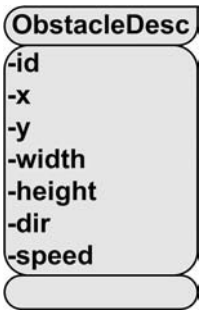


Figure 11-13. *UML diagram of the ObstacleDesc class*

The code for the `ObstacleDesc` class is as follows:

```
function ObstacleDesc(inID, inX, inY, inDir, inSpeed, inWidth, inHeight) {
  this.id = inID;
  this.x = inX;
  this.y = inY;
  this.width = inWidth;
  this.height = inHeight;
  this.dir = inDir;
  this.speed = inSpeed;
} // End ObstacleDesc class.
```

It is, in effect, just a simple data structure. Note that its argument list, which is its constructor (since that's in effect what any function defining a class is in JavaScript), allows you to set all the parameters for the obstacle when the instance is constructed. You will see this in action very shortly.

The `PlayerDesc` class, whose UML diagram can be seen in Figure 11-14, describes the player—that is, the cosmic squirrel. It contains the current X/Y location of the player, and the width and height of the image of the squirrel. It also defines a method that is called whenever the player dies or reaches the acorn, to reset it to its starting position.

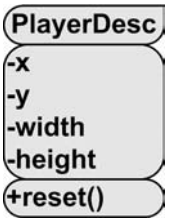


Figure 11-14. *UML diagram of the PlayerDesc class*

The code for the `PlayerDesc` class is just about as simple as the `ObstacleDesc` class, with the addition of the `reset()` method:

```
function PlayerDesc() {
  this.x = null;
  this.y = null;
  this.width = 18;
  this.height = 18;
  this.reset = function() {
    this.x = 91;
    this.y = 180;
  }
  this.reset();
} // End PlayerDesc class.
```

To reset the player is a simple matter of setting his initial starting location. Note the call to `reset()` at the very end. This is what will execute when the class is instantiated, setting it up initially.

Moving right along, take a look at the `AcornDesc` class, shown in Figure 11-15. This class describes the acorn, of course.

Figure 11-15. *UML diagram of the AcornDesc class*

Notice that `AcornDesc` has the same structure as the `PlayerDesc` class, and also defines that same `reset()` method, which randomly places the acorn on the screen when it is reached by the player using the `jscript.math.genRandomNumber()` function, as seen here:

```
function AcornDesc() {
  this.x = null;
  this.y = null;
  this.width = 18;
  this.height = 18;
  this.reset = function() {
    this.x = jscript.math.genRandomNumber(1, 180)
    this.y = 2;
  }
  this.reset();
} // End AcornDesc class.
```

Following these three classes in the `CosmicSquirrel` class are these three lines:

```
this.player = new PlayerDesc();
this.acorn = new AcornDesc();
this.obstacles = new Array();
```

Since there are only one player and one acorn, we have them as soon as the `CosmicSquirrel` class is instantiated. We also have an empty array of obstacles to populate, which is done when `init()` is called on the `CosmicSquirrel` instance.

Starting the Game

Let's see `init()` now:

```
this.init = function() {

  // Configure base game parameters.
  this.gameName = "cosmicSquirrel";

  // Load all the images required for this game.
  loadGameImage("background");
  loadGameImage("acorn");
  loadGameImage("squirrelUp");
  loadGameImage("squirrelDown");
  loadGameImage("squirrelLeft");
  loadGameImage("squirrelRight");
  loadGameImage("squirrelStill");
  loadGameImage("alien1");
  loadGameImage("alien2");
  loadGameImage("ship1");
  loadGameImage("ship2");
  loadGameImage("asteroid1");
  loadGameImage("asteroid2");
  loadGameImage("comet1");
  loadGameImage("comet2");
```

```
    // Create obstacle descriptors and add to array.
    this.obstacles.push(new ObstacleDesc("alien1", 170, 30, "R", 5, 24, 24));
    this.obstacles.push(new ObstacleDesc("alien2", 80, 30, "R", 5, 24, 24));
    this.obstacles.push(new ObstacleDesc("ship1", 110, 60, "L", 2, 32, 24));
    this.obstacles.push(new ObstacleDesc("ship2", 10, 60, "L", 2, 32, 24));
    this.obstacles.push(new ObstacleDesc("asteroid1", 80, 90, "R", 4, 32, 32));
    this.obstacles.push(new ObstacleDesc("asteroid2", 140, 90, "R", 4, 32, 32));
    this.obstacles.push(new ObstacleDesc("comet1", 240, 130, "L", 3, 64, 14));
    this.obstacles.push(new ObstacleDesc("comet2", 70, 130, "L", 3, 64, 14));

  } // End init().
```

First, the `gameName` field is set, which must be done in all mini-games. Notice that the value set there, `"cosmicSquirrel"`, matches the directory where this code is. This is no coincidence; when the `loadGameImage()` function is called, it will use that value to construct the URL to the image being loaded.

Speaking of `loadGameImage()`, next are a batch of those calls. Each one, as you saw previously, creates an `<img>` tag, loads it with the image named, and adds it to the `gameImages` array, which is found in the `MiniGame` base class.

Lastly, we have a series of eight lines of code that are responsible for creating the obstacles the player must avoid. Each one is an `ObstacleDesc` instance, and here you can see where the constructor parameters I mentioned earlier come into play. We simply instantiate an `ObstacleDesc` instance, passing into it the appropriate parameters, and push that object onto the `obstacles` array. Nothing more to it!

Processing a Single Frame of Action

Now we get into the `processFrame()` function, which I remind you will be called 24 times a second, once per frame. The first thing we see happening there is to hide all the images used in this mini-game:

```
    for (img in this.gameImages) {
      this.gameImages[img].style.display = "none";
    }
```

You can see here the use of the `gameImages` array, rather than direct DOM access. This is good for speed!

Following that are two lines of code:

```
    // Blit background.
    blit(this.gameImages["background"], 0, 0);

    // Blit acorn.
    blit(this.gameImages["acorn"], this.acorn.x, this.acorn.y);
```

Recall that the `blit()` function serves to put an image on the screen at a specified location. Here, we're first placing the background image onto the game area, effectively filling the entire game area with the background. Next, we place the acorn at its current location, using the values stored in the `AcornDesc` instance referenced by the `acorn` variable.

After that, it's time to do the same with the obstacles. However, since the `ObstacleDesc` objects are in an array, we need to iterate over that array and `blit()` each, like so:

```
for (i = 0; i < this.obstacles.length; i++) {
  var obstacle = this.obstacles[i];
  blit(this.gameImages[obstacle.id], obstacle.x, obstacle.y);
}
```

So, at this point, the only thing left to do is show the squirrel; otherwise, the game would be pretty boring (I mean, watching the obstacles move around is kind of neat, I suppose, but not much of a game!). So, let's throw the squirrel on the screen:

```
if (gameState.playerDirectionUp) {
  blit(this.gameImages["squirrelUp"], this.player.x, this.player.y);
} else if (gameState.playerDirectionDown) {
  blit(this.gameImages["squirrelDown"], this.player.x, this.player.y);
} else if (gameState.playerDirectionLeft) {
  blit(this.gameImages["squirrelLeft"], this.player.x, this.player.y);
} else if (gameState.playerDirectionRight) {
  blit(this.gameImages["squirrelRight"], this.player.x, this.player.y);
} else {
  blit(this.gameImages["squirrelStill"], this.player.x, this.player.y);
}
```

There is just a little more work to do here because which way the player is moving determines which version of the squirrel we show. So, we have a series of `if` statements that interrogate the four flags in the `GameState` object that tell us which way the squirrel is moving, and we `blit()` the appropriate image. If the player isn't moving at all, we show the squirrel facing up as the default image.

Now that everything is actually drawn on the screen, we need to process the logic of the game for this frame. The first step is to move the obstacles. Since this movement continues unabated, regardless of what the player does, there are no conditions that need to be checked. We simply update their positions, and those changes will be reflected in the next frame drawing. Here is the code that does the movement:

```
for (i = 0; i < this.obstacles.length; i++) {
  var obstacle = this.obstacles[i];
  if (obstacle.dir == "L") {
    obstacle.x = obstacle.x - obstacle.speed;
  }
  if (obstacle.dir == "R") {
    obstacle.x = obstacle.x + obstacle.speed;
  }
  // Bounds checks (comets handled differently because of their size).
  if (obstacle.id.indexOf("comet") != -1) {
    if (obstacle.x < -40) {
      obstacle.x = 240;
    }
```

```
    } else {
      if (obstacle.x < -70) {
        obstacle.x = 240;
      }
    }
    if (obstacle.x > 240) {
      obstacle.x = -40;
    }
  }
```

For each object, we check the value of the `dir` attribute of the `ObstacleDesc` object associated with the obstacle to see which direction it is moving in, and we update its `x` location accordingly, using the `speed` attribute as the change value. Next, we do some checks so that when an obstacle moves completely off the screen in either direction, we reset its `x` location so it reappears on the other side of the screen. Note that because the comets are longer than the other obstacles, we need to check for different values than we do for all the other obstacles, hence the branching logic.

The next piece of business to attend to is moving the player.

```
if (gameState.playerDirectionUp) {
  this.player.y = this.player.y - 2;
  if (this.player.y < 2) {
    this.player.y = 2;
  }
}
if (gameState.playerDirectionDown) {
  this.player.y = this.player.y + 2;
  if (this.player.y > 180) {
    this.player.y = 180;
  }
}
if (gameState.playerDirectionRight) {
  this.player.x = this.player.x + 2;
  if (this.player.x > 180) {
    this.player.x = 180;
  }
}
if (gameState.playerDirectionLeft) {
  this.player.x = this.player.x - 2;
  if (this.player.x < 2) {
    this.player.x = 2;
  }
}
```

Depending on which way the player is currently moving, we increment or decrement either the `x` or `y` location by 2. We then apply some bounds checking to make sure the player can't move off the mini-game screen in any direction. Note that with this logic, the player can move in the four cardinal directions, as well as the four combinatorial directions. For example, if `gameState.playerDirectionUp` and `gameState.playerDirectionRight` were true, then two of

the four `if` statements here would execute, causing the squirrel to move diagonally. This is perfectly acceptable, and works to makes the game far less frustrating than it would be if the player could move in only the four cardinal directions.

We're almost finished with game play, believe it or not! Only two tasks remain in `processFrame()`. First, we need to determine if the player has gotten the acorn, and we do that with this code:

```
if (detectCollision(this.player, this.acorn)){
  this.player.reset();
  this.acorn.reset();
  addToScore(50);
}
```

We already looked at the `detectCollision()` function, so we don't need to go over that again. If that call returns `true`, then we call `reset()` on the `PlayerDesc` instance referenced by the `player` field, which returns the player to his starting position. We then do the same for the acorn, which randomly places it somewhere at the top of the screen. Finally, we add 50 points to the player's score.

In the same vein, we need to check for collisions with the eight obstacles, and the code is very nearly identical:

```
for (i = 0; i <  this.obstacles.length; i++) {
  if (detectCollision(this.player, this.obstacles[i])){
    this.player.reset();
    this.acorn.reset();
    subtractFromScore(25);
  }
}
```

We call `detectCollision()` for each of the eight obstacles. If a collision is detected, we do the same resets as with a collision with the acorn, but this time subtract 25 from the score. Subtracting less than the player earns for getting the acorn is just a nice thing to do (it would be a bit sadistic if it were reversed!). I would bet you've played games that score in a seemingly unfair way, and I know I've certainly been frustrated by that, so I wanted to be just a bit nicer in this game!

Cleaning Up

With `processFrame()` now out of the way, only one task remains to complete Cosmic Squirrel, and that is to clean up when the game ends. This is achieved by implementing the `destroy()` function, like so:

```
this.destroy = function() {

  destroyGameImage("background");
  destroyGameImage("acorn");
  destroyGameImage("squirrelUp");
  destroyGameImage("squirrelDown");
  destroyGameImage("squirrelLeft");
```

```
    destroyGameImage("squirrelRight");
    destroyGameImage("squirrelStill");
    destroyGameImage("alien1");
    destroyGameImage("alien2");
    destroyGameImage("ship1");
    destroyGameImage("ship2");
    destroyGameImage("asteroid1");
    destroyGameImage("asteroid2");
    destroyGameImage("comet1");
    destroyGameImage("comet2");

  } // End destroy().
```

Each call to `loadGameImage()` in the `init()` method is matched with a corresponding call to `destroyGameImage()` in the `destroy()` method. These calls remove the `<img>` element from the DOM and set the element in the array that holds a reference to that element to null, which effectively marks the image object for deletion by the garbage collector. Nothing else really needs to be done to clean up, so it's a short and sweet method.

Inheriting the Basics

At the very end of `CosmicSquirrel.js`, you see that one magical line that makes the inheritance work. This line of code allows the `CosmicSquirrel` class to have implementations of functions it doesn't explicitly need to alter, but which the rest of the game code expects to be implemented, such as `keyUpHandler()` and `keyDownHandler()`. That line of code is as follows:

```
CosmicSquirrel.prototype = new MiniGame;
```

With that single line of code, we ensure that the "plumbing" code that we've previously looked at—the code that calls the methods of the current mini-game class when the various life cycle events occur—will work, because it ensures that the `CosmicSquirrel` class (assuming no one has broken it, of course!) meets the interface requirements that plumbing code expects.

And, believe it or not, that is all the code behind this mini-game!

Writing Deathtrap.js

The Deathtrap game is only slightly more complicated than Cosmic Squirrel. It has about twice as much code, but a fair bit of it is very mundane, repetitive stuff. I won't be listing all that out here, but I'll certainly show you enough of it to get the feel for what's going on.

First, let's again look at the UML diagram of the `Deathtrap` class itself, as shown in Figure 11-16.

Figure 11-16. *UML diagram of the Deathtrap class*

Setting Up the Player

As in Cosmic Squirrel, we again find a `PlayerDesc` class, which is used to describe the characteristics of the player. As you can see in the UML diagram shown in Figure 11-17, the version here has a bit more to it. We still have the X/Y location of the player, but this time we also store the previous X/Y location, and you'll see why in a moment. We also have another X/Y location, defining which tile the player is on. This differs from the literal X/Y location on the screen, and both pieces of information are required to make the game work correctly. Lastly, we store the current state of the player: whether he is alive, dead, or has won the game.

Figure 11-17. *UML diagram of the PlayerDesc class*

We also again have a `reset()` method exposed by the `PlayerDesc` class, and this serves much the same purpose as it did in `CosmicSquirrel`. It will be called, as in that case, when the player dies or wins. Here is the code for the `PlayerDesc` class for this mini-game:

```
function PlayerDesc() {
  this.x = null;
  this.y = null;
  this.prevX = null;
  this.prevY = null;
  this.tileX = null;
  this.tileY = null;
  // State: A=Alive, D=Dead, W=Won
  this.state = null;
  this.reset = function() {
    this.x = 10;
    this.y = 152;
    this.prevX = 0;
    this.prevY = 0;
    this.tileX = 1;
    this.tileY = 8;
    this.state = "A";
  }
  this.reset();
} // End PlayerDesc class.
```

Following that inner class definition are four fields of the `Deathtrap` class:

```
this.player = new PlayerDesc();
this.deadCounter = null;
this.vertMoveCount = null;
this.correctPath = null;
this.regenPath = true;
```

These fields work as follows:

- `player`: The `PlayerDesc` instance describing the player.
- `deadCounter`: Used when the player dies to determine how long he should get zapped.
- `vertMoveCount`: Used when moving the player from tile to tile.
- `correctPath`: Defines which of the ten possible correct paths through the tiles is valid.
- `regenPath`: A flag that determines whether a new `correctPath` will be chosen when `reset()` is called.

Constructing the Death Matrix

The next piece of the code is the `deathMatrix`. The `deathMatrix` is a multidimensional array (10-by-2-by-2) that represents the grid of tiles the player must navigate. For each of the first dimensions, we have a 2-by-2 array, so ten arrays essentially. Each of those ten arrays defines

one possible correct path through the tiles. Each element in the 2-by-2 array is either a zero or one, one being a safe tile.

When the `reset()` method of the `Deathtrap` class is called, if the `regenPath` flag is set to `true`, the `jscript.math.genRandomNumber()` function is used to pick one of the ten possible paths. Therefore, every time the game starts, or when the player wins, a new `correctPath` will be chosen. When the player dies, this *will not* happen. This is again for fair game play. It would be really frustrating if the path were reset every time the player died, because he would not be able to work out the path through trial and error, so it would just be dumb luck each time, and that wouldn't be much fun!

Just for the sake of completeness, here is an example of how the `deathMatrix` is defined (these are the first two elements):

```
this.deathMatrix = new Array(10);
this.deathMatrix[0] = new Array(
  [ 1, 1, 1, 1, 1, 0, 0, 0, 0 ],
  [ 1, 0, 0, 0, 0, 0, 0, 0, 0 ],
  [ 1, 1, 1, 1, 1, 1, 0, 0, 0 ],
  [ 1, 1, 0, 0, 0, 1, 0, 0, 0 ],
  [ 1, 1, 0, 0, 0, 1, 0, 0, 0 ],
  [ 1, 1, 0, 1, 1, 1, 0, 0, 0 ],
  [ 0, 0, 0, 1, 0, 0, 0, 0, 0 ],
  [ 0, 0, 0, 1, 0, 0, 0, 0, 0 ],
  [ 0, 1, 1, 1, 0, 0, 0, 0, 0 ]
);
this.deathMatrix[1] = [
  [ 0, 0, 0, 0, 1, 0, 0, 0, 0 ],
  [ 0, 0, 0, 0, 1, 0, 0, 0, 0 ],
  [ 1, 1, 1, 1, 1, 1, 0, 0, 0 ],
  [ 1, 0, 0, 0, 1, 1, 0, 0, 0 ],
  [ 1, 0, 0, 0, 1, 0, 0, 0, 0 ],
  [ 1, 1, 1, 0, 1, 0, 0, 0, 0 ],
  [ 0, 0, 1, 0, 1, 1, 0, 0, 0 ],
  [ 0, 1, 1, 0, 0, 0, 0, 0, 0 ],
  [ 0, 1, 1, 0, 0, 0, 0, 0, 0 ]
];
```

Constructing the Move Matrix

One more member of the `Deathtrap` class that needs to be discussed is the `moveMatrix`. The `moveMatrix` is another multidimensional array, 10-by-10 in size, where each element represents a tile. The purpose of this array is to determine which directions the player can move from any given tile. Note that on the screen some of the tiles are not complete. This is necessary because of the angular drawing of the room. Those partial tiles should not be valid destinations for player movement. For each element in this array, a string value is present. The string contains one or more of `U`, `D`, `L`, and `R`. For instance, if a particular tile has a value of `"ULR"`, that means the player can move up, left, or right from that tile, but not down. A tile can also have a value of a string with none of those letters but just a space, which means it can never be reached and therefore no moves are valid from it.

Here is the code that defines the `moveMatrix`:

```
this.moveMatrix = new Array(10);
this.moveMatrix[0] = [ "RD", "RDL", "RDL", "RDL", "UDL",
  " ", " ", " ", " " ];
this.moveMatrix[1] = [ "URD", "URDL", "URDL", "URDL", "UDL",
  " ", " ", " ", " " ];
this.moveMatrix[2] = [ "URD", "URDL", "URDL", "URDL", "URDL",
  "DL", " ", " ", " " ];
this.moveMatrix[3] = [ "URD", "URDL", "URDL", "URDL", "URDL",
  "UDL", " ", " ", " " ];
this.moveMatrix[4] = [ "URD", "URDL", "URDL", "URDL", "URDL",
  "URDL", "DL", " ", " " ];
this.moveMatrix[5] = [ "URD", "URDL", "URDL", "URDL", "URDL",
  "URDL", "UDL", " ", " " ];
this.moveMatrix[6] = [ "UR", "URDL", "URDL", "URDL", "URDL",
  "URDL", "UDL", " ", " " ];
this.moveMatrix[7] = [ " ", "URD", "URDL", "URDL", "URDL",
  "URDL", "URDL", "DL", " " ];
this.moveMatrix[8] = [ " ", "UR", "URL", "URL", "URL",
  "URL", "URL", "UL", " " ];
```

Starting the Game

Next up is the `init()` method:

```
this.init = function() {

  // Configure base game parameters.
  this.gameName = "deathtrap";
  this.fullKeyControl = true;

  // Reset the game state.
  this.reset();

  // Load all the images required for this game.
  loadGameImage("background");
  loadGameImage("playerDieing");
  loadGameImage("playerJumping");
  loadGameImage("playerStanding");

} // End init().
```

After the setting of the mini-game name, note the setting of `fullKeyControl` to `true`. In the case of Deathtrap, we need to deal with keyboard events a little differently than we did with Cosmic Squirrel, and it turns out the default handling we saw previously in `keyHandlers.js` interferes with what has to happen here. Therefore, this mini-game needs to deal with keyboard events itself and not leave it to the default code.

After that is a call to `reset()`, followed by a series of `loadGameImage()` calls (notice how few graphics are actually needed for this game!). The `reset()` method doesn't really have a whole lot to do:

```
this.reset = function() {

  this.player.reset();
  this.deadCounter = 0;
  this.vertMoveCount = 0;
  if (this.regenPath) {
    this.correctPath = jscript.math.genRandomNumber(0, 9)
  }
  this.regenPath = false;
} // End reset().
```

The player is first reset to his starting position with a call to `player.reset()`. Next, the `deadCounter` and `vertMoveCount` fields are reset to zero. We then check to see if `regenPath` is `true`, and if so, we pick a new correct path through the tile field. Lastly, `regenPath` is then set to `false`, so that the next time `reset()` is called, unless it is a result of the player winning, we won't choose a new correct path through the tile field.

Handling the Player State: Winner, Dead, or Active

Moving on to the `processFrame()` method, we first encounter the same type of screen-clearing loop we saw in Cosmic Squirrel, which hides all the images in the `gameImages` array. After that is a `blit()` of the background.

Next is a largish `switch` block. This switch is on the state of the player. The first case is if the player has won:

```
  case "W":
    addToScore(1000);
    this.regenPath = true;
    this.reset();
  break;
```

It's a simple matter of adding 1000 to the score (I was in a generous mood!), setting the `regenPath` to `true` so that a new correct path through the tiles will be chosen, and calling `reset()` to get the player back to the starting position, and that's that.

The next case is if the player has died:

```
  case "D":
    blit(this.gameImages["playerDieing"], this.player.x, this.player.y);
    this.deadCounter++;
    if (this.deadCounter > 48) {
      this.reset();
    }
  break;
```

In this case, we `blit()` the player death graphic at the player's current location. That image is an animated GIF of the player being electrocuted. We want that to be shown in two seconds,

which is 48 frames (24 fps). That's where the `deadCounter` variable comes in. It's used to keep track of how many frames have elapsed. When we exceed 48, we just call `reset()`. Note that `regenPath` will be set to `false` at this point, so the same correct path is still in effect.

Now we come to the case where the player is alive. This is where the bulk of the work is done. First things first though—let's get the player on the screen!

```
if (gameState.playerDirectionUp || gameState.playerDirectionDown ||
  gameState.playerDirectionLeft || gameState.playerDirectionRight) {
  blit(this.gameImages["playerJumping"], this.player.x, this.player.y);
} else {
  blit(this.gameImages["playerStanding"], this.player.x, this.player.y);
}
```

A different image is needed when the player is jumping vs. when he is just standing still.

Next are four `if` blocks: one for each possible direction of movement. They are all pretty similar, so let's just review the first one, which is the case of the player moving up:

```
if (gameState.playerDirectionUp) {
  // If movement is done, finish up
  if (this.player.y <= (this.player.prevY - 16)) {
    this.vertMoveCount  = 0;
    this.player.x = this.player.prevX + 10;
    this.player.y = this.player.prevY - 16;
    gameState.playerDirectionUp = false;
    if (this.isDeathTile()) {
      this.player.state = "D";
    }
  } else { // Otherwise, move the player
    this.player.y = this.player.y - 3;
    this.vertMoveCount++;
    if (this.vertMoveCount > 1) {
      this.vertMoveCount = 0;
      this.player.x = this.player.x + 3;
    }
  }
}
```

This case (and the case of moving down) is actually a little more complicated than left and right, because both vertical and horizontal movement are involved. This is due to the fact that the tiles are organized diagonally from each other. So, first we determine if the player has already moved far enough from the previous position, which is where he was when he started the move. If not (the `else` clause), the player is moved three pixels horizontally and three pixels vertically for every six pixels horizontally (that is, the player moves every other frame vertically, while he moves horizontally every frame). When the player finally has moved far enough (the `if` clause), the current position is set to the previous position plus the proper amount horizontally and vertically. This is because the essentially diagonal movements would require fractional moves to be precise, and there is always a slight error when using integers. So to make the player wind up at the proper location in the end, the final values are based on the previous values.

Finally, a call to `isDeathTile()` is made. This function determines whether the tile the player is currently on is an electrified one. Here is the code for `isDeathTile()`:

```
this.isDeathTile = function() {

  if (
    this.deathMatrix[this.correctPath][this.player.tileY][this.player.tileX]
    == 0) {
    return true;
  } else {
    return false;
  }

} // End isDeathTile().
```

A lookup into the `deathMatrix` is done, using the `correctPath` value as the first dimension, and the player's X/Y location as the second and third dimensions. When the value is zero, it's an electrified tile; in which case, the player's `state` value is changed to `"D"` to signify he is dead.

Please do have a look at the other three cases for the other directions of movement. As I said, you'll find them all similar to the one for moving up, but it's certainly a good idea to check them out for yourself.

Handling Player Keyboard Events

Next up is the `keyDownHandler()` function:

```
this.keyDownHandler = function(inKeyCode) {

  // Although the right hand action button does nothing in this game,
  // it looks like things are broken if it doesn't press, so let's let
  // it be pressed, just to keep up appearances!
  if (inKeyCode == KEY_SPACE) {
    gameState.playerAction = true;
  }

} // End keyDownHandler().
```

Recall that this mini-game has full control over the key events, so nothing happens automatically. In this case, it means that although the action button, which is the right handle on the game console, does nothing in this game, it wouldn't even react when the user clicks space. This makes it look like something isn't working, since the button should probably be pressable, regardless of whether it serves a purpose. So, the `keyDownHandler()` function needs to deal with that. It's a simple matter of setting the `playerAction` flag to `true`.

The `keyUpHandler()` has a little more meat to it though. First, we check to be sure the player isn't currently moving and is alive:

```
  if (!gameState.playerDirectionUp && !gameState.playerDirectionDown &&
    !gameState.playerDirectionLeft && !gameState.playerDirectionRight &&
    this.player.state == "A") {
```

This is to avoid the situation where the player starts a move, then releases the key before the on-screen action has completed the jump to the new tile. If we were to allow this code to fire in that case, the player movement flag would be reset prematurely, causing the jump to terminate in the middle. That wouldn't be good! So, once we determine it's OK to proceed, we first check to see if it's the spacebar that is being released; in which case, it's just a matter of setting `playerAction` to `false`.

Once again, we encounter four cases corresponding to each of our cardinal directions. And also again, we'll just take a look at the first one here because the rest are substantially the same.

```
case KEY_UP:
  if (this.moveMatrix[tileY][tileX].indexOf("U") != -1) {
    if (tileY == 0 && tileX == 4) {
      this.player.state = "W";
    } else {
      this.player.tileY--;
      this.player.prevX = this.player.x;
      this.player.prevY = this.player.y;
      gameState.playerDirectionUp = true;
      gameState.playerDirectionRight  = false;
      gameState.playerDirectionDown = false;
      gameState.playerDirectionLeft  = false;
    }
  }
break;
```

First, we do a lookup into the `moveMatrix` for the tile the player is currently on. We see if the string value for that tile contains the letter `U`, indicating the player can move up from that tile. If he can, we need to see if the player is standing on the tile directly in front of the door. In that case, we change the player `state` to `"W"` to indicate a win. If the player hasn't won yet, then we decrement `tileY` to indicate the tile he will wind up on. Then we just set the movement flags accordingly, and we're finished.

The last piece of the puzzle is the `destroy()` method, which just cleans up the images created in `init()`. And, of course, there is also the `prototype` line, as in Cosmic Squirrel, to make sure the `Deathtrap` class extends the `MiniGame` class.

Writing Refluxive.js

Only one more game left to review now, and that's Refluxive. As before, let's begin by looking at the UML diagram of the `Refluxive` class itself, shown in Figure 11-18.

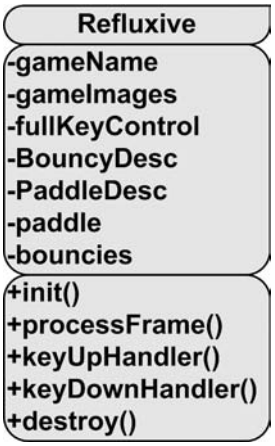


Figure 11-18. *UML diagram of the Refluxive class*

By now, this is all pretty much old hat for you. This game is again the typical `MiniGame`-derived class, with a few game-specific fields.

Setting Up the Bouncies and Paddle

First is another inner class, this time named `BouncyDesc`, as shown in Figure 11-19. The X/Y coordinates of a bouncy, which is the thing you need to keep bouncing in the air, is defined here, as is its width and height, both of which are needed for collision detection. We also define which direction the bouncy is moving. A flag field tells whether the bouncy is still on the screen. When all three bouncies have their `onScreen` fields set to `false`, the game is over. Unlike the other two games, Refluxive can actually end!

Figure 11-19. *UML diagram of the BouncyDesc class*

The next inner class is the `PaddleDesc` class, shown in Figure 11-20. The `PaddleDesc` class describes the paddle—in other words, the player. It's a simple matter of X/Y location plus width and height for collision detection.

Figure 11-20. *UML diagram of the PaddleDesc class*

After that are two fields: `paddle`, which is a pointer to an instance of `PaddleDesc`, and `bouncies`, which is an array of `BouncyDesc` objects.

Starting the Game

Next, we come to some code, starting with the `init()` method:

```
this.init = function() {

  // Configure base game parameters.
  this.gameName = "refluxive";

  // Load all the images required for this game.
  loadGameImage("background");
  loadGameImage("bouncy1");
  loadGameImage("bouncy2");
  loadGameImage("bouncy3");
  loadGameImage("paddle");
  loadGameImage("gameOver");

  // Initial bouncy positions.
  this.bouncies.push(
    new BouncyDesc(jscript.math.genRandomNumber(1, 180), 10, "SE"));
  this.bouncies.push(
    new BouncyDesc(jscript.math.genRandomNumber(1, 180), 70, "SW"));
  this.bouncies.push(
    new BouncyDesc(jscript.math.genRandomNumber(1, 180), 140, "NE"));

} // End init().
```

This is pretty much just like all the other `init()` methods you've seen thus far. We set the game name, load the images needed by the game, and in this case, construct three `BouncyDesc` objects and push each into the `bouncies` array. Their horizontal location is set randomly, and the vertical location is static. Their initial direction of movement is static as well. These last two items are static to help ensure they start out in reasonable positions, are moving in sufficiently different ways, and are separated enough to make the game challenging.

One interesting thing to note is that each of the bouncies is its own image, even though they are all the same underlying GIF. This is necessary because each needs to be individually addressable. This is a bit inefficient, because we are loading the same GIF into memory three

times. The browser and/or operating system might share it somehow, but we can't count on that. In such a limited game, this is hardly a major concern. For something more substantial, this is a shortcoming you would want to address somehow. One way to do this might be to modify the way the code works so that a game element is abstracted from its image or images. Maybe you have some sort of `ImageManager` class that contains all the images, and that deals with ensuring only a single instance of any given image exists, and the game elements would reference their images through that class.

Playing the Game

`processFrame()` is next, and it begins how the other two did: by clearing all the images off the screen. We then see a `blit()` of the background, just as in the other two games. After that comes a bit of code unique to this game:

```
if (!this.bouncies[0].onScreen && !this.bouncies[1].onScreen &&
  !this.bouncies[2].onScreen) {
  blit(this.gameImages["gameOver"], 10, 40);
  return;
}
```

As I mentioned earlier, Refluxive is the only one of the three games that can end before the user decides to quit. It ends when all three of the bouncies are off the screen. So, here we check for that condition, and if it is met, we display the "Game Over" message. Since the rest of `processFrame()` doesn't apply in this case, we simply `return`, and the frame is complete.

If the game is still going, however, we begin by `blit()`'ing the paddle and then the three bouncies:

```
if (this.bouncies[0].onScreen) {
  blit(this.gameImages["bouncy1"], this.bouncies[0].x, this.bouncies[0].y);
}
if (this.bouncies[1].onScreen) {
  blit(this.gameImages["bouncy2"], this.bouncies[1].x, this.bouncies[1].y);
}
if (this.bouncies[2].onScreen) {
  blit(this.gameImages["bouncy3"], this.bouncies[2].x, this.bouncies[2].y);
}
```

For each bouncy, we check if it's on the screen. There wouldn't be much sense in `blit()`'ing something the player can't see!

Next, we deal with player movements:

```
if (gameState.playerDirectionRight) {
  this.paddle.x = this.paddle.x + 4;
  // Stop at edge of screen
  if (this.paddle.x > 174) {
    this.paddle.x = 174;
  }
}
```

```
if (gameState.playerDirectionLeft) {
  this.paddle.x = this.paddle.x - 4;
  // Stop at edge of screen
  if (this.paddle.x < 1) {
    this.paddle.x = 1;
  }
}
```

Some simple bounds checking ensures that the player can't move the paddle off either edge of the screen. Since the player can move only left and right in this game, that's all there is to it! Note that this game doesn't need to implement `keyDownHandler()` or `keyUpHandler()`, as the default implementations do the job just fine, as was the case with Cosmic Squirrel. It's nice to keep the code size down this way!

Next up is bouncy movement. To do that, we loop through the `bouncies` array, and for each bouncy, we first check if it is on the screen. If not, we just continue the loop. If it is on the screen, we start by moving it based on its current `dir` value:

```
  if (this.bouncies[i].dir == "NE") {
    this.bouncies[i].x = this.bouncies[i].x + 3;
    this.bouncies[i].y = this.bouncies[i].y - 3;
  }
  if (this.bouncies[i].dir == "NW") {
    this.bouncies[i].x = this.bouncies[i].x - 3;
    this.bouncies[i].y = this.bouncies[i].y - 3;
  }
  if (this.bouncies[i].dir == "SE") {
    this.bouncies[i].x = this.bouncies[i].x + 3;
    this.bouncies[i].y = this.bouncies[i].y + 3;
  }
  if (this.bouncies[i].dir == "SW") {
    this.bouncies[i].x = this.bouncies[i].x - 3;
    this.bouncies[i].y = this.bouncies[i].y + 3;
  }
```

`"NE"`, as I'm sure you can guess, stands for northeast. Correspondingly, `"NW"` is northwest, `"SE"` is southeast, and `"SW"` is southwest. These are the four possible directions a bouncy can move. For each, we adjust the X and Y coordinates as appropriate.

After that's done, we need to deal with the situation where the bouncies bounce off the sides and top of the screen. To do that, we use this code:

```
  // Bounce off the frame edges (horizontal).
  if (this.bouncies[i].x < 1) {
    if (this.bouncies[i].dir == "NW") {
      this.bouncies[i].dir = "NE";
    } else if (this.bouncies[i].dir == "SW") {
      this.bouncies[i].dir = "SE";
    }
  }
  if (this.bouncies[i].x > 182) {
```

```
    if (this.bouncies[i].dir == "NE") {
      this.bouncies[i].dir = "NW";
    } else if (this.bouncies[i].dir == "SE") {
      this.bouncies[i].dir = "SW";
    }
  }
  // Bounce off the frame edges (vertical).
  if (this.bouncies[i].y < 1) {
    if (this.bouncies[i].dir == "NE") {
      this.bouncies[i].dir = "SE";
    } else if (this.bouncies[i].dir == "NW") {
      this.bouncies[i].dir = "SW";
    }
  }
```

When the X coordinate of the bouncy is less than one, it means it has collided with the left edge of the screen. In that case, we basically reverse the direction of travel. So, if it is moving northwest, we reverse it to northeast, and southwest becomes southeast. Likewise, for the right side of the screen (coordinate 182, because the bouncy is 18 pixels wide, so the right edge is at 182+18=200 at that point), we again reverse the directions. The same basic logic is applied for the top of the screen.

Note that there is no check for the bottom of the screen here, at least, not like these, because that's the one case where the bouncy doesn't bounce. It just exits the bottom of the screen if the player misses it. The last check needed here is that case precisely: when the bouncy leaves the screen:

```
  if (this.bouncies[i].y > 200) {
    this.bouncies[i].onScreen = false;
    subtractFromScore(50);
  }
```

When the player misses a bouncy, it becomes dead, so to speak, by setting its `onScreen` property to `false`. This also costs the player some points—50 in this case.

Only one thing remains to make this a complete game. Can you guess what that is?

We need to make it possible for the player to bounce the bouncies! The code to accomplish that is as follows:

```
  if (detectCollision(this.bouncies[i], this.paddle)) {
    // Reverse bouncy direction.
    if (this.bouncies[i].dir == "SE" &&
      this.bouncies[i].x + 9 < this.paddle.x + 12) {
      this.bouncies[i].dir = "NW";
      addToScore(10);
    }
    if (this.bouncies[i].dir == "SE" &&
      this.bouncies[i].x + 9 > this.paddle.x + 12) {
      this.bouncies[i].dir = "NE";
      addToScore(10);
    }
```

```
    if (this.bouncies[i].dir == "SW" &&
      this.bouncies[i].x + 9 < this.paddle.x + 12) {
      this.bouncies[i].dir = "NW";
      addToScore(10);
    }
    if (this.bouncies[i].dir == "SW" &&
      this.bouncies[i].x + 9 > this.paddle.x + 12) {
      this.bouncies[i].dir = "NE";
      addToScore(10);
    }
  } // End collision detected.
```

This is still inside the `for` loop, so we're working with one specific bouncy. We call the `detectCollision()` function, and if it returns `true`, we do the same sort of direction reversal as you saw earlier. One difference here is that the direction depends on which side of the paddle hit the bouncy. If the bouncy was moving southeast, and the player hits it with the left side of the paddle, the direction changes to northwest. If it hits the right side of the paddle, it changes to northeast. For southwest on the left side, it switches to northwest, and for southwest on the right side, it becomes northeast.

And that, dear friends, concludes the actual game logic behind Refluxive! The only code left is the usual `destroy()` method, which calls `destroyGameImage()` for each image loaded in `init()`, and, again, the `prototype` specification stating the `Refluxive` class extends `MiniGame`.

And that's a wrap folks! I hope you'll agree that this application had a lot to offer in terms of how to do object-oriented programming with JavaScript. More important though, I hope you had as much fun checking this game out as I had writing it! Try not to waste too much time at the office playing it!

Suggested Exercises

I'm sure you don't need me to tell you, but my main suggestion is to write some more mini-games! It is my hope that, as time allows, I will port some of the other mini-games from the full-blown K&G Arcade to JavaScript. I will post them to the Apress download site along with the other code for this book. Adding them should be a simple matter of dropping the appropriate directory in with the resources for the games and updating the `GameSelection` class. And you can do the same!

Come up with one or two simple game ideas, and try to implement them. If you've never written a game before, I suspect you'll get a great deal of pleasure out of it and will learn a lot along the way. You probably won't be able to pull off Final Fantasy, Halo, or anything like those games, so don't think too big. Just aim big enough that it's challenging and yet still fun at the same time. After all, that's what video games are all about . . . or at least should be!

One other suggestion is to save high scores for each mini-game in cookies, and create a Hall of Fame screen to display them. This should give you a good feel for how a screen in the game is developed, and also some practice with cookies. I would suggest this exercise as a first task, just to get your feet wet.

Summary

You might not think it at first, but programming a game is one of the best exercises in any language on any platform to exercise your skills and learn. Games touch a variety of areas of expertise and often require you to stretch your abilities a good bit, and I believe this chapter has shown that.

In this chapter, you saw an object-oriented approach to JavaScript that leads to clean, flexible code. The project demonstrated how inheritance can be achieved in JavaScript. You saw some tricks for maximizing performance, including avoiding superfluous DOM element accesses and speeding up the overall application. You even picked up a tidbit or two on basic game theory! And I believe that we built a game that is actually fun to play!

CHAPTER 12

Ajax: Where the Client and Server Collide

Ajax is all the rage today. It seems that you can't even be a web developer these days without knowing at least something about Ajax!

In this chapter, you will learn a bit about Ajax and put it to use building a one-on-one support chat application, as you can see at the sites of many companies that provide live chat support for their products and services. This will be the one project in this book that uses a server component as well, so you'll see how that all fits together. Additionally, you'll be introduced to a new library, Mootools, and see what it has to offer.

Chat System Requirements and Goals

Let's make believe for a bit. Say we're a new company on the block. We'll call ourselves Metacusoft Systems, for no other reason than the domain name is available and Googling for it returns no results, so I can be reasonably sure I'm not infringing on anyone. The first reader to register the domain name and start a company gets a prize! (No, you really don't, but feel free to take the name.)

Anyway, we sell widgets. Yes, the typical, mundane widget product. It's wonderful, it's amazing, and no one can live without it, and yet we can't come up with a more descriptive name. And, as great a product as it obviously is, it's not always a completely smooth ride for our customers. Sometimes, the widget doesn't, um, widgetate, as it should. Sometimes, although we would never admit this in a financial filing, customers need some assistance to effectively change their lives for the better with our amazing widgets. So, we need to offer some support for them.

We'll have a couple of people available via phone (over in India, of course, where there's apparently no shortage of folks named Bill, Mike, Tom, Sam, and Dave). We'll also offer something for the online crowd, those people who, like the Morlocks,[1] shun human contact. We'll offer a live one-on-one chat system where customers can communicate with human beings in real time, without needing to really talk to them and risk an actual conversation.

1. Morlock is the name of an invented species, offshoots of the human race, created by the famous author H.G. Wells in the novel *The Time Machine*, who exist many, many, *many* years in the future (the Morlocks, not H.G. Wells). The Morlocks live underground and are at that point almost not even identifiable as humans (formerly so anyway). Oh yeah, the Morlocks eat a species known as the Eloi, who are also descendants of the human race. Nice, huh?

It's a relatively simple application—one person types, the other sees it, and that's about the extent of it. Still, we'll try to jazz it up just a bit.

- The application should look halfway decent. After all, this is the public face of our company for those having difficulties with our product. They're probably already a little ticked at us, and we don't want to annoy them further with an ugly web site!
- Let's allow customers the ability to copy a transcript of their chat to their clipboard for pasting in another application. This way, they can have a record of their conversation in case they need to call and yell at us later for something. Let's also give them the ability to print the transcript directly from the application, so that when they sue, they have the documentation they need to win (perhaps we're being a little too good to our customers?).
- The application should of course be Ajax-based and should use the Mootools library to provide that functionality (along with anything else we may need that it offers). It's pretty tough to do a one-on-one chat in HTML and JavaScript without a server component, so we'll need a server in the mix here. We'll code the back-end in Java and Microsoft technologies (ASP specifically), so that at least a majority of developers reading this should be covered.

With those really pretty limited set of requirements in tow, let's get a move on and see how we'll put this thing together, shall we? Well, in fact, let's first look at how we *won't* be doing things and why.

The "Classic" Web Model

In the beginning, there was the Web. And it was good. All manner of catchy new words, phrases, and terms entered the lexicon, and we felt all the more cooler saying them. (Come on, admit it, you felt like Spock the first couple of times you used the word "hypertext" in conversation, didn't you?) *Webapps*, as our work came to be known, were born. These applications were, in a sense, a throwback to years gone by when applications were hosted on "big iron" and were accessed in a time-share fashion, since no processing was done locally on the machine where the user was interacting with the application. They also were in no way, shape, or form as "flashy" as the Visual Basic, PowerBuilder, Delphi, and C++ "fat clients" that followed them (which are still used today, although less so with the advent of webapps).

The webapps that have been built for many years now—indeed are still built today on a daily basis—have one big problem: they are, by and large, redrawing the entire screen each time some event occurs. They are intrinsically server-centric to a large extent. When the user does something (beyond some trivial things that can occur client side such as mouse-over effects and such), a server must get involved. It has to do some processing, and then redraw what the user sees to incorporate the applicable updated data elements. This is, as I'm sure you have guessed, highly inefficient.

This model of application development is what I refer to as the "classic" web design pattern, or model (I haven't heard anyone else use the term in this sense before, but I still can't imagine I'm the first!). The classic web model to me means the paradigm where the server, for nearly every user event, redraws the entire screen. This is how webapps have been built for about 15 years now, since the Web first began to be known in a broad sense. Conversely, the term "modern" web model refers to the new mode of developing webapps, where the client is asked to share the load a bit and play a more prominent role in the functioning of the application.

You may be asking yourself, "If we've been doing the classic web thing for so long, and even still do it today, what's wrong with it?" In many ways, absolutely nothing! In fact, there is still a great deal of value to that way of designing webapps. The classic web model is great for largely linear application flows, and it is also a wonderful medium for delivering information in an accessible way. That model makes it is easy for most people to publish information and to even create rudimentary applications with basic user interactions. The classic web model is simple, ubiquitous, and accessible to most people.

It is not, however, an ideal environment for developing complex applications with a lot of user interaction. The fact that people have been able to do so to this point is a testament to the ingenuity of engineers, rather than an endorsement of the Web as an application distribution medium!

It makes sense to differentiate now between a *webapp* and a *web site*, as summarized in Table 12-1. There are really two different purposes served by the Web at large. One is to deliver information. In this scenario, it is very important that the information be delivered in a manner that is readily accessible to the widest possible audience. This means not only people with disabilities who are using screen readers and such devices, but also those using more limited capability devices like cell phones, PocketPCs, and kiosk terminals. In such situations, there tends to be no user interaction, aside from jumping from static document to static document, or at most very little interaction via simple forms. This mode of operation for the Web can be classified as web sites.

Table 12-1. *Summary Comparison of Webapps vs. Web Sites*

Webapp	Web Site
Designed with much greater user interaction in mind	Very little user interaction, aside from navigation from document to document
Main purpose is to perform some function or functions, usually in real time, based on user inputs	Main purpose is to deliver information, period
Uses techniques that require a lot more of the clients accessing them	Tends to be created for the lowest common denominator in terms of client capabilities
Accessibility tends to take a back seat to functionality out of necessity and the simple fact that it's hard to do complex and yet accessible webapps	Accessibility is usually considered and implemented to allow for the widest possible audience
Tends to be more event-based and nonlinear	Tends to be somewhat linear with a path the user is generally expected to follow, with only minor deviations

Webapps, on the other hand, have a wholly different focus. They are not concerned with simply presenting information, but in performing some function based on what the user does and what data the user provides. The user can be another automated system in many cases, but usually we are talking about a flesh-and-blood human being. Webapps tend to be more complex and much more demanding of the clients that access them. In this case, *clients* refer to web browsers.

This is another problem with the classic model: in order to maintain accessibility for the widest possible audience, you generally need to design to the lowest common denominator, which severely limits what you can do. Let's think a moment about what the lowest common denominator means in this context. Consider what you could and could not use to reach the absolute widest possible audience out there today. Here is a list of what comes to mind:

Client-side scripting: No, you couldn't use this because many mobile devices do not yet have scripting support, or are severely limited. This does not even consider those people on full-blown PCs who simply choose to disable scripting for security or other reasons.

CSS: You could use style sheets, but you would have to be very careful to use an older CSS specification to ensure most browsers would render styles properly—none of the fancier CSS 2.0 capabilities, for instance.

Frames: No, frames are not universally supported, especially on many portable devices. Even when they are supported, you need to be careful because a frame is essentially like having another browser instance in terms of memory (and in some cases, it very literally *is* another browser instance), and this can be a major factor in mobile devices.

Graphics: Graphics can be tricky in terms of accessibility because they tend to convey more information than an `alt` attribute can. So, some of the meaning of the graphic can easily be lost for those with disabilities, no matter how vigilant you are to help them.

Newer HTML specs: Many people out there are still using older browsers that may not even support HTML 4.01, so to be safe, you will probably want to code to HTML 3.0. Obviously, you will lose some capabilities in doing so.

Probably the most important element here, certainly for our purposes in this book, is the lack of client-side scripting. Without client-side scripting, many possibilities are not available to you as a developer. Most important is the fact that you have virtually no choice but to have the server handle every single user interaction and to respond with a completely redrawn view. You may be able to get away with some meta refreshes in frames in some cases, or perhaps other tricks of the trade, but frames may not be supported, so you might not even have that option!

You may be wondering, "What is the problem with the server rendering entire pages?" Certainly, that approach has benefits, and the inherent security of being in complete control of the runtime state of the application (the user can't hack the code) is a big one. Not having to incur the delay of downloading the code to the client is another. However, there are indeed some problems that in many cases overshadow the benefits. Perhaps the most obvious is the load on the server. Asking a server to do all this work on behalf of the client many times over across a number of simultaneous requests means that the server needs to be more robust and

capable than otherwise might be required. This all translates to dollars and cents in the long run, because you'll need to purchase more server power to handle the load.

Now, many people have the "just throw more hardware at it" mentality, and we are indeed in an age where that works most of the time. But that is much like saying that because we can throw bigger and bigger engines in cars to make them go faster, then that's exactly what we should always do when we need or want more speed. In fact, we can make cars go faster by making a smaller engine more efficient in design and execution, which in many ways is much more desirable—that is, if you like clean, fresh air to breathe! Perhaps an even better metaphor would be to say it is like taking a midsized car and continually adding seats tied to it around the outside to allow for more people to ride "in" the car, rather than trying to find a more efficient way for them to get where they are going. While this duct-tape solution might work for a while, eventually someone is going to fall off and get crushed by the 18-wheeler driving behind us!

Another problem with the server-does-it-all approach is that of network traffic. Network technology continues to grow in leaps and bounds at a fantastic rate. Many of us now have broadband connections, which we could not fully saturate if we tried (and I for one have tried!). However, that does not mean we should have applications that are sending far more information per request than necessary. We should still strive for thriftiness, should we not?

The other big problem with the classic approach is simply how the user perceives the application. When the server needs to redraw the entire screen, it generally results in a longer wait time to see the results, not to mention the visual redrawing that many times occurs in webapps, flickering, and things of that nature that users universally dislike in a big way. Users also do not like losing everything they entered when something goes wrong, which is another common failing of the classic model.

At the end of the day, the classic model still works well on a small scale, and for delivering mostly static information, but it doesn't scale very well and it doesn't deal with the dynamic nature of the Web today nearly as well. In this context, *scale* refers to added functionality in the application, not simultaneous request handling capability (although it is quite possible that is in play, too). If things do not work as smoothly, or if breakages result in too much loss, or if perceived speed is diminished, then the approach didn't scale well.

The classic model will continue to serve us well for some time to come in the realm of web sites, but in the realm of webapps—the realm you are likely interested in if you are reading this book—its demise is at hand, and its slayer is the hero of our tale: Ajax!

Ajax

Ajax came to life, so to speak, at the hands of one Jesse James Garrett of Adaptive Path (`http://www.adaptivepath.com`). I am fighting my natural urge to make the obvious outlaw jokes here! Mr. Garrett wrote an essay in February of 2005 (you can see it at `http://www.adaptivepath.com/publications/essays/archives/000385.php`), in which he coined the term *Ajax*.

Ajax, as I'd be willing to bet my dog you know already (I don't have a dog, but I will buy one and give it to you if you don't know what Ajax stands for—OK, not really) stands for Asynchronous JavaScript and XML. The interesting thing about Ajax, though, is that it doesn't have to be asynchronous (but virtually always is), doesn't have to involve JavaScript (but virtually always does), and does not need to use XML at all (but probably does half the time). In fact, one of the

most famous Ajax examples, Google Suggest (`http://www.google.com/webhp?complete=1&hl=en`), doesn't pass back XML at all! The fact is that it does not even pass back data per se; it passes back JavaScript that contains data (which, if you've read Chapter 5 already, is actually something you've seen: the Yahoo and Google web services used there returned data wrapped in a JavaScript function call).

Ajax to the rescue! (Now you'll always know what code/architecture would look like personified as a superhero.)

The Ajax Frame of Mind

Ajax has, at its core, an exceedingly simple, and by no stretch of the imagination original, concept: it is not necessary to refresh the entire contents of a web page for each user interaction, or each "event," if you will. When the user clicks a button, it is no longer necessary to ask the server to render an entirely new page, as is the case with the classic web model. Instead, you can define regions on the page to be updated, and have much more fine-grained control over user events as well. No longer are you limited to simply submitting a form or navigating to a new page when a link is clicked. You can now do something in direct response to a non-submit button being clicked, a key being pressed in a text box—in fact, to any event happening!

The server is no longer completely responsible for rendering what the user sees; some of this logic is now performed in the user's browser. In fact, in a great many cases, it is considerably better to simply return a set of data and not a bunch of markup for the browser to display. As we traced along my admittedly rough history of application development, you saw that the classic model of web development is, in a sense, an aberration to the extent that we actually had it right before then!

Ajax is, most important, a way of thinking, an approach to application development, and a mind-set, not a specific technology. The interesting thing about Ajax is that it is in no way, shape, or form *new*; only the term used to describe it is. I was reminded of this fact at the Philadelphia Java Users Group. A speaker by the name of Steve Banfield was talking about Ajax, and he said (paraphrasing from memory), "You can always tell someone who has actually done Ajax because they are pissed that it is all of a sudden popular." This could not be truer! I was one of those people doing Ajax years and years ago. I just never thought what I was doing was anything special and hence did not give it a "proper" name. Mr. Garrett holds that distinction. It also would not be Ajax in a form we recognize today, but that's because the technological approach may have changed, but the underlying mind-set hasn't, and that's the key point in my opinion.

When you get into the Ajax frame of mind, which is what we are really talking about, you are no longer bound by the rules of the classic web model. You can now take back at least some of the power the fat clients offered, while still keeping the benefits of the Web. Those benefits begin, most important perhaps, with the ubiquity of the web browser.

Have you ever been at work and needed to give a demo of your new fat client app (for example, a Visual Basic app) on a machine you never touched before? Ever had to do it in the boardroom in front of top company executives? Ever had that demo fail miserably because of some DLL conflict you couldn't possibly anticipate? You are a developer, so the answer to all of those questions is likely yes (unless you work in the public sector, and then you probably don't present to corporate executives, but you get the point). If you have never done Windows development, you may not have had these experiences. You will have to take my word for it when I say that such situations were, for a long time, much more common than any of us would have liked. With a web-based application, this is generally not a concern. Ensure the PC has the current browser and version, and off you go 98% of the time.

We've all been there. Live demos and engineers do not mix!

The other major benefit of a webapp is distribution. No longer do you need a three-month shakedown period to ensure your new application does not conflict with the existing suite of corporate applications. An app running in a web browser, security issues aside, will not affect, or be affected by, any other application on the PC (and I am sure we all have war stories about exceptions to that, but they are just that: exceptions).

Of course, you probably knew those benefits already, or you wouldn't be interested in web development in the first place.

Sounding good so far, huh? It's not all roses in Ajax land, however. Ajax is not without its problems. Some of them are arguably only perceived problems, but others are concrete.

Accessibility and Similar Concerns

First and foremost, in my mind at least, is the issue of accessibility. You will lose at least some accessibility in your work by using Ajax because devices like screen readers are designed to read an entire page, and since you will no longer be sending back entire pages, screen readers will have trouble. My understanding is that some screen readers can deal with Ajax to some degree, largely depending on how Ajax is used (if the content is literally inserted into the DOM, it makes a big difference). In any case, extreme caution should be used if you know people with disabilities are a target audience for your application, and you will seriously want to consider (and test!) whether Ajax will work in your situation. I am certain this problem will be addressed better as time goes on, but for now, it is definitely a concern. In the meantime, here are some things you can do to improve accessibility:

Let users know about the dynamic updates. Put a note at the top of the page that says the page will be updated dynamically. This will give users the knowledge that they may need to periodically request a reread of the page from the screen reader to hear the dynamic updates.

Use `alert()` *pop-ups.* Depending on the nature of the Ajax you are using on a page, use `alert()` pop-ups when possible, as these are read by a screen reader. This is a reasonable enough suggestion for things like Ajax-based form submission that will not be happening too frequently, but obviously if you have a timed, repeating Ajax event, this suggestion would not be a good one.

Add visual cues. Remember that it is not only the blind who have accessibility needs; it can be sighted people as well. For them, try to use visual cues whenever possible. For instance, briefly highlighting items that have changed can be a big help. Some people call this the "yellow fade effect", which I talked about back in Chapter 1 as one kind of effect that actually enhances the user experience. The idea is to highlight the changed item in yellow, and then slowly fade it back to the nonhighlighted state. Of course, it does not have to be yellow, and it does not have to fade, but the underlying concept is the same: highlight changed information to provide a visual cue that something has happened. Remember that changes caused by Ajax can sometimes be very subtle, so anything you can do to help people notice them will be appreciated.

Another disadvantage of Ajax to many people is added complexity. Many shops do not have in-house the client-side coding expertise Ajax requires (the use of toolkits that make it easier notwithstanding). The fact is, errors that occur on the client side are still, by and large, harder to track down than server-side problems, and Ajax does not make this any simpler. For example, View Source does not reflect changes made to the DOM.

Another issue is that Ajax applications will many times do away with some time-honored web concepts, most specifically back and forward buttons and bookmarking. Since there are no longer entire pages, but instead fragments of pages being returned, the browser cannot bookmark things in many cases. Moreover, the back and forward buttons cease to have the same meanings because they still refer to the last URL that was requested, and Ajax requests almost never are included (requests made through the `XMLHttpRequest` are not added to history, for example, because the URL generally does not change, especially when the method used is POST).

Ajax: A Paradigm Shift for Many

Ajax does, in fact, represent a paradigm shift for some people (even most people, given what most webapps are today) because it can fundamentally change the way you develop a webapp. More important perhaps is that it represents a paradigm shift for the *users*, and, in fact, it is the users who will drive the adoption of Ajax. Believe me, you will not long be able to ignore Ajax as a weapon in your arsenal.

Put a non-Ajax webapp in front of some users, and then put that same app using Ajax techniques in front of them, and guess which one they are going to want to use all day, nine times out of ten? The Ajax version!

Users can immediately see the increased responsiveness of an Ajax application, and will notice that they no longer need to wait for a response from the server while they stare at a spinning browser logo, wondering if anything is actually happening. They will see that the application alerts them on the fly of error conditions they would have to wait for the server to tell them about in the non-Ajax webapp. They will see functionality like type-ahead suggestions, instantly sortable tables, and master/detail displays that update in real time—things that they would *not* see in a non-Ajax webapp. They will see maps that they can drag around as they can in the full-blown mapping applications they spent $80 on in the past. All of these things will be obvious advantages to the user. Users have become accustomed to the classic webapp model, but when confronted with something that harkens back to those fat-client days in terms of user-friendliness and responsiveness, there is almost an instantaneous realization that the Web as they knew it is dead, or at least should be!

If you think about many of the big technologies to come down the pipe in recent years, it should occur to you that we technology folks, rather than the users, were driving many of them. Do you think a user ever asked for an Enterprise JavaBean (EJB)–based application? No, we just all thought it was a good idea (how wrong we were there!). What about web services? Remember when they were going to fundamentally change the way the world of application construction worked? Sure, we are using them today, but are they, by and large, much more than an interface between cooperating systems? Not usually. Whatever happened to Universal Description, Discovery, and Integration (UDDI) directories and giving an application the ability to find, dynamically link to, and use a registered service on the fly? How good did that sound? To us geeks, it was the next coming, but it didn't even register with users.

Ajax is different, though. Users can see the benefits. The advantages are very real and very tangible to them. In fact, we as technology people, especially those of us doing Java web development, may even recoil at Ajax at first, because more is being done on the client, which is contrary to what we have been drilling into our brains all these years. After all, we all believe scriptlets in JavaServer Pages (JSPs) are bad, eschewing them in favor of custom tags. The users do not care about elegant architectures, separation of concerns, and abstractions allowing for code reuse. Users just want to be able to drag the map around in Google Maps (`http://maps.google.com`) and have it happen in real time, without waiting for the whole page to refresh.

The difference is clear. Users want it, and they want it now (come now, we're adults here!)

RICH INTERNET APPLICATIONS (RIAS)

Ajax is not the only new term floating around these days that essentially refers to the same thing. You may have also heard of Web 2.0 and rich Internet applications (RIAs). RIA is a term I particularly like. Although there is no formal definition with which I am familiar, most people get the gist of its meaning without having to Google for it.

In short, the goal of an RIA is to create a "rich" application that is web-based. The application runs in a web browser but looks, feels, and functions more like a typical fat-client application than a typical web site. Things like partial-page updates are taken for granted, and hence Ajax is always involved in RIAs (although what form of Ajax is involved can vary; indeed, you may not find the `XMLHttpRequest` object, the prototypical Ajax solution, lurking about at all!). These types of applications are always more user-friendly and better received by the user community they serve. In fact, your goal in building RIAs should be for users to say, "I didn't even know it was a webapp!"

Gmail (`http://gmail.google.com`) is a good example of an RIA, although it isn't perfect. While it has definite advantages over a typical web site, it still looks and feels very much like a web page. By the way, Google developers have probably done more to bring Ajax to the forefront of people's minds than anyone else. They were not the first to do it, or even the best necessarily, but they certainly have created some of the most visible examples, and have really shown people what possibilities Ajax opens up.

The "Hello World" of Ajax Examples

So, enough theoretical musings. What does Ajax look like in the flesh? Figure 12-1 shows a very simple sample application on the screen (don't expect much here, folks!).

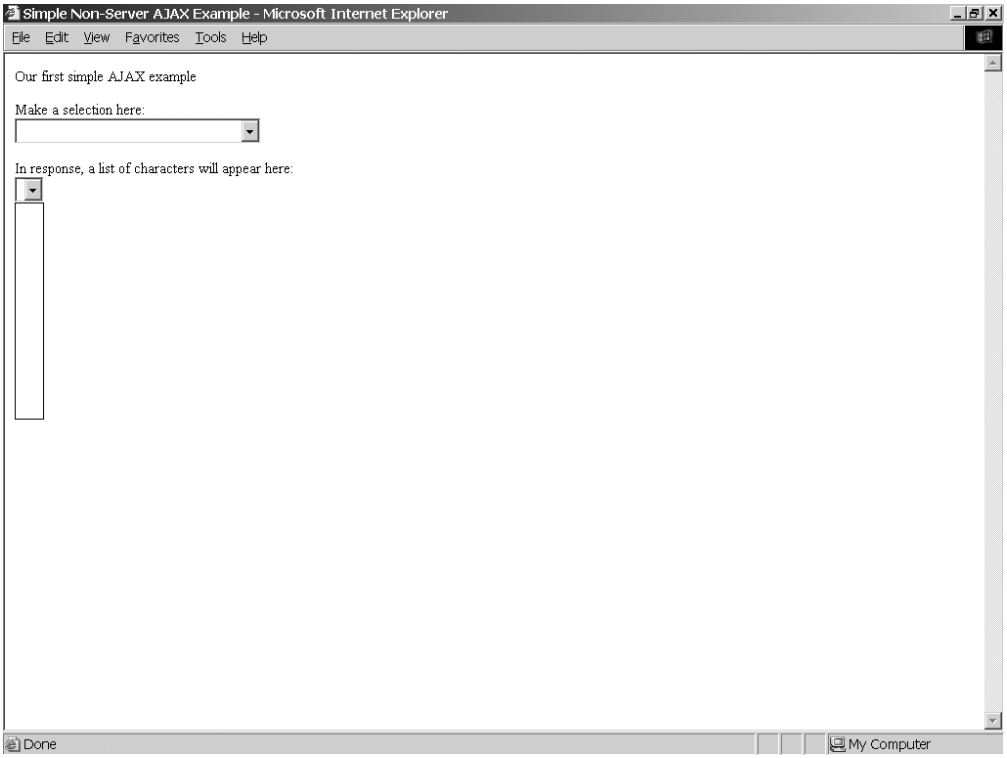


Figure 12-1. *Note that there is no content in the second drop-down list because nothing has been selected yet in the first one.*

As you can see, there is no content in the second drop-down list initially. This list will be dynamically populated once a selection is made in the first drop-down list, as shown in Figure 12-2.

Figure 12-2 shows that when a selection is made in the first drop-down list, the contents of the second are dynamically updated. In this case, you see characters from the greatest television show ever, *Babylon 5*. (Don't bother arguing, you know I'm right. And besides, you'll get your chance to put in your favorites later!) Now let's see how this "magic" is accomplished.

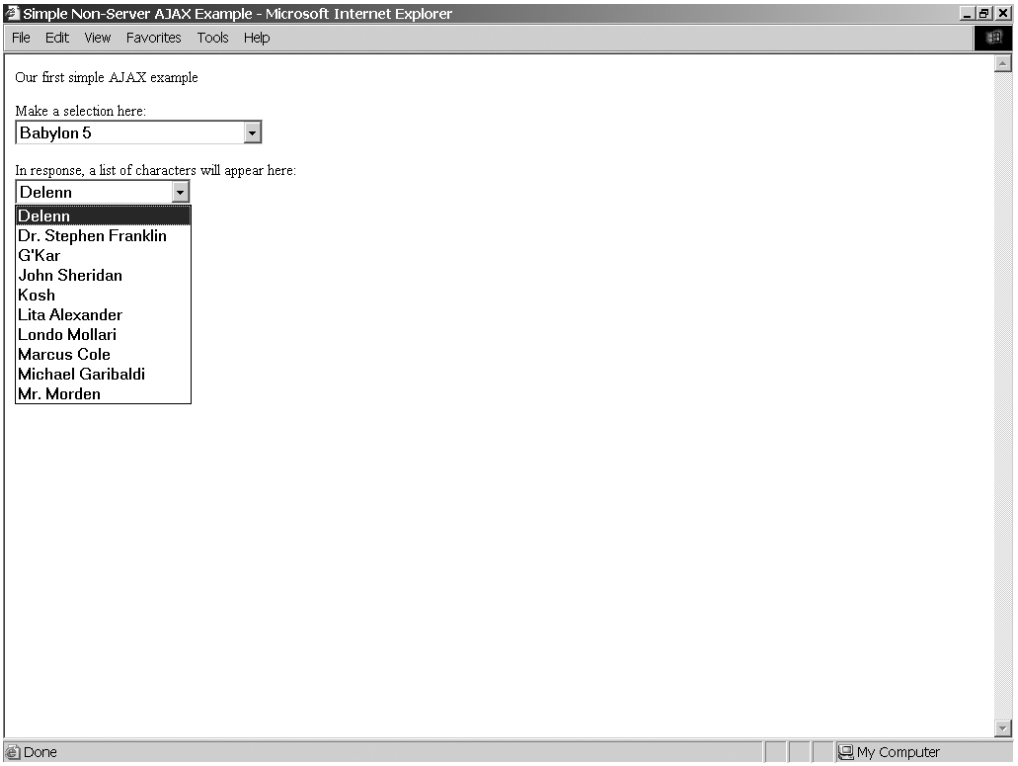


Figure 12-2. *A selection has been made in the first drop-down list, and the contents of the second have been dynamically created from what was returned by the "server."*

Listing 12-1 shows the first page of our simple Ajax example, which performs a fairly typical Ajax-type function: populate one `<select>` box based on the selection made in another. This comes up all the time in web development, and the "classic" way of doing it is to submit a form—whether as a result of the user clicking a button or by a JavaScript event handler—to the server and let it render the page anew with the updated contents for the second `<select>`. With Ajax, none of that is necessary.

Listing 12-1. *Our First Real Ajax Application*

```
<html>

  <head>

    <title>Simple Non-Server AJAX Example</title>

    <script>

      // This is a reference to an XMLHttpRequest object.
      xhr = null;
```

```
    // This function is called any time a selection is made in the first
    // <select> element.
    function updateCharacters() {
      // Instantiate an XMLHttpRequest object.
      if (window.XMLHttpRequest) {
        // Non-IE.
        xhr = new XMLHttpRequest();
      } else {
        // IE.
        xhr = new ActiveXObject("Microsoft.XMLHTTP");
      }
      xhr.onreadystatechange = callbackHandler;
      url = document.getElementById("selShow").value + ".htm";
      xhr.open("post", url, true);
      xhr.send(null);
    }

    // This is the function that will repeatedly be called by our
    // XMLHttpRequest object during the life cycle of the request.
    function callbackHandler() {
      if (xhr.readyState == 4) {
        document.getElementById("divCharacters").innerHTML =
          xhr.responseText;
      }
    }

  </script>

</head>

<body>

  Our first simple AJAX example
  <br><br>

  Make a selection here:
  <br>
  <select onChange="updateCharacters();" id="selShow">
    <option value=""></option>
    <option value="b5">Babylon 5</option>
    <option value="bsg">Battlestar Galactica</option>
    <option value="sg1">Stargate SG-1</option>
    <option value="sttng">Star Trek The Next Generation</option>
  </select>
  <br><br>
```

```
    In response, a list of characters will appear here:
    <br>
    <div id="divCharacters">
      <select></select>
    </div>

  </body>
</html>
```

Let's walk through the code and see what's going on. Note that this is not meant to be a robust, production-quality piece of code. It's meant to give you an understanding of basic Ajax techniques, nothing more. There's no need to write me about all the flaws you find!

First things first: the markup itself. In our `<body>`, we have little more than some text and two `<select>` elements. Notice that they are not part of a `<form>`. You'll find that forms tend to have less meaning in the world of Ajax. Many times, you'll begin to treat all your form UI elements as top-level objects along with all the other elements on your page (in the `<body>` anyway).

The first `<select>` element is given the ID `selShow`. This becomes a node in the DOM of the page. You'll notice the JavaScript event handler attached to this element. Any time the value of the `<select>` changes, we'll be calling the JavaScript function named `updateCharacters()`. This is where all the "magic" will happen. The rest of the element is nothing unusual. I have simply created an `<option>` for some of my favorite shows.

After that is another `<select>` element—well, sort of. It's actually an empty `<select>` element, but wrapped in a `<div>`. You'll find that probably the most commonly performed Ajax function is to replace the contents of some `<div>`. That is exactly what we'll be doing here. In this case, what will be returned by the "server" (more on that in a minute) is the markup for our `<select>` element, complete with `<option>` elements listing characters from the selected television show. So, when you make a show selection, the list of characters will be appropriately populated, and in true Ajax form, the whole page will not be redrawn, but only the portion that has changed—the second `<select>` element in this case (or more precisely, the `<div>` that wraps it).

Let's quickly look at our mock server. Each of the shows in the first `<select>` has its own HTML file that, in essence, represents a server process. You have to take a leap of faith here and pretend a server was rendering the response that is those HTML pages. They all look virtually the same, so I'll show only one as an example. Take a look at Listing 12-2.

Listing 12-2. *Sample Response Listing Characters from the Greatest Show Ever (Babylon 5)*

```
<select>
  <option>Delenn</option>
  <option>Dr. Stephen Franklin</option>
  <option>G'Kar</option>
  <option>John Sheridan</option>
  <option>Kosh</option>
  <option>Lita Alexander</option>
  <option>Londo Mollari</option>
  <option>Marcus Cole</option>
  <option>Michael Garibaldi</option>
  <option>Mr. Morden</option>
</select>
```

As expected, it really is nothing but the markup for our second `<select>` element.

So, now we come to the part that does all the work here: our JavaScript function(s). First is the `updateCharacters()` function. This basic code will very soon be imprinted on the insides of your eyelids if you work with Ajax for any length of time, because it's the prototypical Ajax function. Let's tear it apart, shall we?

The first thing we need, as one would expect, is an `XMLHttpRequest` object, which is the object at the core of Ajax as most people know it. This object, a creation of Microsoft (believe it or not!), is nothing more than a proxy to a socket. It has a few (very few) methods and properties, but that is one of the benefits. It really is a very simple beast.

Notice the branching logic here. It turns out that getting an instance of the `XMLHttpRequest` object is different in IE than in any other browser. Now, before you get your knickers in a knot and get your anti-Microsoft ire up, note that Microsoft invented this object, and it was the rest of the world that followed. So, while it would be nice if Microsoft developers updated their API to match everyone else's, it isn't their fault we need this branching logic! The others could just as easily have duplicated what Microsoft did exactly, too, so let's not throw stones here—we're all in glass houses on this one.

Late-breaking news: IE7 implements `XMLHttpRequest` as a native object that can even work when ActiveX is disabled! This means that the branching code I'm talking about here, theoretically, isn't necessary. However, I've been reading quite a lot about problems with this, and questions about whether the IE team really implemented a native version or just cleverly wrapped the ActiveX version in a JavaScript façade. There is also talk of performance issues that accompanies this. So, at the end of the day, I suggest sticking with the tried-and-true method for a while longer. Really though, you'll likely be using some sort of library for your Ajax functionality most of the time anyway, so you'll largely be insulated from these concerns and won't so much care whether it's truly native or not. Still, this certainly is information worth noting.

This is probably a good time to point out that `XMLHttpRequest` is pretty much a de facto standard. It is also being made a true W3C standard as well, but for now it is not. It is safe to assume that any "modern" browser—that is, a desktop web browser that is no more than a few versions old—will have this object available. More limited devices—such as PocketPCs, cell phones, and the like—may not have it. But by and large, `XMLHttpRequest` is a pretty ubiquitous little piece of code.

Continuing on in our code review, once we have an `XMLHttpRequest` object instance, we assign the reference to it to the variable `xhr` in the global page scope. Think about this for just a minute. What happens if more than one `onChange` event fires at close to the same time? Essentially, the first will be lost because a new `XMLHttpRequest` object is spawned, and `xhr` will point to it. Worse still, because of the asynchronous nature of `XMLHttpRequest`, a situation can arise where the callback function for the first request is executing when the reference is nulled, which means that callback would throw errors due to trying to reference a null object. If that were not bad enough, this will be the case only in some browsers, but not all (although my research indicates most would throw errors), so it might not even be a consistent problem. Remember that I said this was not robust, production-quality code! This is a good example of why. That being said, it is actually many times perfectly acceptable to simply instantiate a new instance and start a new request.

Think about a fat client that you use frequently. Can you spot instances where you can kick off an event that, in essence, cancels a previous event that was in the process of executing? For example, in your web browser, can you click the Home button while a page is loading, thereby causing the page load to be prematurely ended and the new page to begin loading? Yes, you

can, and that is essentially what happens by starting a new Ajax request using the same reference variable. It is not an unusual way for an application to work, and sometimes it is downright desirable.

The next step we need to accomplish is telling the `XMLHttpRequest` instance what callback handler function to use. Ajax requests have a well-defined and specific life cycle, just like any HTTP request (and remember that is all an Ajax request is at the end of the day!). This cycle is defined as the transitions between ready states (hence the property name, `onreadystatechange`). At specific intervals in this life cycle, the JavaScript function you name as the callback handler will be called. For instance, when the request begins, your function will be called. As the request is chunked back to the browser, in most browsers at least (IE being the unfortunate exception), you will get a call for each chunk returned (think about those cool status bars you can finally do with no complex queuing and callback code on the server!). Most important for us in this case, the function will be called when the request completes. We will see this function in just a moment.

The next step is probably pretty obvious: we need to tell the object which URL we want to call. We do this by calling the `open()` method of the object. This method takes three parameters: the HTTP method to perform, the URL to contact, and whether we want the call to be performed asynchronously (`true`) or not (`false`). Because this is a simple example, each television show gets its own HTML file pretending to be the server. The name of the HTML file is simply the value from the `<select>` element with `.htm` appended to the end. So, for each selection the user makes, a different URL is called. This is obviously not how a real solution would work. The real thing would likely call the same URL with some sort of parameter to specify the selected show. But some sacrifices were necessary to keep the example simple and to not need anything on the server side of things.

The HTTP method can be any of the standard HTTP methods: GET POST, HEAD, and so on. Most of the time, you will be passing GET or POST. The URL is self-explanatory, except for one detail: if you are doing a GET, you must construct the query string yourself and append it to the URL. That is one of the drawbacks of `XMLHttpRequest`. You take full responsibility for marshalling and unmarshalling data sent and received. Remember that it is in essence just a very thin wrapper around a socket. This is where any of the numerous Ajax toolkits can come in quite handy, as you'll see when we use the Mootools library for this chapter's chat application.

Once we have the callback registered with the object and we have told it what we're going to connect to and how, we simply call the `send()` method. In this case, we are not actually sending anything, so we pass null. One thing to be aware of is that you can call `send()` with no arguments in IE, and it will work, but it won't work in Firefox (at least this was the case with my tests). Null works in both browsers, though, so null it is.

Of course, if you actually had some content to send, you would do so here. You can pass a string of data into this method, and the data will be sent in the body of the HTTP request. Many times, you will want to send actual parameters, and you do so by constructing essentially a query string in the typical form `var1=val1&var1=val1` and so forth, but without the leading question mark. Alternatively, you can pass in an XML DOM object, and it will be serialized to a string and sent. Lastly, you could send any arbitrary data you want. If a comma-separated list does the trick, you can send that. Anything other than a parameter string will require you to deal with it; the parameter string will result in request parameters as expected.

So far, I've described how a request is sent. It is pretty trivial, right? Well, the next part is what can be even more trivial, or it can be much more complex. In our example, it is the former. I am referring to the callback handler function. Our callback handler function does very little.

First, it checks the `readystate` of the `XMLHttpRequest` object. Remember I said this callback will be called multiple times during the life cycle of the request? Well, the `readystate` code you will see will vary with each life cycle event. For the purposes of this example, we are interested in code 4, which indicates the request has completed. Notice that I didn't say completely *successfully*! Regardless of the response from the server, the `readystate` will be 4. Since this is a simple example, we don't care what the server returns. If an HTTP 404 error (page not found) is received, we don't care in this case. If an HTTP 500 error (server processing error) occurs, we still do not care. The function will do its thing in any of these cases. I repeat my refrain: this is not an industrial-strength example!

When the callback is called as a result of the request completing, we simply set the `innerHTML` property of the `<div>` on the page with the ID `divCharacters` to the text that was returned. In this case, the text returned is the markup for the populated `<select>`, and the end result is the second `<select>` is populated by characters from the selected show.

Now, that wasn't so bad, was it?

■ **Tip** For a fun little exercise, and just to convince yourself of what is really going on, I suggest adding one or two of your own favorite shows in the first `<select>`, and creating the appropriately named HTML file to render the markup for the second `<select>`.

One other point I should make is that in previous chapters, such as in Chapter 5, you saw what I said was essentially Ajax. However, after seeing this simple application, you may be confused. Let me clear up that confusion right now. Remember that I've been trying to enforce the idea that Ajax is more about approach than it is implementation. Just because `XMLHttpRequest` isn't in the equation doesn't mean what you see isn't Ajax. The dynamic `<script>` tag technique is, in my opinion, as much Ajax as anything you see in this chapter. The idea of the client doing more work and of the server not rendering full views any more is what matters. So, while this chapter isn't necessarily the first exposure to Ajax you've had in this book, it's the first example of what most people mean when they say Ajax. It's a bit of conceptual/semantical banter I suppose, but a point I believe is worth making.

If all of this seemed like an attempt to brainwash you about what Ajax is and why it's good, that is because, in a sense, it was! Ajax can seem to some people like a really bad idea, but those people tend to see only the problems and completely ignore the benefits. Because of my belief that Ajax is more about philosophy and thought process than it is about specific technologies, it is important to sell you on the ideas underlying it. It is not enough to simply show you some code and hope you agree! Ajax opens up the Web to fulfilling a lot more of the promise so many people hold for it, and I feel that web developers should understand why it's important and put the underlying concepts to good use.

JSON

Recall when I said that Ajax doesn't require XML be returned or passed to the server at all? As it turns out, XML isn't really even the most common data format. That distinction most likely goes to something called JSON, or JavaScript Object Notation, which I introduced in Chapter 2.

The acronym JSON is, I feel, a bit of a misnomer because, while it *can* represent an object, it often does not, but that is really just a name thing. The basic idea is that it is a way to structure data that is returned to a caller.

JSON is billed as being a lightweight, system-independent data interchange format that is easy for humans to read, easy for computers to parse, and easy for computers to generate. It uses a syntax that will be immediately familiar to most programmers that have any experience with a C-family language (including Java and JavaScript). It is built on two basic concepts that are pretty much universal in programming: a collection of name/value pairs (maps, keyed lists, associative arrays, and so on) and an order list of values (lists or arrays).

Well, enough CompSci gobbledygook! Let's see what JSON looks like.

```
{"firstName":"Frank","lastName":"Zammetti","age":"34"}
```

Really? Is that all there is to it? I wish I could try to impress you with my advanced knowledge and say there is more to JSON than that, but no, that actually is all there is to it! As you can see, it looks similar to an array in Java, but not quite, because two elements are defined between each delimiter. The item to the left of the colon is the key, and the value to the right is the value. Each pair is separated by a comma, and the whole thing is wrapper in curly braces. It's simple!

Where it gets really pretty cool though is when you want to handle a JSON response in JavaScript. All you have to do is this:

```
var json = eval("(" + myJSONString + ")");
```

The result of this, assuming `myJSONString` contained some valid JSON in the form discussed a few sentences ago, is that a new variable, `json`, will be available to your script. From then on, if you want to get the first name in the response, you simply do this:

```
alert(json.firstName);
```

Really, that's it! What actually happened is the `eval()` call created the `json` variable, giving it the value of the response. The `json` variable is an associative array in JavaScript, so you can access the members in the same way you would access members of any other associative array. Neat, isn't it?

Although this chapter's project does not do it, you can send JSON to the server as well. If you go to `http://www.json.org`, you will find some libraries for a number of different languages that help you generate and parse JSON. Of course, we're only talking about generating and parsing a string here. Ultimately, it certainly is not rocket science, as you will see when we get to that code later. However, do keep in mind that even though JSON is quite simple at its core, because you can nest elements within one another, and have arrays of elements, it can actually become a bit of a pain to generate manually in some cases. Think of serializing an entire object graph to JSON, for instance. While the JSON itself may not be terribly challenging to understand, the fact is that writing the code to generate it could be a bit more of a challenge. In such situations, you would be wise to look for help in the form of libraries and existing code to make your job a bit easier.

I should mention that JSON is a general-purpose messaging format, and as such, you can use it quite effectively outside Ajax work. Many people have actually taken to it much more than XML, because it is less verbose but tends to be similarly human-readable. I am sure we have all seen "bad" XML that is difficult to comprehend. Likewise, you can make JSON difficult to understand if you try. For example, in my previous book on Ajax, I showed an Ajax-based game that would return a chunk of JSON like the following:

```
{"dm":"false","pn":"Aragorn The Weak","ht":"100","hp":"1","gp":"10","iu":"true",
"vu":"true","di":"false","wn":"false","ec":"false","mo":"o","es":"false","md":"g
gggggggssss[[gggggggggggggggggg([[[[[[[[[[gggggggg[gggggggGgg[[[[ggggggggg[[[[
[(ggggg[[[[[[[[[[[[[[[[[[g^ggggg[[[[ggggggggg[[gggggggggg[gggggggggggfgggggggggggfgggg
ggggggg"}
```

That does not look terribly readable to me! The names of the elements are obviously not meant for human consumption. Although you can probably guess quite a few of the elements just by knowing a little about the game (do feel free to buy that book, *Practical Ajax Projects with Java Technology*, ISBN: 1-59059-695-1, if you are interested), you may not be able to guess all of them. The reason this is the case here is that, for a game, you generally want things to happen as quickly as possible. Therefore, the choice was to make the JSON messages readable to a human, who would likely never have to read them except perhaps for debugging purposes, or make them as small and efficient as possible so as to (a) not take too long to generate or parse and (b) not take too long to transmit across the wire.

Most applications tend not to be quite as time-sensitive as a game though, so I would absolutely suggest always making your JSON (or XML for that matter!) as human-readable as possible. Using `displayMessage` instead of `dm` and `playerName` instead of `pn`, for example, is what I would suggest in such a case.

At this point though, you are ready to use JSON, believe it or not! Go forth and be fruitful with your new knowledge!

Mootools

Now that you have a good foundation on which to build with regard to Ajax in general, let's see how the very fine Mootools library (`http://mootools.net`) makes it so much easier and cleaner, and less error-prone than what you saw in the little sample application in Listing 12-1.

With a name like Mootools, you would think it invites all sorts of ridicule, but that would be far from what it deserves! Mootools is a lightweight, modular JavaScript framework that covers most of the bases a modern JavaScript developer would need. Mootools is constructed of a number of modules, including the Core module (the base Mootools module), the Native module (where you can find basic JavaScript extensions and utilities), the Remote module (where things like Ajax lives), and the Effects modules (where UI FX and such are found).

One of the coolest things about Mootools is its download page. When you go to it, you will be presented with a list of the available modules and add-ons. You simply check off the ones you want, and the package you selected will then download is a customized version with only those modules you selected, nicely compressed and ready to go. This is incredibly handy and ensures that you get only the code you are really interested in, which gives new meaning to the term lightweight! What's more, as you select modules to include, any other modules that it depends on will automatically be selected. Heck, even if you don't want to download Mootools, the download page is great fun!

■ **Note** The `mootools.js` file in this project includes everything available as of this writing (Mootools v1.0). So, if there's something you want to play with, you don't have to go build your own download if you don't want to—just grab this file in the downloadable source and start playing.

There is obviously a lot to Mootools, and unfortunately, the chat application won't do much more than scratch the surface. So, let's get to scratching right now.

To fire off an Ajax request with Mootools, this is all you have to do:

```
new Ajax("<URL>", {
  postBody :
    Object.toQueryString(
      { "parm1" : $("someID1").value, "parm2" : $("someID2").value }
    ),
  onComplete : function(inResponse) {
    // Do something.
  }
}).request();
```

That's it! Instantiate a new `Ajax` object, passing some parameters to its constructor, and you are off to the races. The first parameter is the URL to call. The next parameter, `postBody`, is the contents that will be POSTed to the URL. You'll notice that Mootools extends some JavaScript classes—`Object` in this case—to offer us the `toQueryString()` method. With this, we can feed it a list of parameters and their values, and a proper query string will be constructed for us. You'll notice, too, the use of the `$()` operator, which you've seen in other chapters. That operator gives us a references to a DOM object whose ID we pass in. Well, Mootools offers an implementation of this function as well, and you can see here it being used to get the value of what is presumably a text field (we guess this because we're going after the `value` attribute).

The last parameter, `onComplete`, is a JavaScript function that will be executed when the request successfully returns. You can do anything you want here, and it does not have to be specified in-line as I show here. You can just reference a function that exists elsewhere. Either way, that's all there is to an Ajax call with Mootools!

Other available options include `evalScripts`, which will evaluate any JavaScript in the response upon its return; `update`, which will automatically insert the response into the named page element; and `evalResponse`, which evaluates the entire response. These are some very handy functions to have available, and it's really nice to not have to write the code yourself!

You should most definitely go rummage through the Mootools documentation (which is pretty good, by the way) and see what's available. It hasn't been around as long as some other libraries, but it definitely has benefited from seeing the mistakes of others, because it gets a great deal right.

Now that we've taken care of the preliminaries, it's time for . . . drum roll please . . . the chat application!

A Preview of the Chat Application

To begin, let's take a peek at the application. Figure 12-3 shows the initial logon screen. The application actually has two different logon screens—one for customers and one for support personnel—but they look very similar (only differing in the text that appears).

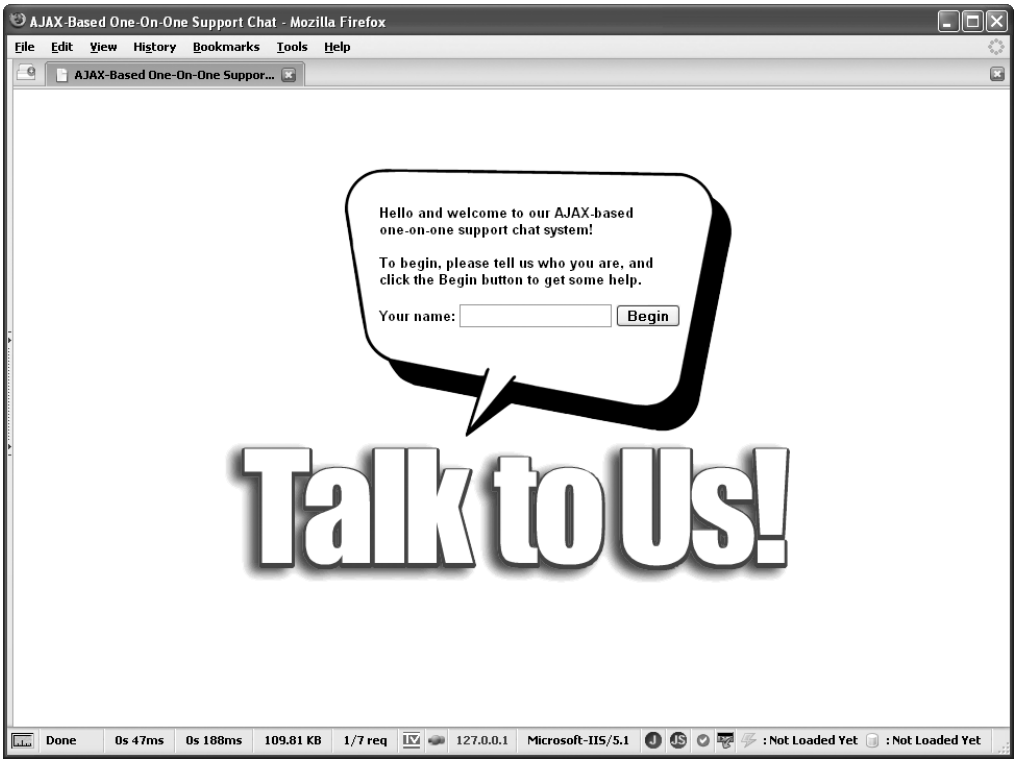


Figure 12-3. *The logon screen for the chat system*

Once you have logged in, you will see the main chat screen, as shown in Figure 12-4. This screen contains several elements. First, we have a greeting near the top, giving a bit of the personal touch. Just to the left of that is a little green talking head, just to give some levity to the customer. Below that is the chat area, where the text of the conversation will appear. To the left of that is the menu of operations the chatter can perform, such as copying the transcript of the chat or exiting the chat session. At the bottom is a constantly updating display of the current date and time. You know how long people can sometimes wait on help lines, and a chat system probably isn't any different, so it's nice to help them keep track of time (although, in this particular implementation, the customer can't log in unless someone is immediately available, but we'll ignore that fact for the sake of the previous sentence making sense).

These two illustrations pretty well cover what the application is all about. There's not much to it, as I said earlier. However, what's behind it is a bit meatier, so let's jump right into that.

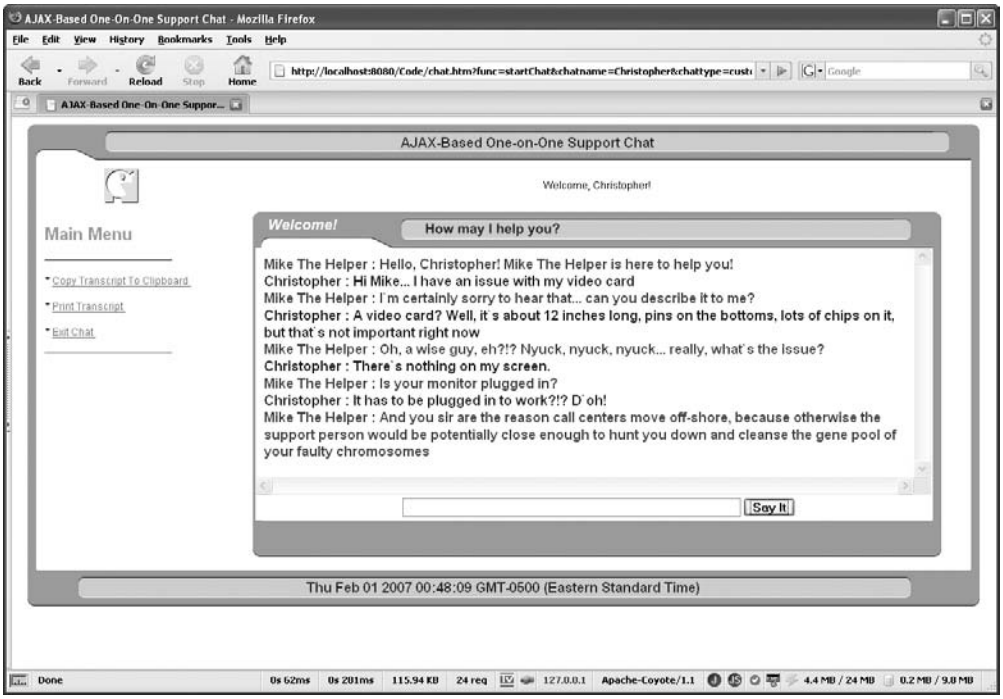


Figure 12-4. *The main chat window where all the "action" (ahem) occurs*

Dissecting the Chat Solution

By now, you undoubtedly know the routine: we start with a look at the directory structure of the application, shown in Figure 12-5, to get a feel for what pieces are involved. However, this final application is, in fact, a bit atypical compared to the rest.

If you're a Java web developer, this will look pretty normal. For those of you who are not in the Java world, let me explain that the `WEB-INF` directory means that this is a Java web application. The single file in there, `web.xml`, defines the application. However, this application is good for the Microsoft folks out there, too.

Since the primary focus of this book is the client side of things, I didn't want to get very far at all into the server side of it. I needed to make the server side as simple and easy as possible, and that means eliminating "extra" steps like compiling code or deploying applications. With that goal in mind, this application is easy to get running. If you're a Java developer, simply copy the entire directory to where applications are deployed in your favorite app server. For instance, if you're using Tomcat (which I very much suggest, by the way), copy it into the `/webapps` directory. Start your server, and you'll be able to access the application immediately (its URL will be something like `http://localhost:8080/chat`, assuming you copied the `chat` directory over, and assuming your installation is listening on port 8080).

If you're a Microsoft technology developer, you're probably familiar with Internet Information Services (IIS). If so, to get this application running there, simply copy the directory into `INetpub/wwwroot`, and you're all set.

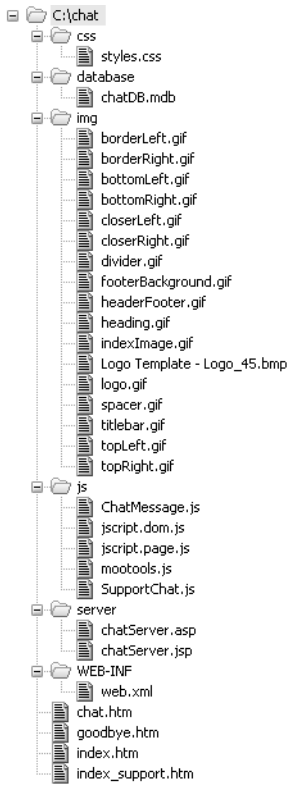


Figure 12-5. *Directory structure of the Chat application*

There actually is one additional configuration step involved in both versions, but it is trivial. It involves specifying which version you're using—ASP or JSP—and pointing to the database file. I'm getting ahead of myself though; we'll get to all that in a bit.

Whether we're talking about the Java version or the Microsoft version, the server side is implemented as a single JSP file or ASP file. This means that there is nothing to compile, no class paths to worry about, or anything like that. Now, before you start throwing things at me from a distance, I'll be the first to admit this probably isn't the way you would implement such an application in an ideal world. It doesn't follow best practices in terms of separation of concerns and the like. However, it *does* mean that you should be up and running in a matter of seconds and there's little chance of anything going wrong. So, please bear with me in terms of overall application architecture, I admit this isn't going to win any prizes! It does, however, have the benefit of working and being pretty easy to understand, so it serves its purpose well. There's no reason you couldn't use this application for real—you just wouldn't want to write your Computer Science thesis on it!

Anyway, thinly veiled apology aside, let's move on. The JSP and ASP files that make up the server component of the application are found in the `server` subdirectory. In the `css` directory is a single `styles.css` file, as you've seen in virtually every application in this book.

In the database directory, you'll find an Access database file named chatDB.mdb. This is another of those choices you probably wouldn't make in an ideal world, but for the sake of a project in a book, it's a good choice. Using an Access database file, and referencing it directly as you'll see when we get to the code, means there is no data source setup to worry about, which again should serve to make getting it running easier.

In the img directory are the image resources for the application. In the js directory are all the JavaScript files, including Mootools and some classes that the application uses. You'll also find two of the packages from Chapter 3 that we'll use: the jscript.dom package and the jscript.page package.

Lastly, in the root directory are four HTML documents. chat.htm is the main portion of the application. index.htm is the page you first access (as a customer) to enter a chat session. index_support.htm is the entry point for support personnel. goodbye.htm is a simple page seen when you exit the chat.

OK, I did some foreshadowing there, so let's dive right in and get to all the details. Although I've typically started with the style and HTML files in other projects, this time, I'm going to jump right in to the JavaScript. It will help you understand some of the hints I dropped previously about specifying which version is in use.

Writing SupportChat.js

The SupportChat class, contained in the SupportChat.js file, is the main client-side class that represents the bulk of the application. As shown in the UML diagram in Figure 12-6, it contains four fields, one to determine the version and three that are transient in nature (contain values used during execution of the application):

- serverType: The value that determines whether we're running the ASP or the JSP version. The value is literally either asp or jsp. This is used to create a reference to the appropriate page in the server directory. So, if you want to drop the application into IIS and run it, you need to change the value here to asp. To run it under Tomcat or another servlet container, make sure the value is jsp. That's all there is to changing between the two versions.
- chattype: Determines whether the chatter is a customer or a support personnel. customer and support are the two possible values.
- chatname: The name the chatter is logged in as.
- lastMessageTime: Holds the last time when the application requested new messages from the server. We'll get to the mechanics of that in a moment, but suffice it to say this is required for it to work.

A number of methods exist in this class, and we'll now look at them one by one.

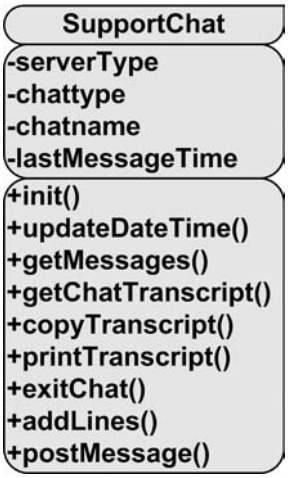


Figure 12-6. *UML diagram for the SupportChat class*

The init() Method: Starting Things Up

The `init()` method is called when the main page, `chat.htm`, loads into the browser:

```
this.init = function() {

  // Get the chatter type
  this.chattype = jscript.page.getParameter("chattype");

  // Get the chatter's name
  this.chatname = jscript.page.getParameter("chatname");

  // Insert greeting.
  $("spnChatname").innerHTML = this.chatname;

  // Set a timer to fire to update the time at the bottom.
  setTimeout(updateDateTime, 0);

  // Set a timer to look for new messages on the server
  // (once every 2 seconds).
  setTimeout(getMessages, 2000);

} // End init().
```

First, this method gets the type of chatter this is by using the `getParameter()` method of the `jscript.page` object, which we built in Chapter 3. The name of the chatter is retrieved the same way. Once that is done, we insert the chatter's name into the `spnChatname <span>`, which gives us the greeting you see at the top.

After that, two timeouts are set. The first is used to update the time at the bottom of the page. Note that the interval is set to zero, meaning this will fire immediately, which is what we want.[2] Otherwise, we would wait a second for the time to initially be displayed at the bottom. The second timeout is used to periodically request new messages from the server.

The updateDateTime() Method: Running the Clock

The first timeout calls the `updateDateTime()` method, so let's look at that.

```
var updateDateTime = function() {

  $("pDateTime").innerHTML = new Date();
  setTimeout(updateDateTime, 1000);

} // End updateDateTime().
```

This is about what you would expect. We insert the string representation of a `Date` object into the `pDateTime` element, and set the timeout again to elapse one second later. It's a piece of cake!

The getMessages() Method: Talking to the Server

The second timeout calls the `getMessages()` method, and there's a little more to see there for sure.

```
var getMessages = function() {

  new Ajax("server/chatServer." + chat.serverType, {
    postBody :
      Object.toQueryString(
        { "func" : "getMessages", "chatname" : chat.chatname,
        "lastMessageTime" : chat.lastMessageTime }
      ),
    onComplete : function(inResponse) {
      // Parse JSON response.
      var messageJSON = eval("(" + inResponse.trim() + ")");
      chat.lastMessageTime = messageJSON.lastMessageTime;
      var lines = new Array();
      // Iterate over messages received.
      for (var i = 0; i < messageJSON.messages.length; i++) {
        var nextMessage = messageJSON.messages[i];
        // Construct a new ChatMessage and add to array.
        var chatMessage = new ChatMessage();
        chatMessage.setTimestamp(nextMessage.timestamp);
        chatMessage.setChatname(nextMessage.chatname);
```

2. You may be wondering why not just call the function directly and then set up the timeout. My answer is that this is simply another way to do it. Either way would work just fine. This seems slightly cleaner to me since it's one line of code rather than two, and the mechanism that fires the function repeatedly is the same one that fires it initially. There's no right or wrong though; it's just one alternative vs. another.

```
            chatMessage.setMessage(nextMessage.message);
            lines.push(chatMessage);
          }
          // Display new message lines.
          addLines(lines);
        }
      }).request();

      // Kick off the timer again.
      setTimeout(getMessages, 2000);

    } // End getMessages().
```

Here, we have our first actual Ajax and the first usage of Mootools. As you can see, the URL is constructed using the value of the `serverType` field, as previously described. We then use the `Object.toQueryString()` method to construct the contents of the POST body. Those contents consist of the parameter `func`, which tells the server which function is being performed; `chatname`, which again is the name of the chatter posting the message as stored in the `chatname` field of the `SupportChat` class; and `lastMessageTime`, which is the time that the last request for messages was made. Any message from the chatter or chat partner in the conversation that is posted subsequent to this `lastMessageTime` will be returned. Since we're checking only for messages every three seconds, it's possible that a number of messages were posted during that time, so we want to get them all in one burst to catch up, so to speak.

We also define in-line a function to execute when the response returns. This callback first parses the JSON being returned, and then iterates over the messages received (which could be none, of course, but this code doesn't break in that situation). For each message present, we construct a `ChatMessage` object, which is basically a Data Transfer Object (DTO) representing a message. You'll see that class shortly, but it's really nothing but a storage container for the attributes of a message, which are its `timestamp` (when it was posted), the name of the chatter who posted it (`chatname`), and the message itself.

After the `ChatMessage` object is created and populated, it is pushed onto the `lines` array, which was created before the iteration over the messages began. After the iteration completes, we pass this array to the `addLines()` method, which is responsible for actually displaying all the messages in the array.

The addLines() Method: Showing Some Messages

Speaking of the `addLines()` method, let's see that right now.

```
var addLines = function(inLines) {

  for (var i = 0; i < inLines.length; i++) {
    var message = inLines[i];
    var styleClass = "cssChatterText";
    if (message.getChatname() != chat.chatname) {
      styleClass = "cssSupportText";
    }
```

```
    htmlOut = "<div class=\"" + styleClass + "\">" +
      message.getChatname() + " : " +
      message.getMessage() +
      "</div>";
    $("divChat").innerHTML = $("divChat").innerHTML + htmlOut;
  }

} // End addLines().
```

This is a pretty straightforward piece of code. It begins to iterate over the array of `ChatMessage` objects passed in, as part of the `inLines` array. For each element in the array, it first determines if the message was posted by the chatter or the chat partner. The `styleClass` variable stores the CSS class name that applies in either case, so we can have our messages in one color and our chat partner's messages in another color. Then, for each message, a `<div>` is constructed, and the name of the chatter and the message are inserted as its contents. Finally, the `<div>` is appended to the existing markup in the `divChat <div>`, and the message is then seen on the screen.

The postMessage() Method: Say It to the World!

You've now seen how messages are retrieved from the server and displayed. The other half of the equation is posting messages, and that is accomplished through the `postMessage()` method.

```
this.postMessage = function(inLines) {

  new Ajax("server/chatServer." + chat.serverType, {
    postBody :
      Object.toQueryString(
        { "func" : "postMessage", "chatname" : chat.chatname,
        "messagetext" : $("postMessage").value }
      )
  }).request();
  $("postMessage").value = "";

} // End addPostMessage().
```

Time for a bit more Ajax! Here, we're doing basically the same thing as you saw in `getMessages()`, but this time the `func` parameter has the value `"postMessage"`, which makes sense I think! Here, we are providing a `messagetext` parameter, passing it the value of the `postMessage` text box. At the very end, we clear that text box so that the chatter can begin typing a new one. That's all there is to it.

The astute reader may be wondering how the chatters see their own messages. There's certainly nothing that handles that display here. The answer is that the message will be displayed as part of the next `getMessage()` cycle. Yes, that means there could be up to a three-second delay between the time chatters post the message and the time they see it on their own screen. This is probably OK to do, although it might be better to put it on the screen immediately (hint, hint).

The only things remaining in this class are the functions to deal with the three menu items, and the one to exit, so let's get to them right now.

The getChatTranscript() Method: For Posterity

First up is `getChatTranscript()`. This is a method used internally by the `copyTranscript()` and `printTranscript()` methods, which are called by the corresponding menu items. `getChatTranscript()` is responsible for literally grabbing the text of the chat session and returning it to the caller.

```
var getChatTranscript = function() {

  // Get the text of the chat.
  var chatTranscript = $("divChat").innerHTML;

  // Now we need to go through the text and remove the HTML components so
  // we are left with nothing but text.  Then, for each line, we make sure
  // there's no trailing or leading whitespace, and we build up a string
  // containing all the lines, separated by linebreaks.
  var transcriptLines = chatTranscript.split(">");
  chatTranscript = "";
  for (var i = 0; i < transcriptLines.length; i++) {
    if (transcriptLines[i].toLowerCase().indexOf("</div") != -1) {
      transcriptLines[i] = transcriptLines[i].replace("</div", "");
      transcriptLines[i] = transcriptLines[i].replace("</DIV", "");
      chatTranscript += transcriptLines[i].trim() + "\r\n";
    }
  }

  return chatTranscript;

} // End getChatTranscript().
```

It begins by getting either the `innerHTML` of the `divChat` `<div>`. This is all the text you see while chatting. Next, the text is split, using the `String` class's `split()` method, on the greater-than sign or the closing of an HTML tag. This results in an array where each element is a single message from the chat, plus lines consisting of just the opening markup of the `<div>` that wraps each line.

A loop then begins to iterate over this array. For each element, we see if the element contains the string `"</div"`. This is only true of elements representing lines of text from the chat; all other elements are the markup of the opening `<div>` only. Note that we need to compare against a lowercase version of the string. This is because in IE, `</div` is present as `</DIV`, so we wouldn't get a match back from `indexOf()` in that case. Once we find a match, we remove the `"</div"` string by replacing it with nothing. Then we add the element to the `chatTranscript` string variable, being sure to `trim()`, and adding a carriage return/line feed sequence after it. The `trim()` function is added to the `String` class by Mootools, and it simply trims all whitespace from both ends of the string.

The final result of all this is that the variable `chatTranscript` contains a text-only version of the chat transcript with each message on its own line, without any blank lines or whitespace on the ends of any lines. This string is returned, and this method's work is done.

The printTranscript() Method: Hurting the Earth to Keep Your Memories

As noted, the `getChatTranscript()` method is used by the `printTranscript()` method, which looks like this:

```
this.printTranscript = function() {

  // Get the transcript of the chat.
  var chatTranscript = getChatTranscript();

  // Open a new window for it.
  var newWindow = window.open();
  newWindow.document.open();
  newWindow.document.write("<pre>" + chatTranscript + "<pre>");
  newWindow.document.close();
  newWindow.print();

} // End printTranscript().
```

There's not much to it. First comes a call to `getChatTranscript()`. Then we just open a new window, write the string we got from `getChatTranscript()` inside a `<pre>` tag, and call the `print()` method on the window. The browser and operating system take it from there, and that's that.

The copyTranscript Method: Direct to Your Operating System

The last method, `copyTranscript()`, is the one that copies the transcript to the operating system's clipboard. There is definitely more involved to this than you might think at first.

```
this.copyTranscript = function() {

  // Get the transcript of the chat.
  var textToCopy = getChatTranscript();

  // Branch based on browser capabilities...
  if (window.clipboardData) {

    // Internet Explorer is easy!
    window.clipboardData.setData("Text", textToCopy);

    // Let the chatter know we're done.
    alert("Chat transcript has been copied to the clipboard");

  } else if (window.netscape) {

    // Netscape/Firefox is hard!  First, ask it for permission to do this.
    try {
      netscape.security.PrivilegeManager.enablePrivilege('UniversalXPConnect');
    } catch (exception) {
```

```
      alert(exception);
      return;
    }

    // Instantiate a clipboard object.
    var clip =
      Components.classes['@mozilla.org/widget/clipboard;1'].createInstance(
      Components.interfaces.nsIClipboard);
    // Instantiate a transferrable object and set it's "flavor."
    var trans =
      Components.classes['@mozilla.org/widget/transferable;1'].createInstance(
      Components.interfaces.nsITransferable);
    trans.addDataFlavor('text/unicode');
    // Instantiate a string object and set its value.
    var str =
      Components.classes["@mozilla.org/supports-string;1"].createInstance(
      Components.interfaces.nsISupportsString);
    str.data = textToCopy;
    // Set the value of the transferrable using the string.
    trans.setTransferData("text/unicode", str, textToCopy.length * 2);
    // Finally, put the text onto the clipboard.
    clip.setData(trans, null,
      Components.interfaces.nsIClipboard.kGlobalClipboard);

    // Let the chatter know we're done.
    alert("Chat transcript has been copied to the clipboard");

  } else {

    // Unsupported browser.
    alert("Unable to copy chat transcript to clipboard.\n\nOnly Internet " +
      "Explorer and Netscape-based browsers (including Firefox) " +
      "are supported.");

  }

} // End copyTranscript().
```

As you would expect by now, we begin with a call to `getChatTranscript()` to get the text of the chat session. Next, we check for the existence of the `clipboardData` attribute of the `window` object. If it is present, we're running in IE, and it's a piece of cake: a quick call to `window.clipboardData.setData()`, passing it the string returned by `getChatTranscript()`, and we're good to go.

Now, the situation is a lot more interesting if the browser is Netscape-based. If the `window` object has a `netscape` attribute, the first thing we need to do is ask for permission to copy data to the clipboard. That's the call to `netscape.security.PrivilegeManager.enablePrivilege()` you see. This results in a query to the user, as shown in Figure 12-7.

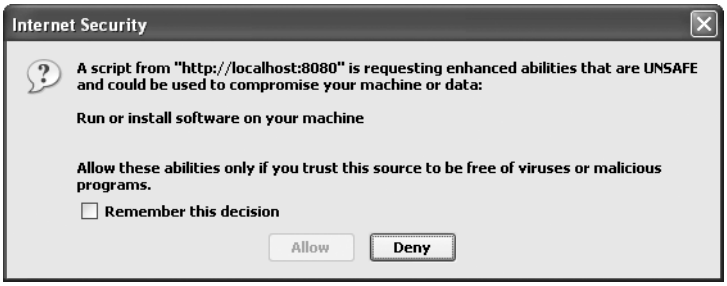


Figure 12-7. *Firefox security privilege query dialog box*

Simply answer Allow, and optionally check the box to remember the setting, and the contents will then be allowed onto the clipboard.

Note that if the Internet Security dialog box does not appear, and instead, you get an alert like the one shown in Figure 12-8, this means you need to go to the advanced configuration options. In the address bar, enter `about:config` and press Enter. You will see a long list of options. The one you are looking for is `signed.applets.codebase_principal_support`. Be sure this is set to `true`. After that setting is adjusted, you should get the dialog box shown in Figure 12-7.

Figure 12-8. *If signed.applets.codebase_principal_support is set to false, you'll see this*

Once we have permission, or have alerted the user that permission is denied, we can move on. The first thing we need to do is create a clipboard object. That's the purpose of the line where the `clip` variable is defined.

Note that the slightly funky syntax you see here, and in the next few object instantiations, is the Netscape style of instantiating native browser objects. It amounts to basically naming the class you want an instance of, then asking the `Components` object to give you an instance. After that, you work with them as you would most any other object.

Next up is creation of a transferrable object, referenced by the `trans` variable. We also have to tell this object that its "flavor" is Unicode, so we are sure to get everything from the chat session properly.

Next, we instantiate a `String` object and store a reference to it in the `str` variable, and set its value to the text of the transcript text we retrieved earlier. Then we pass the `String` along to the transferrable object via a call to `setTransferData()`. Lastly, we copy that object to the clipboard via a call to the `setData()` method of the clipboard object.

We also have an `else` block at the end, covering the case where the browser isn't a supported type. I'm not sure which browsers this might occur in, since I don't have all of them to test with, but IE and Firefox are certainly supported, so the majority of users would have no problem.

The exitChat() Method: Outta Here, I Say!

Last is the `exitChat()` method. This is a simple confirmation pop-up. If the user agrees to exit, `window.location` is set to request from the mock server the exit page, which is just a page saying good-bye to the user.

So, that's the bulk of the client-side code of the application. The next thing to look at is that `ChatMessage` class I mentioned earlier. That won't take long, trust me!

Writing ChatMessage.js

The UML diagram for the simple `ChatMessage` class is shown in Figure 12-9. As previously mentioned, this is just a DTO for storing a single posted message that is part of the transcript of the current chat session.

Figure 12-9. *UML diagram of the ChatMessage class*

Its code is very simple, as you can see in Listing 12-3.

Listing 12-3. *The ChatMessage Class*

```
function ChatMessage() {


  /**
   * The time this message was posted.
   */
  var timestamp = "";


  /**
   * The chatname of the chatter who posted it.
   */
  var chatname = "";
```

```
/**
 * The text of the message
 */
var message = "";


/**
 * Mutator.
 *
 * @param inTime The new field value.
 */
this.setTimestamp = function(inTimestamp) {

  timestamp = inTimestamp;

} // End setTimestamp().


/**
 * Accessor
 *
 * @return The value of the time field.
 */
this.getTimestamp = function() {

  return timestamp;

} // End getTimestamp().


/**
 * Mutator.
 *
 * @param inChatname The new field value.
 */
this.setChatname = function(inChatname) {

  chatname = inChatname;

} // End setChatname().


/**
 * Accessor
 *
 * @return The value of the chatname field.
 */
```

```
  this.getChatname = function() {

    return chatname;

  } // End getChatname().


  /**
   * Mutator.
   *
   * @param inMessage The new field value.
   */
  this.setMessage = function(inMessage) {

    message = inMessage;

  } // End setMessage().


  /**
   * Accessor
   *
   * @return The value of the message field.
   */
  this.getMessage = function() {

    return message;

  } // End getMessage().


  /**
   * Overriden toString() method.
   *
   * @return A meaningful string representation of the object.
   */
  this.toString = function() {

    return "ChatMessage : [ " +
      "timestamp='" + timestamp + "', " +
      "chatname='" + chatname + "', " +
      "message='" + message + "' ]";

  } // End toString().


} // End ChatMessage class.
```

See, I wasn't kidding! Just the three fields—`timestamp`, `chatname`, and `message`—which cover all the data we store about a given message, and the applicable accessor and mutator for each. Also, we have an overridden `toString()`, as you've seen other times throughout this book. This allows us to display a given instance of this class in a meaningful way, which is a practice I highly suggest getting into, because it makes debugging a lot easier.

Writing styles.css

At this point in the book, you've seen a number of style sheets in the other projects, so you probably don't need this one torn apart. I can tell you that there is nothing tricky in it whatsoever, and the comments for each class pretty much give you the complete story of what the styles are.

The one thing I will point out, because it is the first time I've done this in any project so far, is the idea of styling specific HTML elements. The style sheets in other projects have usually taken this form:

```
.cssHeader {
    color : #ff0000;
}
```

You know that this declares a CSS style class named `cssHeader`. You can apply this to any arbitrary element on the page by setting the element's `class` attribute. However, what if we wanted to style all the `<h1>` elements on the page a certain way, and moreover, do so without setting a specific style class on each? You can do that with the following declaration:

```
h1 {
  color : #ff0000;
}
```

Now, any `<h1>` element on the page will be in red. In the style sheet for this project, that is done for a couple of elements, as you can see:

```
/* Style applied to tables. */
table {
  font-size         : 9pt;
  font-family       : arial;
}


/* Style applied to links. */
a:link {
  color             : #6a78a7;
}
```

```
/* Style applied to visited links. */
a:visited {
  color             : #6a78a7;
}


/* Style applied to links that are being hovered over. */
a:link:hover {
  color             : #33305b;
  background-color  : #f0f0f0;
  font-weight       : bold;
}


/* Style applied to h1 elements. */
h1 {
  font-size         : 20pt;
  color             : #000000;
  font-weight       : bold;
}


/* Style applied to h2 elements. */
h2 {
  font-size         : 15pt;
  color             : #8c9cd5;
  margin-top        : 0px;
  margin-left       : 20px;
  margin-bottom     : 0px;
  font-weight       : bold;
}
```

One other thing to mention is how links are styled. For links (and some other elements), you can select what are essentially versions of the element. For instance, if you want to make all links that have not been clicked appear in red, you can set the `color` attribute of the `a:link` selector. If you want to make those links that have been visited show up in blue, you can set the `color` attribute of the `a:visited` selector. Finally, if you want all links to turn green when hovered over, set the `color` attribute of the `a:linkhover` selector. You can see this in the style sheet to give the hover effect on the menu items, which are links.

Other than those points, this style sheet is self-explanatory, so please do have a look at it to be sure you know what it's all about. Now, let's move on.

Writing index.htm and index_support.htm

`index.htm` is the entry point into the application for customers, and `index_support.htm` is where support personnel should go to in order to log in. I'll just show `index.htm` here, in Listing 12-4. `index_support.htm` is basically the same thing, with some changed text.

Listing 12-4. *The index.htm File (Also, More or Less, index_support.htm)*

```
<html>

  <head>

    <title>AJAX-Based One-On-One Support Chat</title>

    <link rel="StyleSheet" href="css/styles.css" type="text/css">

    <script src="js/mootools.js" type="text/javascript"></script>
    <script src="js/jscript.dom.js" type="text/javascript"></script>
    <script src="js/SupportChat.js" type="text/javascript"></script>

    <script>

      /**
       * Called on page load to do some setup.
       */
      function init() {

        var divObj = $('divOuter');
        jscript.dom.layerCenterH(divObj);
        jscript.dom.layerCenterV(divObj);
        $("logonForm").action = "server/chatServer." + chat.serverType;

      } // End init().

    </script>

  </head>

  <body onLoad="init();">

    <div id="divOuter" class="cssDivOuter">
      <img src="img/indexImage.gif" class="cssIndexImg">
      <div class="cssDivInner">
        Hello and welcome to our AJAX-based one-on-one support chat system!
        <br><br>
        To begin, please tell us who you are, and click the Begin button to
        get some help.
        <br><br>
        <form id="logonForm">
          <input type="hidden" name="func" value="logon">
          <input type="hidden" name="chattype" value="customer">
          Your name: <input type="text" name="chatname" size="20">
          <input type="submit" value="Begin" class="cssButton">
        </form>
```

```
      </div>
    </div>

  </body>

</html>
```

This starts out, as most HTML documents do, with a style sheet import and a couple of JavaScript imports. Mootools is imported so the `$()` function can be used. `jscript.dom.js` is imported so we can use the `layerCenterH()` and `layerCenterV()` functions. Lastly, `SupportChat.js` is imported so we have access to the `serverType` field that was discussed earlier.

In the `<head>` is a single JavaScript function, `init()`. This is called when the page is loaded. Its job is twofold. First, it is used to center the contents of the page, which are contained in the `divOuter <div>`, using the `layerCenterH()` and `layerCenterV()` functions. It also sets the action of the form on the page to point to the appropriate `chatServer` version, JSP or ASP.

The markup of the page follows. This consists of a single `<div>` with the ID `divOuter`, which is the `<div>` that is actually centered in `init()`. This contains some text and an HTML form. This form accepts the name the chatter wants to use. It also contains a hidden field named `func`, which has the value `"logon"`. This tells our server component what to do with the parameters submitted. There is also a `chattype` parameter, which in `index.htm` is set to `"customer"`. In `index_support.htm`, it's set to `"support"` to tell the server what type of chatter is logging on.

Writing chat.htm

Now we come to `chat.htm`, which contains the actual page layout. You can see the complete listing of this file in Listing 12-5.

Listing 12-5. *The chat.htm File*

```
<html>

  <head>

    <title>AJAX-Based One-On-One Support Chat</title>

    <link rel="StyleSheet" href="css/styles.css" type="text/css">

    <script src="js/mootools.js" type="text/javascript"></script>
    <script src="js/jscript.page.js" type="text/javascript"></script>
    <script src="js/ChatMessage.js" type="text/javascript"></script>
    <script src="js/SupportChat.js" type="text/javascript"></script>

  </head>


  <body onLoad="chat.init();">
```

```
<table width="100%" cellspacing="0" cellpadding="0"><tr><td>

  <table width="98%" align="center" cellpadding="0" cellspacing="0">

    <tr>
      <td>
        <table width="100%" cellspacing="0" cellpadding="0">
          <tr>
            <td width="46"><img src="img/topLeft.gif"></td>
            <td width="64"><img src="img/spacer.gif"></td>
            <td align="center" class="cssTopText"
              background="img/headerFooter.gif">
              AJAX-Based One-on-One Support Chat
            </td>
            <td width="55"><img src="img/topRight.gif"></td>
          </tr>
        </table>
      </td>
    </tr>

    <tr>
      <td>
        <table width="100%" cellspacing="0" cellpadding="0">
          <tr>
            <td class="cssLeftColumn">
              <div align="center"><img src="img/logo.gif" vspace="10"></div>
            </td>
            <td class="cssRightColumn">
              <table width="100%" border="0" cellspacing="0"
                cellpadding="0">
                <tr>
                  <td class="toptext" align="center">
                    Welcome, <span id="spnChatname"></span>!
                  </td>
                </tr>
              </table>
            </td>
          </tr>
          <tr>
            <td width="220" valign="top" class="cssLeftColumn">
              <div style="width:220px;height:300px;overflow:hidden;"
                <br>
                <h2>Main Menu</h2>
                <br>
                <div class="cssDividers"><img src="img/divider.gif"></div>
                <br>
                      
```

```
    * <a href="javascript:void(0);"
      onClick="chat.copyTranscript();">
      Copy Transcript To Clipboard
    </a><br><br>
          
    * <a href="javascript:void(0);"
      onClick="chat.printTranscript();">
      Print Transcript
    </a><br><br>
          
    * <a href="javascript:void(0);" onClick="chat.exitChat();">
      Exit Chat
    </a><br>
    <br>
    <div class="cssDividers"><img src="img/divider.gif"></div>
  </div>
</td>
 <td class="cssRightColumn">
  <table width="100%" cellspacing="0" cellpadding="0">
    <tr>
      <td class="cssRightColumn">
        <table width="90%" align="center" cellpadding="0"
          cellspacing="0">
          <tr>
            <td>
              <table width="100%" cellpadding="0"
                cellspacing="0">
                <tr>
                  <td class="cssLeftColumn">
                    <div><img src="img/titlebar.gif"></div>
                  </td>
                  <td><img src="img/spacer.gif"></td>
                  <td width="100%" class="cssHeaderFooter"><img
                    src="img/heading.gif"></td>
                  <td class="cssRightColumn"><img
                    src="img/topRight.gif"></td>
                </tr>
              </table>
            </td>
          </tr>
          <tr>
            <td class="cssRightColumn">
              <table width="100%" cellspacing="0"
                cellpadding="0">
                <tr>
                  <td class="cssLeftColumn"> </td>
                  <td>
```

```
                                <div class="cssChatDiv" id="divChat"></div>
                                <span class="cssChatterEntryDiv">
                                  <table border="0" cellpadding="0"
                                    cellspacing="0" width="100%"><tr>
                                    <td valign="middle" class="cssEntry"
                                      align="center">
                                      <input type="text" size="62"
                                        id="postMessage">
                                      <input type="button" value="Say It"
                                        class="cssButton"
                                        onClick="chat.postMessage();">
                                    </td>
                                  </tr></table>
                                </span>
                              </td>
                            </tr>
                          </table>
                        </td>
                      </tr>
                      <tr>
                        <td>
                          <table width="100%" cellspacing="0"
                            cellpadding="0">
                            <tr>
                              <td width="55">
                                <img src="img/bottomLeft.gif">
                              </td>
                              <td class="cssBoxBottom"> </td>
                              <td width="55">
                                <img src="img/bottomRight.gif">
                              </td>
                            </tr>
                          </table>
                        </td>
                      </tr>
                    </table>
                  </td>
                </tr>
              </table>
              <br>
            </td>
          </tr>
        </table>
      </td>
    </tr>
```

```
        <tr>
          <td>
            <table width="100%" cellspacing="0" cellpadding="0">
              <tr>
                <td width="55"><img src="img/bottomLeft.gif"></td>
                <td class="cssBoxBottom">
                  <table width="100%" cellspacing="0" cellpadding="0">
                    <tr>
                      <td width="23"><img src="img/closerLeft.gif"></td>
                      <td valign="middle" class="cssHeaderFooter">
                        <div align="center">
                          <div class="cssDividers" id="pDateTime"></div>
                        </div>
                      </td>
                      <td><img src="img/closerRight.gif"></td>
                    </tr>
                  </table>
                </td>
                <td width="55"><img src="img/bottomRight.gif"></td>
              </tr>
            </table>
          </td>
        </tr>
        <tr>
          <td> </td>
        </tr>
      </table>

    </td></tr></table>

  </body>

</html>
```

I will largely leave this as an exercise for you to review, because it's nothing but typical HTML. After the style sheet and JavaScript imports, you can see that we call `chat.init()` on page load. The variable `chat` is defined in `SupportChat.js`, and it is the one and only instance of the `SupportChat` class. After that call, it really is just straight markup.

The menu items use a handy trick. The `href` of the link is the JavaScript statement `javascript:void(0);`, which results in the link not performing its usual hyperlink function when clicked. Instead, we handle the `onClick` event and call the appropriate method in `ChatSupport`.

About two-thirds of the way through, directly below the `postMessage` text box, is a button that calls `chat.postMessage()` when clicked. This is how messages from the chatter are posted, of course.

Please spend just a few minutes reviewing the code in Listing 12-5, if for no other reason than to convince yourself there's not much going on there. But this is actually a good thing. The presentation is almost entirely separate from the functionality behind it, which is exactly what

we want. So while it may seem like a bit of a cop-out to not go over this in detail, we're focusing on JavaScript and Ajax in particular, and going over a bunch of pretty simple HTML wouldn't really serve that purpose, would it?

Writing goodbye.htm

One last bit of markup is left, and that's the `goodbye.htm` file. This is literally nothing but a straight HTML page that says good-bye to chatters when they log off. Have a look at the code, but if you linger more than about 30 seconds, please go guzzle some coffee and come back when you are more awake!

Creating the Database

Now we come to something just a tad more interesting, and that's the database structure. As previously described, I went with a simple Access database file for the sake of simplicity. This means that this application will run only on a Windows system. This is because for the JSP version, the JDBC:ODBC driver is used (the ODBC driver should come with any recent version of Windows), and the ASP version uses ADO, which again uses the ODBC driver (although no data source is required for either, as direct access to the MDB file is used).

The structure within the database is also very simple, consisting of a grand total of two tables, without any real linkage between them (there is *conceptually* a linkage, but no foreign key relationships or anything like that). Figure 12-10 is a diagram of the two tables.

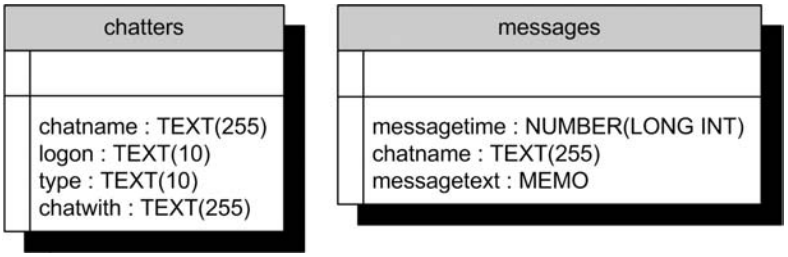


Figure 12-10. *If this database schema scares you, a new career may be in order!*

The `chatters` table stores the list of current chatters. The `chatname` field is the name the chatter gave upon logon. The `logon` field stores the date/time the chatter logged on. The `type` field is either `customer` or `support`, depending on which index HTML file the chatter came through. Finally, `chatwith` is the name of the chatter's chat partner.

The `messages` table holds the messages posted by chatters. The `messagetime` field is the time the message was posted. This is used to determine which messages to return as a response to the periodic Ajax request for messages. The `chatname` field is the name of the chatter who posted the message. This maps to the `chatname` field of the `chatters` table (again, there is no true key relationship here; it is merely a conceptual linkage). Finally, the `messagetext` field is the text of the message itself.

It's a simple database to be sure, but not much more is needed. Note that because of the frequency of change, I saw no point in indexing either of these tables. I don't believe any performance gain would be had by doing so. I also didn't put any constraints on any of the

fields—no required fields and such. These rules are enforced, to the extent they matter frankly, in the server-side code that deals with the database.

One issue to point out is related to using the ASP version of the application. In that case, you will need to ensure that the user account that IIS runs with has full rights to the directory where the MDB file is located, and that the directory has read/write access in IIS itself. If you are toying with the IIS version, you probably knew this already, but it's worth mentioning. Should you try the application and get ADO error number 0x80004005, or any other for that matter but that one specifically, please check those permissions right away, because they are the likely culprit.

Now let's get into the server-side code, which is swaggering to the plate now!

Writing the Server Code

This is going to be a little tricky, because we have both a JSP file and an ASP file to review. However, they are structurally and logically identical, only differing in syntax and some other minor items. In order to get through this in a reasonable way, I will present each file in pieces, and describe what it does conceptually, without getting into all the finer details. My belief is that you will be able to understand one version or the other with little difficulty. I'll point out details where that makes sense, but we'll be going through this with a bird's eye view.

Starting Up

To start, both versions begin with a variable declaration.

Asp

```
filename = "C:\Inetpub\wwwroot\Code\database/chatDB.mdb"
```

Jsp

```
String filename = "K:/tomcat5029/webapps/Code/database/chatDB.mdb";
```

The `filename` variable points to the Access database. You will need to update this variable in order to run the application, as previously described.

Next is some code to open a connection to the database.

Asp

```
  ' variables needed for database work.
  Set conn = Server.CreateObject("ADODB.Connection")

  ' Open connection to database.
  conn.Open "DRIVER={Microsoft Access Driver (*.mdb)}; DBQ=" & filename
  Set rs = Server.CreateObject("ADODB.Recordset")
  rs.CursorLocation = 3
  rs.CursorType = 3
  rs.LockType = 4
```

Jsp

```
// variables needed for database work.
Connection conn = null;
Statement stmt = null;

// Load JDBC driver.
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
String database = "jdbc:odbc:Driver={Microsoft Access Driver " +
  "(*.mdb)};DBQ=" + filename + ";DriverID=22;READONLY=false}";
conn = DriverManager.getConnection( database ,"","");
stmt = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
  ResultSet.CONCUR_UPDATABLE);
```

Both versions are pretty typical code for the respective technologies (ADO for the ASP version and JDBC for the JSP version) to get a connection to a database. In the ASP version, we also get a `RecordSet` object, which we'll reuse throughout the page. In the JSP version, we get a `Statement` object, which again will be reused throughout.

Recall that these pages are the targets of all our Ajax calls, as well as form submissions. Also recall that all of those provide a `func` parameter to tell the server which function is being requested. So, as you might expect, we next get the value of that parameter.

Asp

```
Func = trim(Request("func"))
```

Jsp

```
String func = request.getParameter("func").trim();
```

Next, we need to construct a string that is a timestamp representing the time this request came in. This will be needed for a number of purposes later. The format of this string is *HHMMSSLLL*, where *HH* is the hours (00–23), *MM* is the minutes (00–59), *SS* is the seconds (00–59), and *LLL* is the milliseconds (000–999). We do some string manipulations along the way to ensure that values less than 10 (and less than 100 in the case of milliseconds) are padded with leading zeros so that we always wind up with a nine-character string in the end.

Asp

```
hh = CStr(Hour(Now()))
If Len(hh) = 1 Then hh = "0" & hh End If
mm = CStr(Minute(Now()))
If Len(mm) = 1 Then mm = "0" & mm End If
ss = CStr(Second(Now()))
If Len(ss) = 1 Then ss = "0" & ss End If
ms = "000"
timeStamp = hh + mm + ss + ms
```

Jsp

```
gregoriancalendar calendar = new GregorianCalendar();
String hh = Integer.toString(calendar.get(Calendar.HOUR_OF_DAY));
if (hh.length() == 1) { hh = "0" + hh; }
String mm = Integer.toString(calendar.get(Calendar.MINUTE));
if (mm.length() == 1) { mm = "0" + mm; }
String ss = Integer.toString(calendar.get(Calendar.SECOND));
if (ss.length() == 1) { ss = "0" + ss; }
String ms = Integer.toString(calendar.get(Calendar.MILLISECOND));
if (ms.length() == 1) { ms = "0" + ms; }
if (ms.length() == 2) { ms = "0" + ms; }
String timeStamp = hh + mm + ss + ms;
```

Once that's done, we then do a check of the `func` parameter we got. If it's null, or blank—meaning it wasn't passed in (or possibly was spelled wrong)—we send back an HTML response to indicate an unknown function was requested (which might mean a hacking attempt, so we accuse the chatter of being naughty, just for kicks).

Once we see that `func` was found as a request parameter, we determine which function was requested. This is nothing but a series of `if` checks.

Logging On

The first possible function is logging on, indicated by a `func` value of `"logon"`. This is the longest and most complex of the functions, but it boils down to a few logical steps. Let's first look at the code, and then discuss those steps.

Asp

```
if func = "logon" Then

  ' Processing a logon.
  chatType = Trim(Request("chattype"))

  If chatType = "customer" Then

    ' It's a customer logon.  See if the name is already in use.
    customerChatName = Trim(Request("chatname"))
    rs.Open "select chatname from chatters where " & _
      "chatname='" & customerChatName & "'", conn
    If rs.RecordCount <> 0 Then
      ' Name is already in use, have the chatter select a new one.
      rs.Close
      %>
      <html><head><title>Name already in use</title></head><body>
        I'm sorry but that name is already in use.  Please click
        <a href="../index.htm">HERE</a> and select a new name.
      </body></html>
      <%
```

```
        Else
          ' Name is available, so now we have to see if there are any available
          ' support personnel to chat with.
          rs.Close
          rs.Open ("select chatname from chatters " & _
            "where type='support' and chatwith='none'")
          If rs.RecordCount <> 0 Then
            ' Ok, we got someone.  Now log the chatter into the database and
            ' send them to the chat page.
            supportChatName = rs("chatname")
            rs.Close
            conn.Execute "insert into chatters (chatname, logon, type, " & _
              "chatwith) values (" & _
              "'" & customerChatName & "', " & _
              "'" & timeStamp & "', " & _
              "'customer', '" & supportChatName & "')"
            ' We also need to mark the support person as chatting with this
            ' chatter.
            conn.Execute "update chatters set chatwith='" & _
              customerChatName & "' where chatname='" & supportChatName & "'"
            // Lastly, add a message to the messages table so both chatters see
            // who they are chatting with.
            conn.Execute "insert into messages (messagetime, chatname, " & _
              "messagetext) values ('" & timeStamp & "', '" & _
              supportChatName & "', 'Hello, " & customerChatName & "!  " & _
              supportChatName & " is here to help you!')"
            %>
            <html><head><title>Starting chat</title><script>
              function startChat() {
                window.location =
                  "../chat.htm?func=startChat&" +
                  "chatname=<%=customerChatName%>&chattype=customer&" +
                  "chatwith=<%=supportChatName%>"
              }
              </script></head>
              <body onLoad="startChat();">Starting chat...</body>
            </html>
            <%
          Else
            ' No support personnel available.  Give the chatter the bad news.
            rs.Close
            %>
            <html><head>
            <title>No support personnel available</title>
            </head><body>
              There are currently no support personnel available.  Please click
              <a href="<%=request.ServerVariables("URL")%>?func= ➥
logon&chattype=customer&chatname=<%=customerChatName%>">HERE</a>
```

```
          to check for someone again.
        </body></html>
        <%
      End If
    End If

  Else

    ' It's a support personnel logon.  See if the name is already in use.
    supportChatName = Trim(Request("chatname"))
    rs.Open "select chatname from chatters where " & _
      "chatname='" & supportChatName & "'", conn
    If rs.RecordCount <> 0 Then
      ' Name is already in use, have the chatter select a new one.
      rs.Close
      %>
      <html><head><title>Name already in use</title></head><body>
        I'm sorry but that name is already in use.  Please click
        <a href="../index.htm">HERE</a> and select a new name.
      </body></html>
      <%
    Else
      ' Name is available, so now log the chatter in.
      rs.Close
      conn.Execute "insert into chatters (chatname, logon, type, " & _
        "chatwith) values (" & _
        "'" & supportChatName & "', " & _
        "'" & timeStamp & "', " & _
        "'support', 'none')"
      %>
      <html>
        <head>
          <title>Starting chat</title>
          <script>
            function startChat() {
              window.location =
                "../chat.htm?func=startChat&" +
                "chatname=<%=supportChatName%>&chattype=support";
            }
          </script>
        </head>
        <body onLoad="startChat();">Starting chat...</body>
      </html>
      <%
    End If
  End If

End If ' End "logon" function handling.
```

Jsp

```
if (func.equalsIgnoreCase("logon")) {

   // Processing a logon.
   String chatType = request.getParameter("chattype");

   if (chatType.equalsIgnoreCase("customer")) {

     // It's a customer logon.  See if the name is already in use.
     String customerChatName = request.getParameter("chatname");
     ResultSet rs = stmt.executeQuery(
       "select chatname from chatters where " +
       "chatname='" + customerChatName + "'");
     if (rs.first()) {
       // Name is already in use, have the chatter select a new one.
       rs.close();
       %>
       <html><head><title>Name already in use</title></head><body>
         I'm sorry but that name is already in use.  Please click
         <a href="../index.htm">HERE</a> and select a new name.
       </body></html>
       <%
     } else {
       // Name is available, so now we have to see if there are any available
       // support personnel to chat with.
       rs.close();
       rs = stmt.executeQuery("select chatname from chatters " +
         "where type='support' and chatwith='none'");
       if (rs.first()) {
         // Ok, we got someone.  Now log the chatter into the database and
         // send them to the chat page.
         String supportChatName = rs.getString(1);
         rs.close();
         stmt.executeUpdate("insert into chatters (chatname, logon, type, " +
           "chatwith) values (" +
           "'" + customerChatName + "', " +
           "'" + timeStamp + "', " +
           "'customer', '" + supportChatName + "')");
         // We also need to mark the support person as chatting with this
         // chatter.
         stmt.executeUpdate("update chatters set chatwith='" +
           customerChatName + "' where chatname='" + supportChatName + "'");
         // Lastly, add a message to the messages table so both chatters see
         // who they are chatting with.
         stmt.executeUpdate("insert into messages (messagetime, chatname, " +
           "messagetext) values ('" + timeStamp + "', '" + supportChatName +
           "', 'Hello, " + customerChatName + "!  " + supportChatName +
           " is here to help you!')");
```

```
            %>
            <html><head><title>Starting chat</title><script>
              function startChat() {
              window.location =
                "../chat.htm?func=startChat&" +
                "chatname=<%=customerChatName%>&chattype=customer&" +
                "chatwith=<%=supportChatName%>";
              }
              </script></head>
              <body onLoad="startChat();">Starting chat...</body>
            </html>
            <%
          } else {
            // No support personnel available.  Give the chatter the bad news.
            rs.close();
            %>
            <html><head>
            <title>No support personnel available</title>
            </head><body>
              There are currently no support personnel available.  Please click
              <a href="chatServer.jsp?func=logon&chattype=customer&chatname= ➥
<%=customerChatName%>">HERE</a>
              to check for someone again.
            </body></html>
            <%
          }
        }

      } else {

        // It's a support personnel logon.  See if the name is already in use.
        String supportChatName = request.getParameter("chatname");
        ResultSet rs = stmt.executeQuery(
          "select chatname from chatters where " +
          "chatname='" + supportChatName + "'");
        if (rs.first()) {
          // Name is already in use, have the chatter select a new one.
          rs.close();
          %>
          <html><head><title>Name already in use</title></head><body>
            I'm sorry but that name is already in use.  Please click
            <a href="../index.htm">HERE</a> and select a new name.
          </body></html>
          <%
        } else {
          // Name is available, so now log the chatter in.
          rs.close();
```

```
        stmt.executeUpdate("insert into chatters (chatname, logon, type, " +
          "chatwith) values (" +
          "'" + supportChatName + "', " +
          "'" + timeStamp + "', " +
          "'support', 'none')");
        %>
        <html>
          <head>
            <title>Starting chat</title>
            <script>
              function startChat() {
                window.location =
                  "../chat.htm?func=startChat&" +
                  "chatname=<%=supportChatName%>&chattype=support";
              }
            </script>
          </head>
          <body onLoad="startChat();">Starting chat...</body>
        </html>
        <%
      }
    }

} // End "logon" function handling.
```

The function begins by getting the value of the `chattype` parameter. It then branches on the value of that parameter. The first branch is if it's a customer logging on. In that case, we begin by getting the `chatname` parameter. We then do a query to determine if that name is already in use. If it is, we return a page that indicates this and provides a link back to the logon page, so the chatter can try a different name.

If the name is not already in use, we then do a query to see if there are any support personnel available. This is indicated by finding any chatter in the `chatters` table that has a `type` of `support`, and also that has a value in the `chatwith` field of `none`. If we find someone, we then update the `chatters` table. First, we insert the new chatter into the table. Next, we update the record for the available support person to indicate he is chatting with this new chatter. Lastly, we add a record to the `messages` table that is a quick greeting for both chatters.

After that, we render a response that is a simple HTML page, which upon loading will redirect to `chat.htm`, passing along the necessary information as request parameters.

If no support personnel were available, we return markup indicating this to the chatter, and provide a link the chatter can click to check if anyone is yet available. The chatter can continue to click this link as much as she wants until a support person becomes available, at which point she will be logged on (yes, this isn't the most efficient scheme, and that's why it's the target of one of the suggested exercises at the end of this chapter!).

We next encounter an `else` branch, which is where we deal with logons by support personnel. We again perform a check to see if the name the support chatter provided is available, and do the same things with either outcome (name is available or name is not available) as we did for the customer chatter. If the name is available, we have only a single update to do,

and that's inserting this chatter into the `chatters` table. And that's the end of this particular function!

Getting Posted Messages

The next function in the server code is to handle the periodic Ajax request that gets messages posted since the last check.

Asp

```
if func = "getMessages" Then

 chatname = Trim(Request("chatname"))
 lastMessageTime = Trim(Request("lastMessageTime"))
  ' First, find out who this chatter is chatting with.
 rs.Open "select chatwith from chatters where " & _
   "chatname='" & chatname & "'", conn
 chatwith = rs("chatwith")
 rs.Close
 ' Now, get all messages posted by this chatter, or by who they were
 ' chatting with, since the time of the last message passed in.
 rs.Open "select messagetime, chatname, messagetext from messages " & _
   "where (chatname='" & chatname & "' or " & "chatname='" & chatwith & _
   "') and messagetime >= " & lastMessageTime, conn
 firstMessage = true
 %>
 { "lastMessageTime" : "<%=timeStamp%>",
   "messages" : [
 <% Do While Not rs.EOF
   If firstMessage = true Then
     firstMessage = false
   Else
     response.write ", "
   End If
 %>
      { "timestamp" : "<%=rs("messagetime")%>",
        "chatname" : "<%=rs("chatname")%>",
        "message" : "<%=rs("messagetext")%>"
      }
 <%
      rs.MoveNext
    Loop
 %>
   ] }
 <%
 rs.close()

End If ' End "getMessage" function handling.
```

Jsp

```
if (func.equalsIgnoreCase("getMessages")) {

  String chatname = request.getParameter("chatname");
  String lastMessageTime = request.getParameter("lastMessageTime");
  // First, find out who this chatter is chatting with.
  ResultSet rs = stmt.executeQuery(
    "select chatwith from chatters where " +
    "chatname='" + chatname + "'");
  rs.first();
  String chatwith = rs.getString(1);
  rs.close();
  // Now, get all messages posted by this chatter, or by who they were
  // chatting with, since the time of the last message passed in.
  rs = stmt.executeQuery(
    "select messagetime, chatname, messagetext from messages where " +
    "(chatname='" + chatname + "' or " + "chatname='" + chatwith +
    "') and messagetime >= " + lastMessageTime);
  boolean firstMessage = true;
  %>
  { "lastMessageTime" : "<%=timeStamp%>",
    "messages" : [
  <% while (rs.next()) {
    if (firstMessage) {
      firstMessage = false;
    } else {
      out.print(", ");
    }
  %>
       { "timestamp" : "<%=rs.getString(1)%>",
         "chatname" : "<%=rs.getString(2)%>",
         "message" : "<%=rs.getString(3)%>"
       }
  <% } %>
    ] }
  <%
  rs.close();

} // End "getMessage" function handling.
```

This function begins by getting two incoming request parameters: `chatname` and `lastMessageTime`. The `chatname` parameter is the name of the chatter requesting messages, and `lastMessageTime` is the timestamp when the last such request was made.

The next thing that needs to be done is to find out who this chatter is chatting with (note that this code doesn't care whether the chatter making this request is a customer or a support person, since it works the same either way). We do this because we need to get messages from both sides of the conversation, but the parameters give us only half the information we need.

So, once we have the name of both chatters, we then do a query to find any messages posted by either chatter subsequent to the value of `lastMessageTime`. This results in a collection of records representing all the messages posted by either user since the last time this check was performed, if any.

Then, assuming at least one record was found, it begins to iterate over that collection. Along the way, it constructs a string of JSON, which contains an array of message data elements, namely the time the message was posted (`timestamp`), who posted it (`chatname`), and the message text itself (`message`). This JSON also includes the new value for `lastMessageTime`, which is the value of that `timestamp` string you saw constructed earlier. This value will be stored on the client for the next request to `getMessages`. Finally, the JSON is written out to the response (that technically happens as it's being formed, but you get the picture) and that's that. You saw earlier in our look at `SupportChat.js` how this JSON is consumed. Now you know how it's constructed!

Posting Messages

The next function to look at is posting messages, and it's surprisingly compact.

Asp

```
if func = "postMessage" Then

  chatname = Trim(Request("chatname"))
  messagetext = Trim(Request("messagetext"))
  messagetext = Replace(messagetext, "'", "''")
  conn.Execute "insert into messages (messagetime, chatname, " & _
    "messagetext) values (" & _
    timeStamp & ", " & _
    "'" & chatname & "', " & _
    "'" & messagetext & "')"

End If ' End "postMessage" function handling.
```

Jsp

```
if (func.equalsIgnoreCase("postMessage")) {

  String chatname = request.getParameter("chatname");
  String messagetext = request.getParameter("messagetext");
  messagetext = messagetext.replace('\'', '`');
  stmt.executeUpdate("insert into messages (messagetime, chatname, " +
    "messagetext) values (" +
    timeStamp + ", " +
    "'" + chatname + "', " +
    "'" + messagetext + "')");

} // End "postMessage" function handling.
```

There isn't really much to getting a message posted. First, we get the value of the incoming request parameters `chatname`, which is who posted the message, and `messagetext`, which is the message itself. Then we need to do a quick scan of the string and replace any occurrences of the single apostrophe character with a double apostrophe. This is done so as to not break the constructed SQL statement that you can see next, which is used to insert the message into the `messages` table. That's literally all there is to it.

Logging Off

Only a single function remains, and that's the function that handles logoff. There is perhaps a little more to this than you might imagine, but not too much!

Asp

```
if func = "exitChat" Then

  chatname = Trim(Request("chatname"))
  ' First, find out who this chatter is chatting with.
  rs.Open "select chatwith from chatters where " & _
    "chatname='" & chatname & "'", conn
  chatwith = rs("chatwith")
  rs.Close
  ' Now, delete all messages the chatter posted, as well as messages
  ' posted by who they were chatting with.  After this query, the
  ' "conversation" is effectively deleted from the database.
  conn.Execute "delete from messages where chatname='" & chatname & _
    "' or chatname='" & chatwith & "'"
  ' Next, delete the chatter from the chatters table.
  conn.Execute "delete from chatters where chatname='" & chatname & "'"
  ' Finally, if we find any records in the chatters table where this
  ' chatter is the value of the chatwith field, update that field of
  ' that record to "none".  This covers when the chatter logging off is a
  ' customer, it makes the support personnel available again.  If it's
  ' a support personnel logging off, it does no harm to the chatter,
  ' although the chatter is effectively "orphaned", i.e., their messages
  ' will not be seen by a support personnel, and they will see messages
  ' from no support personnel.
  conn.Execute "update chatters set chatwith='none' where " & _
    "chatwith='" & chatname & "'"
  ' Finally, say goodbye to the chatter.
  %>
  <html>
    <head>
      <title>Exiting chat</title>
      <script>
        function exitChat() {
          window.location = "../goodbye.htm";
        }
```

```
        </script>
      </head>
      <body onLoad="exitChat();">Exiting chat...</body>
    </html>
    <%

  End If ' End "exitChat" function handling.


End If ' End function handling section.
```

Jsp

```
  if (func.equalsIgnoreCase("exitChat")) {

    String chatname = request.getParameter("chatname");
    // First, find out who this chatter is chatting with.
    ResultSet rs = stmt.executeQuery(
      "select chatwith from chatters where " +
      "chatname='" + chatname + "'");
    rs.first();
    String chatwith = rs.getString(1);
    // Now, delete all messages the chatter posted, as well as messages
    // posted by who they were chatting with.  After this query, the
    // "conversation" is effectively deleted from the database.
    stmt.executeUpdate("delete from messages where chatname='" + chatname +
      "' or chatname='" + chatwith + "'");
    // Next, delete the chatter from the chatters table.
    stmt.executeUpdate("delete from chatters where chatname='" + chatname +
      "'");
    // Finally, if we find any records in the chatters table where this
    // chatter is the value of the chatwith field, update that field of
    // that record to "none".  This covers when the chatter logging off is a
    // customer, it makes the support personnel available again.  If it's
    // a support personnel logging off, it does no harm to the chatter,
    // although the user is effectively "orphaned", i.e., their messages
    // will be seen no support personnel, and they will see messages from no
    // support personnel.
    stmt.executeUpdate("update chatters set chatwith='none' where " +
      "chatwith='" + chatname + "'");
    // Finally, say goodbye to the chatter.
    %>
    <html>
      <head>
        <title>Exiting chat</title>
        <script>
          function exitChat() {
```

```
            window.location = "../goodbye.htm";
          }
        </script>
      </head>
      <body onLoad="exitChat();">Exiting chat...</body>
    </html>
    <%

  } // End "exitChat" function handling.


} // End function handling section.
```

First, we get the name of the chatter who is logging off via the incoming `chatname` request parameter. Then we query the `chatters` table to find out who they are chatting with. Next, we delete any record from the `messages` table posted by this chatter, or with the chatter they are chatting with. This is just a bit of cleanup so that the conversation doesn't linger, taking up space unnecessarily. After that, we delete the chatter from the `chatters` table. Finally, we update the `chatters` table for any record we find where the `chatwith` field is equal to the `chatname` received here. This has the effect of making the support personnel this chatter was chatting with available again. If it's a support person logging off, then the customer is essentially orphaned, which is a shortcoming we'll live with for the time being (feel free to add code to check for this in `getMessage` so that you can tell users they are no longer chatting with someone, and perhaps redirecting them to some other page).

The last step is simply to return a page that will immediately redirect the chatter to the `goodbye.htm` page you saw earlier, and the user is officially logged off at that point.

Cleaning Up

The last task in both server files is cleaning up of the database connection.

Asp

```
Set conn = Nothing
```

Jsp

```
if (stmt != null) {
  stmt.close();
}
if (conn != null) {
  conn.close();
}
```

In any case, the `RecordSet` object would have been closed already, and that's why you don't see mention of that here in either version.

And with that, we've completed our look at this application! I hope it's been a good exposure to Ajax, JSON, and Mootools, and that it has shown how the client and server can interact in the Ajax world. This is frankly one of the applications in this book where I purposely left a

number of items, some you might call shortcomings, specifically to give you some good suggestions to reinforce this new knowledge into your cerebellum. So, let's talk about some of those suggestions.

Suggested Exercises

A few items could easily be added to this application to make it better, and to get you a bit more familiar with Ajax and Mootools in the process:

- How about a PHP version? It should just be a matter of converting the code from the Java version to PHP, which probably isn't too tough a translation if you are familiar with PHP.
- It would be nice if a new chatter could be logged in and await an available support person to join her.
- Add tooltips to any element on the page you think makes sense. Mootools has a tips plug-in, which provides this functionality.
- Use some Mootools effects. For instance, perhaps collapse the page when the chatter exits, or if you implement the suggestion to allow a chatter to log in and wait, have some sort of fade-in when a support person joins.
- As hinted at earlier, put the chatters' messages on the screen immediately when they post a message so there's no delay while waiting for the next `getMessage()` cycle to fire.
- Adding a timestamp to each posted message on the screen would be a useful little improvement.

That's the short list. I'm quite sure you can come up with any number of things on your own, but these should serve as a nice start.

Summary

In this, the final leg (er, chapter) of our journey, you've been introduced to what is probably the most famous buzzword of the last two years in web development: Ajax. You've seen it in action, witnessing how it makes a certain class of applications possible, or at least better. You also observed how the server component in the Ajax equation can work. Additionally, you learned how the very fine Mootools JavaScript library can aid you in your Ajax work, as well as other respects. And in the process, you've created a little chat application that can actually be put to use if you so choose to support your customers in this way.

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V