Practical Dojo Projects

ISBN-13 (pbk): 978-1-4302-1066-5

ISBN-13 (electronic): 978-1-4302-1065-8

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

Lead Editor: Steve Anglin
Technical Reviewer: Herman van Rosmalen
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Senior Project Manager: Sofia Marchant
Copy Editor: Sharon Wilkey
Associate Production Director: Kari Brooks-Copony
Senior Production Editor: Laura Cheu
Compositor: Dina Quan
Proofreader: Liz Welch
Indexer: Carol Burbo and Ron Strauss
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

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

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

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

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

CHAPTER 1

Introduction to Dojo

The evolution of client-side development has been a remarkable journey spread out over a very short period of time. From the early days, which were only ten years or so ago, to the present, we have seen a huge change in the way things are done, the expectations placed upon those doing the work, and the results they generate. No longer is it acceptable for someone lacking a real programming background to churn out code for a web page in his or her spare time. No, there's a certain degree of professionalism now asked of us developers, and it's our job to figure out how to deliver the required results in what is usually a very compressed time frame.

Fortunately, because of this evolution, delivering powerful, good-looking, professional-quality web-based applications is far easier than ever before. You are no longer on your own, required to write every last bit of code, or at least to find snippets here and there and cobble it all together into some sort of coherent whole. No, those days are, for the most part, long gone, and it's thanks to things like Dojo that I can say that.

Before we get to Dojo, though, and all the wonderfulness it brings us, let's look back for a little bit of a history lesson, because only then can you fully appreciate what Dojo brings to the table.

JavaScript: A Study in Evolution

In honor of Charles Darwin, let's now take a brief tour of JavaScript and how it's evolved over time. This is in no way meant to be a detailed history lesson, but I think it'll give you a good overview of the past decade or so of client-side web development involving JavaScript.

Birth of a Language

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 a lot of research and on-the-job learning. 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 *The Little Rascals* episode.[1]

1. 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!

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 was that Java wasn't as accessible to many developers as some (specifically, Sun Microsystems, the creator of Java) had hoped. Netscape needed something simpler and more accessible to the masses of developers it hoped to win over.

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, certain trade-offs. 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.[2] 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 Hypertext Markup Language (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 forums, 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.

---

2. 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!

Reasons for JavaScript's Early Rise

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, load that file in a browser, and it worked! You didn't need to go through a compilation cycle or package and 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.

Tying in with the theme of a low barrier to entry was JavaScript's apparent 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 led to a great many of the early problems.

Something Is Rotten in the State of JavaScript

Unfortunately, JavaScript's infancy wasn't all roses by any stretch, as you can see in Figure 1-1. 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 (at least, not as publicly as for a web site).

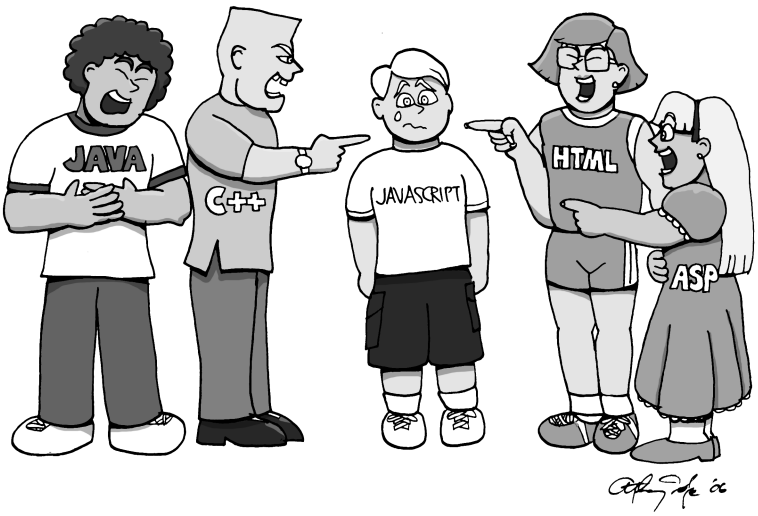

Figure 1-1. *JavaScript: The ugly ducking of the programming world?*

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 hacks. For a long time, JavaScript was very much the ugly duckling of the programming world. It was the Christina Crawford[3] of the programming world, forever being berated by her metaphorical mother, the "real" programmers of the world.

■ **Note** *Script-kiddie* is a term from the hacker underground that refers to a hacker who has no real ability and instead relies on programs and existing scripts to perform his exploits. The term is now used in a more generic sense to describe someone who relies on the ingenuity of others to get things done—for example, coders who can Google real well and copy the code they find rather than writing it themselves. Make no mistake, copying existing code is generally a good thing, but someone who isn't ultimately capable of writing the code if they had to is a script-kiddie.

The Unarguable Problems with JavaScript

Although 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 it 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 usage, and performance.

Browsers for a long time implemented JavaScript in a myriad of different ways. Very simple things such as methods of the `String` object would frequently not work the same way in one browser vs. another. (Anyone who remembers trying to do simple `substring()` calls on a string around the 1996 time frame will know all too well about this!) Today, the differences in JavaScript implementations are generally few and far between and tend to be rather easy to work around. That's certainly not to say you won't occasionally be burned by differences, but typically you'll find that JavaScript itself is probably 99 percent or better compatible between all the major browsers. It's usually differences in the DOM, which there are still plenty of, that can bite you from time to time.

---

3. Christina Crawford was the daughter of Joan Crawford, and her story is told in the classic movie *Mommy Dearest* (`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 Joan to Christina in what was probably the most memorable scene in the movie.

Memory usage with early implementations also left a lot to be desired. JavaScript, being an automatic garbage-collected language, leaves all the memory allocation/deallocation in the hands of the browser and the JavaScript interpreter. It used to be common to find objects gumming up the works long after they should have been destroyed. This is rightly considered an outright bug in the JavaScript engine, but thankfully these types of things rarely occur anymore. To be sure, you can have memory leaks in JavaScript, but they are a result of more-sophisticated techniques being employed; it's rarely the engine itself misbehaving.

Performance was a big problem for a very long time, and even today continues to be an issue. In the early days, it was rare to see an arcade-type game written in JavaScript because the performance of the JavaScript engines was just nowhere near good enough for that. Today, games like that can be found with little problem. Performance problems definitely still exist, and worse still they aren't always consistent. String concatenation is known to be much slower in Microsoft Internet Explorer today than in Mozilla Firefox, and it's probably not even as fast as it could be in Firefox. Still, overall, performance is hugely improved over what it once was.

As Ballmer Said: "Developers! Developers! Developers!..."

All that being said, though, there was one true reason JavaScript wasn't embraced by everyone from the get-go, and that's developers themselves!

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. Although 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, not only because they hurt system performance, but also because they quickly become annoying and distracting.

Here's a quick test: if you are using Microsoft Windows, take a 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 off this stuff, not only because it makes our systems 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. On a Macintosh, the effects that come into play when windows are minimized and maximized are better and not as annoying—at least most people seem think that, and to a certain extent I 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 although there may be degrees of annoyance and obtrusiveness, certain things are still generally annoying, obtrusive, and pointless. Unfortunately, this is what the term *Dynamic HTML (DHTML)* means to many people, and while I wish it weren't so, it isn't at all an undeserved connotation to carry. Some people even go so far as to say DHTML is the Devil's buzzword!

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, based entirely on the perception that it was a playground and not something for serious business.[4] 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 enables you to do something doesn't necessarily mean you should go off and do it.

ARE EFFECTS JUST THE DEANNA TROI OF THE ENTERPRISE BRIDGE CREW? (READ: JUST FOR LOOKS)

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, although one that 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, confusing, and 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, such as 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.

And seriously, how exactly does the ship's shrink manage to always be around on the bridge? Is she necessary personnel for running the Enterprise? I know, I know, she finally passed the bridge crew test and was promoted to commander in the episode "Thine Own Self," but still, she was always floating around the bridge long before then. I suppose she was on the lookout for space madness or something, who knows.

4. I remember a television commercial of a bunch of web developers 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 that it's not even funny!

So, when you hear *DHTML*, don't automatically recoil in fear, as some do. This term 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,[5] not the technology they were using. And at the end of the day, that was the underlying source of the problems people saw back then (and still today, although thankfully to a lesser extent).

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, because 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 toward 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, `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 this technique helps focus users on those changes. It enhances users' 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.

■ **Note** 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 `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 use 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 `www.37signals.com/svn/archives/000558.php`.

Standardization: The Beginning of Sanity

In early 1996, shortly after its creation, JavaScript was submitted to the European Computer Manufacturers Association (ECMA) for standardization. ECMA (`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, but do be clear about it: they are describing the same thing, at least as far as the thought process of most developers goes!

---

5. 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).

The reason this standardization is so important is because it finally gave the browser vendors a common target to shoot for, something they hadn't had before. It was no longer about Microsoft having its vision of what JavaScript should be, and Netscape of course *knowing* what it should be, having created it and all. Now it was about hitting a well-defined target, and only *then* building on top of it (think enhanced feature sets that one browser might provide that another might not). At least if the underlying language implementation was based on a common specification, we developers would no longer have to hit a moving target (well, to be fair, not quite as active a moving target anyway). Without standardization, it's likely that JavaScript would never have become what it is today, wouldn't be used as much as it is today. Either that or we'd have settled on one browser over another just to make our lives as developers easier, and that's no more a desirable outcome than no JavaScript at all.

The Times They Are a Changin': The Experienced Come to Bear

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 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.

Although 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 to enforce the appropriate coding standards. None of this is (typically) present when working with JavaScript. I put *typically* in parentheses because modern development tools now exist to give you all of these features, save for the compile part at least.

Maybe "the bubble" had something to do with it, too. I'm referring to that period when everyone thought they had the surefire 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 (okay, Segway wasn't out then, but work with me here!) recent college grad with an MBA, who promised us that 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, such as 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. It could start to become a first-class citizen, now that developers had the knowledge of how to do it right.

Of course, the last step was yet to come.

What's Old Is New Again: JavaScript in the Present

We've arrived at the present time, meaning the past three to four 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, as Figure 1-2 demonstrates. Perhaps the best evidence of this is that you can now find terms such as *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!

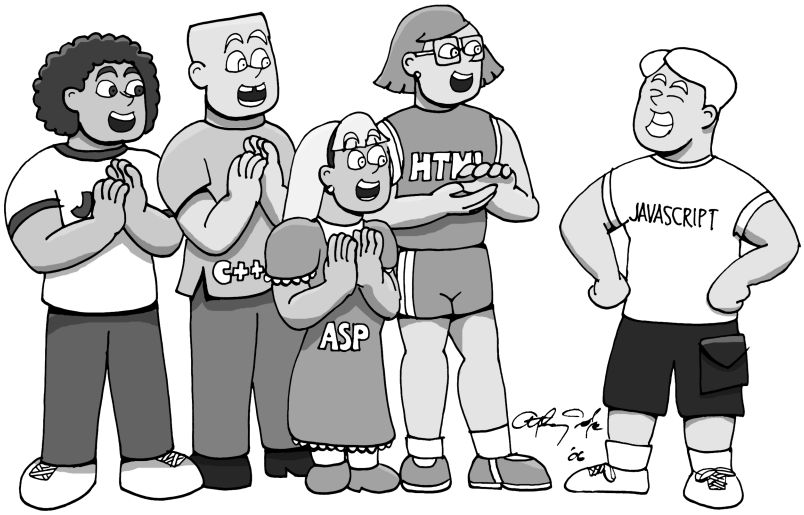

Figure 1-2. *No longer the object of scorn, JavaScript now gets the respect it deserves.*

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.

Browser compatibility isn't so much of an issue anymore, at least when it comes to JavaScript itself (DOM differences are still prevalent, though). Object-oriented design has found its way into JavaScript, leading to much better overall code structure. Graceful degradation is commonplace nowadays, so web sites continue to work even when JavaScript isn't available. Even internationalization and accessibility for those with disabilities is often factored into the design of the code. In fact, this is all a way of saying that JavaScript developers have learned enough that they can be far more responsible with what they design than they

ever could before. JavaScript is no longer something for script-kiddies to mess with but now provides a respectable vocation for a developer to be in!

The White Knight Arrives: JavaScript Libraries

JavaScript libraries have grown leaps and bounds over just the past few years, and this is perhaps one of the biggest reasons JavaScript has gone from pariah to accepted tool in the toolbox of good web developers. 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. If you were fortunate enough to find a true library rather than some stand-alone snippets of code, you couldn't count on its quality or support down the road.

Using libraries is a Good Thing™, as Ogar from Figure 1-3 can attest (and which Tor is finding out the hard way).

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 can find almost more options than you would want! While some libraries out there focus on one area or another (graphical user interface, or GUI; widgets; Ajax; 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.

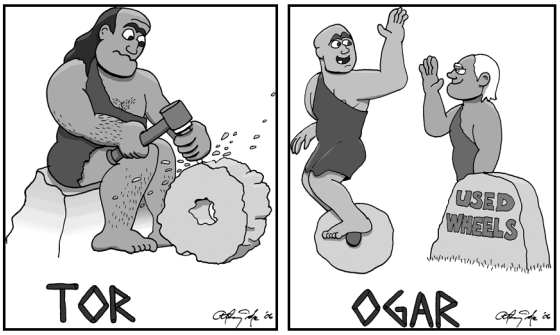

Figure 1-3. *Be like Ogar, not Tor: use those libraries; don't always reinvent the wheel!*

The one thing all these libraries have in common is that their quality is light-years beyond what came before, and they all will make your life considerably easier! Not only are they more solid, but they are better supported than ever before with real organizations, both commercial and otherwise, backing them up.

There's usually no sense in reinventing the wheel, so libraries are indeed a good thing. If you are implementing an application using Ajax techniques, 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 that your UI design requires some more-advanced widgets that the browser doesn't natively provide, these libraries can be invaluable.

There are oodles of libraries out there today, all with their own pluses and minuses. Most of the bigger names out there—jQuery, the Yahoo! UI Library (YUI), Direct Web Remoting

(DWR), Prototype, script.aculo.us, MooTools, and Rico, just to name a few—are all top-notch, and you can hardly go wrong picking one or the other. However, they all have a somewhat narrow focus, for the most part. jQuery is about accessing parts of the DOM as efficiently as possible. YUI is about widgets. DWR is about Ajax. Script.aculo.us is about effects, as is Rico. Prototype and MooTools are both a bit more general-purpose in nature, which is closer in philosophy to the library we're here to discuss, that of course being Dojo.

Without further ado, let's meet the star of the show, the Dojo Toolkit!

The Cream of the Crop: Introducing Dojo!

Dojo, or more precisely, the Dojo Toolkit (but from here on out it'll be just *Dojo* for short) is an open source JavaScript library. No surprise there! Dojo is licensed under either the Academic Free License (AFL) or the BSD license. The AFL is the more liberal of the two and is favored by the Dojo Foundation, which is the parent foundation under which Dojo lives. However, the Free Software Foundation (FSF) has created some ambiguity between the AFL and the two licenses that the FSF is famous for concocting: the GNU General Public License (GPL) and Lesser General Public License (LGPL). So, the fine folks at Dojo give you the choice to use the BSD license or the AFL, the BSD license being compatible with both the GPL and LGPL licenses, as the FSF says.

Dojo seeks to be your "one-stop shop" for all your JavaScript needs. Many libraries out there focus on just one or two areas, which means that you as a developer will usually have to combine two or more libraries to get everything you need done. With Dojo, that's not the case. Dojo covers all the bases, from core JavaScript enhancements to UI widgets, from browser-independent drawing functions to offline storage capabilities.

The kicker to it all, though, is that Dojo does this well! I bet you've seen libraries that purport to do it all, and in fact they do pretty much cover all the bases, yet they fail in one important regard: they're not very good! They tend to have lots of bugs, incomplete functionality, and no support for any of it. Dojo turns that idea upside-down and not only gives you virtually everything you'd need to create rich modern web applications, but enables you to get everything from the same source, and that's definitely a plus.

There's an underlying philosophy to Dojo, and a big part of it is what I just talked about. But before I get any further into that, let's step back and take a quick look at how Dojo came to be in the first place.

Let's Take It from the Top: A Brief History of Dojo

The year was 2004. The air was musky. Oh wait, sorry, I thought I was writing a novel for a moment. Anyway, the year was 2004, early 2004 to be precise. A gentleman by the name of Alex Russell was working for a company named Informatica doing DHTML development. Alex was the creator of a library called netWindows that allowed for creation of a windowed interface (much like the Windows operating system, which was unfortunate as you'll see!). He began looking to hire someone to help him in that work, and so some prominent members of the DHTML programming community on the Web were contacted. This need led to a more generalized discussion among the DHTML community about the future of DHTML, and web development overall.

Ultimately, the job (one of them as it turns out) went to Dylan Schiemann (the other job went to David Schontzler, who worked at Informatica for the summer). As the job progressed,

Alex and Dylan, along with some others, started to have discussions about developing what would be a "standard" library for JavaScript, much like the standard libraries that exist for most other languages, such as C for instance.

A new mailing list (ng-html, which later became the dojo-developer list) was created, initially with folks including Aaron Boodman, Dylan Schiemann, Tom Trenka, Simon Willison, Joyce Park, Mark Anderson, and Leonard Lin on it. Discussions on licensing, intellectual property (IP) rights, coding standards, and such began in earnest. After not too long a time, the really difficult job began.

That job, of course, being what the heck to name the whole thing!

A while earlier, Alex had received a cease-and-desist letter from Microsoft over his use of the word *windows* in his netWindows project. Seriously! I mean, I like to think I'm a pretty creative guy, but even I couldn't make up something that wacky! Which reminds me, I need to go trademark the terms *sock*, *sky*, *person*, and *water*. But I digress. The folks on the mailing list tossed around a bunch of ideas with one of the stated goals (I'd be willing to guess one of the top one or two goals!) to make it something that wouldn't get them sued. Leonard Lin proposed the name *Dojo*, and the rest was history.

The first early code was committed by Alex and Dylan, with the support of Informatica. Soon after, two companies, JotSpot and Renkoo, got behind the project and provided early support. By this point, good momentum was starting to build on the mailing list, with everyone starting to contribute bits and pieces they had lying around, and a JavaScript library was born!

The Dojo Foundation was founded in 2005 to house the copyrights for the Dojo code. The foundation is a 501(c)(6) organization, which is a type of tax-exempt nonprofit organization under the United States Internal Revenue Service code. Alex serves as the president of the foundation, and Dylan is the Secretary/Treasurer.

None of this is terribly important to us code monkeys, I know! But it's interesting to know the history nonetheless, and with those details embedded in our brains, we can move on to more-interesting things, starting with the underlying philosophy behind Dojo.

CODE MONKEY? WHAT DID YOU JUST CALL ME?!?

Some people consider the term *code monkey* to be derogatory, but I never have. To me, a code monkey is someone who programs for a living and enjoys writing code. I suppose by that definition you wouldn't even have to earn a living from programming, so long as you love hacking bits. Either way, it's all about the code!

By the way, just because someone is a code monkey doesn't mean they can't do architecture, and vice versa. If you like all of the facets of building software, if you like being up 'til all hours of the night trying to figure out why your custom-built double-linked list is corrupting elements when you modify them, if you like playing with that new open source library for no other reason than you're curious, if you like the feeling you get from seeing a working application come spewing out the back end of a development cycle (even if it's some otherwise dull business application), then you're a code monkey, plain and simple, and you should never take offense to being called that name.

Of course, some people do mean it in a derogatory way, and you'll know who they are, in which case you should be a good little code monkey and throw feces at them. (Frank Zammetti and Apress cannot be held liable if you actually follow this advice!)

Oh yes, and no discussion of the term code monkey would be complete with referencing the fantastic parody song "Code Monkey" by one Jonathan Coulton. His web site is here: `www.jonathancoulton.com`. Sadly, at the time I wrote this, it appeared to be having problems. Hopefully they are just temporary, but in any case, if you like Weird Al–style funny songs and want to hear a good one about us code monkeys, of which I am proudly one, try that site, and failing that, spend a few minutes with Google trying to find it. (I don't even think the Recording Industry Association of America, or RIAA, the American organization famous for suing grandmothers, children, and even dead people for illegally downloading pirated music, will have a problem with it, but you never can tell, so I'm not suggesting any file-sharing networks you might try!)

The Philosophy of Dojo

Dojo's development has been driven by a core philosophy instilled in the project by those founding members mentioned earlier, and nurtured since by additional contributors who have taken part in making Dojo what it is today. The underlying philosophy is an important aspect of any open source project for us, consumers of the project, to understand. Why is that?

Well, to begin with, understanding the philosophy can give you a feeling of comfort with the project. When you use open source, you are essentially counting on two things: support of those who wrote the code, and the code itself. Being open source means you can get in there and maintain the code as necessary, which is great if the original authors go away. Still, especially if you work in a corporate environment where you may well be betting the future of your company on an open source project, you really want to have some "warm fuzzies" with regard to the prospect of the original authors sticking around to help you when you run into trouble.

More than that, you want to have some understanding of what they're thinking, what they plan to do, and how they plan to do it. An all-volunteer open source project has no real obligation to provide any of that, yet most do because they recognize a certain degree of civil responsibility, so to speak. They recognize that just dumping code into the wild, no matter how good or well-intentioned the motives are, isn't all there is to it. Those involved in most projects feel they have some degree of responsibility to support those who choose to use their code. I'm not talking about coming to their houses and helping them use the code, nor am I talking about reacting to every last issue or feature suggestion everyone makes. It's volunteer work after all. But there's a certain "something" there, for most people, a certain degree of responsibility that they feel obligated to meet.

The Dojo team clearly has that sense of responsibility because they've gone so far as to clearly spell out the philosophy that drives them. Because I've seen it written in a couple of places in a couple of different ways, I'll paraphrase the tenets of the philosophy, try to synthesize it into my own words:

- People should *want* to use your code. I've seen this stated as "minimize the barrier to entry." This covers quite a bit of ground, from how the code is packaged, how it's licensed, how it's designed, how easy it is to get started with, how well it's documented, and so on. The point is, the Dojo team tries to reduce the number of reasons anyone would have for *not* choosing Dojo. You could probably sum it up concisely by simply saying this: produce the best possible code in the best possible way, given the talents of those involved. That's a great motivation and a great guiding principle for any community-driven project.

- Performance is a secondary concern to ease of use. Let me head off a negative thought at the pass here: this in no way, shape, or form implies that Dojo isn't high performance, nor does it imply that the Dojo team doesn't care about performance and strive for it. Quite the opposite, in fact! What it means is that their primary concern is ensuring that the consumers of the library have as easy a time consuming it as possible (and sometimes at the expense of it being more difficult and complex for the library developers themselves). When that goal has been satisfactorily met, only *then* is it time to kick performance up a notch. You may well have heard the saying about never prematurely optimizing code and all the evils that can lead to. This is a variation on that theme.

- Write code that is as simple as possible, but no simpler. Conversely, write code that is only as complex as it needs to be. Don't try to do more than necessary. This is all a way of saying that if the environment the code is running in (read: the browser) can do something, there's no need to do a bunch of heavy lifting to duplicate that functionality in JavaScript. Fill in the gaps between JavaScript and the browser, but don't introduce more pieces than are truly necessary.

- Try to be a "full-stack" library. Dojo tries to be everything to everyone, and tries to do so with a high level of quality, and by and large it succeeds. While other libraries may pick an area or two and focus on them, Dojo tries to do it all so you don't have to worry about the potential difficulties of meshing two or more libraries together to meet all your needs. Much like Java/Java Platform, Enterprise Edition (Java EE) or .NET tries to cover all the bases, so too does Dojo for the JavaScript/web-client world.

- Collaboration is the way. Dojo has, from the outset, been all about talented individuals with similar goals coming together to make something better then they likely could have on their own. This is true of any good community-driven project, and Dojo is certainly no exception.

The Dojo team has also clearly indicated in their own documentation that although these are the goals the project strives for, they may not all be met, or met as fully as the team would like, all the time. The point is, as developers who may look to use Dojo, we know what guides them, we know their underlying thought process, and that's extremely valuable. We can also clearly see that the members of the team, all of whom one presumes buys into these guiding principles, share that sense of civic responsibility and are trying to do right by everyone, to put it in simplest terms. This has to give you a good feeling, a feeling of confidence, in your choice to use Dojo. Indeed, any community-driven/open source project that doesn't seem to share these same goals is likely one you'd be better off avoiding!

The Holy Trinity: The Three Components of Dojo

A long time ago, finding what you needed in Dojo was quite a challenge. There didn't seem to be as much rhyme or reason to the overall structure as you might like. In the end, Dojo users spent a lot of time hunting things down.

That, I'm happy to say, is no longer the case! The Dojo team has now organized the project into three high-level conceptual components, each with a specific focus. Figure 1-4 shows the breakdown.

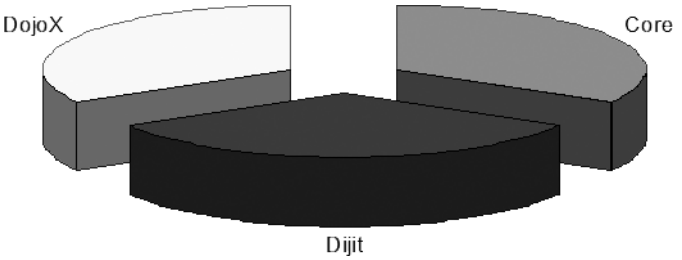

Figure 1-4. *Dojo's three conceptual components. (Note: These are not necessarily scaled to reflect their real sizes in terms of their percentage of the overall Dojo pie.)*

The Holy Trinity, as I like to call it, consists of Core, Dijit, and DojoX. Let's look at each part in turn, at a high level, to see what they're all about.

Core

Like the solid foundation home builders strive to build on, Core provides everything else in Dojo a great foundation to build on. More than that, though, Core offers you, as a JavaScript developer, a whole host of functionality to build your own applications on top of.

As you'll see shortly, using Dojo begins with importing `dojo.js`. What this does for you, in a small amount of space (23KB of code at the time of this writing) is automatically give you a ton of functionality. Beyond that, you'll use the Dojo `include()` mechanism (again, you'll see this in action very shortly), which enables you to "pull in" other parts of Dojo. As far as Core goes, this gives you the ability to bring in more "base" functionality, which is possible because Dojo is designed with a modular approach. Don't worry, this will all make sense very soon. The point for now is simply that Core is, in a sense, split in half: those things you get automatically all the time when using Dojo, and those things that are still part of Core but which you have to explicitly "bring in" to your page.

There's a *ton* of stuff in Core, and you frankly won't even see all of it in this book. There's simply too much to cover, unless the entire book were about Core. Still, I'll do my best to cover and then use in the projects those things you're likely to use most, and as much other stuff as I can. In the meantime, let's run down what's in Core, at a very high level. As you dissect the projects later in the book, you'll see a lot of this in more detail. Table 1-1 and Table 1-2 provide this high-level look. Table 1-1 lists the stuff you can access immediately just by virtue of including `dojo.js` on your page, and Table 1-2 shows the "extra" stuff that's part of Core but that you have to `include()` in your page explicitly.

Table 1-1. *"Automatic" Dojo Core Functionality*

Core Functionality Group	Description
Browser detection	Dojo can tell you whether your application is running in Internet Explorer, Firefox, Opera, Konquerer, and even whether the client is a web browser!
JSON encoding/decoding	Dojo provides functions to turn any object into JavaScript Object Notation (JSON) string, a standard way to represent JavaScript objects, and to turn a JSON string into a JavaScript object.

Continued

Table 1-1. *Continued*

Core Functionality Group	Description
Package loading	The Dojo `include()` mechanism enables modular components to be loaded in a way similar to how packages are handled in languages such as Java and C#.
Powerful Ajax support	No modern JavaScript library that purports to be general-purpose would be worth its salt if it didn't include Ajax functionality, and Dojo's is top-notch. You can quickly and easily make remote calls for content from your client code, all without having to worry about all the potentially sticky issues that can sometimes come into play.
Unified events	Dojo provides a way to hook events to objects of virtually any kind in a consistent manner, allowing for aspect-oriented programming (AOP) techniques in JavaScript. I'm not just talking about the typical UI events you deal with all the time, such as mouse click events, but *events* in a much broader sense. For instance, when one function calls another, that's an event you can hook into.
Animation/effects	Dojo provides all the nifty-keen UI animation effects that are all the rage in the Web 2.0 world. Your UI will be flying, fading, pulsing, sliding, expanding, and shrinking in no time with Dojo!
Asynchronous programming support	I think quoting the Dojo application programming interface (API) documentation here is not only the best bet in terms of accuracy, but frankly it made me chuckle, and I think you will too: "*JavaScript has no threads, and even if it did, threads are hard. Deferreds are a way of abstracting nonblocking events, such as the final response to an* `XMLHttpRequest`. *Deferreds create a promise to return a response at some point in the future and an easy way to register your interest in receiving that response.*" Or, to paraphrase: Dojo provides publish-and-subscribe functionality to JavaScript.
High-performance Cascading Style Sheets (CSS) revision 3 query engine	The ability to look up objects in the DOM of a page is the bread and butter of modern JavaScript programmers, and having a way (or multiple ways even better!) to look up objects quickly and efficiently is of paramount performance. Dojo has you covered!
Language utilities	Dojo supports all kinds of "enhancements" to JavaScript itself, things like collections, array utilities, string manipulation functions, and so on.
CSS style and positioning utilities	The ability to manipulate elements on a page, specifically their style and position attributes, is again a fundamental thing to have to do, and Dojo has all sorts of goodies to help here. This in many ways goes hand in hand with the query engine mentioned previously because usually you'll look up an element, and then manipulate it, and Dojo can make it all a lot easier.

Core Functionality Group	Description
Object-oriented programming (OOP) support	When people first learn that they can do object-oriented development in JavaScript, it's something of an epiphany. Then, especially if they're more familiar with OOP in more-traditional languages such as Java, they quickly begin to realize the limitations. Inheritance, for instance, isn't quite the same as in Java, and there are some things you simply can't do in JavaScript (overloading methods, for instance, isn't possible, at least not in the accepted way). Dojo seeks to overcome most of these limitations by implementing OOP capabilities outside the language in a sense, giving you back all those neat tricks you're used to using elsewhere.
Memory leak protection	Dojo provides some extra event handler hooks that enable you to do some extra cleanup work when your page environment is being destroyed, with the intent, of course, of avoiding memory leaks that can happen in JavaScript sometimes.
Firebug integration	Dojo includes built-in debugging support in the form of integration with the very popular Firebug extension to Firefox. More than that, as you'll see in the next section, it provides a way to get at least some of what Firebug gives you in browsers that don't have Firebug, such as Internet Explorer, without you having to think about it.

Table 1-2. *Functionality in Dojo Core That You Have to Manually Add to the Mix*

Core Functionality Group	Description
Unified data access	Provides functions to read and write data independent of the underlying mechanism
Universal debugging tools	Provides Firebug Lite, which is a much slimmed-down port (really just console logging) of the Firebug plug-in for Firefox to Internet Explorer
Drag and drop	Gives you the ability to implement drag-and-drop functions with little effort
i18n support	Internationalization of text strings on the client
Localizations	More internationalization support
Date formatting	Functions for formatting dates in various ways
Number formatting	Functions for formatting numbers in various ways
String utilities	Additional string functions including `trim()`, `pad()`, and `substitute()`
Advanced Ajax transport layers	Provides ways to execute Ajax requests without using the typical `XMLHttpRequest` object, such as IFrames and JSON with Padding (JSON-P)

Continued

Table 1-2. *Continued*

Core Functionality Group	Description
Cookie handling	Provides functions for reading, writing, and otherwise manipulating cookies
Extended animations	Provides animations and effects built on top of the baseline animations and effects you get automatically
Remote procedure calling	Provides functions for executing remote procedure calls, including JSON-P based services
Back button handling	Provides functions for working with browser history
Baseline CSS styling	Provides functionality for setting uniform font and element sizes

As you can see, there's more variety in Dojo Core than there are alien species represented in the cantina scene of *Star Wars*! It's just the beginning, though, as the Dojo developers have gone on to create two great sequels to Core: Dijit and DojoX. (Unlike George Lucas, who managed only one great sequel with *The Empire Strikes Back*, plus one decent one with *Return of the Jedi*. Of course, the less said about the three prequels the better, although I kind of like *Revenge of the Sith*—the final fight scene was pretty intense—even though it still burns me how quickly and easily Anakin turned; it just didn't make sense.)

Oops, slight tangent there, sorry about that. Let's move on to Dijit, shall we?

Dijit

Perhaps the number one thing that first draws people to Dojo is the widgets, and for good reason: they are overall a very good batch of highly functional and good-looking UI components. What's more, they tend to be quite easy to work with, to get into your application.

What used to be a fairly disorganized collection of widgets is now a lot clearer because they are all grouped together in one place, namely Dijit. (Get it? Dojo widget = Dijit.)

What makes Dojo widgets attractive? A few key points, which most people latch on to pretty quickly, are summarized here:

- Theme support. All of the dijits are *skinnable*, meaning you can rather easily change the look and feel of them just by writing some CSS. Dojo also comes with two built-in themes: Tundra and Soria. Tundra especially is rather nice looking and is designed to blend in well with existing color palettes. Although it may not be a perfect match to your existing application, chances are it will blend decently and not look like an eyesore out of the box.

- Dijits provide a common programming interface. Because all dijits are created from a common set of templates, and because they all extend some baseline classes, they share a basic API. This means that a function that exists for one will often exist for another, at least as far as the basic types of behaviors and features you'd expect any UI widget to have (naturally, each widget will have its own specific features and requirements). This not only means you'll have an easy time picking up widgets, but it also means...

- Creating your own dijit isn't terribly difficult, and it'll automatically, by virtue of having the same basic structure as the rest, work well with Dojo-based applications. Dijit isn't just a collection of ready-to-use UI widgets, although that's the part that you notice first. It's also a complete, powerful, and fully extensible widget architecture that you can build on top of with confidence.
- Dijits are fully accessible by providing keyboard navigation and high-contrast modes built in. Dijits are also built with screen readers in mind, further expanding their accessibility to those with disabilities.
- Dojo exposes a markup-based way of working with dijits, and this leads to graceful degradation by enabling you to have fallback markup present easily.
- Dijits inherently support internationalization and localization, with more than a dozen languages supported right now and more on the way.

I know, I know: that's all well and good; you're excited, but what does Dijit actually *look* like? Well, in the following figures I'll show you examples of Dijit in action, beginning with Figure 1-5, which is the mail reader application you can find on the Dojo web site.

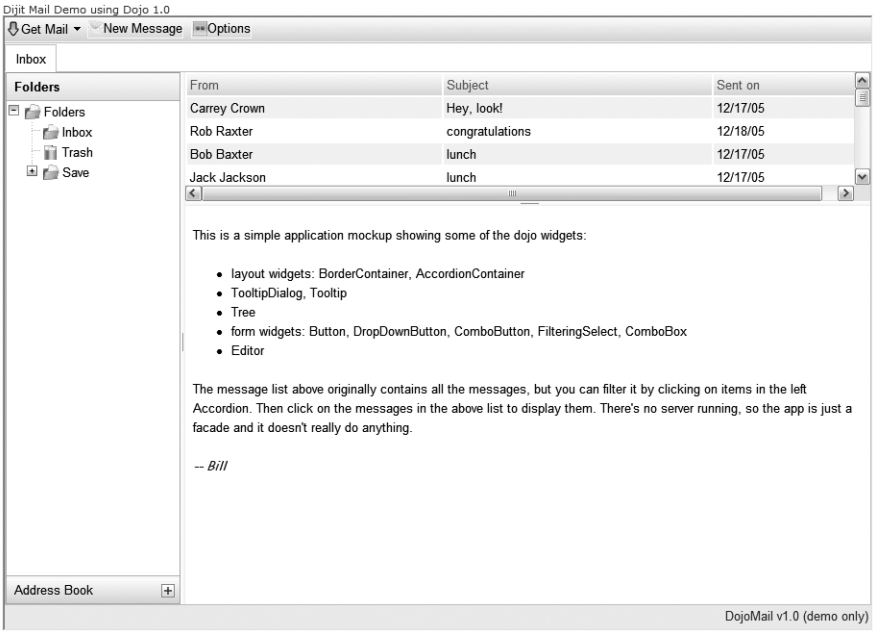

Figure 1-5. *The Dijit mail reader application*

In Figure 1-6, you can see an example of many of the form dijits that are available for building more-complex, powerful, and frankly prettier forms than is possible with basic HTML form elements.

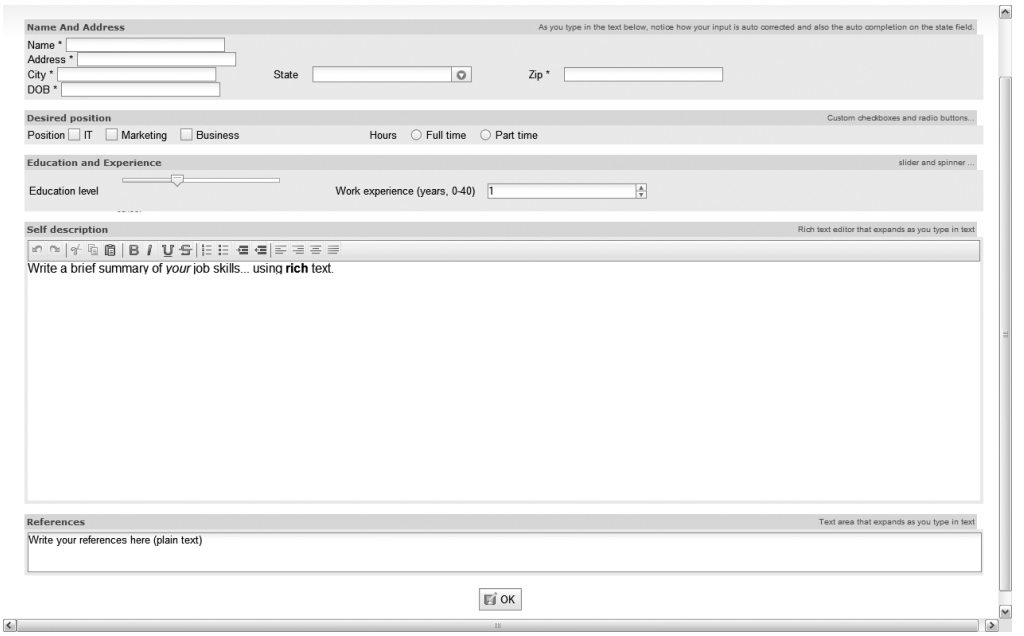

Figure 1-6. *The Dijit form elements in all their glory*

In Figure 1-7, you can see an example of Dijit's support for internationalization, usually, and quite cleverly, abbreviated as *i18n*. You can see that I've selected a new locale from the list, which changes how the form field values are displayed.

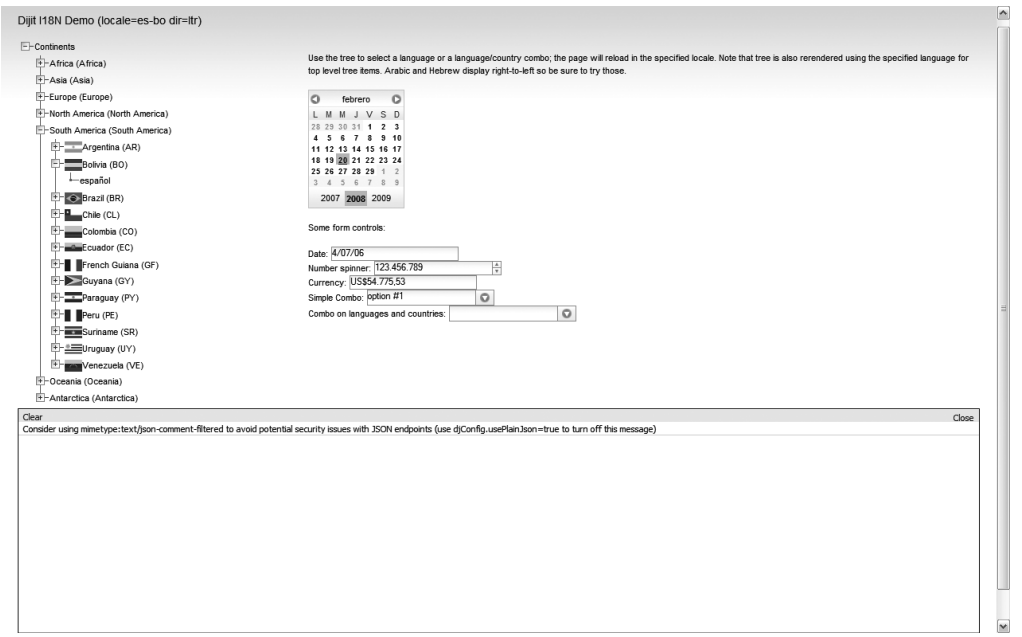

Figure 1-7. *An example of i18n support in Dijit*

Figure 1-8 shows some of the various toolbars Dijit provides. You can also see a tool tip pop-up that is quite a bit more than the usual tool tip: it includes form elements embedded in it.

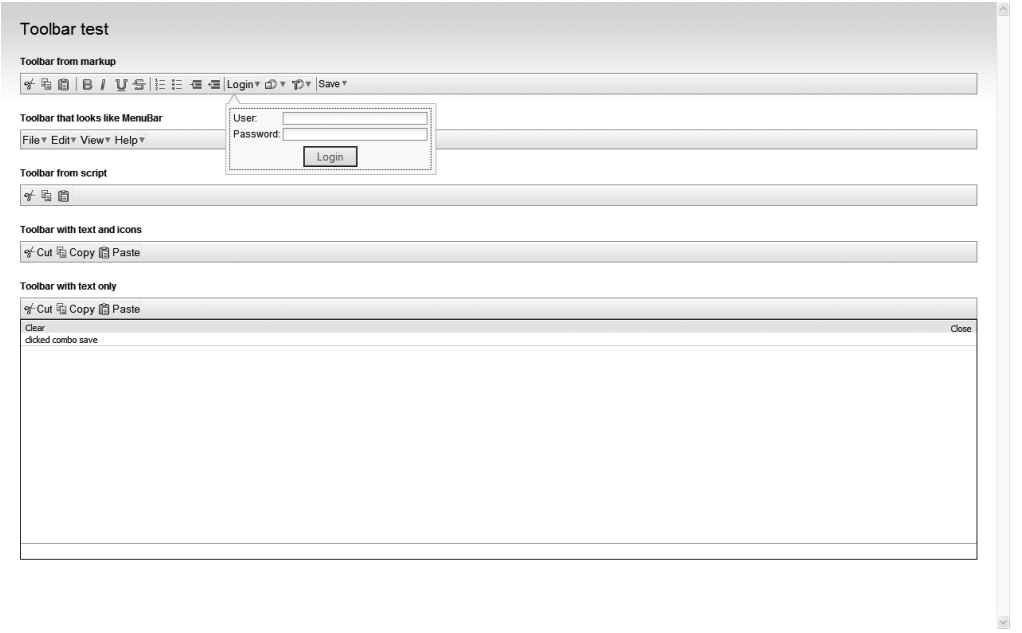

Figure 1-8. *Toolbars in Dijit*

The next few figures are all taken from the Theme Tester application, which comes with the Dojo download bundle. For example, in Figure 1-9, you can see this application with a modal dialog box popped open. This figure is showing the default Tundra theme in action. Unfortunately, this not being a computer screen, you can't get the full effect from the black-and-white pages of a book. Not to worry, though: The examples in this book will use the Tundra scheme quite a bit, so you'll have a chance to see it in all its color and graphical glory as you play with the projects.

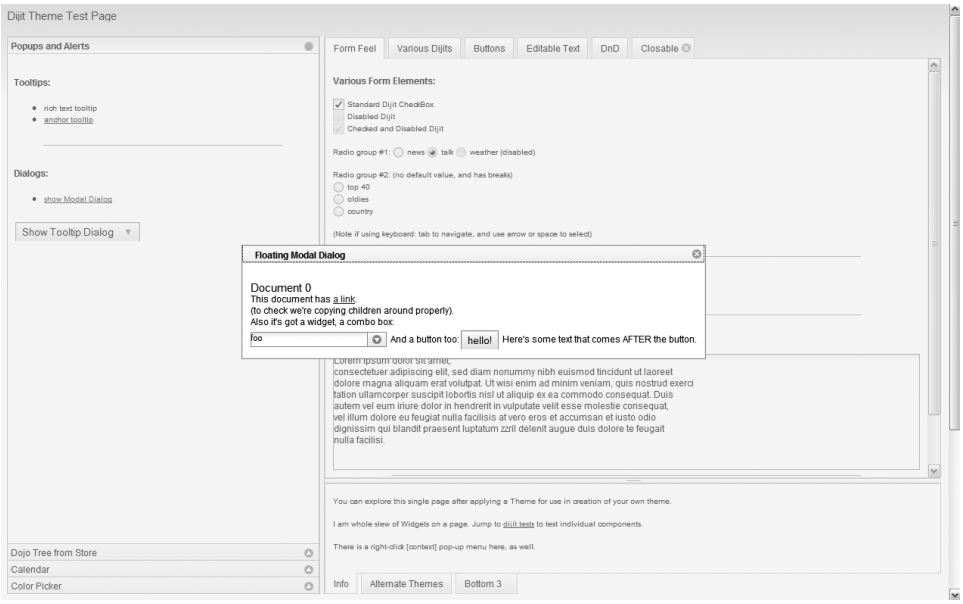

Figure 1-9. *The Theme Tester application, with modal dialog opened up*

Figure 1-10 is another view of this application, but this time you can see a whole slew of dijits! You can also see that I've switched to the Soria theme, and the difference is quite apparent. Note the tabs along the top and bottom. Also note how the right side is an accordion widget, and this time I'm showing a calendar in it.

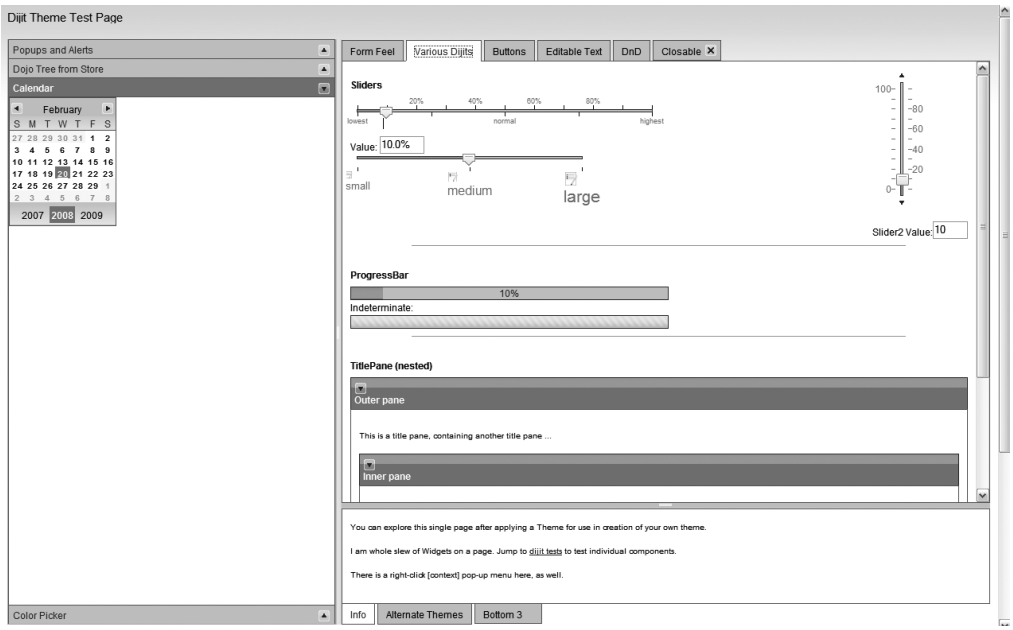

Figure 1-10. *Theme Tester, part 2*

Figure 1-11 provides our final look at the Theme Tester application, this time showing a few more dijits, buttons mostly, plus a color picker on the left. This is again the Soria theme.

Figure 1-11. *Theme Tester, part 3*

As you can plainly see, Dijit has a lot to offer! There are a whole bunch of other widgets that you haven't seen here, and many of those won't be used in this book. I'll try to use as many as possible, but I'll most definitely miss some because there are so many. I encourage you to download Dojo and have a look through it. You'll find numerous example and test HTML files that show all Dijit has to offer.

Now, we have only one stop remaining, and that's DojoX, where we'll walk on the wild side and look at the future of Dojo.

DojoX

The final component of the Dojo project is DojoX (*component* being my way of stating the organizational structure, not a reference to a component in the software design or the GUI component sense of the word). In short, DojoX is a playground for new features in Dojo. It's similar to the Incubator at Apache, where new projects are developed before being moved into the main project itself. (In the case of the Incubator, they may become projects themselves rather than additions to existing projects; DojoX stuff is obviously all going to land in Dojo.)

This makes it sound like the code in DojoX is highly experimental, and indeed in some cases that's true. However, that's not a completely accurate picture. Much of the code found there has been around for quite a while and is rather stable and completely usable; it's just not "final" yet and hasn't graduated into Dojo completely. You should in no way hesitate using DojoX stuff in my estimation because with few exceptions, it's likely to be quite good.

So, what kind of "stuff" is there in DojoX? Let's have a look, shall we? Once again, I need to point out that there's too much to cover every last bit, but I'll hit the big ones that you'll likely find most interesting. You should definitely spend some time exploring outside this book to get the full breadth of what DojoX has to offer.

Charting

Generating charts on a web site is usually an exercise in either (a) some fancy graphics-based client-side code or (b) the more time-tested approach of generating the chart on the server and returning a dynamically-rendered graphic of it. With the charting support in DojoX, you can do what you see in Figure 1-12 entirely on the client dynamically. You won't even need static graphic resources to build this up; it's drawn 100 percent on the fly, so to speak.

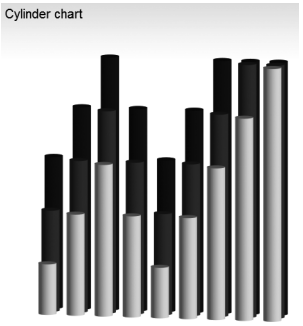

Figure 1-12. *A cylindrical bar chart rendered entirely client-side by DojoX charting support*

And because I'm a big fan of pie, as my ample midsection would likely clue you in on, here's an example in Figure 1-13 of a pie chart, courtesy of DojoX.

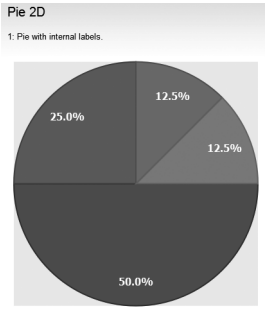

Figure 1-13. *A pie chart (as Homer Simpson would say, "mmmmmm, pie...agh-agh-agh-aga")*

The benefit of not having graphics to download and of not having server processing time involved to generate these charts is a great benefit in many situations. The charting support in DojoX can generate numerous types of charts, including line, pie, bar, area, and so on, with all sorts of options, so it likely has everything you need for any occasion. Better still, the charts are updatable on the fly, which means, mixed in with some Ajax, you can have a bar chart

showing stock values in real time as one example. And of course, because Dojo gives you all the pieces you need to build such a thing, you have just a single library to learn and work with.

Drawing

Typically, showing graphics in a browser means downloading static images that are, at best, generated dynamically on the server (but they're still static in the browser). There are animated GIFs too, which are dynamic in the sense that they are animated and move, but still static in the sense that the file retrieved from the server doesn't actually change at all. You can get into things like Adobe Flash and Microsoft Silverlight if you need to draw graphics from primitive components such as line and circles, but doing that from pure HTML and JavaScript is a hopeless cause.

Or is it?

As it happens, DojoX has just the ticket! The `dojo.gfx` package provides a vector-based drawing API that is cross-browser and 100 percent client-side. If you're familiar with Scalable Vector Graphics (SVG), you have a good idea of what `dojo.gfx` is all about.

Want to see it in action? Your wish is my command; see Figure 1-14.

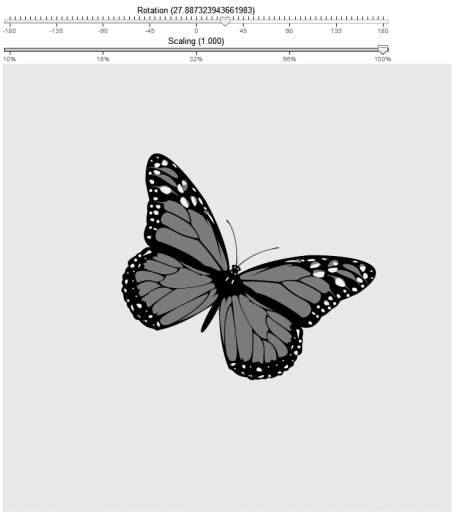

Figure 1-14. *Oooh, how pretty! Better still, it's drawn in real time on the client!*

What's more, there's even a `dojo.gfx3d` package to let you create three-dimensional graphics! Obviously, I can't show you on the printed page, but with the Dojo download comes a boatload of examples of both packages, and a number of them actually perform animation by using this drawing API. (The animation occurs from rapidly redrawing the image.)

This type of capability is something that simply wasn't possible a short time ago, and really, Dojo is beginning to bring it to the masses for the first time.

DojoX Offline

DojoX Offline is, for all intents and purposes, a more robust API wrapped around Google Gears, which is now simply called Gears. (Gears, which you'll see more of in Chapter 5, is a

browser extension that provides capabilities typically needed for running a web app without a network.) In short, DojoX Offline enables you to ensure that the resources (images, HTML files, and so on) that make up your application are available even when the network, or server, is down. This enables you to run your application in an "offline" mode.

This API provides many capabilities that Gears by itself doesn't—encryption, for instance. You may well want to ensure that the resources that are saved client-side for use when the application is offline are encrypted if they are sensitive in nature. DojoX Offline provides this capability on top of Gears, which is a really nice addition.

DojoX Widgets

As if Dijit by itself wasn't enough to fill all your GUI widgets desires, there are even more widgets in DojoX!

One example is the DojoX Grid, shown in Figure 1-15. This is a very good grid implementation that rivals any others out there. What's more, this particular widget integrates with other parts of Dojo, such as `dojo.data` (which is a package for, what else, data access, enabling you to share data between widgets).

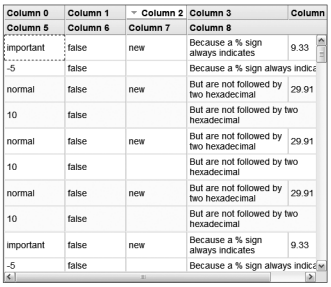

Figure 1-15. *The DojoX Grid widget*

The Grid widget is much more powerful than it seems at first blush because it can essentially host other dijits. For instance, in Figure 1-16, you can see a rich Text Editor dijit in the field I've selected, enabling me to edit the value of that field in place.

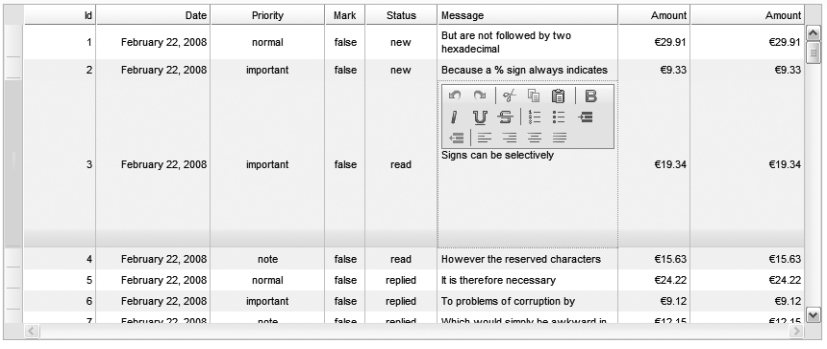

Figure 1-16. *The DojoX Grid hosting other dijits*

The Date fields show a pop-up calendar, and many of the other fields show a drop-down selection dijit when clicked.

But the grid isn't the only widget in DojoX. No, there are plenty more. For instance, the ColorPicker is shown in Figure 1-17.

Figure 1-17. *The DojoX ColorPicker widget*

Then there's the ImageGallery widget, as seen in Figure 1-18, which enables you to show a bunch of images in a graphically pleasing way.

Figure 1-18. *The DojoX ImageGallery widget*

Another good one is the (now) standard Lightbox, which you can see in Figure 1-19. It can contain text, graphics, and quite a bit more.

Figure 1-19. *Dojo's take on the typical lightbox pop-up*

Another one worth mentioning, indeed the best of the bunch for many, is the Fisheye List widget. I won't show that here because I want to keep you in suspense until Chapter 5 when you see it in action for the first time. Rest assured, though, it's a very impressive widget!

The Rest of the Bunch

As I mentioned before, DojoX has quite a bit more. Some of the other things you can find include the following:

- Implementations of many well-known cryptographic algorithms, including Blowfish and MD5
- A set of validation functions for validating things like form input in a variety of common ways such as phone numbers, e-mail addresses, and so on
- A package of XML functionality including, for example, a DOM parser
- Advanced math functions
- A batch of collection classes such as ArrayLists and Dictionaries

You'll see a fair bit of this stuff throughout the projects in this book. Indeed, I'll strive to use as much of it as possible so you can see it in action.

Dojo in Action: What Else? Hello World (Sort Of)

At this point, you've had a decent overview of what Dojo is, how it came to be, its philosophy, and of course what it has to offer. What you've yet to see is an example of it in action. Well, I'm about to remedy that right now!

Listing 1-1 provides your very first (presumably) exposure to Dojo. Take a look at it, and then we'll proceed to rip it apart so you can understand exactly what's going on.

Listing 1-1. *Fisher Price's My First Dojo App!*

```
<html>

  <head>

    <title>My First Dojo App</title>

    <link rel="StyleSheet" type="text/css"
      href="js/dojo/dojo/resources/dojo.css">
    <link rel="StyleSheet" type="text/css"
      href="js/dojo/dijit/themes/tundra/tundra.css">

    <script type="text/javascript">
      var djConfig = {
        baseScriptUri : "js/dojo/",
        parseOnLoad : true,
        extraLocale: ['en-us', 'zh-cn']
      };
    </script>
    <script type="text/javascript" src="js/dojo/dojo/dojo.js"></script>
    <script language="JavaScript" type="text/javascript">
      dojo.require("dojo.parser");
      dojo.require("dijit.form.Button");
      dojo.require("dijit._Calendar");
    </script>

  </head>

  <body class="tundra">

    <div style="position:relative;top:10px;left:10px;width:80%;">
      <button dojoType="dijit.form.Button" id="myButton">
        Press me, NOW!
        <script type="dojo/method" event="onClick">
          alert('You pressed the button');
        </script>
      </button>
      <br><br>
      <table border="0"><tr>
        <td valign="top">
          <input id="calEnglish" dojoType="dijit._Calendar" lang="en-us" />
        </td>
        <td width="25"> </td>
        <td valign="top"
```

```
        <input id="calChinese" dojoType="dijit._Calendar" lang="zh-cn" />
      </td>
    </table>
  </div>

</body>

</html>
```

As you can see, there's not a whole lot of code involved. Before we tear it apart, though, let's have a quick look at the "application," such as it is! I admit it's not going to rival Adobe Photoshop in terms of complexity, but I think it's a good introduction to Dojo. You can be the judge by looking at Figure 1-20.

Figure 1-20. *Look, ma, my very first Dojo application!*

Clicking the button results in a simple `alert()` pop-up. As you can see, the two calendars use different locales, showing how Dojo can simply provide internationalization of its dijits.

So, what's the code all about, you ask? Let's get to that now!

Getting Dojo onto the Page

The way Dojo is brought onto a page, an HTML document here, is simple. First, you access the Dojo web site and download Dojo. Then you decompress the archives you download. This results in the directory structure shown in Figure 1-21.

Figure 1-21. *The directory structure of Dojo, as it is when you downlad it*

Next, you add the following to your HTML document:

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

That's the absolute simplest permutation of using Dojo. This will get you all the stuff in Core that isn't optional. The path itself is as you see it because the `dojo` directory is located in the `js` directory, which is in the root of this mini-application. Under the `dojo` directory is *another* directory named `dojo`, which could have just as easily been named *core*, because that's really what it is, all the Dojo Core code.

There is also another approach that saves you from having to download and host anything on your own web server: the Content Distribution Network. CDN is essentially a simple way of saying that someone has hosted shared content on servers that are designed to handle high levels of traffic from clients all around the world. The Dojo team has partnered with AOL to provide Dojo via CDN, which means you can use Dojo without having to have any of its code present on your own server. This saves you bandwidth and makes setting up your application easier because there are less resources you need to get deployed on your own servers; you just need to point your code at the appropriate URLs and you're off to the races. If you wanted to use this approach, you could use the following line in place of the previous one:

```
<script type="text/javascript"
  src="http://o.aolcdn.com/dojo/1.0.2/dojo/dojo.xd.js "></script>
```

This is indeed handy, and efficient in terms of your own server infrastructure and the traffic it'll have to handle. Even still, I've chosen to not use this approach throughout this book for no other reason than when you write a book, you naturally try to make all the code in it as likely to work as possible. I didn't want to have one more potential point of failure—for example, what if the network is down, or AOL's servers are unavailable, or something like that? Better to have all the Dojo code local so I can be assured (as assured as possible anyway!) that things Just Work as expected.

Importing Style Sheets

Moving on, as you can see from the code in Listing 1-1, there's more to it than just `dojo.js` being imported. Another important piece, which typically matters when you're using Dijit (although it can come into play elsewhere) is to import the appropriate style sheets. This is accomplished thusly (Thusly? Who talks like that?):

```
<link rel="StyleSheet" type="text/css"
  href="js/dojo/dojo/resources/dojo.css">
<link rel="StyleSheet" type="text/css"
  href="js/dojo/dijit/themes/tundra/tundra.css">
```

The first style sheet, `dojo.css`, is a baseline style sheet that Dojo needs to do many things. In this example app, if this style sheet is not present, you'll find that fonts aren't sized quite as they should be. Omitting the style sheet is not a drastic thing here; arguably you could call it optional in this case, but in general you will likely want to import this style sheet and not have to worry about what its absence might do.

The next import is far more critical. Recall earlier that I talked about dijits being fully skinnable and supporting various themes? Well, here's one part of the formula for selecting a theme. Here, I'm using the Tundra theme, which, to my eyes, is somewhat Apple Mac-ish in appearance.

The other part of the formula, which you can see in the `<body>` tag, is to set the class on the body of the document to the theme you wish to use, like so:

```
<body class="tundra">
```

That's all you need to do! From that point on, all dijits you use will use that same theme automatically. Sweeeeeeet, as Eric Cartman[6] would say!

Configuring Dojo

The next step in using Dojo, which is optional although very typical, is configuring Dojo. This is done via the `djConfig` object. This is a JavaScript object that you define *before* the `dojo.js` file is imported. The object contains options you can set, but here we're using only three of them:

```
var djConfig = {
  baseScriptUri : "js/dojo/",
  parseOnLoad : true,
  extraLocale: ['en-us', 'zh-cn']
};
```

The first option, `baseScriptUri`, defines a relative path to the base directory where Dojo is located. All other resources that Dojo loads automatically will use this value when URIs are dynamically written by the Dojo code.

The `parseOnLoad` option tells Dojo to parse the entire document when loaded for Dijit definitions (we'll get to that shortly!), to create those dijits automatically. Alternatively, you can set up an `onLoad` handler to tell Dojo specifically which page elements to parse for dijits, or you can create all your dijits programmatically. This approach, however, is by far the simplest approach.

Finally, `extraLocale` tells Dojo to load additional localizations, other than your own, so that translations can be performed. Here, we're telling Dojo that we want to support English and Chinese. Interestingly, if your system locale is English or Chinese, that language does not need to be specified here. For example, on my PC, which is English-based, I can remove the `en-us` value and things still work as expected. As the name of the option implies, only *extra* locales need to be loaded. But there's no harm in specifying a locale that is the default, so this works just fine, and it also means that if someone happens to run this on a Chinese system, they'll see the English version of the calendar as well.

---

6. I doubt that many readers don't know who Eric Cartman is. He's almost on the level of Captain Kirk in terms of being in the public psyche worldwide. But, on the off chance that you lead a sheltered life: Eric Cartman is the fat kid on the Comedy Central television show *South Park*. He's neurotic, self-centered, spoiled, and extremely funny. If you want to wet yourself laughing (and really, who doesn't want that?), I highly suggest digging up a copy of "Cartman Gets an Anal Probe," which was the pilot episode. I'm sure you'll have no problem finding it on The Internets, and I have every confidence you'll love it.

Importing Other Parts of Dojo

As I mentioned earlier, importing `dojo.js` gets you all the non-optional Dojo Core stuff—and if that's all you need, you're all set. But what if you want to use some of the optional stuff in Core, or you want to use some dijits or something from DojoX? In those situations, you need to get into one of the neatest things about Dojo in my opinion, and that's the include mechanism. This mechanism is on display in this snippet of code:

```
<script language="JavaScript" type="text/javascript">
  dojo.require("dojo.parser");
  dojo.require("dijit.form.Button");
  dojo.require("dijit._Calendar");
</script>
```

With this code, we're telling `dojo` that we want to use three optional components: the parser, the Button dijit, and the Calendar dijit.

The parser component is what creates dijits for us. (I promise, we're getting to that very thing next!) The dijits are, I think, self-explanatory. The nice thing to note here is that you aren't specifying all the resources that might be required, that is, graphics, extra style sheets, or even other JavaScript files. Dojo takes care of those dependencies for you! You just specify the high-level part you want to include, write the appropriate `dojo.require()` statement, and you're off to the races!

Just as a way of teasing something you'll read about in later chapters, you also have the capability of creating a custom build of Dojo, which essentially allows you to get away without even using `dojo.require()`. Basically, everything gets rolled into `dojo.js` and then that's all you need to deal with. But, using `dojo.require()` means that you can keep Dojo modular and then dynamically add pieces as you need them. This *will*, however, increase the number of requests needed to build your page, so there are some performance considerations, which is where custom builds come into play. As I said, though, you'll look at that in more detail in later chapters. For now, `dojo.require()` is the way to go, and the way you'll see most often throughout this book.

Finally: Dijits!

Okay, okay, I've put you off long enough—let's see how those dijits are created! You've already seen the code, of course, but just to focus you in on the appropriate snippet:

```
<div style="position:relative;top:10px;left:10px;width:80%;">
  <button dojoType="dijit.form.Button" id="myButton">
    Press me, NOW!
    <script type="dojo/method" event="onClick">
      alert('You pressed the button');
    </script>
  </button>
  <br><br>
  <table border="0"><tr>
    <td valign="top">
      <input id="calEnglish" dojoType="dijit._Calendar" lang="en-us" />
    </td>
```

```
    <td width="25"> </td>
    <td valign="top"
      <input id="calChinese" dojoType="dijit._Calendar" lang="zh-cn" />
    </td>
  </table>
</div>
```

Whoa, wait a minute, that's just plain old HTML; where's the code? In this case, there is none! Remember that `parseOnLoad` option in `djConfig`? Well, this is what Dojo will be looking for when that option is set to `true`, as it is here.

The parser component, which we imported via `dojo.require()` earlier, scans the page looking for elements with an expando attribute of `dojoType`. This tells Dojo that the element is to be transformed from plain old HTML markup to Dijit, including all the markup and JavaScript that entails. You don't have to write a lick of code yourself to get these dijits on the page. You simply have to tell Dojo to parse the page for dijit definitions and create said definition as done here. Would could be easier? And best of all, from an author's perspective, there's really nothing here for me to explain! Even the `lang` attribute on the `<input>` tags for the calendars is standard HTML, although it's not used too often. Here, it tells Dojo what locality to use to translate the string in the dijit. That's neat, right?

■ **Note** An expando attribute, which might be a term you haven't heard, is when you add an attribute that didn't exist before to an object. This term is usually used in the context of HTML tags (which are ultimately objects in the DOM, remember) when you have an attribute on the tag that isn't strictly allowed given the definition of that tag. Remember that browsers simply ignore things they don't understand, so doing this doesn't break anything (although it *does* make your HTML invalid and it will thus fail strict validation, which some people complain about when it comes to the `dojoType` attribute). Expando attributes are also frequently added to the `document` object, and can in fact be added to virtually any object in the DOM.

You'll be seeing a lot more examples of this markup-based approach to dijit creation as you progress through this book. I think you'll agree that this is a fantastic way to create GUI widgets—although it's not the only way, as you'll also see later.

Getting Help and Information

For all the benefits Dojo has to offer (and as you'll see throughout this book, Dojo has plenty), many times you'll need to get a little help with it. Aside from help, you'll clearly want documentation and examples.

Then again, this book should give you everything you need, so really, you may never need to look elsewhere.

Ahem.

I kid, I kid. In all seriousness, one of the biggest benefits of Dojo is that it has a thriving, helpful community to back it up, and by that I don't just mean the development team, although they certainly count. Tons of fellow users will come to your aid at a moment's notice. Documentation is constantly evolving and getting better for you to refer to. There are message forums and mailing lists and all that good stuff that we've all come to expect and love about open source projects.

You just have to know where to look!

Fortunately, Dojo has a great web site. In fact, have a look for yourself, in Figure 1-22.

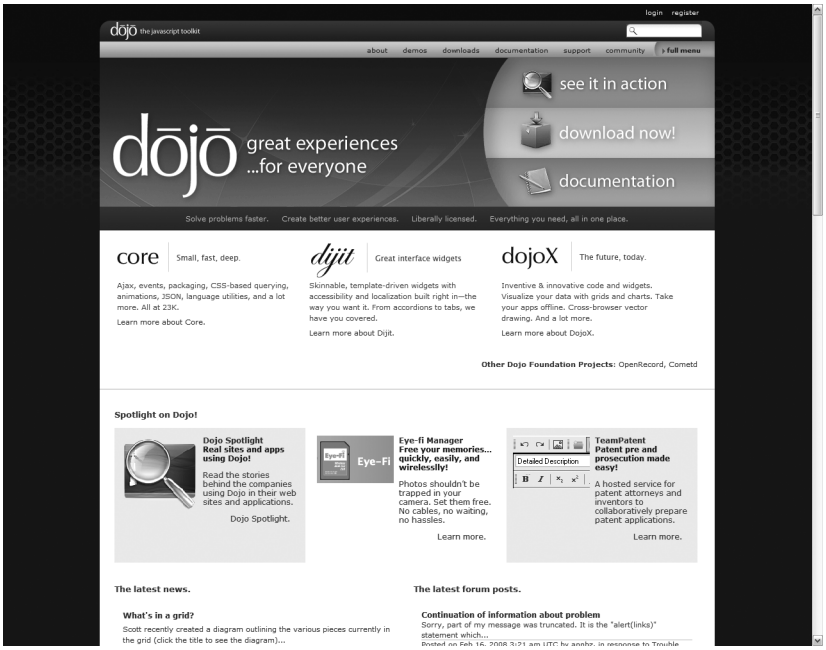

Figure 1-22. *The Dojo web site—'nough said!*

From this site (`www.dojotoolkit.org`) you can reach the message forums, through which most support and discussion takes place. You can also find mailing list archives and subscription information. There is even information on frequently visited Internet Relay Chat (IRC) channels. All of this is available from the Community link on the menu bar up top.

Naturally enough, you can also download pages and issue trackers and even a time line for planned upcoming releases. Not every open source project does this, but it's a really nice thing to do for the community.

One other thing I feel is worth pointing out is the online API documentation, which is generated directly from the Dojo source code and the comments within it. This reference material is one area of the web site you should definitely become acquainted with in a hurry. To access it, visit the Documentation link under the menu bar. Figure 1-23 shows an example of this documentation.

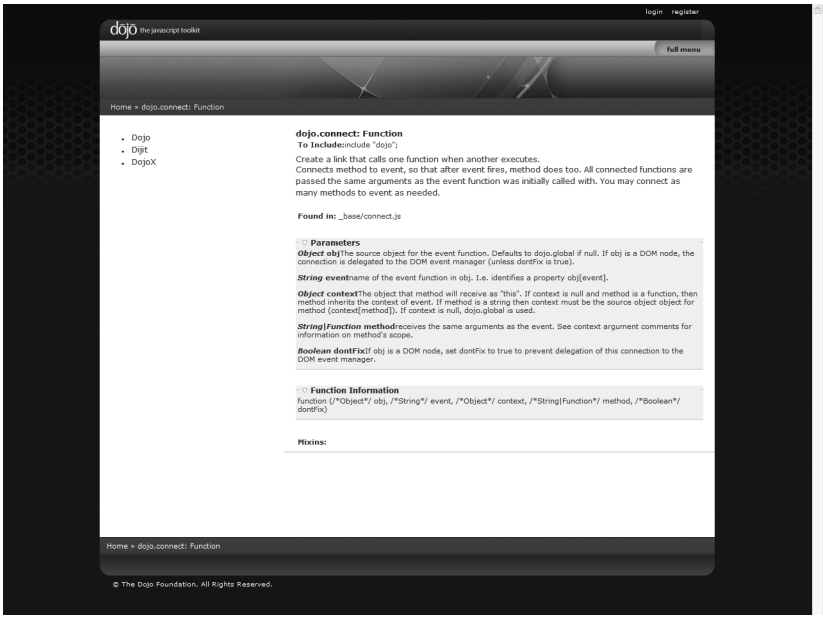

Figure 1-23. *The Dojo online API documentation*

Note that this isn't simple static text: it's all nicely hyperlinked and even uses some cool Dojo effects and widgets, making it a dynamic, interactive bit of reference. This can be very, very helpful when you're trying to figure out what something in the library does (and believe me, I visited it frequently during the course of writing this book!).

THE GOOD...AND THE POSSIBLY NOT SO GOOD

For all the good things about Dojo, and of that there's plenty, one thing that has historically lagged a bit, in my opinion anyway, is documentation and examples. My experience with Dojo started early on, with the very early beta releases. Back then, you had to spend most of your time simply looking at the code to understand how things worked, or how to use this function or that. It was rather difficult, and the fact that Dojo was changing rapidly (which is of course something you accept when using early release code) didn't help matters.

The situation has improved leaps and bounds, especially with the recent 1.0 release (as of the writing of this text). The APIs, outside DojoX anyway, have all stabilized, and documentation has been seriously beefed up. I can also say with complete confidence that the Dojo team is aware that this area needs some work even still and they are diligently putting in effort in this regard.

All of this being said, I feel it's still fair to issue the same caution I've issued in the past about Dojo: if you plan to use it, do yourself a favor and also plan some extra time to find information you need, to research, and to figure things out. The Dojo community is top-notch in terms of extending a helping hand, and that'll likely be your primary resource, not any documentation or examples that you may find on the Web. I've never failed to get an answer to a question I had in short order, and it's always been from polite and helpful people, both Dojo developers and plain old users like me.

Still, if you go to the Dojo web site expecting to find documentation of every last thing, or examples of everything, I believe you will be disappointed. Looking through the source code is often still your best bet. But that's part of the reason I'm writing this book, after all! Hopefully, I will save you a lot of that research time and will answer a lot of the questions you may have, all in one place. You'll certainly have a lot of well-documented example code to look at in these pages; that much I can promise! If I didn't think Dojo was worth all this effort, I wouldn't have written this book, nor would I ever counsel someone to use Dojo.

The fact is, you get out of it a lot more than the effort you put into it, and that's ultimately what you'd want from a good library or toolkit, so don't be put off if you can't immediately find the exact bit of information you need when working with Dojo. Crack open the actual Dojo code, join the community (the forums most especially), ask questions, and I guarantee that you'll get the information you need and Dojo will prove to be an invaluable tool for you. And by the way, I know the Dojo team will thank you very much if you contribute some documentation, so don't be shy! If you've ever wanted to get involved in an open source project, documentation is always an easy inroad into a community.

Summary

In this chapter, we discussed the history and evolution of JavaScript and how it has been applied to client-side development. We looked at the causes of its early, somewhat negative impression and how it came from those humble beginnings to something any good web developer today has to know to some extent. We then took a quick look at Dojo, got a high-level feel for what it's all about, its history, its philosophy, and its structure. We moved on from there to see a basic example of it in action and saw the basics of how to add it to an application. Finally, we saw how to get further help and information about Dojo.

In the next chapter, we'll begin to peel back the layers of the onion and get into detail about what Dojo has to offer, starting with the Core component. Hold on to your hats; you're in for a wild ride from here on out!