Practical JavaScript™, DOM Scripting, and Ajax Projects

Frank W. Zammetti

Apress®

Practical JavaScript™, DOM Scripting, and Ajax Projects

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

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

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

Lead Editor: Matthew Moodie
Technical Reviewer: Herman van Rosmalen
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Jeff Pepper, Paul Sarknas, Dominic Shakeshaft, Jim Sumser, Matt Wade
Project Manager: Tracy Brown Collins
Copy Edit Manager: Nicole Flores
Copy Editor: Marilyn Smith
Assistant Production Director: Kari Brooks-Copony
Production Editor: Laura Esterman
Compositor: Susan Glinert
Proofreaders: Lori Bring and April Eddy
Indexer: Broccoli Information Management
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

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

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

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

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

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

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

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

Contents at a Glance

Contents

PART 2 The Projects

About the Author

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

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

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

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

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

About the Technical Reviewer

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

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

About the Illustrator

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

Acknowledgments

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

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

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

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

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

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

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

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

Introduction

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

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

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

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

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

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

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

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

An Overview of This Book

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

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

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

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

Obtaining This Book's Source Code

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

Obtaining Updates for This Book

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

Ahem . . .

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

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

Contacting the Author

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

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

PART 1

Say Hello to My Little Friend: JavaScript!

Eaten any good books lately?

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

The Internet? Is that thing still around?

Homer Simpson

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

Rich Cook

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

Tom Cargill

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

Bjarne Stroustrup

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

Edward Tufte

CHAPTER 1

A Brief History of JavaScript

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

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

How JavaScript Came to Exist

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

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

---

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

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

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

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

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

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

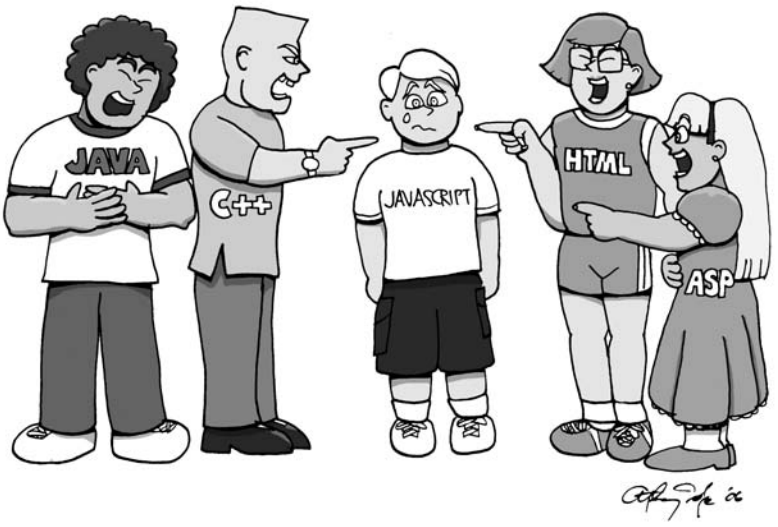
3. As a historical aside, you might be interested to know that version 2.0 of Netscape Navigator introduced not one but two noteworthy features. Aside from JavaScript, frames were also introduced. Of course, one of these has gained popularity, while the other tends to be shunned by the web developer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

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

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

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

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



Poor javascript—other languages can be so cruel!

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

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

The Evolution of JavaScript: Teething Pains

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

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

But It's the Same Code: Browser Incompatibilities

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

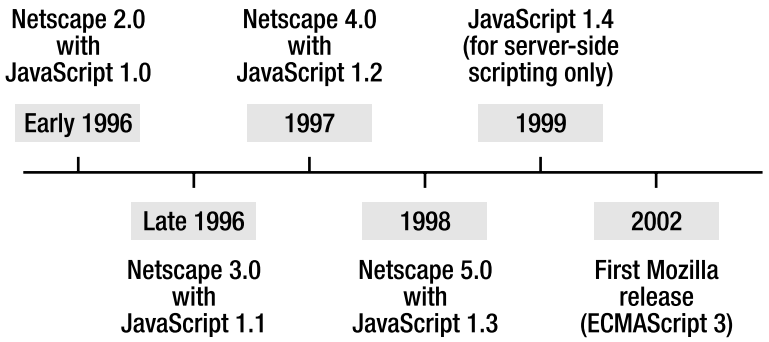


Figure 1-1. *The quick-and-dirty history of Netscape Navigator and JavaScript*

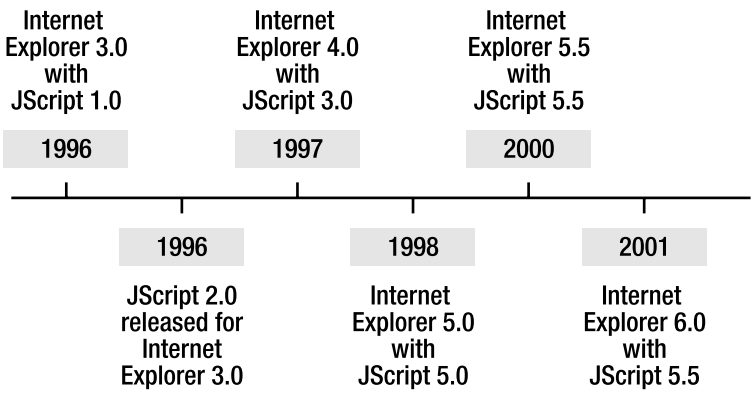


Figure 1-2. *The quick-and-dirty history of Internet Explorer and JScript*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Of Snails and Elephants: JavaScript Performance and Memory Issues

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

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

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

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

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

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

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

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

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

    </script>

  </head>

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

</html>
```

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

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

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

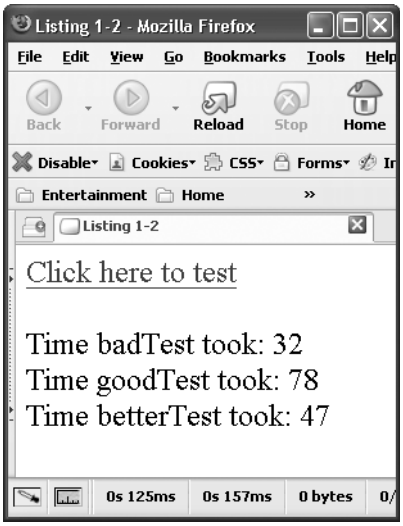


Figure 1-3. *The speed test results in Firefox (1.5.0.6, latest as of this writing)*

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

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

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

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

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

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

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

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

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

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

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

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

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

And speaking of developers and doing something stupid . . .