Build Faster Web Application Interfaces

O'REILLY®

YAHOO! PRESS

Nicholas C. Zakas

High Performance JavaScript

If you're like most developers, you rely heavily on JavaScript to build interactive and quick-responding web applications. The problem is that all of those lines of JavaScript code can slow down your apps. This book reveals techniques and strategies to help you eliminate performance bottlenecks during development. You'll learn how to improve execution time, downloading, interaction with the DOM, page life cycle, and more.

Yahoo! frontend engineer Nicholas C. Zakas and five other JavaScript experts—Ross Harmes, Julien Lecomte, Steven Levithan, Stoyan Stefanov, and Matt Sweeney—demonstrate optimal ways to load code onto a page, and offer programming tips to help your JavaScript run as efficiently and quickly as possible. You'll learn the best practices to build and deploy your files to a production environment, and tools that can help you find problems once your site goes live.

- Identify problem code and use faster alternatives to accomplish the same task
- Improve scripts by learning how JavaScript stores and accesses data
- Implement JavaScript code so that it doesn't slow down interaction with the DOM
- Use optimization techniques to improve runtime performance
- Learn ways to ensure the UI is responsive at all times
- Achieve faster client-server communication
- Use a build system to minify files, and HTTP compression to deliver them to the browser

"High Performance JavaScript *covers all of the performance issues that today's JavaScript developers need to be aware of. It definitely added to my list of performance best practices.*"

—**Steve Souders**

"High Performance JavaScript *is an impressive collection of JavaScript topics, tips, and tricks—written by subject matter experts—all in one place. It is a valuable read for anyone wanting to write high-quality JavaScript.*"

—**Venkat Udayasankar**
Search Performance Guru, Yahoo! Search

Nicholas C. Zakas, principal frontend engineer for the Yahoo! home page and contributor to the Yahoo! User Interface (YUI) library, is a software engineer specializing in web interface design and implementation using JavaScript, HTML, CSS, XML, and XSLT.

O'REILLY®
oreilly.com

An intermediate to advanced understanding of JavaScript is recommended.

US $34.99 CAN $43.99
ISBN: 978-0-596-80279-0

Free online edition
for 45 days with purchase of this book. Details on last page.

High Performance JavaScript

High Performance JavaScript

Nicholas C. Zakas

Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

High Performance JavaScript
by Nicholas C. Zakas

Copyright © 2010 Yahoo!, Inc. All rights reserved.
Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Mary E. Treseler
Production Editor: Adam Zaremba
Copyeditor: Genevieve d'Entremont
Proofreader: Adam Zaremba
Indexer: Fred Brown
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

Printing History:
March 2010: First Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *High Performance JavaScript*, the image of a short-eared owl, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

RepKover™ This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN: 978-0-596-80279-0

[M]

1268245906

This book is dedicated to my family, Mom, Dad, and Greg, whose love and support have kept me going through the years.

Table of Contents

Preface .... **xi**

1. Loading and Execution **1**
Script Positioning 2
Grouping Scripts 4
Nonblocking Scripts 5
Deferred Scripts 5
Dynamic Script Elements 6
XMLHttpRequest Script Injection 9
Recommended Nonblocking Pattern 10
Summary 14

2. Data Access **15**
Managing Scope 16
Scope Chains and Identifier Resolution 16
Identifier Resolution Performance 19
Scope Chain Augmentation 21
Dynamic Scopes 24
Closures, Scope, and Memory 24
Object Members 27
Prototypes 27
Prototype Chains 29
Nested Members 30
Caching Object Member Values 31
Summary 33

3. DOM Scripting **35**
DOM in the Browser World 35
Inherently Slow 36
DOM Access and Modification 36
innerHTML Versus DOM methods 37

Cloning Nodes 41
HTML Collections 42
Walking the DOM 46
Repaints and Reflows 50
When Does a Reflow Happen? 51
Queuing and Flushing Render Tree Changes 51
Minimizing Repaints and Reflows 52
Caching Layout Information 56
Take Elements Out of the Flow for Animations 56
IE and :hover 57
Event Delegation 57
Summary 59

4. Algorithms and Flow Control 61
Loops 61
Types of Loops 61
Loop Performance 63
Function-Based Iteration 67
Conditionals 68
if-else Versus switch 68
Optimizing if-else 70
Lookup Tables 72
Recursion 73
Call Stack Limits 74
Recursion Patterns 75
Iteration 76
Memoization 77
Summary 79

5. Strings and Regular Expressions 81
String Concatenation 81
Plus (+) and Plus-Equals (+=) Operators 82
Array Joining 84
String.prototype.concat 86
Regular Expression Optimization 87
How Regular Expressions Work 88
Understanding Backtracking 89
Runaway Backtracking 91
A Note on Benchmarking 96
More Ways to Improve Regular Expression Efficiency 96
When Not to Use Regular Expressions 99
String Trimming 99
Trimming with Regular Expressions 99

Trimming Without Regular Expressions 102
A Hybrid Solution 103
Summary 104

6. Responsive Interfaces 107
The Browser UI Thread 107
Browser Limits 109
How Long Is Too Long? 110
Yielding with Timers 111
Timer Basics 112
Timer Precision 114
Array Processing with Timers 114
Splitting Up Tasks 116
Timed Code 118
Timers and Performance 119
Web Workers 120
Worker Environment 120
Worker Communication 121
Loading External Files 122
Practical Uses 122
Summary 124

7. Ajax 125
Data Transmission 125
Requesting Data 125
Sending Data 131
Data Formats 134
XML 134
JSON 137
HTML 141
Custom Formatting 142
Data Format Conclusions 144
Ajax Performance Guidelines 145
Cache Data 145
Know the Limitations of Your Ajax Library 148
Summary 149

8. Programming Practices 151
Avoid Double Evaluation 151
Use Object/Array Literals 153
Don’t Repeat Work 154
Lazy Loading 154
Conditional Advance Loading 156

Use the Fast Parts 156
Bitwise Operators 156
Native Methods 159
Summary 161

9. Building and Deploying High-Performance JavaScript Applications 163
Apache Ant 163
Combining JavaScript Files 165
Preprocessing JavaScript Files 166
JavaScript Minification 168
Buildtime Versus Runtime Build Processes 170
JavaScript Compression 170
Caching JavaScript Files 171
Working Around Caching Issues 172
Using a Content Delivery Network 173
Deploying JavaScript Resources 173
Agile JavaScript Build Process 174
Summary 175

10. Tools 177
JavaScript Profiling 178
YUI Profiler 179
Anonymous Functions 182
Firebug 183
Console Panel Profiler 183
Console API 184
Net Panel 185
Internet Explorer Developer Tools 186
Safari Web Inspector 188
Profiles Panel 189
Resources Panel 191
Chrome Developer Tools 192
Script Blocking 193
Page Speed 194
Fiddler 196
YSlow 198
dynaTrace Ajax Edition 199
Summary 202

Index 203

Preface

When JavaScript was first introduced as part of Netscape Navigator in 1996, performance wasn't that important. The Internet was in its infancy and it was, in all ways, slow. From dial-up connections to underpowered home computers, surfing the Web was more often a lesson in patience than anything else. Users expected to wait for web pages to load, and when the page successfully loaded, it was a cause for celebration.

JavaScript's original goal was to improve the user experience of web pages. Instead of going back to the server for simple tasks such as form validation, JavaScript allowed embedding of this functionality directly in the page. Doing so saved a rather long trip back to the server. Imagine the frustration of filling out a long form, submitting it, and then waiting 30–60 seconds just to get a message back indicating that you had filled in a single field incorrectly. JavaScript can rightfully be credited with saving early Internet users a lot of time.

The Internet Evolves

Over the decade that followed, computers and the Internet continued to evolve. To start, both got much faster. The rapid speed-up of microprocessors, the availability of cheap memory, and the appearance of fiber optic connections pushed the Internet into a new age. With high-speed connections more available than ever, web pages started becoming heavier, embedding more information and multimedia. The Web had changed from a fairly bland landscape of interlinked documents into one filled with different designs and interfaces. Everything changed, that is, except JavaScript.

What previously was used to save server roundtrips started to become more ubiquitous. Where there were once dozens of lines of JavaScript code were now hundreds, and eventually thousands. The introduction of Internet Explorer 4 and dynamic HTML (the ability to change aspects of the page without a reload) ensured that the amount of JavaScript on pages would only increase over time.

The last major step in the evolution of browsers was the introduction of the Document Object Model (DOM), a unified approach to dynamic HTML that was adopted by Internet Explorer 5, Netscape 6, and Opera. This was closely followed by the

standardization of JavaScript into ECMA-262, third edition. With all browsers supporting the DOM and (more or less) the same version of JavaScript, a web application platform was born. Despite this huge leap forward, with a common API against which to write JavaScript, the JavaScript engines in charge of executing that code remained mostly unchanged.

Why Optimization Is Necessary

The JavaScript engines that supported web pages with a few dozen lines of JavaScript in 1996 are the same ones running web applications with thousands of lines of JavaScript today. In many ways, the browsers fell behind in their management of the language and in doing the groundwork so that JavaScript could succeed at a large scale. This became evident with Internet Explorer 6, which was heralded for its stability and speed when it was first released but later reviled as a horrible web application platform because of its bugs and slowness.

In reality, IE 6 hadn't gotten any slower; it was just being asked to do more than it had previously. The types of early web applications being created when IE 6 was introduced in 2001 were much lighter and used much less JavaScript than those created in 2005. The difference in the amount of JavaScript code became clear as the IE 6 JavaScript engine struggled to keep up due to its static garbage-collection routine. The engine looked for a fixed number of objects in memory to determine when to collect garbage. Earlier web application developers had run into this threshold infrequently, but with more JavaScript code comes more objects, and complex web applications began to hit this threshold quite often. The problem became clear: JavaScript developers and web applications had evolved while the JavaScript engines had not.

Although other browsers had more logical garbage collection routines, and somewhat better runtime performance, most still used a JavaScript interpreter to execute code. Code interpretation is inherently slower than compilation since there's a translation process between the code and the computer instructions that must be run. No matter how smart and optimized interpreters get, they always incur a performance penalty.

Compilers are filled with all kinds of optimizations that allow developers to write code in whatever way they want without worrying whether it's optimal. The compiler can determine, based on lexical analysis, what the code is attempting to do and then optimize it by producing the fastest-running machine code to complete the task. Interpreters have few such optimizations, which frequently means that code is executed exactly as it is written.

In effect, JavaScript forces the developer to perform the optimizations that a compiler would normally handle in other languages.

Next-Generation JavaScript Engines

In 2008, JavaScript engines got their first big performance boost. Google introduced their brand-new browser called Chrome. Chrome was the first browser released with an optimizing JavaScript engine, codenamed V8. The V8 JavaScript engine is a just-in-time (JIT) compilation engine for JavaScript, which produces machine code from JavaScript code and then executes it. The resulting experience is blazingly fast JavaScript execution.

Other browsers soon followed suit with their own optimizing JavaScript engines. Safari 4 features the Squirrel Fish Extreme (also called Nitro) JIT JavaScript engine, and Firefox 3.5 includes the TraceMonkey engine, which optimizes frequently executed code paths.

With these newer JavaScript engines, optimizations are being done at the compiler-level, where they should be done. Someday, developers may be completely free of worry about performance optimizations in their code. That day, however, is still not here.

Performance Is Still a Concern

Despite advancements in core JavaScript execution time, there are still aspects of JavaScript that these new engines don't handle. Delays caused by network latency and operations affecting the appearance of the page are areas that have yet to be adequately optimized by browsers. While simple optimizations such as function inlining, code folding, and string concatenation algorithms are easily optimized in compilers, the dynamic and multifaceted structure of web applications means that these optimizations solve only part of the performance problem.

Though newer JavaScript engines have given us a glimpse into the future of a much faster Internet, the performance lessons of today will continue to be relevant and important for the foreseeable future.

The techniques and approaches taught in this book address many different aspects of JavaScript, covering execution time, downloading, interaction with the DOM, page life cycle, and more. Of these topics only a small subset, those related to core (ECMAScript) performance, could be rendered irrelevant by advances in JavaScript engines, but that has yet to happen.

The other topics cover ground where faster JavaScript engines won't help: DOM interaction, network latency, blocking and concurrent downloading of JavaScript, and more. These topics will not only continue to be relevant, but will become areas of further focus and research as low-level JavaScript execution time continues to improve.

How This Book Is Organized

The chapters in this book are organized based on a normal JavaScript development life cycle. This begins, in Chapter 1, with the most optimal ways to load JavaScript onto the page. Chapter 2 through Chapter 8 focus on specific programming techniques to help your JavaScript code run as quickly as possible. Chapter 9 discusses the best ways to build and deploy your JavaScript files to a production environment, and Chapter 10 covers performance tools that can help you identify further issues once the code is deployed. Five of the chapters were written by contributing authors:

- Chapter 3, *DOM Scripting*, by Stoyan Stefanov
- Chapter 5, *Strings and Regular Expressions*, by Steven Levithan
- Chapter 7, *Ajax*, by Ross Harmes
- Chapter 9, *Building and Deploying High-Performance JavaScript Applications*, by Julien Lecomte
- Chapter 10, *Tools*, by Matt Sweeney

Each of these authors is an accomplished web developer who has made important contributions to the web development community as a whole. Their names appear on the opening page of their respective chapters to more easily identify their work.

JavaScript Loading

Chapter 1, *Loading and Execution*, starts with the basics of JavaScript: getting code onto the page. JavaScript performance really begins with getting the code onto a page in the most efficient way possible. This chapter focuses on the performance problems associated with loading JavaScript code and presents several ways to mitigate the effects.

Coding Technique

A large source of performance problems in JavaScript is poorly written code that uses inefficient algorithms or utilities. The following seven chapters focus on identifying problem code and presenting faster alternatives that accomplish the same task.

Chapter 2, *Data Access*, focuses on how JavaScript stores and accesses data within a script. Where you store data is just as important as what you store, and this chapter explains how concepts such as the scope chain and prototype chain can affect your overall script performance.

Stoyan Stefanov, who is well versed in the internal workings of a web browser, wrote Chapter 3, *DOM Scripting*. Stoyan explains that DOM interaction is slower than other parts of JavaScript because of the way it is implemented. He covers all aspects of the DOM, including a description of how repaint and reflow can slow down your code.

Chapter 4, *Algorithms and Flow Control*, explains how common programming paradigms such as loops and recursion can work against you when it comes to runtime performance. Optimization techniques such as memoization are discussed, as are browser JavaScript runtime limitations.

Many web applications perform complex string operations in JavaScript, which is why string expert Steven Levithan covers the topic in Chapter 5, *Strings and Regular Expressions*. Web developers have been fighting poor string-handling performance in browsers for years, and Steven explains why some operations are slow and how to work around them.

Chapter 6, *Responsive Interfaces*, puts the spotlight firmly on the user experience. JavaScript can cause the browser to freeze as it executes, leaving users extremely frustrated. This chapter discusses several techniques to ensure that the user interface remains responsive at all times.

In Chapter 7, *Ajax*, Ross Harmes discusses the best ways to achieve fast client-server communication in JavaScript. Ross covers how different data formats can affect Ajax performance and why `XMLHttpRequest` isn't always the best choice.

Chapter 8, *Programming Practices*, is a collection of best practices that are unique to JavaScript programming.

Deployment

Once JavaScript code is written and tested, it's time to make the changes available to everyone. However, you shouldn't just push out your raw source files for use in production. Julien Lecomte shows how to improve the performance of your JavaScript during deployment in Chapter 9, *Building and Deploying High-Performance JavaScript Applications*. Julien discusses using a build system to automatically minify files and using HTTP compression to deliver them to the browser.

Testing

When all of your JavaScript code is deployed, the next step is to begin performance testing. Matt Sweeney covers testing methodology and tools in Chapter 10, *Tools*. He discusses how to use JavaScript to measure performance and also describes common tools both for evaluating JavaScript runtime performance and for uncovering performance problems through HTTP sniffing.

Who This Book Is For

This book is aimed at web developers with an intermediate-to-advanced understanding of JavaScript who are looking to improve the performance of web application interfaces.

Conventions Used in This Book

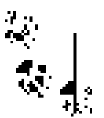
The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.



This icon signifies a tip, suggestion, or general note.

This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*High Performance JavaScript*, by Nicholas C. Zakas. Copyright 2010 Yahoo!, Inc., 978-0-596-80279-0."

If you feel your use of code examples falls outside fair use or the permission given here, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at *http://my.safaribooksonline.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/9780596802790

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our website at:

http://www.oreilly.com

Acknowledgments

First and foremost, I'd like to thank all of the contributing authors: Matt Sweeney, Stoyan Stefanov, Stephen Levithan, Ross Harmes, and Julien Lecomte. Having their combined expertise and knowledge as part of this book made the process more exciting and the end result more compelling.

Thanks to all of the performance gurus of the world that I've had the opportunity to meet and interact with, especially Steve Souders, Tenni Theurer, and Nicole Sullivan. You three helped expand my horizons when it comes to web performance, and I'm incredibly grateful for that.

A big thanks to everyone who reviewed the book prior to publication, including Ryan Grove, Oliver Hunt, Matthew Russell, Ted Roden, Remy Sharp, and Venkateswaran Udayasankar. Their early feedback was invaluable in preparing the book for production.

And a huge thanks to everyone at O'Reilly and Yahoo! that made this book possible. I've wanted to write a book for Yahoo! ever since I joined the company in 2006, and Yahoo! Press was a great way to make this happen.

CHAPTER 1
Loading and Execution

JavaScript performance in the browser is arguably the most important usability issue facing developers. The problem is complex because of the blocking nature of JavaScript, which is to say that nothing else can happen while JavaScript code is being executed. In fact, most browsers use a single process for both user interface (UI) updates and JavaScript execution, so only one can happen at any given moment in time. The longer JavaScript takes to execute, the longer it takes before the browser is free to respond to user input.

On a basic level, this means that the very presence of a `<script>` tag is enough to make the page wait for the script to be parsed and executed. Whether the actual JavaScript code is inline with the tag or included in an external file is irrelevant; the page download and rendering must stop and wait for the script to complete before proceeding. This is a necessary part of the page's life cycle because the script may cause changes to the page while executing. The typical example is using `document.write()` in the middle of a page (as often used by advertisements). For example:

```
<html>
<head>
   <title>Script Example</title>
</head>
<body>
  <p>
  <script type="text/javascript">
    document.write("The date is " + (new Date()).toDateString());
  </script>
  </p>
</body>
</html>
```

When the browser encounters a `<script>` tag, as in this HTML page, there is no way of knowing whether the JavaScript will insert content into the `<p>`, introduce additional elements, or perhaps even close the tag. Therefore, the browser stops processing the page as it comes in, executes the JavaScript code, then continues parsing and rendering the page. The same takes place for JavaScript loaded using the `src` attribute; the browser must first download the code from the external file, which takes time, and then parse

and execute the code. Page rendering and user interaction are completely blocked during this time.

The two leading sources of information on JavaScript affecting page download performance are the Yahoo! Exceptional Performance team (*http://developer.yahoo.com/performance/*) and Steve Souders, author of *High Performance Web Sites* (O'Reilly) and *Even Faster Web Sites* (O'Reilly). This chapter is heavily influenced by their combined research.

Script Positioning

The HTML 4 specification indicates that a `<script>` tag may be placed inside of a `<head>` or `<body>` tag in an HTML document and may appear any number of times within each. Traditionally, `<script>` tags that are used to load external JavaScript files have appeared in the `<head>`, along with `<link>` tags to load external CSS files and other metainformation about the page. The theory was that it's best to keep as many style and behavior dependencies together, loading them first so that the page will come in looking and behaving correctly. For example:

```
<html>
<head>
   <title>Script Example</title>
   <-- Example of inefficient script positioning -->
   <script type="text/javascript" src="file1.js"></script>
   <script type="text/javascript" src="file2.js"></script>
   <script type="text/javascript" src="file3.js"></script>
   <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
  <p>Hello world!</p>
</body>
</html>
```

Though this code seems innocuous, it actually has a severe performance issue: there are three JavaScript files being loaded in the `<head>`. Since each `<script>` tag blocks the page from continuing to render until it has fully downloaded and executed the JavaScript code, the perceived performance of this page will suffer. Keep in mind that browsers don't start rendering anything on the page until the opening `<body>` tag is encountered. Putting scripts at the top of the page in this way typically leads to a noticeable delay, often in the form of a blank white page, before the user can even begin reading or otherwise interacting with the page. To get a good understanding of how this occurs, it's useful to look at a waterfall diagram showing when each resource is downloaded. Figure 1-1 shows when each script and the stylesheet file get downloaded as the page is loading.

Figure 1-1 shows an interesting pattern. The first JavaScript file begins to download and blocks any of the other files from downloading in the meantime. Further, there is

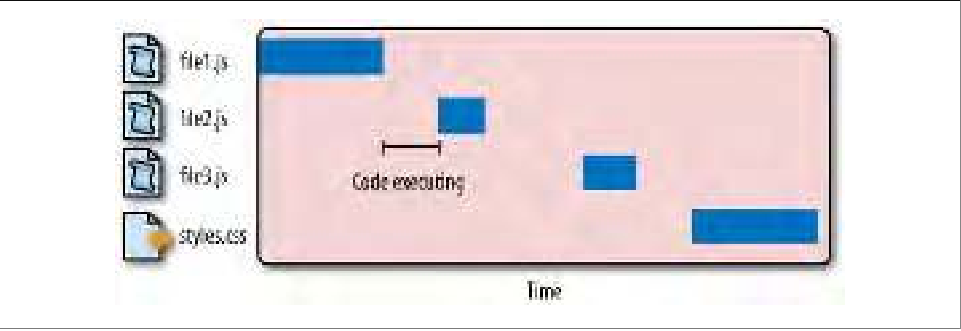


Figure 1-1. JavaScript code execution blocks other file downloads

a delay between the time at which *file1.js* is completely downloaded and the time at which *file2.js* begins to download. That space is the time it takes for the code contained in *file1.js* to fully execute. Each file must wait until the previous one has been downloaded and executed before the next download can begin. In the meantime, the user is met with a blank screen as the files are being downloaded one at a time. This is the behavior of most major browsers today.

Internet Explorer 8, Firefox 3.5, Safari 4, and Chrome 2 all allow parallel downloads of JavaScript files. This is good news because the `<script>` tags don't necessarily block other `<script>` tags from downloading external resources. Unfortunately, JavaScript downloads still block downloading of other resources, such as images. And even though downloading a script doesn't block other scripts from downloading, the page must still wait for the JavaScript code to be downloaded and executed before continuing. So while the latest browsers have improved performance by allowing parallel downloads, the problem hasn't been completely solved. Script blocking still remains a problem.

Because scripts block downloading of all resource types on the page, it's recommended to place all `<script>` tags as close to the bottom of the `<body>` tag as possible so as not to affect the download of the entire page. For example:

```
<html>
<head>
   <title>Script Example</title>
   <link rel="stylesheet" type="text/css" href="styles.css">
</head>
<body>
  <p>Hello world!</p>

   <-- Example of recommended script positioning -->
   <script type="text/javascript" src="file1.js"></script>
   <script type="text/javascript" src="file2.js"></script>
   <script type="text/javascript" src="file3.js"></script>
</body>
</html>
```

This code represents the recommended position for `<script>` tags in an HTML file. Even though the script downloads will block one another, the rest of the page has

already been downloaded and displayed to the user so that the entire page isn't perceived as slow. This is the Yahoo! Exceptional Performance team's first rule about JavaScript: put scripts at the bottom.

Grouping Scripts

Since each `<script>` tag blocks the page from rendering during initial download, it's helpful to limit the total number of `<script>` tags contained in the page. This applies to both inline scripts as well as those in external files. Every time a `<script>` tag is encountered during the parsing of an HTML page, there is going to be a delay while the code is executed; minimizing these delays improves the overall performance of the page.

Steve Souders has also found that an inline script placed after a `<link>` tag referencing an external stylesheet caused the browser to block while waiting for the stylesheet to download. This is done to ensure that the inline script will have the most correct style information with which to work. Souders recommends never putting an inline script after a `<link>` tag for this reason.

The problem is slightly different when dealing with external JavaScript files. Each HTTP request brings with it additional performance overhead, so downloading one single 100 KB file will be faster than downloading four 25 KB files. To that end, it's helpful to limit the number of external script files that your page references.

Typically, a large website or web application will have several required JavaScript files. You can minimize the performance impact by concatenating these files together into a single file and then calling that single file with a single `<script>` tag. The concatenation can happen offline using a build tool (discussed in Chapter 9) or in real-time using a tool such as the Yahoo! combo handler.

Yahoo! created the combo handler for use in distributing the Yahoo! User Interface (YUI) library files through their Content Delivery Network (CDN). Any website can pull in any number of YUI files by using a combo-handled URL and specifying the files to include. For example, this URL includes two files: *http://yui.yahooapis.com/combo?2.7.0/build/yahoo/yahoo-min.js&2.7.0/build/event/event-min.js*.

This URL loads the 2.7.0 versions of the *yahoo-min.js* and *event-min.js* files. These files exist separately on the server but are combined when this URL is requested. Instead of using two `<script>` tags (one to load each file), a single `<script>` tag can be used to load both:

sample content of High Performance JavaScript (Build Faster Web Application Interfaces)

- **read online My Grandmother's Chinese Kitchen: 100 Family Recipes and Life Lessons for free**
- read online The Strategy of Campaigning: Lessons from Ronald Reagan and Boris Yeltsin pdf, azw (kindle)
- **read online The Enlightenment's Fable: Bernard Mandeville and the Discovery of Society (Ideas in Context) pdf, azw (kindle), epub**
- *Illustrated Synopsis of Dermatology & Sexually Transmitted Diseases (4th Edition) online*

- http://test1.batsinbelfries.com/ebooks/My-Grandmother-s-Chinese-Kitchen--100-Family-Recipes-and-Life-Lessons.pdf
- http://diy-chirol.com/lib/Aces-Back-to-Back--The-History-of-the-Grateful-Dead--1965-2013.pdf
- http://kamallubana.com/?library/The-Enlightenment-s-Fable--Bernard-Mandeville-and-the-Discovery-of-Society--Ideas-in-Context-.pdf
- http://redbuffalodesign.com/ebooks/The-Animated-Movie-Guide.pdf