JavaScript Rocks! presents...

JavaScript Performance Rocks

by Thomas Fuchs & Amy Hoy

find more awesome JavaScript stuff at http://www.jsrocks.com

Contents

INTRODUCTION

JavaScript Performance Rocks!

Everybody's Favorite Beta Book

Welcome to our beta book. Thanks for purchasing!

How this book works

Short, sweet, funny—that's our goal.

This Book is a Beta Book

This book is in BETA. That means it's not done. There are some holes in our narrative, and we're adding new content all the time.

BETA ALERT!

Right now, for example, only a couple of our sweet infographics are placed in the book at all... but believe us, we're working on more. And some of the writing's pretty rough, too. (But the code's all been checked. However, we're only human.)

As a purchaser of the beta book, you're entitled to free updates, until the book's complete.

We'll put you on our mailing list, with the purchasing email address used when you paid with PayPal. (If you don't hear from us regarding this, drop us a line at amy@slash7.com.)

And we'd like your feedback!

Licensing & Cost

This ebook **is not open source**. We reserve all commercial and moral rights to the book, materials, and supplied code (where not under an existing license).

However, we have no interest in limiting or taking away your fair use rights to excerpt, review, criticize, and parody. Go right ahead!

This ebook costs $24 a seat for individuals during the beta period, $29 afterwards, and also comes with a subscription to the private email list and forum.

For site licenses of up to 100 individuals employed by the same corporate entity, the price is $199.

To put this into perspective, each one of us charges more for an hour of our consulting time than the 100-seat license.

Why This Book Costs Money & Why You Should Pay

We want to share our hard-won expertise with the world for a reasonable price so we can keep on creating free and cheap content, code, and art

for everyone to enjoy.

That "cheap and free content, code, and art" includes Scriptaculous itself, the hundreds of helpful and entertaining blog posts, and other such goodies that everyone benefits from.

Please buy a copy if you've received this copy without paying.

It costs the equivalent of two movie tickets or two trips to McDonald's. We know it'll be worth more than that to your projects and your business.

Now that we've dispensed with the preaching....

Sections

This book is divided into four major sections:

1. **Intro** (you are here)
2. **Loadtime** — loadtime performance problems & solutions
3. **Runtime** — issues that involve code actually running, & their solutions
4. **iPhone** — tips & tricks specially for the iPhone

With no further ado, let's get started!

Chapter 1

You, Performance. Performance, You.

In our outline for this report, this section was originally penciled in as "introduction to performance." So, dear reader, say hello to performance. You may not have met it yet, but you're about to.

Double éntendres or otherwise, "performance" has become a loaded word. It can be used to mean just about anything. But this isn't spam pimping le weekender, this is a book on one very specific thing: JavaScript performance in highly interactive web applications.

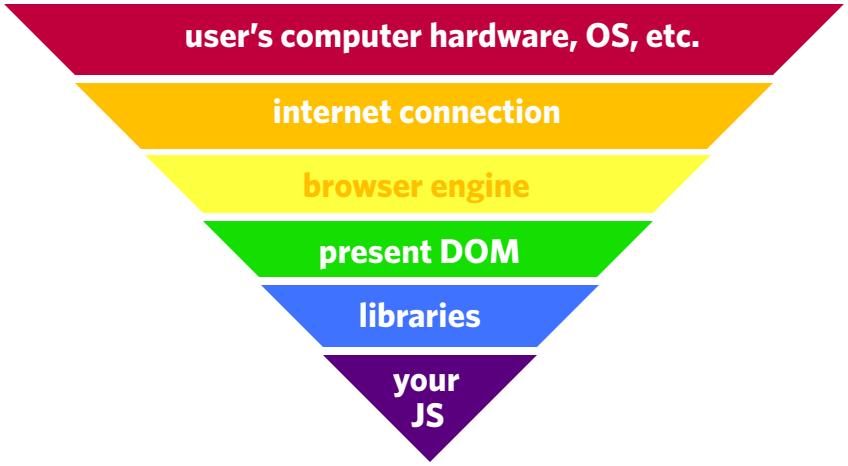
And when it comes to rich, JavaScript-heavy, highly interactive web applications, performance is a many-headed beast. It is a combination of objective metrics and perceptual ones.

In terms of real things you can measure, your web application's performance depends on this hodgepodge of technical factors:

- user's environment (hardware, software)
- network connection & bandwidth
- browser engine
- current DOM
- libraries
- your JS

The Inverted Pyramid

This pyramid describes the factors that determine your application's performance, in the order of the affect that they have. You must have noticed two things right away about this pyramid: 1) it's a rainbow (how cheesy), and 2) it's upside down.



While the rainbow was purely an aesthetic choice which we shall not bother to defend, the upsidedownness is another story. Simply put, we drew the sucker upside down to reflect reality.

The 3 biggest and baddest slices—your user's hardware & OS, their (& your) internet connection, and their browser—are typically outside of your control.

This is a performance fact you must not forget:

If you can't control it, you have to compensate for it.

Tattoo it on the back of your hand, sharpie it on the front of your forehead, or if you're normal, put it on a sticky note in blue ballpoint and slap it on your monitor. It's the number one rule of web app performance. Don't forget it.

And it will be a theme throughout this book.

If an app is on the network but there's no one there to use it, is it slow?

However, most genuine reasons (speaking seriously) fall into two major categories:

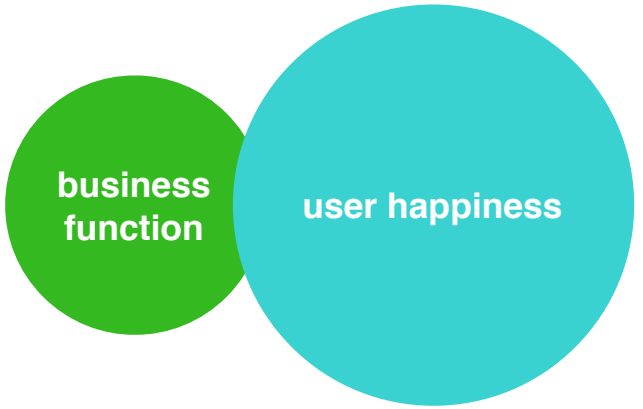
- critical to the application's function
- extremely important for the pleasure and/or sanity of users

The difference is in the observing: is the app's performance being judged by cold, falsifiable logic (necessary business function), or a soft, mushy human (is it affecting user happiness and that alone)?

Take, for example, the concept of perceived performance. You can tweak the HTML bones of a web app in a few ways such that it takes the same amount of time to download (true measurable performance) but 3-4x as long to render (a difference only in perceived performance).

Business Function

There are any number of applications where speed is, in fact, mission critical to the purpose of the application: game communication, business transactions that are time-sensitive, anything where it matters who was first or where there's another kind of deadline.



It's not exactly a Venn diagram because when something is critical to business function, it's simply critical. You can't get around it. The rest of the time, it's just about user happiness.

In these cases, you can measure the performance very objectively. You know the failure case:

Is the app performant? Well, did it respond such that the function isn't compromised due to delay? Did the game coordinates, buy order, or bid come in at the right time and in the right order?

While these types of performance problems can be very sticky wickets indeed, they are the performance problems you know about.

User Happiness

The majority of performance needs, however, fall under the headline of what we'll broadly call "user happiness."

Waiting for an important application to load—or worse, waiting for the application to respond—has been known to cause hair-tearing, keyboard-smashing frustration in human subjects.

The more important the application is to people, the more frustrating it is to wait. It doesn't matter whether "importance" is an official designation (e.g. for business), or just something they have incorporated into their daily lives (e.g. Twitter).

This—like happiness of any sort—is harder to test. It's hardly objective at all:

Is the app performant? Well, Joe keeps using it, but Mary Sue switched to an open source alternative. And Bob and Alice don't log in half as much as they used to. Chris and Sandy use it all the time (but what you don't see is the teeth-gritting and nasty talk around the watercooler and on their blogs).

Here you must rely on the inexact science of knowing how much crap people will take before they leave.

Your users often won't complain about your application's speed (at least not to your face). They may slowly disappear, or they may use it less, or they may grin and bear it—and strangle kittens when you're not looking.

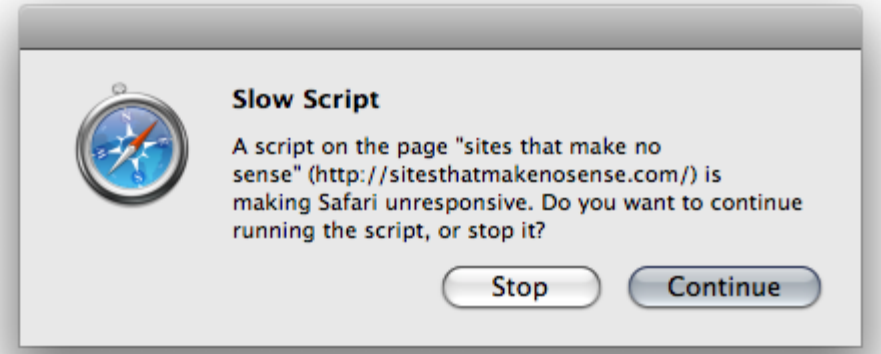
Unless you have psychic kittenvision, these are often the performance problems you don't know about. They're the scary ones.

Sins of premature optimization aside, user happiness—and the silence of user unhappiness—remains the number one reason to be proactive about your application's speed.

Beautiful cascade of failure

Go ahead and enter `javascript:while(true){}` in a browser near you. Notice how the complete browser window freezes, and you can't do a single thing until a dialog pops up.

Meet your worst case scenario:



Don't forget that JavaScript is single-threaded and one apple can spoil the whole bushel.

So. We care about performance, and we've begun to understand that, paradoxically, a lot of what constitutes performance is out of our control.

Time to get started.

Chapter 2

Is Your App Behaving Badly?

Or... Where Does It Hurt? Because It Must Hurt Somewhere

In the next two sections, we'll talk about how to kill all sorts of problems, and the interaction bandages you can use if you really, really can't work around them. But first, a holistic look at your app is in order.

You can't really say you know what position your app's in unless you measure, measure, and measure again.

Traditional web sites are fairly easy to handle, when it comes to performance: you look at page weight, you look at server generation time, and you do the math.

And if the server time is way too long, you dig into the code and the database. This is less simple, but we've had a lot of years to come up with a good process.

Highly interactive apps are a whole 'nother level of complicated.

Adjust your toolbelt, Batman, because it's about to get a lot heavier.

The 3 levels of measurement

Measuring tools & approaches fall into roughly three categories, with increasing levels of detail. To get a complete picture of your app's performance health, you'll need to use all three.

Assessing.

These tools measure the big picture: page size, download speed, DOM complexity, render time, memory usage, whether your caching is good, that sort of thing.

Profiling.

When you profile, you are specifically timing parts of your code's execution time. Profiling tools look at all your code, as it runs, and time each bit of it.

Because of the way profiling tools work, these numbers are not wholly reliable, but they pretty much get the job done.

Benchmarking.

When profiling doesn't go far enough, you benchmark. Benchmarking tools don't really exist, because to get truly accurate numbers, you've got to write customer timers inside your own JavaScript. When each millisecond counts, you don't want to be using a browser plugin, because that will skew the results.

Your Toolbox

For success in profiling, you need the following tools:

Firefox 2

The venerable older version of the web developer's best browser buddy. Make sure to get the right version of Firefox and Firebug, or Firebug won't work properly for you.

http://www.mozilla.com/en-US/firefox/all-older.html

Firebug 1.05

The venerable older version (that is, the version that works) of the number one web developer Firefox extension.

https://addons.mozilla.org/en-US/firefox/addons/versions/1843

YSlow

This plugin for Firebug gives you the skinny on page weight and caching

https://addons.mozilla.org/en-US/firefox/addon/5369

WebKit Nightlies

For both Windows and OS X. These "future Safari" builds offer profiling tools not available in the regular, final Safari.

http://nightly.webkit.org/

DOMMonster

Our utility—a bookmarklet that inspects web application health. Works in Safari & Firefox, also WebKit nightlies.

Packaged with this ebook!

Why are we using an old Firefox? Firebug doesn't work properly in Firefox 3 (and most of the other tools don't seem to, either). When all the tools work in Firefox 3, feel free to upgrade. Although keeping an older browser doesn't hurt.

Strata in the Problemosphere

Where (solvable) problems chiefly occur:

loadtime

- Page weight
- Number of files
- Caching
- DOM complexity
- <script /> practices
- serving set-up

runtime

- code complexity
- DOM complexity
- slow access methods / libraries / wrapping functions / data parsing
- excessive memory use / leaks
- runaway event handlers
- unoptimized code

Custom Benchmarking

Do you want to know if you've thoroughly optimized to the millisecond every `if()`, `and()` or `but()`?

You want to write your own benchmarks.

Regular Profiling is Nice, Benchmarking is Hardcore

Regular profiling tools—like the ones included in Webkit and Firebug—will give you general idea for what sucks, and why.

For most apps, that's enough. In most situations, your performance issues will be big, noticeable ones: lots of nested elements, poor choices for DOM selectors, giant unoptimized loops, that sort of thing.

But maybe you've already knocked out those issues. And maybe your project is a project where every single little CPU cycle counts.

Then you want to get hardcore. You want to optimize to the fullest extent.

We respect that. And we'll teach you how.

Pssst, the secret is... volume

The bottom line is, if you want to achieve meaningful numbers, you have to write your own code to do it, and you've got to run that code a lot.

You have to roll your own because the general profiling tools distort results. They have to—that's just the way that they work. You can't monitor every method called without changing the performance. It's the Heisenbug Uncertainty Principle.

Measurements taken within JavaScript itself are better and more reliable.

Creating Meaningful Benchmarks

Here's how you do meaningful benchmarks:

- You need a way to time method calls. That's what the time method does, surprise!
- You need to log it somewhere. For Webkit browsers and Firebug, we log into the console (in Webkit nightlies, alt-apple-C or peruse the Develop menu, or the console window in Firebug). On others, we just do alerts.
- Call the test script form an onclick event on the page (this makes sure the page is properly loaded before any tests begin!)
- Have a warm-up section in your profiling code – do something that eats some cycles before the real tests begin, to make sure the JavaScript runtime is in a non-initializing state (this prevents distorted results)
- Run the tests multiple times. Compare across browsers. It might be necessary to use a different amount of iterations, depending on the browser. For example, for certain operations IE is 100 times or so slower than recent Webkit JavaScript engines.

This code will set up our test. We're using Firebug's custom console logging features, so you should run this in Firefox with Firebug enabled.

```
<script type="text/javascript">
  if(typeof window['console'] == 'undefined')
   var console = { log: alert };

  function time(scope){
    time.scope = time.scope || {};
    if(time.scope[scope]) {
      var duration = (new Date()).getTime()-time.scope[scope];
      time.scope[scope] = null;
      console.log(scope+': '+(duration/1000).toFixed(3)+'s');
    } else {
      time.scope[scope] = (new Date()).getTime();
    }
  }

  // functions we want to compare
  function method1(n){
    n |= 1;
  }

  function method2(n){
    n = n || 1;
  }
</script>
```

Now that we've set up the things we're testing, it's time to set up the iterations and run them.

```
  var ITERATIONS = 100000;
  function test(){
    // first do a warm up phase, so there are
    // no sudden initializations. this helps
    // normalize the results
    i=ITERATIONS;
    while(i--) m2 = method2(1);

    // next run the timings
    var i = ITERATIONS;
    time('method1');
    while(i--) m1 = method1(1);
    time('method1');

    i=ITERATIONS;
    time('method2');
    while(i--) m2 = method2(1);
    time('method2');
  }
</script>

<a href="#" onclick="test()">run test</a>
```

Getting accurate results

Unfortunately, your web app doesn't run in a vacuum. Not for your users, and not for you, either. Not even when you're profiling.

Anything that's going on will affect your results: normal, everyday stuff like your browser cleaning up its JavaScript object cache, or a backup running in the background, or your virtual memory swapping.

If you want accurate results, you're going to have to eliminate these variables.

Here are some Rules of Thumbs for sane measurements with pure JavaScript:

1. Close other applications. Axe background processes. Do not run while your computer is backing up. Stop those torrents, you piratey pirate you.
2. Disable all browser plugins (including Firebug!)
3. Run thousands or millions of iterations, and repeat the test several times
4. Don't just test in one browser, but all browsers you target, on all platforms
5. Run your test. Run it again. And again. And again. See how the numbers compare.

Don't forget, your computer is not the same as your users' computers. You can do some ninja tactics to see how things perform in the real world. Do your profiling thing—not too intensive, please—and quietly post the results back to your server with Ajax.

Chapter 3

Welcome to Loadtime

YOU'RE GOING TO SPENDING A LOT OF TIME HERE. A WHOOOLE LOT.

Welcome to loadtime, the magical in-between place between your user's browser's HTTP request and the fully downloaded, fully rendered page.

If you're going to optimize, you should always start here. There's tasty low-hanging fruit, just waiting to be picked.

This is where the majority of performance issues occur.

Feel free to implement any or all of the suggestions here without the fear of premature optimization. (Except for excessive minifying. You heard it here first.)

THAT PAGE JUST TAKES FOREVER!

You know you've got a loadtime problem when it takes a long time for your page to "finish."

That means from the time your user clicks on a link or bookmark to the point where he can interact with everything without getting the spinning pizza (or tilting hourglass) of death.

The worst offender in my book? eBay. They have millions of people literally fighting to give them money, but the page takes 2-3 minutes to download and stop churning. Although GoDaddy's pretty high up there, too.

Phases of loadtime include:

- connecting to your server
- transferring all related files (including remote scripts)
- parsing and display the DOM
- rendering images
- executing any JavaScript that needs to get run

Your users hate loadtime problems, because they want to interact with your app already. That's why they're your users, after all.

They don't care where the problems occur or why, they just want to click stuff.

Two types of loadtime laments

Loadtime slow-downs fall into two camps:

download speed issues:

- pure "page weight"
- too many files
- bad or no caching
- slow servers (both software/hardware & bandwidth)
- unresponsive remote script hosts

browser churning issues:

- complex DOM
- bad script load order
- slow computer
- old browser

How we'll fix it

These are the types of remedies at your disposal for loadtime issues:

Non-invasive techniques (mostly):

- loading JS at the proper points
- appropriate caching
- reducing the number of files
- in-lining and precaching
- reducing file size
- increasing the number of possible simultaneous browser streams

And, more invasive but very effective techniques:

- reducing page / DOM complexity (cuts down churning and file size)
- on-demand JavaScript

And one final technique, easy to say but not always fun to implement:

- hosting on a faster server (and/or with better bandwidth)

Making the right choices for your app

You want to do the least amount of work for the most pay-off, right?

Perfect. That's the right attitude for tuning your JavaScript-heavy web app.

The biggest gains can often be had in the "non-invasive" realm: especially good caching and file compression. And they don't involve tweaking your code, not even a little bit—it's a win/win situation.

We recommend you try the non-invasive techniques first, followed by checking your DOM complexity.

You can also use our **DOM Monster** tool and YSlow to identify loadtime trouble areas & beat them into submission.

Moving Forward

Several chapters follow this one, each featuring one of the techniques we'll be using to tune your loadtime experience. These chapters all begin with 'Loadtime.'

Chapter 4

Loadtime: Script Load Order

THE EASIEST & MOST EFFECTIVE FIX, EVER.

Fun browser fact: the browser won't begin to render the very HTML and CSS of your web app unless all JavaScript files are fully loaded (or just about).

For the fastest, simplest, and easiest performance increase you can get... put your `<script />` tags at the very bottom of your `<body />` tag.

Biggest Bang for Your Buck

Placing your script tags at the end won't improve your objective, measurable page load speed (bandwidth determines that).

But it does mean that the browser won't stop rendering the page while it waits for the JavaScript files to transfer and do their thing—drastically improving the perceptual experience for your users.

Implementing this fix can take a first-load experience involving several seconds of blank browser window churnage, and turn it into a page that loads progressively. Sweet!

Especially for Remote Services

This is especially beneficial for external JavaScript sources like Google

Analytics and other such third-party stuff.

When the server behind this external stuff (or the network between you & them) is slow or unresponsive—which seems to happen regularly to various stats services—your app will seemingly take forever to load. With the script tags at the end, at least, your user won't be stuck with a blank screen.

Caveat: Progressive Functionality

The only real caveat with this trick is that the JavaScript functionality of DOM elements won't be available until the JavaScript files are fully transferred and executed.

Makes sense, right? This is a rare side effect but it can occur.

Possible example: that autocomplete text field you have will appear almost immediately, but not necessarily function right away. The strength (or even presence) of this effect is completely determined by bandwidth and browser speed. When everything's snappy, it probably won't be an issue at all.

Chapter 5

Loadtime: The Cachét of Caching

So Good, It's like Christmas for your Software

Caching should be the second tool you reach for, the moment your application goes live.

Browsers have cached files automatically since the days of dial-up, but they're not the smartest cookies in the cookiejar.

Browsers don't keep caches very long or necessarily use them in an intelligent manner.

Luckily, they do respond to commands—you can easily configure your web server or server-side scriting language to order the browsers around, and tell them to keep the right files for longer.

Expiration Headers

Specifically, you want to set the expiration headers on your content files that don't change often—CSS and, of course, JavaScript files.

The goal is to have the browser keep those copies of these files until they change, so that the latest version is always right there on the user's computer.

This single change can save tens of time-wasting requests per page view.

These techniques require you to have access to your web server configuration. So get to it!

Or order your hosting company to do it for you.

The Final Result

You want your finished headers to look something like this:

```
Date: Tue, 22 Jul 2008 19:08:42 GMT
Expires: Wed, 23 Jul 2008 19:08:42 GMT
Cache-Control: max-age=86400
```

In this case, the file was downloaded on July 22nd and will be held until July 23rd—24 hours in the future—at the minimum.

The browser will then check to see if the file has changed and, if so, download the new version. If no changes have occurred, it'll keep happily humming along with the cached copy.

The last line is what makes it happen:

```
Cache-Control: max-age=86400
```

The `Cache-Control` header takes an argument of max-age in seconds; `86,400` seconds is 24 hours.

Caching Strategies

There are two major strategies for caching JavaScript and CSS assets:

- very long cache periods (e.g. months or years; "far future cache")
- short or medium cache periods (days) ("short cache")

They've both got their pros and cons, and, unfortunately, both require a different support system to work right.

For sake of argument, let's pretend we've got this interesting little file that we want to cache and it's called `our_app.js`.

Here's how it'd work with both strategies.

Far future cache periods

With a far-future caching setup, you set the expiration date far in the future (bug surprise)—months or years. That's a long time.

This seems ideal, because then your user's caches will be safe and their experience will be snappy until many happy months go by.

But meanwhile, back at the data center, you want to roll out your spiffy new psychic autocompleter—but nobody will know because you originally set a far-future cache expiration date to 2010.

If you're going to do far-future caching periods, there's just one way to distribute new code & content: change the filename.

The browser will say "Hey, I've got `ourapp.js` but not `ourappv2.js` in my cache, it must be new" and slurp it down fresh.

Yahoo! does this by including a version in their JavaScript filenames:

```
ourapp_2.3.4.js
```

Ruby on Rails does this automatically by adding a query string to the end of the base filename, that reflects the last modification date of the file:

```
ourapp.js?20080822
```

You can take whichever approach you like, or think up your own.

As long as you change the filename when you need to push changes to your users, far-future caching will work for you.

Short cache periods

With short caching periods, the "expiration date" on your cached files comes up much sooner.

When that happens, your user's browser will ping the server for the

`Last-Modified` header on the file.

If the date's different than the copy it's already got saved, the browser will download it fresh.

The browser will always check for the same file name. If you cache ourapp.js with a 24-hour maximum age, the browser will ask for the headers for `ourapp.js` every 24 hours. Rinse and repeat.

You should stick to far-future caching unless you don't have control of your app's filenames.

If you can't change the filename, try the short-cache approach.

Configuring Apache 1.x and 2.x

It's easy to configure Apache to send the correct caching headers using the built-in `ExpiresActive` extension.

Add the following (version appropriate) code to your conf file if it's not already there.

First, you need to load and activate the module:

Loading the Module in Apache 1.3

```
LoadModule expires_module libexec/mod_expires.so
AddModule mod_expires.c
```

Loading the Module in Apache 2.0

```
LoadModule expires_module modules/mod_expires.so
```

Be sure to place the above snippet inside the correct block for the domain name / application you're configuring.

Far future caching (5 years)

```
ExpiresActive on
ExpiresByType text/css “access plus 5 years”
ExpiresByType application/x-javascript “access plus 5 years”
ExpiresByType text/javascript “access plus 5 years”
FileETag none
```

This sets all items to 5 years.

Short cache horizon (24 hours for JS, 5 days for CSS):

```
ExpiresActive on
ExpiresByType text/css “access plus 5 days”
ExpiresByType application/x-javascript “access plus 24 hours”
ExpiresByType text/javascript “access plus 24 hours”
FileETag none
```

Configuring nginx

If you're using nginx, it's simply a matter of putting this nice concise snippet inside your `server{}` configuration block:

Far future caching (5 years for all CSS & JavaScript)

```
location ~* \.(js|css)$ {
  if (-f $request_filename) {
    expires 5y;
    break;
  }
}
```

Short cache horizon (24 hours for JS, 5 days for CSS):

```
location ~* \.css$ {
  if (-f $request_filename) {
    expires 5d;
    break;
  }
}

location ~* \.js$ {
  if (-f $request_filename) {
    expires 24h;
    break;
  }
}
```

Configuring Lighttpd

And, to complete the web server triumvirate, we come to Lighttpd.

First, enable mod_expire in the server.modules directive:

```
code sample missing
```

You'll want to place the following snippets into the configuration file for the application you want to affect.

Far future caching (5 years for all CSS & JavaScript):

```
$HTTP[“url”] =~ “\.(css|js)$” {
     expire.url = ( “” => “access 5 years” )
}
```

```
Short cache horizon (24 hours for JS, 5 days for CSS):

$HTTP[“url”] =~ “\.css$” {
     expire.url = ( “” => “access 5 days” )
}
$HTTP[“url”] =~ “\.js$” {
     expire.url = ( “” => “access 24 hours” )
}
```

Caching Gotchas: JSON & Generated JavaScript

Be careful not to cache things you don't want to cache—namely data and generated JavaScript that's customized to your users.

The easiest way to avoid this problem is to give your JSON and generated JavaScript code a different file extension.

We call our data files `ourdata.json`, for example.

You might use a custom extension for generated code, too.

Chapter 6

Loadtime: Con-ca-te-nation

Fewer Files is Better.

Most web apps are composed of scores of little files. It's just convenient to write them that way, and it makes it easy to find the code you're looking for when you're developing.

Here's another tasty low-hanging fruit: Reduce the number of files you pump to the browser.

Your web app, like so many others, is undoubtedly composed of a million little files.

Unfortunately, browsers (by default) only open 2 to 4 simultaneous connections to the server, regardless of how many stylesheets, images, and JavaScript files are coming down the pipe.

The worst offender in my book? eBay. They have millions of people literally fighting to give them money, but the page takes 2-3 minutes to download and stop churning. Although GoDaddy's pretty high up there, too.

The browser won't start fetching the next batch of files until the current batch of 2 to 4 are done.

Even worse, browsers can't load JavaScript files in parallel: doing so would potentially create procedural issues with interdependent code.

So, reduce the number of files the browser has to load and you can

improve load performance. You probably already do this for images and CSS (and if you don't, you should); doing it for JavaScript can only help.

The trick is to structure your files effectively for both development and production environments: when you're writing code, break out the files however makes sense to you; when you push to the server, smush the files together, preferrably into one file. Be mindful of the order of the code in the file(s) is sane.

What to Concatenate

Your project probably has some combination of library files, original code, and data in JSON or other JavaScript format.

This goes for all the files for any JavaScript libraries you may be using—Prototype, jquery, mootools, YUI, ext.js, & so on—as well as your custom code.

Concatenation Counter-indicators

Concatenation isn't always the answer. If the code is both A) weighty and B) used only on a select few areas of your app, it may not be a good candidate to roll into your monolithic JavaScript file.

For example, a date-picker might be a few hundred lines and yet only used on one or two screens. In this case, you may choose not to have it in the main file for the entire app; it may make more sense to include

it on those areas only, or load it dynamically in the case of a more interactive application.

Concatenating on Deployment

Chapter 6

Loadtime: Inlining & Precaching

Like Inline Skating, but without the scabbing

Mama always told you to separate your concerns. Or was it that there'd be days like this?

Anyway.

Common wisdom in web development circles says: separate content (HTML) from presentation (CSS), and to keep both of those faaaaar away from dirty old function (e.g. JavaScript code).

And just like every other kind of common wisdom, there are exceptions.

Brace yourself, because we're about to tell you that sometimes it makes sense to smush your CSS and JavaScript into the very same file with your HTML.

Inlining Isn't Evil

Sometimes smushing your CSS and JavaScript into one file with your HTML—called inlining—is not merely okay, but actually really beneficial.

It's natural for this to feel wrong. Just know that it's right.

Proof: Google inlines. And their whole motto is "Don't Be Evil." So it

We don't mean squishing your CSS into attributes inside your HTML tags (or JavaScript, either).

We mean inlining the whole contents of files. In the appropriate places.

can't be evil. Ipso facto!

Why Inline

The point of inlining is three-fold:

- it reduces the network load (one file stream instead of several)
- your page will download more quickly, because the browser doesn't have to handle multiple file streams
- when cached, the cache's just one file, potentially improving performane

When to Inline

Based on these benefits, there are a couple scenarios where inlining makes great and perfect sense:

A simple page that absolutely must have top performance all the time
Example: Google's search page, which is, in fact, inlined out the wazoo. Or in the wazoo. Whatever.

A page that is visited only once in a session, like a portal homepage
Example: Any special page/area that has code/CSS that is not shared with any other part of your app (making caching the code or CSS separate a net zero benefit), like a login or thank-you-for-signing-up page.

How to Inline

Inlining can be tougher than it sounds. There are three major steps. Are you with me?

Good.

First, open your CSS and JavaScript files for the page you plan to inline. Second, open up the HTML source for that page.

Paste the CSS and JavaScript in (in their appropriate locations).

Save.

But seriously, if your to-be-inlined area is a big complex beast and you're more comfortable working with the files separated, you may want to consider a deploy hook that will compile them for you.

Remember that your JavaScript still needs to be positioned in the right locations in your HTML file; still, ideally, at the end.

Pre-caching: A Beautiful, Sneaky Trick

You know what caching is.

How about pre-caching?

I hear you: Is that like reading the user's mind and giving them the cached files before they even ask for them?

In a word... yes.

What's Pre-caching?

Pre-caching's a sneaky little low-tech trick that can definitely make your user's experience seem faster.

Here's the basic idea:

1. your customer loads up a page that they have to get through, like a sign-in or country selection page (reserving judgment on the wisdom of country selection pages)
2. while he's busy fiddling with the thing he needs to do, you're loading JavaScript and CSS files in the background—files that aren't even needed on this page!
3. once your customer's done with that page, the external stuff necessary for the next page will already be loaded in his cache

Why Pre-cache

According to Yahoo!'s research, at any given time, 20% of users will have a "no-cache" experience when visiting your web app.

Now, we tend to think this number might be a little skewed because of the Yahoo! web properties, but it's hard to tell.

If you assume those numbers are globally accurate, that means that 20%

of your visitors won't have any of your carefully configured-to-cache files cached.

It's like they're starting fresh. Every damn time!

So if you have to put a road block in their way anyway (like a sign-in page), why not take that opportunity to speed up their subsequent experience?

How to Pre-cache

The simplest way to pre-cache is to simply include the JavaScript or CSS files at the bottom of the roadblock page, just like you would on any other page.

Just be sure they're not going to conflict with anything!

Chapter 8

Loadtime: Under Compressure

JavaScript Compression, Not a David Bowie cover band

This section will be fun and confusing because of the terminology. Let's have a word definition war!

There are essentially three ways to squish down your JavaScript code files, divided into two camps: the types that modify your code, and the types that do not.

MODIFYING THE CODE:

1. **Packing.** The most popular, and most horrid; obfuscates code and requires `eval()` to unpack.
2. **Minifying.** Like packing, but without the horrid: instead of obfuscation, minifying is relatively sane, involving the removal of comments, white space & and the shortening of variable names, etc.

NON-MODIFYING THE CODE:

3. **Deflating.** This is a fancy word for "gzipping." This method doesn't change your code in any way, just gzips it right up, to be quickly decoded by the web browser.

NOTE! Lots of tools that call themselves minifiers actually pack, too. Watch out. Don't obfuscate your code.

The Problems with Packing

One of the most common questions the Prototype team receives is

How can I minify Prototype?

The answer is:

For the love of god, don't!

While packing and minifying are technically different approaches—packing goes just a step too far—when people say "minify," they usually mean the kind that compresses the code (obfuscation).

Packing has caught on in the JavaScript community, but it's an actively destructive method of reducing JavaScript file size.

You shouldn't "minify" your JavaScript with any of the tools that add obfuscation, because the client has to decompress it with JavaScript. This can be slow.

Let's try that again: This can be very, very slow.

And it can break your code!

Some of these suckers are not 100%-capable of following the JavaScript specification. That means they can introduce bugs. Bugs, caused by your performance tweaks! Not a good thing, people.

As an added bonus, debugging packed files is hellish.

If you really want to reduce the size of the content of your JavaScript files, we strongly recommend you stick to the well-behaved white-space removal, variable renaming type.

But, except in extreme circumstances, you may as well just stick to gzipping.

BIG HONKIN' EXCEPTION: the iPhone is hardcore about caching—for obvious reasons, it being a little handheld device and all. To persuade iPhones to cache your stuff, you need to keep each file under 25k. This is where minifying can come in handy. See http://yuiblog.com/blog/2008/02/06/iphone-cacheability/ for more info on this.

Minifying and packing are last-ditch efforts. You should try the other stuff (cleaning up your code, caching, getting a faster server, gzipping, etc.) before trying to minify the heck outta your files. That'd be premature optimization.

And we all know what that means.

Good Minification

I'll assume we've suitably scared you about minification vs packing, and all the fallout that may ensue if you confuse one for the other. But just in case... good minification does not obfuscate your code!

Now we're set on what good minification isn't, but how about what it is?

Definition: Good minification involves the removal of whitespace (tabs, spaces, linebreak) and comments, plus the shortening of variable names (inside functions only, so it'll still play nice with other code).

Good minification would take this function:

```
function upcase(string){
 // this is a pretty useless comment as
 // it is obvious what is going on
 return string.toUpperCase();
}
```

And transform it into:

```
function upcase(s){return s.toUpperCase()}
```

How to Minify (Properly)

We'd recommend the YUI Compressor, a Java-based command line tool from Yahoo!'s JavaScript gurus.

You can invoke it as follows:

```
java -jar path/to/yuicompressor-2.4.1.jar -v source.js -o source.
mini.js
```

Given our longer example above, this is the actual result that the YUI compressor worked out:

```
function upcase(a){return a.toUpperCase()};
```

Looks familiar, huh?

Download the YUI Compressor & check out the docs at http://developer.yahoo.com/yui/compressor/.

Hint: Use Verbose Mode

The `-v` option sets the Compressor to verbose mode. And in verbose mode, it'll chew you out if it finds other things in your code that can be optimized for compressing.

For example, the Compressor doesn't like multiple `var` statments inside one method, & it will tell you so. It'll also identify dangling variables that are declared but never used, and other such good-to-know things.

If we add some useless, badly written code to our example:

```
function upcase(string){
 var a = 1;
 var b = 2;
 return string.toUpperCase();
}
```

The Compressor will not let it pass without comment:

```
[WARNING] Try to use a single ‘var’ statement per scope.
```

```
(string){var a=1; ---> var  <--- b=2;returnstring.toUpperCase()

[WARNING] The symbol a is declared but is apparently never used.
This code can probably be written in a more compact way.
functionupcase(string){var  ---> a <--- =1;var b=2;returnstring

[WARNING] The symbol b is declared but is apparently never used.
This code can probably be written in a more compact way.
string){var a=1;var  ---> b <--- =2;returnstring.toUpperCase();
```

It's just like having a smart, well-meaning, but slightly passive aggressive friend looking over your shoulder, all the time!

But seriously, the Compressor gives good hints. It'd do you good to listen to it.

Even if it could use an adjustment in its bedside manner.

Gee! Gzipping is the Answer

Gzipping is the best solution for JavaScript file size, bar none:

- You can get a 1:4 reduction in size with gzip. That's from 4K to 1K, or 40K to 10K.
- Gzipping doesn't remove white space, or alter your variable or function names, making it easier to debug
- Gzipping is done by your web server on the way out, meaning you can configure it & forget it (after testing, of course)
- Gzipping offers a low performance hit compared to script obfuscation

Want the technical dirt on the gzip algorithm? Be careful what you ask for: http://tools.ietf.org/html/rfc1951

Don't forget that gzip is the same compression method used for GIFs and PNGs (and zip files, too, of course). Web browsers are already all over that. It's not some crazy new hippie web 2.0 thing.

Configuring Apache 2.x

It's easy to configure Apache 2.x to send the correct caching headers using the built-in `AddOutputFilterByType` extension (phew!).

Combine this with proper caching settings, and users will download your gzipped JavaScript, it'll get unzipped by the browser, and that unzipped source will get cached for as long as you need.

Here's our setup for freckle time tracking:

```
AddOutputFilterByType DEFLATE text/html text/plain text/xml
application/xml application/xhtml+xml text/javascript text/
css  application/x-javascript

ExpiresActive On
ExpiresByType image/gif “access plus 5 years”
ExpiresByType image/png “access plus 5 years”
ExpiresByType image/jpeg “access plus 5 years”
ExpiresByType text/javascript “access plus 5 years”
ExpiresByType application/x-javascript “access plus 5 years”
ExpiresByType text/css  “access plus 5 years”

Header unset ETag
FileETag None

Header add Cache-Control “public”
```

NOTE: If you're copying & pasting this into your config file, make sure you're not doubling up on the `ExpiresByType` stuff that we covered earlier! (And don't forget these code samples are all available in the ebook directory, as real text files.)

Google gzips So You Don't Have To

So you don't have access to your Apache's config files. What to do?

Weell... If you've got a lot of custom JavaScript to gzip up, you shoulder consider switching up your hosting situation.

There's only so much magic we can work for ya, you know.

But, I hear you saying, I don't have a lot of custom JavaScript. Just this honkin' big JavaScript framework! It's not even mine!

Well, good news, then! Google to the rescue!

Google offers a bunch of the most popular JavaScript libraries, hosted up gzipped and with good cache settings for your convenience.

Aside from not having to do the work yourself, this centralized hosting means a potentially download-free experience for your users.

For example, Joe visits Web App A, and it uses Google's hosted Prototype 1.6.0.3. Then Joe visits your Web App B, which also uses Google's hosted Prototype 1.6.0.3.

Joe's browser only downloads the library once, the first time. Whoopee!

Offered Libraries

As of time of writing, the libraries included are:

- jQuery
- jQuery UI
- Prototype
- script.aculo.us
- MooTools
- Dojo
- SWFObjectNew!
- Yahoo! User Interface Library (YUI)New!

Nice!

Go to this URL for a full list & detailed instructions:
http://code.google.com/apis/ajaxlibs/.

How it Works

To use Google's hosted libraries, include their JSON API and use the `google.load()` function:

```
<script src="http://www.google.com/jsapi"></script>
<script>
  // Load Prototype
  google.load("prototype", "1.6.0.3");
  // Load jQuery
  google.load("jquery", "1");
</script>
```

And so on.

But Not All is Perfect in Googleland

There are some reasons you shouldn't use Google's hosted code:

- It may incur an additional DNS lookup (boohoo! Not typically a big deal).
- It's remote. There are times when Google may not be reachable by your user, or the Google servers with the libraries may be slow. This sounds unlikely, but we've experienced this.
- You can't have edge libraries. Google only hosts stable versions, not experimental releases or the most current from the source repository.
- You can't hack the source (obviously).

Generally speaking, it makes sense to use your own libraries if you can get the server properly configured for caching, and if you are deploying an app on a long-term basis.

Translation: if you're charging money for your app's performance (or your reputation's riding on it), we recommend you suck it up and get better hosting.

But Google's hosted libraries are perfect for quick projects, and non-mission-critical apps when you've got inexpensive hosting without access to web server configs.

Chapter 9

Loadtime: Cover Your Assets

an EASY way to increase Zippiness

Streams, as in data flowing and burbling over picturesque little rocks, and simultaneous, as in more than one at a time.

Browsers will only open so many connections with one host (e.g. `www.jsrocks.com`) at once. We call these simultaneous streams.

This built-in limit is usually pretty low, about 2 - 5 at once. An unfortunate throwback to the days of bleeps and bloops.

If you've got 20 files to download, it can take a while. And 20's not a very high number when it comes to designing beautiful, dynamic web applications.

The first line of defense against this particular problem is to religiously reduce the number of files you have to download for the page to render: squeeze 'em together, and optimize, to get that number as you can manage (as we've talked about earlier).

In content distribution terms, an asset is any file or resource your HTML document includes, such as CSS files, JavaScript files, images, movies, Flash files, you name it.

The second line of defense, of course, is cheating.

Using Multiple Asset Hosts

Allow us to repeat: these limits are per host. Like we said, a host is a domain name like `www.jsrocks.com`. To underscore: `www1.jsrocks.com` and `www2.jsrocks.com` are not the same host.

You can get around the simultaneous stream limit by using this multiple asset hosts technique.

Instead of:

```
<img src="/images/x.jpg" />
<img src="/images/y.jpg" />
<img src="/images/z.jpg" />
```

Theoretically speaking, if you took this approach instead, your pages could download up to 3x faster (bandwidth allowing):

```
<img src="http://asset1.yourapp.com/images/x.jpg" />
<img src="http://asset2.yourapp.com/images/y.jpg" />
<img src="http://asset3.yourapp.com/images/z.jpg" />
```

Now, you don't actually need 3 different servers to do this. You can configure simple DNS pointers to the same old server, and then you can configure your server software to respond to those hostnames.

Bonus: Some web development frameworks come with support for multiple asset servers out-of-the-box (e.g., our friend Ruby on Rails).

Don't Randomize!

Watch out that you don't just randomly assign hostnames. If you reference the same asset with different hostnames, the browser will not know the assets are identical.

In this case, for example, your user will have to download the image twice, rather than pulling it from cache the second time:

```
<img src="http://asset1.yourapp.com/images/A.jpg" />
<img src="http://asset2.yourapp.com/images/A.jpg" />
```

If your web development framework doesn't internally manage multiple asset servers, then you're going to have to ensure consistency yourself.

The best way is to build a checksum from the filename of the asset and calculate the modulo for the number of asset server hostnames you use. Keep it around to verify.

If that sounded like greek to you, that's probably a good thing.

This technique is really best for small, hand-optimized landing pages, or for bigger web apps that use a framework that has solid support for this.

For Best Results, 4's the Magic Number

You'll get the best results with about 4 asset servers.

Use more than that, and you may start going backwards.

There are three reasons why:

- the extra DNS lookups will defray the benefits of the faster loading cycle
- you'll also have to spawn network connections than necessary
- and almost all servers are already configured to use the HTTP 1.1 keep-alive directive, which means your server will keep sending files down the pipe as long as the connection's open

Of course, if you follow the rest of our advice, this doesn't matter for your JavaScript files. You should only have one of those!

Chapter 10

Loadtime: You Need an Upgrade

C'mon, don't you watch MTV Racks?

If you've already got a high-octane hosting environment with a slew of slices and performance monitoring up to here, you can skip this section. Carry on!

There comes a point in every app's life when it simply outgrows its humble shared hosting origins.

It's not just page weight and open streams and DOM churning that can slow your app down.

Out of Sight, Not Out of Mind

What's happening on your back-end comes before all those things, and could potentially also be a source of slowness. So too with the pipe between your back-end and your customer's front-end.

You know what I mean.

Before doing anything drastic with your JavaScript code or reworking your app in any way, be sure that you're getting good performance out of your server to begin with.

If your web app views are taking a really long time to compose because of complicated queries, slow libraries, or simply very high server load (or very low bandwidth), there's no amount of JavaScript tuning you can do to offset that.

Catching Back-end Issues

Here's how to spot a back-end issue:

- the longest wait is from the browser request to when files start downloading
- pages get to the "mostly complete" phase but the browser keeps waiting on
- the final bits (watch out for remote JavaScript)
- pages render fast once they're downloaded (bandwidth problem)
- the Net tab of Firebug doesn't indicate that download speed is the issue (in the case of a back-end performance problem, not a bandwidth problem)

Profiling Back-end Issues

For Rails apps, there are two great services that can help you identify where the slowdowns occur in your app:

Rails

FiveRuns http://www.fiveruns.com/

New Relic RPM http://newrelic.com/

And some how-to guides on other methods:

How to Profile Your Rails Application

http://cfis.savagexi.com/2007/07/10/how-to-profile-your-rails-application

Profiling Rails end-to-end

http://www.underpantsgnome.com/2006/9/9/profiling-rails-end-to-end

Benchmarking and Profiling Rails

http://guides.rubyonrails.org/benchmarking_and_profiling.html

Django

Profiling Django Applications

http://perro.si/profiling-django-applications

Quick profiling your Django web site with debug_toolbar

http://www.davidcramer.net/code/312/quick-profiling-your-django-website-with-debug_toolbar.html

PHP

Improving Performance by Profiling PHP Applications

http://www.onlamp.com/pub/a/php/2002/02/28/profilingphp.html

PHP Performance Profiling

http://www.linuxjournal.com/article/7213

XDebug

http://xdebug.org/

Chapter 11

Loadtime: Reduce Complexity

'Tis a gift to be simple, especially for Page Rendering

A complicated DOM is an unfortunate beast. Hard to read, hard to parse, slow to render... and it'll slow your JavaScript down, too.

When it comes to the DOM—that is, the structure of a document's HTML and the API that browsers supply for manipulating it—simpler is better. Unlike that foregoing sentence.

Simpler means fewer DOM nodes in the DOM tree, which means faster sites.

Simplifying your DOM can really affect the runtime performance of your app, too, but it also affects your loadtime performance. More complex DOMs take up more bytes, and unnecessary nodes (especially nested divs) can add a lot of rendering time to your user's experience.

There are two basic areas where you can optimize:

- General complexity.
- Browser-specific complexity.

General Complexity

Here's an obvious (and simplified) example of an unnecessarily complex set of nodes:

```
<p><b>This is bold</b></p>
```

This example has two elements (p and b) and three text nodes (whitespace and the text in the b tag).

It could be simplified as:

```
<p class="important">This is bold</p>
```

Yep, that's right kids—with that one stroke, you've axed half the elements and 2/3rds of the text nodes. But, as we said, that is a simplified example.

In reality, sure, you've got unnecessary b and em tags in your app's HTML structure. But what you really need to be watchful for is nested divs and spans.

Reduce CSS complexity. Simplifying your CSS can make your rendering zippier, and significantly improve your animation speed. Get yourself down to as few rules as possible, and avoid having many rules that overwrite other rules (for example, through excessive use of !important). It's a good practice all around!

Watch out for Ajax

Another area to step carefully: Ajax that dynamically updates visible nodes.

Yeah, we love Ajax, too, but browsers vary wildly in the amount of nodes they'll re-render when you insert or update content. That means that if you insert into node A, other nodes around node A may also get re-rendered. Which can be sloooow if you've got a lot of nodes.

This isn't normally an issue. But it depends!

If your page structure is complex, if you do a lot of Ajax updates, and especially if your users aren't visiting on top-notch hardware, things can go south very quickly.

Or slowly. Depends on how you look at it.

Browser-specific Complexity

Lots of people—purists—will tell you that sniffing browsers is a terrible idea. But in certain situations, terrible ideas can actually become very good ideas, indeed.

There are times when only one type of browser (*coughIEcough*) requires additional complexity, when the others are chugging along just fine.

Case in point: rounded corners.

With a number of modern browsers, you can do rounded corners with the border-radius CSS property and just leave it at that. For the others,

well, you've got to nest some divs or apply other hacks.

Why not send those hacks—and that extra complexity—just to the browsers that need 'em? You can keep your pretty non-stabby round corners, and also reduce complexity for every other browser.

In modern web development environments (like view helpers in Ruby on Rails), it's easy to customize the actual HTML output based on your visitor's user agent.

It's not such a bad idea after all.

When one really needs to skin a cat, one can only be so squeamish about the tools one uses to do the skinning. '

Chapter 12

Loadtime: JavaScript, On-Demand!

Right where you want it, when you want it.

If you've tried everything else, on-demand JavaScript may be an option for you.

Sometimes you only rarely have need of certain sets of code. But these bits of code might cause your initial page loading time to creep up towards the unacceptable zone.

Why not load JavaScript on demand in this case?

That feedback form that's just used once every 1000 pageviews – spare your users the cruft and load it on demand.

Inserting JavaScript on Demand

It works by dynamically inserting a script tag into the DOM.

```
// create a script element
var script = document.createElement('script');
script.type = 'text/javascript';
script.src = 'http://www.yoursite.com/path/to/ondemand.js';
```

Once you've prepared that, you simply insert it into the DOM (in the

proper place):

```
// insert into the dom (at the end of the body element)
document.getElementsByTagName('body')[0].appendChild(script);
```

That's all. Your JavaScript will be loaded. Keep in mind that there won't be any events called (the DOM is already loaded), so just initialize things directly at the end of your JavaScript file.

Caveat: mind that the protocol used to load in the JavaScript file should be the same as for the containing page, so if you're on a secure site and use SSL, use https:// to load in the file.

If you're using Google Analytics, you might have seen the following line, that you might want to 'borrow' and use for your own evil purposes:

```
var gaJsHost = (("https:" == document.location.protocol) ?
"https://ssl." : "http://www.");
```

Chapter 13

Welcome to Runtime

Enter at Your Own Risk! Watch out for Falling Cows!

Tuning your JavaScript code should only come after you've done the other kinds of optimizations we recommend. Don't be overhasty. It's fun, we get it, but you can get yourself into trouble if you optimize too much.

That said, most JavaScript code offers a lot of room for optimization. And there are tons of good habits you can learn to use naturally, without compromising your code's future readability.

Just be moderate, okay?

Also: the runtime sections are still heavily under construction.

On Premature Optimization

The most important thing to always remember is DO NOT OPTIMIZE PREMATURELY.

Optimizing comes after you've implemented the functionality.

This doesn't mean there aren't certain 'elegant' ways of writing code that will likely lead to code that is both optimized for speed and size—but the techniques for 'hardcore' optimizing we describe here should only be

The worst offender in my book? eBay. They have millions of people literally fighting to give them money, but the page takes 2-3 minutes to download and stop churning. Although GoDaddy's pretty high up there, too.

used when necessary.

The best code is code you're comfortable with.

This means, you've a thorough understanding of what it does, even when you come back to it after three months. And it means that you can look at it without getting a headache, because it looks ugly.

For the most part the advanced optimization techniques apply to loops or nested constructs – when code is executed repeatedly. This is where performance can bite you most. We'll show you how to make things run smoother.

JavaScript Parsing Speed

JavaScript parsing and interpretation is pretty quick in modern browsers.

Tests reveal that function/object definitions with little execution of actual code (as in case with JavaScript frameworks, like Prototype or jQuery) are quickly parsed, with a performance of a million lines of code per second or better.

Given that anything above 50,000 lines of JavaScript in a REALLY BIG web application is total overkill, the parsing speed isn't an issue.

It gets more interesting with execution speed, with average JavaScript expressions (think one line of code) are executed with a speed of 50kLOCs to 250kLOCs. Not too shabby.

So use your energy on those pesky loops with lots of expressions executed, and don't worry if the library you're using is 2k LOC or 5k LOC (as long as you observe the rules for serving it quickly!)

This also means that there is not much to be gained from JavaScript minifying in terms of optimizing the pure JavaScript parsing time.

Our tests show that your scripts might be parsed 10% to 20% or so faster, but at as stated above this will not be noticeable, unless you've hundreds of thousands of lines of code to parse.

You might still want to consider using minification though, as it can save quite a lot of bandwidth/transmission time. If your want to try out one, the YUI Compressor (http://developer.yahoo.com/yui/compressor/) has proven to produce the best optimized output (with little possibility of breaking your code), plus it can optimize the size of your CSS, too.

DO avoid JavaScript obfuscators though (e.g. http://dean.edwards.name/packer/ with Base62 encode ON). These rely on `eval()` statements that might considerably slow down the loading performance of your JavaScripts.

Execution Speed vs Code Size

A dilemma a old as programming is the struggle of code size versus execution speed – you can't have both things optimal at the same time.

For our purposes we can gladly make up a set of simple rules to achieve a good compromise:

1. Avoid performance problems. Apply the rules for better percepted load time, optimize the serving environment and don't even start out with unnecessary DOM cruft and superflous JavaScript code.
2. Only optimize for execution speed where it actually makes sense. 'Sense' includes: 'feelable' slow-downs, jerky animations, laggy response to keystrokes, non-immediate onset of responses to clicks, hourglasses, beachballs of death. That sort of thing. In most cases, identifying the perpetrators should be easy enough (sadly, the solutions won't be that obvious most of the time).
3. Optimize for code size elsewhere (but don't to it prematurely). If you're doing applications, most of your JavaScripts will gladly reside in browser caches and needs not to be concerned about code size too much. For landing pages, public web sites and mobile applications the importance is higher.

Chapter 14

Runtime: DOM, DOM DOM DOMMM

Cue Dramatic Mozart Music, for the Document Object Model

The JavaScript we write is sullied by the DOM, that unganglest of interfaces between the HTML document tree and JavaScript.

We love making highly interactive web apps, but there's little love lost between us & the Document Object Model.

Also, it's slow. Which is how we come back on point.

Interacting with the DOM is one of the slowest things you can do with JavaScript. (Same thing goes for other XML-type data structures, like, well, XML.)

But there's a nice bag of tricks you can learn to speed things up.

Looking for Elements in All the Wrong Places

We're all little web monkeys, constantly trying to plucking the juiciest DOM element—or elements—out of the tree.

Making a poor choice with your node selection is super easy, and can cost you a lot, performance wise. Gladly, these mistakes are also super easy to fix.

In order of zippiness, here are the ways you can pick up the right node or nodes:

- selecting by ID is the fastest
- selecting by tagName is still quite fast—but chain it off the correct parent container (not the whole document object)
- selecting with CSS selectors is slow, regardless of which library you use

BETA ALERT! We're testing the various libraries and browsers. Expect hard data soon.

Regarding CSS selectors specifically, there is a specification but most browsers don't follow it. Yay, specifications! And the big libraries vary wildly in their performance on this one.

Other techniques to add to your toolbox:

- Find yourself searching repetitively for the same criteria? Create a cache instead, & keep it up-to-date

Now, for those of you who'd like more detail:

By ID: The Absolute Fastest Way

Select a DOM node by ID is the bar-none fastest way to grab it:

```
var node = document.getElementById('banana')
```

By the childNodes Collection: Fast but Sometimes Unreliable

When you're looking for a particular node or nodes inside a parent node (phew), you can use the parent node's `childNodes` collection just like an array. You can grab all of it or cycle through it:

```
var children = node.childNodes;
for(var i=0; i < children.length; i++) {
   // do stuff with all child nodes
}
```

You can select specific elements from the childNodes using array accessors, and even chain them infinitely:

```
node.childNodes[0].childNodes[1].childNodes[2]
```

Be aware, though, that a node isn't just the stuff in an HTML tag. If you write your code this way, your results may be unexpected, because some browsers treat whitespace as nodes—and they differ, too.

So the element you get as node.`childNodes[0]` in IE may not be the same as what you get in Firefox.

By Tag Type: Fast and Reliable

You can work around the whitespace / text node issues across browsers

by using the `getElementsByTagName()` approach instead. But, always start from the narrowest possible starting point: if you know the `div` elements you want are in a certain parent element, go for that sucker first.


```
var nested_nodes = node.getElementsByTagName('div')
```

Starting from the right point, narrowing it down as much as possible first, is a good way to speed things up.

CSS Selectors: Slowest Possible Choice

You like using CSS selectors. I like using CSS selectors.

But the plain fact of the matter is, if you've got performance problems, using a framework's CSS selection engine is a terrible idea.

Event Bubbling

When you're working with DOM node events, you don't need to assign an event—like a mouseover—to every single possible DOM element that might be under the mouse to trigger it.

Instead, you assign the event to a container element, and nodes inside it do or do not "participate" in the event as it travels up the chain.

This is called event bubbling.

Event bubbling has many advantages, such as not having to deal with sudden memory leaks because of the sheer amount of event handlers.

To best use this feature, be sure to use a JavaScript framework or toolkit that provides a cross-browser API for events (otherwise it's pretty messy).

Say you want to check for clicks on list items. Instead of registering an `onclick` event on each list item, just do one event on the UL element, and use the magic of event bubbling:

```
<ul id=”list”>
  <li>hello</li>
  <li>there</li>
</ul>

<script>
var list = document.getElementById(‘list’);
list.onclick = function(event){
  event = event || window.event;
  alert(‘You clicked on: ‘+(event.target||event.srcElement).
innerHTML);
}
</script>
```

As you can see, it's getting pretty complicated because of the different event models in IE and the rest of the browser world, and this example

even makes some shortcuts that you should avoid (like using the `onclick` property directly).

This technique has a couple of advantages with Ajax applications, too– as you update element contents, for example add new elements to the list, these elements will automatically inherit the ability to deal with events, without the need to call on extra initializing through JavaScript:

```
<script>
  list.innerHTML += '<li>JavaScript rocks!</li>';
</script>
```

Magically, clicking does the right thing! Rejoice!

So where should such catch-all event handlers go?

We say use common sense and don't try to make just one page-wide `onclick` handler—instead opt for logical, functional groups.

And look into those frameworks, which can handle a lot of the nasty background idiosyncrasies for you.

All in all using event bubbling extensively gives you leaner HTML, less traffic and a lot less of a mess.

The InnerHTML Accessor

Using `innerHTML` to update elements contents is almost always faster than using the DOM API. If you don't need to deal with events, try to use innerHTML where possible:

```
var element = document.getElementById('blurb');
benchmark(function(){
  element.innerHTML = 'hello <b>world</b>: <i>here i am</i>';
}, 10000, 'innerHTML');

benchmark(function(){
  var node1 = document.createElement('b');
  node1.appendChild(document.createTextNode('world'));
  var node2 = document.createElement('i');
  node2.appendChild(document.createTextNode('here i am'));
  while(element.childNodes.length) element.removeChild(element.
childNodes[0]);
  element.appendChild(document.createTextNode('hello '));
  element.appendChild(node1);
  element.appendChild(document.createTextNode(': '));
  element.appendChild(node2);
}, 10000, 'DOM API');
```

All browsers are about 2.5x faster with the `innerHTML` method than with `appendChild` and `removeChild`.

DOM Complexity

Reduce DOM complexity as much as possible. The fewer elements/nodes there are on your page, the faster everything will be. (We covered this in loadtime, too!)

Checking an Element's Contents

When you want to check if an element's empty in your code, you probably write something that looks like this:

Slow Code Example

```
if (element.innerHTML == '') { // do stuff }
```

This is familiar, natural, and totally slow!

While we said that the innerHTML accessor is faster for replacing DOM node contents, it is much slower than other methods when it comes to comparing them.

In this case, the `innerHTML` must be generated from current DOM tree on demand, to be compared to the string.

Faster Option

```
if (element.childNodes.length == 0) { // do stuff }
```

With this example, you're checking an array length, with no string comparison. It's significantly faster.

Setting Element's Styles

Use the `cssText` property instead of setting individual styles in multiple statements to avoid multiple reflowing cycles for your page.

When using the `cssText` property, the browser only performs one layout reflow (you set all the properties at once).

If you change the individual style properties, the browser might be forced to do more than one reflow.

This is especially important if you have a large & complex DOM structure.

Here's a benchmarking example we used to test (try it yourself!):

```
benchmark(function(){
  element.style.cssText = 'font-size:'+Math.
random()*50+'px;text-indent:'+Math.random()*50+'px';
}, 10000, 'cssText');

benchmark(function(){
  element.style.fontSize = Math.random()*50+'px';
  element.style.textIndent = Math.random()*50+'px';
}, 10000, 'direct styles');
```

Even with the simplest DOM possible, just one `div` node, the `cssText` approach is 25% faster in Safari 3, and 50% faster in Firefox 3.

Keep it Simple

Let's say you're doing a splash page where you need to run some JavaScript if there's some big area clicked.

You could go ahead and include a JavaScript framework, and use its support for events to do some event handling in a method that you put in an external JavaScript file that you call when the DOM is completely loaded.

OR... you could keep it simple and just add a simple `onclick="some();javascript();stuff()"` attribute on the `div` you want clickable.

Does that idea make you feel dirty?

Well, get used to it, because performance tuning can be a dirty job. Think about those pit crew mechanics crawling around on the asphalt and rolling in motor oil. It just comes with the job description.

One of the most important skills you can develop for performance optimization: seeing the big picture. Try to step back sometimes and look at it!

And, by the way, there's absolutely no reason not to use an onclick attribute, even if the web standards zealots tell you otherwise. Again, NO REASON. That is, NO REASON*.

There, we said it.

* Please don't try lecturing us on this by mentioning 'accessibility'. The a-word has been used time and again to make perfectly simple things complex and sluggish. There's not silver bullet to accessibility on the web, except designing and engineering for multiple target audiences. Which means building multiple versions of the front-end, tailored to each group.

Chapter 15

Runtime: Pure, Clear JavaScript

Time for the Nerdiest Optimizations of All

A word to the wise: there are tricks in here for optimizing that should only be used when you really need 'em. When in doubt, benchmark.

This will get filled out with lots of helpful background text, but for the time being, we're just gonna hit you up with examples.

BETA ALERT!

We're adding a ton more content here. If you don't understand the context of everything, you will with the next few versions.

Let us know if you find anything confusing!

The with() Statement

Rarely used, but it can kill performance if used in loops.

```
benchmark(function(){
  var obj = { x: true }, i = 10000;
  with(obj){ while(i--){ !!x; } }
}, 100, 'with');

benchmark(function(){
  var obj = { x: true }, i = 10000;
  while(i--){ !!obj.x; }
}, 100, 'without with');
```

On most browsers, the version without the `with()` statement is 4 to 10 times faster.

Loops: Best Practices

For accessing (nested) properties of objects save a reference in a local variable and use a while loop with the decrement operator for optimal performance and code size.

You'll see the while-loop technique used quite often in the code samples in this book.

Not Good

```
for(var i=0;i<100;i++) {
  something(my.complicated.object.reference, i);
}
```

Better

```
var x = my.complicated.object.reference;

for(var i=0;i<100;i++) {
  // do something with x
}
```

Best

```
var i = 100, x = my.complicated.object.reference;
while(i--){ // do something with x }
```

Method Calls

Method calls are expensive. Inline methods to loops.

Consider:

```
function square(n){ return n*n };
var i=10000, sum = 0;
while(i--) sum += square(i);
```

vs.

```
var i=10000, sum = 0;
while(i--) sum += i*i;
// inline the code from the function into the loop
```

The latter is 2-4 times faster.

Variable Caches

For often-called methods, cache aggressively. Here's one way to do it:

```
var function = (function(){
     var x = someExpensiveOperation();

     return function(){
          return x;
     }
})();
```

All this boils down to: '`someExpensiveOperation`' is called only once, and

on all calls to `f()` the cached result will be returned. This is a great way to introduce caching without having to change the rest of your code:


```
function takesALongTimeAndDoesNothing(){
 var i = 100000; while(i--) i*i; return 1;
}
```

Versus the alternative:

```
// cached implementation, mind the __ on the beginning of the
function
function __takesALongTimeAndDoesNothing(){
 var i = 100000; while(i--) i*i; return 1;
}

// and here the caching happens
var takesALongTimeAndDoesNothing = (function(){
   var result;
   return function(){
     result = result || __takesALongTimeAndDoesNothing();
     return result;
   }
 })();
```

In both cases the result of `takesALongTimeAndDoesNothing()` is `1`, but if `takesALongTimeAndDoesNothing()` is called multiple times, in the latter implementation the calculation has to be done only once (on the first call).

For this example, the cached implementation is a tiny, tiny, tiny bit slower when first called, but 10,000x faster starting from the second invocation.

If() vs Switch()

```
benchmark(function(){
  var a = 5;
  switch(a){
    case 0: {} break;
    case 1: {} break;
    case 2: {} break;
    case 3: {} break;
    case 4: {} break;
  }
}, 100000, 'switch statement');
```

Tested against:

```
benchmark(function(){
  var a = 5;
  if(a==0){};
  if(a==1){};
  if(a==2){};
  if(a==3){};
  if(a==4){};
}, 100000, 'if statement');
```

Results

Firefox 3: `switch()` wins: .015s vs. .027s for `if()`.
Safari 3: `if()` wins: .035s vs. .039s for `switch()`
IE 7: `if()` wins: .375s vs. .453s for `switch()`

String Concatenation

Can vary depending on the browser and might require branches. For example, if you do a lot of concatenation it might be advisable to have one version for Safari 3 and Firefox 3 that uses an array and joins it, and the other version for Firefox 2 and IE, that uses plain string concatenation.

BE AWARE: JAVASCRIPT INTERPRETERS VARY WILDLY.

You can always redefine functions in JavaScript, make use of it!

String concatenation benchmark 1:

```
benchmark(function(){
  var x = 'a'+'b'+'c'+'d';
}, 100000, 'direct concatenation');
benchmark(function(){
  var x = 'a'; x += 'b'; x += 'c'; x += 'd';
}, 100000, 'individual statements');
benchmark(function(){
  var x = ['a','b','c','d'].join('');
}, 100000, 'array join');
```

String concatenation benchmark 1:

```
benchmark(function(){
  var x = '', i = 1000;
  while(i--){ x += 'blech'; };
}, 1000, 'individual statements (1000 loop)');
benchmark(function(){
  var x = [], i = 1000;
  while(i--){ x.push('blech'); };
  x = x.join('');
}, 1000, 'array join (1000 loop)');
```

String Concatenation Results

Firefox 2:
direct concatenation: 0.176s
individual statements: 0.438s
array join: 1.850s

individual statements (1000 loop): 2.179s
array join (1000 loop): 4.193s

Firefox 3:
direct concatenation: 0.011s
individual statements: 0.080s
array join: 0.446s
individual statements (1000 loop): 0.679s
array join (1000 loop): 0.503s

Safari 3:
direct concatenation: 0.229s
individual statements: 0.232s
array join: 0.244s

individual statements (1000 loop): 0.838s
array join (1000 loop): 0.719s

IE 7:
direct concatenation: 0.656s
individual statements: 0.609s
array join: 1.078s

individual statements (1000 loop): 3.047s
array join (1000 loop): 4.187s

Regular Expressions

Here's a code example that tries to match various regular expressions in a huge string (the initialization for the string is a nice way to generate some test data, initialize an array of a certain size, and use join to generate the filling):

```
var hugeString = (new Array(1024*1024)).join('X'); // 1 MiB
benchmark(function(){
  hugeString.match(/ABC/);
  hugeString.match(/ABC$/);
  hugeString.match(/[1-3]$/);
}, 100, 'regexp matching');
```

The various JavaScript engines vary widely on speed—we have a surprise winner here!

Seems that IE's regular expression library is superfast: IE7 1.188s, Safari 3: 2.755s, Firefox 3: 3.2s. Firefox 2 around 7 seconds for this.

Regular expressions can be very fast, and you can do pretty advanced things with them. On the negative side, they tend to make your code unreadable (to the untrained eye– and for more complex regular expressions to all eyes).

Proceed with caution!

Reduce Namespaced Calls

Reducing "namespaced" function calls can speed things up.

Not Good

```
var x = {
  a: function(){ x.b() },
  b: function(){ }
};

x.a();
```

Better

```
(function (){
  function a(){ b() }
  function b() { }
  a();
})();
```

This has two benefits: 1) the code likely runs faster (not by a lot, though) and more importantly, 2) code size is reduced.

By refactoring large JavaScript programs you can save quite a lot by this style of programming.

Note that if you want to do OOP-style programming, you might want to avoid this style of closures, because you can't overwrite or copy methods later, so YMMV with this technique.

Write Simpler Code

It may be tempting to write "well-architected" code like this example:

```
function Rock(){ this.weight = 'a lot'; }
Rock.prototype.getWeight = function(){ return this.weight; }

var stone = new Rock();
stone.getWeight();
```

But it's much more efficient to use stone.weight directly instead:

```
stone.weight;
```

As this benchmark proves:

```
>>> benchmark(function(){ stone.getWeight() }, 100000)
benchmark: 0.372s

>>> benchmark(function(){ stone.weight }, 100000)
benchmark: 0.294s
```

Accessing Object Properties

There are two ways of accessing the properties of an object:

`object.property`
This method is faster than the second method (up to 25% or so, depending on the browser) – but this only matters if you do lots and lots of accesses to the same property.

`object[propertyNameString]`
Slower (but of course you can use dynamic strings!).

Use if necessary, preferring `object.property`.

Note that it doesn't make much of a difference if you use string literal (`object['propertyName']`) or a pre-assigned string variable (`var propertyName = 'propertyName'; object[propertyName]`).

The latter can actually be worse because of the extra lookup.

Testing if a Property Exists

```
if('propertyName' in object){ ... }
```

vs.

```
if(object.propertyName){ ... }
```

The latter is faster, but it doesn't check if a property exists or not, it checks if the property evaluates to true:

```
>>> object = { p: 1 }
Object p=1
>>> !!object.p
true
>>> 'p' in object
true
>>> object.p = false
false
>>> !!object.p
false
>>> 'p' in object
true
>>> object.p = undefined
>>> !!object.p
false
>>> 'p' in object
true
```

So, mind the differences. A `'propertyName' in object` expression normally performs pretty well.

Impact of Try/Catch

Especially with the most modern browsers, like Google Chrome and pre-releases of Safari, that have just-in-time compilers for JavaScript, `try/catch` blocks can be in the way of great performance.

Make sure to use `try/catch` only where absolutely necessary (that is, where you actually expect exceptions to happen).

Low-Hanging Fruit Grab Bag

Some low-hanging performance fruit

Precompiled regular expressions

```
var REGEXP = /abc/;
function method1(string){
  return string.match(REGEXP);
}

function method2(string){
  return string.match(/abc/);
}
```

`method1()`, using the 'pre-compiled' variable, is around 20% or so faster.

Use object and array literals

```
function method1(string){
  var a = [], o = {};
}

function method2(string){
  var a = new Array(), o = new Object();
}
```

`method1()`, using the array and object literals, is faster (and shorter to boot with).

Global variables vs. closures

```
var testVar = 100;
function method1(n){
  return n * testVar;
}

var method2 = (function(){
  var testInt = 100;
  return function(n){
    return n * testInt;
  }
})();
```

There's no winner here, both implementations perform more or less exactly the same. This means closures should be preferred over global variables, especially in larger scripts (to prevent cluttering the global

'namespace').

Reuse functions if possible

```
var emptyFunction = function(){};
function method1(){
  return emptyFunction();
}

function method2(){
  return (function(){})();
}
```

The former implementation, that 'caches' the function, is much faster than the latter (up to 50 times or so, depending on how much optimization the JavaScript runtime does).

Concise Code Techniques

Space-saver techniques.

Multiple var declarations

```
var a=1, b=2, c=3;
```

Function parameters as placeholders

```
function f(a,b,c){
 a=1; b=2; c=3;
}

f();
```

(tends to make code a bit unreadble)

Ternary conditional expressions

Instead of:

```
if(a==1) b=1; else b=2;
```

use:

```
b = a==1 ? 1 : 2;
```

Use return instead of an if

```
function f(a){
 if(a==1){
   // do something
 }
}
```

becomes

```
function f(a){
 if(a!=1) return;
 // do something
}
```

Improve perceived performance with long-running calculations

Sometimes you just can't help it and have to process lots of data on the client—which can mean visible temporay 'lock-ups' of your web app: short periods of time where the browser doesn't act or react.

This is because JavaScript is blocking and the main browser event loop (the one that handles mouse clicks, the browser menu bar, etc) won't do anything until after the JavaScript is executed.

This behaviour is by design. It could be argued that this design isn't a good one, but this design decision was made a decade ago in an comittee far, far away. It's just the way things are for now (several new JavaScript engines try to change that by introducing stuff like threads).

What you can do—that works on all browsers—is break up long operations into chunks and defer the executing of each chunk, so the browser can keep up with user input and at least appear as if everything is allright.

This is rather like multi-tasking on old-school operating systems, think Windows 3.0 or Mac OS 9. No preemptive multitasking here, move along.

You can do this with the `setTimeout()` function.

Here's an example where processing a long array of data is slow and how you can make the perceived performace better with execution chunking (it will still take the same processing time, mind you, but it will seem faster!).

(Note that normally, if you've this situation more than once, it's probably a good idea to make a reusable function out of this, plus it would also be much more trivial to implement if you're using a JavaScript framework that has solid support for array/enumerable handling.)

```
// what we want to do with each array item
function doSomethingSlowWithItem(item){
 // do stuffs
 console.log(item); // just for the demo
}

// original function
function process(data){
 var i = data.length;
 while(i--) doSomethingSlowWithItem(data[i]);
}
```

```
// chunked execution
function processChunked(data){
 var index = data.length,
   CHUNK_SIZE = 5;

 function processNextPart(){
   var i = index,
     target = index-CHUNK_SIZE < 0 ? 0 : index-CHUNK_SIZE;
   while(i--,i>=target) doSomethingSlowWithItem(data[i]);
   index = target;
   if(index>0) setTimeout(processNextPart, 10);
 }

 processNextPart();
}

var data = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16];
process(data);
processChunked(data);
```

Operator Tricks and Hacks

(Did we tell you this was a grab bag at the moment? We weren't kidding.)

To == or to ===, that is the question

The strict equality operator (`===`) has a slight edge for most cases, so use it when you can.

Shorthand operators: use them

Using shorthand operators like `+=` and `/=` can help fight code cruft, add a tiny bit of performance and make you feel like a JavaScript god when the other programmers are starting to congratulate you on your mad JavaScript skills.

```
a += 5; // increase a by 5
```

Of course, where applicable, use the shortcuts for increment and decrement:

```
if(++a){ ... }
// increase a by one and enter the block if the new value
evaluates to true
```

There's also the comma ',' operator (the comma operator evaluates both of its operands and returns the value of the second operand) with which you do further tricks:

```
 if(a++,b==5){ ... }
// always increments a, but only goes into the block if b equals
5
```

The forgotten |= shorthand operator

This can be used to apply default values to function parameters:

```
 function someFunction(n){
   n |= 1; // same as n = n || 1;
 }
```

This method is the fastest and shortest way to provide default values.

Chapter 16

iPhone Tips & Tricks

Is that a web browser in your pocket, or are you just... No? OK.

Wish I had a witty intro here for ya, but we're too busy working on the real content to get funny. So sorry!

JavaScript on the iPhone is slow.

Our tests show that you should think **about 40x slower** than a current desktop computer. That's forty times. A lot that seems swift and agile on your shiny new iMac is really going to hurt people on the run.

To make it worse, DOM rendering is also slow. Of course it is, this is hardware optimized for miuscule size and power consumption. Far from the super-fast 'like on a real computer' experience the ads suggest.

Here's some rules you can apply to iPhone JavaScript development:

Separate Versions Required

If possible, make a special iPhone version of your app or site.

Shrinky Dink, Under 25K

Keep the size (before gzip) of external JavaScript files under 25k, to enable proper caching*. You can split bigger files up in chunks to enable this. Also do this for CSS.

* The iPhone doesn't cache any individual page component over 25k.

http://yuiblog.com/blog/2008/02/06/iphone-cacheability/

Exploit Webkit

Take advantage of Webkit: use canvas and SVG to replace image files.

Simplify, Simplify, Simplify

Keep the JavaScript small and simple, doing any 'heavy lifting' of data or computationally intensive operations on the backend.

For example, spare the iPhone from having to convert data from one form to an another, and use your backend to process the data in a form JavaScript can understand easily (use JSON).

This is not a bad idea for non-iPhone JavaScript development, either.

Make a special case for the iPhone

```
var isPhone = !!navigator.userAgent.match(/
Apple.*Mobile.*Safari/)
```

... and branch accordingly. For example, if you do form validation, validate the form on submit only, not on every change in any field as you might do on the desktop.

Advice Grab Bag

Do not use large images ever (in resolution and/or file size). This applies to img tags and to background images.

Do not use more than a handful of images files on one page.

Use the viewport command and disable zooming*

Make use of CSS3 and WebKit-specific CSS extensions, e.g. -webkit-border-radius, RGBA colors and shadows.

With animations (JavaScript-based or by using Apple's CSS animation system), avoid complex DOM structure for better framerates.