*Build Faster Web Application Interfaces*

Free Sampler

*High Performance*

# JavaScript

O'REILLY® | YAHOO! PRESS

*Nicholas C. Zakas*

# O'Reilly Ebooks—Your bookshelf on your devices!


When you buy an ebook through oreilly.com, you get lifetime access to the book, and whenever possible we provide it to you in four, DRM-free file formats—PDF, .epub, Kindle-compatible .mobi, and Android .apk ebook—that you can use on the devices of your choice. Our ebook files are fully searchable and you can cut-and-paste and print them. We also alert you when we've updated the files with corrections and additions.

**Learn more at http://oreilly.com/ebooks/**

You can also purchase O'Reilly ebooks through iTunes, the Android Marketplace, and Amazon.com.

**High Performance JavaScript**
by Nicholas C. Zakas



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



**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.




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

ISBN: 978-0-596-80279-0

[M]

1268245906

# Table of Contents

# 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

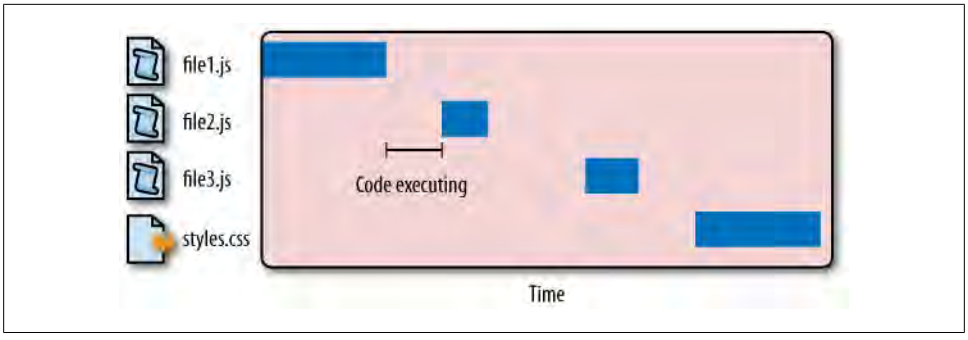


*Figure 1-1. JavaScript code execution blocks other file downloads*

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:

```
<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="
http://yui.yahooapis.com/combo?2.7.0/build/yahoo/yahoo-min.js&
2.7.0/build/event/event-min.js "></script>
</body>
</html>
```

This code has a single `<script>` tag at the bottom of the page that loads multiple JavaScript files, showing the best practice for including external JavaScript on an HTML page.

# Nonblocking Scripts

JavaScript's tendency to block browser processes, both HTTP requests and UI updates, is the most notable performance issue facing developers. Keeping JavaScript files small and limiting the number of HTTP requests are only the first steps in creating a responsive web application. The richer the functionality an application requires, the more JavaScript code is required, and so keeping source code small isn't always an option. Limiting yourself to downloading a single large JavaScript file will only result in locking the browser out for a long period of time, despite it being just one HTTP request. To get around this situation, you need to incrementally add more JavaScript to the page in a way that doesn't block the browser.

The secret to nonblocking scripts is to load the JavaScript source code after the page has finished loading. In technical terms, this means downloading the code after the `window`'s `load` event has been fired. There are a few techniques for achieving this result.

## Deferred Scripts

HTML 4 defines an additional attribute for the `<script>` tag called `defer`. The `defer` attribute indicates that the script contained within the element is not going to modify the DOM and therefore execution can be safely deferred until a later point in time. The `defer` attribute is supported only in Internet Explorer 4+ and Firefox 3.5+, making it less than ideal for a generic cross-browser solution. In other browsers, the `defer` attribute is simply ignored and so the `<script>` tag is treated in the default (blocking) manner. Still, this solution is useful if your target browsers support it. The following is an example usage:

```
<script type="text/javascript" src="file1.js" defer></script>
```

A `<script>` tag with `defer` may be placed anywhere in the document. The JavaScript file will begin downloading at the point that the `<script>` tag is parsed, but the code will not be executed until the DOM has been completely loaded (before the `onload` event handler is called). When a deferred JavaScript file is downloaded, it doesn't block the browser's other processes, and so these files can be downloaded in parallel with others on the page.

Any `<script>` element marked with `defer` will not execute until after the DOM has been completely loaded; this holds true for inline scripts as well as for external script files. The following simple page demonstrates how the `defer` attribute alters the behavior of scripts:

```
<html>
<head>
    <title>Script Defer Example</title>
</head>
<body>
    <script defer>
        alert("defer");
    </script>
    <script>
        alert("script");
    </script>
    <script>
        window.onload = function(){
            alert("load");
        };
    </script>
</body>
</html>
```

This code displays three alerts as the page is being processed. In browsers that don't support `defer`, the order of the alerts is *"defer"*, *"script"*, and *"load"*. In browsers that support `defer`, the order of the alerts is *"script"*, *"defer"*, and *"load"*. Note that the deferred `<script>` element isn't executed until after the second but is executed before the `onload` event handler is called.

If your target browsers include only Internet Explorer and Firefox 3.5, then deferring scripts in this manner can be helpful. If you have a larger cross-section of browsers to support, there are other solutions that work in a more consistent manner.

## Dynamic Script Elements

The Document Object Model (DOM) allows you to dynamically create almost any part of an HTML document using JavaScript. At its root, the `<script>` element isn't any different than any other element on a page: references can be retrieved through the DOM, and they can be moved, removed from the document, and even created. A new `<script>` element can be created very easily using standard DOM methods:

```
var script = document.createElement("script");
script.type = "text/javascript";
script.src = "file1.js";
document.getElementsByTagName("head")[0].appendChild(script);
```

This new `<script>` element loads the source file *file1.js*. The file begins downloading as soon as the element is added to the page. The important thing about this technique is that the file is downloaded and executed without blocking other page processes, regardless of where the download is initiated. You can even place this code in the `<head>` of a document without affecting the rest of the page (aside from the one HTTP connection that is used to download the file).

It's generally safer to add new `<script>` nodes to the `<head>` element instead of the `<body>`, especially if this code is executing during page load. Internet Explorer may experience an "operation aborted" error if all of the `<body>` contents have not yet been loaded.*

When a file is downloaded using a dynamic script node, the retrieved code is typically executed immediately (except in Firefox and Opera, which will wait until any previous dynamic script nodes have executed). This works well when the script is self-executing but can be problematic if the code contains only interfaces to be used by other scripts on the page. In that case, you need to track when the code has been fully downloaded and is ready for use. This is accomplished using events that are fired by the dynamic `<script>` node.

Firefox, Opera, Chrome, and Safari 3+ all fire a `load` event when the `src` of a `<script>` element has been retrieved. You can therefore be notified when the script is ready by listening for this event:

```
var script = document.createElement("script")
script.type = "text/javascript";

//Firefox, Opera, Chrome, Safari 3+
script.onload = function(){
    alert("Script loaded!");
};

script.src = "file1.js";
document.getElementsByTagName("head")[0].appendChild(script);
```

Internet Explorer supports an alternate implementation that fires a `readystatechange` event. There is a `readyState` property on the `<script>` element that is changed at various times during the download of an external file. There are five possible values for `readyState`:

* See "The dreaded operation aborted error" at *http://www.nczonline.net/blog/2008/03/17/the-dreaded-operation-aborted-error/* for a more in-depth discussion of this issue.

`"uninitialized"`
: The default state

`"loading"`
: Download has begun

`"loaded"`
: Download has completed

`"interactive"`
: Data is completely downloaded but isn't fully available

`"complete"`
: All data is ready to be used

Microsoft's documentation for `readyState` and each of the possible values seems to indicate that not all states will be used during the lifetime of the `<script>` element, but there is no indication as to which will always be used. In practice, the two states of most interest are `"loaded"` and `"complete"`. Internet Explorer is inconsistent with which of these two `readyState` values indicates the final state, as sometimes the `<script>` element will reach the `"loaded"` state but never reach `"complete"` whereas other times `"complete"` will be reached without `"loaded"` ever having been used. The safest way to use the `readystatechange` event is to check for both of these states and remove the event handler when either one occurs (to ensure the event isn't handled twice):

```
var script = document.createElement("script")
script.type = "text/javascript";

//Internet Explorer
script.onreadystatechange = function(){
    if (script.readyState == "loaded" || script.readyState == "complete"){
        script.onreadystatechange = null;
        alert("Script loaded.");
    }
};

script.src = "file1.js";
document.getElementsByTagName("head")[0].appendChild(script);
```

In most cases, you'll want to use a single approach to dynamically load JavaScript files. The following function encapsulates both the standard and IE-specific functionality:

```
function loadScript(url, callback){

    var script = document.createElement("script")
    script.type = "text/javascript";

    if (script.readyState){  //IE
        script.onreadystatechange = function(){
            if (script.readyState == "loaded" || script.readyState == "complete"){
                script.onreadystatechange = null;
                callback();
            }
        };
```

```
        } else {  //Others
            script.onload = function(){
                callback();
            };
        }

        script.src = url;
        document.getElementsByTagName("head")[0].appendChild(script);
    }
```

This function accepts two arguments: the URL of the JavaScript file to retrieve and a callback function to execute when the JavaScript has been fully loaded. Feature detection is used to determine which event handler should monitor the script's progress. The last step is to assign the `src` property and add the `<script>` element to the page. The `loadScript()` function is used as follows:

```
loadScript("file1.js", function(){
    alert("File is loaded!");
});
```

You can dynamically load as many JavaScript files as necessary on a page, but make sure you consider the order in which files must be loaded. Of all the major browsers, only Firefox and Opera guarantee that the order of script execution will remain the same as you specify. Other browsers will download and execute the various code files in the order in which they are returned from the server. You can guarantee the order by chaining the downloads together, such as:

```
loadScript("file1.js", function(){
    loadScript("file2.js", function(){
        loadScript("file3.js", function(){
            alert("All files are loaded!");
        });
    });
});
```

This code waits to begin loading *file2.js* until *file1.js* is available and also waits to download *file3.js* until *file2.js* is available. Though possible, this approach can get a little bit difficult to manage if there are multiple files to download and execute.

If the order of multiple files is important, the preferred approach is to concatenate the files into a single file where each part is in the correct order. That single file can then be downloaded to retrieve all of the code at once (since this is happening asynchronously, there's no penalty for having a larger file).

Dynamic script loading is the most frequently used pattern for nonblocking JavaScript downloads due to its cross-browser compatibility and ease of use.

## XMLHttpRequest Script Injection

Another approach to nonblocking scripts is to retrieve the JavaScript code using an XMLHttpRequest (XHR) object and then inject the script into the page. This technique

involves creating an XHR object, downloading the JavaScript file, then injecting the JavaScript code into the page using a dynamic `<script>` element. Here's a simple example:

```
var xhr = new XMLHttpRequest();
xhr.open("get", "file1.js", true);
xhr.onreadystatechange = function(){
    if (xhr.readyState == 4){
        if (xhr.status >= 200 && xhr.status < 300 || xhr.status == 304){
            var script = document.createElement("script");
            script.type = "text/javascript";
            script.text = xhr.responseText;
            document.body.appendChild(script);
        }

    }
};
xhr.send(null);
```

This code sends a `GET` request for the file *file1.js*. The `onreadystatechange` event handler checks for a `readyState` of 4 and then verifies that the HTTP status code is valid (anything in the 200 range means a valid response, and 304 means a cached response). If a valid response has been received, then a new `<script>` element is created and its `text` property is assigned to the `responseText` received from the server. Doing so essentially creates a `<script>` element with inline code. Once the new `<script>` element is added to the document, the code is executed and is ready to use.

The primary advantage of this approach is that you can download the JavaScript code without executing it immediately. Since the code is being returned outside of a `<script>` tag, it won't automatically be executed upon download, allowing you to defer its execution until you're ready. Another advantage is that the same code works in all modern browsers without exception cases.

The primary limitation of this approach is that the JavaScript file must be located on the same domain as the page requesting it, which makes downloading from CDNs impossible. For this reason, XHR script injection typically isn't used on large-scale web applications.

## Recommended Nonblocking Pattern

The recommend approach to loading a significant amount of JavaScript onto a page is a two-step process: first, include the code necessary to dynamically load JavaScript, and then load the rest of the JavaScript code needed for page initialization. Since the first part of the code is as small as possible, potentially containing just the `loadScript()` function, it downloads and executes quickly, and so shouldn't cause much interference with the page. Once the initial code is in place, use it to load the remaining JavaScript. For example:

```
<script type="text/javascript" src="loader.js"></script>
<script type="text/javascript">
    loadScript("the-rest.js", function(){
        Application.init();
    });
</script>
```

Place this loading code just before the closing `</body>` tag. Doing so has several benefits. First, as discussed earlier, this ensures that JavaScript execution won't prevent the rest of the page from being displayed. Second, when the second JavaScript file has finished downloading, all of the DOM necessary for the application has been created and is ready to be interacted with, avoiding the need to check for another event (such as `window.onload`) to know when the page is ready for initialization.

Another option is to embed the `loadScript()` function directly into the page, thus avoiding another HTTP request. For example:

```
<script type="text/javascript">
    function loadScript(url, callback){

        var script = document.createElement("script")
        script.type = "text/javascript";

        if (script.readyState){  //IE
            script.onreadystatechange = function(){
                if (script.readyState == "loaded" ||
                        script.readyState == "complete"){
                    script.onreadystatechange = null;
                    callback();
                }
            };
        } else {  //Others
            script.onload = function(){
                callback();
            };
        }

        script.src = url;
        document.getElementsByTagName("head")[0].appendChild(script);
    }

    loadScript("the-rest.js", function(){
        Application.init();
    });
</script>
```

If you decide to take the latter approach, it's recommended to minify the initial script using a tool such as YUI Compressor (see Chapter 9) for the smallest byte-size impact on your page.

Once the code for page initialization has been completely downloaded, you are free to continue using `loadScript()` to load additional functionality onto the page as needed.

### The YUI 3 approach

The concept of a small initial amount of code on the page followed by downloading additional functionality is at the core of the YUI 3 design. To use YUI 3 on your page, begin by including the YUI seed file:

```
<script type="text/javascript"
src="http://yui.yahooapis.com/combo?3.0.0/build/yui/yui-min.js"></script>
```

The seed file is around 10 KB (6 KB gzipped) and includes enough functionality to download any other YUI components from the Yahoo! CDN. For example, if you'd like to use the DOM utility, you specify its name (`"dom"`) with the YUI `use()` method and then provide a callback that will be executed when the code is ready:

```
YUI().use("dom", function(Y){
    Y.DOM.addClass(docment.body, "loaded");
});
```

This example creates a new instance of the `YUI` object and then calls the `use()` method. The seed file has all of the information about filenames and dependencies, so specifying `"dom"` actually builds up a combo-handler URL with all of the correct dependency files and creates a dynamic script element to download and execute those files. When all of the code is available, the callback method is called and the `YUI` instance is passed in as the argument, allowing you to immediately start using the newly downloaded functionality.

### The LazyLoad library

For a more general-purpose tool, Ryan Grove of Yahoo! Search created the LazyLoad library (available at *http://github.com/rgrove/lazyload/*). LazyLoad is a more powerful version of the `loadScript()` function. When minified, the LazyLoad file is around 1.5 KB (minified, not gzipped). Example usage:

```
<script type="text/javascript" src="lazyload-min.js"></script>
<script type="text/javascript">
    LazyLoad.js("the-rest.js", function(){
        Application.init();
    });
</script>
```

LazyLoad is also capable of downloading multiple JavaScript files and ensuring that they are executed in the correct order in all browsers. To load multiple JavaScript files, just pass an array of URLs to the `LazyLoad.js()` method:

```
<script type="text/javascript" src="lazyload-min.js"></script>
<script type="text/javascript">
    LazyLoad.js(["first-file.js", "the-rest.js"], function(){
        Application.init();
    });
</script>
```

Even though the files are downloaded in a nonblocking fashion using dynamic script loading, it's recommended to have as few files as possible. Each download is still a separate HTTP request, and the callback function won't execute until all of the files have been downloaded and executed.

LazyLoad is also capable of loading CSS files dynamically. This is typically less of an issue because CSS file downloads are always done in parallel and don't block other page activities.

### The LABjs library

Another take on nonblocking JavaScript loading is LABjs (*http://labjs.com/*), an open source library written by Kyle Simpson with input from Steve Souders. This library provides more fine-grained control over the loading process and tries to download as much code in parallel as possible. LABjs is also quite small, 4.5 KB (minified, not gzipped), and so has a minimal page footprint. Example usage:

```
<script type="text/javascript" src="lab.js"></script>
<script type="text/javascript">
    $LAB.script("the-rest.js")
        .wait(function(){
            Application.init();
        });
</script>
```

The `$LAB.script()` method is used to define a JavaScript file to download, whereas `$LAB.wait()` is used to indicate that execution should wait until the file is downloaded and executed before running the given function. LABjs encourages chaining, so every method returns a reference to the `$LAB` object. To download multiple JavaScript files, just chain another `$LAB.script()` call:

```
<script type="text/javascript" src="lab.js"></script>
<script type="text/javascript">
    $LAB.script("first-file.js")
        .script("the-rest.js")
        .wait(function(){
            Application.init();
        });
</script>
```

What sets LABjs apart is its ability to manage dependencies. Normal inclusion with `<script>` tags means that each file is downloaded (either sequentially or in parallel, as mentioned previously) and then executed sequentially. In some cases this is truly necessary, but in others it is not.

LABjs allows you to specify which files should wait for others by using `wait()`. In the previous example, the code in *first-file.js* is not guaranteed to execute before the code in *the-rest.js*. To guarantee this, you must add a `wait()` call after the first `script()`:

```
<script type="text/javascript" src="lab.js"></script>
<script type="text/javascript">
    $LAB.script("first-file.js").wait()
        .script("the-rest.js")
        .wait(function(){
            Application.init();
        });
</script>
```

Now the code in *first-file.js* is guaranteed to execute before the code in *the-rest.js*, although the contents of the files are downloaded in parallel.

## Summary

Managing JavaScript in the browser is tricky because code execution blocks other browser processes such as UI painting. Every time a `<script>` tag is encountered, the page must stop and wait for the code to download (if external) and execute before continuing to process the rest of the page. There are, however, several ways to minimize the performance impact of JavaScript:

- Put all `<script>` tags at the bottom of the page, just inside of the closing `</body>` tag. This ensures that the page can be almost completely rendered before script execution begins.
- Group scripts together. The fewer `<script>` tags on the page, the faster the page can be loaded and become interactive. This holds true both for `<script>` tags loading external JavaScript files and those with inline code.
- There are several ways to download JavaScript in a nonblocking fashion:
  - Use the `defer` attribute of the `<script>` tag (Internet Explorer and Firefox 3.5+ only)
  - Dynamically create `<script>` elements to download and execute the code
  - Download the JavaScript code using an XHR object, and then inject the code into the page

By using these strategies, you can greatly improve the perceived performance of a web application that requires a large amount of JavaScript code.