A BOOK APART
Brief books for people who make websites
№ 20

Mat Marquis

JAVASCRIPT FOR WEB DESIGNERS

FOREWORD BY Lara Hogan

Publisher: Jeffrey Zeldman
Designer: Jason Santa Maria
Executive Director: Katel LeDû
Editor: Erin Kissane
Technical Editors: Peter Richardson, Mike Pennisi
Copyeditor: Lisa Maria Martin
Proofreader: Katel LeDû
Compositor: Rob Weychert
Ebook Producer: Ron Bilodeau

ISBN: 978-1-937557-47-8

A Book Apart
New York, New York
http://abookapart.com

10 9 8 7 6 5 4 3 2 1

TABLE OF CONTENTS

FOREWORD

LIKE MANY PEOPLE in front-end development, as I've grown and learned, I've also come to specialize in different aspects of the field—I've built my expertise in HTML, CSS, and how to keep websites speedy. But, no matter what skills I've explored, there's an ever-growing—and tremendously intimidating—list of tools and tech to learn.

JavaScript is on that list, and for many years, I successfully avoided learning too much about it. Instead, I relied on Googling error messages, or copying and pasting scripts and trying to tweak them, never fully grokking what the code was doing.

More than a decade into my career, I was handed *JavaScript for Web Designers* and I felt that old twinge of intimidation. JavaScript was still a foreign language to me; I've been able to rely on search engines and the kindness of strangers to help get me this far— why not continue to punt on this scary, big piece of technology?

Ten pages in, all my fears of this nebulous JavaScript beast were gone.

I've learned more about this language from JavaScript for Web Designers than in the entire time I've worked in this industry. Mat's writing makes JavaScript remarkably straightforward to learn. From data types to loops, I was able to make sense of—and then easily try my hand at—this language that had been a black box to me. The book successfully builds on foundational concepts and doesn't rush things, instead navigating the reader safely through easy-to-try examples.

But this book's not just good at teaching you about JavaScript; its superpower is how effectively it'll calm your fears. I walked away from the book feeling unafraid: I can finally approach and use JavaScript without getting anxious about typing in my console. No matter your background—designer, developer, new to the game, or just new to JavaScript—you'll find that JavaScript for Web Designers is accessible, empathetic, and fun to read. I'm so excited for you to turn the page and dig in.

—Lara Hogan

INTRODUCTION

IN ALL FAIRNESS, I should start this book with an apology—not to you, reader, though I don't doubt that I'll owe you at least one by the time we get to the end. I owe JavaScript a number of apologies for the things I said to it during the early years of my career—things strong enough to etch glass.

This is my not-so-subtle way of saying that JavaScript can be a tricky thing to learn.

HTML and CSS are tough in their own ways, but we can learn about them piecemeal. They're simpler in that we type something and it happens: `border-radius` rounds corners the way we tell it to; a `p` tag is for paragraphs, full stop.

When I was just starting out with JavaScript, on the other hand, everything I learned seemed to be the tip of a new and terrifying iceberg—a link in a chain of interconnected concepts, each one more complicated than the last—variables and logic and vocabulary and mathematics that I assumed I would never fully understand. When I'd type something, it wouldn't always mean exactly what it said. If I pasted something into the wrong place, it could *anger* JavaScript. When I made a mistake, things *broke*.

If it feels like I've just plagiarized the scariest pages of your dream journal and you've already thrown this book across the room in horror, take heart: I say all this not to confirm your fears about JavaScript, but to say that I had the same ones. It wasn't long ago that I was copying and pasting pre-made scripts of dubious quality, then crossing my fingers and hoping for the best as I reloaded a page. An entire programming language seemed like too much to ever fully understand, and I was certain that I wasn't tuned for it. I was a developer, sure, but I wasn't a *developer*-developer. I didn't have the requisite robot brain; I just put borders on things for a living.

I was intimidated.

You might be in the same place I once was, standing here on the precipice of a hundred-odd pages, waiting to get blindsided by talk of "variable hoisting" and "scope chains" somewhere around page ten. I won't do that to you, and I won't talk down to you, either.

JavaScript isn't the easiest thing to understand, and my goal isn't to cover all of it in this book. I don't *know* all of it, and I'm not certain anyone does. My goal is to help you reach what is arguably the most important part of learning any language: the moment when things start to click.

What Even Is JavaScript?

The name itself could be a little confusing. Seeing "Java" in there might conjure up images of ancient browser "applets" or server-side programming languages. When JavaScript first came about back in 1995 (http://bkaprt.com/jsfwd/00-01/), it was originally named "LiveScript"—a nod to the fact that it runs as soon as the browser requests and parses it. But Java was the new hotness back in 1995—and the two languages shared a *few* syntactical similarities—so for the sake of marketing, "LiveScript" became "JavaScript."

JavaScript is a lightweight but incredibly powerful scripting language. Unlike many other programming languages, JavaScript doesn't need to be translated from human-readable code into a form that the browser can understand—there's no compiler step. Our scripts are sent across the wire at more or less the same time as our other assets—markup, images, and stylesheets—and interpreted on the fly.

While we most frequently encounter it through our browsers, JavaScript has snuck into everything from native applications to ebooks. It's one of the most popular programming languages in the world, largely because you can find it in so many different environments. JavaScript doesn't have any strict requirements for where it runs as long as there's an interpreter to make sense of it, and open-source browsers mean open-source JavaScript interpreters—the part of the browser that parses and executes JavaScript. When developers drop these interpreters into new contexts, we end up with JavaScript-powered web servers (http://nodejs.org) and homemade robots (http://johnny-five.io). We'll be looking at JavaScript as we encounter it in the browser, but there's good news if you find yourself feeling particularly mad-scientist-y by the time we're finished: the syntax you're learning here is the same

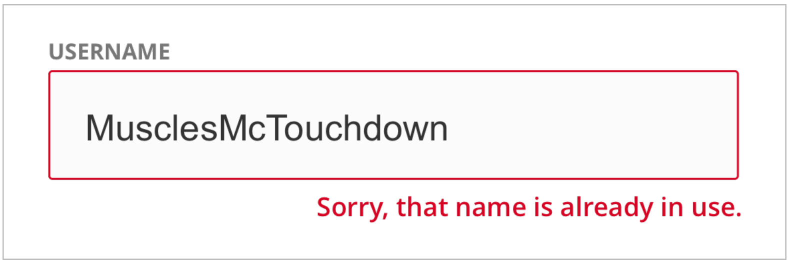


FIG 0.1: Validating the contents of an input as data is entered—rather than when the page is submitted—is a textbook example of JavaScript enhancement.

syntax you might one day end up using in your JavaScript-powered freeze ray.

The interactive layer

JavaScript allows us to add *interaction* to our pages as a complement to the *structural* layer that is markup and the *presentational* layer that is CSS.

It gives us a tremendous amount of control over a user's interactions with a page—that control even extends beyond the page itself and allows us to alter the browser's built-in behaviors. You've likely encountered a simple example of JavaScript-altered browser behavior more times than you can count: form-input validation. Before a form can be submitted, a validation script loops through all associated inputs, checks their values against a set of rules, and either allows the form submission to go through, or prevents it (**FIG 0.1**).

With JavaScript, we're able to build richer experiences for users, like pages that respond to their interactions without needing to direct them to a new page, even when requesting new data from the server. It also allows us to fill in gaps where a browser's built-in functionality might fall short, work around major bugs, or port brand-new features back to older browsers that lack native support for them. In short, JavaScript allows us to create more advanced interfaces than HTML and CSS could do alone.

What JavaScript Isn't (Anymore)

Though it gives us a lot of power over browser behavior, it isn't hard to imagine how JavaScript might get a bad reputation. To render a page unusable with CSS (no pun intended), we have to be explicit about it. `body { display: none; }` isn't something that generally makes it into our stylesheets by accident, though I wouldn't necessarily put it past me. It's even harder to make a markup mistake that would prevent the page from functioning at all—a `strong` tag mistakenly left open may not result in the prettiest page ever, but it isn't likely to completely prevent someone from using it. And when CSS and markup errors *do* cause major issues, it's apt to happen in a visible way, so if HTML or CSS completely break the page, we're likely to see it in our testing.

JavaScript differs there, though. For example: if we include a small script to validate a street address entered into a form input, the page will render just as expected—and when we punch in "5 Address Street" to test it and get no errors, our form validation may seem to be going according to plan. But if we're not careful about the rules for our validation script, a user with an oddly formatted address could very well be prevented from submitting valid information. For us to test thoroughly, we'd need to try as many strange addresses as we could find, and we'd be bound to miss a few.

Back when the web was younger and the web development profession was brand-new, we didn't have clearly defined best practices for handling JavaScript enhancements. Consistent testing was all but impossible, and browser support was *incredibly* spotty. This combination led to a lot of flaky, site-obliterating scripts making their way into the wild. Meanwhile, some of the internet's more unsavory types suddenly found themselves with the power to influence the behavior of users' browsers, held back only by boundaries that were inconsistent at best and non-existent at worst. This was not, as one might expect, always used for good.

JavaScript caught a lot of flak in those days. It was seen as unreliable, and even dangerous—a shoddily-built popup-window engine lurking somewhere beneath the surface of the browser.

Times have changed, though. The same kinds of web standards efforts that brought us semantically-meaningful markup and sane CSS support have also made JavaScript's syntax more consistent from browser to browser, and set reasonable constraints around the parts of a browser's behavior it can influence. At the same time, JavaScript "helper" frameworks like jQuery—built on a foundation of best practices and designed to normalize browser quirks and bugs—now help developers write better, faster JavaScript.

The DOM: How JavaScript Communicates with the Page

JavaScript communicates with the contents of our pages by way of an API named the *Document Object Model,* or DOM (http://bkaprt.com/jsfwd/00-02/). The DOM is what allows us to access and manipulate the contents of a document with JavaScript. Just like Flickr's API might allow us to read from and write information to their service, the DOM API allows us to read, alter, and remove information from documents—to change things on the webpage itself.

The DOM serves two purposes. The first is providing JavaScript with a structured "map" of the page by translating our markup to a form that JavaScript (and many other languages) can understand. Without the DOM, JavaScript wouldn't have any sense of a document's contents. The *entirety* of the document's contents—every individual part of our document—is a "node" that JavaScript is able to access via the DOM. Every element, comment, and even snippet of text is a node.

The DOM's second purpose is to provide JavaScript with a set of methods and functions that allow access to the mapped nodes—fetching a list of all `p` tags in the document, for example, or gathering up all elements with a class of `.toggle` that have a `.collapsible` parent element. These methods are standardized across browsers, with catchy names like `getElementsByTagName` or `createTextNode`. Having these methods built into JavaScript can lead to a little confusion over

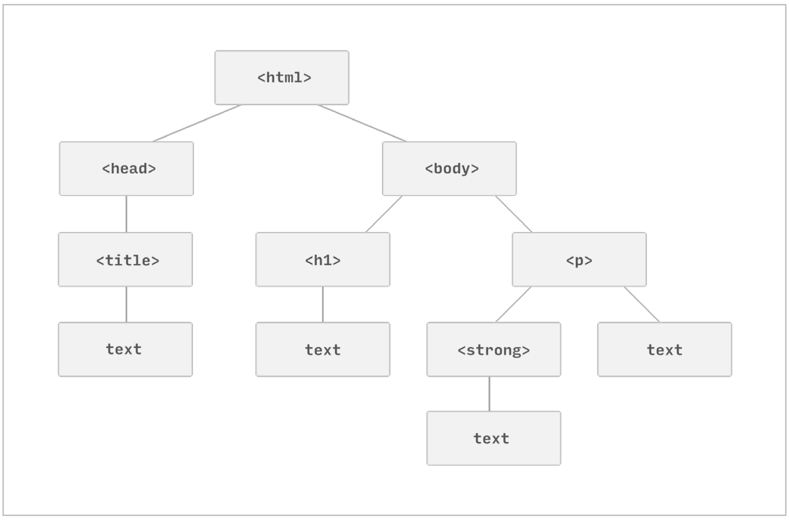


FIG 0.2: The phrase "DOM tree" makes more sense if you stand on your head.

where JavaScript ends and the DOM begins, but fortunately that isn't something we have to worry about just yet.

Let's Get Started

Over the course of this book, we'll learn the rules of the JavaScript game as it is played, spend a little quality time wading through the DOM, and pull apart some real scripts to see what makes them tick. Before that, though—before we go toe-to-toe with the beast that is JavaScript—we have to get familiar with the tools of our new trade.

I'm from a carpentry background, and if someone had sent me up on a roof on day one, I'm not sure I would have fared too well—or been around to write any of this in the first place. So in this next chapter, we'll start getting a feel for the development tools and debugging environments built into modern browsers, and we'll set up a development environment so we're ready to start writing some scripts.

1 GETTING SET UP

BEFORE WE SET FOOT on JavaScript's turf, let's get a feel for our development environment. First we're going to set up a page where we can do a little work, then we're going to have a look at that page through the lens of our browsers' built-in JavaScript debugging tools, and finally, we'll start making sense of the basic rules of JavaScript syntax.

INCLUDING JAVASCRIPT IN A PAGE

If you've worked with CSS at all, you'll find that including a script in your page follows more or less the same pattern as including a stylesheet, though with a slightly different syntax and a few small caveats.

Just like CSS, you can embed scripts within the document itself—by wrapping it in `<script></script>`, the same as you'd use `<style></style>` tags to include your CSS.

```
<html>
<head>
    ...
<script>
    // Your scripts go here.
</script>
</head>
```

All the same drawbacks as in-page styles apply here: if an identical script is meant to be used across multiple pages, there's no sense in copying and pasting it into every document. You'll end up with maintenance headaches and pages falling out of sync if you're updating things as you go along.

Fortunately, just as we use a single stylesheet across multiple pages, we can easily reference an external script anywhere we need it. It looks a bit different from the way CSS uses `link`, instead adding an `src` attribute to the `script` tag.

```
<html>
<head>
    ...
<script src="js/script.js"></script>
</head>
```

If you've encountered examples of external scripts around the internet, you might have noticed that older examples of `script` tags tend to have JavaScript-specific attributes bolted onto them by default, such as `language` or `type` attributes:

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

These have all been deprecated or made optional in HTML5. We're better off not bothering with any of them, and sticking with `<script src="js/script.js"></script>`.

Naturally, `script` accepts any of HTML's global attributes—`class`, `id`, `data-` attributes, and so on—and HTML5 has added a few helpful (and optional) attributes to the `script` element that we'll get to in a bit.

Where in a document we choose to include our external stylesheets—whether we use `<link href="css/all.css" rel="stylesheet">` in the `head` of the document or just before `</body>`—doesn't make a tremendous difference, so we conventionally include them in `head`. When it comes to JavaScript, though, we need to put a little more thought into placement, no matter whether the scripts are external or part of the page itself.

Script placement

At the risk of oversimplifying, browsers parse the contents of a file from top to bottom. When we include a script file toward the top of an HTML page, the browser parses, interprets, and executes that script before figuring out what elements are actually on the page. So, if we're planning on using the DOM to access elements within the page, we need to give the browser time to assemble a map of those elements by looking through the rest of the page—otherwise, when JavaScript goes looking for one in our script, we're apt to get an error saying that the elements don't exist.

There are a couple of methods for dealing with this problem within the script itself—the browser has methods for notifying JavaScript when the page has been fully parsed, and we can tell our scripts to wait for that event—but there are other downsides to including script files at the top of a page.

Including too many scripts in the `head` can make our pages feel slow. Upon encountering a remote script in the `head` of the document, the browser completely stops parsing the page while it fetches and parses the script, then moves on to either parsing the next script—unless we intervene, scripts are always run in order—or parsing the page itself. By including a great deal of JavaScript in the `head` of the document, we introduce the potential for users to experience a delay before the page appears.

An alternative to this rendering delay and potential for error is to include scripts at the bottom of the page, just before `</body>`. Since the page is parsed top-to-bottom, this ensures that all our markup is ready—and the page is rendered—before our scripts are requested.

This means we're shifting the burden of that *slight* delay from the rendering of a page to the request for a script, which isn't always ideal; there are times when we might want a script to be parsed as quickly as possible, even before the DOM is available. For example, Modernizr—a collection of scripts that test browser support for CSS and JavaScript features—recommends you include it in the `head` so the results of those tests are available for immediate use (https://modernizr.com/). Modernizr is light enough that the rendering delay it causes is very slight, and the results of its feature tests need to be available to any other scripts on the page, so speed is of the essence—it makes sense to block the page render for a fraction of a second to ensure it works reliably.

defer and async

While HTML5 removed the need for a lot of crufty old attributes on `script`, it did add a few new ones to deal with some of the concerns above: `<script async>` and `<script defer>`.

The `async` attribute on a `script` element tells the browser that it should—predictably—execute the script asynchronously. Upon encountering `<script src="script.js" async>` at the top of the document, the browser will initiate a request for the file and parse it as soon as it's available, but go on to parse the rest of the page in the meantime. This handles the issue of "blocking" requests for scripts in the `head`, but still doesn't guarantee the page will have been parsed in time for any DOM scripting, so we'll only want to use this in situations where we're not going to access the DOM—or where we're programmatically waiting for the document to finish loading before our script does anything with the DOM. It brings up a new issue, as well: if we're loading multiple scripts using `async`, we no longer know if they'll be loaded in the order in which they appear in the page, so we shouldn't use `async` for any scripts that are involved in dependencies.

`defer` solves the issue of waiting for the DOM to be fully available by indicating to the browser that it should request these scripts but not parse them until it has finished parsing the DOM. `defer` means our scripts at the top of the document are requested in parallel with the parsing of the page itself—so there's less chance of a perceptible delay for the user, and the scripts don't fire until the page is ready for modification. And unlike `async`, `defer` executes our scripts in the order it encounters them.

These two attributes handily solve all our problems with blocking requests and timing, save for one small catch: while `defer` has been around for a long time, it was only recently standardized, and `async` is brand new, so we can't guarantee they'll be available in all browsers.

In A Book Apart's own *Responsible Responsive Design,* Scott Jehl recommends loading scripts asynchronously using JavaScript itself: a tiny "loader" script in the `head` of the document that requests additional scripts as needed (http://bkaprt.com/jsfwd/01-01/). This not only allows us to load scripts efficiently and asynchronously, but also to decide whether they should be loaded at all: if we detect that a user is on a device that supports touch events, for example, we can load a script that gives our interface custom touch events. If touch events aren't supported, we never make that request—and the most efficient possible request is the one we never make.

All of this is a lot to take in, and we're still only scratching the surface of script loading. In the examples that follow, though, our needs are simple. We want to make sure the page has been completely parsed before any of our scripts run, so nothing needs to be in the `head` of the document—meaning that there's no need for `defer`. An external script outside the `head` won't cause any blocking behavior, so we won't need `async` either. Since there isn't a strong case for blocking the page render with the scripts we'll be writing—and we're going to need the DOM to be available, later on—we'll include our external scripts just before the `</body>` tag.

A blank slate

Before we can write any serious JavaScript, we need to set up a blank canvas—a directory with a plain ol' HTML document.

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
</head>
<body>
</body>
</html>
```

For the sake of keeping things simple and consistent, we're going to load our external script just before we close the `body` tag. Since it's external, our `script` element will have an `src` attribute pointing to our script file—which, for now, is just an empty file named script.js. I usually save mine in a js/ subdirectory—it's not a requirement by any stretch, but it can help keep things organized.

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
</head>
<body>

    <script src="js/script.js"></script>
</body>
</html>
```

Your editor and you

As with HTML and CSS, there isn't a lot of overhead to get started with JavaScript—any plaintext editor will do. JavaScript syntax can be a little harder to parse with the naked eye, though—at least until you've been steeped in it for a while.

Having your editor assign colors to keywords, functions, variables, and so on will make a script file a lot easier to understand at a glance.

Fortunately, you'd be hard-pressed to find a modern code editor that doesn't handle JavaScript syntax highlighting right out of the box. It's a safe bet that your editor of choice handles JavaScript syntax highlighting just as handily as it does markup and CSS, just by merit of our file having the .js extension—not that we'd know that from opening up our still-empty script.js file.

We'll need that for sure once we start assembling JavaScript's component parts into purposeful, functional script files. While we're getting the hang of the basics, though, let's open our preferred browser and start getting a feel for in-browser development tools.

OUR DEV TOOLS

There was a time—not all that long ago, honestly—where browsers didn't give us much help dealing with JavaScript. At best we'd get a heads-up that there was an error of some kind, and a sometimes-accurate guess at the line number where it occurred. Debugging a script meant making a change, reloading the page, hoping nothing blew up, and repeating the process if it did. There was usually no way of getting more details on the issue—at least, no particularly *helpful* details.

Luckily, our development tools grew up as JavaScript got more and more advanced, and modern desktop browsers all come with advanced JavaScript debugging tools built in. We can still approach development that way, of course, but it's a bit like using the back of a nail gun to pound nails into a board, and just as likely to end with us getting shot in the foot.

In the grand scheme, getting a feel for your dev tools will end up saving you a tremendous amount of time tracking down bugs. For our purposes, it gives us a space to start experimenting with JavaScript's syntax.

We'll be looking at Chrome's dev tools here, but the basics will apply to whatever browser you most prefer. Browsers' dev tools are usually similar right down to the command you use to open them up: command+option+i on a Mac and control+shift+i on a PC. Things will look a little different from browser to browser, but you'll find that all of them share a very similar layout.

Now, if you've already spent some quality time with these tools in your development browser of choice, inspecting elements and debugging CSS issues, then you'll be familiar with the "elements" tab, showing all the elements on the page and their associated styles. Beyond that, the other tabs will vary a bit from browser to browser.

Most developer tools also feature a "network" tab, allowing you to monitor the number and size of the requests made by a page and the time it takes to load them all, as well as a "resources" tab, to let you look through all the resources associated with a page, from stylesheets to JavaScript files to cookies. Some browsers will also include network information under "resources." There's usually some form of "timeline" tab that charts information about how a page is rendered, such as the number of times the browser has to go through and "repaint" the styles throughout the page—you may have to reload the page with this tab open before you see much (**FIG 1.1**).

We'll be spending most of our debugging time in the "console" tab, which allows you to run JavaScript in what's known as a REPL, or *read-eval-print loop* (http://bkaprt.com/jsfwd/01-02/). You can write JavaScript into the console and execute it the same way the browser's JavaScript engine would if it lived in the page. This has a couple of benefits: first, we can start tinkering with JavaScript on any page we want, without worrying too much about a dev environment. Second, anything we write in the console is designed to be ephemeral: by default, any changes you've made in JavaScript will be wiped out if you reload the page.

The JavaScript console

The JavaScript console has two main functions when it comes to testing and debugging: it provides us with a place to log errors

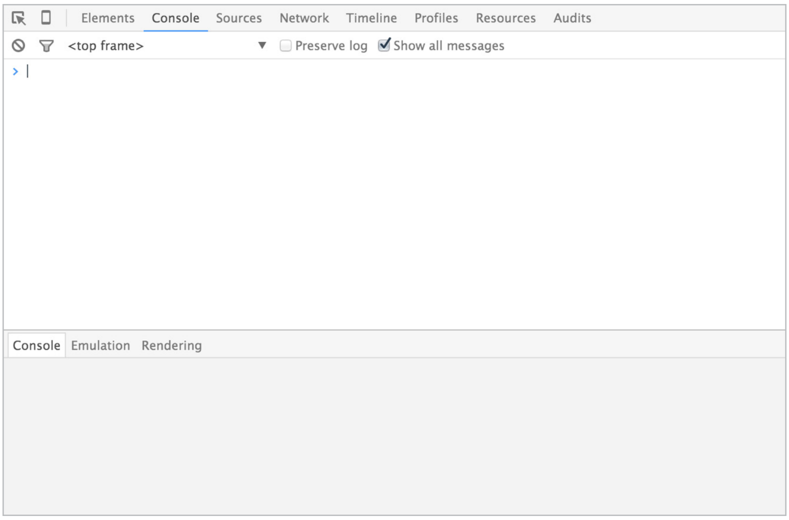


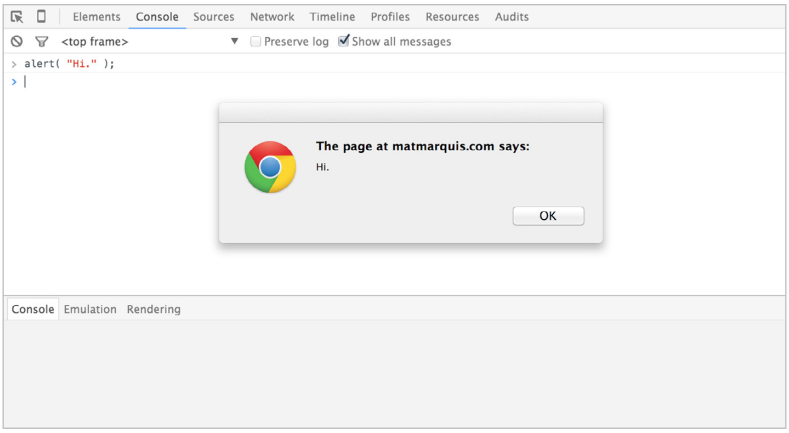
FIG 1.1: Chrome's dev tools open to the "console" tab.

and information, and a JavaScript prompt for interacting with the page and our scripts directly.

In its simplest form, the JavaScript console serves to show you any syntax errors in your scripts—if a typo should sneak into your script or part of the script references something that doesn't exist, you're no longer left wondering what's keeping your script from running.

Most dev consoles go further than only showing outright errors, providing you with warnings about features that browsers might be removing soon, failed requests, and so on. It's very rare that I do any development without the console open, just to be on the safe side.

Oftentimes, though, we'll encounter a situation where there aren't any outright *errors* in our scripts, but things still don't seem to be working quite the way we'd expect them to—or we need a simple way to flag for ourselves that certain parts of a script are being executed, since so much of our logic is happening invisibly. In these cases, you can use some methods built into the browser to send up the occasional signal flare—to



```
> alert( "Hi." );
> |
```

The page at matmarquis.com says:

Hi.

FIG 1.2: Using `alert()` on a live website is the JavaScript equivalent of shouting "fire" in a crowded theatre: it isn't illegal, but certainly isn't going to win you any friends.

send yourself messages, inspect the contents of variables, and leave a trail of breadcrumbs showing a path through the logic of a script.

In the olden days, we repurposed a few of JavaScript's earliest built-in methods to perform some basic debugging—the much-derided `alert()`, a method that causes a native modal window to appear bearing our message of choice, in quotes between the parentheses, and an OK button to dismiss it (**FIG 1.2**).

There were a few equally exciting variations on this method: `confirm()` allows the user to "OK" or "cancel" the text we specify, and `prompt()`, which allows the user to input text in the modal window—both of these reported the user's selection and input back to us for further use in our scripts. If you've agreed to an "end-use license agreement" or spent any quality time with Geocities in the past, you've likely encountered all of these at some point or another.

JavaScript developers learned pretty quickly that this was an obnoxious way to interact with users, and none of these methods see much use these days—at least, not much unironic use.

What we did learn is that it gave us a quick and easy way of communicating things to ourselves while debugging, allowing us a little insight as to what was going on in our scripts. Inefficient as it was, we could set `alert`s telling us how far a script had progressed, whether parts of a script were being executed in the right order, and (by seeing which was the last `alert` to fire before we ran into an error) track down glaring issues line-by-line. This sort of debugging wouldn't tell us much, of course—`alert` was really only designed to pass along a string of text, which would often mean inscrutable feedback like `[object Object]` when we wanted to look closer at what a part of our script meant.

These days, browsers compete on the quality of their dev tools, and we have tons of options for digging into the internals of our scripts—the simplest of which is still to send ourselves a message from inside the script, but with the potential to contain much more information than a simple string of text.

Writing to the console

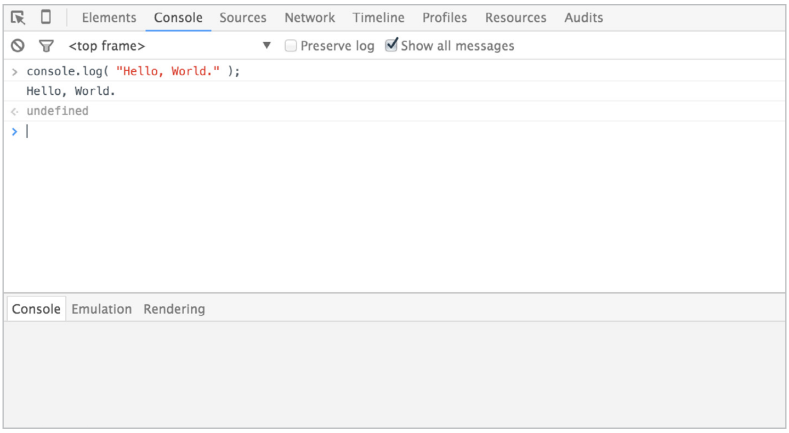
The simplest way to output something to the console from our script is a method named `console.log()`. In its simplest form, `console.log()` works just like `alert()`—allowing you to pass yourself a note within your script.

We've officially reached the point where some things are easier to show than to tell, so let's open up script.js in our editor and try the following line out for ourselves:

```
console.log("Hello, World.");
```

Save the file, switch back over to your browser, reload the page—and we've just written our very first line of JavaScript together (**FIG 1.3**).

Now, I know this doesn't seem like the most exciting thing in the world just yet, but we can use `console.log` to do a tremendous amount of work. It comes in a couple of different flavors, as well: we can use `console.warn` and `console.error` the same way we use `console.log`, to make particular issues and messages stand out (**FIG 1.4**).



```
> console.log( "Hello, World." );
  Hello, World.
<· undefined
> |
```

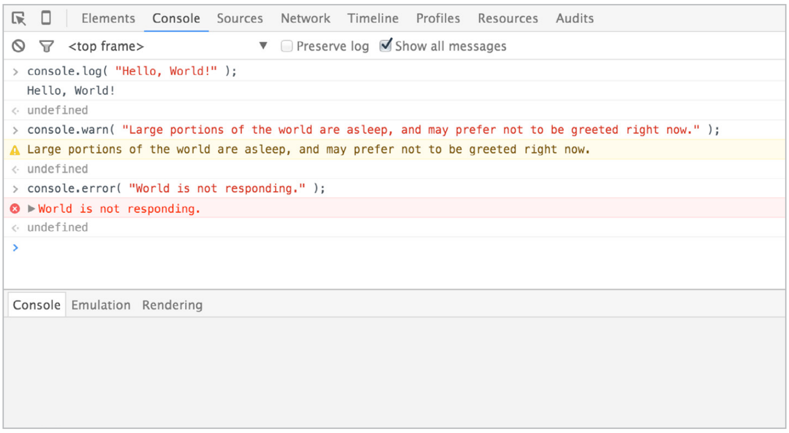
FIG 1.3: Well, hello to you too, dev console.

One final note on the topic of console.log and its ilk: while writing to a console is supported in every modern browser, support isn't *universal*—some older browsers don't have a console at all. In particular, IE6 and IE7 are famous for breaking down upon encountering the unrecognized `console.log` method, throwing errors that are likely to break your scripts.

Fortunately, these methods have little place in production code—they're really only something we'll be using when writing and debugging our scripts, so there's little risk of it causing any problems for users—unless we leave one in by accident. Be sure to check for any leftover debugging code like `console.log` before using a script on a live website, just to be safe.

Working in the console

Now, the JavaScript console is more than just a place to log messages—you likely noticed the blinking prompt below the logs earlier. This input is the REPL I mentioned earlier—the *read-eval[uate]-print loop*.



```
> console.log( "Hello, World!" );
  Hello, World!
< undefined
> console.warn( "Large portions of the world are asleep, and may prefer not to be greeted right now." );
⚠ Large portions of the world are asleep, and may prefer not to be greeted right now.
< undefined
> console.error( "World is not responding." );
⊗ ▶World is not responding.
< undefined
>
```

FIG 1.4: `console.warn` and `console.error` are both useful when debugging a script.

This short explanation of the REPL is that it allows us to send things directly to the browser's JavaScript parser, without needing to update our script file and reload the page. If you enter the same `console.log("Hello, World.");` in this space and hit return, it appears in the console.

We can use this to get information about the current state of elements on the page, check the output of scripts, or even add functionality to the page for the sake of testing. Right now, we can use it to try out new methods and get immediate feedback. If you punch in `alert("Test")` and hit enter, you'll see what I mean: no changing files, no reloading the page, no-fuss-no-muss. Just an ol'-fashioned obnoxious modal window, made to order.

It bears repeating that we can't do any real damage by way of the console. Any changes we make to the page or to our scripts by way of the console REPL will evaporate as soon as you reload the page, with no changes made to any of our files.

Now we have a couple of options for experimenting with JavaScript via the console, and a dev environment where we can start cobbling our first few scripts together. We're ready to get started learning the rules of JavaScript.

THE FUNDAMENTAL RULES

JavaScript is complex for sure, but the global rules of the language are surprisingly simple—and generally pretty forgiving, with a few exceptions. It's helpful to run through some of these rules upfront, and don't worry if they don't all make perfect sense before we have a chance to see them in action.

Case-sensitivity

One strict rule—one that occasionally trips me up to this day—is that JavaScript is case-sensitive. That means that `avariable` and `aVariable` are treated as two completely different things. This can seem a little tricky when JavaScript's built-in methods for accessing the DOM have names like `getElementsByTagName`, which doesn't exactly roll off the tip of one's keyboard.

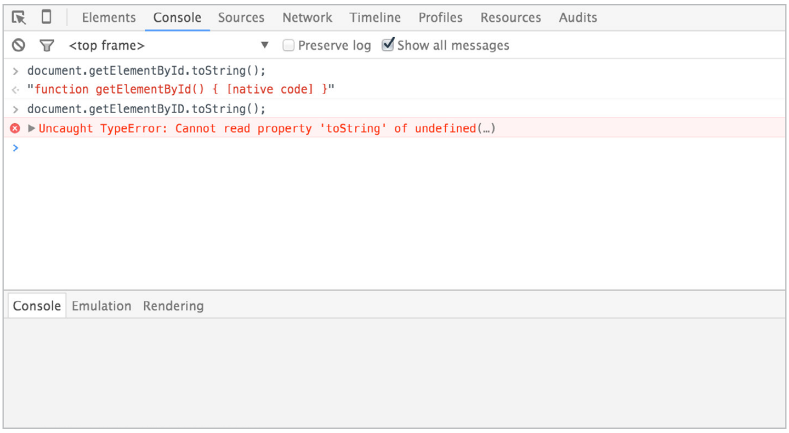
For the most part, we can expect built-in methods to use camel case, capitalizing every word after the first, as in `querySelector` or `getElementById`.

You can see this rule in action via the console: try entering `document.getElementById.toString()` and you'll likely get a response that mentions native code—the browser recognizes this as a built-in method for accessing an item in the DOM. Enter `document.getElementByID.toString()` however—with the *D* in "Id" capitalized—and the browser only returns an error (**FIG 1.5**).

Semicolons

A statement in JavaScript should almost always end in a semicolon, which is a way to tell a JavaScript parser that it has reached the end of a command, the same way a period ends a sentence in English. This rule is a little flexible: a line break can also signal to a parser that it's the end of a statement, thanks to something called *Automatic Semicolon Insertion,* or ASI.

Now, if you can believe this, programmers are an opinionated group. It won't take much searching to find endless



```
> document.getElementById.toString();
< "function getElementById() { [native code] }"
> document.getElementByID.toString();
✖ ▶Uncaught TypeError: Cannot read property 'toString' of undefined(…)
>
```

FIG 1.5: So close, yet so far away.

debates over whether to *always* use a semicolon at the end of a statement, or to just save yourself the occasional byte and let ASI do its job. Personally, I'm in the former camp—I'd always rather be explicit by using a semicolon than risk omitting one that I wasn't supposed to, and I find that it makes code easier to read for the next person who comes along and needs to maintain it. For now, I'd definitely recommend you do the same. It takes a while to get the hang of where semicolons are absolutely necessary and where ASI can fill in the blanks, so we're better off erring on the side of caution.

White space

Perhaps weirder still, line breaks are the only form of white space—which includes tabs and spaces—that has any real significance to JavaScript, and even then it's only the first one that counts. Whether you use fifty line breaks between lines of code, or start every line with ten tabs and a space for good luck, JavaScript will ignore it all the same. Only the first newline, where ASI comes swooping in to assume you're between statements, has any real significance.

Comments

JavaScript allows you to leave comments that are ignored when the script is executed, so you can leave reminders and explanations throughout your code. I find this helpful on a day-to-day basis: leaving myself pointers and reminders of why things are set up a certain way, or comments telling myself that part of the code still needs some work.

Far more important than that, however, is keeping in mind that you won't always be the single owner of a codebase—even if you're not working on a team, there's a chance that someone else may end up making changes to your code someday. Well-commented code serves as a roadmap for other developers, and helps them understand what decisions you made and why.

There are two flavors of comment native to JavaScript, one of which will be familiar to anyone who's spent quality time with CSS. Multiline comments are handled using the same syntax as CSS comments:

```
/* This is a multi-line comment.

Anything between these sets of characters will be
ignored when the script is executed. This form of
comment needs to be closed. */
```

JavaScript also allows for single-line comments, which don't need to be explicitly closed. Instead, they close as soon as you start a new line.

```
// This is a single-line comment.
```

Unexpectedly, single-line comments can wrap to as many lines as necessary in your *editor,* so long as they don't contain a line break—as soon as you press Return to start a new line, the comment is closed, and you're back into executable code. The wrapping that might be performed by your code editor—

depending on the editor itself and your settings—is called "soft wrapping." Single-line comments won't be impacted by soft wrapping, since it's strictly an editor-level convenience.

```
console.log("Hello, World."); // Note to self:
  should "World" be capitalized here?
```

WE'RE READY

Now that we know the rules of the game and we've set up a couple of places to experiment, we're ready to start learning about the building blocks that make up JavaScript. In terms of sitting down and writing a script from start to finish, they might not seem like they amount to much on their own—but the things we're about to cover in the next chapter are critical to understanding how JavaScript treats data.

2

UNDERSTANDING DATA TYPES

THINGS ARE ABOUT TO GET REAL.

By the end of this chapter, you'll have a sense for the types of data you'll encounter during your JavaScript-writing travels. Some types of data will seem self-explanatory, at least on the surface: numbers are numbers, strings of text are strings of text. Some types of data will veer a little more toward the philosophical-sounding: `true`, as a keyword in JavaScript, represents the very essence of trueness.

Things go a bit deeper than that, however, and sometimes in particularly confusing ways. Numbers can be *truthy* or *falsy*, while text will always be truthy. `NaN`—a JavaScript keyword meaning "not a number"—is itself something JavaScript considers to be a number. `({}+[])[!+[]+!+[]+!+[]]+(![]+[])[!+[]+!+[]+!+[]]` is perfectly valid JavaScript. Really.

It's not hard to see where JavaScript gets a reputation for being difficult to understand intuitively—the statements above read more like a riddle than the rules of a scripting language. There are methods to the madness, however, and getting the hang of JavaScript's data types is how we start learning to think like JavaScript.

JavaScript is a "weakly typed" language, meaning that we don't have to be explicit about whether something should be treated as a number or a string of text. Unlike a "strongly typed" language—which requires us to define data as a certain type—JavaScript infers the meaning from context. This makes sense, considering that more often than not, we'll want `7` to be treated as a number and not a string of text.

In the event that we *do* want something handled as a specific type, there are a number of ways to perform *type coercion*, which changes the way JavaScript interprets data. Fortunately, we don't need to worry about that yet, so let's take a look at the data types themselves.

PRIMITIVE TYPES

We hold some data types to be self-evident, and *primitive* data types are exactly that. Primitive types can't be reduced any further than what they already are: a number is a number, `true` is true. Primitives are the simplest form of data in JavaScript: numbers, strings, `undefined`, `null`, and `true` and `false`.

Numbers

The number type is a set of all possible number values. JavaScript is pretty good at numbers, up to a point. If you punch `7` into your console and hit return, the result shouldn't be too surprising: the output is *7*. JavaScript has acknowledged that this is the number seven. You and I, we have done strong work with the JavaScript console today.

There are a few special cases under the number umbrella: the "not a number" value (`NaN`), and a value representing infinity, which can be either positive (`Infinity`) or negative (`-Infinity`). If you type either of these into your console and press return, you'll get them echoed back to you just like the *7*—which is really just JavaScript saying, "I am aware of this concept." If you try entering `infinity` or `Nan`, however, both of those will come back as *undefined*—remember, JavaScript is case-sensitive.

OPERATOR	DESCRIPTION	USAGE	RESULT
`+`	Addition	`2+2`	`4`
`-`	Subtraction	`4-2`	`2`
`*`	Multiplication	`2*5`	`10`
`/`	Division	`5/10`	`2`
`++`	Add one to a number	`2++`	`3`
`--`	Subtract one from a number	`3--`	`2`
`%`	Return the remainder after dividing two numbers	`12 % 5`	`2`

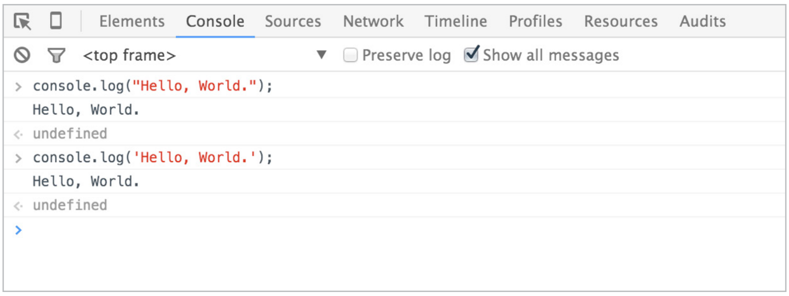
FIG 2.1: There won't be a quiz, I promise.

Likewise, it shouldn't come as too much of a surprise that we can use mathematical operators in JavaScript, which play out just the way you might expect. If you enter `2+2` in your console, JavaScript will return *4*.

A lot of these operators will be familiar, even to those of us that just *barely* scraped by in high school algebra classes (ahem); a few operators are more unique to programming (**FIG 2.1**).

The mathematical Order of Operations applies here: any expressions wrapped in parentheses are evaluated first, followed by exponents, multiplication, division, addition, and subtraction. I bet you never thought you'd be hearing the phrase "Please Excuse My Dear Aunt Sally" again, but time and math make fools of us all.

```
2*2-2
2

2*(2-2)
0
```



```
> console.log("Hello, World.");
  Hello, World.
<· undefined
> console.log('Hello, World.');
  Hello, World.
<· undefined
>
```

FIG 2.2: Double quotes and single quotes give us the exact same result. In your face, Strunk & White.

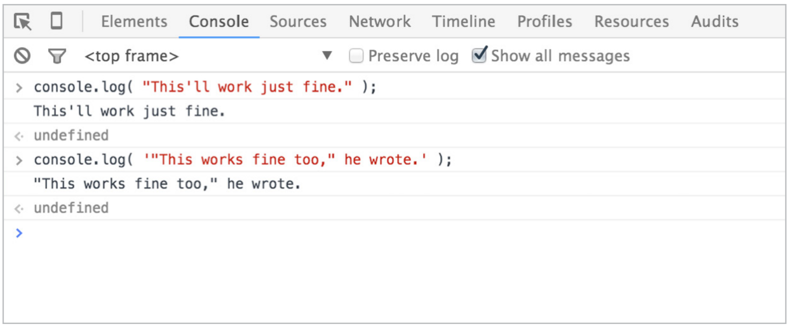
You're not apt to run into `Infinity` or `NaN` too often during the course of your JavaScript career—at least, not on purpose. If you try out `2/0` in the console, assuming your computer doesn't collapse into a singularity, JavaScript will return *`Infinity`*.

`NaN` is a special case we'll see a bit more frequently. Any time we try to treat non-numbers as numbers, JavaScript will return *NaN*—for example, if we take the phrase `"Hello, World."` that we tried out in `console.log` earlier and multiply it by two (`"Hello, World." * 2`), we'll get *NaN* as a result. JavaScript doesn't know what you're supposed to get when you multiply a word by a number, but it knows for sure that whatever you'd end up with wouldn't be a number.

Strings

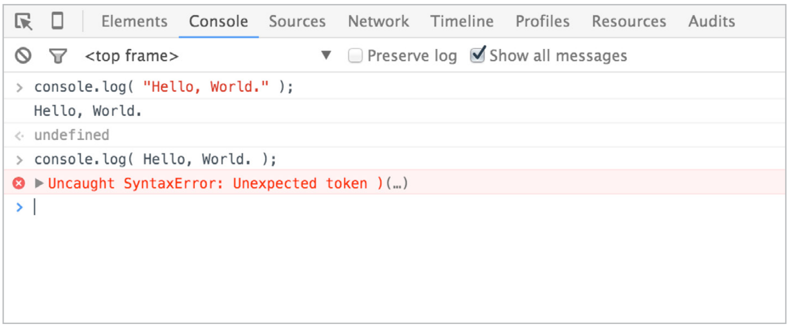
Strings of text are quite possibly the simplest data type to understand. Any set of characters—letters, numbers, symbols, and so on—between a set of double or single quotes is a "string."

As a matter of fact, we've already been introduced to strings—when we wrote `console.log("Hello, World.");` in the console in the previous chapter, `"Hello, World."` was a string. We would see the same result with single quotes, as in `console.log('Hello, World.');` (**FIG 2.2**).



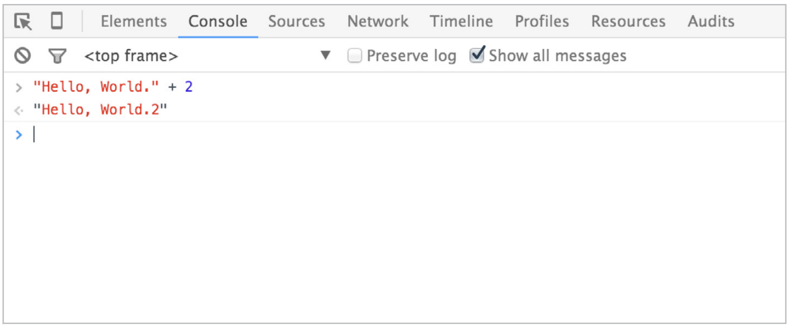
```
> console.log( "This'll work just fine." );
  This'll work just fine.
<- undefined
> console.log( '"This works fine too," he wrote.' );
  "This works fine too," he wrote.
<- undefined
>
```

FIG 2.3: All's well, so long as our quotes are paired properly.



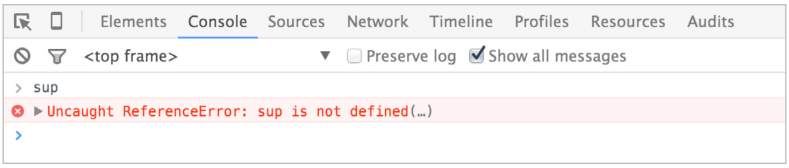
```
> console.log( "Hello, World." );
  Hello, World.
<- undefined
> console.log( Hello, World. );
✖ ▶Uncaught SyntaxError: Unexpected token )(…)
> |
```

FIG 2.4: Uh-oh.



```
> "Hello, World." + 2
<- "Hello, World.2"
> |
```

FIG 2.5: The sequels are never as good as the originals.



```
> sup
✖ ▶Uncaught ReferenceError: sup is not defined(…)
>
```

FIG 2.6: We haven't told JavaScript that `sup` means anything, and JavaScript is lousy at slang.

Single and double quotes are functionally identical as long as you pair them properly, and a string using double quotes can contain single quotes, or vice-versa (**FIG 2.3**).

If we *omit* the quotes, however, the results are very different. Without quotation marks, JavaScript attempts to read `Hello, World.` as part of the script rather than as a string of text, and throws a syntax error (**FIG 2.4**).

Strings are refreshingly uncomplicated: just some letters and/or numbers inside of a set of quotes. There's one other important facet to strings, though: we can create new ones by joining them together, or joining them with a number.

Combining multiple sources into a single string is called *string concatenation*. You can join two or more strings by using a plus sign, which pulls double-duty for both mathematical addition and string concatenation, depending on its surrounding context (**FIG 2.5**).

When dealing with strings instead of numbers, `+` doesn't attempt to perform mathematical addition. Instead, it concatenates two data types into a single string. Even though the above example contains a number, involving a string at all means JavaScript treats `2` as a string as well.

undefined

`undefined`, as you might expect, is the type for anything that isn't predefined by JavaScript, or defined by us as part of our script. You've seen examples of this data type already: when we were playing around with case-sensitivity in our dev consoles, we entered a few things that JavaScript didn't recognize, and got an error back (**FIG 2.6**).

If we use `typeof`—an operator that returns a string indicating the type of unevaluated operand—to determine the type of `sup`, we'll see that it has a type of *`undefined`*. `sup` has no meaning or value, so far as JavaScript knows—we never gave it one.

null

`null` represents a non-value: something that has been *defined*, but has no inherent value. For example: we might define a variable as `null` with the expectation that it gets assigned a value at some point in a script, or assign the `null` value to an existing reference to zero out a previous value.

Booleans

Boolean values—the keywords `true` and `false`—represent inherent trueness and falseness. They're concepts you'll come across in any programming language. If we ask JavaScript to compare any two values and they come up equal, the entire statement evaluates to *`true`*—if not, *`false`*.

Let's take a look at our console again, and in doing so *gaze into the very face of absolute universal truth*:

```
2 + 2 == 4
true
```

And while we're at it, let's use our developer console to put some Orwellian newspeak to the test:

```
2 + 2 == 5
false
```

Okay, it may not seem quite as dramatic as I wanted it to sound, but these kinds of comparisons are the basis for a tremendous amount of scripting logic.

Note that we're using `==` to perform a comparison here, rather than the `=` you might have expected: JavaScript sees a single equals sign as an attempt to *assign* something a value, rather than performing a comparison between one value and another. More on this in a bit—and we'll discuss comparison operators further in the next chapter.

OBJECT TYPES

The concept of "an object" in JavaScript maps nicely to the concept of an object here in the real world. In both cases, an object is a collection of *properties* and *methods*—that is, traits belonging to an object, and functions that the object can perform. In the real world, for example, "hammer" is an abstraction of properties ("handle," "a weighted striking surface") and purposes ("hitting things"). But the concept of "hammer" is mutable: if we were to change the properties ("MC," "unusual pants," "cannot be touched") and purposes ("breaking it down"), "hammer" comes to mean something altogether different.

An object in JavaScript is the same idea: a named, mutable collection of properties and methods. Outside of the primitive types listed above, every bit of JavaScript we write is an "object," from the strings and numbers we define, up to the entire document itself.

That sounds a little overwhelming, but the specific types of objects we're apt to run into day-to-day are clearly differentiated.

Variables

A variable is a symbolic name for a value. Like so many *x*'s in so many eighth-grade algebra classes, variables in JavaScript act as containers for a value that can be any kind of data: strings, numbers, elements we've retrieved via the DOM, even entire functions. They give us a single point of reference for that value, to be used in all or part of our script. We can modify the value of that variable at any time and in whatever ways we want.

There are two ways to declare a variable, both of which use the single equals sign, which performs an *assignment* rather than making a comparison. The simplest way to declare a variable doesn't use much else, really: we specify the identifier, and use a single = to assign it a value.

```
foo = 5;
5
```

When we first create a variable, our console acknowledges us by parroting back the new variable's value.

If we now punch in `foo` and hit enter, we get the same result—we've made JavaScript aware of a variable named `foo`, and defined its value as the number five. Once defined, the behavior of a variable is identical to the data it contains. Checking the type of variable `foo` using `typeof` is revealing:

```
foo = 5;
5

foo;
5

typeof foo;
"number"
```

`foo`'s type is now "number," not "variable." As far as JavaScript is concerned, the variable `foo` is functionally identical to the number five. That's not a permanent condition, however: we can reuse a variable by assigning it a new value.

```
foo = 5;
5

foo = 10;
10
```

We can even reassign a value to a variable using the variable itself:

```
foo = 100;
100

foo = foo * foo;
10000
```

Of course, we won't always know upfront what value our variable should contain. The whole idea, after all, is that variables can represent any number of values in a predictable, easy-to-reference package. In the event that we don't need or want our variable to have a starting value, we can still make JavaScript aware of it. Using `var foo;`, we declare a new variable (`foo`) as `undefined`, as confusing as that might sound. So "foo" is now a word that JavaScript identifies as a variable, but without any assigned value. Try this out in your JavaScript console and you'll see what I mean.

```
var bar;
undefined

bar;
undefined

whatever;
Uncaught ReferenceError: whatever is not defined
```

We defined `bar` as a variable, so when we enter it into our console, the REPL dutifully parrots its value back to us. That value, since we didn't give it one, is *undefined*. If we try the same thing with a variable we haven't defined—`whatever`, in this case—JavaScript throws an error.

Note that `var` up there? It's not required that you use the `var` keyword to declare a variable if you're assigning it a value immediately, but for reasons I'll explain soon, it's a good idea to always declare your variables with the `var` keyword even when it's not required. Similarly, though it isn't always required by the rules of JavaScript, it's best to always end variable assignments with a semicolon.

```
var foo = 5;
undefined
```

Don't sweat your console's *undefined* response after assigning a value to an identifier—the JavaScript engine doesn't actually have anything to output in response to tasks like declaring a variable, so we get `undefined` in return.

We can also declare more than one variable at once. As with so many things in JavaScript, we have a couple of options for defining multiple variables, using two different but equivalent syntaxes. The first uses a single `var` keyword and splits the sets of variable names and assigned data with commas (ending with a semicolon, of course):

```
var foo = "hello",
bar = "world",
baz = 3;
```

The second method uses individual `var` keywords:

```
var foo = "hello";
var bar = "world";
var baz = 3;
```

There are no catches, in this case. These two syntaxes work the exact same way, and choosing one over the other is entirely a matter of personal preference. This, of course, means that it is a *hotly contested* subject in JavaScript developer circles.

Now, it would be irresponsible of me to foist my personal opinion on you here, reader, so I'll leave it at this: always adhere to the existing code conventions of a project, rather than mixing and matching. On a brand-new project, use whichever syntax you find the most comfortable, but keep an open mind—we have trickier problems to solve than fighting over personal preferences. And when in doubt: do the thing that I like best, because I am right.

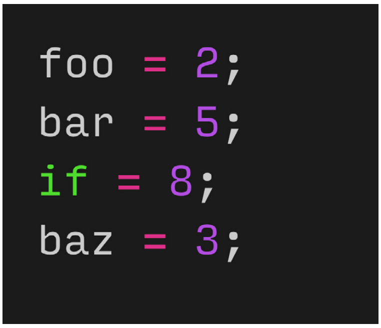
No, really, use whichever one you find more comfortable.

Pretty sure I'm right, though.

Identifiers

The name we give a variable is called an *identifier*.

Like everything in JavaScript, identifiers are case-sensitive, and come with a few special rules, as well:



```
foo = 2;
bar = 5;
if = 8;
baz = 3;
```

FIG 2.7: Syntax highlighting can make it easier to catch errors on the fly.

- They must start with a letter, underscore, or dollar sign—not a number.
- They can't contain spaces.
- They can't contain special characters (`! . , / \ + - * =`).

There are a set of words in JavaScript that can't be used as identifiers, like null, for example. These are called *keywords*—words that already have an immutable meaning to JavaScript, or are set aside just in case they get added to JavaScript one day:

```
abstract boolean break byte case catch char class
  const continue debugger default delete do double
  else enum export extends false final finally
  float for function goto if implements import in
  instanceof int interface long native new null
  package private protected public return short
  static super switch synchronized this throw throws
  transient true try typeof var void volatile while
  with
```

That's a scary block of words, but this isn't something you need to have committed to memory—I certainly don't. It does make a good case for an editor with syntax highlighting, though, which can help you avoid mysterious-seeming errors when assigning an identifier to a variable (**FIG 2.7**).

Outside of these rules, an identifier can contain any combination of letters, digits, and underscores. It's a good idea to use identifiers that are brief (`totalCost` vs. `valueOfAllItemsIncTaxAndShipping`) and easy to understand at a glance (`selectedValue` vs. `v1`). The "foo," "bar," and "baz" that I've been using in my examples are lousy identifiers—the words have no meaning whatsoever, so coming across them in a script would give you no clues as to the nature of the data they contain. At the same time, we should avoid identifiers that describe their potential values in t*oo much* detail, since we may not always be able to immediately predict the values a variable will contain. A variable originally named `miles` may need to contain a value in kilometers one day—confusing for the developers who end up maintaining that code, ourselves included. `distance` works much better.

Variable scope

We'll get into this more when we look at functions, but we can't talk about variables without discussing something called *variable scope*.

Think of variable scope as the section of your source code where you've assigned something an identifier. Outside of that section, that variable is not defined, and the identifier may be reused for something else. JavaScript applications can be huge, with tens of thousands of lines of code being parsed and executed. Because variables have their own scope, we can elect to make them available to the entire application *or* constrained to individual sections of our code, so we don't have hundreds of variables potentially tripping us up throughout an application. If we had to keep a mental inventory of what identifiers were already in use so we didn't run the risk of accidentally reusing or redefining a variable, we *would* need those special programmer robot-brains we talked about at the outset.

There are two kinds of variable scope: *local* and *global*. A variable defined outside of a function is global. And because global variables are, well, *global*, they can be accessed anywhere in the entire application.

A variable defined *inside* a function can be either local or global, depending on how we define it—which really comes down to whether we declare it by using the keyword `var`. Inside a function, `var` declares a variable in that function's local scope, but omitting `var` means that variable should be global—in other words, exposed to the entire application.

```
(function() {
  var foo = 5;
}());
undefined

console.log( foo );
Uncaught ReferenceError: foo is not defined
```

Variable scope is a complicated topic, and we'll get into the gritty details when we start learning about functions. For now, just know that it's a good idea to always define your variables using `var`. Always using `var` means local variables stay local and global variables stay global—which means we don't spend hours of debugging time trying to track down the function that unexpectedly changed a global variable's value. And when the time comes to expose a local variable to the global scope, we'll talk through better ways of doing it than omitting `var`.

Arrays

Arrays aren't all that different from variables, with one major exception: while a variable contains a single value, an array can contain multiple values, like a list. The syntax is similar to variables' syntax, too:

```
var myFirstArray = [ "item1", "item2" ];
undefined
```

This should look pretty familiar: a `var` keyword, followed by an identifier that we think up, and then a single equals sign to perform an assignment. All the same identifier rules apply here, too—in fact, all the rules of variables apply to arrays, including scope.

Things differ a little beyond that, though: instead of pointing the identifier at a single data type, we create a list—in this example, a pair of strings—inside a set of square brackets and separated by a comma. Remember that spaces inside the array don't matter—they're just a matter of personal preference. `var myFirstArray = [ "item1", "item2" ];` is 100% identical, so far as JavaScript cares, to `var myFirstArray=["item1","item2"];`. I just find the former a little easier to read.

Just as with variables, arrays can be made up of any data types:

```
var myFirstArray = [ "item1", 2, 3, true, "last item" ];
undefined
```

When we punch that identifier into our developer console, the console parrots back the value, just like a variable:

```
var myFirstArray = [ "item1", 2, 3, true, "last item" ];
undefined

myFirstArray
["item1", 2, 3, true, "last item"]
```

We likely won't need to access the entire array all at once, though. We're much more likely to use an array to package up several items of related data, with intent to access them individually. We access them using *indexes*: numbers that correspond with the positions within the array.

```
var mySecondArray = [ "first", "second", "third" ];

undefined

mySecondArray;
["first", "second", "third"]
```

```
mySecondArray[ 0 ];
"first"

mySecondArray[ 1 ];
"second"

mySecondArray[ 2 ];
"third"
```

You may note that JavaScript breaks with an easy assumption here: while you might expect the first item in the array to correspond with the index `1`, JavaScript is *zero-indexed*, which means that JavaScript starts indexing at `0` and counts up from there.

When we reference a position within an array using an index, it isn't much different from working with variables: any reference to an array position takes on the data type of the data it contains—and just like a variable, we can reassign data to a given array position using a single equals sign.

```
var mySecondArray = [ "first", "second", "third" ];

mySecondArray[ 2 ];
"third"

typeof mySecondArray[ 2 ];
"string"

mySecondArray[ 2 ] = 3;
3

mySecondArray;
["first", "second", 3]

typeof mySecondArray[ 2 ];
"number"

mySecondArray[ 3 ] = "numero cuatro";
"numero quattro"
```

```
mySecondArray;
["first", "second", 3, "numero cuatro"]
```

So far, we've only used brackets when initializing an array—and we'll always want to use brackets when *accessing* information in an array—but there's an alternative method for initializing an array:

```
var myFirstArray = new Array( "item1", "item2" );
undefined

myFirstArray;
["item1", "item2"]
```

As we've used them here, with strings, there's really no difference between using brackets and using the `new Array()` syntax.

Likewise, we can use either the bracket syntax or the `new Array()` syntax to initialize an array with no defined items, just like we can initialize a variable but leave it undefined. To do this, we use either an empty set of brackets or the `new Array()` syntax with nothing in the parentheses:

```
var arrayThree = [];
undefined

var arrayFour = new Array();
undefined
```

Again, these are functionally identical: both syntaxes initialize an empty array.

There *is* one thing that the `new Array()` syntax can do that brackets can't, and that's initialize an array with a set number of items—even when those items are undefined:

```
var threeItemArray = new Array( 3 );
undefined

threeItemArray
[undefined × 3]
```

All this means is that a new array has been created, with three as-yet-undefined items. Beyond that, the behavior is the same as the arrays we've seen so far: you're not limited to those three items, and you can set and access information the exact same way.

This syntax can get a little confusing, however: you're passing the `new Array()` syntax the number of items you want in the array the same way you'd pass it the data you wanted *in* the array. That means you can end up with very different results from the bracket syntax when you're storing number data types. JavaScript is smart enough to know that *multiple* numbers in the `new Array()` parentheses mean you're creating an array of numbers:

```
var numberArray = [ 777, 42, 13, 289 ];
undefined

numberArray;
[777, 42, 13, 289]

var otherNumberArray = new Array( 777, 42, 13, 289 );
undefined

otherNumberArray;
[777, 42, 13, 289]
```

But if you're looking to initialize an array containing a single item—and that item is a number type—we get very different results with the two different syntaxes. Bracket notation works as we might expect—an array containing a single item with the value we assigned it:

```
var numberArray = [ 777 ];
undefined

numberArray;
[777]

numberArray[ 0 ];
777
```

With the `new Array()` syntax, things get weird. We end up with an array containing seven hundred seventy-seven *undefined* items.

```
var otherNumberArray = new Array( 777 );
undefined

otherNumberArray;
Array[777]

otherNumberArray[ 0 ];
undefined
```

Now, I'll be perfectly honest: I've never needed to initialize an array with a given number of `undefined` items right off the bat—your mileage may vary, of course, but I get by just fine with bracket notation.

Once defined, arrays come with a number of associated methods for navigating and changing their data. For example, the `.length` property on an array describes the number of items in that array:

```
var theFinalArray = [ "first item", "second item",
  "third item" ];
undefined

theFinalArray.length;
3
```

And since the index itself is a plain ol' number data type, we can get a little creative with how we access information in an array:

```
var theFinalArray = [ "first item", "second item",
  "third item" ];
undefined
```

```
// Get the last item in the array:
theFinalArray[ theFinalArray.length - 1 ];
"third item"
```

Here we're using the `.length` of the array to find the index of the last item. Since an array can be any length, we can't just use a number to get to the last item. We can use the `.length` property to get a count of all the items in the array, so we know how many items it contains. JavaScript is zero-indexed, though, so we can't *just* use the array's length—there are three items in the array, but the indexes start counting at zero. Easy enough to deal with: we just subtract one from the array's length—a number data type—to get the index of the last item.

Objects and Properties

An object can contain multiple values as *properties*. Unlike an array that accepts a set of data types and assigns each item a numbered index, an object's properties are named using strings.

```
var myDog = {
  "name" : "Zero",
  "color" : "orange",
  "ageInYears" : 3.5,
  "wellBehaved" : false
};
undefined
```

Each property is made of up a *key/value pair*. The "key" in "key/value" is a string we define that points to a value—as with naming a variable, we want our keys to have names that are predictable, flexible, and easy to understand. In the above example, the keys for each property of the `myDog` object are `name`, `color`, `ageInYears`, and `wellBehaved`, and the respective values are the strings `Zero` and `orange`, the number `3.5`, and the Boolean `false`.

The properties of an object can themselves be treated as objects with properties of their own, allowing us to bundle up a tremendous amount of information in a highly portable package.

```
var myDog = {
  "name" : {
    "first" :"Zero",
    "middle" : "Baskerville",
    "last" : "Marquis"
  },

  "color" : "orange",
  "ageInYears" : 3,
  "wellBehaved": false
};
undefined
```

Remember that the whitespace in these examples—the indentation, line breaks, and spaces around the colons—won't matter to JavaScript. Those are just there to keep things human-readable.

Defining an object

There are two ways to define a new object. One is with the `new` keyword, whose syntax shouldn't be entirely unfamiliar at this point:

```
var myDog = new Object();
undefined
```

The second way is with *object literal* notation:

```
var myDog = {};
undefined
```

Both of these work a lot like declaring a variable: we use the `var` keyword, followed by an identifier and a single equals sign.

These two methods of defining an object work the same way, except for one major difference. The `new` keyword requires us to first define an object, *then* start filling it with data:

```
var myDog = new Object();
undefined

myDog.nickname = "Falsy";
"Falsy"
```

Object literal notation allows us to define *and assign data to* an object all at once:

```
var myDog = {
  "nickname": "Falsy"
};
undefined
```

You'll find that a lot of developers favor object literal notation for the sake of simplicity, and we'll be doing the same in this and future chapters.

Accessing and changing properties

Once we've defined an object using either of the above methods, there are two ways to access and change the information inside an object: *dot notation* and *bracket notation*.

To access information in an object's property using dot notation, you use a period between the object identifier and the property key.

```
var myDog = {
  "name": "Zero"
};
undefined
```

```
myDog.name;
Zero
```

Bracket notation uses a set of brackets and a string that points to the key we're looking to access, just like the way we'd use an index in an array. Unlike dot notation, we use a string data type to point to our keys—so, we need to wrap `name` in quotes.

```
var myDog = {
  "name": "Zero"
};
undefined

myDog[ "name" ];
Zero
```

The reason bracket notation requires a string is the reason bracket notation exists at all: in complex scripts, we might need to programmatically access certain keys based on custom logic that we've coded. In order to do that, we may need to put together a custom string from strings, numbers, variables, and so on. Say we had a script that randomly selected one of the keys in the following object:

```
var cars = {
  "car1" : "red",
  "car2" : "blue",
  "car3" : "green"
}
undefined
```

We might have a variable that contains a number between one and three, and use that to create a string that points to one of those three keys. There are plenty of ways to generate a random number with JavaScript, but for the sake of keeping things uncomplicated: we'll just use the number two, and create a concatenated string that reads `car2`.

```
var cars = {
  "car1" : "red",
  "car2" : "blue",
  "car3" : "green"
}
undefined

var carKey = "car" + 2;
undefined

carKey
"car2"

cars.carKey
undefined
```

We won't be able to use dot notation in a situation like this, since JavaScript isn't going to treat `carKey` like a variable. Given how dot notation syntax works, JavaScript thinks `carKey` is the *identifier* of the key that we're looking for, not the string it contains.

Bracket notation, however, *expects* a string—and since `carKey` contains a string, the following works just fine:

```
var cars = {
  "car1" : "red",
  "car2" : "blue",
  "car3" : "green"
}
undefined

var carKey = "car" + 2;
undefined

carKey
"car2"

cars[ carKey ];
"blue"
```

You'll find a lot of ways to get clever with bracket notation during the course of your JavaScripting career. Unless you *need* to get clever, though, dot notation is the simpler of the two syntaxes, and I find it much easier to read at a glance.

Functions

A *function* is a block of reusable code that allows us to perform repetitive tasks without repeating the same code throughout a script. Instead, we use an identifier to reference a function containing that code, and pass the function any information it needs to perform a task for us.

In fewer words, a function is an object that *does* something, rather than just holding a value.

Defining a function involves a little more code than we're used to, though the first few parts shouldn't be too surprising. As usual, `var` defines the scope, then we define an identifier of our choosing, and use a single equals sign to assign that identifier a value. Instead of a simple string, number, Boolean, etc., we follow the equals sign with the keyword `function` and a set of parentheses. Then, between two *curly braces*, we put all the code we want that function to execute whenever we call it. As usual, we end the statement with a semicolon.

```
var whatup = function() {
  console.log( "Hello again, world." );
};
undefined
```

If we paste this code into our console: nothing happens. No *`Hello again, world.`*—at least, not yet. So far, all we've done is define a function with the identifier `whatup` that, when called, will output the sentence, "Hello again, world."

If we type `whatup` into our console, your dev console will either respond with *`function whatup()`* or the entirety of the function's code, depending on the browser—in either case, this is just the browser acknowledging that it knows about a function with that identifier. In order to actually *execute* the function, we have to call it using the identifier and a pair of parentheses:

```
var whatup = function() {
  console.log( "Hello again, world." );
};
undefined

whatup;
function whatup()

whatup();
Hello again, world.
```

In their simplest form, functions might not seem all that useful, since we'll rarely want to execute the *exact same* lines of code—leading to the exact same output—over and over again. The real power of functions lies in passing them information for that code to act upon, leading to different results. The parentheses that follow the function's identifier can do more than just tell the browser to execute the function we've assigned to the identifier `whatup`: they can be used pass information along to the code inside the function, in the form of *arguments*.

```
var greet = function( username ) {
  console.log( "Hello again, " + username + "." );
};
undefined
```

By adding `username` between the parentheses when defining a function, we're saying the function should create a variable named `username`, and that variable should contain whatever value we pass along between the parentheses when we execute the function. In this case, the function is expecting us to pass along a string that gets concatenated into our "hello again" greeting:

```
var greet = function( username ) {
  console.log( "Hello again, " + username + "." );
};
undefined
```

```
greet( "Wilto" );
Hello again, Wilto.
```

Of course, we're not doing much to validate the data being passed to the function—and we'll get into that later on—but for now, string concatenation is pretty resilient thanks to JavaScript's type coercion. Even if we pass along another data type, things generally work as expected.

```
var greet = function( username ) {
  console.log( "Hello again, " + username + "." );
};
undefined

greet( 8 );
Hello again, 8.

greet( true );
Hello again, true.
```

Things *do* get a little weird if we omit the argument altogether, though:

```
var greet = function( username ) {
  console.log( "Hello again, " + username + "." );
};
undefined

greet();
Hello again, undefined.
```

Since we didn't populate `username` with any information—but JavaScript was aware of the identifier—`username` was an undefined data type. Thanks to type coercion, the undefined data type became the *string* `"undefined"` Not the most elegant way to phrase things, but not inaccurate either—the function is greeting someone whose name we never defined, after all.

One of the more common—and powerful—uses of functions is to provide you with a packaged, reusable method of *calcu-*

lating something. I don't mean that in a strictly mathematical sense, though you can certainly do that as well. By setting a function up to "return" a value, we allow a function to be treated the same way as we would treat a variable: as a container for data that behaves just like the data it contains.

Functions can potentially return—and behave like—the final result of infinitely complex logic rather than data that we've hand-defined. We don't cover *infinitely complex logic* until next chapter, so for now we'll have our function return something relatively simple: the sum of two values.

```
function addTwoNumbers( num1, num2 ) {
  return num1 + num2;
}
undefined

addTwoNumbers( 4, 9 );
13

typeof addTwoNumbers( 2, 2 );
"number"
```

To go one step further, we can even assign a function with a return value to a variable:

```
function addTwoNumbers( num1, num2 ) {
  return num1 + num2;
}
undefined

var sum = addTwoNumbers( 2, 3 );
undefined

sum
5

typeof sum
"number"
```

It’s important to keep in mind that using the `return` keyword means that the final purpose of a function is to return a value. If we include any code inside a function *after* a `return` statement, that code is never executed.

```
function combineStrings( firstString, secondString ) {
  return firstString + secondString;
  console.log( "Hello? Can anyone hear me?" );
}
undefined

combineStrings( "Test", " strings" );
"Test strings"
```

Since a `return` statement comes before `console.log`, the `console.log` is never executed. In fact, your editor might even highlight it as an error.

Pretty much everything is an object

These are the common object types you’ll run into during your adventures in JavaScripting. While we haven't used them in terribly complex ways yet, they combine to make up the entirety of JavaScript itself. From top to bottom, JavaScript is made of predefined objects that behave just like the custom ones we’ve been defining here.

Under certain conditions, everything but `null` and `undefined` can be considered objects—even strings, which are arguably the simplest data type of them all. A new string comes with methods and properties *built in*—just like arrays—even though all we did was define a snippet of text:

```
"test".length
4
```

Technically this string itself isn’t an object—it doesn’t have any methods or properties of its own. When we ask for the value of the `length` property, though, JavaScript knows what we mean—it has a list of predefined methods and properties that it applies to all strings.

It that a necessary distinction? Not at this point, no; in fact, it's a little confusing. But the further you dig into the makings of JavaScript itself, the more sense that distinction will start to make. Until then, you're likely to see this behavior described much more succinctly whenever the subject of what is and isn't a JavaScript object comes up: "everything is an object…kinda."

Now that we've got a feel for some of the building blocks that will make up our scripts, we can start writing some logic around them. In other words: now that we understand the basics, we can start writing scripts that *do things*—things apart from blindly chucking text into our developer console, that is.

By default, a browser "reads" a script the same way you would read this page in English: from left to right and from top to bottom. *Control flow* statements are used to control what portions of our code are run at a given time, and whether they're executed at all.

It sounds complicated at first, but it breaks down to a handful of very simple statements that allow us to do amazingly complex things in concert. For our purposes, control flow statements fit pretty neatly into two categories: conditional statements and loops. That's what we'll be digging into in the next two chapters.

3 CONDITIONAL STATEMENTS

CONDITIONAL STATEMENTS are a type of control flow concerned with logic: they determine when and where to execute code, based on conditions you specify.

if/else STATEMENTS

Conditional statements are almost entirely some variation on "given *X*, do *Y*." The most common example of this—and one nearly ubiquitous in terms of any programming language—is the `if`/`else` statement. Saying "if this, do that" is about as uncomplicated as a logical statement gets, but by the end of this chapter you'll see how something so simple on the surface can make up the lion's share of logic in our scripts.

if

In its most simple form, an `if` statement will execute whatever code you specify between a set of curly braces, but only if the contents of the parentheses that follow the `if` keyword evaluate to `true`.

From the previous chapter we know that JavaScript returns *true* for `2 + 2 == 4`, if we punch it into our developer console. Instead of putting that alone in our console, let's try it out in our very first `if` statement. Remember that a single equals sign (`=`) is used to *assign* values, while two (`==`) equals signs are used to perform a basic comparison.

```
if( 2 + 2 == 4 ) {
  console.log( "Hi there." );
}
Hi there.
```

Nothing too surprising here: "Hi there." appears in our developer console. If we enter a statement that we know to be false, the line containing `console.log`—and *any* code we put between those curly braces—will be skipped.

```
if( 2 + 2 == 5 ) {
  console.log( "Hi there." );
}
undefined
```

Now, this doesn't seem terribly useful when we're entering statements that we already know to be true or false, but the purpose of an `if` statement isn't just to periodically make sure the rules of mathematics still apply. Considering that JavaScript objects can contain all manner of complex data that we'll need to act on in different ways throughout a script—and remembering that objects are treated exactly the same as the data they contain—we can make some incredibly complex decisions about the flow of a script using simple `if` statements. For now, let's just initialize a single variable containing a number data type, so we can experiment a little.

```
var maths = 5;

if( maths == 5 ) {
  console.log( "This number is five." );
}
This number is five.
```

Since `if` blindly evaluates the contents of the two parentheses that follow it for truth, we don't always have to make a specific assertion there—we can use it to check a Boolean value the same way.

```
var foo = false;

if( foo ) {
  /* Any code placed here will never execute, unless
  you change `foo` to `true` */
}
```

else

`else` is used to run alternate lines of code, in the event that the contents of an `if` evaluate to `false`. The `else` keyword follows the closing curly brace for the `if` statement, and is followed by a set of curly braces that contain whatever code should run in the event that the `if` code doesn't run. We don't need a set of parentheses here, since we're not evaluating any new data—we're just taking a different action, depending on the conditions of the `if`.

```
var maths = 2;

if( maths > 5 ) {
  console.log( "Greater than five." );
} else {
  console.log( "Less than or equal to five." );
}
  Less than or equal to five.
```

else if

There's a shorthand for stringing together a number of `if` statements: `else if`. While it isn't necessarily the neatest way of performing complex comparisons, it's definitely worth knowing.

```
if( lunch == "gravel" ) {
  console.log( "That isn't food.");
} else if ( lunch == "burrito" ) {
  console.log( "A burrito is an excellent choice." );
} else {
  console.log( "It might not have been a burrito,
    but at least it wasn't gravel." );
}
```

This script performs a series of tests against the variable `lunch`: the first `if` checks to make sure lunch does not, in fact, have a value of `"gravel"`—and having passed this critical lunchtime test, we can now see whether it has a value of `"burrito"`, or—with the final `else`—none of the above.

`else if` isn't a JavaScript keyword in the same way that `if` and `else` are individually—`else if` is more of a syntactical workaround, a shorthand for multiple nested `if`/`else` statements. The code above is structurally identical to the following:

```
if( lunch == "gravel" ) {
  console.log( "That isn't food.”);
} else {
  if ( lunch == "burrito" ) {
    console.log( "A burrito is an excellent choice." );
  } else {
    console.log( "It might not have been a burrito,
      but at least it wasn't gravel.” );
  }
}
```

Whether using `else if` or nesting multiple `if`/`else` statements, this isn't the easiest code to read and understand. There are much better ways of performing multiple comparisons in a single go, and we'll discuss some of those a little later.

COMPARISON OPERATORS

We wouldn't get a lot of use out of conditional statements if all we could do with them is test whether two values are equal. Fortunately—and perhaps expectedly, at this point—there's a lot more we can do with a few simple conditional statements. You saw a little of this earlier when we used an `if` to determine whether an identifier had a `number` value greater than five: conditional statements can be used to compare all kinds of values in all kinds of ways, all by replacing the `==` we've been using in our comparisons so far.

Equality

I've mentioned a couple of times that we should use `==` for comparisons, but that isn't—and this awful pun will only make sense in a little while—*strictly* true.

JavaScript provides us with two different approaches to comparison: the `==` we've been using so far, and `===`, which is the "strict equals." Two equals signs together perform a "loose" comparison between two values, which means that typing `2 == "2"` into our developer console will give us *`true`*, even though we're comparing a number to a string. JavaScript is smart enough to coerce two dissimilar data types to matching ones when a comparison is performed with `==`, and make a guess at what we meant to compare.

`2 === "2"`, on the other hand, gives us back *`false`*—no type coercion is performed behind the scenes when a comparison is made using `===`. The two values being compared have to be not only equal, but also of the same type—they have to be identical.

If you think that makes `==` feel a little too magical for its own good, you're right—developers have a strong preference for using `===` whenever possible, as it does away with any ambiguity that might result from auto-coercion.

Truthy and falsy

There's one potentially useful thing `==` gets us that `===` doesn't: the ability to divine "truthy" and "falsy" values.

Those aren't strangely consistent typos: everything in JavaScript can be coerced to a `true` or `false` Boolean value when using the non-strict comparison operator.

This sounds a little confusing, but you won't have to maintain a spreadsheet of which values are truthy and which are falsy—they follow a clear line of reasoning: "If something, truthy; if nothing, falsy." For example, `0` is a falsy value—likewise `null`, `undefined`, `NaN`, and an empty string (`""`). Everything else is truthy—a string, a number, and so on.

The uses for this might not be immediately obvious, but imagine a situation where you're writing a function that outputs a string—like the ones we used when we were first covering string concatenation:

```
function greetUser( name ) {
  console.log( "Welcome, " + name + "!" );
}

greetUser( "Muscles McTouchdown" );
Welcome, Muscles McTouchdown!
```

If you remember, omitting the argument containing the user's name didn't result in an error since JavaScript is aware that the `name` variable *exists*, but the `undefined` value gets coerced to a string.

```
greetUser();
Welcome, undefined!
```

That isn't particularly desirable behavior; I personally wouldn't be too keen on being called "undefined." Fortunately, we can use an `if`/`else` statement to tailor the output a little:

```
function greetUser( name ) {
  if( name ) {
    console.log( "Welcome, " + name + "!" );
  } else {
    console.log( "Welcome, whoever you are!" );
  }
}
```

By default, JavaScript evaluates the contents of those parentheses in a way that coerces to a Boolean value—it looks for truthy and falsy values, and a string is a truthy value. *Now* if we try out this new function without passing along a name as an argument:

```
greetUser( "Mat" );
Welcome, Mat!

greetUser();
Welcome, whoever you are!
```

It works!

This function is a little clunky, though. Instead of having two places where this function writes to the console (or later to the page, to another part of our script, etc.), our code would be better organized if we could reduce these two mostly redundant `console.log` calls to a single one. For whoever ends up maintaining our code after us—and for our own sanity—it's a good idea to keep your scripts as terse as possible. You'll frequently see this concept referred to as DRY, which stands for *don't repeat yourself*. If you have to change something in your code later, you're better off only needing to do so in one place. In our example, it might be as simple as changing "Welcome" to "Hi," but, in a sufficiently complex script, it would be impossible to keep a mental inventory of all the places you'd need to make redundant changes. Keeping our code DRY means we're closer to following a single path through our code.

So in our function, instead of two lines outputting different strings to the console, we'll conditionally tailor the string itself and output the final result at the end.

```
function greetUser( name ) {
  if( name === undefined ) {
    name = "whomever you are";
  }
  console.log( "Welcome, " + name + "!" );
}
```

Much more succinct. If `name` doesn't have a value, we give it one—rather than relying on truthy/falsy coercion when we already know we're dealing with a potentially `undefined` variable, we check for that specifically. Then, by the time we reach the `console.log` statement, we know `name` is defined.

One of the ways I keep the complexity of my own code in check is stepping through it in plain English. Previously, our function did the following:

> *If* `name` *has a truthy value, output a string containing* `name` *to the console, but if* `name` *has a falsy value, output an alternate string to the console.*

That's as awkward to say out loud as it is to read in the code itself. Compare that to the way we'd walk through our new function:

> *If* `name` *is undefined, define it. Output a string containing* `name` *to the console.*

Much better—and congratulations on your first JavaS-cript refactor.

Inequality

`!` is called a *logical NOT operator*, which means that it negates whatever immediately follows it:

```
true
true

false
false

!true
false
```

When we use the logical NOT operator (`!`) in front of another data type—like a number or a string—it reverses the truthy/falsy value of that data.

```
"string"
"string"

!"string"
false

0
0

!0
true
```

The same way `==` and `===` return a `true` value if the two values being compared are loosely or strictly equal, the `!=` and `!==` operators return a `true` value if the two values being compared *aren't* equal. That's a little hard to picture in text, but makes a lot more sense in the context of an `if` statement.

```
var foo = 2;

if( foo != 5 ) {
  console.log( "`foo` is not equal to five" );
}
```

Just like with `==`, `!=` attempts to coerce the data types being compared so they match. If we use `!=` to compare the number

`>`	Greater than
`<`	Less than
`>=`	Greater than or equal to
`<=`	Less than or equal to

FIG 3.1: A quick rundown of relational operators.

`2` to the string `"2"`, JavaScript considers the two to be equal—so the result is `false`.

```
2 != "3"
true

2 != "2"
false
```

Relational operators

Relational operators are a little more intuitive than the equality operators (**FIG 3.1**).

These work just the way you might expect—no catches, no crazy negation operators to mull over. You'll use these to compare one number value to another:

```
3 > 1
true

3 < 1
false

10 >= 5
true

5 >= 5
true
```

```
var bikeName = "Bonneville";
if( bikeName.length <= 10 ) {
  console.log( "There are at least ten characters in
    this bike’s name." );
}
```

LOGICAL OPERATORS

`if`/`else` statements can do a lot of work using only what we know so far, but logical operators allow us to form even more complex logic by chaining comparisons together in a single expression. You’ve already met one of the logical operators, the logical NOT (`!`) that negates any value that follows it, but `!` is the odd operator out when compared to the other two: logical OR (`||`) and logical AND (`&&`).

`||` and `&&` allow you to evaluate multiple values within the same expression: multiple comparisons separated by `||` mean the entire expression will return *true* if *any* of the expressions evaluate to true, while comparisons separated by `&&` mean that the expression will only return *true* if *all* the expressions evaluate to true. This is another tough one to visualize without seeing it in action, so back to the dev console we go:

```
5 < 2 || 10 > 2
true
```

Five obviously isn’t less than two—that statement alone would never evaluate to `true`. Ten *is* greater than two, however—and since we’re using a logical OR between the two comparisons, this entire statement evaluates to *true*.

```
10 > 5 && "toast" === 2
false
```

Ten is greater than five, sure—that part of the expression is true. But the string `toast` clearly has nothing to do with the number two; they’re not equal, and certainly not *strictly* equal. Since we’re using the logical AND between these two expressions and one of them returned *false*, this entire statement evaluates to *false*.

Grouping expressions

Multiple expressions separated by `&&` and/or `||` (get it? "and/or?") will be evaluated from left to right. In the following statement, JavaScript never gets as far as evaluating the number of characters in `myString`.

```
2 + 2 === 9 && "myString".length > 2
false
```

Since JavaScript saw an expression that returned *false* followed by a logical AND, the entire statement couldn't possibly be true. The same goes for a logical OR:

```
2 + 2 !== 9 || "myString".length > 2
true
```

Since the first expression evaluates to *true* and is then followed by a logical OR, there's no need for JavaScript to continue evaluating the statement—the whole thing evaluates to *true* right away.

We can change the way this evaluation behaves using parentheses, and thus do we creep back toward algebra, in a way. We'll start with a set of three statements that evaluates to *true* all together, and we'll use Booleans so we can see it all the way JavaScript does:

```
false && true || true
true
```

The first thing JavaScript looks at here is whether `false && true` evaluates to *true*, which it doesn't. `true && true` would, sure, but "false AND" means that the statement is certain to return *false*. But following that is a logical OR—so having evaluated the first half of the statement to *false*, JavaScript is now evaluating the second half of the statement as `false || true`. Since we're using a logical OR and one of the values is `true`, this entire statement—read left to right—is *`true`*.

If we add a set of parentheses around the second part of the statement, however, we change the way they're evaluated:

```
false && ( true || true )
false
```

Now the first thing JavaScript evaluates is still `false &&`, but the parentheses mean everything afterwards is a single expression to be evaluated—not three things to evaluate, but two. Since the left-hand side of the logical AND is *`false`*, evaluation stops there. "false AND" can never possibly return true, so with a single pair of parenthesis, we've changed this statement to false.

Got a headache yet? I'm not far behind you, and I do this stuff all day. That's why I'm in the habit of using parentheses to *clarify* how JavaScript evaluates these complex statements as much as I use them to *alter* it. Now that we know JavaScript evaluates expressions wrapped in parentheses as a single expression, that first statement—the one that evaluated to *`true`*—might be a little easier to read when written like this:

```
( false && true ) || true
true
```

Think back to that convoluted `else if` example earlier on. Now that we can perform more advanced comparisons within the span of a single `if` statement, we can do away with a lot of the complexity, even when we add more functionality—for example, tacos:

```
var lunch = "tacos";
if( lunch !== "gravel" && ( lunch === "burrito" ||
  lunch === "tacos" ) ) {
  console.log( "Delicious." );
}
```

And, for good measure, let’s turn that into a function:

```
function mealChecker( lunch ) {
  if( lunch !== "gravel" && ( lunch === "burrito" ||
    lunch === "tacos" ) ) {
    console.log( "Delicious." );
  }
}

mealChecker( "Tacos" );
undefined
```

Something has gone horribly wrong! We passed the function a string the way it might expect, but we made one little mistake: we capitalized the *T*. Since JavaScript is case-sensitive, `Tacos` isn’t equal to `tacos`—and our script was expecting the latter.

Now, we could make a rule for ourselves that this function should only ever receive an all-lowercase value, but that’s one more thing we have to document—or worse, “just keep in mind from now on.” A better approach would be to plan for both upper- and lowercase values by getting JavaScript to normalize things for us.

If you recall from the previous chapter, even though we don’t define them with methods or properties the way we might define an object from scratch, any string we define will come with a set of built-in properties: `.length` gives you the number of characters in the string, for example. Here, we’ll use one of the native methods for transforming a string to make sure we’re comparing apples to apples and tacos to tacos: `.toLowerCase()`. Just like the name implies, it returns the all-lowercase value of a string:

```
"THIS IS A STRING".toLowerCase();
this is a string
```

While it returns the lowercase value of a string, it’s important to keep in mind that this—and methods like it—don’t *change* a string to lowercase. If we have a string stored in a variable and call `.toLowerCase()` on it, the variable remains unchanged.

```
var foo = "A String";
undefined

foo
"A String"

foo.toLowerCase();
"a string"

foo
"A String"
```

That means there are a couple of ways to handle the comparisons in our function. The first way is to call `.toLowerCase()` on every instance of the `lunch` variable throughout the function:

```
function mealChecker( lunch ) {
  if( lunch.toLowerCase() !== "gravel" && (
    lunch.toLowerCase() === "burrito" ||
    lunch.toLowerCase() === "tacos" ) ) {
    console.log( "Delicious." );
  }
}

mealChecker( "Tacos" );
Delicious.
```

That *does* work, but it isn't very DRY code. I think we can do better. Instead, we'll only use `.toLowerCase()` once—at the top of our function, before we do any string comparison—and use it to change the value of the `lunch` variable to the lowercased version of itself that `.toLowerCase()` returns.

```
function mealChecker( lunch ) {
  lunch = lunch.toLowerCase();
  if( lunch !== "gravel" && ( lunch === "burrito" ||
    lunch === "tacos" ) ) {
    console.log( "Delicious." );
  }
}
mealChecker( "TACOS" );
Delicious.
```

We don't need `var` when we assign a value to `lunch`, since using `lunch` as an argument means we've already defined it as a variable local to the function.

switch

A `switch` statement works a lot like that unwieldy series of `else if` statements we tried out, but performs the same sort of comparisons in a more compact, sensible way. The syntax is a little different from the `if` syntaxes we're used to, however:

```
var theNumber = 5
switch( theNumber ) {
  case 1:
    console.log( "This is the number one." );
    break;
  case 2:
    console.log( "This is the number two." );
    break;
  case 3:
  case 4:
    console.log( "This is either three or four." );
    break;
  case 5:
    console.log( "This is the number five." );
}
```

There's a lot going on in there, so let's go through this one line by line. Before we do, remember that JavaScript doesn't care what we use for whitespace—all that indentation is something we're doing for the sake of readability, but it isn't *required*.

The first line is old hat by this point: we're defining a variable with the identifier `theNumber` and giving it a value of the number data type `5`—and since we're just tinkering, we won't worry about how `theNumber` isn't a very descriptive identifier, accurate though it may be.

The second line looks a little bit like the `if` statements we now know and hopefully love: the keyword `switch` followed by a set of parentheses and a pair of curly braces. `switch` differs from `if` in that we're not performing a comparison between the parentheses, though—instead, we're just passing along the information we want to compare, the same way we were passing a string to our function a few minutes ago. `switch( theNumber ) {}` only says that the variable `theNumber` is the value we want to compare within the `switch` statement—and that'll make sense in just a moment.

The third line—`case 1:`—is where we perform the actual comparison. The `case` keyword is followed by a value that gets compared to the value we passed along in the parentheses after the `switch` keyword, followed by a colon. So, the line `case 1:` is really saying, "If `theNumber` is equal to the number one, do the following." All the comparisons in a `switch` are strict—`case "1":` wouldn't match, since "`1`" would be a string type, not a number type.

```
switch( "1" ) {
  case 1:
    console.log( "This is the number one." );
    break;
  case "1":
    console.log( "This is the string '1'" );
  }
  This is the string '1'
```

The break statement then says, “We’ve found our match, so stop comparing.” It isn’t always the case that you’d want to stop the comparison right away; if you skip ahead a few lines, you’ll see that we’re checking against `case 3` and `case 4`, and if *either* of those comparisons match, we’re going on to `console.log` that the value is either three or four, then breaking after that. The reason this works is that a matching `case` inside `switch`, strictly speaking, tells JavaScript, “Run every line of code that follows the matching `case` *until you hit a* `break` *keyword* (or the end of the `switch`’s curly braces).”

To illustrate that behavior, let’s say we wanted to put together a function—for whatever reason—that accepts the current numeric day of the week (1-7), and spits out the names of all the days we’ve seen so far this week.

```
function daysPassedThisWeek( numericDay ) {
  console.log( "The following days have already
    happened this week:" );
  switch( numericDay ) {
    case 7:
      console.log( "Saturday" );
    case 6:
      console.log( "Friday" );
    case 5:
      console.log( "Thursday" );
    case 4:
      console.log( "Wednesday" );
    case 3:
      console.log( "Tuesday" );
    case 2:
      console.log( "Monday" );
    case 1:
      console.log( "Sunday" );
      break;
    default:
      console.log( "Wait, that isn’t a numbered day
        of the week." );
  }
}
```

```
daysPassedThisWeek( 3 ); /* Today is the third day
  of the week. */
The following days have already happened this week:
Tuesday
Monday
Sunday
```

Not the most useful function around, but you get the idea: when `case 3:` matches, all the `console.log` statements between there and the next `break` statement are executed, and we get a list of days.

You'll also notice something new in this `switch` statement: a `default` keyword.

The `default` keyword is the `else` of a `switch`, if that phrase could possibly make sense: in the event that none of the `case` values above it return `true`, the code that follows `default` will be executed.

```
daysPassedThisWeek( 75 );
The following days have already happened this week:
Wait, that isn't a numbered day of the week.
```

Since we're putting the `default` code last, we don't need a `break` afterwards—but we do need one *before* the `default`, or else that error will appear with the rest of our list. Though, if you'd prefer, the function above could be written with the `default` case coming first, followed by a `break`.

```
function daysPassedThisWeek( numericDay ) {
  console.log( "The following days have already
    happened this week:" );
  switch( numericDay ) {
    default:
      console.log( "Wait, that isn't a numbered day
        of the week." );
      break;
    case 7:
      console.log( "Saturday" );
      case 6:
        console.log( "Friday" );
      case 5:
        console.log( "Thursday" );
      case 4:
        console.log( "Wednesday" );
      case 3:
        console.log( "Tuesday" );
      case 2:
        console.log( "Monday" );
      case 1:
        console.log( "Sunday" );
    }
  }
  daysPassedThisWeek( 5 ); /* Today is the fifth day
    of the week. */
  The following days have already happened this week:
  Thursday
  Wednesday
  Tuesday
  Monday
  Sunday
```

`switch` is a weird one, for sure, but there are a few situations where performing a series of comparisons against a single object will make a lot of sense. For instance, imagine a script that accepts keyboard input and uses it to move a sprite—many games ask you to use either the arrow keys or the *A* and *D* keys to move left and right. In JavaScript, key presses are represented by an event object—which we'll discuss a little later on—with a property containing a numeric value that corresponds to the key that was pressed.

```
function movePlayer( keyCode ) {
  switch( keyCode ) {
    case 65: // Keycode for the A key
    case 37: // Keycode for the left arrow
      moveLeft();
      break;
    case 68: // Keycode for the D key
    case 39: // Keycode for the right arrow
      moveRight();
  }
}
```

You could write this as a series of `if` statements as well, but adding new controls over time would mean chaining one `if` after another, all performing comparisons against the same `keyCode` object. `switch` ends up being a much more flexible—and DRY—way to go about it.

THAT ESCALATED QUICKLY

From "if this, do that" to methods of expressing incredibly complex logic, all wrapped up in the humble `if` and `switch` keywords. It's a lot to keep in your head, but again: we don't have to. Instead, we can walk away from this chapter knowing that there's *some* way to express whatever conditional logic we might need—and if you can't remember the exact syntax off the top of your head, well, this chapter isn't going anywhere.

Conditional statements allow us to selectively execute code, but there's more to control flow than that. Programming frequently involves performing a set of tasks over and over again—for example, iterating over all elements of a certain type within a page and checking a common attribute against an expected value, or iterating over all the items in an array and checking their values against a conditional statement. In order to do those kinds of things, we'll need to learn about loops.

4 LOOPS

LOOPS ALLOW US to repeat lines of code until a certain set of conditions are met. It's another simple concept on the surface, but one that allows us to do a surprising amount of work.

for

You'll use for loops in situations where you'll be running a loop for a known number of times, or *iterations*. (By "known," I don't mean that *we*, the brains between the keyboards and chairs, necessarily know how many times we'll need to go through a loop in advance; I just mean that we'll be looping through a known quantity.)

The syntax for a loop is a little tricky, since we're packing a lot of information into just a few characters. The basics are almost expected by this point: a for keyword, followed by a set of parentheses, followed by a pair of curly braces that we want our loop to execute however many times.

But it's the syntax between the parentheses that's unlike anything we've encountered so far. A for loop accepts three expressions: the *initialization*, the *condition*, and the somewhat redundantly named *final expression*, all separated by semicolons.

```
for( var i = 0; i < 3; i++ ) {
  console.log( "This loop will run three times.")
}
(3) This loop will run three times.
```

The initialization is almost always used for one thing: to initialize a variable that will act as a counter. It gets initialized the same as any other variable, with a `var` keyword, an identifier, and an assignment. You'll see a lot of these counter variables with the identifier `i`, which stands for "iteration." It *does* break the rule against giving identifiers single-character names, but it's a well-established convention. Since JavaScript starts indexing things at zero, it's a good idea for us to always start from zero as well—that way we don't get into the habit of counting up from one or run into any mismatched counters in our scripts.

The condition is how we define the point at which the loop stops. So, we're defining `i` as starting at zero, and we want the loop to run for as long as `i` is less than three.

The final expression is the statement we want executed at the end of every iteration through the loop—so, this is where we tick the `i` variable's value up by one. If you think all the way back to the mathematical operators we learned about earlier, you'll recall that the `++` syntax increments a value by one: `i++` as the final expression says to increase i by one every time the loop finishes.

So, in plain English, the `for` syntax above says this: "Start `i` at zero. Only run the following code if `i` is smaller than three, and add one to `i` after every loop."

One of the most common uses for a `for` loop is iterating over each item in an array. As we learned in the last chapter, arrays come with a property for determining how many items it contains—the `.length` property—so we'll always be dealing with a known quantity. If we use the array's length in the condition, we get a loop that iterates as many times as there are items in our array:

```
var loopArray = ["first", "second", "third"];
for( var i = 0; i < loopArray.length; i++ ) {
  console.log( "Loop." );
}
(3) Loop.
```

And if we add an item to our test array, the number of iterations changes to match:

```
var loopArray = ["first", "second", "third", 4];

for( var i = 0; i < loopArray.length; i++ ) {
  console.log( "Loop." );
}
(4) Loop.
```

Cool—we can make a thing that runs arbitrary code as many times as we have items in an array. I know that isn't particularly useful or exciting on the surface, but what's *very* exciting (I don't get out much) is that `i` is a plain ol' number type variable that's available to us on every iteration of the loop, and since we're counting from zero, it contains a value that corresponds to the index of each item in our array. A `for` loop gives us a method of iterating over the data in an array all at once:

```
var names = [ "Ed", "Al" ];

for( var i = 0; i < names.length; i++ ) {
  var name = names[ i ];
  console.log( "Hello, " + name + "!" );
}
Hello, Ed!
Hello, Al!
```

Now we can reach back into the `names` variable that we're iterating over, and using i as the index, we can tap into each item inside the array.

We didn't have to initialize a new name variable here, of course—we could have used names[i] inside our console.log and had everything work the same way. Storing the array's data in a variable on each iteration is a good idea if you're likely to access that data multiple times during your loop, just for the sake of convenience.

for/in

for/in loops start out the same way as the for loops above, using a for keyword, a set of parentheses, and a pair of curly braces containing whatever code we want to iterate over. Likewise, you'll use the for/in syntax to iterate over multiple items—but not necessarily the way we would want to with an array. for/in is used to iterate over the properties of an object—but in an arbitrary order, not a sequential one.

Instead of an initialization, condition, and final expression, a for/in loop starts with us initializing a variable that will correspond with the keys in our object, followed by the in keyword, and the object we want to iterate over:

```
var nameObject = {
  "first": "Mat",
  "last": "Marquis"
};

for( var name in nameObject ) {
  console.log( "Loop." );
}
(2) Loop.
```

Unlike iterating over an array using for—where we have a handy i variable at our disposal—we now have to do a little more work to figure out what strings are used as keys in our object. Those keys get assigned to the name variable we initialized in the parentheses.

```
var nameObject = {
  "first": "Mat",
  "last": "Marquis"
};

for( var name in nameObject ) {
  console.log( name );
}
first
last
```

Not too useful on the surface, but just like a regular `for` loop gave us a number data type we could use to access our data on each iteration, `for`/`in` gives us the string we need to access the data in an object:

```
var fullName = {
  "first": "Mat",
  "last": "Marquis"
};

for( var name in fullName ) {
  console.log( name + ": " + fullName[ name ] );
}
first: Mat
last: Marquis
```

You'll notice we're using bracket notation instead of dot notation here—that's because we have to. If we were to try to access `fullName.name`, we'd be trying to access exactly that: `fullName.name`—a property with the key `name` inside `fullName`—instead of a property with a key that matches the string that `name` *contains*.

Now, you might be thinking, "But if pretty much everything is an object—sort of—does that mean we can use `for`/`in` to iterate over an array?" You can, in fact. Arrays behave just like any other objects using `for`/`in`, but `for`/`in` isn't quite as array-friendly as a regular `for` loop. For one thing, we can't guarantee that `for`/`in` will iterate over an array in sequential order—and that might be okay, depending on what you're doing.

The bigger problem is that `for`/`in` has a catch that a regular `for` loop doesn't: since pretty much everything is an object—and you can add properties to any object—that means `for`/`in` can end up iterating over properties we never meant for it to know about.

I've previously mentioned that everything in JavaScript has "built-in" methods and properties—how a string, even though we only define it as a handful of characters, will come with properties like `.length` and methods like `.toLowerCase()`.

These methods and properties aren't completely buried in the dark recesses of JavaScript. We can see—and even *change*—the methods and properties that come attached to data types, arrays, objects, and so on.

Prototypal inheritance

Most objects have an internal property named `prototype` that contains those built-in properties, but accessing it for ourselves is a little strange. When we access a property on an object—even one we've created ourselves—the JavaScript runtime first checks to see if that property is something *we* defined. If not, it looks for that key as a `prototype` property for that object's constructor—a sort of overarching template that JavaScript follows whenever dealing with a certain type of object. All strings, for example, inherit all the `prototype` properties and methods from the `String` constructor. With *most* browser tools, typing `String.prototype` into your console will show you all of those built-in properties.

If your first instinct is that JavaScript allowing us to override built-in methods is a little scary, you're absolutely right. The `toString` method, for example, is a pretty cut-and-dry way of converting any object to a string, but with a few lines of code we can overwrite it, and make that method do whatever we want.

```
var myObject = {};
var otherObject = {};
```

```
myObject.toString();
"[object Object]"

myObject.toString = function() {
  console.log( "I just broke JavaScript a little." );
};

myObject.toString();
I just broke JavaScript a little.

otherObject.toString();
"[object Object]"
```

That's kind of a scary amount of power, but it gets scarier.

We can access a constructor's `prototype` from *any* object of that type by using the `__proto__` property—a reference to the `prototype` for *all* objects of that same type. It isn't available in all browsers just yet, but that's okay—we probably shouldn't be messing with it anyway. `__proto__` allows us to add, remove, and completely change how JavaScript's built-in properties work on any one object—and in doing so, change the set of properties and methods that are built into *all related objects*.

```
var myObject = {};
var unrelatedObject = {};

myObject.toString();
"[object Object]"

myObject.__proto__.toString = function() {
  console.log( "I just broke JavaScript a LOT." );
};

myObject.toString();
I just broke JavaScript a LOT.

unrelatedObject.toString(); // Uh oh.
I just broke JavaScript a LOT.
```

Here, we managed to change the `toString` method not just on `myObject`, but on *all* objects, by changing it at the `prototype` level. In a vacuum like this, where we can see all our code at a glance, it may not seem like an especially terrifying prospect—but by tampering with the way JavaScript does things, we run a huge risk of breaking things on a real site.

Luckily, you won't see this much. By the time anyone has advanced enough in their script-writing to think they have a need to mess with `prototype`, they've learned not to. What you will see on occasion are *additions* to `prototype`—methods and functions added to `prototype` so they're available to all objects of the same type.

We can add methods and properties to *all* objects of a certain type by making changes to the `prototype` property of a constructor directly—we just can't change the properties that have already been defined. This works the way you might expect: making additions to `String.prototype` works the way you'd add properties on an object you created yourself.

Now, adding methods to `prototype` isn't a great idea either—especially when it comes to `for`/`in` loops. But for the sake of argument, let's say you'll frequently need to check objects for the presence of a key with the identifier `name`. In theory, we could just add a method to *all* objects by adding a new method to the `Object` constructor's `prototype` property. We won't need to use `__proto__` here, since we're not looking to change `Object.prototype` by way of an object data type itself:

```
var firstObject = {
  "foo" : false
};
undefined

var secondObject = {
  "name" : "Hawkeye",
  "location" : "Maine"
};
undefined
```

```
Object.prototype.containsAName = function() {
  var result = false;
  for( var key in this ) {
    if( key === "name" ) {
      result = true;
    }
  }
  return result;
}
function Object.containsAName()

firstObject.containsAName();
false

secondObject.containsAName();
true
```

Once you've added something to an object's prototype, that property or method will be available to all instances of that data type.

Now, don't commit that code to memory—it's an incredibly convoluted way to see whether an object contains the key `name`. It works well enough, but in addition to being a strange way to handle a simple task, it comes with an unintended side-effect: an object's built-in properties aren't *enumerable*, meaning that they don't show up when we loop through an object's properties using `for`/`in`. When we add *new* properties to `prototype`, however, they *are* enumerable:

```
Object.prototype.containsAName = function() {
  var result = false;
  for( var key in this ) {
    if( key === "name" ) {
      result = true;
    }
  }
  return result;
}
function Object.containsAName()
```

```
var newObject = { "name": "BJ" };
for( var key in newObject ) {
  console.log( key );
}
name
containsAName
```

Now every time we run a `for`/`in` loop, our `containsAName` method is going to show up—certainly not ideal, and probably reason enough to leave `prototype` alone. Whenever we tinker with `prototype`, we're making global changes to JavaScript's internals—adding a method or property means `for`/`in` loops throughout all the scripts on our page could end up behaving unexpectedly.

That means we have to look at the issue from the other side—what happens to our `for`/`in` loops when someone else starts tinkering with `prototype`? A page can contain scripts written by multiple developers, third-party scripts from external sources, and so on—we can't be certain that our `for`/`in` loops won't be affected by unexpected enumerable properties.

hasOwnProperty

The reason for this treacherous journey through `prototype` was two-fold: first, to teach you a little bit about `prototype`, since we were in the neighborhood and all. Second, to introduce you to the `hasOwnProperty` method, which we can use to safeguard our `for`/`in` loops against unexpected enumerable properties on `prototype`:

```
Object.prototype.containsAName = function() {
  var result = false;
  for( var key in this ) {
    if( key === "name" ) {
      result = true;
    }
  }
  return result;
}
```

```
function Object.containsAName()

var mysteryObject = {
  "name" : "Frank"
}

mysteryObject.hasOwnProperty( "name" );
true

mysteryObject.hasOwnProperty( "containsAName" );
false
```

It just so happens that we can use `hasOwnProperty` to do what we were trying to do when we were messing with `prototype`: determine whether an object we created contains a certain property. But even more importantly—at least, for the sake of our loops—`hasOwnProperty` *doesn't* apply to properties inherited from `prototype`. We can use it to safeguard our `for`/`in` loops against misguided `prototype` shenanigans.

```
Object.prototype.customPrototypeMethod = function() {
  console.log( "Hello again." );
};
function Object.customPrototypeMethod()

var swamp = {
  "bunk1" : "Hawkeye",
  "bunk2" : "BJ",
  "bunk3" : "Frank"
};
undefined

for( var bunk in swamp ) {
  console.log( swamp[ bunk ] );
}
Hawkeye
BJ
Frank
function Object.containsAName()
```

```
for( var bunk in swamp ) {
  if( swamp.hasOwnProperty( bunk ) ) {
    console.log( swamp[ bunk ] );
  }
}
Hawkeye
BJ
Frank
```

With `hasOwnProperty` in place, no additions to `prototype` can change the results we expect from our `for`/`in` loops. After all, we won't always be able to guarantee that we control every line of code that makes it into our websites, or know for certain whether another developer decided to tinker with a constructor's `prototype`.

while

After that brief excursion into `prototype`, the syntax for `while` loops is going to be a refreshing change of pace. Like all the others, it starts with a keyword—`while`—followed by a set of parentheses and a set of curly braces. The only thing we'll put between the parentheses of a `while` loop is a *condition*—and just like the keyword implies, the loop will continue to execute for as long as that condition evaluates to *`true`*.

```
var i = 0;
while( i < 3 ) {
  console.log( "Loop." );
  i++;
}
(3) Loop
```

The code above is really just another way of writing our first `for` loop. Instead of putting the initialization, condition, and final expression between the parentheses, we're creating the counter variable before the loop and incrementing the counter inside the loop.

This isn't a common case for `while`, however—we could write this much more concisely using `for`, after all. We'll use `while` when we *don't* have any way of measuring the number of iterations necessary, and instead want to continue to run the loop until a certain condition is met. For example, the following snippet of code will continually generate a number between zero and nine, only stopping when that random number equals three:

```
var random = Math.floor( Math.random() * 10 );

while( random !== 3 ){
  console.log( "Nope, not " + random );
  var random = Math.floor( Math.random() * 10 );
}
console.log( "Got it!" );
Nope, not 5
Nope, not 9
Nope, not 2
Got it!
```

If we run this code again, that `while` loop could run any number of times before continuing on to the `console.log` that follows it—or `random` could contain a `3` right off the bat, and the loop will never run at all.

do/while

`do`/`while` loops serve largely the same purpose as `while` loops: to iterate over a loop until a given condition evaluates to *`true`*, as many times as is needed. The syntax is a little different—in fact, compared to the conditional logic you've seen so far, `do`/`while` loops look a little backwards. We start with a `do` keyword, immediately followed by a set of curly braces—no parentheses, no conditions—containing the code we want to iterate over. *After* the curly braces, we use the `while` keyword and parentheses containing the condition; as long as that condition evaluates to *`true`*, the loop will keep on running.

```
var i = 0;

do {
  console.log( "Loop." );
  i++;
} while (i < 3);
(3) Loop.
```

There's really only one difference between a `do`/`while` and a plain ol' `while` loop: the contents of a `while` loop may never run at all—as in our random number example above—but the code in a `do`/`while` loop will always execute at least once.

Instead of evaluating the condition then deciding whether or not to run the code, the way all the other loops work, `do`/`while` will execute the code, then stop the loop *after* the condition is found. If we were to write our random-number guessing game above using `do`/`while`, the code will run even once the condition is met:

```
do {
  var random = Math.floor( Math.random() * 10 );
  console.log( "Is it... " + random + "?" );
} while( random !== 3 );
console.log( "Got it!" );
Is it... 7?
Is it... 9?
Is it... 6?
Is it… 3?
Got it!
```

Now we don't have to generate a random number before the loop *and* regenerate it on each iteration: since the code inside the `do` curly braces will always be executed before the condition is evaluated, we can generate the random number for the first time *and* regenerate it on each subsequent iteration on that one line. We then log each "guess" to our console—and since the code executes before the condition is checked, that will include the matching number. Now that the condition matches, the loop stops, and we move on.

continue AND break

All of the loop syntaxes handle both iteration and termination—they all accept some form of condition for stopping the loop. If we need more finely grained control, we can do a little steering within our loops using the `continue` and `break` keywords.

To play with these keywords a little, let's start with a `for` loop that counts from zero to four:

```
for( var i = 0; i < 5; i++ ) {
  console.log( i );
}
0
1
2
3
4
```

`continue` allows us to skip ahead to the next iteration of a loop without executing any of the code inside the loop that follows the `continue` statement.

```
for( var i = 0; i < 5; i++ ) {
  if( i === 2 ) {
    continue;
  }
  console.log( i );
}
0
1
3
4
```

This loop skips the third `console.log` by using `continue` when `i` is equal to two—which skips the third iteration, remember, since we start counting at zero. We never see a `2` in our console.

`break`—a deeply satisfying keyword to type after a few hours of programming, in my experience—not only stops the current iteration of a loop, but stops the loop completely:

```
for( var i = 0; i < 5; i++ ) {
  if( i === 2 ) {
    break;
  }
  console.log( i );
}
0
1
```

Upon encountering a `break`, JavaScript reacts the same way it does when it reaches a loop's built-in condition for termination: it stops iterating over the loop completely. As a matter of fact, we don't *need* to build in that condition at all—we can write a loop that iterates forever by default, and control its behavior using `break`.

```
var concatString = "a";
while( true ) {
  concatString = concatString + "a";

  if( concatString.length === 5 ) {
    break;
  }
  console.log( concatString );
}
aa
aaa
aaaa
```

`true` can't possibly stop evaluating to `true`, so using it as a `while` condition means the loop has to run forever—unless we use `break` to stop it, once our string reaches five characters long.

INFINITE LOOPS

We're standing on the precipice of a very touchy subject for developers of any language. What happens if we pass `while` the keyword `true` as a condition, or write a `for` loop that counts upwards from zero, but only stops iterating if `i` is equal to `-1`? We've just created an infinite loop, and here be dragons.

Upon encountering an infinite loop, some modern browsers eventually offer you the option of aborting the script—but only newer browsers, and not always. More often than not, an infinite loop means a browser crash.

It happens to the best of us, and there's no serious harm done (as long as the offending code didn't make it out to a production website). Close and reopen your browser and you'll be back to business as usual—just be sure to fix the infinite loop before you attempt to run your script again.

PUTTING IT ALL TOGETHER

We've covered a lot of ground so far. Now that we have a sense of how JavaScript handles data, logic, and loops, we can start putting it all together into something more useful than a couple of lines in our developer consoles. It's time to write a script in the context it was meant to occupy: a real web page.

5 DOM SCRIPTING

BEFORE WE CAN DO ANYTHING with a page, we have to first revisit something we touched on near the start: the Document Object Model. There are two purposes to the DOM: providing JavaScript with a map of all the elements on our page, and providing us with a set of methods for accessing those elements, their attributes, and their contents.

The "object" part of Document Object Model should make a lot more sense now than it did the first time the DOM came up, though: the DOM is a representation of a web page in the form of an object, made up of properties that represent each of the document's child elements and subproperties representing each of those elements' child elements, and so on. It's objects all the way down.

`window`: THE GLOBAL CONTEXT

Everything we do with JavaScript falls within the scope of a single object: `window`. The `window` object represents, predictably enough, the entire browser window. It contains the entire DOM, as well as—and this is the tricky part—the whole of JavaScript.

When we first talked about variable scope, we touched on the concept of there being "global" and "local" scopes, meaning that a variable could be made available either to every part of our scripts or to their enclosing function alone.

The `window` object *is* that global scope. All of the functions and methods built into JavaScript are built off of the `window` object. We don't have to reference `window` constantly, of course, or you would've seen a lot of it before now—since `window` is the global scope, JavaScript checks `window` for any variables we haven't defined ourselves. In fact, the `console` object that you've hopefully come to know and love is a method of the `window` object:

```
window.console.log
function log() { [native code] }
```

It's hard to visualize globally vs. locally scoped variables before knowing about `window`, but much easier after: when we introduce a variable to the global scope, we're making it a property of `window`—and since we don't explicitly have to reference `window` whenever we're accessing one of its properties or methods, we can call that variable anywhere in our scripts by just using its identifier. When we access an identifier, what we're really doing is this:

```
function ourFunction() {
  var localVar = "I'm local.";
  globalVar = "I'm global.";

  return "I'm global too!";
};
undefined

window.ourFunction();
I'm global too!
```

```
window.localVar;
undefined

window.globalVar;
I’m global.
```

The DOM’s entire representation of the page is a property of `window`: specifically, `window.document`. Just entering `window.document` in your developer console will return all of the markup on the current page in one enormous string, which isn’t particularly useful—but everything on the page can be accessed as subproperties of `window.document` the exact same way. Remember that we don’t need to specify `window` in order to access its `document` property—`window` is the only game in town, after all.

```
document.head
<head>...</head>

document.body
<body>...</body>
```

Those two properties are themselves objects that contain properties that are objects, and so on down the chain. (“Everything is an object, kinda.”)

USING THE DOM

The objects in `window.document` make up JavaScript’s map of the document, but it isn’t terribly useful for us—at least, not when we’re trying to access DOM nodes the way we’d access any other object. Winding our way through the `document` object manually would be a huge headache for us, and that means our scripts would completely fall apart as soon as any markup changed.

But `window.document` isn't just a representation of the page; it also provides us with a smarter API for accessing that information. For instance, if we want to find every `p` element on a page, we don't have to write out a string of property keys—we use a helper method built into `document` that gathers them all into an array-like list for us. Open up any site you want—so long as it likely has a paragraph element or two in it—and try this out in your console:

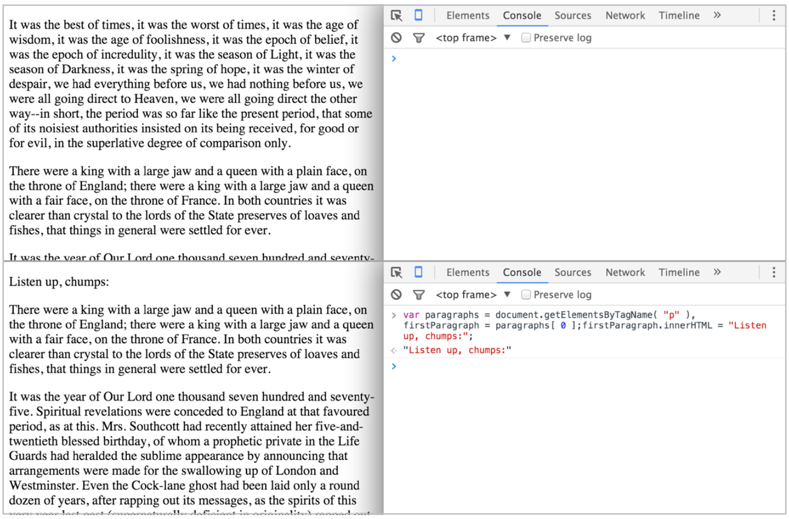
```
document.getElementsByTagName( "p" );
[<p>...</p>, <p>...</p>, <p>...</p>, <p>...</p>]
```

Since we're dealing with such familiar data types, we already have some idea how to work with them:

```
var paragraphs = document.getElementsByTagName( "p" );
undefined

paragraphs.length
4

paragraphs[ 0 ];
<p>...</p>
```

But DOM methods don't give us arrays, strictly speaking. Methods like `getElementsByTagName` return "node lists," which behave a lot like arrays. Each item in a `nodeList` refers to an individual node in the DOM—like a `p` or a `div`—and will come with a number of DOM-specific methods built in. For example, the `innerHTML` method will return any markup a node contains—elements, text, and so on—as a string:

```
var paragraphs = document.getElementsByTagName( "p" ),
  lastIndex = paragraphs.length - 1, /* Use the length
  of the `paragraphs` node list minus 1 (because of
  zero-indexing) to get the last paragraph on the page
  */
  lastParagraph = paragraphs[ lastIndex ];

lastParagraph.innerHTML;
And that's how I spent my summer vacation.
```


It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it was the epoch of belief, it was the epoch of incredulity, it was the season of Light, it was the season of Darkness, it was the spring of hope, it was the winter of despair, we had everything before us, we had nothing before us, we were all going direct to Heaven, we were all going direct the other way--in short, the period was so far like the present period, that some of its noisiest authorities insisted on its being received, for good or for evil, in the superlative degree of comparison only.

There were a king with a large jaw and a queen with a plain face, on the throne of England; there were a king with a large jaw and a queen with a fair face, on the throne of France. In both countries it was clearer than crystal to the lords of the State preserves of loaves and fishes, that things in general were settled for ever.

It was the year of Our Lord one thousand seven hundred and seventy-

Listen up, chumps:

There were a king with a large jaw and a queen with a plain face, on the throne of England; there were a king with a large jaw and a queen with a fair face, on the throne of France. In both countries it was clearer than crystal to the lords of the State preserves of loaves and fishes, that things in general were settled for ever.

It was the year of Our Lord one thousand seven hundred and seventy-five. Spiritual revelations were conceded to England at that favoured period, as at this. Mrs. Southcott had recently attained her five-and-twentieth blessed birthday, of whom a prophetic private in the Life Guards had heralded the sublime appearance by announcing that arrangements were made for the swallowing up of London and Westminster. Even the Cock-lane ghost had been laid only a round dozen of years, after rapping out its messages, as the spirits of this

```
> var paragraphs = document.getElementsByTagName( "p" ),
  firstParagraph = paragraphs[ 0 ];firstParagraph.innerHTML = "Listen
  up, chumps:";
<· "Listen up, chumps:"
>
```

FIG 5.1: First drafts are always tough.

The same way these methods give us access to information on the rendered page, they allow us to *alter* that information, as well. For example, the `innerHTML` method does this the same way we'd change the value of any other object: a single equals sign, followed by the new value.

```
var paragraphs = document.getElementsByTagName( "p" ),
  firstParagraph = paragraphs[ 0 ];
firstParagraph.innerHTML = "Listen up, chumps:";
"Listen up, chumps:"
```

JavaScript's map of the DOM works both ways: `document` is updated whenever any markup changes, and our markup is updated whenever anything within `document` changes (**FIG 5.1**).

Likewise, the DOM API gives us a number of methods for creating, adding, and removing elements. They're all more or less spelled out in plain English, so even though things can seem a little verbose, it isn't too hard to break down.

DOM SCRIPTING

Before we get started, let's abandon our developer console for a bit. Ages ago now, we walked through setting up a bare-bones HTML template that pulls in a remote script, and we're going to revisit that setup now. Between the knowledge you've gained about JavaScript so far and an introduction to the DOM, we're done with just telling our console to parrot things back to us—it's time to build something.

We're going to add a "cut" to an index page full of text—a teaser paragraph followed by a link to reveal the full text. We're not going to make the user navigate to another page, though. Instead, we'll use JavaScript to show the full text on the same page.

Let's start by setting up an HTML document that links out to an external stylesheet and external script file—nothing fancy. Both our stylesheet and script files are empty with .css and .js extensions, for now—I like to keep my CSS in a /css subdirectory and my JavaScript in a /js subdirectory, but do whatever makes you most comfortable.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <link rel="stylesheet" type="text/css"
      href="css/style.css">
  </head>
  <body>

    <script src="js/script.js"></script>
  </body>
</html>
```

We're going to populate that page with several paragraphs of text. Any ol' text you can find laying around will do, including—with apologies to the content strategists in the audience—a little old-fashioned lorem ipsum. We're just mocking up a quick article page, like a blog post.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <link rel="stylesheet" type="text/css"
      href="css/style.css">
  </head>
  <body>
    <h1>JavaScript for Web Designers</h1>

    <p>In all fairness, I should start this book
      with an apology—not to you, reader, though I
      don’t doubt that I’ll owe you at least one by
      the time we get to the end. I owe JavaScript a
      number of apologies for the things I’ve said
      to it during the early years of my career,
      some of which were strong enough to etch
      glass.</p>

    <p>This is my not-so-subtle way of saying that
      JavaScript can be a tricky thing to learn.</p>

    [ … ]

    <script src="js/script.js"></script>
  </body>
</html>
```

Feel free to open up the stylesheet and play with the typography, but don’t get too distracted. We’ll need to write a little CSS later, but for now: we’ve got scripting to do.

We can break this script down into a few discrete tasks: we need to add a Read More link to the first paragraph, we need to hide all the `p` elements apart from the first one, and we need to reveal those hidden elements when the user interacts with the Read More link.

We’ll start by adding that Read More link to the end of the first paragraph. Open up your still-empty script.js file and enter the following:

```
var newLink = document.createElement( "a" );
```

First, we're intializing the variable `newLink`, which uses `document.createElement( "a" )` to—just like it says on the tin—create a new `a` element. This element doesn't really exist anywhere yet—to get it to appear on the page we'll need to add it manually. First, though, `<a></a>` without any attributes or contents isn't very useful. Before adding it to the page, let's populate it with whatever information it needs.

We could do this *after* adding the link to the DOM, of course, but there's no sense in making multiple updates to the element on the page instead of one update that adds the final result—doing all the work on that element before dropping it into the page helps keep our code predictable.

Making a single trip to the DOM whenever possible is also better for performance—but performance micro-optimization is easy to obsess over. As you've seen, JavaScript frequently offers us multiple ways to do the same thing, and one of those methods may *technically* outperform the other. This invariably leads to "excessively clever" code—convoluted loops that require in-person explanations to make any sense at all, just for the sake of shaving off precious picoseconds of load time. I've done it; I still catch myself doing it; but you should try not to. So while making as few round-trips to the DOM as possible is a good habit to be in for the sake of performance, the main reason is that it keeps our code readable and predictable. By only making trips to the DOM when we really need to, we avoid repeating ourselves and we make our interaction points with the DOM more obvious for future maintainers of our scripts.

So. Back to our empty, attribute-less `<a></a>` floating in the JavaScript ether, totally independent of our document.

Now we can use two other DOM interfaces to make that link more useful: `setAttribute` to give it attributes, and `innerHTML` to populate it with text. These have a slightly different syntax. We can just assign a string using `innerHTML`, the way we'd assign a value to any other object. `setAttribute`, on the other hand, expects two arguments: the attribute *and* the value we want for that attribute, in that order. Since we don't actually plan to have this link go anywhere, we'll just set a hash as the `href`—a link to the page you're already on.

```
var newLink = document.createElement( "a" );

newLink.setAttribute( "href", "#" );
newLink.innerHTML = "Read more";
```

You'll notice we're using these interfaces on our stored reference to the element instead of on `document` itself. *All* the DOM's nodes have access to methods like the ones we're using here—we only use `document.getElementsByTagName( "p" )` because we want to get all the paragraph elements in the document. If we only wanted to get all the paragraph elements inside a certain `div`, we could do the same thing with a reference to that `div`—something like `ourSpecificDiv.getElementsByTagName( "p" );`. And since we'll want to set the `href` attribute and the inner HTML of the link we've created, we reference these properties using `newLink.setAttribute` and `newLink.innerHTML`.

Next: we want this link to come at the end of our first paragraph, so our script will need a way to reference that first paragraph. We already know that `document.getElementsByTagName( "p" )` gives us a node list of all the paragraphs in the page. Since node lists behave like arrays, we can reference the first item in the node list one by using the index `0`.

```
var newLink = document.createElement( "a" );
var allParagraphs = document.getElementsByTagName(
  "p" );
var firstParagraph = allParagraphs[ 0 ];

newLink.setAttribute( "href", "#" );
newLink.innerHTML = "Read more";
```

For the sake of keeping our code readable, it's a good idea to initialize our variables up at the top of a script—even if only by initializing them as `undefined` (by giving them an identifier but no value)—if we plan to assign them a value later on. This way we know all the identifiers in play.

So now we have everything we need in order to append a link to the end of the first paragraph: the element that we want to append (`newLink`) and the element we want to append it to (`firstParagraph`).

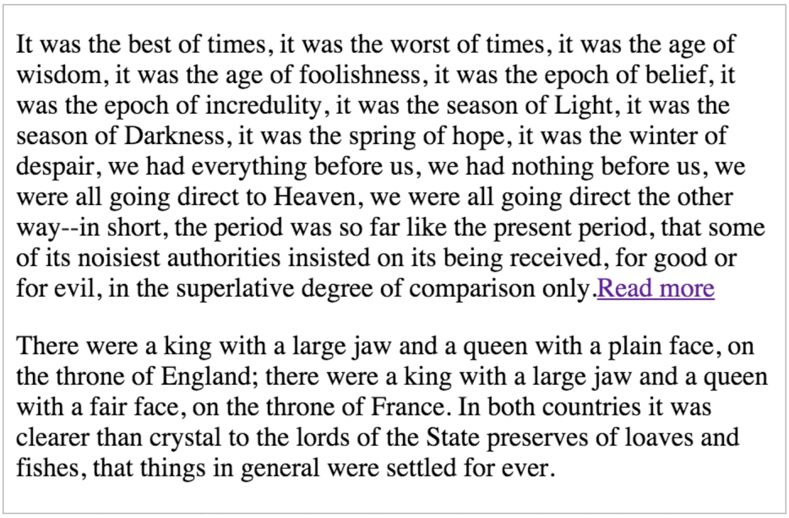
One of the built-in methods on all DOM nodes is `appendChild`, which—as the name implies—allows us to append a child element to that DOM node. We'll call that `appendChild` method on our saved reference to the first paragraph in the document, passing it `newLink` as an argument.

```
var newLink = document.createElement( "a" );
var allParagraphs = document.getElementsByTagName(
  "p" );
var firstParagraph = allParagraphs[ 0 ];

newLink.setAttribute( "href", "#" );
newLink.innerHTML = "Read more";

firstParagraph.appendChild( newLink );
```

Now—finally—we have something we can point at when we reload the page. If everything has gone according to plan, you'll now have a Read More link at the end of the first paragraph on the page. If everything hasn't gone according to plan—because of a misplaced semicolon or mismatched parentheses, for example—your developer console will give you a heads-up that something has gone wrong, so be sure to keep it open.

Pretty close, but a little janky-looking—our link is crashing into the paragraph above it, since that link is `display: inline` by default (**FIG 5.2**).

We have a couple of options for dealing with this: I won't get into all the various syntaxes here, but the DOM also gives us access to *styling* information about elements—though, in its most basic form, it will only allow us to read and change styling information associated with a `style` attribute. Just to get a feel for how that works, let's change the link to `display: inline-block` and add a few pixels of margin to the


It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it was the epoch of belief, it was the epoch of incredulity, it was the season of Light, it was the season of Darkness, it was the spring of hope, it was the winter of despair, we had everything before us, we had nothing before us, we were all going direct to Heaven, we were all going direct the other way--in short, the period was so far like the present period, that some of its noisiest authorities insisted on its being received, for good or for evil, in the superlative degree of comparison only.Read more

There were a king with a large jaw and a queen with a plain face, on the throne of England; there were a king with a large jaw and a queen with a fair face, on the throne of France. In both countries it was clearer than crystal to the lords of the State preserves of loaves and fishes, that things in general were settled for ever.

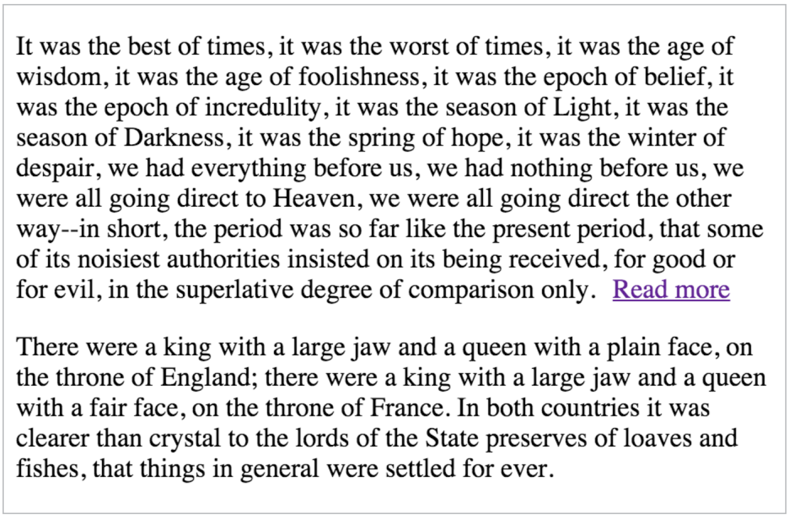
FIG 5.2: Well, it's a start.

left side, so it isn't colliding with our text. Just like setting attributes, we'll do this before we add the link to the page:

```
var newLink = document.createElement( "a" );
var allParagraphs = document.getElementsByTagName(
  "p" );
var firstParagraph = allParagraphs[ 0 ];

newLink.setAttribute( "href", "#" );
newLink.innerHTML = "Read more";
newLink.style.display = "inline-block";
newLink.style.marginLeft = "10px";

firstParagraph.appendChild( newLink );
```

Well, adding those lines *worked,* but not without a couple of catches. First, let's talk about that syntax (**FIG 5.3**).

Remember that identifiers can't contain hyphens, and since everything is an object (sort of), the DOM references styles in


It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it was the epoch of belief, it was the epoch of incredulity, it was the season of Light, it was the season of Darkness, it was the spring of hope, it was the winter of despair, we had everything before us, we had nothing before us, we were all going direct to Heaven, we were all going direct the other way--in short, the period was so far like the present period, that some of its noisiest authorities insisted on its being received, for good or for evil, in the superlative degree of comparison only. Read more

There were a king with a large jaw and a queen with a plain face, on the throne of England; there were a king with a large jaw and a queen with a fair face, on the throne of France. In both countries it was clearer than crystal to the lords of the State preserves of loaves and fishes, that things in general were settled for ever.

FIG 5.3: Now we're talking.

object format as well. Any CSS property that contains a hyphen instead gets camel-cased: `margin-left` becomes `marginLeft`, `border-radius-top-left` becomes `borderRadiusTopLeft`, and so on. Since the *value* we set for those properties is a string, however, hyphens are just fine. A little awkward and one more thing to remember, but this is manageable enough—certainly no reason to avoid styling in JavaScript, if the situation makes it absolutely necessary.

A better reason to avoid styling in JavaScript is to maintain a separation of behavior and presentation. JavaScript is our "behavioral" layer the way CSS is our "presentational" layer, and seldom the twain should meet. Changing styles on a page shouldn't mean rooting through line after line of functions and variables, the same way we wouldn't want to bury styles in our markup. The people who might end up maintaining the styles for the site may not be completely comfortable editing JavaScript—and since changing styles in JavaScript means we're indirectly adding styles via `style` attributes, whatever we write in a script is going to override the contents of a stylesheet by default.

We can maintain that separation of concerns by instead using `setAttribute` again to give our link a class. So, let's scratch out those two styling lines and add one setting a class in their place.

```
var newLink = document.createElement( "a" );
var allParagraphs = document.getElementsByTagName(
  "p" );
var firstParagraph = allParagraphs[ 0 ];

newLink.setAttribute( "href", "#" );
newLink.setAttribute( "class", "more-link" );
newLink.innerHTML = "Read more";

firstParagraph.appendChild( newLink );
```

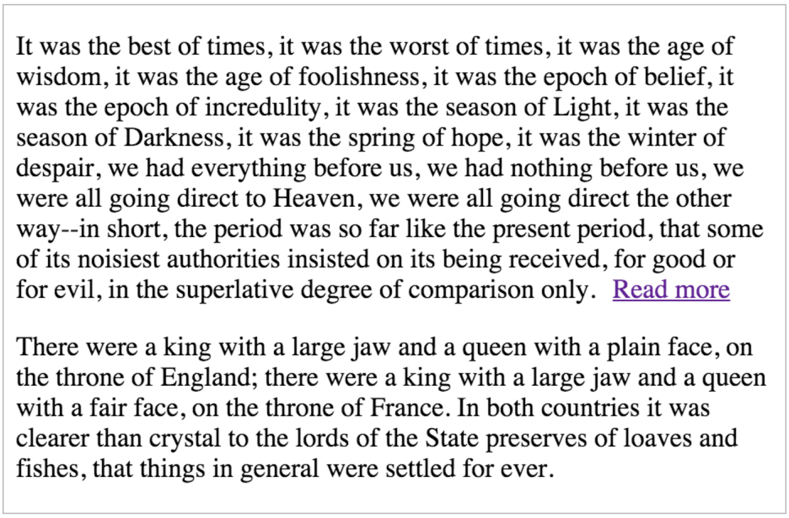
Now we can style `.more-link` in our stylesheets as usual:

```
.more-link {
  display: inline-block;
  margin-left: 10px;
}
```

Much better (**FIG 5.4**). It's worth keeping in mind for the future that using `setAttribute` this way on a node in the DOM would mean overwriting any classes already on the element, but that's not a concern where we're putting this element together from scratch.

Now we're ready to move on to the second item on our to-do list: hiding all the other paragraphs.

Since we've made changes to code we know already worked, be sure to reload the page to make sure everything is still working as expected. We don't want to introduce a bug here and continue on writing code, or we'll eventually get stuck digging back through all the changes we made. If everything has gone according to plan, the page should look the same when we reload it now.


It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it was the epoch of belief, it was the epoch of incredulity, it was the season of Light, it was the season of Darkness, it was the spring of hope, it was the winter of despair, we had everything before us, we had nothing before us, we were all going direct to Heaven, we were all going direct the other way--in short, the period was so far like the present period, that some of its noisiest authorities insisted on its being received, for good or for evil, in the superlative degree of comparison only. Read more

There were a king with a large jaw and a queen with a plain face, on the throne of England; there were a king with a large jaw and a queen with a fair face, on the throne of France. In both countries it was clearer than crystal to the lords of the State preserves of loaves and fishes, that things in general were settled for ever.

FIG 5.4: No visible changes, but this change keeps our styling decisions in our CSS and our behavioral decisions in JavaScript.

Now we have a list of all the paragraphs on the page, and we need to act on each of them. We need a loop—and since we're iterating over an array-like node list, we need a `for` loop. Just to make sure we have our loop in order, we'll log each paragraph to the console before we go any further:

```
var newLink = document.createElement( "a" );
var allParagraphs = document.getElementsByTagName(
  "p" );
var firstParagraph = allParagraphs[ 0 ];

newLink.setAttribute( "href", "#" );
newLink.setAttribute( "class", "more-link" );
newLink.innerHTML = "Read more";

for( var i = 0; i < allParagraphs.length; i++ ) {
  console.log( allParagraphs[ i ] );
}
```

```
firstParagraph.appendChild( newLink );
```

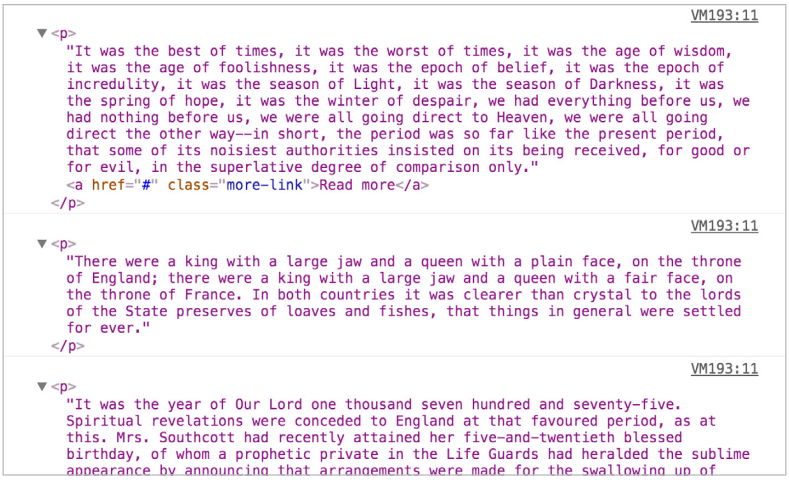
Your Read More link should still be kicking around in the first paragraph as usual, and your console should be rich with filler text (**FIG 5.5**).

Now we have to hide those paragraphs with `display: none`, and we have a couple of options: we could use a class the way we did before, but it wouldn't be a terrible idea to use styles in JavaScript for this. We're controlling all the hiding and showing from our script, and there's no chance we'll want that behavior to be overridden by something in a stylesheet. In this case, it makes sense to use the DOM's built-in methods for applying styles:

```
var newLink = document.createElement( "a" );
var allParagraphs = document.getElementsByTagName(
  "p" );
var firstParagraph = allParagraphs[ 0 ];

newLink.setAttribute( "href", "#" );
newLink.setAttribute( "class", "more-link" );
newLink.innerHTML = "Read more";

for( var i = 0; i < allParagraphs.length; i++ ) {
  allParagraphs[ i ].style.display = "none";
}

firstParagraph.appendChild( newLink );
```

If we reload the page now, everything is gone: our JavaScript loops through the entire list of paragraphs and hides them all. We need to make an exception for the first paragraph, and that means conditional logic—an `if` statement, and the `i` variable gives us an easy value to check against:

```
var newLink = document.createElement( "a" );
var allParagraphs = document.getElementsByTagName(
  "p" );
var firstParagraph = allParagraphs[ 0 ];
```



```
                                                                VM193:11
▼<p>
   "It was the best of times, it was the worst of times, it was the age of wisdom,
   it was the age of foolishness, it was the epoch of belief, it was the epoch of
   incredulity, it was the season of Light, it was the season of Darkness, it was
   the spring of hope, it was the winter of despair, we had everything before us, we
   had nothing before us, we were all going direct to Heaven, we were all going
   direct the other way—in short, the period was so far like the present period,
   that some of its noisiest authorities insisted on its being received, for good or
   for evil, in the superlative degree of comparison only."
   <a href="#" class="more-link">Read more</a>
 </p>
                                                                VM193:11
▼<p>
   "There were a king with a large jaw and a queen with a plain face, on the throne
   of England; there were a king with a large jaw and a queen with a fair face, on
   the throne of France. In both countries it was clearer than crystal to the lords
   of the State preserves of loaves and fishes, that things in general were settled
   for ever."
 </p>
                                                                VM193:11
▼<p>
   "It was the year of Our Lord one thousand seven hundred and seventy-five.
   Spiritual revelations were conceded to England at that favoured period, as at
   this. Mrs. Southcott had recently attained her five-and-twentieth blessed
   birthday, of whom a prophetic private in the Life Guards had heralded the sublime
   appearance by announcing that arrangements were made for the swallowing up of
```

FIG 5.5: Looks like our loop is doing what we expect.

```
newLink.setAttribute( "href", "#" );
newLink.setAttribute( "class", "more-link" );
newLink.innerHTML = "Read more";

for( var i = 0; i < allParagraphs.length; i++ ) {

  if( i === 0 ) {
    continue;
  }
  allParagraphs[ i ].style.display = "none";
}

firstParagraph.appendChild( newLink );
```

If this is the first time through of the loop, the `continue` keyword skips the rest of the current iteration and then—unlike if we'd used `break`—the loop continues on to the next iteration.

If you reload the page now, we'll have a single paragraph with a Read More link at the end, but all the others will be hidden. Things are looking good so far—and if things aren't looking quite so good for you, double-check your console to make sure nothing is amiss.

DOM EVENTS

Okay, one last thing to do: we need our Read More link to *do* something. If you click on it now, it just jumps you back to the top of the page and adds a hash to the URL.

In terms of failure-proofing, we're in a pretty safe situation with this script. No matter what kinds of mishaps might take place down the road—an error in one of our scripts, an error in a third-party script that we can't control, or even an error in a user's browser—the full text will be available.

We're inserting the View More link with JavaScript instead of hard-coding it into the markup, so if JavaScript is unavailable to a user for any reason, there won't be a useless link floating at the end of the first paragraph. We're also relying on JavaScript to add the class that hides the other paragraphs, rather than hard-coding a class and hiding them through our stylesheets—because then, if a script should break, the content is still available to the user.

The idea of starting with something usable and layering JavaScript enhancements over that baseline is called *progressive enhancement*, and we'll talk more about that in a bit. Right now, we have a script to finish.

DOM events are effectively an API for the activity taking place in a browser. This includes the user's actions, CSS animations, and internal browser events like the point where an image is completely loaded, just to name a few.

We're squarely in user event territory—we just need to be able to write some behavior for users who click our generated link. We don't need to make a second trip to the DOM, or root through every link on the page—we already have a reference to our link, and we'll use a built-in DOM method to listen for events: `addEventListener`.

So, let's start by writing our function: when the link is clicked, what do we want to happen?

Well, first we want to show all the hidden paragraphs on the page, so we'll need to change their styles back to `display: block`. Once we've shown all those paragraphs, a Read More link won't make sense to the user—so after we show the full text, we'll want remove that link from the DOM.

We'll create a new function with the identifier `revealCopy`, and for now we'll just put a `console.log` in that function so we know everything is working. Then we'll use `addEventListener` on `newLink` to listen for a click event. `addEventListener` accepts two arguments: a string with the event type we want to listen for—in this case `"click"`—and the function to be executed when that event takes place.

```
var newLink = document.createElement( "a" );
var allParagraphs = document.getElementsByTagName(
  "p" );
var firstParagraph = allParagraphs[ 0 ];

function revealCopy() {
  console.log( "Clicked!" );
}

newLink.setAttribute( "href", "#" );
newLink.setAttribute( "class", "more-link" );
newLink.innerHTML = "Read more";

newLink.addEventListener( "click", revealCopy );

for( var i = 0; i < allParagraphs.length; i++ ) {
  if( i === 0 ) {
    continue;
  }
  allParagraphs[ i ].style.display = "none";
}

firstParagraph.appendChild( newLink );
```

So far, so good! Click on our generated link and you'll get a `Clicked!` in your console.

The browser is still following that link, though. That isn't really a big deal now—since we're setting that link's `href` to a hash—but there might be times where we're adding custom behaviors to real links, and we don't want the browser jumping the user to a new page instead of showing any of our behavior.

Luckily, `addEventListener` gives us information about the user's click event in the form of—you guessed it—an object. And just as you might expect, that object contains a number of properties *about* the event, and methods we can use to control the browser's behavior. That event object is passed along as an argument, but we can't use it until we give it an identifier—the common convention is `e`, short for "event." Let's add that as an argument, and change our `console.log` to show us that object:

```
var newLink = document.createElement( "a" );
var allParagraphs = document.getElementsByTagName(
  "p" );
var firstParagraph = allParagraphs[ 0 ];

function revealCopy( e ) {
  console.log( e );
}

newLink.setAttribute( "href", "#" );
newLink.setAttribute( "class", "more-link" );
newLink.innerHTML = "Read more";

newLink.addEventListener( "click", revealCopy );

for( var i = 0; i < allParagraphs.length; i++ ) {
  if( i === 0 ) {
    continue;
  }
  allParagraphs[ i ].style.display = "none";
}

firstParagraph.appendChild( newLink );
```

Now we'll get a cryptic looking object in our console when we click on that generated link:

```
MouseEvent {dataTransfer: null, which: 1, toElement:
  a.more-link, fromElement: null, y: 467…}
```

There's a lot going on in there, but we're only going to need one method from that event object: `e.preventDefault()`, which prevents the browser's default behavior when an event takes place—in this case, following a link. That function can appear *anywhere* in the function that's bound to an event—as long as it exists in `revealCopy` somewhere, the browser won't attempt to follow our link.

```
var newLink = document.createElement( "a" );
var allParagraphs = document.getElementsByTagName(
  "p" );
var firstParagraph = allParagraphs[ 0 ];

function revealCopy( e ) {
  e.preventDefault();
};

newLink.setAttribute( "href", "#" );
newLink.setAttribute( "class", "more-link" );
newLink.innerHTML = "Read more";

newLink.addEventListener( "click", revealCopy );

for( var i = 0; i < allParagraphs.length; i++ ) {
  if( i === 0 ) {
    continue;
  }
  allParagraphs[ i ].style.display = "none";
}

firstParagraph.appendChild( newLink );
```

Now the hash in the link's `href` will be ignored completely. Even if we were to change `newLink.setAttribute( "href", "#" )` to point to a real URL, clicking on the link wouldn't take you anywhere. Perfect—now we just need our function to...y'know, do things.

Since we'll need to change the `display` property of each hidden paragraph back to a visible `display` value—like `block`—we'll need to loop through them again. For now, we can just copy and paste our `allParagraphs` variable and corresponding loop into the function, and change the `display` value to `"block"`.

```
var newLink = document.createElement( "a" );
var allParagraphs = document.getElementsByTagName(
  "p" );
var firstParagraph = allParagraphs[ 0 ];

function revealCopy( e ) {
  var allParagraphs = document.getElementsByTagName(
    "p" );

  for( var i = 0; i < allParagraphs.length; i++ ) {
    if( i === 0 ) {
      continue;
    }
    allParagraphs[ i ].style.display = "block";
  }
  e.preventDefault();
}

newLink.setAttribute( "href", "#" );
newLink.setAttribute( "class", "more-link" );
newLink.innerHTML = "Read more";
newLink.addEventListener( "click", revealCopy );
```

```
for( var i = 0; i < allParagraphs.length; i++ ) {
  if( i === 0 ) {
    continue;
  }
  allParagraphs[ i ].style.display = "none";
}

firstParagraph.appendChild( newLink );
```

Copying and pasting code that way isn't very DRY, but it does work—and we can clean things up a little once we've got everything working. Give this a try now, and we're almost done: all paragraphs but the first one are hidden once the page finishes loading, and they're all revealed again when we click on that link.

We have one more thing to do: we should remove the Read More link from the DOM once clicked, since it won't do anything anymore. This is pretty painless: there's a `remove` method built into to each DOM node, and it does exactly what you might expect.

First, though, we need a reference to the Read More link we're removing. We won't need to make another trip to the DOM for that: the `this` keyword inside a function that's attached to an event refers to the element that initiated the event. Inside `revealCopy`, `this` refers to our Read More node, and we'll call `remove()` on it:

```
var newLink = document.createElement( "a" );
var allParagraphs = document.getElementsByTagName(
  "p" );
var firstParagraph = allParagraphs[ 0 ];

function revealCopy( e ) {
  var allParagraphs = document.getElementsByTagName(
    "p" );
    for( var i = 0; i < allParagraphs.length; i++ ) {
      if( i === 0 ) {
        continue;
      }

      allParagraphs[ i ].style.display = "block";
    }
    this.remove();
    e.preventDefault();
  }

  newLink.setAttribute( "href", "#" );
  newLink.setAttribute( "class", "more-link" );
  newLink.innerHTML = "Read more";
  newLink.addEventListener( "click", revealCopy );

  for( var i = 0; i < allParagraphs.length; i++ ) {
    if( i === 0 ) {
      continue;
    }
    allParagraphs[ i ].style.display = "none";
  }

  firstParagraph.appendChild( newLink );
```

It works! We have reached *minimum viable product*, here—this isn't the neatest code ever, but we've built exactly what we were aiming to build. No gold stars awarded just yet, but at least we're all getting a participant ribbon.

Now we can optimize.

Remember how we copied and pasted that loop? There's some room for improvement there: we already have a function that loops through all our paragraphs and changes the `display` property on all but the first one, and functions are all about reuse. Since there are two situations where we'll need to change that `display` property—to `none` initially, and to `block` when our link is clicked—we'll refactor that function to serve both cases.

First things first: we should change the name of that function. We won't just be using it to *reveal* our paragraphs; we'll be using it to hide them as well. Since we're toggling those paragraphs' visibility, we'll change the identifier and update the reference inside `addEventListener` to match—`toggleCopy` makes sense to me. Then, let's try calling that function in place of our original loop:

```
var newLink = document.createElement( "a" );
var allParagraphs = document.getElementsByTagName(
  "p" );
var firstParagraph = allParagraphs[ 0 ];

function toggleCopy( e ) {
  var allParagraphs = document.getElementsByTagName(
    "p" );

  for( var i = 0; i < allParagraphs.length; i++ ) {
    if( i === 0 ) {
      continue;
    }
    allParagraphs[ i ].style.display = "block";
  }
  this.remove();
  e.preventDefault();
}

newLink.setAttribute( "href", "#" );
newLink.setAttribute( "class", "more-link" );
newLink.innerHTML = "Read more";
newLink.addEventListener( "click", toggleCopy );

toggleCopy();

firstParagraph.appendChild( newLink );
```

Uh oh.

```
Uncaught TypeError: this.remove is not a function
```

We've built a few assumptions into our function. JavaScript is expecting `this` to reference a DOM node that has a `remove()` method—that won't apply outside of our event. Our script didn't get as far as the line after, but that would cause an error too—again, the function assumed an `e` argument with a `preventDefault` method attached, which won't exist if we're not calling this function in response to an event. `e` has an identifier, but without `addEventListener` to give it an object, it's just an identifier that contains `undefined`.

First we'll deal with the error in our console. We need to make sure `this` is a reference to the link we've conjured up, and if so, remove it. That's an easy one: we already have a reference to the Read More—the `newLink` identifier. We'll just make sure `this` and `newLink` are equal.

```
var newLink = document.createElement( "a" );
var allParagraphs = document.getElementsByTagName(
  "p" );
var firstParagraph = allParagraphs[ 0 ];

function toggleCopy( e ) {
  var allParagraphs = document.getElementsByTagName(
    "p" );

  for( var i = 0; i < allParagraphs.length; i++ ) {
    if( i === 0 ) {
      continue;
    }
    allParagraphs[ i ].style.display = "block";
    }

    if( this === newLink ) {
       this.remove();
    }

  e.preventDefault();
}
```

```
newLink.setAttribute( "href", "#" );
newLink.setAttribute( "class", "more-link" );
newLink.innerHTML = "Read more";
newLink.addEventListener( "click", toggleCopy );

toggleCopy();

firstParagraph.appendChild( newLink );
```

That's one error down, but here's that `preventDefault` issue we've been expecting:

```
Uncaught TypeError: Cannot read property
  'preventDefault' of undefined
```

When we invoke `toggleCopy` without an argument, `e` gets a value of `undefined`—and `undefined` definitely doesn't have a `preventDefault` method. That `undefined` default value means that the `e` identifier gives us just what we need to get our function back in working order: a condition we can test for. We'll only invoke `e.preventDefault` if `e` doesn't have a value of `undefined`:

```
var newLink = document.createElement( "a" );
var allParagraphs = document.getElementsByTagName(
  "p" );
var firstParagraph = allParagraphs[ 0 ];

function toggleCopy( e ) {
  var allParagraphs = document.getElementsByTagName(
    "p" );

  for( var i = 0; i < allParagraphs.length; i++ ) {
    if( i === 0 ) {
      continue;
    }
    allParagraphs[ i ].style.display = "block";
  }
```

```
    if( this === newLink ) {
      this.remove();
    }

    if( e !== undefined ) {
      e.preventDefault();
    }
  }

  newLink.setAttribute( "href", "#" );
  newLink.setAttribute( "class", "more-link" );
  newLink.innerHTML = "Read more";
  newLink.addEventListener( "click", toggleCopy );

  toggleCopy();

  firstParagraph.appendChild( newLink );
```

Now we're error-free, but we're still making a big assumption here—if we were to pass this function an argument when invoking it outside of an event, `e` wouldn't be `undefined`. `e` would take on the value of the argument, which very likely wouldn't have a `preventDefault` method, and then we'd have an error on our hands. *We* know better than to chuck a stray argument into `toggleCopy` because, well, we built the thing—we know it wouldn't do anything of value. There's certainly no harm in doing a little error-proofing for the sake of whomever ends up maintaining our code after us.

Just to be extra safe, we'll make our conditional a little more explicit: first, see if there's an argument at all. If there is, see if that argument has a `preventDefault` method. Since we're checking against two values that both need to evaluate to `true`, we'll use `&&`.

```
var newLink = document.createElement( "a" );
var allParagraphs = document.getElementsByTagName(
  "p" );
var firstParagraph = allParagraphs[ 0 ];
```

```
function toggleCopy( e ) {
  var allParagraphs = document.getElementsByTagName(
    "p" );

  for( var i = 0; i < allParagraphs.length; i++ ) {
    if( i === 0 ) {
      continue;
    }
    allParagraphs[ i ].style.display = "block";
  }

  if( this === newLink ) {
    this.remove();
  }
  if( e !== undefined && e.preventDefault !==
    undefined ) {
    e.preventDefault();
  }
}

newLink.setAttribute( "href", "#" );
newLink.setAttribute( "class", "more-link" );
newLink.innerHTML = "Read more";
newLink.addEventListener( "click", toggleCopy );

toggleCopy();

firstParagraph.appendChild( newLink );
```

Still no errors; all is well.

Nothing is hidden either, though—we're still just setting `display` to `block`. What we need to do is set those elements to `block` *only* if they're already hidden—we'll need one more `if` that checks to see if the paragraph's display property is set to `none` and, if so, set it to `block`. For any other value, we'll set that value to `none`.

```
var newLink = document.createElement( "a" );
var allParagraphs = document.getElementsByTagName(
  "p" );
var firstParagraph = allParagraphs[ 0 ];

function toggleCopy( e ) {
  var allParagraphs = document.getElementsByTagName(
    "p" );

  for( var i = 0; i < allParagraphs.length; i++ ) {
    if( i === 0 ) {
      continue;
    }
    if( allParagraphs[ i ].style.display === "none" ) {
      allParagraphs[ i ].style.display = "block";
    } else {
      allParagraphs[ i ].style.display = "none";
    }
  }

  if( this === newLink ) {
    this.remove();
  }

  if( e !== undefined && e.preventDefault !==
    undefined ) {
    e.preventDefault();
  }
}

newLink.setAttribute( "href", "#" );
newLink.setAttribute( "class", "more-link" );
newLink.innerHTML = "Read more";
newLink.addEventListener( "click", toggleCopy );

toggleCopy();

firstParagraph.appendChild( newLink );
```

Working again! One more little thing, though: we're repeating `allParagraphs[ i ]` over and over when we could just be referencing it by a single identifier.

We're nit-picking a little now, but what could it hurt?

```
var newLink = document.createElement( "a" );
var allParagraphs = document.getElementsByTagName(
  "p" );
var firstParagraph = allParagraphs[ 0 ];

function toggleCopy( e ) {
  var allParagraphs = document.getElementsByTagName(
    "p" );

  for( var i = 0; i < allParagraphs.length; i++ ) {
    var para = allParagraphs[ i ];

    if( i === 0 ) {
      continue;
    }
    if( para.style.display === "none" ) {
    para.style.display = "block";
    } else {
    para.style.display = "none";
    }
  }

  if( this === newLink ) {
    this.remove();
  }

  if( e !== undefined && e.preventDefault !==
    undefined ) {
    e.preventDefault();
  }
}

newLink.setAttribute( "href", "#" );
newLink.setAttribute( "class", "more-link" );
newLink.innerHTML = "Read more";
newLink.addEventListener( "click", toggleCopy );

toggleCopy();

firstParagraph.appendChild( newLink );
```

And while we're at it, we're polluting the global scope a little—all of these variables are getting added to `window`, since they're not in an enclosing function. Try it out for yourself in the console:

```
window.newLink
<a></a>
```

We don't want to clutter up the global scope. Let's wrap this whole thing in a function; and since we don't to reference it outside of the moment the page is loaded, we don't need to give it an identifier—we'll wrap all of our JavaScript in an anonymous function that executes it right away, called an *immediately-invoked functional expression* or IIFE. The syntax is a little strange and can be written in a couple of different ways, but the usual gist is this: by wrapping an anonymous function in parentheses, we tell JavaScript that any instance of the `function` keyword is an expression, not a declaration—*invoking* a function, not potentially *defining* one with an identifier. We then follow that up with a matched set of parentheses—`()`—to kick that newly created function off right away.

We're into slightly academic territory here, truth be told: the IIFE pattern is important for sure, but we don't need to know how JavaScript feels about parentheses to make use of it. For now, we can take it at face value.

```
(function() {
  var newLink = document.createElement( "a" );
  var allParagraphs = document.getElementsByTagName(
    "p" );
  var firstParagraph = allParagraphs[ 0 ];
```

```
    function toggleCopy( e ) {
      var allParagraphs = document.getElementsByTagName(
        "p" );

      for( var i = 0; i < allParagraphs.length; i++ ) {
        var para = allParagraphs[ i ];

        if( i === 0 ) {
          continue;
        }
        if( para.style.display === "none" ) {
          para.style.display = "block";
        } else {
          para.style.display = "none";
        }
      }

      if( this === newLink ) {
        this.remove();
      }

      if( e !== undefined && e.preventDefault !==
        undefined ) {
        e.preventDefault();
      }
    };

    newLink.setAttribute( "href", "#" );
    newLink.setAttribute( "class", "more-link" );
    newLink.innerHTML = "Read more";
    newLink.addEventListener( "click", toggleCopy );

    toggleCopy();

    firstParagraph.appendChild( newLink );
  }());
```

Now, if we punch `window.newLink` into our developer console, we get back *undefined*—we're not polluting the global scope with identifiers we'll never need to access outside the scope of our IIFE.

```
window.newLink
undefined
```

Perfect.

Well, okay. Not *perfect*—in fact, probably never *perfect*. There are always more tweaks we can make to a script, tiny optimization after tiny optimization and so on unto infinity. But this is pretty damn good, if I do say so myself: we're being responsible about polluting the global scope, we're being DRY throughout our code, and we've written something that will be easy to read and maintain long after we've moved on to bigger and better scripts.

PROGRESSIVE ENHANCEMENT

Scripting behavior in a responsible way isn't always easy. We're standing in for the browser, taking over the user's experience of something as common and predictable as clicking on a link. Done in an unobtrusive way, we're able to create a completely fluid experience—better, in many ways, than the browser itself could.

If we don't do it responsibly, though, we've done something far worse than simply presenting the user with a misaligned `div`—we've built something they might not be able to use at all. The web is an unpredictable medium, and we have to plan for that—when writing JavaScript more so than HTML or CSS, by a wide margin.

We could have cut some corners in the paragraph-toggling script we built today, for example. We could have hidden those paragraphs at the outset using CSS and relied on JavaScript to show them again, or hard-coded the Read More link and assumed the functionality in our script would always be available. The latter case would be a nuisance if anything went wrong—an error elsewhere in a script causing ours to fail, for example. The user would be left with a Read More link that didn't *do* anything. The former case would be far more dire: in the event that JavaScript failed in any way, the user would be left with no way to access the contents of the page.

A site that fully relies on JavaScript for critical functionality—a website built on the expectation that JavaScript will always run, no matter what—is a fragile one. Users' browsing conditions can change minute to minute, and we can't plan for—we can't *know*—the ways that our scripts might break down.

A handful of years ago I worked on the responsive *Boston Globe* site with Ethan Marcotte, Scott Jehl, and the whole crew at Filament Group. It was built with progressive enhancement in mind, which didn't hold us back in the least—there are some incredible features on that site, if you don't mind my saying so (https://www.bostonglobe.com/).

We got to solve some tricky problems on that project, but made sure we were doing so with progressive enhancement squarely in mind—"If and when this feature fails, how do we ensure the user still has access to the underlying information?" On the surface, that may seem like an exercise in edge cases. The tiny decisions that go into building a website don't necessarily feel like a big deal at the time.

The Boston Marathon bombings happened a few years later. Being able to reach up-to-date information on what was happening throughout the city was tremendously important to a huge number of people that day, and many of them looked to the *Boston Globe* for that. Due to the increased traffic, the *Boston Globe*'s CDN—the server that delivers assets like CSS and JavaScript—was overwhelmed, and went down. For a period of time that afternoon, the *Boston Globe*'s website was HTML-only.

The website *looked* broken, and none of our advanced JavaScript features were there consistently: no offline reading, no dropdown menus in the navigation. Sometimes the whole site was just black Times New Roman on a white background. Sometimes only the CSS would come through, or some part of it—likewise with the JavaScript. Rarely did any of our scripts run without errors, through no fault of our own.

But visitors to BostonGlobe.com that afternoon could still navigate the site. They could still read the news. The website *worked*. If we'd relied on CSS to hide parts of the navigation and assumed JavaScript would always be there to reveal them again, some users wouldn't have been able to navigate that day. If we'd relied on JavaScript to fetch and render critical parts of the page, that content might never have appeared. If we'd hard-coded controls that required JavaScript in order to do anything at all, they would have been useless—confusing and frustrating for the site's users at the worst imaginable time.

Progressive enhancement is yet another thing we'll need to factor in when we're writing our JavaScript. To be honest, progressive enhancement will sometimes mean doing more work—but that's the craft. The decisions that went into writing the JavaScript for BostonGlobe.com could have seemed inconsequential in the grand scheme of things, but on that day—for tens of thousands of users—those decisions added up to something huge. For those users, progressive enhancement meant the difference between finding the information they needed right away, or being forced to keep searching for it—between knowing and not knowing.

CONCLUSION

We've come a long way, from "Hello, world" to a script that accepts user input and changes the entire page.

This has been a hell of a lot of JavaScript over a short number of pages, so if you didn't catch every single definition or punch every snippet of code into your console, don't sweat it. The goal was never absolute mastery over JavaScript, after all. As promised, we've only just barely scratched the surface of all the things JavaScript can do. And JavaScript is constantly evolving, like any other web standard—we can't know it all, and we'll never have to. To this day, I spend a lot of time searching the web for the best way to do *X* or *Y* with JavaScript.

But for all the exciting new APIs and features and powers over the browser that JavaScript grants us—or will grant us, someday—the basics won't change. An array is an array; an event is an event. Anything you've read here is something you'll be able to use, something you'll be able to point to and recognize when you're looking at a script someone else wrote.

You don't suddenly have a special programmer-brain by reaching this point, because nobody does. That isn't what makes a developer. What makes a developer is a curiosity, a willingness to learn, and maybe the drive to solve a puzzle or two.

If you're here, reading this, you're already on your way.

RESOURCES

So, where to from here? An entire programming language is sprawling out before you. Well, there are plenty more books to read—and read them you should—but now that you know what to look for, there's no substitute for reading code. Take a little time to read through the scripts you've seen flying around a project at work, or dig into the code that powers your favorite open source tools. There'll be a lot more going on than we've covered here, but I bet you'll recognize more than you think.

Next steps

- **Mozilla Developer Network**. If you've made it this far, it's time I revealed the secret to a successful career in professional JavaScript development: *cheating*. JavaScript is too much for any mortal to commit to memory. When the time comes to look something up, whether it's about capitalization or browser support—and that time will come, more than once per day—MDN is the place to do it (http://bkaprt.com/jsfwd/07-01/).
- ***Responsible Responsive Design***, Scott Jehl. Progressive enhancement is a concept we barely touched on, but it formed the foundation for the script we built. JavaScript enhancements aren't always guaranteed to be available to your users, and not just because users have turned it off at the browser (though some do). *Responsible Responsive Design* is a closer look at inclusive front-end development practices, from progressive enhancement to accessibility to performance (http://bkaprt.com/jsfwd/07-02/).
- ***If Hemingway Wrote JavaScript***, Angus Croll. You might not know it from my Twitter account, but I'm a pretty big fan of the written word. *If Hemingway Wrote JavaScript* is an imagined look at the coding styles of authors like Jane Austen and William Shakespeare, based on their writing. Not all of it is in JavaScript, but every language you'll encounter is made of the principles you'll be familiar with now. It's a fun way to look at the signatures people leave on their code (http://bkaprt.com/jsfwd/07-03/).

Digging deeper

Now you've got enough of a foundation to get into JavaScript in a major way. If you're champing at the bit to learn more advanced JavaScript and you're excited to keep on unraveling JavaScript puzzles, I've got more reading material to share.

- ***Eloquent JavaScript***, Marijn Haverbeke. You won't find the tenor of this book terribly unfamiliar, unless you skipped right to the end of the one you're reading now: Marjin draws a similar path through the concepts of JavaScript, though they're going to come at you a little faster and in grittier detail http://eloquentjavascript.net/).
- ***JavaScript Patterns***, Stoyan Stefanov. I've come back to this book a few times over the course of my career. I'd read through it until I felt like I'd gotten a little lost, then put it down for a while. With time and experience, I'd end up landing a little further into the book each time—and, in doing so, find countless ways to reexamine parts of JavaScript I thought I knew inside and out (http://bkaprt.com/jsfwd/07-04/).
- ***Effective JavaScript: 68 Specific Ways to Harness the Power of JavaScript***, David Herman. Like others in this list, *Effective JavaScript* isn't the lightest read. It *is*, however, a highly approachable follow-up to the concepts we've covered together, all packaged up as an indispensable list of best practices and useful advice (http://effectivejs.com/).

ACKNOWLEDGMENTS

Man, this section is something I never thought I'd end up writing. Same goes for the rest of the book, if I'm being honest. I guess I should've known better, though. I should've seen this all coming.

Not because of *me*—God, no—but because of the people who've helped me with my writing, with my speaking, with landing a job with a desk, with *everything*. With those people supporting me—with their brilliance, their kindness, and their willingness to curse me out as needed—even a scrub like me had a chance to do something like this.

Katel LeDû, Jeffrey Zeldman, and Jason Santa Maria: you've built something incredible in A Book Apart, and I am *tremendously* honored to have been able to play some small part in it. I'll never be able to thank you enough for this opportunity.

Peter Richardson and Mike Pennisi, I can't imagine two better people to put to rest the haunting feeling that I'd flubbed a term, dropped a semicolon, or—horror of horrors—mistaken `undefined` for a function. I owe you one, for every "actually" I don't hear after this goes to print.

Erin Kissane, without you involved, there was no way I could've worded goodly enough for book-making. Knowing that you would be my lead editor, after admiring so much of your work over the years—it might've been the first time I felt like I could actually pull this thing off.

To the crews at Bocoup and Filament Group, past and present: I wouldn't have had anything to write about if I hadn't been fortunate enough to learn from so many brilliant and responsible JavaScripters. I only hope I've managed to do you proud.

LMM, it is rad that you edited my book. It is radder that you are my girlfriend. It is radder still that you *remained* my girlfriend, y'know, despite having to work with me on my book. I couldn't have done this without you. I couldn't do damn near anything without you. Thank you for so, so much; for everything.

Lastly, hey, Ma—remember the time Dad told me to "get something published" for him, a couple years back? Between that and "we should fix up an old bike someday," he's *still* a pain in the ass.

I guess one out of two ain't bad.

REFERENCES

Shortened URLs are numbered sequentially; the related long URLs are listed below for reference.

Introduction

00-01	https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript
00-02	https://css-tricks.com/dom/

Chapter 1

01-01	https://abookapart.com/products/responsible-responsive-design
01-02	https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop

Resources

07-01	https://developer.mozilla.org/en-US/
07-02	https://abookapart.com/products/responsible-responsive-design
07-03	https://www.nostarch.com/hemingway
07-04	http://shop.oreilly.com/product/9780596806767.do

INDEX

ABOUT THE AUTHOR

Mat "Wilto" Marquis makes websites for a living, and curses at his broken-down motorcycle for free, on the streets of north Cambridge. He is the chair of the Responsive Issues Community Group, a technical editor at *A List Apart*, a former member of the jQuery team, and an editor of the HTML5 specification. Of all these things, Mat is most proud of having finished *Mega Man 2*—on "difficult"—without losing a single life.

ABOUT A BOOK APART

COLOPHON

The text is set in FF Yoga and its companion, FF Yoga Sans, both by Xavier Dupré. Headlines and cover are set in Titling Gothic by David Berlow.