KYLE SIMPSON

UP & GOING

YOU DON'T KNOW JS

Up & Going

Kyle Simpson

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

Table of Contents

Foreword

What was the last new thing you learned?

Perhaps it was a foreign language, like Italian or German. Or maybe it was a graphics editor, like Photoshop. Or a cooking technique or woodworking or an exercise routine. I want you to remember that feeling when you finally got it: the lightbulb moment. When things went from blurry to crystal clear, as you mastered the table saw or understood the difference between masculine and feminine nouns in French. How did it feel? Pretty amazing, right?

Now I want you to travel back a little bit further in your memory to right before you learned your new skill. How did *that* feel? Probably slightly intimidating and maybe a little bit frustrating, right? At one point, we all did not know the things that we know now, and that's totally OK; we all start somewhere. Learning new material is an exciting adventure, especially if you are looking to learn the subject efficiently.

I teach a lot of beginner coding classes. The students who take my classes have often tried teaching themselves subjects like HTML or JavaScript by reading blog posts or copying and pasting code, but they haven't been able to truly master the material that will allow them to code their desired outcome. And because they don't truly grasp the ins and outs of certain coding topics, they can't write powerful code or debug their own work because they don't really understand what is happening.

I always believe in teaching my classes the proper way, meaning I teach web standards, semantic markup, well-commented code, and other best practices. I cover the subject in a thorough manner to explain the hows and whys, without just tossing out code to copy

and paste. When you strive to comprehend your code, you create better work and become better at what you do. The code isn't just your *job* anymore, it's your *craft*. This is why I love *Up & Going*. Kyle takes us on a deep dive through syntax and terminology to give a great introduction to JavaScript without cutting corners. This book doesn't skim over the surface but really allows us to genuinely understand the concepts.

Because it's not enough to be able to duplicate jQuery snippets into your website, the same way it's not enough to learn how to open, close, and save a document in Photoshop. Sure, once I learned a few basics about the program, I could create and share a design I made. But without legitimately knowing the tools and what is behind them, how can I define a grid, or craft a legible type system, or optimize graphics for web use. The same goes for JavaScript. Without knowing how loops work, or how to define variables, or what scope is, we won't be writing the best code we can. We don't want to settle for anything less—this is, after all, our craft.

The more you are exposed to JavaScript, the clearer it becomes. Words like closures, objects, and methods might seem out of reach to you now, but this book will help those terms come into clarity. I want you to keep those two feelings of before and after you learn something in mind as you begin this book. It might seem daunting, but you've picked up this book because you are starting an awesome journey to hone your knowledge. *Up & Going* is the start of our path to understanding programming. Enjoy the lightbulb moments!

—Jenn Lukas (http://jennlukas.com, @jennlukas),
Frontend consultant

Preface

I'm sure you noticed, but "JS" in the series title is not an abbreviation for words used to curse about JavaScript, though cursing at the language's quirks is something we can probably all identify with!

From the earliest days of the Web, JavaScript has been a foundational technology that drives interactive experience around the content we consume. While flickering mouse trails and annoying pop-up prompts may be where JavaScript started, nearly two decades later, the technology and capability of JavaScript has grown many orders of magnitude, and few doubt its importance at the heart of the world's most widely available software platform: the Web.

But as a language, it has perpetually been a target for a great deal of criticism, owing partly to its heritage but even more to its design philosophy. Even the name evokes, as Brendan Eich once put it, "dumb kid brother" status next to its more mature older brother, Java. But the name is merely an accident of politics and marketing. The two languages are vastly different in many important ways. "JavaScript" is as related to "Java" as "Carnival" is to "Car."

Because JavaScript borrows concepts and syntax idioms from several languages, including proud C-style procedural roots as well as subtle, less obvious Scheme/Lisp-style functional roots, it is exceedingly approachable to a broad audience of developers, even those with little to no programming experience. The "Hello World" of JavaScript is so simple that the language is inviting and easy to get comfortable with in early exposure.

While JavaScript is perhaps one of the easiest languages to get up and running with, its eccentricities make solid mastery of the language a vastly less common occurrence than in many other lan-

guages. Where it takes a pretty in-depth knowledge of a language like C or C++ to write a full-scale program, full-scale production JavaScript can, and often does, barely scratch the surface of what the language can do.

Sophisticated concepts that are deeply rooted into the language tend instead to surface themselves in *seemingly* simplistic ways, such as passing around functions as callbacks, which encourages the JavaScript developer to just use the language as is and not worry too much about what's going on under the hood.

It is simultaneously a simple, easy-to-use language that has broad appeal, and a complex and nuanced collection of language mechanics that without careful study will elude *true understanding* even for the most seasoned of JavaScript developers.

Therein lies the paradox of JavaScript, the Achilles' heel of the language, the challenge we are presently addressing. Because JavaScript *can* be used without understanding, the understanding of the language is often never attained.

Mission

If at every point that you encounter a surprise or frustration in JavaScript, your response is to add it to the blacklist (as some are accustomed to doing), you soon will be relegated to a hollow shell of the richness of JavaScript.

While this subset has been famously dubbed "The Good Parts," I would implore you, dear reader, to instead consider it the "The Easy Parts," "The Safe Parts," or even "The Incomplete Parts."

This *You Don't Know JS* series offers a contrary challenge: learn and deeply understand *all* of JavaScript, even and especially "The Tough Parts."

Here, we address head-on the tendency of JS developers to learn just enough to get by, without ever forcing themselves to learn exactly how and why the language behaves the way it does. Furthermore, we eschew the common advice to retreat when the road gets rough.

I am not content, nor should you be, at stopping once something just works and not really knowing *why*. I gently challenge you to journey down that bumpy "road less traveled" and embrace all that JavaScript is and can do. With that knowledge, no technique, no framework, and no popular buzzword acronym of the week will be beyond your understanding.

These books each take on specific core parts of the language that are most commonly misunderstood or under-understood, and dive deep and exhaustively into them. You should come away from reading with a firm confidence in your understanding, not just of the theoretical, but the practical "what you need to know" bits.

The JavaScript you know right now is probably parts handed down to you by others who've been burned by incomplete understanding. *That* JavaScript is but a shadow of the true language. You don't really know JavaScript *yet*, but if you dig into this series, you will. Read on, my friends. JavaScript awaits you.

Review

JavaScript is awesome. It's easy to learn partially, and much harder to learn completely (or even *sufficiently*). When developers encounter confusion, they usually blame the language instead of their lack of understanding. These books aim to fix that, inspiring a strong appreciation for the language you can now, and *should*, deeply know.

Many of the examples in this book assume modern (and future-reaching) JavaScript engine environments, such as ES6. Some code may not work as described if run in older (pre-ES6) engines.

Conventions Used in This Book

The following typographical conventions are used in this book:

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

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

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

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

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *http://bit.ly/ydkjs-up-going-code*.

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

does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*You Don't Know JavaScript: Up & Going* by Kyle Simpson (O'Reilly). Copyright 2015 Getify Solutions, Inc., 978-1-491-92446-4."

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

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, education, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

How to Contact Us

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

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

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/ydkjs_up-and-going*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

CHAPTER 1
Into Programming

Welcome to the *You Don't Know JS* (*YDKJS*) series.

Up & Going is an introduction to several basic concepts of programming—of course we lean toward JavaScript (often abbreviated JS) specifically—and how to approach and understand the rest of the titles in this series. Especially if you're just getting into programming and/or JavaScript, this book will briefly explore what you need to get *up and going*.

This book starts off explaining the basic principles of programming at a very high level. It's mostly intended if you are starting *YDKJS* with little to no prior programming experience, and are looking to these books to help get you started along a path to understanding programming through the lens of JavaScript.

Chapter 1 should be approached as a quick overview of the things you'll want to learn more about and practice to get *into programming*. There are also many other fantastic programming introduction resources that can help you dig into these topics further, and I encourage you to learn from them in addition to this chapter.

Once you feel comfortable with general programming basics, Chapter 2 will help guide you to a familiarity with JavaScript's flavor of programming. Chapter 2 introduces what JavaScript is about, but again, it's not a comprehensive guide—that's what the rest of the *YDKJS* books are for!

If you're already fairly comfortable with JavaScript, first check out Chapter 3 as a brief glimpse of what to expect from *YDKJS*, then jump right in!

Code

Let's start from the beginning.

A program, often referred to as *source code* or just *code*, is a set of special instructions to tell the computer what tasks to perform. Usually code is saved in a text file, although with JavaScript you can also type code directly into a developer console in a browser, which we'll cover shortly.

The rules for valid format and combinations of instructions is called a *computer language*, sometimes referred to as its *syntax*, much the same as English tells you how to spell words and how to create valid sentences using words and punctuation.

Statements

In a computer language, a group of words, numbers, and operators that performs a specific task is a *statement*. In JavaScript, a statement might look as follows:

```
a = b * 2;
```

The characters `a` and `b` are called *variables* (see "Variables" on page 14), which are like simple boxes you can store any of your stuff in. In programs, variables hold values (like the number `42`) to be used by the program. Think of them as symbolic placeholders for the values themselves.

By contrast, the `2` is just a value itself, called a *literal value*, because it stands alone without being stored in a variable.

The `=` and `*` characters are *operators* (see "Operators" on page 8)—they perform actions with the values and variables such as assignment and mathematic multiplication.

Most statements in JavaScript conclude with a semicolon (`;`) at the end.

The statement `a = b * 2;` tells the computer, roughly, to get the current value stored in the variable `b`, multiply that value by `2`, then store the result back into another variable we call `a`.

Programs are just collections of many such statements, which together describe all the steps that it takes to perform your program's purpose.

Expressions

Statements are made up of one or more *expressions*. An expression is any reference to a variable or value, or a set of variable(s) and value(s) combined with operators.

For example:

```
a = b * 2;
```

This statement has four expressions in it:

- `2` is a *literal value expression*.
- `b` is a *variable expression*, which means to retrieve its current value.
- `b * 2` is an *arithmetic expression*, which means to do the multiplication.
- `a = b * 2` is an *assignment expression*, which means to assign the result of the `b * 2` expression to the variable `a` (more on assignments later).

A general expression that stands alone is also called an *expression statement*, such as the following:

```
b * 2;
```

This flavor of expression statement is not very common or useful, as generally it wouldn't have any effect on the running of the program—it would retrieve the value of `b` and multiply it by `2`, but then wouldn't do anything with that result.

A more common expression statement is a *call expression* statement (see "Functions" on page 22), as the entire statement is the function call expression itself:

```
alert( a );
```

Executing a Program

How do those collections of programming statements tell the computer what to do? The program needs to be *executed*, also referred to as *running the program*.

Statements like `a = b * 2` are helpful for developers when reading and writing, but are not actually in a form the computer can directly understand. So a special utility on the computer (either an *interpreter* or a *compiler*) is used to translate the code you write into commands a computer can understand.

For some computer languages, this translation of commands is typically done from top to bottom, line by line, every time the program is run, which is usually called *interpreting* the code.

For other languages, the translation is done ahead of time, called *compiling* the code, so when the program *runs* later, what's running is actually the already compiled computer instructions ready to go.

It's typically asserted that JavaScript is *interpreted*, because your JavaScript source code is processed each time it's run. But that's not entirely accurate. The JavaScript engine actually *compiles* the program on the fly and then immediately runs the compiled code.

For more information on JavaScript compiling, see the first two chapters of the *Scope & Closures* title of this series.

Try It Yourself

This chapter is going to introduce each programming concept with simple snippets of code, all written in JavaScript (obviously!).

It cannot be emphasized enough: while you go through this chapter —and you may need to spend the time to go over it several times— you should practice each of these concepts by typing the code yourself. The easiest way to do that is to open up the developer tools console in your nearest browser (Firefox, Chrome, IE, etc.).

Typically, you can launch the developer console with a keyboard shortcut or from a menu item. For more detailed information about launching and using the console in your favorite browser, see "Mastering The Developer Tools Console".

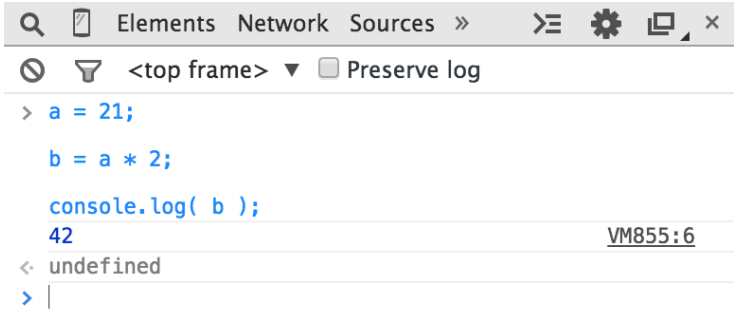
To type multiple lines into the console at once, use `<shift> + <enter>` to move to the next new line. Once you hit `<enter>` by itself, the console will run everything you've just typed.

Let's get familiar with the process of running code in the console. First, I suggest opening up an empty tab in your browser. I prefer to do this by typing `about:blank` into the address bar. Then, make sure your developer console is open, as we just mentioned.

Now, type this code and see how it runs:

```
a = 21;

b = a * 2;

console.log( b );
```

Typing the preceding code into the console in Chrome should produce something like the following:



Go on, try it. The best way to learn programming is to start coding!

Output

In the previous code snippet, we used `console.log(..)`. Briefly, let's look at what that line of code is all about.

You may have guessed, but that's exactly how we print text (aka *output* to the user) in the developer console. There are two characteristics of that statement that we should explain.

First, the `log( b )` part is referred to as a function call (see "Functions" on page 22). What's happening is we're handing the `b` variable to that function, which asks it to take the value of `b` and print it to the console.

Second, the `console.` part is an object reference where the `log(..)` function is located. We cover objects and their properties in more detail in Chapter 2.

Another way of creating output that you can see is to run an `alert(..)` statement. For example:

```
alert( b );
```

If you run that, you'll notice that instead of printing the output to the console, it shows a pop-up "OK" box with the contents of the `b` variable. However, using `console.log(..)` is generally going to make learning about coding and running your programs in the console easier than using `alert(..)` because you can output many values at once without interrupting the browser interface.

For this book, we'll use `console.log(..)` for output.

Input

While we're discussing output, you may also wonder about *input* (i.e., receiving information from the user).

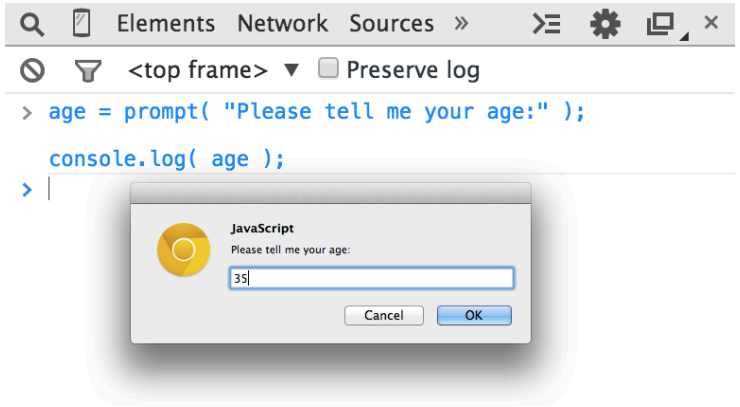
The most common way that happens is for the HTML page to show form elements (like text boxes) to a user that she can type into, and then use JS to read those values into your program's variables.

But there's an easier way to get input for simple learning and demonstration purposes such as what you'll be doing throughout this book. Use the `prompt(..)` function:

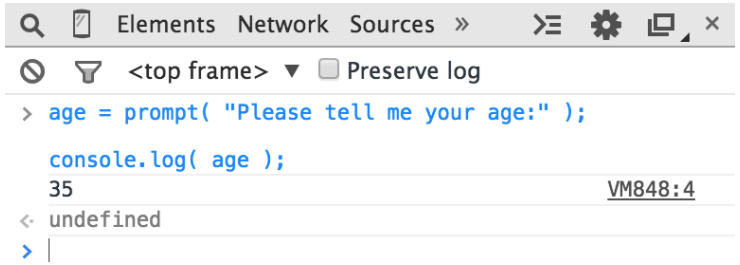
```
age = prompt( "Please tell me your age:" );

console.log( age );
```

As you may have guessed, the message you pass to `prompt(..)`—in this case, `"Please tell me your age:"`—is printed into the pop up.

This should look similar to the following:



Once you submit the input text by clicking "OK," you'll observe that the value you typed is stored in the `age` variable, which we then *output* with `console.log(..)`:



To keep things simple while we're learning basic programming concepts, the examples in this book will not require input. But now that you've seen how to use `prompt(..)`, if you want to challenge yourself, you can try to use input in your explorations of the examples.

Operators

Operators are how we perform actions on variables and values. We've already seen two JavaScript operators, the = and the *.

The * operator performs mathematic multiplication. Simple enough, right?

The = equals operator is used for *assignment*—we first calculate the value on the *right-hand side* (source value) of the = and then put it into the variable that we specify on the *left-hand side* (target variable).

This may seem like a strange reverse order to specify assignment. Instead of `a = 42`, some might prefer to flip the order so the source value is on the left and the target variable is on the right, like `42 -> a` (this is not valid JavaScript!). Unfortunately, the `a = 42` ordered form, and similar variations, is quite prevalent in modern programming languages. If it feels unnatural, just spend some time rehearsing that order in your mind to get accustomed to it.

Consider:

```
a = 2;
b = a + 1;
```

Here, we assign the `2` value to the `a` variable. Then, we get the value of the `a` variable (still `2`), add `1` to it resulting in the value `3`, then store that value in the `b` variable.

While not technically an operator, you'll need the keyword `var` in every program, as it's the primary way you *declare* (aka *create*) *var*iables (see "Variables" on page 14).

You should always declare the variable by name before you use it. But you only need to declare a variable once for each *scope* (see "Scope" on page 24); it can be used as many times after that as needed. For example:

```
var a = 20;

a = a + 1;
a = a * 2;
```

```
console.log( a );   // 42
```

Here are some of the most common operators in JavaScript:

Assignment
: `=`, as in `a = 2`.

Math
: `+` (addition), `-` (subtraction), `*` (multiplication), and `/` (division), as in `a * 3`.

Compound assignment
: `+=`, `-=`, `*=`, and `/=` are compound operators that combine a math operation with assignment, as in `a += 2` (same as `a = a + 2`).

Increment/decrement
: `++` (increment), `--` (decrement), as in `a++` (similar to `a = a + 1`).

Object property access
: `.` as in `console.log()`.

  Objects are values that hold other values at specific named locations called properties. `obj.a` means an object value called `obj` with a property of the name `a`. Properties can alternatively be accessed as `obj["a"]`. See Chapter 2.

Equality
: `==` (loose-equals), `===` (strict-equals), `!=` (loose not-equals), `!==` (strict not-equals), as in `a == b`.

  See "Values & Types" on page 10 and Chapter 2.

Comparison
: `<` (less than), `>` (greater than), `<=` (less than or loose-equals), `>=` (greater than or loose-equals), as in `a <= b`.

  See "Values & Types" on page 10 and Chapter 2.

Logical
: `&&` (and), `||` (or), as in `a || b` that selects either `a` *or* `b`.

  These operators are used to express compound conditionals (see "Conditionals" on page 18), like if either `a` *or* `b` is true.

For much more detail, and coverage of operators not mentioned here, see the Mozilla Developer Network (MDN)'s "Expressions and Operators".

Values & Types

If you ask an employee at a phone store how much a certain phone costs, and he says "ninety-nine, ninety-nine" (i.e., $99.99), he's giving you an actual numeric dollar figure that represents what you'll need to pay (plus taxes) to buy it. If you want to buy two of those phones, you can easily do the mental math to double that value to get $199.98 for your base cost.

If that same employee picks up another similar phone but says it's "free" (perhaps with air quotes), he's not giving you a number, but instead another kind of representation of your expected cost ($0.00) —the word "free."

When you later ask if the phone includes a charger, the answer can only be "yes" or "no."

In very similar ways, when you express values in a programs, you choose different representations for those values based on what you plan to do with them.

These different representations for values are called *types* in programming terminology. JavaScript has built-in types for each of these so-called *primitive* values:

- When you need to do math, you want a `number`.
- When you need to print a value on the screen, you need a `string` (one or more characters, words, or sentences).
- When you need to make a decision in your program, you need a `boolean` (`true` or `false`).

Values that are included directly in the source code are called *literals*. `string` literals are surrounded by double quotes (`"..."`) or single quotes (`'...'`)—the only difference is stylistic preference. `number` and `boolean` literals are just presented as is (e.g., `42`, `true`, etc.).

Consider:

```
"I am a string";
'I am also a string';

42;

true;
false;
```

Beyond `string`/`number`/`boolean` value types, it's common for programming languages to provide *arrays*, *objects*, *functions*, and more. We'll cover much more about values and types throughout this chapter and the next.

Converting Between Types

If you have a `number` but need to print it on the screen, you need to convert the value to a `string`, and in JavaScript this conversion is called "coercion." Similarly, if someone enters a series of numeric characters into a form on an ecommerce page, that's a `string`, but if you need to then use that value to do math operations, you need to *coerce* it to a `number`.

JavaScript provides several different facilities for forcibly coercing between *types*. For example:

```
var a = "42";
var b = Number( a );

console.log( a );   // "42"
console.log( b );   // 42
```

Using `Number(..)` (a built-in function) as shown is an *explicit* coercion from any other type to the `number` type. That should be pretty straightforward.

But a controversial topic is what happens when you try to compare two values that are not already of the same type, which would require *implicit* coercion.

When comparing the string `"99.99"` to the number `99.99`, most people would agree they are equivalent. But they're not exactly the same, are they? It's the same value in two different representations, two different *types*. You could say they're "loosely equal," couldn't you?

To help you out in these common situations, JavaScript will sometimes kick in and *implicitly* coerce values to the matching types.

So if you use the `==` loose-equals operator to make the comparison `"99.99" == 99.99`, JavaScript will convert the left-hand side `"99.99"` to its `number` equivalent `99.99`. The comparison then becomes `99.99 == 99.99`, which is of course `true`.

While designed to help you, implicit coercion can create confusion if you haven't taken the time to learn the rules that govern its behavior. Most JS developers never have, so the common feeling is that implicit coercion is confusing and harms programs with unexpected bugs, and should thus be avoided. It's even sometimes called a flaw in the design of the language.

However, implicit coercion is a mechanism that *can be learned*, and moreover *should be learned* by anyone wishing to take JavaScript programming seriously. Not only is it not confusing once you learn the rules, it can actually make your programs better! The effort is well worth it.

For more information on coercion, see Chapter 2 of this title and Chapter 4 of the *Types & Grammar* title of this series.

Code Comments

The phone store employee might jot down some notes on the features of a newly released phone or on the new plans her company offers. These notes are only for the employee—they're not for customers to read. Nevertheless, these notes help the employee do her job better by documenting the hows and whys of what she should tell customers.

One of the most important lessons you can learn about writing code is that it's not just for the computer. Code is every bit as much, if not more, for the developer as it is for the compiler.

Your computer only cares about machine code, a series of binary 0s and 1s, that comes from *compilation*. There's a nearly infinite number of programs you could write that yield the same series of 0s and 1s. The choices you make about how to write your program matter

—not only to you, but to your other team members and even to your future self.

You should strive not just to write programs that work correctly, but programs that make sense when examined. You can go a long way in that effort by choosing good names for your variables (see "Variables" on page 14) and functions (see "Functions" on page 22).

But another important part is code comments. These are bits of text in your program that are inserted purely to explain things to a human. The interpreter/compiler will always ignore these comments.

There are lots of opinions on what makes well-commented code; we can't really define absolute universal rules. But some observations and guidelines are quite useful:

- Code without comments is suboptimal.
- Too many comments (one per line, for example) is probably a sign of poorly written code.
- Comments should explain *why*, not *what*. They can optionally explain *how* if what's written is particularly confusing.

In JavaScript, there are two types of comments possible: a single-line comment and a multiline comment.

Consider:

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

/* But this is
       a multiline
            comment.
                    */
```

The `//` single-line comment is appropriate if you're going to put a comment right above a single statement, or even at the end of a line. Everything on the line after the `//` is treated as the comment (and thus ignored by the compiler), all the way to the end of the line. There's no restriction to what can appear inside a single-line comment.

Consider:

```
var a = 42;     // 42 is the meaning of life
```

The `/* .. */` multiline comment is appropriate if you have several lines worth of explanation to make in your comment.

Here's a common usage of multiline comments:

```
/* The following value is used because
   it has been shown that it answers
   every question in the universe. */
var a = 42;
```

It can also appear anywhere on a line, even in the middle of a line, because the `*/` ends it. For example:

```
var a = /* arbitrary value */ 42;

console.log( a );   // 42
```

The only thing that cannot appear inside a multiline comment is a `*/`, because that would be interpreted to end the comment.

You will definitely want to begin your learning of programming by starting off with the habit of commenting code. Throughout the rest of this chapter, you'll see I use comments to explain things, so do the same in your own practice. Trust me, everyone who reads your code will thank you!

Variables

Most useful programs need to track a value as it changes over the course of the program, undergoing different operations as called for by your program's intended tasks.

The easiest way to go about that in your program is to assign a value to a symbolic container, called a *variable*—so called because the value in this container can *vary* over time as needed.

In some programming languages, you declare a variable (container) to hold a specific type of value, such as `number` or `string`. *Static typing*, otherwise known as *type enforcement*, is typically cited as a benefit for program correctness by preventing unintended value conversions.

Other languages emphasize types for values instead of variables. *Weak typing*, otherwise known as *dynamic typing*, allows a variable to hold any type of value at any time. It's typically cited as a benefit for program flexibility by allowing a single variable to represent a

value no matter what type form that value may take at any given moment in the program's logic flow.

JavaScript uses the latter approach, *dynamic typing*, meaning variables can hold values of any *type* without any *type* enforcement.

As mentioned earlier, we declare a variable using the `var` statement —notice there's no other *type* information in the declaration. Consider this simple program:

```
var amount = 99.99;

amount = amount * 2;

console.log( amount );      // 199.98

// convert `amount` to a string, and
// add "$" on the beginning
amount = "$" + String( amount );

console.log( amount );      // "$199.98"
```

The `amount` variable starts out holding the number `99.99`, and then holds the `number` result of `amount * 2`, which is `199.98`.

The first `console.log(..)` command has to *implicitly* coerce that `number` value to a `string` to print it out.

Then the statement `amount = "$" + String(amount)` *explicitly* coerces the `199.98` value to a `string` and adds a `"$"` character to the beginning. At this point, `amount` now holds the `string` value `"$199.98"`, so the second `console.log(..)` statement doesn't need to do any coercion to print it out.

JavaScript developers will note the flexibility of using the `amount` variable for each of the `99.99`, `199.98`, and the `"$199.98"` values. Static-typing enthusiasts would prefer a separate variable like `amountStr` to hold the final `"$199.98"` representation of the value, because it's a different type.

Either way, you'll note that `amount` holds a running value that changes over the course of the program, illustrating the primary purpose of variables: managing program *state*.

In other words, *state* is tracking the changes to values as your program runs.

Another common usage of variables is for centralizing value setting. This is more typically called *constants*, when you declare a variable with a value and intend for that value to *not change* throughout the program.

You declare these constants, often at the top of a program, so that it's convenient for you to have one place to go to alter a value if you need to. By convention, JavaScript variables as constants are usually capitalized, with underscores _ between multiple words.

Here's a silly example:

```
var TAX_RATE = 0.08;    // 8% sales tax

var amount = 99.99;

amount = amount * 2;

amount = amount + (amount * TAX_RATE);

console.log( amount );              // 215.9784
console.log( amount.toFixed( 2 ) ); // "215.98"
```

Similar to how `console.log(..)` is a function `log(..)` accessed as an object property on the `console` value, `toFixed(..)` here is a function that can be accessed on `number` values. JavaScript `numbers` aren't automatically formatted for dollars—the engine doesn't know what your intent is, and there's no type for currency. `toFixed(..)` lets us specify how many decimal places we'd like the `number` rounded to, and it produces the `string` as necessary.

The `TAX_RATE` variable is only *constant* by convention—there's nothing special in this program that prevents it from being changed. But if the city raises the sales tax rate to 9%, we can still easily update our program by setting the `TAX_RATE` assigned value to `0.09` in one place, instead of finding many occurrences of the value `0.08` strewn throughout the program and updating all of them.

The newest version of JavaScript at the time of this writing (commonly called "ES6") includes a new way to declare constants, by using `const` instead of `var`:

```
// as of ES6:
const TAX_RATE = 0.08;

var amount = 99.99;

// ..
```

Constants are useful just like variables with unchanged values, except that constants also prevent accidentally changing value somewhere else after the initial setting. If you tried to assign any different value to `TAX_RATE` after that first declaration, your program would reject the change (and in strict mode, fail with an error—see "Strict Mode" on page 45 in Chapter 2).

By the way, that kind of "protection" against mistakes is similar to the static-typing type enforcement, so you can see why static types in other languages can be attractive!

For more information about how different values in variables can be used in your programs, see the *Types & Grammar* title of this series.

Blocks

The phone store employee must go through a series of steps to complete the checkout as you buy your new phone.

Similarly, in code we often need to group a series of statements together, which we often call a *block*. In JavaScript, a block is defined by wrapping one or more statements inside a curly-brace pair `{ .. }`. Consider:

```
var amount = 99.99;

// a general block
{
    amount = amount * 2;
    console.log( amount );  // 199.98
}
```

This kind of standalone `{ .. }` general block is valid, but isn't as commonly seen in JS programs. Typically, blocks are attached to some other control statement, such as an `if` statement (see "Conditionals" on page 18) or a loop (see "Loops" on page 20). For example:

```
var amount = 99.99;

// is amount big enough?
if (amount > 10) {          // <-- block attached to `if`
    amount = amount * 2;
    console.log( amount );  // 199.98
}
```

We'll explain `if` statements in the next section, but as you can see, the `{ .. }` block with its two statements is attached to `if (amount > 10)`; the statements inside the block will only be processed if the conditional passes.

Unlike most other statements like `console.log(amount);`, a block statement does not need a semicolon (`;`) to conclude it.

Conditionals

"Do you want to add on the extra screen protectors to your purchase, for $9.99?" The helpful phone store employee has asked you to make a decision. And you may need to first consult the current *state* of your wallet or bank account to answer that question. But obviously, this is just a simple "yes or no" question.

There are quite a few ways we can express *conditionals* (aka decisions) in our programs.

The most common one is the `if` statement. Essentially, you're saying, "*If* this condition is true, do the following…". For example:

```
var bank_balance = 302.13;
var amount = 99.99;

if (amount < bank_balance) {
    console.log( "I want to buy this phone!" );
}
```

The `if` statement requires an expression in between the parentheses `( )` that can be treated as either `true` or `false`. In this program, we provided the expression `amount < bank_balance`, which indeed will either evaluate to `true` or `false`, depending on the amount in the `bank_balance` variable.

You can even provide an alternative if the condition isn't true, called an `else` clause. Consider:

```
const ACCESSORY_PRICE = 9.99;

var bank_balance = 302.13;
var amount = 99.99;

amount = amount * 2;

// can we afford the extra purchase?
if ( amount < bank_balance ) {
    console.log( "I'll take the accessory!" );
    amount = amount + ACCESSORY_PRICE;
}
// otherwise:
else {
    console.log( "No, thanks." );
}
```

Here, if `amount < bank_balance` is `true`, we'll print out `"I'll take the accessory!"` and add the `9.99` to our `amount` variable. Otherwise, the `else` clause says we'll just politely respond with `"No, thanks."` and leave `amount` unchanged.

As we discussed in "Values & Types" on page 10, values that aren't already of an expected type are often coerced to that type. The `if` statement expects a `boolean`, but if you pass it something that's not already `boolean`, coercion will occur.

JavaScript defines a list of specific values that are considered "falsy" because when coerced to a `boolean`, they become `false`—these include values like `0` and `""`. Any other value not on the "falsy" list is automatically "truthy"—when coerced to a `boolean` they become `true`. Truthy values include things like `99.99` and `"free"`. See "Truthy & falsy" on page 36 in Chapter 2 for more information.

Conditionals exist in other forms besides the `if`. For example, the `switch` statement can be used as a shorthand for a series of `if..else` statements (see Chapter 2). Loops (see "Loops" on page 20) use a *conditional* to determine if the loop should keep going or stop.

For deeper information about the coercions that can occur implicitly in the test expressions of *conditionals*, see Chapter 4 of the *Types & Grammar* title of this series.

Loops

During busy times, there's a waiting list for customers who need to speak to the phone store employee. While there's still people on that list, she just needs to keep serving the next customer.

Repeating a set of actions until a certain condition fails—in other words, repeating only while the condition holds—is the job of programming loops; loops can take different forms, but they all satisfy this basic behavior.

A loop includes the test condition as well as a block (typically as `{ .. }`). Each time the loop block executes, that's called an *iteration*.

For example, the `while` loop and the `do..while` loop forms illustrate the concept of repeating a block of statements until a condition no longer evaluates to `true`:

```
while (numOfCustomers > 0) {
    console.log( "How may I help you?" );

    // help the customer...

    numOfCustomers = numOfCustomers - 1;
}

// versus:

do {
    console.log( "How may I help you?" );

    // help the customer...

    numOfCustomers = numOfCustomers - 1;
} while (numOfCustomers > 0);
```

The only practical difference between these loops is whether the conditional is tested before the first iteration (`while`) or after the first iteration (`do..while`).

In either form, if the conditional tests as `false`, the next iteration will not run. That means if the condition is initially `false`, a `while` loop will never run, but a `do..while` loop will run just the first time.

Sometimes you are looping for the intended purpose of counting a certain set of numbers, like from `0` to `9` (10 numbers). You can do that by setting a loop iteration variable like `i` at value `0` and incrementing it by `1` each iteration.

For a variety of historical reasons, programming languages almost always count things in a zero-based fashion, meaning starting with `0` instead of `1`. If you're not familiar with that mode of thinking, it can be quite confusing at first. Take some time to practice counting starting with `0` to become more comfortable with it!

The conditional is tested on each iteration, much as if there is an implied `if` statement inside the loop.

We can use JavaScript's `break` statement to stop a loop. Also, we can observe that it's awfully easy to create a loop that would otherwise run forever without a `breaking` mechanism.

Let's illustrate:

```
var i = 0;

// a `while..true` loop would run forever, right?
while (true) {
    // keep the loop going?
    if (i <= 9) {
        console.log( i );
        i = i + 1;
    }
    // time to stop the loop!
    else {
        break;
    }
}
// 0 1 2 3 4 5 6 7 8 9
```

This is not necessarily a practical form you'd want to use for your loops. It's presented here for illustration purposes only.

While a `while` (or `do..while`) can accomplish the task manually, there's another syntactic form called a `for` loop for just that purpose:

```
for (var i = 0; i <= 9; i = i + 1) {
    console.log( i );
}
// 0 1 2 3 4 5 6 7 8 9
```

As you can see, in both cases the conditional `i <= 9` is `true` for the first 10 iterations (`i` of values `0` through `9`) of either loop form, but becomes `false` once `i` is value `10`.

The `for` loop has three clauses: the initialization clause (`var i=0`), the conditional test clause (`i <= 9`), and the update clause (`i = i + 1`). So if you're going to do counting with your loop iterations, `for` is a more compact and often easier form to understand and write.

There are other specialized loop forms that are intended to iterate over specific values, such as the properties of an object (see Chapter 2) where the implied conditional test is just whether all the properties have been processed. The "loop until a condition fails" concept holds no matter what the form of the loop.

Functions

The phone store employee probably doesn't carry around a calculator to figure out the taxes and final purchase amount. That's a task she needs to define once and reuse over and over again. Odds are, the company has a checkout register (computer, tablet, etc.) with those "functions" built in.

Similarly, your program will almost certainly want to break up the code's tasks into reusable pieces, instead of repeatedly repeating yourself repetitiously (pun intended!). The way to do this is to define a `function`.

A function is generally a named section of code that can be "called" by name, and the code inside it will be run each time. Consider:

```
function printAmount() {
    console.log( amount.toFixed( 2 ) );
}

var amount = 99.99;

printAmount(); // "99.99"
```

```
amount = amount * 2;

printAmount(); // "199.98"
```

Functions can optionally take arguments (aka parameters)—values you pass in. And they can also optionally return a value back:

```
function printAmount(amt) {
    console.log( amt.toFixed( 2 ) );
}

function formatAmount() {
    return "$" + amount.toFixed( 2 );
}

var amount = 99.99;

printAmount( amount * 2 );      // "199.98"

amount = formatAmount();
console.log( amount );    // "$99.99"
```

The function `printAmount(..)` takes a parameter that we call `amt`. The function `formatAmount()` returns a value. Of course, you can also combine those two techniques in the same function.

Functions are often used for code that you plan to call multiple times, but they can also be useful just to organize related bits of code into named collections, even if you only plan to call them once.

Consider:

```
const TAX_RATE = 0.08;

function calculateFinalPurchaseAmount(amt) {
    // calculate the new amount with the tax
    amt = amt + (amt * TAX_RATE);

    // return the new amount
    return amt;
}

var amount = 99.99;

amount = calculateFinalPurchaseAmount( amount );

console.log( amount.toFixed( 2 ) );     // "107.99"
```

Although `calculateFinalPurchaseAmount(..)` is only called once, organizing its behavior into a separate named function makes the code that uses its logic (the `amount = calculateFinal...` state-

ment) cleaner. If the function had more statements in it, the benefits would be even more pronounced.

Scope

If you ask the phone store employee for a phone model that her store doesn't carry, she will not be able to sell you the phone you want. She only has access to the phones in her store's inventory. You'll have to try another store to see if you can find the phone you're looking for.

Programming has a term for this concept: *scope* (technically called *lexical scope*). In JavaScript, each function gets its own scope. Scope is basically a collection of variables as well as the rules for how those variables are accessed by name. Only code inside that function can access that function's *scoped* variables.

A variable name has to be unique within the same scope—there can't be two different `a` variables sitting right next to each other. But the same variable name `a` could appear in different scopes:

```
function one() {
    // this `a` only belongs to the `one()` function
    var a = 1;
    console.log( a );
}

function two() {
    // this `a` only belongs to the `two()` function
    var a = 2;
    console.log( a );
}

one();      // 1
two();      // 2
```

Also, a scope can be nested inside another scope, just like if a clown at a birthday party blows up one balloon inside another balloon. If one scope is nested inside another, code inside the innermost scope can access variables from either scope.

Consider:

```
function outer() {
    var a = 1;

    function inner() {
        var b = 2;

        // we can access both `a` and `b` here
        console.log( a + b );   // 3
    }

    inner();

    // we can only access `a` here
    console.log( a );           // 1
}

outer();
```

Lexical scope rules say that code in one scope can access variables of either that scope or any scope outside of it.

So, code inside the `inner()` function has access to both variables `a` and `b`, but code only in `outer()` has access only to `a`—it cannot access `b` because that variable is only inside `inner()`.

Recall this code snippet from earlier:

```
const TAX_RATE = 0.08;

function calculateFinalPurchaseAmount(amt) {
    // calculate the new amount with the tax
    amt = amt + (amt * TAX_RATE);

    // return the new amount
    return amt;
}
```

The `TAX_RATE` constant (variable) is accessible from inside the `calculateFinalPurchaseAmount(..)` function, even though we didn't pass it in, because of lexical scope.

For more information about lexical scope, see the first three chapters of the *Scope & Closures* title of this series.

Practice

There is absolutely no substitute for practice in learning programming. No amount of articulate writing on my part is alone going to make you a programmer.

With that in mind, let's try practicing some of the concepts we learned here in this chapter. I'll give the "requirements," and you try it first. Then consult the code listing below to see how I approached it:

- Write a program to calculate the total price of your phone purchase. You will keep purchasing phones (hint: loop!) until you run out of money in your bank account. You'll also buy accessories for each phone as long as your purchase amount is below your mental spending threshold.
- After you've calculated your purchase amount, add in the tax, then print out the calculated purchase amount, properly formatted.
- Finally, check the amount against your bank account balance to see if you can afford it or not.
- You should set up some constants for the "tax rate," "phone price," "accessory price," and "spending threshold," as well as a variable for your "bank account balance."
- You should define functions for calculating the tax and for formatting the price with a "$" and rounding to two decimal places.
- **Bonus Challenge:** Try to incorporate input into this program, perhaps with the `prompt(..)` covered in "Input" on page 6. You may prompt the user for their bank account balance, for example. Have fun and be creative!

OK, go ahead. Try it. Don't peek at my code listing until you've given it a shot yourself!

Because this is a JavaScript book, I'm obviously going to solve the practice exercise in JavaScript. But you can do it in another language for now if you feel more comfortable.

Here's my JavaScript solution for this exercise:

```
const SPENDING_THRESHOLD = 200;
const TAX_RATE = 0.08;
const PHONE_PRICE = 99.99;
const ACCESSORY_PRICE = 9.99;

var bank_balance = 303.91;
var amount = 0;

function calculateTax(amount) {
    return amount * TAX_RATE;
}

function formatAmount(amount) {
    return "$" + amount.toFixed( 2 );
}

// keep buying phones while you still have money
while (amount < bank_balance) {
    // buy a new phone!
    amount = amount + PHONE_PRICE;

    // can we afford the accessory?
    if (amount < SPENDING_THRESHOLD) {
        amount = amount + ACCESSORY_PRICE;
    }
}

// don't forget to pay the government, too
amount = amount + calculateTax( amount );

console.log(
    "Your purchase: " + formatAmount( amount )
);
// Your purchase: $334.76

// can you actually afford this purchase?
if (amount > bank_balance) {
    console.log(
        "You can't afford this purchase. :("
    );
}
// You can't afford this purchase. :(
```

The simplest way to run this JavaScript program is to type it into the developer console of your nearest browser.

How did you do? It wouldn't hurt to try it again now that you've seen my code. And play around with changing some of the constants to see how the program runs with different values.

Review

Learning programming doesn't have to be a complex and overwhelming process. There are just a few basic concepts you need to wrap your head around.

These act like building blocks. To build a tall tower, you start first by putting block on top of block on top of block. The same goes with programming. Here are some of the essential programming building blocks:

- You need *operators* to perform actions on.
- You need values and *types* to perform different kinds of actions like math on `numbers` or output with `strings`.
- You need *variables* to store data (aka *state*) during your program's execution.
- You need *conditionals* like `if` statements to make decisions.
- You need *loops* to repeat tasks until a condition stops being true.
- You need *functions* to organize your code into logical and reusable chunks.

Code comments are one effective way to write more readable code, which makes your program easier to understand, maintain, and fix later if there are problems.

Finally, don't neglect the power of practice. The best way to learn how to write code is to write code.

I'm excited you're well on your way to learning how to code, now! Keep it up. Don't forget to check out other beginner programming resources (books, blogs, online training, etc.). This chapter and this book are a great start, but they're just a brief introduction.

The next chapter will review many of the concepts from this chapter, but from a more JavaScript-specific perspective, which will highlight most of the major topics that are addressed in deeper detail throughout the rest of the series.

CHAPTER 2
Into JavaScript

In the previous chapter, I introduced the basic building blocks of programming, such as variables, loops, conditionals, and functions. Of course, all the code shown has been in JavaScript. But in this chapter, we want to focus specifically on things you need to know about JavaScript to get up and going as a JS developer.

We will introduce quite a few concepts in this chapter that will not be fully explored until subsequent *YDKJS* books. You can think of this chapter as an overview of the topics covered in detail throughout the rest of this series.

Especially if you're new to JavaScript, you should expect to spend quite a bit of time reviewing the concepts and code examples here multiple times. Any good foundation is laid brick by brick, so don't expect that you'll immediately understand it all the first pass through.

Your journey to deeply learn JavaScript starts here.

As I said in Chapter 1, you should definitely try all this code yourself as you read and work through this chapter. Be aware that some of the code here assumes capabilities introduced in the newest version of JavaScript at the time of this writing (commonly referred to as "ES6" for the 6th edition of ECMAScript—the official name of the JS specification). If you happen to be using an older, pre-ES6 browser, the code may not work. A recent update of a modern browser (like Chrome, Firefox, or IE) should be used.

Values & Types

As we asserted in Chapter 1, JavaScript has typed values, not typed variables. The following built-in types are available:

- `string`
- `number`
- `boolean`
- `null` and `undefined`
- `object`
- `symbol` (new to ES6)

JavaScript provides a `typeof` operator that can examine a value and tell you what type it is:

```
var a;
typeof a;               // "undefined"

a = "hello world";
typeof a;               // "string"

a = 42;
typeof a;               // "number"

a = true;
typeof a;               // "boolean"

a = null;
typeof a;               // "object"--weird, bug

a = undefined;
typeof a;               // "undefined"
```

```
a = { b: "c" };
typeof a;                // "object"
```

The return value from the `typeof` operator is always one of six (seven as of ES6!) string values. That is, `typeof "abc"` returns `"string"`, not `string`.

Notice how in this snippet the `a` variable holds every different type of value, and that despite appearances, `typeof a` is not asking for the "type of `a`," but rather for the "type of the value currently in `a`." Only values have types in JavaScript; variables are just simple containers for those values.

`typeof null` is an interesting case because it errantly returns `"object"` when you'd expect it to return `"null"`.

This is a long-standing bug in JS, but one that is likely never going to be fixed. Too much code on the Web relies on the bug, and thus fixing it would cause a lot more bugs!

Also, note `a = undefined`. We're explicitly setting `a` to the `undefined` value, but that is behaviorally no different from a variable that has no value set yet, like with the `var a;` line at the top of the snippet. A variable can get to this "undefined" value state in several different ways, including functions that return no values and usage of the `void` operator.

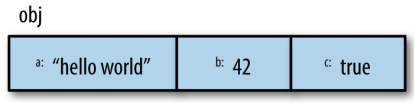
Objects

The `object` type refers to a compound value where you can set properties (named locations) that each hold their own values of any type. This is perhaps one of the most useful value types in all of JavaScript:

```
var obj = {
    a: "hello world",
    b: 42,
    c: true
};

obj.a;      // "hello world"
obj.b;      // 42
obj.c;      // true
```

```
obj["a"];   // "hello world"
obj["b"];   // 42
obj["c"];   // true
```

It may be helpful to think of this `obj` value visually:



Properties can either be accessed with *dot notation* (i.e., `obj.a`) or *bracket notation* (i.e., `obj["a"]`). Dot notation is shorter and generally easier to read, and is thus preferred when possible.

Bracket notation is useful if you have a property name that has special characters in it, like `obj["hello world!"]`—such properties are often referred to as *keys* when accessed via bracket notation. The `[ ]` notation requires either a variable (explained next) or a `string` *literal* (which needs to be wrapped in `" .. "` or `' .. '`).

Of course, bracket notation is also useful if you want to access a property/key but the name is stored in another variable, such as:

```
var obj = {
    a: "hello world",
    b: 42
};

var b = "a";

obj[b];         // "hello world"
obj["b"];       // 42
```

For more information on JavaScript `objects`, see the *this & Object Prototypes* title of this series, specifically Chapter 3.

There are a couple of other value types that you will commonly interact with in JavaScript programs: *array* and *function*. But rather than being proper built-in types, these should be thought of more like subtypes—specialized versions of the `object` type.

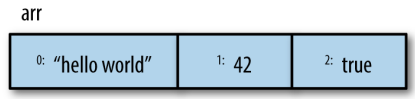
Arrays

An array is an `object` that holds values (of any type) not particularly in named properties/keys, but rather in numerically indexed positions. For example:

```
var arr = [
    "hello world",
    42,
    true
];

arr[0];         // "hello world"
arr[1];         // 42
arr[2];         // true
arr.length;     // 3

typeof arr;     // "object"
```

Languages that start counting at zero, like JS does, use `0` as the index of the first element in the array.

It may be helpful to think of `arr` visually:



Because arrays are special objects (as `typeof` implies), they can also have properties, including the automatically updated `length` property.

You theoretically could use an array as a normal object with your own named properties, or you could use an `object` but only give it numeric properties (`0`, `1`, etc.) similar to an array. However, this would generally be considered improper usage of the respective types.

The best and most natural approach is to use arrays for numerically positioned values and use `objects` for named properties.

Functions

The other `object` subtype you'll use all over your JS programs is a function:

```
function foo() {
    return 42;
}

foo.bar = "hello world";

typeof foo;         // "function"
typeof foo();       // "number"
typeof foo.bar;     // "string"
```

Again, functions are a subtype of `objects`—`typeof` returns `"function"`, which implies that a `function` is a main type—and can thus have properties, but you typically will only use function object properties (like `foo.bar`) in limited cases.

For more information on JS values and their types, see the first two chapters of the *Types & Grammar* title of this series.

Built-In Type Methods

The built-in types and subtypes we've just discussed have behaviors exposed as properties and methods that are quite powerful and useful.

For example:

```
var a = "hello world";
var b = 3.14159;

a.length;               // 11
a.toUpperCase();        // "HELLO WORLD"
b.toFixed(4);           // "3.1416"
```

The "how" behind being able to call `a.toUpperCase()` is more complicated than just that method existing on the value.

Briefly, there is a `String` (capital `S`) object wrapper form, typically called a "native," that pairs with the primitive `string` type; it's this object wrapper that defines the `toUpperCase()` method on its prototype.

When you use a primitive value like `"hello world"` as an `object` by referencing a property or method (e.g., `a.toUpperCase()` in the previous snippet), JS automatically "boxes" the value to its object wrapper counterpart (hidden under the covers).

A `string` value can be wrapped by a `String` object, a `number` can be wrapped by a `Number` object, and a `boolean` can be wrapped by a `Boolean` object. For the most part, you don't need to worry about or directly use these object wrapper forms of the values—prefer the primitive value forms in practically all cases and JavaScript will take care of the rest for you.

For more information on JS natives and "boxing," see Chapter 3 of the *Types & Grammar* title of this series. To better understand the prototype of an object, see Chapter 5 of the *this & Object Prototypes* title of this series.

Comparing Values

There are two main types of value comparison that you will need to make in your JS programs: *equality* and *inequality*. The result of any comparison is a strictly `boolean` value (`true` or `false`), regardless of what value types are compared.

Coercion

We talked briefly about coercion in Chapter 1, but let's revisit it here.

Coercion comes in two forms in JavaScript: *explicit* and *implicit*. Explicit coercion is simply that you can see from the code that a conversion from one type to another will occur, whereas implicit coercion is when the type conversion can happen as more of a non-obvious side effect of some other operation.

You've probably heard sentiments like "coercion is evil" drawn from the fact that there are clearly places where coercion can produce some surprising results. Perhaps nothing evokes frustration from developers more than when the language surprises them.

Coercion is not evil, nor does it have to be surprising. In fact, the majority of cases you can construct with type coercion are quite sensible and understandable, and can even be used to *improve* the readability of your code. But we won't go much further into that

debate—Chapter 4 of the *Types & Grammar* title of this series covers all sides.

Here's an example of *explicit* coercion:

```
var a = "42";

var b = Number( a );

a;              // "42"
b;              // 42--the number!
```

And here's an example of *implicit* coercion:

```
var a = "42";

var b = a * 1;  // "42" implicitly coerced to 42 here

a;              // "42"
b;              // 42--the number!
```

Truthy & falsy

In Chapter 1, we briefly mentioned the "truthy" and "falsy" nature of values: when a non-`boolean` value is coerced to a `boolean`, does it become `true` or `false`, respectively?

The specific list of "falsy" values in JavaScript is as follows:

- `""` (empty string)
- `0`, `-0`, `NaN` (invalid `number`)
- `null`, `undefined`
- `false`

Any value that's not on this "falsy" list is "truthy." Here are some examples of those:

- `"hello"`
- `42`
- `true`
- `[ ]`, `[ 1, "2", 3 ]` (arrays)
- `{ }`, `{ a: 42 }` (objects)
- `function foo() { .. }` (functions)

It's important to remember that a non-`boolean` value only follows this "truthy"/"falsy" coercion if it's actually coerced to a `boolean`. It's not all that difficult to confuse yourself with a situation that seems like it's coercing a value to a `boolean` when it's not.

Equality

There are four equality operators: `==`, `===`, `!=`, and `!==`. The `!` forms are of course the symmetric "not equal" versions of their counterparts; *non-equality* should not be confused with *inequality*.

The difference between `==` and `===` is usually characterized that `==` checks for value equality and `===` checks for both value and type equality. However, this is inaccurate. The proper way to characterize them is that `==` checks for value equality with coercion allowed, and `===` checks for value equality without allowing coercion; `===` is often called "strict equality" for this reason.

Consider the implicit coercion that's allowed by the `==` loose-equality comparison and not allowed with the `===` strict-equality:

```
var a = "42";
var b = 42;

a == b;         // true
a === b;        // false
```

In the `a == b` comparison, JS notices that the types do not match, so it goes through an ordered series of steps to coerce one or both values to a different type until the types match, where then a simple value equality can be checked.

If you think about it, there's two possible ways `a == b` could give `true` via coercion. Either the comparison could end up as `42 == 42` or it could be `"42" == "42"`. So which is it?

The answer: `"42"` becomes `42`, to make the comparison `42 == 42`. In such a simple example, it doesn't really seem to matter which way that process goes, as the end result is the same. There are more complex cases where it matters not just what the end result of the comparison is, but *how* you get there.

The `a === b` produces `false`, because the coercion is not allowed, so the simple value comparison obviously fails. Many developers feel that `===` is more predictable, so they advocate always using that form and staying away from `==`. I think this view is very shortsighted. I

believe == is a powerful tool that helps your program, *if you take the time to learn how it works.*

We're not going to cover all the nitty-gritty details of how the coercion in == comparisons works here. Much of it is pretty sensible, but there are some important corner cases to be careful of. You can read section 11.9.3 of the ES5 specification to see the exact rules, and you'll be surprised at just how straightforward this mechanism is, compared to all the negative hype surrounding it.

To boil down a whole lot of details to a few simple takeaways, and help you know whether to use == or === in various situations, here are my simple rules:

- If either value (aka side) in a comparison could be the `true` or `false` value, avoid == and use ===.
- If either value in a comparison could be of these specific values (`0`, `""`, or `[]`—empty array), avoid == and use ===.
- In *all* other cases, you're safe to use ==. Not only is it safe, but in many cases it simplifies your code in a way that improves readability.

What these rules boil down to is requiring you to think critically about your code and about what kinds of values can come through variables that get compared for equality. If you can be certain about the values, and == is safe, use it! If you can't be certain about the values, use ===. It's that simple.

The `!=` non-equality form pairs with ==, and the `!==` form pairs with ===. All the rules and observations we just discussed hold symmetrically for these non-equality comparisons.

You should take special note of the == and === comparison rules if you're comparing two non-primitive values, like `objects` (including `function` and `array`). Because those values are actually held by reference, both == and === comparisons will simply check whether the references match, not anything about the underlying values.

For example, `arrays` are by default coerced to `strings` by simply joining all the values with commas (`,`) in between. You might think that two `arrays` with the same contents would be == equal, but they're not:

```
var a = [1,2,3];
var b = [1,2,3];
var c = "1,2,3";

a == c;     // true
b == c;     // true
a == b;     // false
```

For more information about the == equality comparison rules, see the ES5 specification (section 11.9.3) and also consult Chapter 4 of the *Types & Grammar* title of this series; see Chapter 2 for more information about values versus references.

Inequality

The <, >, <=, and >= operators are used for inequality, referred to in the specification as "relational comparison." Typically they will be used with ordinally comparable values like `numbers`. It's easy to understand that `3 < 4`.

But JavaScript `string` values can also be compared for inequality, using typical alphabetic rules (`"bar" < "foo"`).

What about coercion? Similar rules as == comparison (though not exactly identical!) apply to the inequality operators. Notably, there are no "strict inequality" operators that would disallow coercion the same way === "strict equality" does.

Consider:

```
var a = 41;
var b = "42";
var c = "43";

a < b;      // true
b < c;      // true
```

What happens here? In section 11.8.5 of the ES5 specification, it says that if both values in the < comparison are `strings`, as it is with `b < c`, the comparison is made lexicographically (aka alphabetically like a dictionary). But if one or both is not a `string`, as it is with `a < b`, then both values are coerced to be `numbers`, and a typical numeric comparison occurs.

The biggest gotcha you may run into here with comparisons between potentially different value types—remember, there are no "strict inequality" forms to use—is when one of the values cannot be made into a valid number, such as:

```
var a = 42;
var b = "foo";

a < b;      // false
a > b;      // false
a == b;     // false
```

Wait, how can all three of those comparisons be `false`? Because the `b` value is being coerced to the "invalid number value" `NaN` in the `<` and `>` comparisons, and the specification says that `NaN` is neither greater than nor less than any other value.

The `==` comparison fails for a different reason. `a == b` could fail if it's interpreted either as `42 == NaN` or `"42" == "foo"`—as we explained earlier, the former is the case.

For more information about the inequality comparison rules, see section 11.8.5 of the ES5 specification and also consult Chapter 4 of the *Types & Grammar* title of this series.

Variables

In JavaScript, variable names (including function names) must be valid *identifiers*. The strict and complete rules for valid characters in identifiers are a little complex when you consider nontraditional characters such as Unicode. If you only consider typical ASCII alphanumeric characters, though, the rules are simple.

An identifier must start with `a-z`, `A-Z`, `$`, or `_`. It can then contain any of those characters plus the numerals `0-9`.

Generally, the same rules apply to a property name as to a variable identifier. However, certain words cannot be used as variables, but are OK as property names. These words are called "reserved words," and include the JS keywords (`for`, `in`, `if`, etc.) as well as `null`, `true`, and `false`.

For more information about reserved words, see Appendix A of the *Types & Grammar* title of this series.

Function Scopes

You use the `var` keyword to declare a variable that will belong to the current function scope, or the global scope if at the top level outside of any function.

Hoisting

Wherever a `var` appears inside a scope, that declaration is taken to belong to the entire scope and accessible everywhere throughout.

Metaphorically, this behavior is called *hoisting*, when a `var` declaration is conceptually "moved" to the top of its enclosing scope. Technically, this process is more accurately explained by how code is compiled, but we can skip over those details for now.

Consider:

```
var a = 2;

foo();                  // works because `foo()`
                        // declaration is "hoisted"

function foo() {
    a = 3;

    console.log( a );   // 3

    var a;              // declaration is "hoisted"
                        // to the top of `foo()`
}

console.log( a );   // 2
```

It's not common or a good idea to rely on variable *hoisting* to use a variable earlier in its scope than its `var` declaration appears; it can be quite confusing. It's much more common and accepted to use *hoisted* function declarations, as we do with the `foo()` call appearing before its formal declaration.

Nested scopes

When you declare a variable, it is available anywhere in that scope, as well as any lower/inner scopes. For example:

```
function foo() {
    var a = 1;

    function bar() {
        var b = 2;

        function baz() {
            var c = 3;

            console.log( a, b, c ); // 1 2 3
        }

        baz();
        console.log( a, b );        // 1 2
    }

    bar();
    console.log( a );               // 1
}

foo();
```

Notice that `c` is not available inside of `bar()`, because it's declared only inside the inner `baz()` scope, and that `b` is not available to `foo()` for the same reason.

If you try to access a variable's value in a scope where it's not available, you'll get a `ReferenceError` thrown. If you try to set a variable that hasn't been declared, you'll either end up creating a variable in the top-level global scope (bad!) or getting an error, depending on "strict mode" (see "Strict Mode" on page 45). Let's take a look:

```
function foo() {
    a = 1;  // `a` not formally declared
}

foo();
a;          // 1--oops, auto global variable :(
```

This is a very bad practice. Don't do it! Always formally declare your variables.

In addition to creating declarations for variables at the function level, ES6 *lets* you declare variables to belong to individual blocks (pairs of `{ .. }`), using the `let` keyword. Besides some nuanced

details, the scoping rules will behave roughly the same as we just saw with functions:

```
function foo() {
    var a = 1;

    if (a >= 1) {
        let b = 2;

        while (b < 5) {
            let c = b * 2;
            b++;

            console.log( a + c );
        }
    }
}

foo();
// 5 7 9
```

Because of using `let` instead of `var`, `b` will belong only to the `if` statement and thus not to the whole `foo()` function's scope. Similarly, `c` belongs only to the `while` loop. Block scoping is very useful for managing your variable scopes in a more fine-grained fashion, which can make your code much easier to maintain over time.

For more information about scope, see the *Scope & Closures* title of this series. See the *ES6 & Beyond* title of this series for more information about `let` block scoping.

Conditionals

In addition to the `if` statement we introduced briefly in Chapter 1, JavaScript provides a few other conditionals mechanisms that we should take a look at.

Sometimes you may find yourself writing a series of `if..else..if` statements like this:

```
if (a == 2) {
    // do something
}
else if (a == 10) {
    // do another thing
}
```

```
else if (a == 42) {
    // do yet another thing
}
else {
    // fallback to here
}
```

This structure works, but it's a little verbose because you need to specify the `a` test for each case. Here's another option, the `switch` statement:

```
switch (a) {
    case 2:
        // do something
        break;
    case 10:
        // do another thing
        break;
    case 42:
        // do yet another thing
        break;
    default:
        // fallback to here
}
```

The `break` is important if you want only the statement(s) in one `case` to run. If you omit `break` from a `case`, and that `case` matches or runs, execution will continue with the next `case`'s statements regardless of that `case` matching. This so called "fall through" is sometimes useful/desired:

```
switch (a) {
    case 2:
    case 10:
        // some cool stuff
        break;
    case 42:
        // other stuff
        break;
    default:
        // fallback
}
```

Here, if `a` is either `2` or `10`, it will execute the "some cool stuff" code statements.

Another form of conditional in JavaScript is the "conditional operator," often called the "ternary operator." It's like a more concise form of a single `if..else` statement, such as:

```
var a = 42;

var b = (a > 41) ? "hello" : "world";

// similar to:

// if (a > 41) {
//    b = "hello";
// }
// else {
//    b = "world";
// }
```

If the test expression (`a > 41` here) evaluates as `true`, the first clause (`"hello"`) results; otherwise, the second clause (`"world"`) results, and whatever the result is then gets assigned to `b`.

The conditional operator doesn't have to be used in an assignment, but that's definitely the most common usage.

For more information about testing conditions and other patterns for `switch` and `? :`, see the *Types & Grammar* title of this series.

Strict Mode

ES5 added a "strict mode" to the language, which tightens the rules for certain behaviors. Generally, these restrictions are seen as keeping the code to a safer and more appropriate set of guidelines. Also, adhering to strict mode makes your code generally more optimizable by the engine. Strict mode is a big win for code, and you should use it for all your programs.

You can opt in to strict mode for an individual function, or an entire file, depending on where you put the strict mode pragma:

```
function foo() {
    "use strict";

    // this code is strict mode

    function bar() {
        // this code is strict mode
    }
}
```

```
// this code is not strict mode
```

Compare that to:

```
"use strict";

function foo() {
    // this code is strict mode

    function bar() {
        // this code is strict mode
    }
}

// this code is strict mode
```

One key difference (improvement!) with strict mode is disallowing the implicit auto-global variable declaration from omitting the `var`:

```
function foo() {
    "use strict";   // turn on strict mode
    a = 1;          // `var` missing, ReferenceError
}

foo();
```

If you turn on strict mode in your code, and you get errors, or code starts behaving buggy, your temptation might be to avoid strict mode. But that instinct would be a bad idea to indulge. If strict mode causes issues in your program, it's almost certainly a sign that you have things in your program you should fix.

Not only will strict mode keep your code to a safer path, and not only will it make your code more optimizable, but it also represents the future direction of the language. It'd be easier on you to get used to strict mode now than to keep putting it off—it'll only get harder to convert later!

For more information about strict mode, see Chapter 5 of the *Types & Grammar* title of this series.

Functions as Values

So far, we've discussed functions as the primary mechanism of *scope* in JavaScript. You recall typical `function` declaration syntax as follows:

```
function foo() {
    // ..
}
```

Though it may not seem obvious from that syntax, `foo` is basically just a variable in the outer enclosing scope that's given a reference to the `function` being declared. That is, the `function` itself is a value, just like `42` or `[1,2,3]` would be.

This may sound like a strange concept at first, so take a moment to ponder it. Not only can you pass a value (argument) *to* a function, but *a function itself can be a value* that's assigned to variables or passed to or returned from other functions.

As such, a function value should be thought of as an expression, much like any other value or expression.

Consider:

```
var foo = function() {
    // ..
};

var x = function bar(){
    // ..
};
```

The first function expression assigned to the `foo` variable is called *anonymous* because it has no `name`.

The second function expression is *named* (`bar`), even as a reference to it is also assigned to the `x` variable. *Named function expressions* are generally more preferable, though *anonymous function expressions* are still extremely common.

For more information, see the *Scope & Closures* title of this series.

Immediately Invoked Function Expressions (IIFEs)

In the previous snippet, neither of the function expressions are executed—we could if we had included `foo()` or `x()`, for instance.

There's another way to execute a function expression, which is typically referred to as an *immediately invoked function expression* (IIFE):

```
(function IIFE(){
    console.log( "Hello!" );
})();
// "Hello!"
```

The outer `( .. )` that surrounds the `(function IIFE(){ .. })` function expression is just a nuance of JS grammar needed to prevent it from being treated as a normal function declaration.

The final `()` on the end of the expression—the `})();` line—is what actually executes the function expression referenced immediately before it.

That may seem strange, but it's not as foreign as first glance. Consider the similarities between `foo` and `IIFE` here:

```
function foo() { .. }

// `foo` function reference expression,
// then `()` executes it
foo();

// `IIFE` function expression,
// then `()` executes it
(function IIFE(){ .. })();
```

As you can see, listing the `(function IIFE(){ .. })` before its executing `()` is essentially the same as including `foo` before its executing `()`; in both cases, the function reference is executed with `()` immediately after it.

Because an IIFE is just a function, and functions create variable *scope*, using an IIFE in this fashion is often used to declare variables that won't affect the surrounding code outside the IIFE:

```
var a = 42;

(function IIFE(){
    var a = 10;
    console.log( a );   // 10
})();

console.log( a );       // 42
```

IIFEs can also have return values:

```
var x = (function IIFE(){
    return 42;
})();

x;  // 42
```

The `42` value gets `returned` from the `IIFE`-named function being executed, and is then assigned to `x`.

Closure

Closure is one of the most important, and often least understood, concepts in JavaScript. I won't cover it in deep detail here, and instead refer you to the *Scope & Closures* title of this series. But I want to say a few things about it so you understand the general concept. It will be one of the most important techniques in your JS skillset.

You can think of closure as a way to "remember" and continue to access a function's scope (its variables) even once the function has finished running.

Consider:

```
function makeAdder(x) {
    // parameter `x` is an inner variable

    // inner function `add()` uses `x`, so
    // it has a "closure" over it
    function add(y) {
        return y + x;
    };

    return add;
}
```

The reference to the inner `add(..)` function that gets returned with each call to the outer `makeAdder(..)` is able to remember whatever `x` value was passed in to `makeAdder(..)`. Now, let's use `makeAdder(..)`:

```
// `plusOne` gets a reference to the inner `add(..)`
// function with closure over the `x` parameter of
// the outer `makeAdder(..)`
var plusOne = makeAdder( 1 );

// `plusTen` gets a reference to the inner `add(..)`
// function with closure over the `x` parameter of
// the outer `makeAdder(..)`
```

```
var plusTen = makeAdder( 10 );

plusOne( 3 );         // 4  <-- 1 + 3
plusOne( 41 );        // 42 <-- 1 + 41

plusTen( 13 );        // 23 <-- 10 + 13
```

More on how this code works:

1. When we call `makeAdder(1)`, we get back a reference to its inner `add(..)` that remembers `x` as `1`. We call this function reference `plusOne(..)`.
2. When we call `makeAdder(10)`, we get back another reference to its inner `add(..)` that remembers `x` as `10`. We call this function reference `plusTen(..)`.
3. When we call `plusOne(3)`, it adds `3` (its inner `y`) to the `1` (remembered by `x`), and we get `4` as the result.
4. When we call `plusTen(13)`, it adds `13` (its inner `y`) to the `10` (remembered by `x`), and we get `23` as the result.

Don't worry if this seems strange and confusing at first—it can be! It'll take lots of practice to understand it fully.

But trust me, once you do, it's one of the most powerful and useful techniques in all of programming. It's definitely worth the effort to let your brain simmer on closures for a bit. In the next section, we'll get a little more practice with closure.

Modules

The most common usage of closure in JavaScript is the module pattern. Modules let you define private implementation details (variables, functions) that are hidden from the outside world, as well as a public API that *is* accessible from the outside.

Consider:

```
function User(){
    var username, password;

    function doLogin(user,pw) {
        username = user;
        password = pw;

        // do the rest of the login work
    }
```

```
    var publicAPI = {
        login: doLogin
    };

    return publicAPI;
}

// create a `User` module instance
var fred = User();

fred.login( "fred", "12Battery34!" );
```

The `User()` function serves as an outer scope that holds the variables `username` and `password`, as well as the inner `doLogin()` function; these are all private inner details of this `User` module that cannot be accessed from the outside world.

We are not callling `new User()` here, on purpose, despite the fact that probably seems more common to most readers. `User()` is just a function, not a class to be instantiated, so it's just called normally. Using `new` would be inappropriate and actually waste resources.

Executing `User()` creates an *instance* of the `User` module—a whole new scope is created, and thus a whole new copy of each of these inner variables/functions. We assign this instance to `fred`. If we run `User()` again, we'd get a new instance entirely separate from `fred`.

The inner `doLogin()` function has a closure over `username` and `password`, meaning it will retain its access to them even after the `User()` function finishes running.

`publicAPI` is an object with one property/method on it, `login`, which is a reference to the inner `doLogin()` function. When we return `publicAPI` from `User()`, it becomes the instance we call `fred`.

At this point, the outer `User()` function has finished executing. Normally, you'd think the inner variables like `username` and `password` have gone away. But here they have not, because there's a closure in the `login()` function keeping them alive.

That's why we can call `fred.login(..)`—the same as calling the inner `doLogin(..)`—and it can still access `username` and `password` inner variables.

There's a good chance that with just this brief glimpse at closure and the module pattern, some of it is still a bit confusing. That's OK! It takes some work to wrap your brain around it.

From here, go read the *Scope & Closures* title of this series for a much more in-depth exploration.

this Identifier

Another very commonly misunderstood concept in JavaScript is the `this` keyword. Again, there's a couple of chapters on it in the *this & Object Prototypes* title of this series, so here we'll just briefly introduce the concept.

While it may often seem that `this` is related to "object-oriented patterns," in JS `this` is a different mechanism.

If a function has a `this` reference inside it, that `this` reference usually points to an `object`. But which `object` it points to depends on how the function was called.

It's important to realize that `this` *does not* refer to the function itself, as is the most common misconception.

Here's a quick illustration:

```
function foo() {
    console.log( this.bar );
}

var bar = "global";

var obj1 = {
    bar: "obj1",
    foo: foo
};

var obj2 = {
    bar: "obj2"
};

// --------

foo();              // "global"
```

```
obj1.foo();          // "obj1"
foo.call( obj2 );    // "obj2"
new foo();           // undefined
```

There are four rules for how `this` gets set, and they're shown in those last four lines of that snippet:

1. `foo()` ends up setting `this` to the global object in non-strict mode—in strict mode, `this` would be `undefined` and you'd get an error in accessing the `bar` property—so `"global"` is the value found for `this.bar`.
2. `obj1.foo()` sets `this` to the `obj1` object.
3. `foo.call(obj2)` sets `this` to the `obj2` object.
4. `new foo()` sets `this` to a brand new empty object.

Bottom line: to understand what `this` points to, you have to examine how the function in question was called. It will be one of those four ways just shown, and that will then answer what `this` is.

For more information about `this`, see Chapters 1 and 2 of the *this & Object Prototypes* title of this series.

Prototypes

The prototype mechanism in JavaScript is quite complicated. We will only glance at it here. You will want to spend plenty of time reviewing Chapters 4-6 of the *this & Object Prototypes* title of this series for all the details.

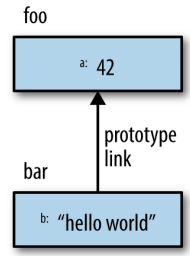
When you reference a property on an object, if that property doesn't exist, JavaScript will automatically use that object's internal prototype reference to find another object to look for the property on. You could think of this almost as a fallback if the property is missing.

The internal prototype reference linkage from one object to its fallback happens at the time the object is created. The simplest way to illustrate it is with a built-in utility called `Object.create(..)`.

Consider:

```
var foo = {
    a: 42
};

// create `bar` and link it to `foo`
var bar = Object.create( foo );

bar.b = "hello world";

bar.b;      // "hello world"
bar.a;      // 42 <-- delegated to `foo`
```

It may help to visualize the `foo` and `bar` objects and their relationship:



The `a` property doesn't actually exist on the `bar` object, but because `bar` is prototype-linked to `foo`, JavaScript automatically falls back to looking for `a` on the `foo` object, where it's found.

This linkage may seem like a strange feature of the language. The most common way this feature is used—and I would argue, abused—is to try to emulate/fake a "class" mechanism with "inheritance."

But a more natural way of applying prototypes is a pattern called "behavior delegation," where you intentionally design your linked objects to be able to *delegate* from one to the other for parts of the needed behavior.

For more information about prototypes and behavior delegation, see Chapters 4-6 of the *this & Object Prototypes* title of this series.

Old & New

Some of the JS features we've already covered, and certainly many of the features covered in the rest of this series, are newer additions and will not necessarily be available in older browsers. In fact, some of the newest features in the specification aren't even implemented in any stable browsers yet.

So, what do you do with the new stuff? Do you just have to wait around for years or decades for all the old browsers to fade into obscurity?

That's how many people think about the situation, but it's really not a healthy approach to JS.

There are two main techniques you can use to "bring" the newer JavaScript stuff to the older browsers: polyfilling and transpiling.

Polyfilling

The word "polyfill" is an invented term (by Remy Sharp) used to refer to taking the definition of a newer feature and producing a piece of code that's equivalent to the behavior, but is able to run in older JS environments.

For example, ES6 defines a utility called `Number.isNaN(..)` to provide an accurate, non-buggy check for `NaN` values, deprecating the original `isNaN(..)` utility. But it's easy to polyfill that utility so that you can start using it in your code regardless of whether the end user is in an ES6 browser or not.

Consider:

```
if (!Number.isNaN) {
    Number.isNaN = function isNaN(x) {
        return x !== x;
    };
}
```

The `if` statement guards against applying the polyfill definition in ES6 browsers where it will already exist. If it's not already present, we define `Number.isNaN(..)`.

The check we do here takes advantage of a quirk with `NaN` values, which is that they're the only value in the whole language that is not equal to itself. So the `NaN` value is the only one that would make `x !== x` be `true`.

Not all new features are fully polyfillable. Sometimes most of the behavior can be polyfilled, but there are still small deviations. You should be really, really careful in implementing a polyfill yourself, to make sure you are adhering to the specification as strictly as possible.

Or better yet, use an already vetted set of polyfills that you can trust, such as those provided by ES5-Shim and ES6-Shim.

Transpiling

There's no way to polyfill new syntax that has been added to the language. The new syntax would throw an error in the old JS engine as unrecognized/invalid.

So the better option is to use a tool that converts your newer code into older code equivalents. This process is commonly called "transpiling," a term for transforming + compiling.

Essentially, your source code is authored in the new syntax form, but what you deploy to the browser is the transpiled code in old syntax form. You typically insert the transpiler into your build process, similar to your code linter or your minifier.

You might wonder why you'd go to the trouble to write new syntax only to have it transpiled away to older code—why not just write the older code directly?

There are several important reasons you should care about transpiling:

- The new syntax added to the language is designed to make your code more readable and maintainable. The older equivalents are often much more convoluted. You should prefer writing newer and cleaner syntax, not only for yourself but for all other members of the development team.
- If you transpile only for older browsers, but serve the new syntax to the newest browsers, you get to take advantage of browser

performance optimizations with the new syntax. This also lets browser makers have more real-world code to test their implementations and optimizations on.

- Using the new syntax earlier allows it to be tested more robustly in the real world, which provides earlier feedback to the JavaScript committee (TC39). If issues are found early enough, they can be changed/fixed before those language design mistakes become permanent.

Here's a quick example of transpiling. ES6 adds a feature called "default parameter values." It looks like this:

```
function foo(a = 2) {
    console.log( a );
}

foo();      // 2
foo( 42 );  // 42
```

Simple, right? Helpful, too! But it's new syntax that's invalid in pre-ES6 engines. So what will a transpiler do with that code to make it run in older environments?

```
function foo() {
    var a = arguments[0] !== (void 0) ? arguments[0] : 2;
    console.log( a );
}
```

As you can see, it checks to see if the `arguments[0]` value is `void 0` (aka `undefined`), and if so provides the `2` default value; otherwise, it assigns whatever was passed.

In addition to being able to now use the nicer syntax even in older browsers, looking at the transpiled code actually explains the intended behavior more clearly.

You may not have realized just from looking at the ES6 version that `undefined` is the only value that can't get explicitly passed in for a default-value parameter, but the transpiled code makes that much more clear.

The last important detail to emphasize about transpilers is that they should now be thought of as a standard part of the JS development ecosystem and process. JS is going to continue to evolve, much more quickly than before, so every few months new syntax and new features will be added.

If you use a transpiler by default, you'll always be able to make that switch to newer syntax whenever you find it useful, rather than always waiting for years for today's browsers to phase out.

There are quite a few great transpilers for you to choose from. Here are some good options at the time of this writing:

Babel (formerly 6to5)
: Transpiles ES6+ into ES5

Traceur
: Transpiles ES6, ES7, and beyond into ES5

Non-JavaScript

So far, the only things we've covered are in the JS language itself. The reality is that most JS is written to run in and interact with environments like browsers. A good chunk of the stuff that you write in your code is, strictly speaking, not directly controlled by JavaScript. That probably sounds a little strange.

The most common non-JavaScript JavaScript you'll encounter is the DOM API. For example:

```
var el = document.getElementById( "foo" );
```

The `document` variable exists as a global variable when your code is running in a browser. It's not provided by the JS engine, nor is it particularly controlled by the JavaScript specification. It takes the form of something that looks an awful lot like a normal JS `object`, but it's not really exactly that. It's a special `object`, often called a "host object."

Moreover, the `getElementByID(..)` method on `document` looks like a normal JS function, but it's just a thinly exposed interface to a built-in method provided by the DOM from your browser. In some (newer-generation) browsers, this layer may also be in JS, but traditionally the DOM and its behavior is implemented in something more like C/C++.

Another example is with input/output (I/O).

Everyone's favorite `alert(..)` pops up a message box in the user's browser window. `alert(..)` is provided to your JS program by the browser, not by the JS engine itself. The call you make sends the

message to the browser internals and it handles drawing and displaying the message box.

The same goes with `console.log(..)`; your browser provides such mechanisms and hooks them up to the developer tools.

This book, and this whole series, focuses on JavaScript the language. That's why you don't see any substantial coverage of these non-JavaScript JavaScript mechanisms. Nevertheless, you need to be aware of them, as they'll be in every JS program you write!

Review

The first step to learning JavaScript's flavor of programming is to get a basic understanding of its core mechanisms like values, types, function closures, `this`, and prototypes.

Of course, each of these topics deserves much greater coverage than you've seen here, but that's why they have chapters and books dedicated to them throughout the rest of this series. After you feel pretty comfortable with the concepts and code samples in this chapter, the rest of the series awaits you to really dig in and get to know the language deeply.

The final chapter of this book will briefly summarize each of the other titles in the series and the other concepts they cover besides what we've already explored.

CHAPTER 3

Into YDKJS

What is this series all about? Put simply, it's about taking seriously the task of learning *all parts of JavaScript*, not just some subset of the language that someone called "the good parts," and not just whatever minimal amount you need to get your job done at work.

Serious developers in other languages expect to put in the effort to learn most or all of the language(s) they primarily write in, but JS developers seem to stand out from the crowd in the sense of typically not learning very much of the language. This is not a good thing, and it's not something we should continue to allow to be the norm.

The *You Don't Know JS* (*YDKJS*) series stands in stark contrast to the typical approaches to learning JS, and is unlike almost any other JS books you will read. It challenges you to go beyond your comfort zone and to ask the deeper "why" questions for every single behavior you encounter. Are you up for that challenge?

I'm going to use this final chapter to briefly summarize what to expect from the rest of the books in the series, and how to most effectively go about building a foundation of JS learning on top of *YDKJS*.

Scope & Closures

Perhaps one of the most fundamental things you'll need to quickly come to terms with is how scoping of variables really works in JavaScript. It's not enough to have anecdotal fuzzy *beliefs* about scope.

The *Scope & Closures* title starts by debunking the common misconception that JS is an "interpreted language" and therefore not compiled. Nope.

The JS engine compiles your code right before (and sometimes during!) execution. So we use some deeper understanding of the compiler's approach to our code to understand how it finds and deals with variable and function declarations. Along the way, we see the typical metaphor for JS variable scope management, "hoisting."

This critical understanding of "lexical scope" is what we then base our exploration of closure on for the last chapter of the book. Closure is perhaps the single most important concept in all of JS, but if you haven't first grasped firmly how scope works, closure will likely remain beyond your grasp.

One important application of closure is the module pattern, as we briefly introduced in this book in Chapter 2. The module pattern is perhaps the most prevalent code organization pattern in all of JavaScript; deep understanding of it should be one of your highest priorities.

this & Object Prototypes

Perhaps one of the most widespread and persistent mistruths about JavaScript is that the `this` keyword refers to the function it appears in. Terribly mistaken.

The `this` keyword is dynamically bound based on how the function in question is executed, and it turns out there are four simple rules to understand and fully determine `this` binding.

Closely related to the `this` keyword is the object prototype mechanism, which is a look-up chain for properties, similar to how lexical scope variables are found. But wrapped up in the prototypes is the other huge miscue about JS: the idea of emulating (fake) classes and (so-called "prototypal") inheritance.

Unfortunately, the desire to bring class and inheritance design pattern thinking to JavaScript is just about the worst thing you could try to do, because while the syntax may trick you into thinking there's something like classes present, in fact the prototype mechanism is fundamentally opposite in its behavior.

What's at issue is whether it's better to ignore the mismatch and pretend that what you're implementing is "inheritance," or whether it's more appropriate to learn and embrace how the object prototype system actually works. The latter is more appropriately named "behavior delegation."

This is more than syntactic preference. Delegation is an entirely different, and more powerful, design pattern, one that replaces the need to design with classes and inheritance. But these assertions will absolutely fly in the face of nearly every other blog post, book, and conference talk on the subject for the entirety of JavaScript's lifetime.

The claims I make regarding delegation versus inheritance come not from a dislike of the language and its syntax, but from the desire to see the true capability of the language properly leveraged and the endless confusion and frustration wiped away.

But the case I make regarding prototypes and delegation is a much more involved one than what I will indulge here. If you're ready to reconsider everything you think you know about JavaScript "classes" and "inheritance," I offer you the chance to "take the red pill" (*The Matrix*, 1999) and check out Chapters 4-6 of the *this & Object Prototypes* title of this series.

Types & Grammar

The third title in this series primarily focuses on tackling yet another highly controversial topic: type coercion. Perhaps no topic causes more frustration with JS developers than when you talk about the confusions surrounding implicit coercion.

By far, the conventional wisdom is that implicit coercion is a "bad part" of the language and should be avoided at all costs. In fact, some have gone so far as to call it a "flaw" in the design of the language. Indeed, there are tools whose entire job is to do nothing but scan your code and complain if you're doing anything even remotely like coercion.

But is coercion really so confusing, so bad, so treacherous, that your code is doomed from the start if you use it?

I say no. After having built up an understanding of how types and values really work in Chapters 1-3, Chapter 4 takes on this debate and fully explains how coercion works, in all its nooks and crevices.

We see just what parts of coercion really are surprising and what parts actually make complete sense if given the time to learn.

But I'm not merely suggesting that coercion is sensible and learnable; I'm asserting that coercion is an incredibly useful and totally underestimated tool that *you should be using in your code*. I'm saying that coercion, when used properly, not only works, but makes your code better. All the naysayers and doubters will surely scoff at such a position, but I believe it's one of the main keys to upping your JS game.

Do you want to just keep following what the crowd says, or are you willing to set all the assumptions aside and look at coercion with a fresh perspective? The *Types & Grammar* title of this series will coerce your thinking.

Async & Performance

The first three titles of this series focus on the core mechanics of the language, but the fourth title branches out slightly to cover patterns on top of the language mechanics for managing asynchronous programming. Asynchrony is not only critical to the performance of our applications, it's increasingly becoming *the* critical factor in writability and maintainability.

The book starts first by clearing up a lot of terminology and concept confusion around things like "async," "parallel," and "concurrent," and explains in depth how such things do and do not apply to JS.

Then we move into examining callbacks as the primary method of enabling asynchrony. But it's here that we quickly see that the callback alone is hopelessly insufficient for the modern demands of asynchronous programming. We identify two major deficiencies of callbacks-only coding: *Inversion of Control* (IoC) trust loss and lack of linear reason-ability.

To address these two major deficiencies, ES6 introduces two new mechanisms (and indeed, patterns): *promises* and *generators*.

Promises are a time-independent wrapper around a "future value," which lets you reason about and compose them regardless of if the value is ready or not yet. Moreover, they effectively solve the IoC trust issues by routing callbacks through a trustable and composable promise mechanism.

Generators introduce a new mode of execution for JS functions, whereby the generator can be paused at `yield` points and be resumed asynchronously later. The pause-and-resume capability enables synchronous, sequential-looking code in the generator to be processed asynchronously behind the scenes. By doing so, we address the non-linear, non-local-jump confusions of callbacks and thereby make our asynchronous code sync-looking so as to be more reason-able.

But it's the combination of promises and generators that "yields" our most effective asynchronous coding pattern to date in JavaScript. In fact, much of the future sophistication of asynchrony coming in ES7 and later will certainly be built on this foundation. To be serious about programming effectively in an async world, you're going to need to get really comfortable with combining promises and generators.

If promises and generators are about expressing patterns that let our programs run more concurrently and thus get more processing accomplished in a shorter period, JS has many other facets of performance optimization worth exploring.

Chapter 5 delves into topics like program parallelism with Web Workers and data parallelism with SIMD, as well as low-level optimization techniques like ASM.js. Chapter 6 takes a look at performance optimization from the perspective of proper benchmarking techniques, including what kinds of performance to worry about and what to ignore.

Writing JavaScript effectively means writing code that can break the constraint barriers of being run dynamically in a wide range of browsers and other environments. It requires a lot of intricate and detailed planning and effort on our parts to take a program from "it works" to "it works well."

The *Async & Performance* title is designed to give you all the tools and skills you need to write reasonable and performant JavaScript code.

ES6 & Beyond

No matter how much you feel you've mastered JavaScript to this point, the truth is that JavaScript is never going to stop evolving, and moreover, the rate of evolution is increasing rapidly. This fact is

almost a metaphor for the spirit of this series, to embrace that we'll never fully *know* every part of JS, because as soon as you master it all, there's going to be new stuff coming down the line that you'll need to learn.

This title is dedicated to both the short- and mid-term visions of where the language is headed, not just the *known* stuff like ES6 but the *likely* stuff beyond.

While all the titles of this series embrace the state of JavaScript at the time of this writing, which is midway through ES6 adoption, the primary focus in the series has been more on ES5. Now, we want to turn our attention to ES6, ES7, and beyond…

Since ES6 is nearly complete at the time of this writing, *ES6 & Beyond* starts by dividing up the concrete stuff from the ES6 landscape into several key categories, including new syntax, new data structures (collections), and new processing capabilities and APIs. We cover each of these new ES6 features, in varying levels of detail, including reviewing details that are touched on in other books of this series.

Some exciting ES6 things to look forward to reading about: destructuring, default parameter values, symbols, concise methods, computed properties, arrow functions, block scoping, promises, generators, iterators, modules, proxies, weakmaps, and much, much more! Phew, ES6 packs quite a punch!

The first part of the book is a roadmap for all the stuff you need to learn to get ready for the new and improved JavaScript you'll be writing and exploring over the next couple of years.

The latter part of the book turns attention to briefly glance at things that we can likely expect to see in the near future of JavaScript. The most important realization here is that post-ES6, JS is likely going to evolve feature by feature rather than version by version, which means we can expect to see these near-future things coming much sooner than you might imagine.

The future for JavaScript is bright. Isn't it time we start learning it?

Review

The *YDKJS* series is dedicated to the proposition that all JS developers can and should learn all of the parts of this great language. No person's opinion, no framework's assumptions, and no project's deadline should be the excuse for why you never learn and deeply understand JavaScript.

We take each important area of focus in the language and dedicate a short but very dense book to fully explore all the parts of it that you perhaps thought you knew but probably didn't fully.

"You Don't Know JS" isn't a criticism or an insult. It's a realization that all of us, myself included, must come to terms with. Learning JavaScript isn't an end goal but a process. We don't know JavaScript, yet. But we will!

About the Author

Kyle Simpson is an Open Web Evangelist from Austin, TX, who's passionate about all things JavaScript. He's an author, workshop trainer, tech speaker, and OSS contributor/leader.

Scope and Closures

Kyle Simpson

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

Table of Contents

Foreword

When I was a young child, I would often enjoy taking things apart and putting them back together again—old mobile phones, hi-fi stereos, and anything else I could get my hands on. I was too young to really use these devices, but whenever one broke, I would instantly ask if I could figure out how it worked.

I remember once looking at a circuit board for an old radio. It had this weird long tube with copper wire wrapped around it. I couldn't work out its purpose, but I immediately went into research mode. What does it do? Why is it in a radio? It doesn't look like the other parts of the circuit board, why? Why does it have copper wrapped around it? What happens if I remove the copper?! Now I know it was a loop antenna, made by wrapping copper wire around a ferrite rod, which are often used in transistor radios.

Did you ever become addicted to figuring out all of the answers to every *why* question? Most children do. In fact it is probably my favorite thing about children—their desire to learn.

Unfortunately, now I'm considered a *professional* and spend my days making things. When I was young, I loved the idea of one day making the things that I took apart. Of course, most things I make now are with JavaScript and not ferrite rods…but close enough! However, despite once loving the idea of making things, I now find myself longing for the desire to figure things out. Sure, I often figure out the best way to solve a problem or fix a bug, but I rarely take the time to question my tools.

And that is exactly why I am so excited about this "You Don't Know JS" series of books. Because it's right. I don't know JS. I use JavaScript

day in, day out and have done for many years, but do I really understand it? No. Sure, I understand a lot of it and I often read the specs and the mailing lists, but no, I don't understand as much as my inner six-year-old wishes I did.

Scope and Closures is a brilliant start to the series. It is very well targeted at people like me (and hopefully you, too). It doesn't teach JavaScript as if you've never used it, but it does make you realize how little about the inner workings you probably know. It is also coming out at the perfect time: ES6 is finally settling down and implementation across browsers is going well. If you've not yet made time for learning the new features (such as `let` and `const`), this book will be a great introduction.

So I hope that you enjoy this book, but moreso, that Kyle's way of critically thinking about how every tiny bit of the language works will creep into your mindset and general workflow. Instead of just using the antenna, figure out how and why it works.

—Shane Hudson
www.shanehudson.net

Preface

I'm sure you noticed, but "JS" in the book series title is not an abbreviation for words used to curse about JavaScript, though cursing at the language's quirks is something we can probably all identify with!

From the earliest days of the Web, JavaScript has been a foundational technology that drives interactive experience around the content we consume. While flickering mouse trails and annoying pop-up prompts may be where JavaScript started, nearly two decades later, the technology and capability of JavaScript has grown many orders of magnitude, and few doubt its importance at the heart of the world's most widely available software platform: the Web.

But as a language, it has perpetually been a target for a great deal of criticism, owing partly to its heritage but even more to its design philosophy. Even the name evokes, as Brendan Eich once put it, "dumb kid brother" status next to its more mature older brother, Java. But the name is merely an accident of politics and marketing. The two languages are vastly different in many important ways. "JavaScript" is as related to "Java" as "Carnival" is to "Car."

Because JavaScript borrows concepts and syntax idioms from several languages, including proud C-style procedural roots as well as subtle, less obvious Scheme/Lisp-style functional roots, it is exceedingly approachable to a broad audience of developers, even those with just little to no programming experience. The "Hello World" of JavaScript is so simple that the language is inviting and easy to get comfortable with in early exposure.

While JavaScript is perhaps one of the easiest languages to get up and running with, its eccentricities make solid mastery of the language a

vastly less common occurrence than in many other languages. Where it takes a pretty in-depth knowledge of a language like C or C++ to write a full-scale program, full-scale production JavaScript can, and often does, barely scratch the surface of what the language can do.

Sophisticated concepts that are deeply rooted into the language tend instead to surface themselves in *seemingly* simplistic ways, such as passing around functions as callbacks, which encourages the JavaScript developer to just use the language as-is and not worry too much about what's going on under the hood.

It is simultaneously a simple, easy-to-use language that has broad appeal and a complex and nuanced collection of language mechanics that without careful study will elude *true understanding* even for the most seasoned of JavaScript developers.

Therein lies the paradox of JavaScript, the Achilles' heel of the language, the challenge we are presently addressing. Because JavaScript *can* be used without understanding, the understanding of the language is often never attained.

Mission

If at every point that you encounter a surprise or frustration in JavaScript, your response is to add it to the blacklist, as some are accustomed to doing, you soon will be relegated to a hollow shell of the richness of JavaScript.

While this subset has been famoulsy dubbed "The Good Parts," I would implore you, dear reader, to instead consider it the "The Easy Parts," "The Safe Parts," or even "The Incomplete Parts."

This "You Don't Know JavaScript" book series offers a contrary challenge: learn and deeply understand *all* of JavaScript, even and especially "The Tough Parts."

Here, we address head on the tendency of JS developers to learn "just enough" to get by, without ever forcing themselves to learn exactly how and why the language behaves the way it does. Furthermore, we eschew the common advice to *retreat* when the road gets rough.

I am not content, nor should you be, at stopping once something *just works*, and not really knowing *why*. I gently challenge you to journey down that bumpy "road less traveled" and embrace all that JavaScript is and can do. With that knowledge, no technique, no framework, no

popular buzzword acronym of the week, will be beyond your understanding.

These books each take on specific core parts of the language that are most commonly misunderstood or under-understood, and dive very deep and exhaustively into them. You should come away from reading with a firm confidence in your understanding, not just of the theoretical, but the practical "what you need to know" bits.

The JavaScript you know *right now* is probably *parts* handed down to you by others who've been burned by incomplete understanding. *That* JavaScript is but a shadow of the true language. You don't *really* know JavaScript, *yet*, but if you dig into this series, you *will*. Read on, my friends. JavaScript awaits you.

Review

JavaScript is awesome. It's easy to learn partially, but much harder to learn completely (or even *sufficiently*). When developers encounter confusion, they usually blame the language instead of their lack of understanding. These books aim to fix that, inspiring a strong appreciation for the language you can now, and *should*, deeply *know*.

Many of the examples in this book assume modern (and future-reaching) JavaScript engine environments, such as ECMAScript version 6 (ES6). Some code may not work as described if run in older (pre-ES6) environments.

Conventions Used in This Book

The following typographical conventions are used in this book:

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

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

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

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

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *http://bit.ly/1c8HEWF*.

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

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Scope and Closures* by Kyle Simpson (O'Reilly). Copyright 2014 Kyle Simpson, 978-1-449-33558-8."

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

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

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

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

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

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Check out the full *You Don't Know JS* series: *http://YouDontKnowJS.com*

CHAPTER 1
What Is Scope?

One of the most fundamental paradigms of nearly all programming languages is the ability to store values in variables, and later retrieve or modify those values. In fact, the ability to store values and pull values out of variables is what gives a program *state*.

Without such a concept, a program could perform some tasks, but they would be extremely limited and not terribly interesting.

But the inclusion of variables into our program begets the most interesting questions we will now address: where do those variables *live*? In other words, where are they stored? And, most important, how does our program find them when it needs them?

These questions speak to the need for a well-defined set of rules for storing variables in some location, and for finding those variables at a later time. We'll call that set of rules: *scope*.

But, where and how do these *scope* rules get set?

Compiler Theory

It may be self-evident, or it may be surprising, depending on your level of interaction with various languages, but despite the fact that JavaScript falls under the general category of "dynamic" or "interpreted" languages, it is in fact a compiled language. It is *not* compiled well in advance, as are many traditionally compiled languages, nor are the results of compilation portable among various distributed systems.

But, nevertheless, the JavaScript engine performs many of the same steps, albeit in more sophisticated ways than we may commonly be aware, of any traditional language compiler.

In traditional compiled-language process, a chunk of source code, your program, will undergo typically three steps *before* it is executed, roughly called "compilation":

Tokenizing/Lexing
: Breaking up a string of characters into meaningful (to the language) chunks, called tokens. For instance, consider the program `var a = 2;`. This program would likely be broken up into the following tokens: `var`, `a`, `=`, `2`, and `;`. Whitespace may or may not be persisted as a token, depending on whether its meaningful or not.

The difference between tokenizing and lexing is subtle and academic, but it centers on whether or not these tokens are identified in a *stateless* or *stateful* way. Put simply, if the tokenizer were to invoke stateful parsing rules to figure out whether `a` should be considered a distinct token or just part of another token, *that* would be *lexing*.

Parsing
: taking a stream (array) of tokens and turning it into a tree of nested elements, which collectively represent the grammatical structure of the program. This tree is called an "AST" (*a*bstract *s*yntax *t*ree).

  The tree for `var a = 2;` might start with a top-level node called `VariableDeclaration`, with a child node called `Identifier` (whose value is `a`), and another child called `AssignmentExpression`, which itself has a child called `NumericLiteral` (whose value is `2`).

Code-Generation
: The process of taking an AST and turning it into executable code. This part varies greatly depending on the language, the platform it's targeting, and so on.

  So, rather than get mired in details, we'll just handwave and say that there's a way to take our previously described AST for `var a = 2;` and turn it into a set of machine instructions to actually *create*

a variable called `a` (including reserving memory, etc.), and then store a value into `a`.

The details of how the engine manages system resources are deeper than we will dig, so we'll just take it for granted that the engine is able to create and store variables as needed.

The JavaScript engine is vastly more complex than *just* those three steps, as are most other language compilers. For instance, in the process of parsing and code-generation, there are certainly steps to optimize the performance of the execution, including collapsing redundant elements, etc.

So, I'm painting only with broad strokes here. But I think you'll see shortly why these details we *do* cover, even at a high level, are relevant.

For one thing, JavaScript engines don't get the luxury (like other language compilers) of having plenty of time to optimize, because JavaScript compilation doesn't happen in a build step ahead of time, as with other languages.

For JavaScript, the compilation that occurs happens, in many cases, mere microseconds (or less!) before the code is executed. To ensure the fastest performance, JS engines use all kinds of tricks (like JITs, which lazy compile and even hot recompile, etc.) that are well beyond the "scope" of our discussion here.

Let's just say, for simplicity sake, that any snippet of JavaScript has to be compiled before (usually *right* before!) it's executed. So, the JS compiler will take the program `var a = 2;` and compile it *first*, and then be ready to execute it, usually right away.

Understanding Scope

The way we will approach learning about scope is to think of the process in terms of a conversation. But, *who* is having the conversation?

The Cast

Let's meet the cast of characters that interact to process the program `var a = 2;`, so we understand their conversations that we'll listen in on shortly:

Engine
: Responsible for start-to-finish compilation and execution of our JavaScript program.

Compiler
: One of Engine's friends; handles all the dirty work of parsing and code-generation (see previous section).

Scope
: Another friend of Engine; collects and maintains a look-up list of all the declared identifiers (variables), and enforces a strict set of rules as to how these are accessible to currently executing code.

For you to *fully understand* how JavaScript works, you need to begin to *think* like Engine (and friends) think, ask the questions they ask, and answer those questions the same.

Back and Forth

When you see the program `var a = 2;`, you most likely think of that as one statement. But that's not how our new friend Engine sees it. In fact, Engine sees two distinct statements, one that Compiler will handle during compilation, and one that Engine will handle during execution.

So, let's break down how Engine and friends will approach the program `var a = 2;`.

The first thing Compiler will do with this program is perform lexing to break it down into tokens, which it will then parse into a tree. But when Compiler gets to code generation, it will treat this program somewhat differently than perhaps assumed.

A reasonable assumption would be that Compiler will produce code that could be summed up by this pseudocode: "Allocate memory for a variable, label it `a`, then stick the value `2` into that variable." Unfortunately, that's not quite accurate.

Compiler will instead proceed as:

1. Encountering `var a`, Compiler asks Scope to see if a variable `a` already exists for that particular scope collection. If so, Compiler ignores this declaration and moves on. Otherwise, Compiler asks Scope to declare a new variable called a for that scope collection.
2. Compiler then produces code for Engine to later execute, to handle the `a = 2` assignment. The code Engine runs will first ask Scope

if there is a variable called `a` accessible in the current scope collection. If so, Engine uses that variable. If not, Engine looks *elsewhere* (see "Nested Scope" on page 8).

If Engine eventually finds a variable, it assigns the value `2` to it. If not, Engine will raise its hand and yell out an error!

To summarize: two distinct actions are taken for a variable assignment: First, Compiler declares a variable (if not previously declared) in the current Scope, and second, when executing, Engine looks up the variable in Scope and assigns to it, if found.

Compiler Speak

We need a little bit more compiler terminology to proceed further with understanding.

When Engine executes the code that Compiler produced for step 2, it has to look up the variable `a` to see if it has been declared, and this look-up is consulting Scope. But the type of look-up Engine performs affects the outcome of the look-up.

In our case, it is said that Engine would be performing an LHS look-up for the variable `a`. The other type of look-up is called RHS.

I bet you can guess what the "L" and "R" mean. These terms stand for lefthand side and righthand side.

Side…of what? *Of an assignment operation.*

In other words, an LHS look-up is done when a variable appears on the lefthand side of an assignment operation, and an RHS look-up is done when a variable appears on the righthand side of an assignment operation.

Actually, let's be a little more precise. An RHS look-up is indistinguishable, for our purposes, from simply a look-up of the value of some variable, whereas the LHS look-up is trying to find the variable container itself, so that it can assign. In this way, RHS doesn't *really* mean "righthand side of an assignment" per se, it just, more accurately, means "not lefthand side".

Being slightly glib for a moment, you could think RHS instead means "retrieve his/her source (value)," implying that RHS means "go get the value of…"

Let's dig into that deeper.

When I say:

```
console.log( a );
```

The reference to `a` is an RHS reference, because nothing is being assigned to `a` here. Instead, we're looking up to retrieve the value of `a`, so that the value can be passed to `console.log(..)`.

By contrast:

```
a = 2;
```

The reference to `a` here is an LHS reference, because we don't actually care what the current value is, we simply want to find the variable as a target for the `= 2` assignment operation.

LHS and RHS meaning "left/righthand side of an assigment" doesn't necessarily literally mean "left/right side of the `=` assignment operator." There are several other ways that assignments happen, and so it's better to conceptually think about it as: "Who's the target of the assignment (LHS)?" and "Who's the source of the assignment (RHS)?"

Consider this program, which has both LHS and RHS references:

```
function foo(a) {
    console.log( a ); // 2
}

foo( 2 );
```

The last line that invokes `foo(..)` as a function call requires an RHS reference to `foo`, meaning, "Go look up the value of `foo`, and give it to me." Moreover, `(..)` means the value of `foo` should be executed, so it'd better actually be a function!

There's a subtle but important assignment here.

You may have missed the implied `a = 2` in this code snippet. It happens when the value `2` is passed as an argument to the `foo(..)` function, in which case the `2` value is *assigned* to the parameter `a`. To (implicitly) assign to parameter `a`, an LHS look-up is performed.

There's also an RHS reference for the value of `a`, and that resulting value is passed to `console.log(..)`. `console.log(..)` needs a

reference to execute. It's an RHS look-up for the `console` object, then a property resolution occurs to see if it has a method called `log`.

Finally, we can conceptualize that there's an LHS/RHS exchange of passing the value `2` (by way of variable `a`'s RHS look-up) into `log(..)`. Inside of the native implementation of `log(..)`, we can assume it has parameters, the first of which (perhaps called `arg1`) has an LHS reference look-up, before assigning `2` to it.

You might be tempted to conceptualize the function declaration `function foo(a) {…` as a normal variable declaration and assignment, such as `var foo` and `foo = function(a){…`. In so doing, it would be tempting to think of this function declaration as involving an LHS look-up.

However, the subtle but important difference is that Compiler handles both the declaration and the value definition during code-generation, such that when Engine is executing code, there's no processing necessary to "assign" a function value to `foo`. Thus, it's not really appropriate to think of a function declaration as an LHS look-up assignment in the way we're discussing them here.

Engine/Scope Conversation

```
function foo(a) {
    console.log( a ); // 2
}

foo( 2 );
```

Let's imagine the above exchange (which processes this code snippet) as a conversation. The conversation would go a little something like this:

> Engine: Hey Scope, I have an RHS reference for `foo`. Ever heard of it?
>
> Scope: Why yes, I have. Compiler declared it just a second ago. It's a function. Here you go.
>
> Engine: Great, thanks! OK, I'm executing `foo`.
>
> Engine: Hey, Scope, I've got an LHS reference for `a`, ever heard of it?
>
> Scope: Why yes, I have. Compiler declared it as a formal parameter to `foo` just recently. Here you go.
>
> Engine: Helpful as always, Scope. Thanks again. Now, time to assign `2` to `a`.

Engine: Hey, Scope, sorry to bother you again. I need an RHS look-up for `console`. Ever heard of it?

Scope: No problem, Engine, this is what I do all day. Yes, I've got `console`. It's built-in. Here ya go.

Engine: Perfect. Looking up `log(..)`. OK, great, it's a function.

Engine: Yo, Scope. Can you help me out with an RHS reference to `a`. I think I remember it, but just want to double-check.

Scope: You're right, Engine. Same variable, hasn't changed. Here ya go.

Engine: Cool. Passing the value of `a`, which is `2`, into `log(..)`.

…

Quiz

Check your understanding so far. Make sure to play the part of Engine and have a "conversation" with Scope:

```
function foo(a) {
    var b = a;
    return a + b;
}

var c = foo( 2 );
```

1. Identify all the LHS look-ups (there are 3!).
2. Identify all the RHS look-ups (there are 4!).

See the chapter review for the quiz answers!

Nested Scope

We said that Scope is a set of rules for looking up variables by their identifier name. There's usually more than one scope to consider, however.

Just as a block or function is nested inside another block or function, scopes are nested inside other scopes. So, if a variable cannot be found in the immediate scope, Engine consults the next outercontaining

scope, continuing until is found or until the outermost (a.k.a., global) scope has been reached.

Consider the following:

```
function foo(a) {
    console.log( a + b );
}

var b = 2;

foo( 2 ); // 4
```

The RHS reference for `b` cannot be resolved inside the function `foo`, but it can be resolved in the scope surrounding it (in this case, the global).

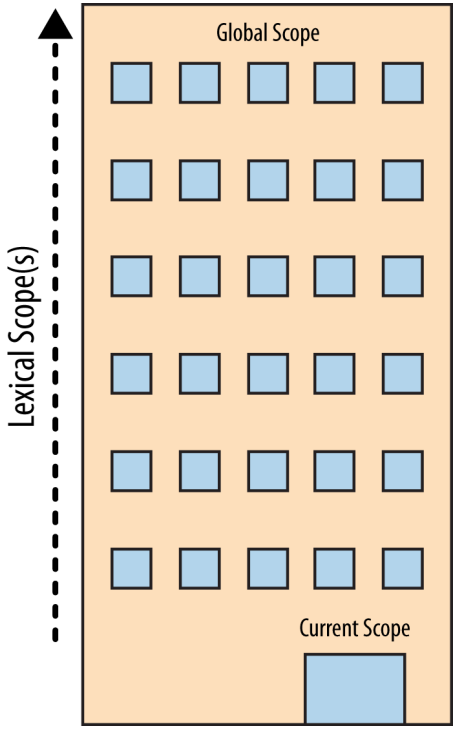
So, revisiting the conversations between Engine and Scope, we'd overhear:

> Engine: "Hey, Scope of `foo`, ever heard of `b`? Got an RHS reference for it."
>
> Scope: "Nope, never heard of it. Go fish."
>
> Engine: "Hey, Scope outside of `foo`, oh you're the global scope, OK cool. Ever heard of `b`? Got an RHS reference for it."
>
> Scope: "Yep, sure have. Here ya go."

The simple rules for traversing nested scope: Engine starts at the currently executing scope, looks for the variable there, then if not found, keeps going up one level, and so on. If the outermost global scope is reached, the search stops, whether it finds the variable or not.

Building on Metaphors

To visualize the process of nested scope resolution, I want you to think of this tall building:



The building represents our program's nested scope ruleset. The first floor of the building represents your currently executing scope, wherever you are. The top level of the building is the global scope.

You resolve LHS and RHS references by looking on your current floor, and if you don't find it, taking the elevator to the next floor, looking there, then the next, and so on. Once you get to the top floor (the global scope), you either find what you're looking for, or you don't. But you have to stop regardless.

Errors

Why does it matter whether we call it LHS or RHS?

Because these two types of look-ups behave differently in the circumstance where the variable has not yet been declared (is not found in any consulted scope).

Consider:

```
function foo(a) {
    console.log( a + b );
    b = a;
}

foo( 2 );
```

When the RHS look-up occurs for `b` the first time, it will not be found. This is said to be an "undeclared" variable, because it is not found in the scope.

If an RHS look-up fails to ever find a variable, anywhere in the nested `scope`s, this results in a `ReferenceError` being thrown by the engine. It's important to note that the error is of the type `ReferenceError`.

By contrast, if the engine is performing an LHS look-up, and it arrives at the top floor (global scope) without finding it, if the program is not running in "Strict Mode,"[1] then the global scope will create a new variable of that name *in the global scope*, and hand it back to Engine.

"No, there wasn't one before, but I was helpful and created one for you."

"Strict Mode," which was added in ES5, has a number of different behaviors from normal/relaxed/lazy mode. One such behavior is that it disallows the automatic/implicit global variable creation. In that case, there would be no global scoped variable to hand back from an LHS look-up, and Engine would throw a `ReferenceError` similarly to the RHS case.

Now, if a variable is found for an RHS look-up, but you try to do something with its value that is impossible, such as trying to execute-as-function a nonfunction value, or reference a property on a `null` or `undefined` value, then Engine throws a different kind of error, called a `TypeError`.

`ReferenceError` is scope resolution-failure related, whereas `TypeError` implies that scope resolution was successful, but that there was an illegal/impossible action attempted against the result.

Review

Scope is the set of rules that determines where and how a variable (identifier) can be looked up. This look-up may be for the purposes of

1. See the MDN's break down of Strict Mode

assigning to the variable, which is an LHS (lefthand-side) reference, or it may be for the purposes of retrieving its value, which is an RHS (righthand-side) reference.

LHS references result from assignment operations. Scope-related assignments can occur either with the `=` operator or by passing arguments to (assign to) function parameters.

The JavaScript engine first compiles code before it executes, and in so doing, it splits up statements like `var a = 2;` into two separate steps:

1. First, `var a` to declare it in that scope. This is performed at the beginning, before code execution.
2. Later, `a = 2` to look up the variable (LHS reference) and assign to it if found.

Both LHS and RHS reference look-ups start at the currently executing scope, and if need be (that is, they don't find what they're looking for there), they work their way up the nested scope, one scope (floor) at a time, looking for the identifier, until they get to the global (top floor) and stop, and either find it, or don't.

Unfulfilled RHS references result in `ReferenceError`s being thrown. Unfulfilled LHS references result in an automatic, implicitly created global of that name (if not in Strict Mode), or a `ReferenceError` (if in Strict Mode).

Quiz Answers

```
function foo(a) {
    var b = a;
    return a + b;
}

var c = foo( 2 );
```

1. Identify all the LHS look-ups (there are 3!).

   `c = ..;`, `a = 2 (implicit param assignment)` and `b = ..`

2. Identify all the RHS look-ups (there are 4!).

   `foo(2.., = a;`, `a .. and .. b`

CHAPTER 2
Lexical Scope

In Chapter 1, we defined "scope" as the set of rules that govern how the engine can look up a variable by its identifier name and find it, either in the current scope, or in any of the nested scopes it's contained within.

There are two predominant models for how scope works. The first of these is by far the most common, used by the vast majority of programming languages. It's called *lexical scope*, and we will examine it in depth. The other model, which is still used by some languages (such as Bash scripting, some modes in Perl, etc) is called *dynamic scope*.

Dynamic scope is covered in Appendix A. I mention it here only to provide a contrast with lexical scope, which is the scope model that JavaScript employs.

Lex-time

As we discussed in Chapter 1, the first traditional phase of a standard language compiler is called lexing (a.k.a., tokenizing). If you recall, the lexing process examines a string of source code characters and assigns semantic meaning to the tokens as a result of some stateful parsing.

It is this concept that provides the foundation to understand what lexical scope is and where the name comes from.

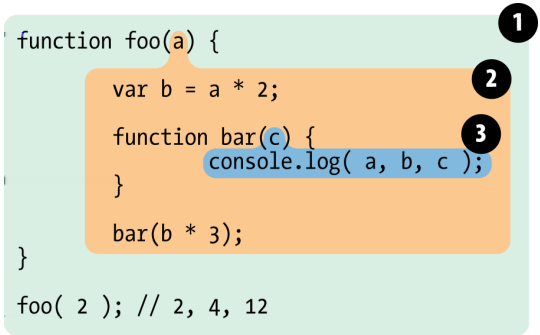
To define it somewhat circularly, lexical scope is scope that is defined at lexing time. In other words, lexical scope is based on where variables and blocks of scope are authored, by you, at write time, and thus is (mostly) set in stone by the time the lexer processes your code.

We will see in a little bit that there are some ways to cheat lexical scope, thereby modifying it after the lexer has passed by, but these are frowned upon. It is considered best practice to treat lexical scope as, in fact, lexical-only, and thus entirely author-time in nature.

Let's consider this block of code:

```
function foo(a) {

    var b = a * 2;

    function bar(c) {
        console.log( a, b, c );
    }

    bar( b * 3 );
}

foo( 2 ); // 2, 4, 12
```

There are three nested scopes inherent in this code example. It may be helpful to think about these scopes as bubbles inside of each other.



Bubble 1 encompasses the global scope and has just one identifier in it: `foo`.

Bubble 2 encompasses the scope of `foo`, which includes the three identifiers: `a`, `bar`, and `b`.

Bubble 3 encompasses the scope of `bar`, and it includes just one identifier: `c`.

Scope bubbles are defined by where the blocks of scope are written, which one is nested inside the other, etc. In the next chapter, we'll

discuss different units of scope, but for now, let's just assume that each function creates a new bubble of scope.

The bubble for `bar` is entirely contained within the bubble for `foo`, because (and only because) that's where we chose to define the function `bar`.

Notice that these nested bubbles are strictly nested. We're not talking about Venn diagrams where the bubbles can cross boundaries. In other words, no bubble for some function can simultaneously exist (partially) inside two other outer scope bubbles, just as no function can partially be inside each of two parent functions.

Look-ups

The structure and relative placement of these scope bubbles fully explains to the engine all the places it needs to look to find an identifier.

In the previous code snippet, the engine executes the `console.log(..)` statement and goes looking for the three referenced variables `a`, `b`, and `c`. It first starts with the innermost scope bubble, the scope of the `bar(..)` function. It won't find `a` there, so it goes up one level, out to the next nearest scope bubble, the scope of `foo(..)`. It finds `a` there, and so it uses that `a`. Same thing for `b`. But `c`, it does find inside of `bar(..)`.

Had there been a `c` both inside of `bar(..)` and inside of `foo(..)`, the `console.log(..)` statement would have found and used the one in `bar(..)`, never getting to the one in `foo(..)`.

Scope look-up stops once it finds the first match. The same identifier name can be specified at multiple layers of nested scope, which is called "shadowing" (the inner identifer "shadows" the outer identifier). Regardless of shadowing, scope look-up always starts at the innermost scope being executed at the time, and works its way outward/upward until the first match, and stops.

Global variables are automatically also properties of the global object (`window` in browsers, etc.), so it *is* possible to reference a global variable not directly by its lexical name, but instead indirectly as a property reference of the global object.

```
window.a
```

This technique gives access to a global variable that would otherwise be inaccessible due to it being shadowed. However, non-global shadowed variables cannot be accessed.

No matter *where* a function is invoked from, or even *how* it is invoked, its lexical scope is *only* defined by where the function was declared.

The lexical scope look-up process *only* applies to first-class identifiers, such as the `a`, `b`, and `c`. If you had a reference to `foo.bar.baz` in a piece of code, the lexical scope look-up would apply to finding the `foo` identifier, but once it locates that variable, object property-access rules take over to resolve the `bar` and `baz` properties, respectively.

Cheating Lexical

If lexical scope is defined only by where a function is declared, which is entirely an author-time decision, how could there possibly be a way to "modify" (a.k.a., cheat) lexical scope at runtime?

JavaScript has two such mechanisms. Both of them are equally frowned upon in the wider community as bad practices to use in your code. But the typical arguments against them are often missing the most important point: *cheating lexical scope leads to poorer performance.*

Before I explain the performance issue, though, let's look at how these two mechanisms work.

eval

The `eval(..)` function in JavaScript takes a string as an argument and treats the contents of the string as if it had actually been authored code at that point in the program. In other words, you can programatically generate code inside of your authored code, and run the generated code as if it had been there at author time.

Evaluating `eval(..)` (pun intended) in that light, it should be clear how `eval(..)` allows you to modify the lexical scope environment by

cheating and pretending that author-time (a.k.a., lexical) code was there all along.

On subsequent lines of code after an `eval(..)` has executed, the engine will not "know" or "care" that the previous code in question was dynamically interpreted and thus modified the lexical scope environment. The engine will simply perform its lexical scope lookups as it always does.

Consider the following code:

```
function foo(str, a) {
    eval( str ); // cheating!
    console.log( a, b );
}

var b = 2;

foo( "var b = 3;", 1 ); // 1, 3
```

The string `"var b = 3;"` is treated, at the point of the `eval(..)` call, as code that was there all along. Because that code happens to declare a new variable `b`, it modifies the existing lexical scope of `foo(..)`. In fact, as mentioned earlier, this code actually creates variable `b` inside of `foo(..)` that shadows the `b` that was declared in the outer (global) scope.

When the `console.log(..)` call occurs, it finds both `a` and `b` in the scope of `foo(..)`, and never finds the outer `b`. Thus, we print out "1, 3" instead of "1, 2" as would have normally been the case.

In this example, for simplicity sake, the string of "code" we pass in was a fixed literal. But it could easily have been programatically created by adding characters together based on your program's logic. `eval(..)` is usually used to execute dynamically created code, as dynamically evaluating essentially static code from a string literal would provide no real benefit to just authoring the code directly.

By default, if a string of code that `eval(..)` executes contains one or more declarations (either variables or functions), this action modifies the existing lexical scope in which the `eval(..)` resides. Technically, `eval(..)` can be invoked indirectly, through various tricks (beyond our discussion here), which causes it to instead execute in the context

of the global scope, thus modifying it. But in either case, `eval(..)` can at runtime modify an author-time lexical scope.

`eval(..)` when used in a strict-mode program operates in its own lexical scope, which means declarations made inside of the `eval()` do not actually modify the enclosing scope.

```
function foo(str) {
   "use strict";
   eval( str );
   console.log( a ); // ReferenceError: a is not defined
}

foo( "var a = 2" );
```

There are other facilities in JavaScript that amount to a very similar effect to `eval(..)`. `setTimeout(..)` and `setInterval(..)` *can* take a string for their respective first argument, the contents of which are `eval`uated as the code of a dynamically generated function. This is old, legacy behavior and long-since deprecated. Don't do it!

The `new Function(..)` function constructor similarly takes a string of code in its *last* argument to turn into a dynamically generated function (the first argument(s), if any, are the named parameters for the new function). This function-constructor syntax is slightly safer than `eval(..)`, but it should still be avoided in your code.

The use-cases for dynamically generating code inside your program are incredibly rare, as the performance degradations are almost never worth the capability.

with

The other frowned-upon (and now deprecated!) feature in JavaScript that cheats lexical scope is the `with` keyword. There are multiple valid ways that `with` can be explained, but I will choose here to explain it from the perspective of how it interacts with and affects lexical scope.

`with` is typically explained as a shorthand for making multiple property references against an object *without* repeating the object reference itself each time.

For example:

```
var obj = {
    a: 1,
    b: 2,
```

```
    c: 3
};

// more "tedious" to repeat "obj"
obj.a = 2;
obj.b = 3;
obj.c = 4;

// "easier" short-hand
with (obj) {
    a = 3;
    b = 4;
    c = 5;
}
```

However, there's much more going on here than just a convenient shorthand for object property access. Consider:

```
function foo(obj) {
    with (obj) {
        a = 2;
    }
}

var o1 = {
    a: 3
};

var o2 = {
    b: 3
};

foo( o1 );
console.log( o1.a ); // 2

foo( o2 );
console.log( o2.a ); // undefined
console.log( a ); // 2—Oops, leaked global!
```

In this code example, two objects `o1` and `o2` are created. One has an `a` property, and the other does not. The `foo(..)` function takes an object reference `obj` as an argument, and calls `with (obj) { .. }` on the reference. Inside the `with` block, we make what appears to be a normal lexical reference to a variable `a`, an LHS reference in fact (see Chapter 1), to assign to it the value of `2`.

When we pass in `o1`, the `a = 2` assignment finds the property `o1.a` and assigns it the value `2`, as reflected in the subsequent `console.log(o1.a)` statement. However, when we pass in `o2`, since it does

not have an `a` property, no such property is created, and `o2.a` remains `undefined`.

But then we note a peculiar side-effect, the fact that a global variable `a` was created by the `a = 2` assignment. How can this be?

The `with` statement takes an object, one that has zero or more properties, and treats that object as if it is a wholly separate lexical scope, and thus the object's properties are treated as lexically defined identifiers in that scope.

Even though a `with` block treats an object like a lexical scope, a normal var declaration inside that `with` block will not be scoped to that `with` block, but instead the containing function scope.

While the `eval(..)` function can modify existing lexical scope if it takes a string of code with one or more declarations in it, the `with` statement actually creates a *whole new lexical scope* out of thin air, from the object you pass to it.

Understood in this way, the scope declared by the `with` statement when we passed in `o1` was `o1`, and that scope had an identifier in it which corresponds to the `o1.a` property. But when we used `o2` as the scope, it had no such `a` identifier in it, and so the normal rules of LHS identifier look-up (see Chapter 1) occurred.

Neither the scope of `o2`, nor the scope of `foo(..)`, nor the global scope even, has an `a` identifier to be found, so when `a = 2` is executed, it results in the automatic global being created (since we're in non-strict mode).

It is a strange sort of mind-bending thought to see `with` turning, at runtime, an object and its properties into a scope *with* identifiers. But that is the clearest explanation I can give for the results we see.

In addition to being a bad idea to use, both `eval(..)` and `with` are affected (restricted) by Strict Mode. `with` is outright disallowed, whereas various forms of indirect or unsafe `eval(..)` are disallowed while retaining the core functionality.

Performance

Both `eval(..)` and `with` cheat the otherwise author-time defined lexical scope by modifying or creating new lexical scope at runtime.

So, what's the big deal, you ask? If they offer more sophisticated functionality and coding flexibility, aren't these *good* features? *No*.

The JavaScript engine has a number of performance optimizations that it performs during the compilation phase. Some of these boil down to being able to essentially statically analyze the code as it lexes, and predetermine where all the variable and function declarations are, so that it takes less effort to resolve identifiers during execution.

But if the engine finds an `eval(..)` or `with` in the code, it essentially has to *assume* that all its awareness of identifier location may be invalid, because it cannot know at lexing time exactly what code you may pass to `eval(..)` to modify the lexical scope, or the contents of the object you may pass to `with` to create a new lexical scope to be consulted.

In other words, in the pessimistic sense, most of those optimizations it *would* make are pointless if `eval(..)` or `with` are present, so it simply doesn't perform the optimizations *at all*.

Your code will almost certainly tend to run slower simply by the fact that you include an `eval(..)` or `with` anywhere in the code. No matter how smart the engine may be about trying to limit the side-effects of these pessmistic assumptions, **there's no getting around the fact that without the optimizations, code runs slower.**

Review

Lexical scope means that scope is defined by author-time decisions of where functions are declared. The lexing phase of compilation is essentially able to know where and how all identifiers are declared, and thus predict how they will be looked up during execution.

Two mechanisms in JavaScript can "cheat" lexical scope: `eval(..)` and `with`. The former can modify existing lexical scope (at runtime) by evaluating a string of "code" that has one or more declarations in it. The latter essentially creates a whole new lexical scope (again, at runtime) by treating an object reference *as* a scope and that object's properties as scoped identifiers.

The downside to these mechanisms is that it defeats the engine's ability to perform compile-time optimizations regarding scope look-up, because the engine has to assume pessimistically that such optimizations will be invalid. Code *will* run slower as a result of using either feature. *Don't use them.*

CHAPTER 3

Function Versus Block Scope

As we explored in Chapter 2, scope consists of a series of "bubbles" that each act as a container or bucket, in which identifiers (variables, functions) are declared. These bubbles nest neatly inside each other, and this nesting is defined at author time.

But what exactly makes a new bubble? Is it only the function? Can other structures in JavaScript create bubbles of scope?

Scope From Functions

The most common answer to those questions is that JavaScript has function-based scope. That is, each function you declare creates a bubble for itself, but no other structures create their own scope bubbles. As we'll see in just a little bit, this is not quite true.

But first, let's explore function scope and its implications.

Consider this code:

```
function foo(a) {
    var b = 2;

    // some code

    function bar() {
        // ...
    }

    // more code

    var c = 3;
}
```

In this snippet, the scope bubble for `foo(..)` includes identifiers `a`, `b`, `c`, and `bar`. It doesn't matter *where* in the scope a declaration appears, the variable or function belongs to the containing scope bubble, regardless. We'll explore how exactly *that* works in the next chapter.

`bar(..)` has its own scope bubble. So does the global scope, which has just one identifier attached to it: `foo`.

Because `a`, `b`, `c`, and `bar` all belong to the scope bubble of `foo(..)`, they are not accessible outside of `foo(..)`. That is, the following code would all result in `ReferenceError` errors, as the identifiers are not available to the global scope:

```
bar(); // fails

console.log( a, b, c ); // all 3 fail
```

However, all these identifiers (`a`, `b`, `c`, `foo`, and `bar`) are accessible *inside* of `foo(..)`, and indeed also available inside of `bar(..)` (assuming there are no shadow identifier declarations inside `bar(..)`).

Function scope encourages the idea that all variables belong to the function, and can be used and reused throughout the entirety of the function (and indeed, accessible even to nested scopes). This design approach can be quite useful, and certainly can make full use of the "dynamic" nature of JavaScript variables to take on values of different types as needed.

On the other hand, if you don't take careful precautions, variables existing across the entirety of a scope can lead to some unexpected pitfalls.

Hiding in Plain Scope

The traditional way of thinking about functions is that you declare a function and then add code inside it. But the inverse thinking is equally powerful and useful: take any arbitrary section of code you've written and wrap a function declaration around it, which in effect "hides" the code.

The practical result is to create a scope bubble around the code in question, which means that any declarations (variable or function) in that code will now be tied to the scope of the new wrapping function, rather than the previously enclosing scope. In other words, you can

"hide" variables and functions by enclosing them in the scope of a function.

Why would "hiding" variables and functions be a useful technique?

There's a variety of reasons motivating this scope-based hiding. They tend to arise from the software design principle Principle of Least Privilege[1], also sometimes called Least Authority or Least Exposure. This principle states that in the design of software, such as the API for a module/object, you should expose only what is minimally necessary, and "hide" everything else.

This principle extends to the choice of which scope to contain variables and functions. If all variables and functions were in the global scope, they would of course be accessible to any nested scope. But this would violate the "Least…" principle in that you are (likely) exposing many variables or functions that you should otherwise keep private, as proper use of the code would discourage access to those variables/functions.

For example:

```
function doSomething(a) {
    b = a + doSomethingElse( a * 2 );

    console.log( b * 3 );
}

function doSomethingElse(a) {
    return a - 1;
}

var b;

doSomething( 2 ); // 15
```

In this snippet, the `b` variable and the `doSomethingElse(..)` function are likely "private" details of how `doSomething(..)` does its job. Giving the enclosing scope "access" to `b` and `doSomethingElse(..)` is not only unnecessary but also possibly "dangerous," in that they may be used in unexpected ways, intentionally or not, and this may violate precondition assumptions of `doSomething(..)`. A more "proper" design would hide these private details inside the scope of `doSomething(..)`, such as:

1. Principle of Least Privilege

```
function doSomething(a) {
    function doSomethingElse(a) {
        return a - 1;
    }

    var b;

    b = a + doSomethingElse( a * 2 );

    console.log( b * 3 );
}

doSomething( 2 ); // 15
```

Now, `b` and `doSomethingElse(..)` are not accessible to any outside influence, instead controlled only by `doSomething(..)`. The functionality and end result has not been affected, but the design keeps private details private, which is usually considered better software.

Collision Avoidance

Another benefit of "hiding" variables and functions inside a scope is to avoid unintended collision between two different identifiers with the same name but different intended usages. Collision results often in unexpected overwriting of values.

For example:

```
function foo() {
    function bar(a) {
        i = 3; // changing the `i` in the enclosing scope's
               // for-loop
        console.log( a + i );
    }

    for (var i=0; i<10; i++) {
        bar( i * 2 ); // oops, inifinite loop ahead!
    }
}

foo();
```

The `i = 3` assignment inside of `bar(..)` overwrites, unexpectedly, the `i` that was declared in `foo(..)` at the `for` loop. In this case, it will result in an infinite loop, because `i` is set to a fixed value of `3` and that will forever remain `< 10`.

The assignment inside `bar(..)` needs to declare a local variable to use, regardless of what identifier name is chosen. `var i = 3;` would fix

the problem (and would create the previously mentioned "shadowed variable" declaration for `i`). An *additional*, not alternate, option is to pick another identifier name entirely, such as `var j = 3;`. But your software design may naturally call for the same identifier name, so utilizing scope to "hide" your inner declaration is your best/only option in that case.

Global namespaces

A particularly strong example of (likely) variable collision occurs in the global scope. Multiple libraries loaded into your program can quite easily collide with each other if they don't properly hide their internal/private functions and variables.

Such libraries typically will create a single variable declaration, often an object, with a sufficiently unique name, in the global scope. This object is then used as a *namespace* for that library, where all specific exposures of functionality are made as properties off that object (namespace), rather than as top-level lexically scoped identifiers themselves.

For example:

```
var MyReallyCoolLibrary = {
    awesome: "stuff",
    doSomething: function() {
        // ...
    },
    doAnotherThing: function() {
        // ...
    }
};
```

Module management

Another option for collision avoidance is the more modern *module* approach, using any of various dependency managers. Using these tools, no libraries ever add any identifiers to the global scope, but are instead required to have their identifier(s) be explicitly imported into another specific scope through usage of the dependency manager's various mechanisms.

It should be observed that these tools do not possess "magic" functionality that is exempt from lexical scoping rules. They simply use the rules of scoping as explained here to enforce that no identifiers are injected into any shared scope, and are instead kept in private,

non-collision-susceptible scopes, which prevents any accidental scope collisions.

As such, you can code defensively and achieve the same results as the dependency managers do without actually needing to use them, if you so choose. See the Chapter 5 for more information about the module pattern.

Functions as Scopes

We've seen that we can take any snippet of code and wrap a function around it, and that effectively "hides" any enclosed variable or function declarations from the outside scope inside that function's inner scope.

For example:

```
var a = 2;

function foo() { // <-- insert this

    var a = 3;
    console.log( a ); // 3

} // <-- and this
foo(); // <-- and this

console.log( a ); // 2
```

While this technique works, it is not necessarily very ideal. There are a few problems it introduces. The first is that we have to declare a named-function `foo()`, which means that the identifier name `foo` itself "pollutes" the enclosing scope (global, in this case). We also have to explicitly call the function by name (`foo()`) so that the wrapped code actually executes.

It would be more ideal if the function didn't need a name (or, rather, the name didn't pollute the enclosing scope), and if the function could automatically be executed.

Fortunately, JavaScript offers a solution to both problems.

```
var a = 2;

(function foo(){ // <-- insert this

    var a = 3;
    console.log( a ); // 3
```

```
})(); // <-- and this

console.log( a ); // 2
```

Let's break down what's happening here.

First, notice that the wrapping function statement starts with `(function`... as opposed to just `function`.... While this may seem like a minor detail, it's actually a major change. Instead of treating the function as a standard declaration, the function is treated as a function-expression.

The easiest way to distinguish declaration vs. expression is the position of the word `function` in the statement (not just a line, but a distinct statement). If `function` is the very first thing in the statement, then it's a function declaration. Otherwise, it's a function expression.

The key difference we can observe here between a function declaration and a function expression relates to where its name is bound as an identifier.

Compare the previous two snippets. In the first snippet, the name `foo` is bound in the enclosing scope, and we call it directly with `foo()`. In the second snippet, the name `foo` is not bound in the enclosing scope, but instead is bound only inside of its own function.

In other words, `(function foo(){ .. })` as an expression means the identifier `foo` is found *only* in the scope where the `..` indicates, not in the outer scope. Hiding the name `foo` inside itself means it does not pollute the enclosing scope unnecessarily.

Anonymous Versus Named

You are probably most familiar with function expressions as callback parameters, such as:

```
setTimeout( function(){
    console.log("I waited 1 second!");
}, 1000 );
```

This is called an *anonymous function expression*, because `function()` ... has no name identifier on it. Function expressions can be anonymous, but function declarations cannot omit the name—that would be illegal JS grammar.

Anonymous function expressions are quick and easy to type, and many libraries and tools tend to encourage this idiomatic style of code. However, they have several drawbacks to consider:

1. Anonymous functions have no useful name to display in stack traces, which can make debugging more difficult.
2. Without a name, if the function needs to refer to itself, for recursion, etc., the *deprecated* `arguments.callee` reference is unfortunately required. Another example of needing to self-reference is when an event handler function wants to unbind itself after it fires.
3. Anonymous functions omit a name, which is often helpful in providing more readable/understandable code. A descriptive name helps self-document the code in question.

Inline function expressions are powerful and useful—the question of anonymous versus named doesn't detract from that. Providing a name for your function expression quite effectively addresses all these drawbacks, but has no tangible downsides. The best practice is to always name your function expressions:

```
setTimeout( function timeoutHandler(){ // <-- Look, I have a
                                       // name!
    console.log( "I waited 1 second!" );
}, 1000 );
```

Invoking Function Expressions Immediately

```
var a = 2;

(function foo(){

    var a = 3;
    console.log( a ); // 3

})();

console.log( a ); // 2
```

Now that we have a function as an expression by virtue of wrapping it in a `( )` pair, we can execute that function by adding another `()` on the end, like `(function foo(){ .. })()`. The first enclosing `( )` pair makes the function an expression, and the second `()` executes the function.

This pattern is so common, a few years ago the community agreed on a term for it: *IIFE*, which stands for immediately invoked function expression.

Of course, IIFEs don't need names, necessarily—the most common form of IIFE is to use an anonymous function expression. While certainly less common, naming an IIFE has all the aforementioned benefits over anonymous function expressions, so it's a good practice to adopt.

```
var a = 2;

(function IIFE(){

    var a = 3;
    console.log( a ); // 3

})();

console.log( a ); // 2
```

There's a slight variation on the traditional IIFE form, which some prefer: `(function(){ .. }())`. Look closely to see the difference. In the first form, the function expression is wrapped in `( )`, and then the invoking `()` pair is on the outside right after it. In the second form, the invoking `()` pair is moved to the inside of the outer `( )` wrapping pair.

These two forms are identical in functionality. *It's purely a stylistic choice which you prefer.*

Another variation on IIFEs that is quite common is to use the fact that they are, in fact, just function calls, and pass in argument(s).

For instance:

```
var a = 2;

(function IIFE( global ){

    var a = 3;
    console.log( a ); // 3
    console.log( global.a ); // 2

})( window );

console.log( a ); // 2
```

We pass in the `window` object reference, but we name the parameter `global`, so that we have a clear stylistic delineation for global versus

nonglobal references. Of course, you can pass in anything from an enclosing scope you want, and you can name the parameter(s) anything that suits you. This is mostly just stylistic choice.

Another application of this pattern addresses the (minor niche) concern that the default `undefined` identifier might have its value incorrectly overwritten, causing unexpected results. By naming a parameter `undefined`, but not passing any value for that argument, we can guarantee that the `undefined` identifier is in fact the undefined value in a block of code:

```
undefined = true; // setting a land-mine for other code! avoid!

(function IIFE( undefined ){

    var a;
    if (a === undefined) {
        console.log( "Undefined is safe here!" );
    }

})();
```

Still another variation of the IIFE inverts the order of things, where the function to execute is given second, *after* the invocation and parameters to pass to it. This pattern is used in the UMD (Universal Module Definition) project. Some people find it a little cleaner to understand, though it is slightly more verbose.

```
var a = 2;

(function IIFE( def ){
    def( window );
})(function def( global ){

    var a = 3;
    console.log( a ); // 3
    console.log( global.a ); // 2

});
```

The `def` function expression is defined in the second-half of the snippet, and then passed as a parameter (also called `def`) to the `IIFE` function defined in the first half of the snippet. Finally, the parameter `def` (the function) is invoked, passing `window` in as the `global` parameter.

Blocks as Scopes

While functions are the most common unit of scope, and certainly the most widespread of the design approaches in the majority of JS in circulation, other units of scope are possible, and the usage of these other scope units can lead to even better, cleaner to maintain code.

Many languages other than JavaScript support block scope, and so developers from those languages are accustomed to the mindset, whereas those who've primarily only worked in JavaScript may find the concept slightly foreign.

But even if you've never written a single line of code in block-scoped fashion, you are still probably familiar with this extremely common idiom in JavaScript:

```
for (var i=0; i<10; i++) {
    console.log( i );
}
```

We declare the variable `i` directly inside the `for` loop head, most likely because our *intent* is to use `i` only within the context of that `for` loop, and essentially ignore the fact that the variable actually scopes itself to the enclosing scope (function or global).

That's what block-scoping is all about. Declaring variables as close as possible, as local as possible, to where they will be used. Another example:

```
var foo = true;

if (foo) {
    var bar = foo * 2;
    bar = something( bar );
    console.log( bar );
}
```

We are using a `bar` variable only in the context of the `if` statement, so it makes a kind of sense that we would declare it inside the `if` block. However, where we declare variables is not relevant when using `var`, because they will always belong to the enclosing scope. This snippet is essentially fake block-scoping, for stylistic reasons, and relying on self-enforcement not to accidentally use `bar` in another place in that scope.

Block scope is a tool to extend the earlier Principle of Least *Privilege* from hiding information in functions to hiding information in blocks of our code.

Consider the `for` loop example again:

```
for (var i=0; i<10; i++) {
    console.log( i );
}
```

Why pollute the entire scope of a function with the `i` variable that is only going to be (or only *should be*, at least) used for the `for` loop?

But more important, developers may prefer to *check* themselves against accidentally (re)using variables outside of their intended purpose, such being issued an error about an unknown variable if you try to use it in the wrong place. Block-scoping (if it were possible) for the `i` variable would make `i` available only for the `for` loop, causing an error if `i` is accessed elsewhere in the function. This helps ensure variables are not reused in confusing or hard-to-maintain ways.

But, the sad reality is that, on the surface, JavaScript has no facility for block scope.

That is, until you dig a little further.

with

We learned about `with` in Chapter 2. While it is a frowned-upon construct, it *is* an example of (a form of) block scope, in that the scope that is created from the object only exists for the lifetime of that `with` statement, and not in the enclosing scope.

try/catch

It's a *very* little known fact that JavaScript in ES3 specified the variable declaration in the `catch` clause of a `try/catch` to be block-scoped to the `catch` block.

For instance:

```
try {
    undefined(); // illegal operation to force an exception!
}
catch (err) {
    console.log( err ); // works!
}

console.log( err ); // ReferenceError: `err` not found
```

As you can see, `err` exists only in the `catch` clause, and throws an error when you try to reference it elsewhere.

While this behavior has been specified and true of practically all standard JS environments (except perhaps old IE), many linters seem to still complain if you have two or more `catch` clauses in the same scope that each declare their error variable with the same identifier name. This is not actually a redefinition, since the variables are safely block-scoped, but the linters still seem to, annoyingly, complain about this fact.

To avoid these unnecessary warnings, some devs will name their `catch` variables `err1`, `err2`, etc. Other devs will simply turn off the linting check for duplicate variable names.

The block-scoping nature of `catch` may seem like a useless academic fact, but see Appendix B for more information on just how useful it might be.

let

Thus far, we've seen that JavaScript only has some strange niche behaviors that expose block scope functionality. If that were all we had, and *it was* for many, many years, then block scoping would not be terribly useful to the JavaScript developer.

Fortunately, ES6 changes that, and introduces a new keyword `let`, which sits alongside `var` as another way to declare variables.

The `let` keyword attaches the variable declaration to the scope of whatever block (commonly a `{ .. }` pair) it's contained in. In other words, `let` implicitly hijacks any block's scope for its variable declaration.

```
var foo = true;

if (foo) {
    let bar = foo * 2;
    bar = something( bar );
    console.log( bar );
}

console.log( bar ); // ReferenceError
```

Using `let` to attach a variable to an existing block is somewhat implicit. It can confuse if you're not paying close attention to which blocks have variables scoped to them and are in the habit of moving blocks around, wrapping them in other blocks, etc., as you develop and evolve code.

Creating explicit blocks for block-scoping can address some of these concerns, making it more obvious where variables are attached and not. Usually, explicit code is preferable over implicit or subtle code. This explicit block-scoping style is easy to achieve and fits more naturally with how block-scoping works in other languages:

```
var foo = true;

if (foo) {
    { // <-- explicit block
        let bar = foo * 2;
        bar = something( bar );
        console.log( bar );
    }
}

console.log( bar ); // ReferenceError
```

We can create an arbitrary block for `let` to bind to by simply including a `{ .. }` pair anywhere a statement is valid grammar. In this case, we've made an explicit block *inside* the `if` statement, which may be easier as a whole block to move around later in refactoring, without affecting the position and semantics of the enclosing `if` statment.

For another way to express explicit block scopes, see Appendix B.

In Chapter 4, we will address hoisting, which talks about declarations being taken as existing for the entire scope in which they occur.

However, declarations made with `let` will not hoist to the entire scope of the block they appear in. Such declarations will not observably "exist" in the block until the declaration statement.

```
{
   console.log( bar ); // ReferenceError!
   let bar = 2;
}
```

Garbage collection

Another reason block-scoping is useful relates to closures and garbage collection to reclaim memory. We'll briefly illustrate here, but the closure mechanism is explained in detail in Chapter 5.

Consider:

```
function process(data) {
    // do something interesting
}

var someReallyBigData = { .. };

process( someReallyBigData );

var btn = document.getElementById( "my_button" );

btn.addEventListener( "click", function click(evt){
    console.log("button clicked");
}, /*capturingPhase=*/false );
```

The `click` function click handler callback doesn't *need* the `someReallyBigData` variable at all. That means, theoretically, after `process(..)` runs, the big memory-heavy data structure could be garbage collected. However, it's quite likely (though implementation dependent) that the JS engine will still have to keep the structure around, since the `click` function has a closure over the entire scope.

Block-scoping can address this concern, making it clearer to the engine that it does not need to keep `someReallyBigData` around:

```
function process(data) {
    // do something interesting
}

// anything declared inside this block can go away after!
{
    let someReallyBigData = { .. };

    process( someReallyBigData );
}

var btn = document.getElementById( "my_button" );

btn.addEventListener( "click", function click(evt){
    console.log("button clicked");
}, /*capturingPhase=*/false );
```

Declaring explicit blocks for variables to locally bind to is a powerful tool that you can add to your code toolbox.

let loops

A particular case where `let` shines is in the `for` loop case as we discussed previously.

```
for (let i=0; i<10; i++) {
    console.log( i );
}

console.log( i ); // ReferenceError
```

Not only does `let` in the `for` loop header bind the `i` to the `for` loop body, but in fact, it *rebinds it* to each *iteration* of the loop, making sure to reassign it the value from the end of the previous loop iteration.

Here's another way of illustrating the per-iteration binding behavior that occurs:

```
{
    let j;
    for (j=0; j<10; j++) {
        let i = j; // re-bound for each iteration!
        console.log( i );
    }
}
```

The reason why this per-iteration binding is interesting will become clear in Chapter 5 when we discuss closures.

Because `let` declarations attach to arbitrary blocks rather than to the enclosing function's scope (or global), there can be gotchas where existing code has a hidden reliance on function-scoped `var` declarations, and replacing the `var` with `let` may require additional care when refactoring code.

Consider:

```
var foo = true, baz = 10;

if (foo) {
    var bar = 3;

    if (baz > bar) {
        console.log( baz );
    }

    // ...
}
```

This code is fairly easily refactored as:

```
var foo = true, baz = 10;

if (foo) {
    var bar = 3;
```

```
    // ...
}

if (baz > bar) {
    console.log( baz );
}
```

But, be careful of such changes when using block-scoped variables:

```
var foo = true, baz = 10;

if (foo) {
    let bar = 3;

    if (baz > bar) { // <-- don't forget `bar` when moving!
        console.log( baz );
    }
}
```

See Appendix B for an alternate (more explicit) style of block-scoping that may provide easier to maintain/refactor code that's more robust to these scenarios.

const

In addition to `let`, ES6 introduces `const`, which also creates a block-scoped variable, but whose value is fixed (constant). Any attempt to change that value at a later time results in an error.

```
var foo = true;

if (foo) {
    var a = 2;
    const b = 3; // block-scoped to the containing `if`

    a = 3; // just fine!
    b = 4; // error!
}

console.log( a ); // 3
console.log( b ); // ReferenceError!
```

Review

Functions are the most common unit of scope in JavaScript. Variables and functions that are declared inside another function are essentially "hidden" from any of the enclosing scopes, which is an intentional design principle of good software.

But functions are by no means the only unit of scope. Block scope refers to the idea that variables and functions can belong to an arbitrary block (generally, any `{ .. }` pair) of code, rather than only to the enclosing function.

Starting with ES3, the `try/catch` structure has block scope in the `catch` clause.

In ES6, the `let` keyword (a cousin to the `var` keyword) is introduced to allow declarations of variables in any arbitrary block of code. `if (..) { let a = 2; }` will declare a variable `a` that essentially hijacks the scope of the `if`'s `{ .. }` block and attaches itself there.

Though some seem to believe so, block scope should not be taken as an outright replacement of `var` function scope. Both functionalities co-exist, and developers can and should use both function-scope and block-scope techniques where respectively appropriate to produce better, more readable/maintainable code.

CHAPTER 4
Hoisting

By now, you should be fairly comfortable with the idea of scope, and how variables are attached to different levels of scope depending on where and how they are declared. Both function scope and block scope behave by the same rules in this regard: any variable declared within a scope is attached to that scope.

But there's a subtle detail of how scope attachment works with declarations that appear in various locations within a scope, and that detail is what we will examine here.

Chicken or the Egg?

There's a temptation to think that all of the code you see in a JavaScript program is interpreted line-by-line, top-down in order, as the program executes. While that is substantially true, there's one part of that assumption that can lead to incorrect thinking about your program.

Consider this code:

```
a = 2;

var a;

console.log( a );
```

What do you expect to be printed in the `console.log(..)` statement?

Many developers would expect `undefined`, since the `var a` statement comes after the `a = 2`, and it would seem natural to assume that the

variable is redefined, and thus assigned the default `undefined`. However, the output will be `2`.

Consider another piece of code:

```
console.log( a );

var a = 2;
```

You might be tempted to assume that, since the previous snippet exhibited some less-than-top-down looking behavior, perhaps in this snippet, `2` will also be printed. Others may think that since the `a` variable is used before it is declared, this must result in a `ReferenceError` being thrown.

Unfortunately, both guesses are incorrect. `undefined` is the output.

So, what's going on here? It would appear we have a chicken-and-the-egg question. Which comes first, the declaration ("egg"), or the assignment ("chicken")?

The Compiler Strikes Again

To answer this question, we need to refer back to Chapter 1, and our discussion of compilers. Recall that the engine actually will compile your JavaScript code before it interprets it. Part of the compilation phase was to find and associate all declarations with their appropriate scopes. Chapter 2 showed us that this is the heart of lexical scope.

So, the best way to think about things is that all declarations, both variables and functions, are processed first, before any part of your code is executed.

When you see `var a = 2;`, you probably think of that as one statement. But JavaScript actually thinks of it as two statements: `var a;` and `a = 2;`. The first statement, the declaration, is processed during the compilation phase. The second statement, the assignment, is left *in place* for the execution phase.

Our first snippet then should be thought of as being handled like this:

```
var a;

a = 2;

console.log( a );
```

…where the first part is the compilation and the second part is the execution.

Similarly, our second snippet is actually processed as:

```
var a;

console.log( a );

a = 2;
```

So, one way of thinking, sort of metaphorically, about this process, is that variable and function declarations are "moved" from where they appear in the flow of the code to the top of the code. This gives rise to the name *hoisting*.

In other words, *the egg (declaration) comes before the chicken (assignment)*.

Only the declarations themselves are hoisted, while any assignments or other executable logic are left *in place*. If hoisting were to re-arrange the executable logic of our code, that could wreak havoc.

```
foo();

function foo() {
    console.log( a ); // undefined

    var a = 2;
}
```

The function `foo`'s declaration (which in this case *includes* the implied value of it as an actual function) is hoisted, such that the call on the first line is able to execute.

It's also important to note that hoisting is *per-scope*. So while our previous snippets were simplified in that they only included global scope, the `foo(..)` function we are now examining itself exhibits that `var a` is hoisted to the top of `foo(..)` (not, obviously, to the top of the program). So the program can perhaps be more accurately interpreted like this:

```
function foo() {
    var a;

    console.log( a ); // undefined

    a = 2;
```

```
}

foo();
```

Function declarations are hoisted, as we just saw. But function expressions are not.

```
foo(); // not ReferenceError, but TypeError!

var foo = function bar() {
    // ...
};
```

The variable identifier `foo` is hoisted and attached to the enclosing scope (global) of this program, so `foo()` doesn't fail as a `ReferenceError`. But `foo` has no value yet (as it would if it had been a true function declaration instead of expression). So, `foo()` is attempting to invoke the `undefined` value, which is a `TypeError` illegal operation.

Also recall that even though it's a named function expression, the name identifier is not available in the enclosing scope:

```
foo(); // TypeError
bar(); // ReferenceError

var foo = function bar() {
    // ...
};
```

This snippet is more accurately interpreted (with hoisting) as:

```
var foo;

foo(); // TypeError
bar(); // ReferenceError

foo = function() {
    var bar = ...self...
    // ...
}
```

Functions First

Both function declarations and variable declarations are hoisted. But a subtle detail (that *can* show up in code with multiple "duplicate" declarations) is that functions are hoisted first, and then variables.

Consider:

```
foo(); // 1

var foo;

function foo() {
    console.log( 1 );
}

foo = function() {
    console.log( 2 );
};
```

`1` is printed instead of `2`! This snippet is interpreted by the *Engine* as:

```
function foo() {
    console.log( 1 );
}

foo(); // 1

foo = function() {
    console.log( 2 );
};
```

Notice that `var foo` was the duplicate (and thus ignored) declaration, even though it came before the `function foo()`... declaration, because function declarations are hoisted before normal variables.

While multiple/duplicate `var` declarations are effectively ignored, subsequent function declarations *do* override previous ones.

```
foo(); // 3

function foo() {
    console.log( 1 );
}

var foo = function() {
    console.log( 2 );
};

function foo() {
    console.log( 3 );
}
```

While this all may sound like nothing more than interesting academic trivia, it highlights the fact that duplicate definitions in the same scope are a really bad idea and will often lead to confusing results.

Function declarations that appear inside of normal blocks typically hoist to the enclosing scope, rather than being conditional as this code implies:

```
foo(); // "b"

var a = true;
if (a) {
   function foo() { console.log("a"); }
}
else {
   function foo() { console.log("b"); }
}
```

However, it's important to note that this behavior is not reliable and is subject to change in future versions of JavaScript, so it's probably best to avoid declaring functions in blocks.

Review

We can be tempted to look at `var a = 2;` as one statement, but the JavaScript engine does not see it that way. It sees `var a` and `a = 2` as two separate statements, the first one a compiler-phase task, and the second one an execution-phase task.

What this leads to is that all declarations in a scope, regardless of where they appear, are processed *first* before the code itself is executed. You can visualize this as declarations (variables and functions) being "moved" to the top of their respective scopes, which we call *hoisting*.

Declarations themselves are hoisted, but assignments, even assignments of function expressions, are *not* hoisted.

Be careful about duplicate declarations, especially mixed between normal `var` declarations and function declarations—peril awaits if you do!

CHAPTER 5
Scope Closure

We arrive at this point with hopefully a very healthy, solid understanding of how scope works.

We turn our attention to an incredibly important, but persistently elusive, *almost mythological*, part of the language: *closure*. If you have followed our discussion of lexical scope thus far, the payoff is that closure is going to be, largely, anticlimactic, almost self-obvious. *There's a man behind the wizard's curtain, and we're about to see him*. No, his name is not Crockford!

If however you have nagging questions about lexical scope, now would be a good time to go back and review Chapter 2 before proceeding.

Enlightenment

For those who are somewhat experienced in JavaScript but have perhaps never fully grasped the concept of closures, understanding closure can seem like a special nirvana that one must strive and sacrifice to attain.

I recall years back when I had a firm grasp on JavaScript but had no idea what closure was. The hint that there was this other side to the language, one that promised even more capability than I already possessed, but it teased and taunted me. I remember reading through the source code of early frameworks trying to understand how it actually worked. I remember the first time something of the "module pattern" began to emerge in my mind. I remember the aha! moments quite vividly.

What I didn't know back then, what took me years to understand, and what I hope to impart to you presently, is this secret: *closure is all around you in JavaScript, you just have to recognize and embrace it.* Closures are not a special opt-in tool that you must learn new syntax and patterns for. No, closures are not even a weapon that you must learn to wield and master as Luke trained in the Force.

Closures happen as a result of writing code that relies on lexical scope. They just happen. You do not even really have to intentionally create closures to take advantage of them. Closures are created and used for you all over your code. What you are *missing* is the proper mental context to recognize, embrace, and leverage closures for your own will.

The enlightenment moment should be: oh, closures are already occurring all over my code, I can finally see them now. Understanding closures is like when Neo sees the Matrix for the first time.

Nitty Gritty

OK, enough hyperbole and shameless movie references.

Here's a down-and-dirty definition of what you need to know to understand and recognize closures:

> Closure is when a function is able to remember and access its lexical scope even when that function is executing outside its lexical scope.

Let's jump into some code to illustrate that definition.

```
function foo() {
    var a = 2;

    function bar() {
        console.log( a ); // 2
    }

    bar();
}

foo();
```

This code should look familiar from our discussions of nested scope. Function `bar()` has *access* to the variable `a` in the outer enclosing scope because of lexical scope look-up rules (in this case, it's an RHS reference look-up).

Is this closure?

Well, technically…*perhaps*. But by our what-you-need-to-know definition above…*not exactly*. I think the most accurate way to explain `bar()` referencing `a` is via lexical scope look-up rules, and those rules are *only* (an important!) *part* of what closure is.

From a purely academic perspective, what is said of the above snippet is that the function `bar()` has a *closure* over the scope of `foo()` (and indeed, even over the rest of the scopes it has access to, such as the global scope in our case). Put slightly differently, it's said that `bar()` closes over the scope of `foo()`. Why? Because `bar()` appears nested inside of `foo()`. Plain and simple.

But, closure defined in this way is not directly *observable*, nor do we see closure *exercised* in that snippet. We clearly see lexical scope, but closure remains sort of a mysterious shifting shadow behind the code.

Let us then consider code that brings closure into full light:

```
function foo() {
    var a = 2;

    function bar() {
        console.log( a );
    }

    return bar;
}

var baz = foo();

baz(); // 2 -- Whoa, closure was just observed, man.
```

The function `bar()` has lexical scope access to the inner scope of `foo()`. But then, we take `bar()`, the function itself, and pass it *as* a value. In this case, we `return` the function object itself that `bar` references.

After we execute `foo()`, we assign the value it returned (our inner `bar()` function) to a variable called `baz`, and then we actually invoke `baz()`, which of course is invoking our inner function `bar()`, just by a different identifier reference.

`bar()` is executed, for sure. But in this case, it's executed *outside* of its declared lexical scope.

After `foo()` executed, normally we would expect that the entirety of the inner scope of `foo()` would go away, because we know that the

engine employs a garbage collector that comes along and frees up memory once it's no longer in use. Since it would appear that the contents of `foo()` are no longer in use, it would seem natural that they should be considered *gone*.

But the "magic" of closures does not let this happen. That inner scope is in fact *still* in use, and thus does not go away. Who's using it? The function `bar()` itself.

By virtue of where it was declared, `bar()` has a lexical scope closure over that inner scope of `foo()`, which keeps that scope alive for `bar()` to reference at any later time.

`bar()` still has a reference to that scope, and that reference is called closure.

So, a few microseconds later, when the variable `baz` is invoked (invoking the inner function we initially labeled `bar`), it duly has *access* to author-time lexical scope, so it can access the variable `a` just as we'd expect.

The function is being invoked well outside of its author-time lexical scope. Closure lets the function continue to access the lexical scope it was defined in at author time.

Of course, any of the various ways that functions can be *passed around* as values, and indeed invoked in other locations, are all examples of observing/exercising closure.

```
function foo() {
    var a = 2;

    function baz() {
        console.log( a ); // 2
    }

    bar( baz );
}

function bar(fn) {
    fn(); // look ma, I saw closure!
}
```

We pass the inner function `baz` over to `bar`, and call that inner function (labeled `fn` now), and when we do, its closure over the inner scope of `foo()` is observed by accessing `a`.

These passings-around of functions can be indirect, too.

```
var fn;

function foo() {
    var a = 2;

    function baz() {
        console.log( a );
    }

    fn = baz; // assign baz to global variable
}

function bar() {
    fn(); // look ma, I saw closure!
}

foo();

bar(); // 2
```

Whatever facility we use to *transport* an inner function outside of its lexical scope, it will maintain a scope reference to where it was originally declared, and wherever we execute him, that closure will be exercised.

Now I Can See

The previous code snippets are somewhat academic and artifically constructed to illustrate *using closure*. But I promised you something more than just a cool new toy. I promised that closure was something all around you in your existing code. Let us now *see* that truth.

```
function wait(message) {

    setTimeout( function timer(){
        console.log( message );
    }, 1000 );

}

wait( "Hello, closure!" );
```

We take an inner function (named `timer`) and pass it to `setTimeout(..)`. But `timer` has a scope closure over the scope of `wait(..)`, indeed keeping and using a reference to the variable `message`.

A thousand milliseconds after we have executed `wait(..)`, and its inner scope should otherwise be long gone, that anonymous function still has closure over that scope.

Deep down in the guts of the engine, the built-in utility `setTimeout(..)` has reference to some parameter, probably called `fn` or `func` or something like that. Engine goes to invoke that function, which is invoking our inner `timer` function, and the lexical scope reference is still intact.

Closure.

Or, if you're of the jQuery persuasion (or any JS framework, for that matter):

```
function setupBot(name,selector) {
    $( selector ).click( function activator(){
        console.log( "Activating: " + name );
    } );
}

setupBot( "Closure Bot 1", "#bot_1" );
setupBot( "Closure Bot 2", "#bot_2" );
```

I am not sure what kind of code you write, but I regularly write code that is responsible for controlling an entire global drone army of closure bots, so this is totally realistic!

(Some) joking aside, essentially *whenever* and *wherever* you treat functions (that access their own respective lexical scopes) as first-class values and pass them around, you are likely to see those functions exercising closure. Be that timers, event handlers, Ajax requests, cross-window messaging, web workers, or any of the other asynchronous (or synchronous!) tasks, when you pass in a *callback function*, get ready to sling some closure around!

Chapter 3 introduced the IIFE pattern. While it is often said that IIFE (alone) is an example of observed closure, I would somewhat disagree, by our previous definition.

```
var a = 2;

(function IIFE(){
    console.log( a );
})();
```

This code works, but it's not strictly an observation of closure. Why? Because the function (which we named `IIFE` here) is not executed outside its lexical scope. It's still invoked right there in the same scope as it was declared (then enclosing/global scope that also holds `a`). `a` is found via normal lexical scope look-up, not really via closure.

While closure might technically be happening at declaration time, it is *not* strictly observable, and so, as they say, *it's a tree falling in the forest with no one around to hear it.*

Though an IIFE is not *itself* an example of observed closure, it absolutely creates scope, and it's one of the most common tools we use to create scope which can be closed over. So IIFEs are indeed heavily related to closure, even if not exercising closure themselves.

Put this book down right now, dear reader. I have a task for you. Go open up some of your recent JavaScript code. Look for your functions-as-values and identify where you are already using closure and maybe didn't even know it before.

I'll wait.

Now you see!

Loops and Closure

The most common canonical example used to illustrate closure involves the humble `for` loop.

```
for (var i=1; i<=5; i++) {
    setTimeout( function timer(){
        console.log( i );
    }, i*1000 );
}
```

Linters often complain when you put functions inside of loops, because the mistakes of not understanding closure are *so common among developers*. We explain how to do so properly here, leveraging the full power of closure. But that subtlety is often lost on linters, and they will complain regardless, assuming you don't *actually* know what you're doing.

The spirit of this code snippet is that we would normally *expect* for the behavior to be that the numbers 1, 2,…5 would be printed out, one at a time, one per second, respectively.

In fact, if you run this code, you get 6 printed out five times, at the one-second intervals.

Huh?

First, let's explain where `6` comes from. The terminating condition of the loop is when `i` is *not* `<=5`. The first time that's the case is when `i` is 6. So, the output is reflecting the final value of the `i` after the loop terminates.

This actually seems obvious on second glance. The timeout function callbacks are all running well after the completion of the loop. In fact, as timers go, even if it was `setTimeout(.., 0)` on each iteration, all those function callbacks would still run strictly after the completion of the loop, and thus print `6` each time.

But there's a deeper question at play here. What's *missing* from our code to actually have it behave as we semantically have implied?

What's missing is that we are trying to *imply* that each iteration of the loop "captures" its own copy of `i`, at the time of the iteration. But, the way scope works, all five of those functions, though they are defined separately in each loop iteration, *are closed over the same shared global scope*, which has, in fact, only one `i` in it.

Put that way, *of course* all functions share a reference to the same `i`. Something about the loop structure tends to confuse us into thinking there's something else more sophisticated at work. There is not. There's no difference than if each of the five timeout callbacks were just declared one right after the other, with no loop at all.

OK, so, back to our burning question. What's missing? We need more closured scope. Specifically, we need a new closured scope for each iteration of the loop.

We learned in Chapter 3 that the IIFE creates scope by declaring a function and immediately executing it.

Let's try:

```
for (var i=1; i<=5; i++) {
    (function(){
        setTimeout( function timer(){
            console.log( i );
        }, i*1000 );
    })();
}
```

Does that work? Try it. Again, I'll wait.

I'll end the suspense for you. *Nope.* But why? We now obviously have more lexical scope. Each timeout function callback is indeed closing over its own per-iteration scope created respectively by each IIFE.

It's not enough to have a scope to close over *if that scope is empty*. Look closely. Our IIFE is just an empty do-nothing scope. It needs *something* in it to be useful to us.

It needs its own variable, with a copy of the `i` value at each iteration.

```
for (var i=1; i<=5; i++) {
    (function(){
        var j = i;
        setTimeout( function timer(){
            console.log( j );
        }, j*1000 );
    })();
}
```

Eureka! It works!

A slight variation some prefer is:

```
for (var i=1; i<=5; i++) {
    (function(j){
        setTimeout( function timer(){
            console.log( j );
        }, j*1000 );
    })( i );
}
```

Of course, since these IIFEs are just functions, we can pass in `i`, and we can call it `j` if we prefer, or we can even call it `i` again. Either way, the code works now.

The use of an IIFE inside each iteration created a new scope for each iteration, which gave our timeout function callbacks the opportunity to close over a new scope for each iteration, one which had a variable with the right per-iteration value in it for us to access.

Problem solved!

Block Scoping Revisited

Look carefully at our analysis of the previous solution. We used an IIFE to create new scope per-iteration. In other words, we actually *needed* a per-iteration *block scope*. Chapter 3 showed us the `let`

declaration, which hijacks a block and declares a variable right there in the block.

It essentially turns a block into a scope that we can close over. So, the following awesome code just works:

```
for (var i=1; i<=5; i++) {
    let j = i; // yay, block-scope for closure!
    setTimeout( function timer(){
        console.log( j );
    }, j*1000 );
}
```

But, that's not all! (in my best Bob Barker voice). There's a special behavior defined for `let` declarations used in the head of a `for` loop. This behavior says that the variable will be declared not just once for the loop, **but each iteration**. And, it will, helpfully, be initialized at each subsequent iteration with the value from the end of the previous iteration.

```
for (let i=1; i<=5; i++) {
    setTimeout( function timer(){
        console.log( i );
    }, i*1000 );
}
```

How cool is that? Block scoping and closure working hand-in-hand, solving all the world's problems. I don't know about you, but that makes me a happy JavaScripter.

Modules

There are other code patterns that leverage the power of closure but that do not on the surface appear to be about callbacks. Let's examine the most powerful of them: *the module*.

```
function foo() {
    var something = "cool";
    var another = [1, 2, 3];

    function doSomething() {
        console.log( something );
    }

    function doAnother() {
        console.log( another.join( " ! " ) );
    }
}
```

As this code stands right now, there's no observable closure going on. We simply have some private data variables `something` and `another`, and a couple of inner functions `doSomething()` and `doAnother()`, which both have lexical scope (and thus closure!) over the inner scope of `foo()`.

But now consider:

```
function CoolModule() {
    var something = "cool";
    var another = [1, 2, 3];

    function doSomething() {
        console.log( something );
    }

    function doAnother() {
        console.log( another.join( " ! " ) );
    }

    return {
        doSomething: doSomething,
        doAnother: doAnother
    };
}

var foo = CoolModule();

foo.doSomething(); // cool
foo.doAnother(); // 1 ! 2 ! 3
```

This is the pattern in JavaScript we call *module*. The most common way of implementing the module pattern is often called *revealing module*, and it's the variation we present here.

Let's examine some things about this code.

First, `CoolModule()` is just a function, but it *has to be invoked* for there to be a module instance created. Without the execution of the outer function, the creation of the inner scope and the closures would not occur.

Second, the `CoolModule()` function returns an object, denoted by the object-literal syntax `{ key: value, … }`. The object we return has references on it to our inner functions, but *not* to our inner data variables. We keep those hidden and private. It's appropriate to think of this object return value as essentially a *public API for our module*.

This object return value is ultimately assigned to the outer variable `foo`, and then we can access those property methods on the API, like `foo.doSomething()`.

It is not required that we return an actual object (literal) from our module. We could just return back an inner function directly. jQuery is actually a good example of this. The `jQuery` and `$` identifiers are the public API for the jQuery module, but they are, themselves, just functions (which can themselves have properties, since all functions are objects).

The `doSomething()` and `doAnother()` functions have closure over the inner scope of the module instance (arrived at by actually invoking `CoolModule()`). When we transport those functions outside of the lexical scope, by way of property references on the object we return, we have now set up a condition by which closure can be observed and exercised.

To state it more simply, there are two requirements for the module pattern to be exercised:

1. There must be an outer enclosing function, and it must be invoked at least once (each time creates a new module instance).
2. The enclosing function must return back at least one inner function, so that this inner function has closure over the private scope, and can access and/or modify that private state.

An object with a function property on it alone is not *really* a module. An object that is returned from a function invocation that only has data properties on it and no closured functions is not *really* a module, in the observable sense.

The previous code snippet shows a standalone module creator called `CoolModule()`, which can be invoked any number of times, each time creating a new module instance. A slight variation on this pattern is when you only care to have one instance, a singleton of sorts:

```
var foo = (function CoolModule() {
    var something = "cool";
    var another = [1, 2, 3];

    function doSomething() {
        console.log( something );
    }
```

```
        function doAnother() {
            console.log( another.join( " ! " ) );
        }

        return {
            doSomething: doSomething,
            doAnother: doAnother
        };
    })();

    foo.doSomething(); // cool
    foo.doAnother(); // 1 ! 2 ! 3
```

Here, we turned our module function into an IIFE (see Chapter 3), and we *immediately* invoked it and assigned its return value directly to our single module instance identifier `foo`.

Modules are just functions, so they can receive parameters:

```
    function CoolModule(id) {
        function identify() {
            console.log( id );
        }

        return {
            identify: identify
        };
    }

    var foo1 = CoolModule( "foo 1" );
    var foo2 = CoolModule( "foo 2" );

    foo1.identify(); // "foo 1"
    foo2.identify(); // "foo 2"
```

Another slight but powerful variation on the module pattern is to name the object you are returning as your public API:

```
    var foo = (function CoolModule(id) {
        function change() {
            // modifying the public API
            publicAPI.identify = identify2;
        }

        function identify1() {
            console.log( id );
        }

        function identify2() {
            console.log( id.toUpperCase() );
        }
```

```
    var publicAPI = {
        change: change,
        identify: identify1
    };

    return publicAPI;
})( "foo module" );

foo.identify(); // foo module
foo.change();
foo.identify(); // FOO MODULE
```

By retaining an inner reference to the public API object inside your module instance, you can modify that module instance *from the inside*, including adding and removing methods and properties, *and* changing their values.

Modern Modules

Various module dependency loaders/managers essentially wrap up this pattern of module definition into a friendly API. Rather than examine any one particular library, let me present a *very simple* proof of concept *for illustration purposes (only)*:

```
var MyModules = (function Manager() {
    var modules = {};

    function define(name, deps, impl) {
        for (var i=0; i<deps.length; i++) {
            deps[i] = modules[deps[i]];
        }
        modules[name] = impl.apply( impl, deps );
    }

    function get(name) {
        return modules[name];
    }

    return {
        define: define,
        get: get
    };
})();
```

The key part of this code is `modules[name] = impl.apply(impl, deps)`. This is invoking the definition wrapper function for a module (passing in any dependencies), and storing the return value, the module's API, into an internal list of modules tracked by name.

And here's how I might use it to define some modules:

```
MyModules.define( "bar", [], function(){
    function hello(who) {
        return "Let me introduce: " + who;
    }

    return {
        hello: hello
    };
} );

MyModules.define( "foo", ["bar"], function(bar){
    var hungry = "hippo";

    function awesome() {
        console.log( bar.hello( hungry ).toUpperCase() );
    }

    return {
        awesome: awesome
    };
} );

var bar = MyModules.get( "bar" );
var foo = MyModules.get( "foo" );

console.log(
    bar.hello( "hippo" )
); // Let me introduce: hippo

foo.awesome(); // LET ME INTRODUCE: HIPPO
```

Both the `"foo"` and `"bar"` modules are defined with a function that returns a public API. `"foo"` even receives the instance of `"bar"` as a dependency parameter, and can use it accordingly.

Spend some time examining these code snippets to fully understand the power of closures put to use for our own good purposes. The key take-away is that there's not really any particular "magic" to module managers. They fulfill both characteristics of the module pattern I listed above: invoking a function definition wrapper, and keeping its return value as the API for that module.

In other words, modules are just modules, even if you put a friendly wrapper tool on top of them.

Future Modules

ES6 adds first-class syntax support for the concept of modules. When loaded via the module system, ES6 treats a file as a separate module. Each module can both import other modules or specific API members, as well export their own public API members.

Function-based modules aren't a statically recognized pattern (something the compiler knows about), so their API semantics aren't considered until runtime. That is, you can actually modify a module's API during the runtime (see earlier `publiCAPI` discussion).

By contrast, ES6 module APIs are static (the APIs don't change at runtime). Since the compiler knows *that*, it can (and does!) check during (file loading and) compilation that a reference to a member of an imported module's API *actually exists*. If the API reference doesn't exist, the compiler throws an "early" error at compile time, rather than waiting for traditional dynamic runtime resolution (and errors, if any).

ES6 modules *do not* have an "inline" format, they must be defined in separate files (one per module). The browsers/engines have a default "module loader" (which is overridable, but that's well-beyond our discussion here), which synchronously loads a module file when it's imported.

Consider:

bar.js

```
function hello(who) {
    return "Let me introduce: " + who;
}

export hello;
```

foo.js

```
// import only `hello()` from the "bar" module
import hello from "bar";

var hungry = "hippo";

function awesome() {
    console.log(
        hello( hungry ).toUpperCase()
    );
```

```
}

export awesome;
```

baz.js

```
// import the entire "foo" and "bar" modules
module foo from "foo";
module bar from "bar";

console.log(
    bar.hello( "rhino" )
); // Let me introduce: rhino

foo.awesome(); // LET ME INTRODUCE: HIPPO
```

Separate files *foo.js* and *bar.js* would need to be created, with the contents as shown in the first two snippets, respectively. Then, your program *baz.js* would load/import those modules to use them, as shown in the third snippet.

`import` imports one or more members from a module's API into the current scope, each to a bound variable (`hello` in our case). `module` imports an entire module API to a bound variable (`foo`, `bar` in our case). `export` exports an identifier (variable, function) to the public API for the current module. These operators can be used as many times in a module's definition as is necessary.

The contents inside the *module file* are treated as if enclosed in a scope closure, just like with the function-closure modules seen earlier.

Review

Closure seems to the unenlightened like a mystical world set apart inside of JavaScript that only the few bravest souls can reach. But it's actually just a standard and almost obvious fact of how we write code in a lexically scoped environment, where functions are values and can be passed around at will.

Closure is when a function can remember and access its lexical scope even when it's invoked outside its lexical scope.

Closures can trip us up, for instance with loops, if we're not careful to recognize them and how they work. But they are also an immensely powerful tool, enabling patterns like *modules* in their various forms.

Modules require two key characteristics: 1) an outer wrapping function being invoked, to create the enclosing scope 2) the return value of the wrapping function must include reference to at least one inner function that then has closure over the private inner scope of the wrapper.

Now we can see closures all around our existing code, and we have the ability to recognize and leverage them to our own benefit!

APPENDIX A
Dynamic Scope

In Chapter 2, we talked about dynamic scope as a contrast to the lexical scope model, which is how scope works in JavaScript (and in fact, most other languages).

We will briefly examine dynamic scope, to hammer home the contrast. But, more important, dynamic scope actually is a near cousin to another mechanism (`this`) in JavaScript, which we cover in the *this & Object Prototypes* title of the *You Don't Know JS* book series.

As we saw in Chapter 2, lexical scope is the set of rules about how the engine can look up a variable and where it will find it. The key characteristic of lexical scope is that it is defined at author time, when the code is written (assuming you don't cheat with `eval()` or `with`).

Dynamic scope seems to imply, and for good reason, that there's a model whereby scope can be determined dynamically at runtime, rather than statically at author time. That is in fact the case. Let's illustrate via code:

```
function foo() {
    console.log( a ); // 2
}

function bar() {
    var a = 3;
    foo();
}

var a = 2;

bar();
```

Lexical scope holds that the RHS reference to `a` in `foo()` will be resolved to the global variable `a`, which will result in value `2` being output.

Dynamic scope, by contrast, doesn't concern itself with how and where functions and scopes are declared, but rather *where they are called from*. In other words, the scope chain is based on the call-stack, not the nesting of scopes in code.

So, if JavaScript had dynamic scope, when `foo()` is executed, *theoretically* the code below would instead result in `3` as the output.

```
function foo() {
    console.log( a ); // 3  (not 2!)
}

function bar() {
    var a = 3;
    foo();
}

var a = 2;

bar();
```

How can this be? Because when `foo()` cannot resolve the variable reference for `a`, instead of stepping up the nested (lexical) scope chain, it walks up the call stack, to find where `foo()` was *called from*. Since `foo()` was called from `bar()`, it checks the variables in scope for `bar()`, and finds an `a` there with value `3`.

Strange? You're probably thinking so, at the moment.

But that's just because you've probably only ever worked on (or at least deeply considered) code that is lexically scoped. So dynamic scoping seems foreign. If you had only ever written code in a dynamically scoped language, it would seem natural, and lexical scope would be the odd ball.

To be clear, JavaScript does not, in fact, have dynamic scope. It has lexical scope. Plain and simple. But the `this` mechanism is kind of like dynamic scope.

The key contrast: lexical scope is write-time, whereas dynamic scope (and `this`!) are runtime. Lexical scope cares where a function was declared, but dynamic scope cares where a function was called from.

Finally, `this` cares how a function was called, which shows how closely related the `this` mechanism is to the idea of dynamic scoping. To dig more into `this`, read the *You Don't Know JS* title *this & Object Prototypes*.

APPENDIX B
Polyfilling Block Scope

In Chapter 3, we explored block scope. We saw that `with` and the `catch` clause are both tiny examples of block scope that have existed in JavaScript since at least the introduction of ES3.

But it's ES6's introduction of `let` that finally gives full, unfettered block scoping capability to our code. There are many exciting things, both functionally and code-stylistically, that block scope will enable.

But what if we wanted to use block scope in pre-ES6 environments?

Consider this code:

```
{
    let a = 2;
    console.log( a ); // 2
}

console.log( a ); // ReferenceError
```

This will work great in ES6 environments. But can we do so pre-ES6? `catch` is the answer.

```
try{throw 2}catch(a){
    console.log( a ); // 2
}

console.log( a ); // ReferenceError
```

Whoa! That's some ugly, weird looking code. We see a `try/catch` that appears to forcibly throw an error, but the "error" it throws is just a value `2`, and then the variable declaration that receives it is in the `catch(a)` clause. Mind: blown.

That's right, the `catch` clause has block-scoping to it, which means it can be used as a polyfill for block scope in pre-ES6 environments.

"But", you say, "no one wants to write ugly code like that!" That's true. No one writes (some of) the code output by the CoffeeScript compiler, either. That's not the point.

The point is that tools can transpile ES6 code to work in pre-ES6 environments. You can write code using block scoping, and benefit from such functionality, and let a build-step tool take care of producing code that will actually *work* when deployed.

This is actually the preferred migration path for all (ahem, most) of ES6: to use a code transpiler to take ES6 code and produce ES5-compatible code during the transition from pre-ES6 to ES6.

Traceur

Google maintains a project called Traceur[1], which is exactly tasked with transpiling ES6 features into pre-ES6 (mostly ES5, but not all!) for general usage. The TC39 committee relies on this tool (and others) to test out the semantics of the features they specify.

What does Traceur produce from our snippet? You guessed it!

```
{
    try {
        throw undefined;
    } catch (a) {
        a = 2;
        console.log( a );
    }
}

console.log( a );
```

So, with the use of such tools, we can start taking advantage of block scope regardless of if we are targeting ES6 or not, because `try/catch` has been around (and worked this way) from ES3 days.

1. Google Traceur

Implicit Versus Explicit Blocks

In Chapter 3, we identified some potential pitfalls to code maintainability/refactorability when we introduce block scoping. Is there another way to take advantage of block scope but to reduce this downside?

Consider this alternate form of `let`, called the `let` block or `let` statement (contrasted with `let` declarations from before).

```
let (a = 2) {
    console.log( a ); // 2
}

console.log( a ); // ReferenceError
```

Instead of implicitly hijacking an existing block, the `let` statement creates an explicit block for its scope binding. Not only does the explicit block stand out more, and perhaps fare more robustly in code refactoring, it produces somewhat cleaner code by, grammatically, forcing all the declarations to the top of the block. This makes it easier to look at any block and know what's scoped to it and not.

As a pattern, it mirrors the approach many people take in function scoping when they manually move/hoist all their `var` declarations to the top of the function. The `let` statement puts them there at the top of the block by intent, and if you don't use `let` declarations strewn throughout, your block-scoping declarations are somewhat easier to identify and maintain.

But, there's a problem. The `let` statement form is not included in ES6. Neither does the official Traceur compiler accept that form of code.

We have two options. We can format using ES6-valid syntax and a little sprinkle of code discipline:

```
/*let*/ { let a = 2;
    console.log( a );
}

console.log( a ); // ReferenceError
```

But, tools are meant to solve our problems. So the other option is to write explicit `let` statement blocks, and let a tool convert them to valid, working code.

So, I built a tool called *let-er*[2] to address just this issue. *let-er* is a build-step code transpiler, but its only task is to find `let` statement forms and transpile them. It will leave alone any of the rest of your code, including any `let` declarations. You can safely use *let-er* as the first ES6 transpiler step, and then pass your code through something like Traceur if necessary.

Moreover, *let-er* has a configuration flag `--es6`, which when turned on (off by default), changes the kind of code produced. Instead of the `try/catch` ES3 polyfill hack, *let-er* would take our snippet and produce the fully ES6-compliant, non-hacky:

```
{
    let a = 2;
    console.log( a );
}

console.log( a ); // ReferenceError
```

So, you can start using *let-er* right away, and target all pre-ES6 environments, and when you only care about ES6, you can add the flag and instantly target only ES6.

And most important, you can use the more preferable and more explicit `let` statement form even though it is not an official part of any ES version (yet).

Performance

Let me add one last quick note on the performance of `try/catch`, and/or to address the question, "Why not just use an IIFE to create the scope?"

First, the performance of `try/catch` *is* slower, but there's no reasonable assumption that it *has* to be that way, or even that it *always will be* that way. Since the official TC39-approved ES6 transpiler uses `try/catch`, the Traceur team has asked Chrome to improve the performance of `try/catch`, and they are obviously motivated to do so.

Secondly, IIFE is not a fair apples-to-apples comparison with `try/catch`, because a function wrapped around any arbitrary code changes the meaning, inside of that code, of `this`, `return`, `break`, and

2. let-er on GitHub

`continue`. IIFE is not a suitable general substitute. It could only be used manually in certain cases.

The question really becomes: do you want block scoping, or not. If you do, these tools provide you that option. If not, keep using `var` and go on about your coding!

APPENDIX C
Lexical this

Though this title does not address the `this` mechanism in any detail, there's one ES6 topic that relates `this` to lexical scope in an important way, which we will quickly examine.

ES6 adds a special syntactic form of function declaration called the *arrow function*. It looks like this:

```
var foo = a => {
    console.log( a );
};

foo( 2 ); // 2
```

The so-called "fat arrow" is often mentioned as a shorthand for the *tediously verbose* (sarcasm) `function` keyword.

But there's something much more important going on with arrow functions that has nothing to do with saving keystrokes in your declaration. Briefly, this code suffers a problem:

```
var obj = {
    id: "awesome",
    cool: function coolFn() {
        console.log( this.id );
    }
};

var id = "not awesome"

obj.cool(); // awesome

setTimeout( obj.cool, 100 ); // not awesome
```

The problem is the loss of `this` binding on the `cool()` function. There are various ways to address that problem, but one often-repeated solution is `var self = this;`.

That might look like:

```
var obj = {
    count: 0,
    cool: function coolFn() {
        var self = this;

        if (self.count < 1) {
            setTimeout( function timer(){
                self.count++;
                console.log( "awesome?" );
            }, 100 );
        }
    }
};

obj.cool(); // awesome?
```

Without getting too much into the weeds here, the `var self = this` "solution" just ends-around the whole problem of understanding and properly using `this` binding, and instead falls back to something we're perhaps more comfortable with: lexical scope. `self` becomes just an identifier that can be resolved via lexical scope and closure, and cares not what happened to the `this` binding along the way.

People don't like writing verbose stuff, especially when they do it over and over again. So, a motivation of ES6 is to help alleviate these scenarios, and indeed, *fix* common idiom problems, such as this one.

The ES6 solution, the arrow function, introduces a behavior called lexical `this`.

```
var obj = {
    count: 0,
    cool: function coolFn() {
        if (this.count < 1) {
            setTimeout( () => { // arrow-function ftw?
                this.count++;
                console.log( "awesome?" );
            }, 100 );
        }
    }
};

obj.cool(); // awesome?
```

The short explanation is that arrow functions do not behave at all like normal functions when it comes to their `this` binding. They discard all the normal rules for `this` binding, and instead take on the `this` value of their immediate lexical enclosing scope, whatever it is.

So, in that snippet, the arrow function doesn't get its `this` unbound in some unpredictable way, it just "inherits" the `this` binding of the `cool()` function (which is correct if we invoke it as shown!).

While this makes for shorter code, my perspective is that arrow functions are really just codifying into the language syntax a common *mistake* of developers, which is to confuse and conflate `this` binding rules with lexical scope rules.

Put another way: why go to the trouble and verbosity of using the `this` style coding paradigm, only to cut it off at the knees by mixing it with lexical references. It seems natural to embrace one approach or the other for any given piece of code, and not mix them in the same piece of code.

One other detraction from arrow functions is that they are anonymous, not named. See Chapter 3 for the reasons why anonymous functions are less desirable than named functions.

A more appropriate approach, in my perspective, to this "problem," is to use and embrace the `this` mechanism correctly.

```
var obj = {
    count: 0,
    cool: function coolFn() {
        if (this.count < 1) {
            setTimeout( function timer(){
                this.count++; // `this` is safe
                              // because of `bind(..)`
                console.log( "more awesome" );
            }.bind( this ), 100 ); // look, `bind()`!
        }
    }
};

obj.cool(); // more awesome
```

Whether you prefer the new lexical `this` behavior of arrow functions, or you prefer the tried-and-true `bind()`, it's important to note that arrow functions are *not* just about less typing of `function`.

They have an *intentional behavioral difference* that we should learn and understand, and if we so choose, leverage.

Now that we fully understand lexical scoping (and closure!), understanding lexical `this` should be a breeze!

About the Author

Kyle Simpson is an Open Web Evangelist from Austin, TX. He's passionate about JavaScript, HTML5, real-time/peer-to-peer communications, and web performance. Otherwise, he's probably bored by it. Kyle is an author, workshop trainer, tech speaker, and avid OSS community member.