

Table of Contents

Introduction	1.1
Preface	1.2
Foreword	1.3
up & going	1.4
Into Programming	1.4.1
Into JavaScript	1.4.2
Into YDKJS	1.4.3
Scope & Closures	1.5
What is Scope?	1.5.1
Lexical Scope	1.5.2
Function vs. Block Scope	1.5.3
Hoisting	1.5.4
Scope Closure	1.5.5
Dynamic Scope	1.5.6
Polyfilling Block Scope	1.5.7
Lexical-this	1.5.8
Types & Grammar	1.6
Types	1.6.1
Values	1.6.2
Natives	1.6.3
Coercion	1.6.4
Grammar	1.6.5
Mixed Environment JavaScript	1.6.6
this & Object Prototypes	1.7
this Or That?	1.7.1
this All Makes Sense Now!	1.7.2
Objects	1.7.3
Mixing (Up) "Class" Objects	1.7.4
Prototypes	1.7.5
Behavior Delegation	1.7.6

ES6 class	1.7.7
Async & Performance	1.8
Asynchrony: Now & Later	1.8.1
Callbacks	1.8.2
Promises	1.8.3
Generators	1.8.4
Program Performance	1.8.5
Benchmarking & Tuning	1.8.6
asynquence Library	1.8.7
Advanced Async Patterns	1.8.8
ES6 & Beyond	1.9
ES? Now & Future	1.9.1
Syntax	1.9.2
Organization	1.9.3
Async Flow Control	1.9.4
Collections	1.9.5
API Additions	1.9.6
Meta Programming	1.9.7
Beyond ES6	1.9.8
Acknowledgments	1.10

You Don't Know JS (book series)

This is a series of books diving deep into the core mechanisms of the JavaScript language.
The first edition of the series is now complete.

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 2 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 which 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 which 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 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 which 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.

Summary

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.

Note: 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.

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, as 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 we will be writing.

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 learn 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

jennlukas.com, @jennlukas

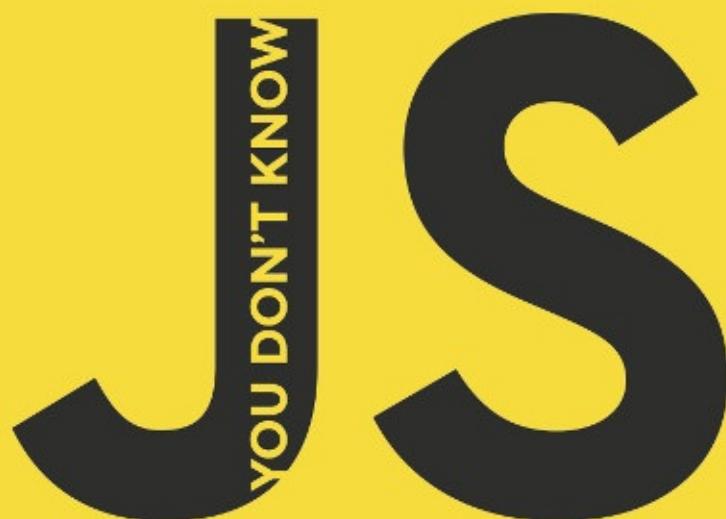
Front-end consultant

O'REILLY®

"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*."
—JENN LUKAS, Frontend consultant

KYLE SIMPSON

UP & GOING



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"), 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") -- 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"), 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.

Note: 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.).

Tip: 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"

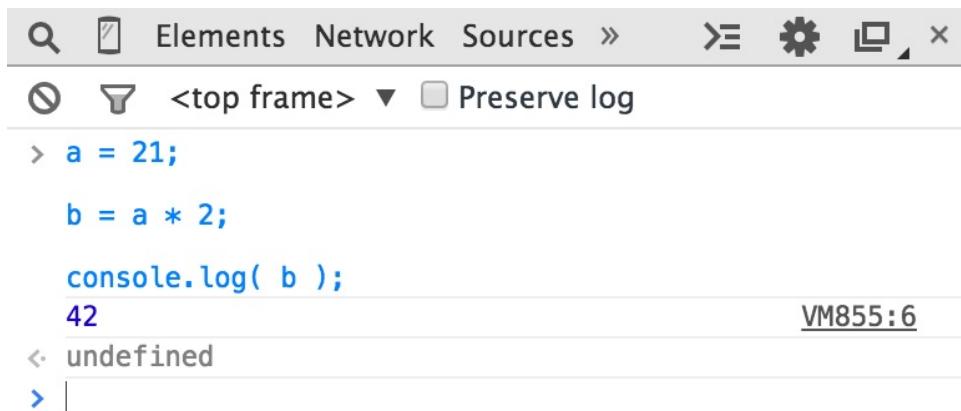
(<http://blog.teamtreehouse.com/mastering-developer-tools-console>). To type multiple lines into the console at once, use `<shift> + <enter>` to move to the next new line. Once you hit `<enter>` by itself, the console will run everything you've just typed.

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:



The screenshot shows the developer tools console in Chrome. The top navigation bar includes 'Elements', 'Network', 'Sources', and other developer tool tabs. Below the tabs, there are filter and log level controls. The main console area shows the following interaction:

```
> a = 21;  
b = a * 2;  
console.log( b );  
42  
< undefined  
> |
```

The output of the `console.log` statement is highlighted in blue and labeled `VM855:6`.

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"). 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 popup "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).

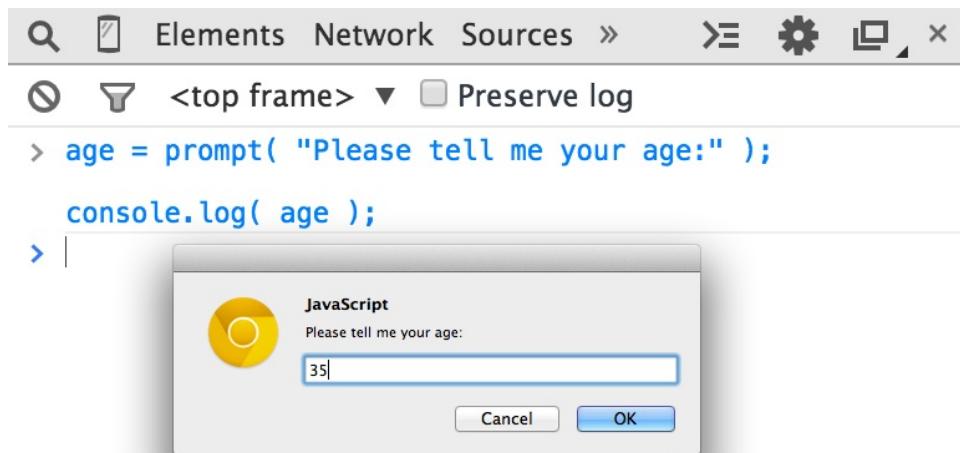
The most common way that happens is for the HTML page to show form elements (like text boxes) to a user that they can type into, and then using JS to read those values into your program's variables.

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 popup.

This should look similar to the following:

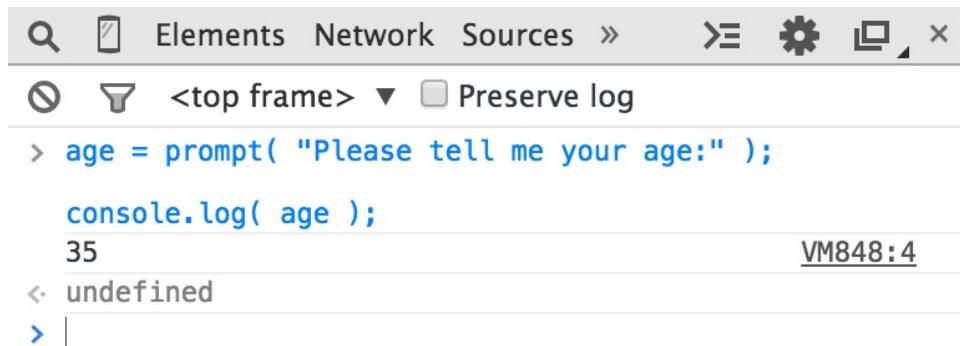


A screenshot of a browser's developer tools console. The URL bar shows '<top frame>'. The console has the following content:

```
> age = prompt( "Please tell me your age:" );
> console.log( age );
> |
```

Below the console, a modal dialog box titled 'JavaScript' with the message 'Please tell me your age:' is displayed. A text input field contains '35'. There are 'Cancel' and 'OK' buttons at the bottom.

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(..)`:



A screenshot of a browser's developer tools console. The URL bar shows '<top frame>'. The console has the following content:

```
> age = prompt( "Please tell me your age:" );
> console.log( age );
> |
```

The output shows '35' followed by the timestamp 'VM848:4'. The line '`< undefined`' is also visible.

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).

Warning: 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 ordering 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*) variables (see "Variables").

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"); 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" 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" 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"), like if either `a` or `b` is true.

Note: For much more detail, and coverage of operators not mentioned here, see the Mozilla Developer Network (MDN)'s "Expressions and Operators" (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators).

Values & Types

If you ask an employee at a phone store how much a certain phone costs, and they say "ninety-nine, ninety-nine" (i.e., \$99.99), they're 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), they're 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, that answer could only have been either "yes" or "no."

In very similar ways, when you express values in a program, 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, 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 (i.e., `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.

Note: 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") and functions (see "Functions").

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 that's 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"

```

Note: 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 `number`s 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" 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!

Note: 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") or a loop (see "Loops"). 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.

Note: 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" earlier, 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" 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") use a *conditional* to determine if the loop should keep going or stop.

Note: 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` (ten numbers). You can do that by setting a loop iteration variable like `i` at value `0` and incrementing it by `1` each iteration.

Warning: 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) {
    // stop the loop?
    if ((i <= 9) === false) {
        break;
    }

    console.log( i );
    i = i + 1;
}

// 0 1 2 3 4 5 6 7 8 9

```

Warning: 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...` statement) 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 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.

Note: 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" earlier. 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!

Note: 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. :(
```

Note: 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 values.
- You need values and *types* to perform different kinds of actions like math on `number` `s` or output with `string` `s`.
- 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.

Note: 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! - the "symbol" type) 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"`.

Warning: 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:

`obj`

a: "hello world"	b: 42	c: true
---------------------	----------	------------

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

```

Note: For more information on JavaScript `object`s, 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"
```

Note: 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:

arr

0: "hello world"	1: 42	2: true
------------------	-------	---------

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 `object`s 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.

Note: 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.

Note: 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 obviously 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 (<http://www.ecma-international.org/ecma-262/5.1/>) 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 `object` s (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, `array` s are by default coerced to `string` s by simply joining all the values with commas (`,`) in between. You might think that two `array` s 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
```

Note: 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 `number` s. 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 `string`s, 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 `number`s, 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.

Note: 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`.

Note: 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
```

Warning: 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"). 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.

Note: 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.

Note: 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, almost certainly it's 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!

Note: For more information about strict mode, see the 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 `return ed` 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.

Warning: We are not calling `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` identifier. 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.

Note: 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.

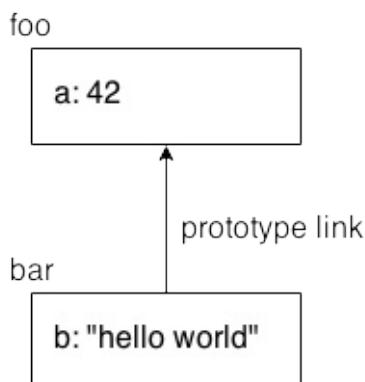
The internal prototype reference linkage from one object to its fallback happens at the time the object is created. The simplest way to illustrate it is with a built-in utility called

```
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.

Note: 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) (<https://remysharp.com/2010/10/08/what-is-a-polyfill>) 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(..)`.

Note: 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 (<https://github.com/es-shims/es5-shim>) and ES6-Shim (<https://github.com/es-shims/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 (<https://babeljs.io>) (formerly 6to5): Transpiles ES6+ into ES5
- Traceur (<https://github.com/google/traceur-compiler>): 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" (*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 reasonable.

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 mid-way 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 ...

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.

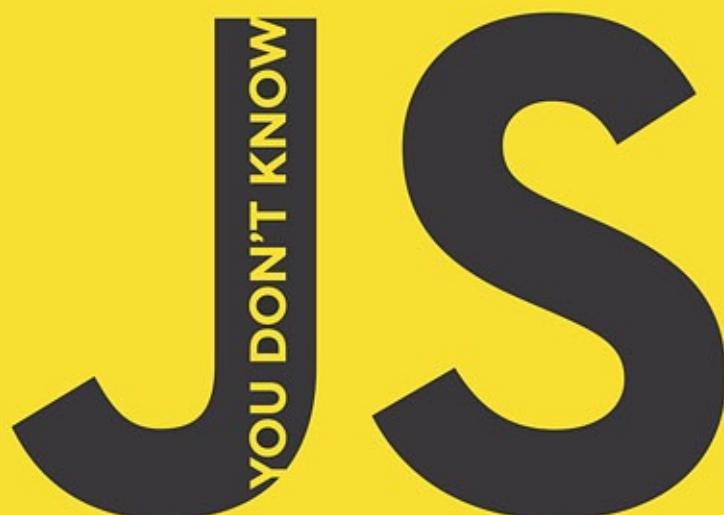
"You Don't Know JS" isn't a criticism or an insult. It's a realization that all of us, myself included, must come to terms with. Learning JavaScript isn't an end goal but a process. We don't know JavaScript, yet. But we will!

O'REILLY®

"Kyle's way of critically thinking about every tiny bit of the language will creep into your mindset and general workflow."
—SHANE HUDSON, freelance frontend website developer

KYLE SIMPSON

SCOPE & CLOSURES



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

1. **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 it's meaningful or not.

Note: 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**.

2. **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" (**Abstract Syntax Tree**).

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`).

3. **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, etc.

So, rather than get mired in details, we'll just handwave and say that there's a way to take our above 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`.

Note: 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 re-compile, etc.) which are well beyond the "scope" of our discussion here.

Let's just say, for simplicity's 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:

1. *Engine*: responsible for start-to-finish compilation and execution of our JavaScript program.
2. *Compiler*: one of *Engine*'s friends; handles all the dirty work of parsing and code-generation (see previous section).
3. *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 & 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 which *Compiler* will handle during compilation, and one which *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 pseudo-code: "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* section below).

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 "Left-hand Side" and "Right-hand Side".

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

In other words, an LHS look-up is done when a variable appears on the left-hand side of an assignment operation, and an RHS look-up is done when a variable appears on the right-hand 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 "right-hand side of an assignment" per se, it just, more accurately, means "not left-hand side".

Being slightly glib for a moment, you could also 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.

Note: LHS and RHS meaning "left/right-hand side of an assignment" 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. **Did you spot it?**

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.

Note: 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. He'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`. He'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 guy, 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 the *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!).

Note: 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 outer containing scope, continuing until found or until the outermost (aka, global) scope has been reached.

Consider:

```
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."

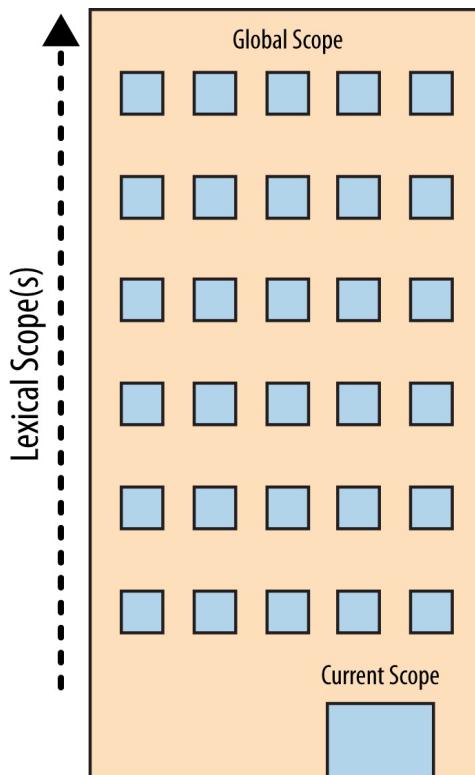
Engine: "Hey, *Scope* outside of `foo`, oh you're the global *Scope*, ok cool. Ever heard of `b`? Got an RHS reference for it."

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** rule set. 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 Scopes, 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 arrives at the top floor (global Scope) without finding it, and if the program is not running in "Strict Mode" [note-strictmode](#), 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" [note-strictmode](#), 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 Scope'd 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 non-function 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 (TL;DR)

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 assigning to the variable, which is an LHS (left-hand-side) reference, or it may be for the purposes of retrieving its value, which is an RHS (right-hand-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` being thrown. Unfulfilled LHS references result in an automatic, implicitly-created global of that name (if not in "Strict Mode" [note-strictmode](#)), or a `ReferenceError` (if in "Strict Mode" [note-strictmode](#)).

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`

[note-strictmode](#). MDN: [Strict Mode ↴](#)

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 (aka, 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 which 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.

Note: We will see in a little bit 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.

```
function foo(a) {
  var b = a * 2;
  function bar(c) {
    console.log( a, b, c );
  }
  bar(b * 3);
}

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

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 above 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 identifier "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.

Note: Global variables are also automatically 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 which 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" (aka, cheat) lexical scope at run-time?

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 programmatically 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 (aka, 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 look-ups 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 above, 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.

Note: In this example, for simplicity's sake, the string of "code" we pass in was a fixed literal. But it could easily have been programmatically 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.

Note: `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 which 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 evaluated 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 which 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 short-hand 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 short-hand 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 which 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".

Note: 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.

Note: 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 pre-determine 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 pessimistic assumptions, **there's no getting around the fact that without the optimizations, code runs slower.**

Review (TL;DR)

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" which 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 vs. 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" [note-leastprivilege](#), 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 which 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 pre-condition assumptions of `doSomething(..)`.

A more "proper" design would hide these private details inside the scope of `doSomething(..)`, such as:

```
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, infinite 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 of 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.

Note: 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 vs. 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 draw-backs 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 that 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 vs. named doesn't detract from that. Providing a name for your function expression quite effectively addresses all these draw-backs, 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, IIFE's 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 IIFE's which 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 vs. non-global 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 wide-spread 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 Exposure" [note-leastprivilege](#) 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 importantly, developers may prefer to *check* themselves against accidentally (re)using variables outside of their intended purpose, such as 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 re-used 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.

Note: 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 which each declare their error variable with the same identifier name. This is not actually a re-definition, 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 which 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 you 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`-statement.

Note: 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 **re-binds it** to each *iteration* of the loop, making sure to re-assign 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 re-factored 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 which 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 (TL;DR)

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.

note-leastprivilege . [Principle of Least Privilege ↵](#)

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 which 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 re-defined, 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)**.

Note: 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 (TL;DR)

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 which promised even more capability than I already possessed, 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 *a-ha!* 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-n-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 which 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 it, that closure will be exercised.

Now I Can See

The previous code snippets are somewhat academic and artificially 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 inner function `timer` 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 which 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 (which 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!

Note: 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 definition above.

```
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 (the 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 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 + 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 );
}
```

Note: 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 5 times, at the one-second intervals.

Huh?

Firstly, 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 5 of those functions, though they are defined separately in each loop iteration, all **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 5 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 `cowbell` closed scope. Specifically, we need a new closed 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 which leverage the power of closure but which 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.

Firstly, `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.

Secondly, 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()`.

Note: 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 a function (which can itself 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 which is returned from a function invocation which only has data properties on it and no closed functions is not *really* a module, in the observable sense.

The code snippet above 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, 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.

Note: Function-based modules aren't a statically recognized pattern (something the compiler knows about), so their API semantics aren't considered until run-time. That is, you can actually modify a module's API during the run-time (see earlier `publicAPI` discussion).

By contrast, ES6 Module APIs are static (the APIs don't change at run-time). 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 run-time 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;
```

```
// 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
```

Note: 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 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 (TL;DR)

Closure seems to the un-enlightened like a mystical world set apart inside of JavaScript which 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 importantly, dynamic scope actually is a near cousin to another mechanism (`this`) in JavaScript, which we covered in the "*this & Object Prototypes*" title of this 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 which 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 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" [note-traceur](#), 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.

Implicit vs. 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" [note-let_er](#) 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 importantly, **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?"

Firstly, 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 `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!

note-traceur . [Google Traceur ↵](#)

`note-let_er`\: `let-er`

Appendix C: Lexical-this

Though this title does not address the `this` mechanism in any detail, there's one ES6 topic which 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 short-hand 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 dispenses with 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.

Note: 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.

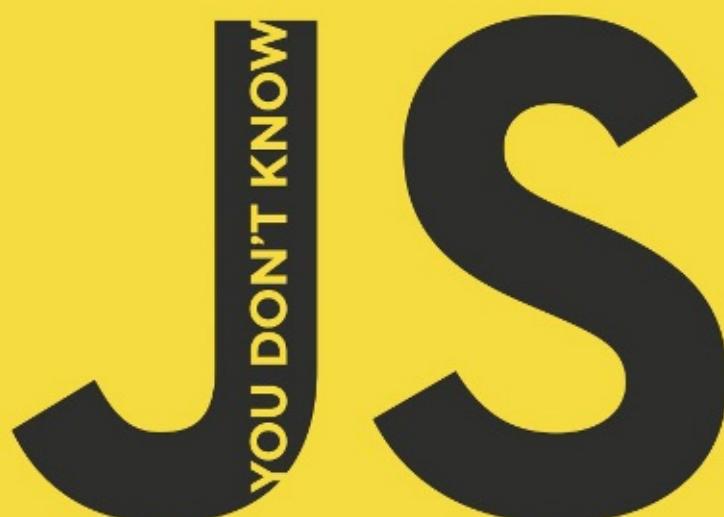
Now that we fully understand lexical scoping (and closure!), understanding lexical-this should be a breeze!

O'REILLY®

"An excellent look at the core JavaScript fundamentals that copy
and paste and JavaScript toolkits don't and could never teach you."
—DAVID WALSH, Senior Web Developer, Mozilla

KYLE SIMPSON

TYPES & GRAMMAR



Chapter 1: Types

Most developers would say that a dynamic language (like JS) does not have *types*. Let's see what the ES5.1 specification (<http://www.ecma-international.org/ecma-262/5.1/>) has to say on the topic:

Algorithms within this specification manipulate values each of which has an associated type. The possible value types are exactly those defined in this clause. Types are further sub classified into ECMAScript language types and specification types.

An ECMAScript language type corresponds to values that are directly manipulated by an ECMAScript programmer using the ECMAScript language. The ECMAScript language types are Undefined, Null, Boolean, String, Number, and Object.

Now, if you're a fan of strongly typed (statically typed) languages, you may object to this usage of the word "type." In those languages, "type" means a whole lot *more* than it does here in JS.

Some people say JS shouldn't claim to have "types," and they should instead be called "tags" or perhaps "subtypes".

Bah! We're going to use this rough definition (the same one that seems to drive the wording of the spec): a *type* is an intrinsic, built-in set of characteristics that uniquely identifies the behavior of a particular value and distinguishes it from other values, both to the engine **and to the developer**.

In other words, if both the engine and the developer treat value `42` (the number) differently than they treat value `"42"` (the string), then those two values have different *types* -- `number` and `string`, respectively. When you use `42`, you are *intending* to do something numeric, like math. But when you use `"42"`, you are *intending* to do something string-ish, like outputting to the page, etc. **These two values have different types.**

That's by no means a perfect definition. But it's good enough for this discussion. And it's consistent with how JS describes itself.

A Type By Any Other Name...

Beyond academic definition disagreements, why does it matter if JavaScript has *types* or not?

Having a proper understanding of each *type* and its intrinsic behavior is absolutely essential to understanding how to properly and accurately convert values to different types (see Coercion, Chapter 4). Nearly every JS program ever written will need to handle value coercion in some shape or form, so it's important you do so responsibly and with confidence.

If you have the `number` value `42`, but you want to treat it like a `string`, such as pulling out the `"2"` as a character in position `1`, you obviously must first convert (coerce) the value from `number` to `string`.

That seems simple enough.

But there are many different ways that such coercion can happen. Some of these ways are explicit, easy to reason about, and reliable. But if you're not careful, coercion can happen in very strange and surprising ways.

Coercion confusion is perhaps one of the most profound frustrations for JavaScript developers. It has often been criticized as being so *dangerous* as to be considered a flaw in the design of the language, to be shunned and avoided.

Armed with a full understanding of JavaScript types, we're aiming to illustrate why coercion's *bad reputation* is largely overhyped and somewhat undeserved -- to flip your perspective, to seeing coercion's power and usefulness. But first, we have to get a much better grip on values and types.

Built-in Types

JavaScript defines seven built-in types:

- `null`
- `undefined`
- `boolean`
- `number`
- `string`
- `object`
- `symbol` -- added in ES6!

Note: All of these types except `object` are called "primitives".

The `typeof` operator inspects the type of the given value, and always returns one of seven string values -- surprisingly, there's not an exact 1-to-1 match with the seven built-in types we just listed.

```

typeof undefined === "undefined"; // true
typeof true === "boolean"; // true
typeof 42 === "number"; // true
typeof "42" === "string"; // true
typeof { life: 42 } === "object"; // true

// added in ES6!
typeof Symbol() === "symbol"; // true

```

These six listed types have values of the corresponding type and return a string value of the same name, as shown. `Symbol` is a new data type as of ES6, and will be covered in Chapter 3.

As you may have noticed, I excluded `null` from the above listing. It's *special* -- special in the sense that it's buggy when combined with the `typeof` operator:

```
typeof null === "object"; // true
```

It would have been nice (and correct!) if it returned `"null"`, but this original bug in JS has persisted for nearly two decades, and will likely never be fixed because there's too much existing web content that relies on its buggy behavior that "fixing" the bug would *create* more "bugs" and break a lot of web software.

If you want to test for a `null` value using its type, you need a compound condition:

```

var a = null;

(!a && typeof a === "object"); // true

```

`null` is the only primitive value that is "falsy" (aka false-like; see Chapter 4) but that also returns `"object"` from the `typeof` check.

So what's the seventh string value that `typeof` can return?

```
typeof function a(){ /* .. */ } === "function"; // true
```

It's easy to think that `function` would be a top-level built-in type in JS, especially given this behavior of the `typeof` operator. However, if you read the spec, you'll see it's actually a "subtype" of object. Specifically, a function is referred to as a "callable object" -- an object that has an internal `[[Call]]` property that allows it to be invoked.

The fact that functions are actually objects is quite useful. Most importantly, they can have properties. For example:

```
function a(b,c) {  
    /* .. */  
}
```

The function object has a `length` property set to the number of formal parameters it is declared with.

```
a.length; // 2
```

Since you declared the function with two formal named parameters (`b` and `c`), the "length of the function" is `2`.

What about arrays? They're native to JS, so are they a special type?

```
typeof [1,2,3] === "object"; // true
```

Nope, just objects. It's most appropriate to think of them also as a "subtype" of object (see Chapter 3), in this case with the additional characteristics of being numerically indexed (as opposed to just being string-keyed like plain objects) and maintaining an automatically updated `.length` property.

Values as Types

In JavaScript, variables don't have types -- **values have types**. Variables can hold any value, at any time.

Another way to think about JS types is that JS doesn't have "type enforcement," in that the engine doesn't insist that a *variable* always holds values of the *same initial type* that it starts out with. A variable can, in one assignment statement, hold a `string`, and in the next hold a `number`, and so on.

The `value 42` has an intrinsic type of `number`, and its *type* cannot be changed. Another value, like `"42"` with the `string` type, can be created *from* the `number` value `42` through a process called **coercion** (see Chapter 4).

If you use `typeof` against a variable, it's not asking "what's the type of the variable?" as it may seem, since JS variables have no types. Instead, it's asking "what's the type of the *value in the variable*?"

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

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

The `typeof` operator always returns a string. So:

```
typeof typeof 42; // "string"
```

The first `typeof 42` returns `"number"`, and `typeof "number"` is `"string"`.

undefined vs "undeclared"

Variables that have no value *currently*, actually have the `undefined` value. Calling `typeof` against such variables will return `"undefined"`:

```
var a;

typeof a; // "undefined"

var b = 42;
var c;

// later
b = c;

typeof b; // "undefined"
typeof c; // "undefined"
```

It's tempting for most developers to think of the word "undefined" and think of it as a synonym for "undeclared." However, in JS, these two concepts are quite different.

An "undefined" variable is one that has been declared in the accessible scope, but *at the moment* has no other value in it. By contrast, an "undeclared" variable is one that has not been formally declared in the accessible scope.

Consider:

```
var a;

a; // undefined
b; // ReferenceError: b is not defined
```

An annoying confusion is the error message that browsers assign to this condition. As you can see, the message is "b is not defined," which is of course very easy and reasonable to confuse with "b is undefined." Yet again, "undefined" and "is not defined" are very different things. It'd be nice if the browsers said something like "b is not found" or "b is not declared," to reduce the confusion!

There's also a special behavior associated with `typeof` as it relates to undeclared variables that even further reinforces the confusion. Consider:

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

The `typeof` operator returns `"undefined"` even for "undeclared" (or "not defined") variables. Notice that there was no error thrown when we executed `typeof b`, even though `b` is an undeclared variable. This is a special safety guard in the behavior of `typeof`.

Similar to above, it would have been nice if `typeof` used with an undeclared variable returned "undeclared" instead of conflating the result value with the different "undefined" case.

typeof Undeclared

Nevertheless, this safety guard is a useful feature when dealing with JavaScript in the browser, where multiple script files can load variables into the shared global namespace.

Note: Many developers believe there should never be any variables in the global namespace, and that everything should be contained in modules and private/separate namespaces. This is great in theory but nearly impossible in practicality; still it's a good goal to strive toward! Fortunately, ES6 added first-class support for modules, which will eventually make that much more practical.

As a simple example, imagine having a "debug mode" in your program that is controlled by a global variable (flag) called `DEBUG`. You'd want to check if that variable was declared before performing a debug task like logging a message to the console. A top-level global `var DEBUG = true` declaration would only be included in a "debug.js" file, which you only load into the browser when you're in development/testing, but not in production.

However, you have to take care in how you check for the global `DEBUG` variable in the rest of your application code, so that you don't throw a `ReferenceError`. The safety guard on `typeof` is our friend in this case.

```
// oops, this would throw an error!
if (DEBUG) {
    console.log( "Debugging is starting" );
}

// this is a safe existence check
if (typeof DEBUG !== "undefined") {
    console.log( "Debugging is starting" );
}
```

This sort of check is useful even if you're not dealing with user-defined variables (like `DEBUG`). If you are doing a feature check for a built-in API, you may also find it helpful to check without throwing an error:

```
if (typeof atob === "undefined") {
    atob = function() { /*...*/ };
}
```

Note: If you're defining a "polyfill" for a feature if it doesn't already exist, you probably want to avoid using `var` to make the `atob` declaration. If you declare `var atob` inside the `if` statement, this declaration is hoisted (see the *Scope & Closures* title of this series) to the top of the scope, even if the `if` condition doesn't pass (because the global `atob` already exists!). In some browsers and for some special types of global built-in variables (often called "host objects"), this duplicate declaration may throw an error. Omitting the `var` prevents this hoisted declaration.

Another way of doing these checks against global variables but without the safety guard feature of `typeof` is to observe that all global variables are also properties of the global object, which in the browser is basically the `window` object. So, the above checks could have been done (quite safely) as:

```
if (window.DEBUG) {
    // ...
}

if (!window.atob) {
    // ...
}
```

Unlike referencing undeclared variables, there is no `ReferenceError` thrown if you try to access an object property (even on the global `window` object) that doesn't exist.

On the other hand, manually referencing the global variable with a `window` reference is something some developers prefer to avoid, especially if your code needs to run in multiple JS environments (not just browsers, but server-side node.js, for instance), where the global object may not always be called `window`.

Technically, this safety guard on `typeof` is useful even if you're not using global variables, though these circumstances are less common, and some developers may find this design approach less desirable. Imagine a utility function that you want others to copy-and-paste into their programs or modules, in which you want to check to see if the including program has defined a certain variable (so that you can use it) or not:

```
function doSomethingCool() {
  var helper =
    (typeof FeatureXYZ !== "undefined") ?
    FeatureXYZ :
    function() { /*.. default feature ..*/ };

  var val = helper();
  // ...
}
```

`doSomethingCool()` tests for a variable called `FeatureXYZ`, and if found, uses it, but if not, uses its own. Now, if someone includes this utility in their module/program, it safely checks if they've defined `FeatureXYZ` or not:

```
// an IIFE (see "Immediately Invoked Function Expressions"
// discussion in the *Scope & Closures* title of this series)
(function(){
  function FeatureXYZ() { /*.. my XYZ feature ..*/ }

  // include `doSomethingCool(..)`
  function doSomethingCool() {
    var helper =
      (typeof FeatureXYZ !== "undefined") ?
      FeatureXYZ :
      function() { /*.. default feature ..*/ };

    var val = helper();
    // ...
  }

  doSomethingCool();
})();
```

Here, `FeatureXYZ` is not at all a global variable, but we're still using the safety guard of `typeof` to make it safe to check for. And importantly, here there is *no* object we can use (like we did for global variables with `window.____`) to make the check, so `typeof` is quite helpful.

Other developers would prefer a design pattern called "dependency injection," where instead of `doSomethingCool()` inspecting implicitly for `FeatureXYZ` to be defined outside/around it, it would need to have the dependency explicitly passed in, like:

```
function doSomethingCool(FeatureXYZ) {  
  var helper = FeatureXYZ ||  
    function() { /*.. default feature ..*/ };  
  
  var val = helper();  
  // ..  
}
```

There are lots of options when designing such functionality. No one pattern here is "correct" or "wrong" -- there are various tradeoffs to each approach. But overall, it's nice that the `typeof` undeclared safety guard gives us more options.

Review

JavaScript has seven built-in *types*: `null`, `undefined`, `boolean`, `number`, `string`, `object`, `symbol`. They can be identified by the `typeof` operator.

Variables don't have types, but the values in them do. These types define intrinsic behavior of the values.

Many developers will assume "undefined" and "undeclared" are roughly the same thing, but in JavaScript, they're quite different. `undefined` is a value that a declared variable can hold. "Undeclared" means a variable has never been declared.

JavaScript unfortunately kind of conflates these two terms, not only in its error messages ("ReferenceError: a is not defined") but also in the return values of `typeof`, which is "`undefined`" for both cases.

However, the safety guard (preventing an error) on `typeof` when used against an undeclared variable can be helpful in certain cases.

Chapter 2: Values

`array` `s`, `string` `s`, and `number` `s` are the most basic building-blocks of any program, but JavaScript has some unique characteristics with these types that may either delight or confound you.

Let's look at several of the built-in value types in JS, and explore how we can more fully understand and correctly leverage their behaviors.

Arrays

As compared to other type-enforced languages, JavaScript `array` `s` are just containers for any type of value, from `string` to `number` to `object` to even another `array` (which is how you get multidimensional `array` `s`).

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

a.length;          // 3
a[0] === 1;        // true
a[2][0] === 3;    // true
```

You don't need to presize your `array` `s` (see "Arrays" in Chapter 3), you can just declare them and add values as you see fit:

```
var a = [ ];

a.length;          // 0

a[0] = 1;
a[1] = "2";
a[2] = [ 3 ];

a.length;          // 3
```

Warning: Using `delete` on an `array` value will remove that slot from the `array`, but even if you remove the final element, it does **not** update the `length` property, so be careful! We'll cover the `delete` operator itself in more detail in Chapter 5.

Be careful about creating "sparse" `array` `s` (leaving or creating empty/missing slots):

```
var a = [ ];
a[0] = 1;
// no `a[1]` slot set here
a[2] = [ 3 ];
a[1];      // undefined
a.length;   // 3
```

While that works, it can lead to some confusing behavior with the "empty slots" you leave in between. While the slot appears to have the `undefined` value in it, it will not behave the same as if the slot is explicitly set (`a[1] = undefined`). See "Arrays" in Chapter 3 for more information.

`array`s are numerically indexed (as you'd expect), but the tricky thing is that they also are objects that can have `string` keys/properties added to them (but which don't count toward the `length` of the `array`):

```
var a = [ ];
a[0] = 1;
a["foobar"] = 2;

a.length;      // 1
a["foobar"];   // 2
a.foobar;      // 2
```

However, a gotcha to be aware of is that if a `string` value intended as a key can be coerced to a standard base-10 `number`, then it is assumed that you wanted to use it as a `number` index rather than as a `string` key!

```
var a = [ ];
a["13"] = 42;

a.length; // 14
```

Generally, it's not a great idea to add `string` keys/properties to `array`s. Use `object`s for holding values in keys/properties, and save `array`s for strictly numerically indexed values.

Array-Likes

There will be occasions where you need to convert an `array`-like value (a numerically indexed collection of values) into a true `array`, usually so you can call array utilities (like `indexOf(..)`, `concat(..)`, `forEach(..)`, etc.) against the collection of values.

For example, various DOM query operations return lists of DOM elements that are not true `array`s but are `array`-like enough for our conversion purposes. Another common example is when functions expose the `arguments` (`array`-like) object (as of ES6, deprecated) to access the arguments as a list.

One very common way to make such a conversion is to borrow the `slice(..)` utility against the value:

```
function foo() {
  var arr = Array.prototype.slice.call( arguments );
  arr.push( "bam" );
  console.log( arr );
}

foo( "bar", "baz" ); // ["bar", "baz", "bam"]
```

If `slice()` is called without any other parameters, as it effectively is in the above snippet, the default values for its parameters have the effect of duplicating the `array` (or, in this case, `array`-like).

As of ES6, there's also a built-in utility called `Array.from(..)` that can do the same task:

```
...
var arr = Array.from( arguments );
...
```

Note: `Array.from(..)` has several powerful capabilities, and will be covered in detail in the *ES6 & Beyond* title of this series.

Strings

It's a very common belief that `string`s are essentially just `array`s of characters. While the implementation under the covers may or may not use `array`s, it's important to realize that JavaScript `string`s are really not the same as `array`s of characters. The similarity is mostly just skin-deep.

For example, let's consider these two values:

```
var a = "foo";
var b = ["f", "o", "o"];
```

Strings do have a shallow resemblance to `array`s -- `array`-likes, as above -- for instance, both of them having a `length` property, an `indexOf(..)` method (`array` version only as of ES5), and a `concat(..)` method:

```
a.length;                      // 3
b.length;                      // 3

a.indexOf( "o" );               // 1
b.indexOf( "o" );               // 1

var c = a.concat( "bar" );      // "foobar"
var d = b.concat( ["b", "a", "r"] ); // ["f", "o", "o", "b", "a", "r"]

a === c;                        // false
b === d;                        // false

a;                             // "foo"
b;                             // ["f", "o", "o"]
```

So, they're both basically just "arrays of characters", right? **Not exactly**:

```
a[1] = "O";
b[1] = "O";

a; // "foo"
b; // ["f", "O", "o"]
```

JavaScript `string`s are immutable, while `array`s are quite mutable. Moreover, the `a[1]` character position access form was not always widely valid JavaScript. Older versions of IE did not allow that syntax (but now they do). Instead, the *correct* approach has been

```
a.charAt(1) .
```

A further consequence of immutable `string`s is that none of the `string` methods that alter its contents can modify in-place, but rather must create and return new `string`s. By contrast, many of the methods that change `array` contents actually *do* modify in-place.

```
c = a.toUpperCase();
a === c;      // false
a;            // "foo"
c;            // "FOO"

b.push( "!" );
b;            // ["f", "o", "o", "!"]
```

Also, many of the `array` methods that could be helpful when dealing with `string`s are not actually available for them, but we can "borrow" non-mutation `array` methods against our `string`:

```
a.join;          // undefined
a.map;          // undefined

var c = Array.prototype.join.call( a, "-" );
var d = Array.prototype.map.call( a, function(v){
    return v.toUpperCase() + ".";
} ).join( "" );

c;            // "f-o-o"
d;            // "F.O.O."
```

Let's take another example: reversing a `string` (incidentally, a common JavaScript interview trivia question!). `array`s have a `reverse()` in-place mutator method, but `string`s do not:

```
a.reverse;      // undefined

b.reverse();    // ["!", "o", "o", "f"]
b;              // ["!", "o", "o", "f"]
```

Unfortunately, this "borrowing" doesn't work with `array` mutators, because `string`s are immutable and thus can't be modified in place:

```
Array.prototype.reverse.call( a );
// still returns a String object wrapper (see Chapter 3)
// for "foo" :(
```

Another workaround (aka hack) is to convert the `string` into an `array`, perform the desired operation, then convert it back to a `string`.

```
var c = a
  // split `a` into an array of characters
  .split( "" )
  // reverse the array of characters
  .reverse()
  // join the array of characters back to a string
  .join( "" );

c; // "oof"
```

If that feels ugly, it is. Nevertheless, *it works* for simple `string`s, so if you need something quick-n-dirty, often such an approach gets the job done.

Warning: Be careful! This approach **doesn't work** for `string`s with complex (unicode) characters in them (astral symbols, multibyte characters, etc.). You need more sophisticated library utilities that are unicode-aware for such operations to be handled accurately. Consult Mathias Bynens' work on the subject: *Esrever* (<https://github.com/mathiasbynens/esrever>).

The other way to look at this is: if you are more commonly doing tasks on your "strings" that treat them as basically *arrays of characters*, perhaps it's better to just actually store them as `array`s rather than as `string`s. You'll probably save yourself a lot of hassle of converting from `string` to `array` each time. You can always call `join("")` on the `array of characters` whenever you actually need the `string` representation.

Numbers

JavaScript has just one numeric type: `number`. This type includes both "integer" values and fractional decimal numbers. I say "integer" in quotes because it's long been a criticism of JS that there are not true integers, as there are in other languages. That may change at some point in the future, but for now, we just have `number`s for everything.

So, in JS, an "integer" is just a value that has no fractional decimal value. That is, `42.0` is as much an "integer" as `42`.

Like most modern languages, including practically all scripting languages, the implementation of JavaScript's `number`s is based on the "IEEE 754" standard, often called "floating-point." JavaScript specifically uses the "double precision" format (aka "64-bit binary") of the standard.

There are many great write-ups on the Web about the nitty-gritty details of how binary floating-point numbers are stored in memory, and the implications of those choices. Because understanding bit patterns in memory is not strictly necessary to understand how to correctly

use `number`s in JS, we'll leave it as an exercise for the interested reader if you'd like to dig further into IEEE 754 details.

Numeric Syntax

Number literals are expressed in JavaScript generally as base-10 decimal literals. For example:

```
var a = 42;
var b = 42.3;
```

The leading portion of a decimal value, if `0`, is optional:

```
var a = 0.42;
var b = .42;
```

Similarly, the trailing portion (the fractional) of a decimal value after the `.`, if `0`, is optional:

```
var a = 42.0;
var b = 42.;
```

Warning: `42.` is pretty uncommon, and probably not a great idea if you're trying to avoid confusion when other people read your code. But it is, nevertheless, valid.

By default, most `number`s will be outputted as base-10 decimals, with trailing fractional `0`s removed. So:

```
var a = 42.300;
var b = 42.0;

a; // 42.3
b; // 42
```

Very large or very small `number`s will by default be outputted in exponent form, the same as the output of the `toExponential()` method, like:

```
var a = 5E10;
a;                      // 50000000000
a.toExponential();      // "5e+10"

var b = a * a;
b;                      // 2.5e+21

var c = 1 / a;
c;                      // 2e-11
```

Because `number` values can be boxed with the `Number` object wrapper (see Chapter 3), `number` values can access methods that are built into the `Number.prototype` (see Chapter 3). For example, the `toFixed(..)` method allows you to specify how many fractional decimal places you'd like the value to be represented with:

```
var a = 42.59;

a.toFixed( 0 ); // "43"
a.toFixed( 1 ); // "42.6"
a.toFixed( 2 ); // "42.59"
a.toFixed( 3 ); // "42.590"
a.toFixed( 4 ); // "42.5900"
```

Notice that the output is actually a `string` representation of the `number`, and that the value is `0`-padded on the right-hand side if you ask for more decimals than the value holds.

`toPrecision(..)` is similar, but specifies how many *significant digits* should be used to represent the value:

```
var a = 42.59;

a.toPrecision( 1 ); // "4e+1"
a.toPrecision( 2 ); // "43"
a.toPrecision( 3 ); // "42.6"
a.toPrecision( 4 ); // "42.59"
a.toPrecision( 5 ); // "42.590"
a.toPrecision( 6 ); // "42.5900"
```

You don't have to use a variable with the value in it to access these methods; you can access these methods directly on `number` literals. But you have to be careful with the `.` operator. Since `.` is a valid numeric character, it will first be interpreted as part of the `number` literal, if possible, instead of being interpreted as a property accessor.

```
// invalid syntax:  
42.toFixed( 3 ); // SyntaxError  
  
// these are all valid:  
(42).toFixed( 3 ); // "42.000"  
0.42.toFixed( 3 ); // "0.420"  
42..toFixed( 3 ); // "42.000"
```

`42.toFixed(3)` is invalid syntax, because the `.` is swallowed up as part of the `42.` literal (which is valid -- see above!), and so then there's no `.` property operator present to make the `.toFixed` access.

`42..toFixed(3)` works because the first `.` is part of the `number` and the second `.` is the property operator. But it probably looks strange, and indeed it's very rare to see something like that in actual JavaScript code. In fact, it's pretty uncommon to access methods directly on any of the primitive values. Uncommon doesn't mean *bad* or *wrong*.

Note: There are libraries that extend the built-in `Number.prototype` (see Chapter 3) to provide extra operations on/with `number`s, and so in those cases, it's perfectly valid to use something like `10..makeItRain()` to set off a 10-second money raining animation, or something else silly like that.

This is also technically valid (notice the space):

```
42 .toFixed(3); // "42.000"
```

However, with the `number` literal specifically, **this is particularly confusing coding style** and will serve no other purpose but to confuse other developers (and your future self). Avoid it.

`number`s can also be specified in exponent form, which is common when representing larger `number`s, such as:

```
var onethousand = 1E3; // means 1 * 10^3  
var onemilliononehundredthousand = 1.1E6; // means 1.1 * 10^6
```

`number` literals can also be expressed in other bases, like binary, octal, and hexadecimal.

These formats work in current versions of JavaScript:

```
0xf3; // hexadecimal for: 243  
0xF3; // ditto  
  
0363; // octal for: 243
```

Note: Starting with ES6 + `strict` mode, the `0363` form of octal literals is no longer allowed (see below for the new form). The `0363` form is still allowed in non-`strict` mode, but you should stop using it anyway, to be future-friendly (and because you should be using `strict` mode by now!).

As of ES6, the following new forms are also valid:

```
00363;          // octal for: 243
00363;          // ditto

0b11110011;    // binary for: 243
0B11110011;    // ditto
```

Please do your fellow developers a favor: never use the `00363` form. `0` next to capital `O` is just asking for confusion. Always use the lowercase predicates `0x`, `0b`, and `0o`.

Small Decimal Values

The most (in)famous side effect of using binary floating-point numbers (which, remember, is true of **all** languages that use IEEE 754 -- not *just* JavaScript as many assume/pretend) is:

```
0.1 + 0.2 === 0.3; // false
```

Mathematically, we know that statement should be `true`. Why is it `false`?

Simply put, the representations for `0.1` and `0.2` in binary floating-point are not exact, so when they are added, the result is not exactly `0.3`. It's **really** close: `0.30000000000000004`, but if your comparison fails, "close" is irrelevant.

Note: Should JavaScript switch to a different `number` implementation that has exact representations for all values? Some think so. There have been many alternatives presented over the years. None of them have been accepted yet, and perhaps never will. As easy as it may seem to just wave a hand and say, "fix that bug already!", it's not nearly that easy. If it were, it most definitely would have been changed a long time ago.

Now, the question is, if some `number`'s can't be *trusted* to be exact, does that mean we can't use `number`'s at all? **Of course not.**

There are some applications where you need to be more careful, especially when dealing with fractional decimal values. There are also plenty of (maybe most?) applications that only deal with whole numbers ("integers"), and moreover, only deal with numbers in the millions or trillions at maximum. These applications have been, and always will be, **perfectly safe** to use numeric operations in JS.

What if we *did* need to compare two `number`s, like `0.1 + 0.2` to `0.3`, knowing that the simple equality test fails?

The most commonly accepted practice is to use a tiny "rounding error" value as the *tolerance* for comparison. This tiny value is often called "machine epsilon," which is commonly `2^-52` (`2.220446049250313e-16`) for the kind of `number`s in JavaScript.

As of ES6, `Number.EPSILON` is predefined with this tolerance value, so you'd want to use it, but you can safely polyfill the definition for pre-ES6:

```
if (!Number.EPSILON) {
  Number.EPSILON = Math.pow(2, -52);
}
```

We can use this `Number.EPSILON` to compare two `number`s for "equality" (within the rounding error tolerance):

```
function numbersCloseEnoughToEqual(n1, n2) {
  return Math.abs( n1 - n2 ) < Number.EPSILON;
}

var a = 0.1 + 0.2;
var b = 0.3;

numbersCloseEnoughToEqual( a, b );           // true
numbersCloseEnoughToEqual( 0.0000001, 0.0000002 ); // false
```

The maximum floating-point value that can be represented is roughly `1.798e+308` (which is really, really, really huge!), predefined for you as `Number.MAX_VALUE`. On the small end, `Number.MIN_VALUE` is roughly `5e-324`, which isn't negative but is really close to zero!

Safe Integer Ranges

Because of how `number`s are represented, there is a range of "safe" values for the whole `number` "integers", and it's significantly less than `Number.MAX_VALUE`.

The maximum integer that can "safely" be represented (that is, there's a guarantee that the requested value is actually representable unambiguously) is `2^53 - 1`, which is `9007199254740991`. If you insert your commas, you'll see that this is just over 9 quadrillion. So that's pretty darn big for `number`s to range up to.

This value is actually automatically predefined in ES6, as `Number.MAX_SAFE_INTEGER`. Unsurprisingly, there's a minimum value, `-9007199254740991`, and it's defined in ES6 as `Number.MIN_SAFE_INTEGER`.

The main way that JS programs are confronted with dealing with such large numbers is when dealing with 64-bit IDs from databases, etc. 64-bit numbers cannot be represented accurately with the `number` type, so must be stored in (and transmitted to/from) JavaScript using `string` representation.

Numeric operations on such large ID `number` values (besides comparison, which will be fine with `string`s) aren't all that common, thankfully. But if you *do* need to perform math on these very large values, for now you'll need to use a *big number* utility. Big numbers may get official support in a future version of JavaScript.

Testing for Integers

To test if a value is an integer, you can use the ES6-specified `Number.isInteger(..)`:

```
Number.isInteger( 42 );           // true
Number.isInteger( 42.000 );       // true
Number.isInteger( 42.3 );         // false
```

To polyfill `Number.isInteger(..)` for pre-ES6:

```
if (!Number.isInteger) {
    Number.isInteger = function(num) {
        return typeof num == "number" && num % 1 == 0;
    };
}
```

To test if a value is a *safe integer*, use the ES6-specified `Number.isSafeInteger(..)`:

```
Number.isSafeInteger( Number.MAX_SAFE_INTEGER );      // true
Number.isSafeInteger( Math.pow( 2, 53 ) );            // false
Number.isSafeInteger( Math.pow( 2, 53 ) - 1 );         // true
```

To polyfill `Number.isSafeInteger(..)` in pre-ES6 browsers:

```
if (!Number.isSafeInteger) {
    Number.isSafeInteger = function(num) {
        return Number.isInteger( num ) &&
            Math.abs( num ) <= Number.MAX_SAFE_INTEGER;
    };
}
```

32-bit (Signed) Integers

While integers can range up to roughly 9 quadrillion safely (53 bits), there are some numeric operations (like the bitwise operators) that are only defined for 32-bit `number`s, so the "safe range" for `number`s used in that way must be much smaller.

The range then is `Math.pow(-2, 31)` (`-2147483648`, about -2.1 billion) up to `Math.pow(2, 31)-1` (`2147483647`, about +2.1 billion).

To force a `number` value in `a` to a 32-bit signed integer value, use `a | 0`. This works because the `|` bitwise operator only works for 32-bit integer values (meaning it can only pay attention to 32 bits and any other bits will be lost). Then, "or'ing" with zero is essentially a no-op bitwise speaking.

Note: Certain special values (which we will cover in the next section) such as `NaN` and `Infinity` are not "32-bit safe," in that those values when passed to a bitwise operator will pass through the abstract operation `ToInt32` (see Chapter 4) and become simply the `+0` value for the purpose of that bitwise operation.

Special Values

There are several special values spread across the various types that the `alert` JS developer needs to be aware of, and use properly.

The Non-value Values

For the `undefined` type, there is one and only one value: `undefined`. For the `null` type, there is one and only one value: `null`. So for both of them, the label is both its type and its value.

Both `undefined` and `null` are often taken to be interchangeable as either "empty" values or "non" values. Other developers prefer to distinguish between them with nuance. For example:

- `null` is an empty value
- `undefined` is a missing value

Or:

- `undefined` hasn't had a value yet
- `null` had a value and doesn't anymore

Regardless of how you choose to "define" and use these two values, `null` is a special keyword, not an identifier, and thus you cannot treat it as a variable to assign to (why would you!?). However, `undefined` is (unfortunately) an identifier. Uh oh.

Undefined

In non-`strict` mode, it's actually possible (though incredibly ill-advised!) to assign a value to the globally provided `undefined` identifier:

```
function foo() {
    undefined = 2; // really bad idea!
}

foo();
```

```
function foo() {
    "use strict";
    undefined = 2; // TypeError!
}

foo();
```

In both non-`strict` mode and `strict` mode, however, you can create a local variable of the name `undefined`. But again, this is a terrible idea!

```
function foo() {
    "use strict";
    var undefined = 2;
    console.log( undefined ); // 2
}

foo();
```

Friends don't let friends override `undefined`. Ever.

void Operator

While `undefined` is a built-in identifier that holds (unless modified -- see above!) the built-in `undefined` value, another way to get this value is the `void` operator.

The expression `void __` "voids" out any value, so that the result of the expression is always the `undefined` value. It doesn't modify the existing value; it just ensures that no value comes back from the operator expression.

```
var a = 42;

console.log( void a, a ); // undefined 42
```

By convention (mostly from C-language programming), to represent the `undefined` value stand-alone by using `void`, you'd use `void 0` (though clearly even `void true` or any other `void` expression does the same thing). There's no practical difference between `void 0`, `void 1`, and `undefined`.

But the `void` operator can be useful in a few other circumstances, if you need to ensure that an expression has no result value (even if it has side effects).

For example:

```
function doSomething() {
  // note: `APP.ready` is provided by our application
  if (!APP.ready) {
    // try again later
    return void setTimeout( doSomething, 100 );
  }

  var result;

  // do some other stuff
  return result;
}

// were we able to do it right away?
if (doSomething()) {
  // handle next tasks right away
}
```

Here, the `setTimeout(...)` function returns a numeric value (the unique identifier of the timer interval, if you wanted to cancel it), but we want to `void` that out so that the return value of our function doesn't give a false-positive with the `if` statement.

Many devs prefer to just do these actions separately, which works the same but doesn't use the `void` operator:

```
if (!APP.ready) {
  // try again later
  setTimeout( doSomething, 100 );
  return;
}
```

In general, if there's ever a place where a value exists (from some expression) and you'd find it useful for the value to be `undefined` instead, use the `void` operator. That probably won't be terribly common in your programs, but in the rare cases you do need it, it can be quite helpful.

Special Numbers

The `number` type includes several special values. We'll take a look at each in detail.

The Not Number, Number

Any mathematic operation you perform without both operands being `number`s (or values that can be interpreted as regular `number`s in base 10 or base 16) will result in the operation failing to produce a valid `number`, in which case you will get the `NaN` value.

`NaN` literally stands for "not a `number`", though this label/description is very poor and misleading, as we'll see shortly. It would be much more accurate to think of `NaN` as being "invalid number," "failed number," or even "bad number," than to think of it as "not a number."

For example:

```
var a = 2 / "foo";           // NaN

typeof a === "number";      // true
```

In other words: "the type of not-a-number is 'number'!" Hooray for confusing names and semantics.

`NaN` is a kind of "sentinel value" (an otherwise normal value that's assigned a special meaning) that represents a special kind of error condition within the `number` set. The error condition is, in essence: "I tried to perform a mathematic operation but failed, so here's the failed `number` result instead."

So, if you have a value in some variable and want to test to see if it's this special failed-number `NaN`, you might think you could directly compare to `NaN` itself, as you can with any other value, like `null` or `undefined`. Nope.

```
var a = 2 / "foo";

a == NaN;    // false
a === NaN;   // false
```

`NaN` is a very special value in that it's never equal to another `NaN` value (i.e., it's never equal to itself). It's the only value, in fact, that is not reflexive (without the Identity characteristic `x === x`). So, `NaN !== NaN`. A bit strange, huh?

So how do we test for it, if we can't compare to `NaN` (since that comparison would always fail)?

```
var a = 2 / "foo";  
  
isNaN( a ); // true
```

Easy enough, right? We use the built-in global utility called `isNaN(..)` and it tells us if the value is `Nan` or not. Problem solved!

Not so fast.

The `isNaN(..)` utility has a fatal flaw. It appears it tried to take the meaning of `Nan` ("Not a Number") too literally -- that its job is basically: "test if the thing passed in is either not a `number` or is a `number` ." But that's not quite accurate.

```
var a = 2 / "foo";  
var b = "foo";  
  
a; // NaN  
b; // "foo"  
  
window.isNaN( a ); // true  
window.isNaN( b ); // true -- ouch!
```

Clearly, `"foo"` is literally *not a* `number`, but it's definitely not the `Nan` value either! This bug has been in JS since the very beginning (over 19 years of *ouch*).

As of ES6, finally a replacement utility has been provided: `Number.isNaN(..)`. A simple polyfill for it so that you can safely check `Nan` values now even in pre-ES6 browsers is:

```
if (!Number.isNaN) {  
    Number.isNaN = function(n) {  
        return (  
            typeof n === "number" &&  
            window.isNaN( n )  
        );  
    };  
}  
  
var a = 2 / "foo";  
var b = "foo";  
  
Number.isNaN( a ); // true  
Number.isNaN( b ); // false -- phew!
```

Actually, we can implement a `Number.isNaN(..)` polyfill even easier, by taking advantage of that peculiar fact that `Nan` isn't equal to itself. `Nan` is the *only* value in the whole language where that's true; every other value is always **equal to itself**.

So:

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

Weird, huh? But it works!

`Nan`s are probably a reality in a lot of real-world JS programs, either on purpose or by accident. It's a really good idea to use a reliable test, like `Number.isNaN(..)` as provided (or polyfilled), to recognize them properly.

If you're currently using just `isNaN(..)` in a program, the sad reality is your program *has a bug*, even if you haven't been bitten by it yet!

Infinities

Developers from traditional compiled languages like C are probably used to seeing either a compiler error or runtime exception, like "Divide by zero," for an operation like:

```
var a = 1 / 0;
```

However, in JS, this operation is well-defined and results in the value `Infinity` (aka `Number.POSITIVE_INFINITY`). Unsurprisingly:

```
var a = 1 / 0;      // Infinity
var b = -1 / 0;     // -Infinity
```

As you can see, `-Infinity` (aka `Number.NEGATIVE_INFINITY`) results from a divide-by-zero where either (but not both!) of the divide operands is negative.

JS uses finite numeric representations (IEEE 754 floating-point, which we covered earlier), so contrary to pure mathematics, it seems it *is* possible to overflow even with an operation like addition or subtraction, in which case you'd get `Infinity` or `-Infinity`.

For example:

```
var a = Number.MAX_VALUE;      // 1.7976931348623157e+308
a + a;                      // Infinity
a + Math.pow( 2, 970 );       // Infinity
a + Math.pow( 2, 969 );       // 1.7976931348623157e+308
```

According to the specification, if an operation like addition results in a value that's too big to represent, the IEEE 754 "round-to-nearest" mode specifies what the result should be. So, in a crude sense, `Number.MAX_VALUE + Math.pow(2, 969)` is closer to `Number.MAX_VALUE` than to `Infinity`, so it "rounds down," whereas `Number.MAX_VALUE + Math.pow(2, 970)` is closer to `Infinity` so it "rounds up".

If you think too much about that, it's going to make your head hurt. So don't. Seriously, stop!

Once you overflow to either one of the *infinities*, however, there's no going back. In other words, in an almost poetic sense, you can go from finite to infinite but not from infinite back to finite.

It's almost philosophical to ask: "What is infinity divided by infinity". Our naive brains would likely say "1" or maybe "infinity." Turns out neither is true. Both mathematically and in JavaScript, `Infinity / Infinity` is not a defined operation. In JS, this results in `Nan`.

But what about any positive finite `number` divided by `Infinity`? That's easy! `0`. And what about a negative finite `number` divided by `Infinity`? Keep reading!

Zeros

While it may confuse the mathematics-minded reader, JavaScript has both a normal zero `0` (otherwise known as a positive zero `+0`) and a negative zero `-0`. Before we explain why the `-0` exists, we should examine how JS handles it, because it can be quite confusing.

Besides being specified literally as `-0`, negative zero also results from certain mathematical operations. For example:

```
var a = 0 / -3; // -0
var b = 0 * -3; // -0
```

Addition and subtraction cannot result in a negative zero.

A negative zero when examined in the developer console will usually reveal `-0`, though that was not the common case until fairly recently, so some older browsers you encounter may still report it as `0`.

However, if you try to stringify a negative zero value, it will always be reported as `"0"`, according to the spec.

```

var a = 0 / -3;

// (some browser) consoles at least get it right
a;                                // -0

// but the spec insists on lying to you!
a.toString();                      // "0"
a + "";                            // "0"
String( a );                      // "0"

// strangely, even JSON gets in on the deception
JSON.stringify( a );              // "0"

```

Interestingly, the reverse operations (going from `string` to `number`) don't lie:

```

"-0";                           // -0
Number( "-0" );                 // -0
JSON.parse( "-0" );             // -0

```

Warning: The `JSON.stringify(-0)` behavior of `"0"` is particularly strange when you observe that it's inconsistent with the reverse: `JSON.parse("-0")` reports `-0` as you'd correctly expect.

In addition to stringification of negative zero being deceptive to hide its true value, the comparison operators are also (intentionally) configured to *lie*.

```

var a = 0;
var b = 0 / -3;

a == b;                // true
-0 == 0;               // true

a === b;               // true
-0 === 0;               // true

0 > -0;                // false
a > b;                  // false

```

Clearly, if you want to distinguish a `-0` from a `0` in your code, you can't just rely on what the developer console outputs, so you're going to have to be a bit more clever:

```

function isNegZero(n) {
  n = Number( n );
  return (n === 0) && (1 / n === -Infinity);
}

isNegZero( -0 );           // true
isNegZero( 0 / -3 );      // true
isNegZero( 0 );            // false

```

Now, why do we need a negative zero, besides academic trivia?

There are certain applications where developers use the magnitude of a value to represent one piece of information (like speed of movement per animation frame) and the sign of that number to represent another piece of information (like the direction of that movement).

In those applications, as one example, if a variable arrives at zero and it loses its sign, then you would lose the information of what direction it was moving in before it arrived at zero. Preserving the sign of the zero prevents potentially unwanted information loss.

Special Equality

As we saw above, the `NaN` value and the `-0` value have special behavior when it comes to equality comparison. `NaN` is never equal to itself, so you have to use ES6's `Number.isNaN(...)` (or a polyfill). Similarly, `-0` lies and pretends that it's equal (even `==` strict equal -- see Chapter 4) to regular positive `0`, so you have to use the somewhat hackish `isNegZero(...)` utility we suggested above.

As of ES6, there's a new utility that can be used to test two values for absolute equality, without any of these exceptions. It's called `Object.is(...)`:

```

var a = 2 / "foo";
var b = -3 * 0;

Object.is( a, NaN );    // true
Object.is( b, -0 );     // true

Object.is( b, 0 );       // false

```

There's a pretty simple polyfill for `Object.is(...)` for pre-ES6 environments:

```

if (!Object.is) {
  Object.is = function(v1, v2) {
    // test for `0`
    if (v1 === 0 && v2 === 0) {
      return 1 / v1 === 1 / v2;
    }
    // test for `NaN`
    if (v1 !== v1) {
      return v2 !== v2;
    }
    // everything else
    return v1 === v2;
  };
}

```

`Object.is(..)` probably shouldn't be used in cases where `==` or `===` are known to be *safe* (see Chapter 4 "Coercion"), as the operators are likely much more efficient and certainly are more idiomatic/common. `Object.is(..)` is mostly for these special cases of equality.

Value vs. Reference

In many other languages, values can either be assigned/passed by value-copy or by reference-copy depending on the syntax you use.

For example, in C++ if you want to pass a `number` variable into a function and have that variable's value updated, you can declare the function parameter like `int& myNum`, and when you pass in a variable like `x`, `myNum` will be a **reference to** `x`; references are like a special form of pointers, where you obtain a pointer to another variable (like an *alias*). If you don't declare a reference parameter, the value passed in will *always* be copied, even if it's a complex object.

In JavaScript, there are no pointers, and references work a bit differently. You cannot have a reference from one JS variable to another variable. That's just not possible.

A reference in JS points at a (shared) **value**, so if you have 10 different references, they are all always distinct references to a single shared value; **none of them are references/pointers to each other**.

Moreover, in JavaScript, there are no syntactic hints that control value vs. reference assignment/passing. Instead, the *type* of the value *solely* controls whether that value will be assigned by value-copy or by reference-copy.

Let's illustrate:

```

var a = 2;
var b = a; // `b` is always a copy of the value in `a`
b++;
a; // 2
b; // 3

var c = [1,2,3];
var d = c; // `d` is a reference to the shared `[1,2,3]` value
d.push( 4 );
c; // [1,2,3,4]
d; // [1,2,3,4]

```

Simple values (aka scalar primitives) are *always* assigned/passed by value-copy: `null`, `undefined`, `string`, `number`, `boolean`, and ES6's `symbol`.

Compound values -- `object` `s` (including `array` `s`, and all boxed object wrappers -- see Chapter 3) and `function` `s` -- *always* create a copy of the reference on assignment or passing.

In the above snippet, because `2` is a scalar primitive, `a` holds one initial copy of that value, and `b` is assigned another *copy* of the value. When changing `b`, you are in no way changing the value in `a`.

But **both** `c` and `d` are separate references to the same shared value `[1,2,3]`, which is a compound value. It's important to note that neither `c` nor `d` more "owns" the `[1,2,3]` value -- both are just equal peer references to the value. So, when using either reference to modify (`.push(4)`) the actual shared `array` value itself, it's affecting just the one shared value, and both references will reference the newly modified value `[1,2,3,4]`.

Since references point to the values themselves and not to the variables, you cannot use one reference to change where another reference is pointed:

```

var a = [1,2,3];
var b = a;
a; // [1,2,3]
b; // [1,2,3]

// later
b = [4,5,6];
a; // [1,2,3]
b; // [4,5,6]

```

When we make the assignment `b = [4,5,6]`, we are doing absolutely nothing to affect *where* `a` is still referencing (`[1,2,3]`). To do that, `b` would have to be a pointer to `a` rather than a reference to the `array` -- but no such capability exists in JS!

The most common way such confusion happens is with function parameters:

```
function foo(x) {
  x.push( 4 );
  x; // [1,2,3,4]

  // later
  x = [4,5,6];
  x.push( 7 );
  x; // [4,5,6,7]
}

var a = [1,2,3];

foo( a );

a; // [1,2,3,4] not [4,5,6,7]
```

When we pass in the argument `a`, it assigns a copy of the `a` reference to `x`. `x` and `a` are separate references pointing at the same `[1,2,3]` value. Now, inside the function, we can use that reference to mutate the value itself (`push(4)`). But when we make the assignment `x = [4,5,6]`, this is in no way affecting where the initial reference `a` is pointing -- still points at the (now modified) `[1,2,3,4]` value.

There is no way to use the `x` reference to change where `a` is pointing. We could only modify the contents of the shared value that both `a` and `x` are pointing to.

To accomplish changing `a` to have the `[4,5,6,7]` value contents, you can't create a new array and assign -- you must modify the existing array value:

```
function foo(x) {
  x.push( 4 );
  x; // [1,2,3,4]

  // later
  x.length = 0; // empty existing array in-place
  x.push( 4, 5, 6, 7 );
  x; // [4,5,6,7]
}

var a = [1,2,3];

foo( a );

a; // [4,5,6,7] not [1,2,3,4]
```

As you can see, `x.length = 0` and `x.push(4,5,6,7)` were not creating a new `array`, but modifying the existing shared `array`. So of course, `a` references the new `[4,5,6,7]` contents.

Remember: you cannot directly control/override value-copy vs. reference -- those semantics are controlled entirely by the type of the underlying value.

To effectively pass a compound value (like an `array`) by value-copy, you need to manually make a copy of it, so that the reference passed doesn't still point to the original. For example:

```
foo( a.slice() );
```

`slice(..)` with no parameters by default makes an entirely new (shallow) copy of the `array`. So, we pass in a reference only to the copied `array`, and thus `foo(..)` cannot affect the contents of `a`.

To do the reverse -- pass a scalar primitive value in a way where its value updates can be seen, kinda like a reference -- you have to wrap the value in another compound value (`object`, `array`, etc) that *can* be passed by reference-copy:

```
function foo(wrapper) {
  wrapper.a = 42;
}

var obj = {
  a: 2
};

foo( obj );

obj.a; // 42
```

Here, `obj` acts as a wrapper for the scalar primitive property `a`. When passed to `foo(..)`, a copy of the `obj` reference is passed in and set to the `wrapper` parameter. We now can use the `wrapper` reference to access the shared object, and update its property. After the function finishes, `obj.a` will see the updated value `42`.

It may occur to you that if you wanted to pass in a reference to a scalar primitive value like `2`, you could just box the value in its `Number` object wrapper (see Chapter 3).

It *is* true a copy of the reference to this `Number` object *will* be passed to the function, but unfortunately, having a reference to the shared object is not going to give you the ability to modify the shared primitive value, like you may expect:

```

function foo(x) {
  x = x + 1;
  x; // 3
}

var a = 2;
var b = new Number(a); // or equivalently `Object(a)`

foo(b);
console.log(b); // 2, not 3

```

The problem is that the underlying scalar primitive value is *not mutable* (same goes for `String` and `Boolean`). If a `Number` object holds the scalar primitive value `2`, that exact `Number` object can never be changed to hold another value; you can only create a whole new `Number` object with a different value.

When `x` is used in the expression `x + 1`, the underlying scalar primitive value `2` is unboxed (extracted) from the `Number` object automatically, so the line `x = x + 1` very subtly changes `x` from being a shared reference to the `Number` object, to just holding the scalar primitive value `3` as a result of the addition operation `2 + 1`. Therefore, `b` on the outside still references the original unmodified/immutable `Number` object holding the value `2`.

You *can* add properties on top of the `Number` object (just not change its inner primitive value), so you could exchange information indirectly via those additional properties.

This is not all that common, however; it probably would not be considered a good practice by most developers.

Instead of using the wrapper object `Number` in this way, it's probably much better to use the manual object wrapper (`obj`) approach in the earlier snippet. That's not to say that there's no clever uses for the boxed object wrappers like `Number` -- just that you should probably prefer the scalar primitive value form in most cases.

References are quite powerful, but sometimes they get in your way, and sometimes you need them where they don't exist. The only control you have over reference vs. value-copy behavior is the type of the value itself, so you must indirectly influence the assignment/passing behavior by which value types you choose to use.

Review

In JavaScript, `array`s are simply numerically indexed collections of any value-type.

`string`s are somewhat "array-like", but they have distinct behaviors and care must be taken if you want to treat them as `array`s. Numbers in JavaScript include both "integers" and floating-point values.

Several special values are defined within the primitive types.

The `null` type has just one value: `null`, and likewise the `undefined` type has just the `undefined` value. `undefined` is basically the default value in any variable or property if no other value is present. The `void` operator lets you create the `undefined` value from any other value.

`number`s include several special values, like `NaN` (supposedly "Not a Number", but really more appropriately "invalid number"); `+Infinity` and `-Infinity`; and `-0`.

Simple scalar primitives (`string`s, `number`s, etc.) are assigned/passed by value-copy, but compound values (`object`s, etc.) are assigned/passed by reference-copy. References are not like references/pointers in other languages -- they're never pointed at other variables/references, only at the underlying values.

Chapter 3: Natives

Several times in Chapters 1 and 2, we alluded to various built-ins, usually called "natives," like `String` and `Number`. Let's examine those in detail now.

Here's a list of the most commonly used natives:

- `String()`
- `Number()`
- `Boolean()`
- `Array()`
- `Object()`
- `Function()`
- `RegExp()`
- `Date()`
- `Error()`
- `Symbol()` -- added in ES6!

As you can see, these natives are actually built-in functions.

If you're coming to JS from a language like Java, JavaScript's `String()` will look like the `String(..)` constructor you're used to for creating string values. So, you'll quickly observe that you can do things like:

```
var s = new String( "Hello World!" );
console.log( s.toString() ); // "Hello World!"
```

It *is* true that each of these natives can be used as a native constructor. But what's being constructed may be different than you think.

```
var a = new String( "abc" );
typeof a; // "object" ... not "String"
a instanceof String; // true
Object.prototype.toString.call( a ); // "[object String]"
```

The result of the constructor form of value creation (`new String("abc")`) is an object wrapper around the primitive (`"abc"`) value.

Importantly, `typeof` shows that these objects are not their own special *types*, but more appropriately they are subtypes of the `object` type.

This object wrapper can further be observed with:

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

The output of that statement varies depending on your browser, as developer consoles are free to choose however they feel it's appropriate to serialize the object for developer inspection.

Note: At the time of writing, the latest Chrome prints something like this: `String {0: "a", 1: "b", 2: "c", length: 3, [[PrimitiveValue]]: "abc"}`. But older versions of Chrome used to just print this: `String {0: "a", 1: "b", 2: "c"}`. The latest Firefox currently prints `String ["a", "b", "c"]`, but used to print `"abc"` in italics, which was clickable to open the object inspector. Of course, these results are subject to rapid change and your experience may vary.

The point is, `new String("abc")` creates a string wrapper object around `"abc"`, not just the primitive `"abc"` value itself.

Internal `[[Class]]`

Values that are `typeof "object"` (such as an array) are additionally tagged with an internal `[[class]]` property (think of this more as an internal *classification* rather than related to classes from traditional class-oriented coding). This property cannot be accessed directly, but can generally be revealed indirectly by borrowing the default

`Object.prototype.toString()` method called against the value. For example:

```
Object.prototype.toString.call( [1,2,3] );           // "[object Array]"  
Object.prototype.toString.call( /regex-literal/i );    // "[object RegExp]"
```

So, for the array in this example, the internal `[[Class]]` value is `"Array"`, and for the regular expression, it's `"RegExp"`. In most cases, this internal `[[Class]]` value corresponds to the built-in native constructor (see below) that's related to the value, but that's not always the case.

What about primitive values? First, `null` and `undefined`:

```
Object.prototype.toString.call( null );           // "[object Null]"
Object.prototype.toString.call( undefined );     // "[object Undefined]"
```

You'll note that there are no `Null()` or `Undefined()` native constructors, but nevertheless the `"Null"` and `"Undefined"` are the internal `[[Class]]` values exposed.

But for the other simple primitives like `string`, `number`, and `boolean`, another behavior actually kicks in, which is usually called "boxing" (see "Boxing Wrappers" section next):

```
Object.prototype.toString.call( "abc" );      // "[object String]"
Object.prototype.toString.call( 42 );          // "[object Number]"
Object.prototype.toString.call( true );        // "[object Boolean]"
```

In this snippet, each of the simple primitives are automatically boxed by their respective object wrappers, which is why `"String"`, `"Number"`, and `"Boolean"` are revealed as the respective internal `[[Class]]` values.

Note: The behavior of `toString()` and `[[Class]]` as illustrated here has changed a bit from ES5 to ES6, but we cover those details in the *ES6 & Beyond* title of this series.

Boxing Wrappers

These object wrappers serve a very important purpose. Primitive values don't have properties or methods, so to access `.length` or `.toString()` you need an object wrapper around the value. Thankfully, JS will automatically *box* (aka *wrap*) the primitive value to fulfill such accesses.

```
var a = "abc";

a.length; // 3
a.toUpperCase(); // "ABC"
```

So, if you're going to be accessing these properties/methods on your string values regularly, like a `i < a.length` condition in a `for` loop for instance, it might seem to make sense to just have the object form of the value from the start, so the JS engine doesn't need to implicitly create it for you.

But it turns out that's a bad idea. Browsers long ago performance-optimized the common cases like `.length`, which means your program will *actually go slower* if you try to "preoptimize" by directly using the object form (which isn't on the optimized path).

In general, there's basically no reason to use the object form directly. It's better to just let the boxing happen implicitly where necessary. In other words, never do things like `new String("abc")`, `new Number(42)`, etc -- always prefer using the literal primitive values `"abc"` and `42`.

Object Wrapper Gotchas

There are some gotchas with using the object wrappers directly that you should be aware of if you *do* choose to ever use them.

For example, consider `Boolean` wrapped values:

```
var a = new Boolean( false );

if (!a) {
    console.log( "Oops" ); // never runs
}
```

The problem is that you've created an object wrapper around the `false` value, but objects themselves are "truthy" (see Chapter 4), so using the object behaves oppositely to using the underlying `false` value itself, which is quite contrary to normal expectation.

If you want to manually box a primitive value, you can use the `Object(...)` function (no `new` keyword):

```
var a = "abc";
var b = new String( a );
var c = Object( a );

typeof a; // "string"
typeof b; // "object"
typeof c; // "object"

b instanceof String; // true
c instanceof String; // true

Object.prototype.toString.call( b ); // "[object String]"
Object.prototype.toString.call( c ); // "[object String]"
```

Again, using the boxed object wrapper directly (like `b` and `c` above) is usually discouraged, but there may be some rare occasions you'll run into where they may be useful.

Unboxing

If you have an object wrapper and you want to get the underlying primitive value out, you can use the `valueOf()` method:

```
var a = new String( "abc" );
var b = new Number( 42 );
var c = new Boolean( true );

a.valueOf(); // "abc"
b.valueOf(); // 42
c.valueOf(); // true
```

Unboxing can also happen implicitly, when using an object wrapper value in a way that requires the primitive value. This process (coercion) will be covered in more detail in Chapter 4, but briefly:

```
var a = new String( "abc" );
var b = a + ""; // `b` has the unboxed primitive value "abc"

typeof a; // "object"
typeof b; // "string"
```

Natives as Constructors

For `array`, `object`, `function`, and regular-expression values, it's almost universally preferred that you use the literal form for creating the values, but the literal form creates the same sort of object as the constructor form does (that is, there is no nonwrapped value).

Just as we've seen above with the other natives, these constructor forms should generally be avoided, unless you really know you need them, mostly because they introduce exceptions and gotchas that you probably don't really *want* to deal with.

Array(..)

```
var a = new Array( 1, 2, 3 );
a; // [1, 2, 3]

var b = [1, 2, 3];
b; // [1, 2, 3]
```

Note: The `Array(..)` constructor does not require the `new` keyword in front of it. If you omit it, it will behave as if you have used it anyway. So `Array(1,2,3)` is the same outcome as `new Array(1,2,3)`.

The `Array` constructor has a special form where if only one `number` argument is passed, instead of providing that value as *contents* of the array, it's taken as a length to "pre-size the array" (well, sorta).

This is a terrible idea. Firstly, you can trip over that form accidentally, as it's easy to forget.

But more importantly, there's no such thing as actually presizing the array. Instead, what you're creating is an otherwise empty array, but setting the `length` property of the array to the numeric value specified.

An array that has no explicit values in its slots, but has a `length` property that *implies* the slots exist, is a weird exotic type of data structure in JS with some very strange and confusing behavior. The capability to create such a value comes purely from old, deprecated, historical functionalities ("array-like objects" like the `arguments` object).

Note: An array with at least one "empty slot" in it is often called a "sparse array."

It doesn't help matters that this is yet another example where browser developer consoles vary on how they represent such an object, which breeds more confusion.

For example:

```
var a = new Array( 3 );  
  
a.length; // 3  
a;
```

The serialization of `a` in Chrome is (at the time of writing): `[undefined × 3]`. **This is really unfortunate.** It implies that there are three `undefined` values in the slots of this array, when in fact the slots do not exist (so-called "empty slots" -- also a bad name!).

To visualize the difference, try this:

```
var a = new Array( 3 );  
var b = [ undefined, undefined, undefined ];  
var c = [];  
c.length = 3;  
  
a;  
b;  
c;
```

Note: As you can see with `c` in this example, empty slots in an array can happen after creation of the array. Changing the `length` of an array to go beyond its number of actually-defined slot values, you implicitly introduce empty slots. In fact, you could even call `delete b[1]` in the above snippet, and it would introduce an empty slot into the middle of `b`.

For `b` (in Chrome, currently), you'll find `[undefined, undefined, undefined]` as the serialization, as opposed to `[undefined x 3]` for `a` and `c`. Confused? Yeah, so is everyone else.

Worse than that, at the time of writing, Firefox reports `[, , ,]` for `a` and `c`. Did you catch why that's so confusing? Look closely. Three commas implies four slots, not three slots like we'd expect.

What!? Firefox puts an extra `,` on the end of their serialization here because as of ES5, trailing commas in lists (array values, property lists, etc.) are allowed (and thus dropped and ignored). So if you were to type in a `[, , ,]` value into your program or the console, you'd actually get the underlying value that's like `[, ,]` (that is, an array with three empty slots). This choice, while confusing if reading the developer console, is defended as instead making copy-n-paste behavior accurate.

If you're shaking your head or rolling your eyes about now, you're not alone! Shrugs.

Unfortunately, it gets worse. More than just confusing console output, `a` and `b` from the above code snippet actually behave the same in some cases **but differently in others**:

```
a.join( "-" ); // "--"
b.join( "-" ); // "--"

a.map(function(v,i){ return i; }); // [ undefined x 3 ]
b.map(function(v,i){ return i; }); // [ 0, 1, 2 ]
```

Ugh.

The `a.map(...)` call *fails* because the slots don't actually exist, so `map(...)` has nothing to iterate over. `join(...)` works differently. Basically, we can think of it implemented sort of like this:

```

function fakeJoin(arr, connector) {
  var str = "";
  for (var i = 0; i < arr.length; i++) {
    if (i > 0) {
      str += connector;
    }
    if (arr[i] !== undefined) {
      str += arr[i];
    }
  }
  return str;
}

var a = new Array( 3 );
fakeJoin( a, "-" ); // "--"

```

As you can see, `join(..)` works by just *assuming* the slots exist and looping up to the `length` value. Whatever `map(..)` does internally, it (apparently) doesn't make such an assumption, so the result from the strange "empty slots" array is unexpected and likely to cause failure.

So, if you wanted to *actually* create an array of actual `undefined` values (not just "empty slots"), how could you do it (besides manually)?

```

var a = Array.apply( null, { length: 3 } );
a; // [ undefined, undefined, undefined ]

```

Confused? Yeah. Here's roughly how it works.

`apply(..)` is a utility available to all functions, which calls the function it's used with but in a special way.

The first argument is a `this` object binding (covered in the *this & Object Prototypes* title of this series), which we don't care about here, so we set it to `null`. The second argument is supposed to be an array (or something *like* an array -- aka an "array-like object"). The contents of this "array" are "spread" out as arguments to the function in question.

So, `Array.apply(..)` is calling the `Array(..)` function and spreading out the values (of the `{ length: 3 }` object value) as its arguments.

Inside of `apply(..)`, we can envision there's another `for` loop (kinda like `join(..)` from above) that goes from `0` up to, but not including, `length` (`3` in our case).

For each index, it retrieves that key from the object. So if the array-object parameter was named `arr` internally inside of the `apply(..)` function, the property access would effectively be `arr[0]`, `arr[1]`, and `arr[2]`. Of course, none of those properties exist on

the `{ length: 3 }` object value, so all three of those property accesses would return the value `undefined`.

In other words, it ends up calling `Array(...)` basically like this:

`Array(undefined, undefined, undefined)`, which is how we end up with an array filled with `undefined` values, and not just those (crazy) empty slots.

While `Array.apply(null, { length: 3 })` is a strange and verbose way to create an array filled with `undefined` values, it's **vastly** better and more reliable than what you get with the footgun'ish `Array(3)` empty slots.

Bottom line: **never ever, under any circumstances**, should you intentionally create and use these exotic empty-slot arrays. Just don't do it. They're nuts.

Object(..), Function(..), and RegExp(..)

The `Object(..)` / `Function(..)` / `RegExp(..)` constructors are also generally optional (and thus should usually be avoided unless specifically called for):

```
var c = new Object();
c.foo = "bar";
c; // { foo: "bar" }

var d = { foo: "bar" };
d; // { foo: "bar" }

var e = new Function("a", "return a * 2;");
var f = function(a) { return a * 2; };
function g(a) { return a * 2; }

var h = new RegExp("^a*b+", "g");
var i = /^a*b+/g;
```

There's practically no reason to ever use the `new Object()` constructor form, especially since it forces you to add properties one-by-one instead of many at once in the object literal form.

The `Function` constructor is helpful only in the rarest of cases, where you need to dynamically define a function's parameters and/or its function body. **Do not just treat `Function(..)` as an alternate form of `eval(..)`**. You will almost never need to dynamically define a function in this way.

Regular expressions defined in the literal form (`/^a*b+/g`) are strongly preferred, not just for ease of syntax but for performance reasons -- the JS engine precompiles and caches them before code execution. Unlike the other constructor forms we've seen so far, `RegExp(..)`

has some reasonable utility: to dynamically define the pattern for a regular expression.

```
var name = "Kyle";
var namePattern = new RegExp( "\\b(?: " + name + ")+\\b", "ig" );

var matches = someText.match( namePattern );
```

This kind of scenario legitimately occurs in JS programs from time to time, so you'd need to use the `new RegExp("pattern", "flags")` form.

Date(..) and Error(..)

The `Date(..)` and `Error(..)` native constructors are much more useful than the other natives, because there is no literal form for either.

To create a date object value, you must use `new Date()`. The `Date(..)` constructor accepts optional arguments to specify the date/time to use, but if omitted, the current date/time is assumed.

By far the most common reason you construct a date object is to get the current timestamp value (a signed integer number of milliseconds since Jan 1, 1970). You can do this by calling `getTime()` on a date object instance.

But an even easier way is to just call the static helper function defined as of ES5:

`Date.now()`. And to polyfill that for pre-ES5 is pretty easy:

```
if (!Date.now) {
    Date.now = function(){
        return (new Date()).getTime();
    };
}
```

Note: If you call `Date()` without `new`, you'll get back a string representation of the date/time at that moment. The exact form of this representation is not specified in the language spec, though browsers tend to agree on something close to: `"Fri Jul 18 2014 00:31:02 GMT-0500 (CDT)"`.

The `Error(..)` constructor (much like `Array()` above) behaves the same with the `new` keyword present or omitted.

The main reason you'd want to create an error object is that it captures the current execution stack context into the object (in most JS engines, revealed as a read-only `.stack` property once constructed). This stack context includes the function call-stack and the line-number where the error object was created, which makes debugging that error much easier.

You would typically use such an error object with the `throw` operator:

```
function foo(x) {
  if (!x) {
    throw new Error( "x wasn't provided" );
  }
  // ...
}
```

Error object instances generally have at least a `message` property, and sometimes other properties (which you should treat as read-only), like `type`. However, other than inspecting the above-mentioned `stack` property, it's usually best to just call `toString()` on the error object (either explicitly, or implicitly through coercion -- see Chapter 4) to get a friendly-formatted error message.

Tip: Technically, in addition to the general `Error(...)` native, there are several other specific-error-type natives: `Evaluator(...)`, `RangeError(...)`, `ReferenceError(...)`, `SyntaxError(...)`, `TypeError(...)`, and `URIError(...)`. But it's very rare to manually use these specific error natives. They are automatically used if your program actually suffers from a real exception (such as referencing an undeclared variable and getting a `ReferenceError` error).

Symbol(..)

New as of ES6, an additional primitive value type has been added, called "Symbol". Symbols are special "unique" (not strictly guaranteed!) values that can be used as properties on objects with little fear of any collision. They're primarily designed for special built-in behaviors of ES6 constructs, but you can also define your own symbols.

Symbols can be used as property names, but you cannot see or access the actual value of a symbol from your program, nor from the developer console. If you evaluate a symbol in the developer console, what's shown looks like `Symbol(Symbol.create)`, for example.

There are several predefined symbols in ES6, accessed as static properties of the `Symbol` function object, like `Symbol.create`, `Symbol.iterator`, etc. To use them, do something like:

```
obj[Symbol.iterator] = function(){ /*...*/ };
```

To define your own custom symbols, use the `Symbol(..)` native. The `Symbol(..)` native "constructor" is unique in that you're not allowed to use `new` with it, as doing so will throw an error.

```

var mysym = Symbol( "my own symbol" );
mysym;           // Symbol(my own symbol)
mysym.toString(); // "Symbol(my own symbol)"
typeof mysym;    // "symbol"

var a = { };
a[mysym] = "foobar";

Object.getOwnPropertySymbols( a );
// [ Symbol(my own symbol) ]

```

While symbols are not actually private (`Object.getOwnPropertySymbols(..)` reflects on the object and reveals the symbols quite publicly), using them for private or special properties is likely their primary use-case. For most developers, they may take the place of property names with `_` underscore prefixes, which are almost always by convention signals to say, "hey, this is a private/special/internal property, so leave it alone!"

Note: `Symbol`s are *not* `object`s, they are simple scalar primitives.

Native Prototypes

Each of the built-in native constructors has its own `.prototype` object -- `Array.prototype`, `String.prototype`, etc.

These objects contain behavior unique to their particular object subtype.

For example, all string objects, and by extension (via boxing) `string` primitives, have access to default behavior as methods defined on the `String.prototype` object.

Note: By documentation convention, `String.prototype.XYZ` is shortened to `String#XYZ`, and likewise for all the other `.prototype`s.

- `String#indexOf(..)` : find the position in the string of another substring
- `String#charAt(..)` : access the character at a position in the string
- `String#substr(..)`, `String#substring(..)`, and `String#slice(..)` : extract a portion of the string as a new string
- `String#toUpperCase()` and `String#toLowerCase()` : create a new string that's converted to either uppercase or lowercase
- `String#trim()` : create a new string that's stripped of any trailing or leading whitespace

None of the methods modify the string *in place*. Modifications (like case conversion or trimming) create a new value from the existing value.

By virtue of prototype delegation (see the *this & Object Prototypes* title in this series), any string value can access these methods:

```
var a = " abc ";
a.indexOf( "c" ); // 3
a.toUpperCase(); // " ABC "
a.trim(); // "abc"
```

The other constructor prototypes contain behaviors appropriate to their types, such as `Number#toFixed(...)` (stringifying a number with a fixed number of decimal digits) and `Array#concat(...)` (merging arrays). All functions have access to `apply(..)`, `call(..)`, and `bind(..)` because `Function.prototype` defines them.

But, some of the native prototypes aren't *just* plain objects:

```
typeof Function.prototype;           // "function"
Function.prototype();               // it's an empty function!

RegExp.prototype.toString();        // "/(?:)/" -- empty regex
"abc".match( RegExp.prototype );   // [""]
```

A particularly bad idea, you can even modify these native prototypes (not just adding properties as you're probably familiar with):

```
Array.isArray( Array.prototype );    // true
Array.prototype.push( 1, 2, 3 );     // 3
Array.prototype;                   // [1,2,3]

// don't leave it that way, though, or expect weirdness!
// reset the `Array.prototype` to empty
Array.prototype.length = 0;
```

As you can see, `Function.prototype` is a function, `RegExp.prototype` is a regular expression, and `Array.prototype` is an array. Interesting and cool, huh?

Prototypes As Defaults

`Function.prototype` being an empty function, `RegExp.prototype` being an "empty" (e.g., non-matching) regex, and `Array.prototype` being an empty array, make them all nice "default" values to assign to variables if those variables wouldn't already have had a value of the proper type.

For example:

```

function isThisCool(vals,fn,rx) {
  vals = vals || Array.prototype;
  fn = fn || Function.prototype;
  rx = rx || RegExp.prototype;

  return rx.test(
    vals.map( fn ).join( "" )
  );
}

isThisCool();           // true

isThisCool(
  ["a","b","c"],
  function(v){ return v.toUpperCase(); },
  /D/
);
// false

```

Note: As of ES6, we don't need to use the `vals = vals || ..` default value syntax trick (see Chapter 4) anymore, because default values can be set for parameters via native syntax in the function declaration (see Chapter 5).

One minor side-benefit of this approach is that the `.prototype`s are already created and built-in, thus created *only once*. By contrast, using `[]`, `function(){}()`, and `/(?:)/` values themselves for those defaults would (likely, depending on engine implementations) be recreating those values (and probably garbage-collecting them later) for *each call* of `isThisCool(..)`. That could be memory/CPU wasteful.

Also, be very careful not to use `Array.prototype` as a default value **that will subsequently be modified**. In this example, `vals` is used read-only, but if you were to instead make in-place changes to `vals`, you would actually be modifying `Array.prototype` itself, which would lead to the gotchas mentioned earlier!

Note: While we're pointing out these native prototypes and some usefulness, be cautious of relying on them and even more wary of modifying them in any way. See Appendix A "Native Prototypes" for more discussion.

Review

JavaScript provides object wrappers around primitive values, known as natives (`String`, `Number`, `Boolean`, etc). These object wrappers give the values access to behaviors appropriate for each object subtype (`String#trim()` and `Array#concat(..)`).

If you have a simple scalar primitive value like `"abc"` and you access its `length` property or some `String.prototype` method, JS automatically "boxes" the value (wraps it in its respective object wrapper) so that the property/method accesses can be fulfilled.

Chapter 4: Coercion

Now that we much more fully understand JavaScript's types and values, we turn our attention to a very controversial topic: coercion.

As we mentioned in Chapter 1, the debates over whether coercion is a useful feature or a flaw in the design of the language (or somewhere in between!) have raged since day one. If you've read other popular books on JS, you know that the overwhelmingly prevalent message out there is that coercion is magical, evil, confusing, and just downright a bad idea.

In the same overall spirit of this book series, rather than running away from coercion because everyone else does, or because you get bitten by some quirk, I think you should run toward that which you don't understand and seek to *get it* more fully.

Our goal is to fully explore the pros and cons (yes, there *are* pros!) of coercion, so that you can make an informed decision on its appropriateness in your program.

Converting Values

Converting a value from one type to another is often called "type casting," when done explicitly, and "coercion" when done implicitly (forced by the rules of how a value is used).

Note: It may not be obvious, but JavaScript coercions always result in one of the scalar primitive (see Chapter 2) values, like `string`, `number`, or `boolean`. There is no coercion that results in a complex value like `object` or `function`. Chapter 3 covers "boxing," which wraps scalar primitive values in their `object` counterparts, but this is not really coercion in an accurate sense.

Another way these terms are often distinguished is as follows: "type casting" (or "type conversion") occur in statically typed languages at compile time, while "type coercion" is a runtime conversion for dynamically typed languages.

However, in JavaScript, most people refer to all these types of conversions as *coercion*, so the way I prefer to distinguish is to say "implicit coercion" vs. "explicit coercion."

The difference should be obvious: "explicit coercion" is when it is obvious from looking at the code that a type conversion is intentionally occurring, whereas "implicit coercion" is when the type conversion will occur as a less obvious side effect of some other intentional operation.

For example, consider these two approaches to coercion:

```
var a = 42;

var b = a + "";           // implicit coercion

var c = String( a );     // explicit coercion
```

For `b`, the coercion that occurs happens implicitly, because the `+` operator combined with one of the operands being a `string` value (`" "`) will insist on the operation being a `string` concatenation (adding two strings together), which *as a (hidden) side effect* will force the `42` value in `a` to be coerced to its `string` equivalent: `"42"`.

By contrast, the `String(..)` function makes it pretty obvious that it's explicitly taking the value in `a` and coercing it to a `string` representation.

Both approaches accomplish the same effect: `"42"` comes from `42`. But it's the *how* that is at the heart of the heated debates over JavaScript coercion.

Note: Technically, there's some nuanced behavioral difference here beyond the stylistic difference. We cover that in more detail later in the chapter, in the "Implicitly: Strings <--> Numbers" section.

The terms "explicit" and "implicit," or "obvious" and "hidden side effect," are *relative*.

If you know exactly what `a + ""` is doing and you're intentionally doing that to coerce to a `string`, you might feel the operation is sufficiently "explicit." Conversely, if you've never seen the `String(..)` function used for `string` coercion, its behavior might seem hidden enough as to feel "implicit" to you.

But we're having this discussion of "explicit" vs. "implicit" based on the likely opinions of an *average, reasonably informed, but not expert or JS specification devotee* developer. To whatever extent you do or do not find yourself fitting neatly in that bucket, you will need to adjust your perspective on our observations here accordingly.

Just remember: it's often rare that we write our code and are the only ones who ever read it. Even if you're an expert on all the ins and outs of JS, consider how a less experienced teammate of yours will feel when they read your code. Will it be "explicit" or "implicit" to them in the same way it is for you?

Abstract Value Operations

Before we can explore *explicit* vs *implicit* coercion, we need to learn the basic rules that govern how values *become* either a `string`, `number`, or `boolean`. The ES5 spec in section 9 defines several "abstract operations" (fancy spec-speak for "internal-only operation") with

the rules of value conversion. We will specifically pay attention to: `Tostring`, `ToNumber`, and `ToBoolean`, and to a lesser extent, `ToPrimitive`.

Tostring

When any non-`string` value is coerced to a `string` representation, the conversion is handled by the `ToString` abstract operation in section 9.8 of the specification.

Built-in primitive values have natural stringification: `null` becomes `"null"`, `undefined` becomes `"undefined"` and `true` becomes `"true"`. `numbers` are generally expressed in the natural way you'd expect, but as we discussed in Chapter 2, very small or very large `numbers` are represented in exponent form:

```
// multiplying `1.07` by `1000`, seven times over
var a = 1.07 * 1000 * 1000 * 1000 * 1000 * 1000 * 1000;

// seven times three digits => 21 digits
a.toString(); // "1.07e21"
```

For regular objects, unless you specify your own, the default `toString()` (located in `Object.prototype.toString()`) will return the *internal* `[[Class]]` (see Chapter 3), like for instance `"[object Object]"`.

But as shown earlier, if an object has its own `toString()` method on it, and you use that object in a `string`-like way, its `toString()` will automatically be called, and the `string` result of that call will be used instead.

Note: The way an object is coerced to a `string` technically goes through the `ToPrimitive` abstract operation (ES5 spec, section 9.1), but those nuanced details are covered in more detail in the `ToNumber` section later in this chapter, so we will skip over them here.

Arrays have an overridden default `toString()` that stringifies as the (string) concatenation of all its values (each stringified themselves), with `,` in between each value:

```
var a = [1,2,3];

a.toString(); // "1,2,3"
```

Again, `toString()` can either be called explicitly, or it will automatically be called if a non-`string` is used in a `string` context.

JSON Stringification

Another task that seems awfully related to `ToString` is when you use the `JSON.stringify(...)` utility to serialize a value to a JSON-compatible `string` value.

It's important to note that this stringification is not exactly the same thing as coercion. But since it's related to the `ToString` rules above, we'll take a slight diversion to cover JSON stringification behaviors here.

For most simple values, JSON stringification behaves basically the same as `toString()` conversions, except that the serialization result is *always a `string`*:

```
JSON.stringify( 42 );      // "42"
JSON.stringify( "42" );    // ""42"" (a string with a quoted string value in it)
JSON.stringify( null );   // "null"
JSON.stringify( true );   // "true"
```

Any **JSON-safe** value can be stringified by `JSON.stringify(...)`. But what is **JSON-safe**? Any value that can be represented validly in a JSON representation.

It may be easier to consider values that are **not** JSON-safe. Some examples: `undefined`, `function`, `s`, (ES6+) `symbol` `s`, and `object` `s` with circular references (where property references in an object structure create a never-ending cycle through each other). These are all illegal values for a standard JSON structure, mostly because they aren't portable to other languages that consume JSON values.

The `JSON.stringify(...)` utility will automatically omit `undefined`, `function`, and `symbol` values when it comes across them. If such a value is found in an `array`, that value is replaced by `null` (so that the array position information isn't altered). If found as a property of an `object`, that property will simply be excluded.

Consider:

```
JSON.stringify( undefined );                      // undefined
JSON.stringify( function(){ } );                  // undefined

JSON.stringify( [1,undefined,function(){}],4 );    // "[1,null,null,4]"
JSON.stringify( { a:2, b:function(){} } );          // '{"a":2}
```

But if you try to `JSON.stringify(...)` an `object` with circular reference(s) in it, an error will be thrown.

JSON stringification has the special behavior that if an `object` value has a `toJSON()` method defined, this method will be called first to get a value to use for serialization.

If you intend to JSON stringify an object that may contain illegal JSON value(s), or if you just have values in the `object` that aren't appropriate for the serialization, you should define a `toJSON()` method for it that returns a *JSON-safe* version of the `object`.

For example:

```
var o = { };

var a = {
  b: 42,
  c: o,
  d: function(){}
};

// create a circular reference inside `a`
o.e = a;

// would throw an error on the circular reference
// JSON.stringify( a );

// define a custom JSON value serialization
a.toJSON = function() {
  // only include the `b` property for serialization
  return { b: this.b };
};

JSON.stringify( a ); // "{"b":42}"
```

It's a very common misconception that `toJSON()` should return a JSON stringification representation. That's probably incorrect, unless you're wanting to actually stringify the `string` itself (usually not!). `toJSON()` should return the actual regular value (of whatever type) that's appropriate, and `JSON.stringify(..)` itself will handle the stringification.

In other words, `toJSON()` should be interpreted as "to a JSON-safe value suitable for stringification," not "to a JSON string" as many developers mistakenly assume.

Consider:

```

var a = {
  val: [1,2,3],

  // probably correct!
  toJSON: function(){
    return this.val.slice( 1 );
  }
};

var b = {
  val: [1,2,3],

  // probably incorrect!
  toJSON: function(){
    return "[" +
      this.val.slice( 1 ).join() +
    "]";
  }
};

JSON.stringify( a ); // "[2,3]"

JSON.stringify( b ); // ""[2,3]"""

```

In the second call, we stringified the returned `string` rather than the `array` itself, which was probably not what we wanted to do.

While we're talking about `JSON.stringify(..)`, let's discuss some lesser-known functionalities that can still be very useful.

An optional second argument can be passed to `JSON.stringify(..)` that is called *replacer*. This argument can either be an `array` or a `function`. It's used to customize the recursive serialization of an `object` by providing a filtering mechanism for which properties should and should not be included, in a similar way to how `toJSON()` can prepare a value for serialization.

If *replacer* is an `array`, it should be an `array` of `string`s, each of which will specify a property name that is allowed to be included in the serialization of the `object`. If a property exists that isn't in this list, it will be skipped.

If *replacer* is a `function`, it will be called once for the `object` itself, and then once for each property in the `object`, and each time is passed two arguments, *key* and *value*. To skip a *key* in the serialization, return `undefined`. Otherwise, return the *value* provided.

```
var a = {
  b: 42,
  c: "42",
  d: [1,2,3]
};

JSON.stringify( a, ["b", "c"] ); // '{"b":42, "c":"42"}'

JSON.stringify( a, function(k,v){
  if (k !== "c") return v;
} );
// {"b":42, "d":[1,2,3]}
```

Note: In the `function replacer` case, the key argument `k` is `undefined` for the first call (where the `a` object itself is being passed in). The `if` statement **filters out** the property named `"c"`. Stringification is recursive, so the `[1,2,3]` array has each of its values (`1`, `2`, and `3`) passed as `v` to `replacer`, with indexes (`0`, `1`, and `2`) as `k`.

A third optional argument can also be passed to `JSON.stringify(..)`, called `space`, which is used as indentation for prettier human-friendly output. `space` can be a positive integer to indicate how many space characters should be used at each indentation level. Or, `space` can be a `string`, in which case up to the first ten characters of its value will be used for each indentation level.

```

var a = {
  b: 42,
  c: "42",
  d: [1,2,3]
};

JSON.stringify( a, null, 3 );
// "{"
//   "b": 42,
//   "c": "42",
//   "d": [
//     1,
//     2,
//     3
//   ]
// }"

JSON.stringify( a, null, "-----" );
// "{"
// -----"b": 42,
// -----"c": "42",
// -----"d": [
//       1,
//       2,
//       3
//     ]
// }"

```

Remember, `JSON.stringify(..)` is not directly a form of coercion. We covered it here, however, for two reasons that relate its behavior to `ToString` coercion:

1. `string`, `number`, `boolean`, and `null` values all stringify for JSON basically the same as how they coerce to `string` values via the rules of the `ToString` abstract operation.
2. If you pass an `object` value to `JSON.stringify(..)`, and that `object` has a `toJSON()` method on it, `toJSON()` is automatically called to (sort of) "coerce" the value to be `JSON-safe` before stringification.

ToNumber

If any non-`number` value is used in a way that requires it to be a `number`, such as a mathematical operation, the ES5 spec defines the `ToNumber` abstract operation in section 9.3.

For example, `true` becomes `1` and `false` becomes `0`. `undefined` becomes `Nan`, but (curiously) `null` becomes `0`.

`ToNumber` for a `string` value essentially works for the most part like the rules/syntax for numeric literals (see Chapter 3). If it fails, the result is `Nan` (instead of a syntax error as with `number` literals). One example difference is that `0`-prefixed octal numbers are not handled as octals (just as normal base-10 decimals) in this operation, though such octals are valid as `number` literals (see Chapter 2).

Note: The differences between `number` literal grammar and `ToNumber` on a `string` value are subtle and highly nuanced, and thus will not be covered further here. Consult section 9.3.1 of the ES5 spec for more information.

Objects (and arrays) will first be converted to their primitive value equivalent, and the resulting value (if a primitive but not already a `number`) is coerced to a `number` according to the `ToNumber` rules just mentioned.

To convert to this primitive value equivalent, the `ToPrimitive` abstract operation (ES5 spec, section 9.1) will consult the value (using the internal `DefaultValue` operation -- ES5 spec, section 8.12.8) in question to see if it has a `valueOf()` method. If `valueOf()` is available and it returns a primitive value, *that* value is used for the coercion. If not, but `toString()` is available, it will provide the value for the coercion.

If neither operation can provide a primitive value, a `TypeError` is thrown.

As of ES5, you can create such a noncoercible object -- one without `valueOf()` and `toString()` -- if it has a `null` value for its `[[Prototype]]`, typically created with `Object.create(null)`. See the *this & Object Prototypes* title of this series for more information on `[[Prototype]]`s.

Note: We cover how to coerce to `number`s later in this chapter in detail, but for this next code snippet, just assume the `Number(...)` function does so.

Consider:

```

var a = {
    valueOf: function(){
        return "42";
    }
};

var b = {
    toString: function(){
        return "42";
    }
};

var c = [4,2];
c.toString = function(){
    return this.join( "" );      // "42"
};

Number( a );                  // 42
Number( b );                  // 42
Number( c );                  // 42
Number( "" );                 // 0
Number( [] );                 // 0
Number( [ "abc" ] );          // NaN

```

ToBoolean

Next, let's have a little chat about how `boolean`s behave in JS. There's **lots of confusion and misconception** floating out there around this topic, so pay close attention!

First and foremost, JS has actual keywords `true` and `false`, and they behave exactly as you'd expect of `boolean` values. It's a common misconception that the values `1` and `0` are identical to `true / false`. While that may be true in other languages, in JS the `number`s are `number`s and the `boolean`s are `boolean`s. You can coerce `1` to `true` (and vice versa) or `0` to `false` (and vice versa). But they're not the same.

Falsy Values

But that's not the end of the story. We need to discuss how values other than the two `boolean`s behave whenever you coerce to their `boolean` equivalent.

All of JavaScript's values can be divided into two categories:

1. values that will become `false` if coerced to `boolean`
2. everything else (which will obviously become `true`)

I'm not just being facetious. The JS spec defines a specific, narrow list of values that will coerce to `false` when coerced to a `boolean` value.

How do we know what the list of values is? In the ES5 spec, section 9.2 defines a `ToBoolean` abstract operation, which says exactly what happens for all the possible values when you try to coerce them "to boolean."

From that table, we get the following as the so-called "falsy" values list:

- `undefined`
- `null`
- `false`
- `+0`, `-0`, and `Nan`
- `""`

That's it. If a value is on that list, it's a "falsy" value, and it will coerce to `false` if you force a `boolean` coercion on it.

By logical conclusion, if a value is *not* on that list, it must be on *another list*, which we call the "truthy" values list. But JS doesn't really define a "truthy" list per se. It gives some examples, such as saying explicitly that all objects are truthy, but mostly the spec just implies: **anything not explicitly on the falsy list is therefore truthy.**

Falsy Objects

Wait a minute, that section title even sounds contradictory. I literally *just said* the spec calls all objects truthy, right? There should be no such thing as a "falsy object."

What could that possibly even mean?

You might be tempted to think it means an object wrapper (see Chapter 3) around a falsy value (such as `""`, `0` or `false`). But don't fall into that *trap*.

Note: That's a subtle specification joke some of you may get.

Consider:

```
var a = new Boolean( false );
var b = new Number( 0 );
var c = new String( "" );
```

We know all three values here are objects (see Chapter 3) wrapped around obviously falsy values. But do these objects behave as `true` or as `false`? That's easy to answer:

```
var d = Boolean( a && b && c );
d; // true
```

So, all three behave as `true`, as that's the only way `d` could end up as `true`.

Tip: Notice the `Boolean(...)` wrapped around the `a && b && c` expression -- you might wonder why that's there. We'll come back to that later in this chapter, so make a mental note of it. For a sneak-peek (trivia-wise), try for yourself what `d` will be if you just do `d = a && b && c` without the `Boolean(...)` call!

So, if "falsy objects" are **not just objects wrapped around falsy values**, what the heck are they?

The tricky part is that they can show up in your JS program, but they're not actually part of JavaScript itself.

What!?

There are certain cases where browsers have created their own sort of *exotic* values behavior, namely this idea of "falsy objects," on top of regular JS semantics.

A "falsy object" is a value that looks and acts like a normal object (properties, etc.), but when you coerce it to a `boolean`, it coerces to a `false` value.

Why!?

The most well-known case is `document.all`: an array-like (object) provided to your JS program by the DOM (not the JS engine itself), which exposes elements in your page to your JS program. It *used* to behave like a normal object--it would act truthy. But not anymore.

`document.all` itself was never really "standard" and has long since been deprecated/abandoned.

"Can't they just remove it, then?" Sorry, nice try. Wish they could. But there's far too many legacy JS code bases out there that rely on using it.

So, why make it act falsy? Because coercions of `document.all` to `boolean` (like in `if` statements) were almost always used as a means of detecting old, nonstandard IE.

IE has long since come up to standards compliance, and in many cases is pushing the web forward as much or more than any other browser. But all that old `if (document.all) { /* it's IE */ }` code is still out there, and much of it is probably never going away. All this legacy code is still assuming it's running in decade-old IE, which just leads to bad browsing experience for IE users.

So, we can't remove `document.all` completely, but IE doesn't want `if (document.all) { .. }` code to work anymore, so that users in modern IE get new, standards-compliant code logic.

"What should we do?" ***"I've got it! Let's bastardize the JS type system and pretend that `document.all` is falsy!"

Ugh. That sucks. It's a crazy gotcha that most JS developers don't understand. But the alternative (doing nothing about the above no-win problems) sucks *just a little bit more*.

So... that's what we've got: crazy, nonstandard "falsy objects" added to JavaScript by the browsers. Yay!

Truthy Values

Back to the truthy list. What exactly are the truthy values? Remember: **a value is truthy if it's not on the falsy list.**

Consider:

```
var a = "false";
var b = "0";
var c = "";

var d = Boolean( a && b && c );

d;
```

What value do you expect `d` to have here? It's gotta be either `true` or `false`.

It's `true`. Why? Because despite the contents of those `string` values looking like falsy values, the `string` values themselves are all truthy, because `""` is the only `string` value on the falsy list.

What about these?

```
var a = [];           // empty array -- truthy or falsy?
var b = {};          // empty object -- truthy or falsy?
var c = function(){}; // empty function -- truthy or falsy?

var d = Boolean( a && b && c );

d;
```

Yep, you guessed it, `d` is still `true` here. Why? Same reason as before. Despite what it may seem like, `[]`, `{}`, and `function(){}()` are *not* on the falsy list, and thus are truthy values.

In other words, the truthy list is infinitely long. It's impossible to make such a list. You can only make a finite falsy list and consult *it*.

Take five minutes, write the falsy list on a post-it note for your computer monitor, or memorize it if you prefer. Either way, you'll easily be able to construct a virtual truthy list whenever you need it by simply asking if it's on the falsy list or not.

The importance of truthy and falsy is in understanding how a value will behave if you coerce it (either explicitly or implicitly) to a `boolean` value. Now that you have those two lists in mind, we can dive into coercion examples themselves.

Explicit Coercion

Explicit coercion refers to type conversions that are obvious and explicit. There's a wide range of type conversion usage that clearly falls under the *explicit* coercion category for most developers.

The goal here is to identify patterns in our code where we can make it clear and obvious that we're converting a value from one type to another, so as to not leave potholes for future developers to trip into. The more explicit we are, the more likely someone later will be able to read our code and understand without undue effort what our intent was.

It would be hard to find any salient disagreements with *explicit* coercion, as it most closely aligns with how the commonly accepted practice of type conversion works in statically typed languages. As such, we'll take for granted (for now) that *explicit* coercion can be agreed upon to not be evil or controversial. We'll revisit this later, though.

Explicitly: Strings <--> Numbers

We'll start with the simplest and perhaps most common coercion operation: coercing values between `string` and `number` representation.

To coerce between `string` `s` and `number` `s`, we use the built-in `String(...)` and `Number(...)` functions (which we referred to as "native constructors" in Chapter 3), but **very importantly**, we do not use the `new` keyword in front of them. As such, we're not creating object wrappers.

Instead, we're actually *explicitly coercing* between the two types:

```
var a = 42;
var b = String( a );

var c = "3.14";
var d = Number( c );

b; // "42"
d; // 3.14
```

`String(..)` coerces from any other value to a primitive `string` value, using the rules of the `ToString` operation discussed earlier. `Number(..)` coerces from any other value to a primitive `number` value, using the rules of the `ToNumber` operation discussed earlier.

I call this *explicit* coercion because in general, it's pretty obvious to most developers that the end result of these operations is the applicable type conversion.

In fact, this usage actually looks a lot like it does in some other statically typed languages.

For example, in C/C++, you can say either `(int)x` or `int(x)`, and both will convert the value in `x` to an integer. Both forms are valid, but many prefer the latter, which kinda looks like a function call. In JavaScript, when you say `Number(x)`, it looks awfully similar. Does it matter that it's *actually* a function call in JS? Not really.

Besides `String(..)` and `Number(..)`, there are other ways to "explicitly" convert these values between `string` and `number`:

```
var a = 42;
var b = a.toString();

var c = "3.14";
var d = +c;

b; // "42"
d; // 3.14
```

Calling `a.toString()` is ostensibly explicit (pretty clear that "toString" means "to a string"), but there's some hidden implicitness here. `toString()` cannot be called on a *primitive* value like `42`. So JS automatically "boxes" (see Chapter 3) `42` in an object wrapper, so that `toString()` can be called against the object. In other words, you might call it "explicitly implicit."

`+c` here is showing the *unary operator* form (operator with only one operand) of the `+` operator. Instead of performing mathematic addition (or string concatenation -- see below), the unary `+` explicitly coerces its operand (`c`) to a `number` value.

Is `+c` *explicit* coercion? Depends on your experience and perspective. If you know (which you do, now!) that unary `+` is explicitly intended for `number` coercion, then it's pretty explicit and obvious. However, if you've never seen it before, it can seem awfully confusing, implicit, with hidden side effects, etc.

Note: The generally accepted perspective in the open-source JS community is that unary `+` is an accepted form of *explicit* coercion.

Even if you really like the `+c` form, there are definitely places where it can look awfully confusing. Consider:

```
var c = "3.14";
var d = 5 + +c;

d; // 8.14
```

The unary `-` operator also coerces like `+` does, but it also flips the sign of the number. However, you cannot put two `--` next to each other to unflip the sign, as that's parsed as the decrement operator. Instead, you would need to do: `--"3.14"` with a space in between, and that would result in coercion to `3.14`.

You can probably dream up all sorts of hideous combinations of binary operators (like `+` for addition) next to the unary form of an operator. Here's another crazy example:

```
1 + - + + - + 1; // 2
```

You should strongly consider avoiding unary `+` (or `-`) coercion when it's immediately adjacent to other operators. While the above works, it would almost universally be considered a bad idea. Even `d = +c` (or `d += c` for that matter!) can far too easily be confused for `d += c`, which is entirely different!

Note: Another extremely confusing place for unary `+` to be used adjacent to another operator would be the `++` increment operator and `--` decrement operator. For example: `a +++b`, `a + ++b`, and `a + + +b`. See "Expression Side-Effects" in Chapter 5 for more about `++`.

Remember, we're trying to be explicit and **reduce** confusion, not make it much worse!

Date To number

Another common usage of the unary `+` operator is to coerce a `Date` object into a `number`, because the result is the unix timestamp (milliseconds elapsed since 1 January 1970 00:00:00 UTC) representation of the date/time value:

```
var d = new Date( "Mon, 18 Aug 2014 08:53:06 CDT" );
+d; // 1408369986000
```

The most common usage of this idiom is to get the current *now* moment as a timestamp, such as:

```
var timestamp = +new Date();
```

Note: Some developers are aware of a peculiar syntactic "trick" in JavaScript, which is that the `()` set on a constructor call (a function called with `new`) is *optional* if there are no arguments to pass. So you may run across the `var timestamp = +new Date;` form. However, not all developers agree that omitting the `()` improves readability, as it's an uncommon syntax exception that only applies to the `new fn()` call form and not the regular `fn()` call form.

But coercion is not the only way to get the timestamp out of a `Date` object. A noncoercion approach is perhaps even preferable, as it's even more explicit:

```
var timestamp = new Date().getTime();
// var timestamp = (new Date()).getTime();
// var timestamp = (new Date).getTime();
```

But an *even more* preferable noncoercion option is to use the ES5 added `Date.now()` static function:

```
var timestamp = Date.now();
```

And if you want to polyfill `Date.now()` into older browsers, it's pretty simple:

```
if (!Date.now) {
  Date.now = function() {
    return +new Date();
  };
}
```

I'd recommend skipping the coercion forms related to dates. Use `Date.now()` for current *now* timestamps, and `new Date(...).getTime()` for getting a timestamp of a specific *non-now* date/time that you need to specify.

The Curious Case of the `~`

One coercive JS operator that is often overlooked and usually very confused is the tilde `~` operator (aka "bitwise NOT"). Many of those who even understand what it does will often times still want to avoid it. But sticking to the spirit of our approach in this book and series, let's dig into it to find out if `~` has anything useful to give us.

In the "32-bit (Signed) Integers" section of Chapter 2, we covered how bitwise operators in JS are defined only for 32-bit operations, which means they force their operands to conform to 32-bit value representations. The rules for how this happens are controlled by the `ToInt32` abstract operation (ES5 spec, section 9.5).

`ToInt32` first does a `ToNumber` coercion, which means if the value is `"123"`, it's going to first become `123` before the `ToInt32` rules are applied.

While not *technically* coercion itself (since the type doesn't change!), using bitwise operators (like `|` or `~`) with certain special `number` values produces a coercive effect that results in a different `number` value.

For example, let's first consider the `|` "bitwise OR" operator used in the otherwise no-op idiom `0 | x`, which (as Chapter 2 showed) essentially only does the `ToInt32` conversion:

```
0 | -0;           // 0
0 | NaN;          // 0
0 | Infinity;    // 0
0 | -Infinity;   // 0
```

These special numbers aren't 32-bit representable (since they come from the 64-bit IEEE 754 standard -- see Chapter 2), so `ToInt32` just specifies `0` as the result from these values.

It's debatable if `0 | __` is an *explicit* form of this coercive `ToInt32` operation or if it's more *implicit*. From the spec perspective, it's unquestionably *explicit*, but if you don't understand bitwise operations at this level, it can seem a bit more *implicitly* magical. Nevertheless, consistent with other assertions in this chapter, we will call it *explicit*.

So, let's turn our attention back to `~`. The `~` operator first "coerces" to a 32-bit `number` value, and then performs a bitwise negation (flipping each bit's parity).

Note: This is very similar to how `!` not only coerces its value to `boolean` but also flips its parity (see discussion of the "unary `!`" later).

But... what!? Why do we care about bits being flipped? That's some pretty specialized, nuanced stuff. It's pretty rare for JS developers to need to reason about individual bits.

Another way of thinking about the definition of `~` comes from old-school computer science/discrete Mathematics: `~` performs two's-complement. Great, thanks, that's totally clearer!

Let's try again: `~x` is roughly the same as `-(x+1)`. That's weird, but slightly easier to reason about. So:

```
~42; // -(42+1) ==> -43
```

You're probably still wondering what the heck all this `~` stuff is about, or why it really matters for a coercion discussion. Let's quickly get to the point.

Consider `-(x+1)`. What's the only value that you can perform that operation on that will produce a `0` (or `-0` technically!) result? `-1`. In other words, `~` used with a range of `number` values will produce a falsy (easily coercible to `false`) `0` value for the `-1` input value, and any other truthy `number` otherwise.

Why is that relevant?

`-1` is commonly called a "sentinel value," which basically means a value that's given an arbitrary semantic meaning within the greater set of values of its same type (`number`s). The C-language uses `-1` sentinel values for many functions that return `>= 0` values for "success" and `-1` for "failure."

JavaScript adopted this precedent when defining the `string` operation `indexof(..)`, which searches for a substring and if found returns its zero-based index position, or `-1` if not found.

It's pretty common to try to use `indexof(..)` not just as an operation to get the position, but as a `boolean` check of presence/absence of a substring in another `string`. Here's how developers usually perform such checks:

```

var a = "Hello World";

if (a.indexOf( "lo" ) >= 0) {      // true
    // found it!
}
if (a.indexOf( "lo" ) != -1) {      // true
    // found it
}

if (a.indexOf( "ol" ) < 0) {      // true
    // not found!
}
if (a.indexOf( "ol" ) == -1) {      // true
    // not found!
}

```

I find it kind of gross to look at `>= 0` or `== -1`. It's basically a "leaky abstraction," in that it's leaking underlying implementation behavior -- the usage of sentinel `-1` for "failure" -- into my code. I would prefer to hide such a detail.

And now, finally, we see why `~` could help us! Using `~` with `indexOf()` "coerces" (actually just transforms) the value **to be appropriately boolean -coercible**:

```

var a = "Hello World";

~a.indexOf( "lo" );           // -4    <- truthy!

if (~a.indexOf( "lo" )) {     // true
    // found it!
}

~a.indexOf( "ol" );           // 0    <- falsy!
!~a.indexOf( "ol" );          // true

if (!~a.indexOf( "ol" )) {    // true
    // not found!
}

```

`~` takes the return value of `indexOf(..)` and transforms it: for the "failure" `-1` we get the falsy `0`, and every other value is truthy.

Note: The `-(x+1)` pseudo-algorithm for `~` would imply that `~-1` is `-0`, but actually it produces `0` because the underlying operation is actually bitwise, not mathematic.

Technically, `if (~a.indexOf(..))` is still relying on *implicit* coercion of its resultant `0` to `false` or nonzero to `true`. But overall, `~` still feels to me more like an *explicit* coercion mechanism, as long as you know what it's intended to do in this idiom.

I find this to be cleaner code than the previous `>= 0 / == -1` clutter.

Truncating Bits

There's one more place `~` may show up in code you run across: some developers use the double tilde `~~` to truncate the decimal part of a `number` (i.e., "coerce" it to a whole number "integer"). It's commonly (though mistakenly) said this is the same result as calling

```
Math.floor(...).
```

How `~~` works is that the first `~` applies the `ToI32` "coercion" and does the bitwise flip, and then the second `~` does another bitwise flip, flipping all the bits back to the original state. The end result is just the `ToI32` "coercion" (aka truncation).

Note: The bitwise double-flip of `~~` is very similar to the parity double-negate `!!` behavior, explained in the "Explicitly: * --> Boolean" section later.

However, `~~` needs some caution/clarification. First, it only works reliably on 32-bit values. But more importantly, it doesn't work the same on negative numbers as `Math.floor(...)` does!

```
Math.floor( -49.6 );      // -50
~~-49.6;                  // -49
```

Setting the `Math.floor(...)` difference aside, `~~x` can truncate to a (32-bit) integer. But so does `x | 0`, and seemingly with (slightly) *less effort*.

So, why might you choose `~~x` over `x | 0`, then? Operator precedence (see Chapter 5):

```
~~1E20 / 10;           // 166199296
1E20 | 0 / 10;         // 1661992960
(1E20 | 0) / 10;       // 166199296
```

Just as with all other advice here, use `~` and `~~` as explicit mechanisms for "coercion" and value transformation only if everyone who reads/writes such code is properly aware of how these operators work!

Explicitly: Parsing Numeric Strings

A similar outcome to coercing a `string` to a `number` can be achieved by parsing a `number` out of a `string`'s character contents. There are, however, distinct differences between this parsing and the type conversion we examined above.

Consider:

```

var a = "42";
var b = "42px";

Number( a );    // 42
parseInt( a );  // 42

Number( b );    // NaN
parseInt( b );  // 42

```

Parsing a numeric value out of a string is *tolerant* of non-numeric characters -- it just stops parsing left-to-right when encountered -- whereas coercion is *not tolerant* and fails resulting in the `Nan` value.

Parsing should not be seen as a substitute for coercion. These two tasks, while similar, have different purposes. Parse a `string` as a `number` when you don't know/care what other non-numeric characters there may be on the right-hand side. Coerce a `string` (to a `number`) when the only acceptable values are numeric and something like `"42px"` should be rejected as a `number`.

Tip: `parseInt(...)` has a twin, `parseFloat(...)`, which (as it sounds) pulls out a floating-point number from a string.

Don't forget that `parseInt(...)` operates on `string` values. It makes absolutely no sense to pass a `number` value to `parseInt(...)`. Nor would it make sense to pass any other type of value, like `true`, `function(){...}` or `[1,2,3]`.

If you pass a non-`string`, the value you pass will automatically be coerced to a `string` first (see "`ToString`" earlier), which would clearly be a kind of hidden *implicit* coercion. It's a really bad idea to rely upon such a behavior in your program, so never use `parseInt(...)` with a non-`string` value.

Prior to ES5, another gotcha existed with `parseInt(...)`, which was the source of many JS programs' bugs. If you didn't pass a second argument to indicate which numeric base (aka radix) to use for interpreting the numeric `string` contents, `parseInt(...)` would look at the beginning character(s) to make a guess.

If the first two characters were `"0x"` or `"0X"`, the guess (by convention) was that you wanted to interpret the `string` as a hexadecimal (base-16) `number`. Otherwise, if the first character was `"0"`, the guess (again, by convention) was that you wanted to interpret the `string` as an octal (base-8) `number`.

Hexadecimal `string`s (with the leading `0x` or `0X`) aren't terribly easy to get mixed up. But the octal number guessing proved devilishly common. For example:

```
var hour = parseInt( selectedHour.value );
var minute = parseInt( selectedMinute.value );

console.log( "The time you selected was: " + hour + ":" + minute);
```

Seems harmless, right? Try selecting `08` for the hour and `09` for the minute. You'll get `0:0`. Why? because neither `8` nor `9` are valid characters in octal base-8.

The pre-ES5 fix was simple, but so easy to forget: **always pass `10` as the second argument**. This was totally safe:

```
var hour = parseInt( selectedHour.value, 10 );
var minute = parseInt( selectedMinute.value, 10 );
```

As of ES5, `parseInt(...)` no longer guesses octal. Unless you say otherwise, it assumes base-10 (or base-16 for `"0x"` prefixes). That's much nicer. Just be careful if your code has to run in pre-ES5 environments, in which case you still need to pass `10` for the radix.

Parsing Non-Strings

One somewhat infamous example of `parseInt(...)`'s behavior is highlighted in a sarcastic joke post a few years ago, poking fun at this JS behavior:

```
parseInt( 1/0, 19 ); // 18
```

The assumptive (but totally invalid) assertion was, "If I pass in `Infinity`, and parse an integer out of that, I should get `Infinity` back, not 18." Surely, JS must be crazy for this outcome, right?

Though this example is obviously contrived and unreal, let's indulge the madness for a moment and examine whether JS really is that crazy.

First off, the most obvious sin committed here is to pass a non-`string` to `parseInt(...)`. That's a no-no. Do it and you're asking for trouble. But even if you do, JS politely coerces what you pass in into a `string` that it can try to parse.

Some would argue that this is unreasonable behavior, and that `parseInt(...)` should refuse to operate on a non-`string` value. Should it perhaps throw an error? That would be very Java-like, frankly. I shudder at thinking JS should start throwing errors all over the place so that `try..catch` is needed around almost every line.

Should it return `Nan`? Maybe. But... what about:

```
parseInt( new String( "42" ) );
```

Should that fail, too? It's a non-`string` value. If you want that `String` object wrapper to be unboxed to `"42"`, then is it really so unusual for `42` to first become `"42"` so that `42` can be parsed back out?

I would argue that this half-explicit, half-implicit coercion that can occur can often be a very helpful thing. For example:

```
var a = {
    num: 21,
    toString: function() { return String( this.num * 2 ); }
};

parseInt( a ); // 42
```

The fact that `parseInt(..)` forcibly coerces its value to a `string` to perform the parse on is quite sensible. If you pass in garbage, and you get garbage back out, don't blame the trash can -- it just did its job faithfully.

So, if you pass in a value like `Infinity` (the result of `1 / 0` obviously), what sort of `string` representation would make the most sense for its coercion? Only two reasonable choices come to mind: `"Infinity"` and `"∞"`. JS chose `"Infinity"`. I'm glad it did.

I think it's a good thing that **all values** in JS have some sort of default `string` representation, so that they aren't mysterious black boxes that we can't debug and reason about.

Now, what about base-19? Obviously, completely bogus and contrived. No real JS programs use base-19. It's absurd. But again, let's indulge the ridiculousness. In base-19, the valid numeric characters are `0 - 9` and `a - i` (case insensitive).

So, back to our `parseInt(1/0, 19)` example. It's essentially `parseInt("Infinity", 19)`. How does it parse? The first character is `"I"`, which is value `18` in the silly base-19. The second character `"n"` is not in the valid set of numeric characters, and as such the parsing simply politely stops, just like when it ran across `"p"` in `"42px"`.

The result? `18`. Exactly like it sensibly should be. The behaviors involved to get us there, and not to an error or to `Infinity` itself, are **very important** to JS, and should not be so easily discarded.

Other examples of this behavior with `parseInt(..)` that may be surprising but are quite sensible include:

```

parseInt( 0.000008 );           // 0   ("0" from "0.000008")
parseInt( 0.0000008 );         // 8   ("8" from "8e-7")
parseInt( false, 16 );          // 250 ("fa" from "false")
parseInt( parseInt, 16 );        // 15 ("f" from "function..")

parseInt( "0x10" );            // 16
parseInt( "103", 2 );          // 2

```

`parseInt(..)` is actually pretty predictable and consistent in its behavior. If you use it correctly, you'll get sensible results. If you use it incorrectly, the crazy results you get are not the fault of JavaScript.

Explicitly: * --> Boolean

Now, let's examine coercing from any non-`boolean` value to a `boolean`.

Just like with `String(..)` and `Number(..)` above, `Boolean(..)` (without the `new`, of course!) is an explicit way of forcing the `ToBoolean` coercion:

```

var a = "0";
var b = [];
var c = {};

var d = "";
var e = 0;
var f = null;
var g;

Boolean( a ); // true
Boolean( b ); // true
Boolean( c ); // true

Boolean( d ); // false
Boolean( e ); // false
Boolean( f ); // false
Boolean( g ); // false

```

While `Boolean(..)` is clearly explicit, it's not at all common or idiomatic.

Just like the unary `+` operator coerces a value to a `number` (see above), the unary `!` negate operator explicitly coerces a value to a `boolean`. The *problem* is that it also flips the value from truthy to falsy or vice versa. So, the most common way JS developers explicitly coerce to `boolean` is to use the `!!` double-negate operator, because the second `!` will flip the parity back to the original:

```

var a = "0";
var b = [];
var c = {};

var d = "";
var e = 0;
var f = null;
var g;

!!a;    // true
!!b;    // true
!!c;    // true

!!d;    // false
!!e;    // false
!!f;    // false
!!g;    // false

```

Any of these `ToBoolean` coercions would happen *implicitly* without the `Boolean(..)` or `!!`, if used in a `boolean` context such as an `if(..) ..` statement. But the goal here is to explicitly force the value to a `boolean` to make it clearer that the `ToBoolean` coercion is intended.

Another example use-case for explicit `ToBoolean` coercion is if you want to force a `true / false` value coercion in the JSON serialization of a data structure:

```

var a = [
  1,
  function(){ /*..*/ },
  2,
  function(){ /*..*/ }
];

JSON.stringify( a ); // "[1,null,2,null]"

JSON.stringify( a, function(key,val){
  if (typeof val == "function") {
    // force `ToBoolean` coercion of the function
    return !!val;
  }
  else {
    return val;
  }
} );
// "[1,true,2,true]"

```

If you come to JavaScript from Java, you may recognize this idiom:

```
var a = 42;  
  
var b = a ? true : false;
```

The `? :` ternary operator will test `a` for truthiness, and based on that test will either assign `true` or `false` to `b`, accordingly.

On its surface, this idiom looks like a form of *explicit* `ToBoolean`-type coercion, since it's obvious that only either `true` or `false` come out of the operation.

However, there's a hidden *implicit* coercion, in that the `a` expression has to first be coerced to `boolean` to perform the truthiness test. I'd call this idiom "explicitly implicit." Furthermore, I'd suggest **you should avoid this idiom completely** in JavaScript. It offers no real benefit, and worse, masquerades as something it's not.

`Boolean(a)` and `!!a` are far better as *explicit* coercion options.

Implicit Coercion

Implicit coercion refers to type conversions that are hidden, with non-obvious side-effects that implicitly occur from other actions. In other words, *implicit coercions* are any type conversions that aren't obvious (to you).

While it's clear what the goal of *explicit* coercion is (making code explicit and more understandable), it might be *too* obvious that *implicit* coercion has the opposite goal: making code harder to understand.

Taken at face value, I believe that's where much of the ire towards coercion comes from. The majority of complaints about "JavaScript coercion" are actually aimed (whether they realize it or not) at *implicit* coercion.

Note: Douglas Crockford, author of "*JavaScript: The Good Parts*", has claimed in many conference talks and writings that JavaScript coercion should be avoided. But what he seems to mean is that *implicit* coercion is bad (in his opinion). However, if you read his own code, you'll find plenty of examples of coercion, both *implicit* and *explicit*! In truth, his angst seems to primarily be directed at the `==` operation, but as you'll see in this chapter, that's only part of the coercion mechanism.

So, is **implicit coercion** evil? Is it dangerous? Is it a flaw in JavaScript's design? Should we avoid it at all costs?

I bet most of you readers are inclined to enthusiastically cheer, "Yes!"

Not so fast. Hear me out.

Let's take a different perspective on what *implicit* coercion is, and can be, than just that it's "the opposite of the good explicit kind of coercion." That's far too narrow and misses an important nuance.

Let's define the goal of *implicit* coercion as: to reduce verbosity, boilerplate, and/or unnecessary implementation detail that clutters up our code with noise that distracts from the more important intent.

Simplifying Implicitly

Before we even get to JavaScript, let me suggest something pseudo-code-ish from some theoretical strongly typed language to illustrate:

```
SomeType x = SomeType( AnotherType( y ) )
```

In this example, I have some arbitrary type of value in `y` that I want to convert to the `SomeType` type. The problem is, this language can't go directly from whatever `y` currently is to `SomeType`. It needs an intermediate step, where it first converts to `AnotherType`, and then from `AnotherType` to `SomeType`.

Now, what if that language (or definition you could create yourself with the language) *did* just let you say:

```
SomeType x = SomeType( y )
```

Wouldn't you generally agree that we simplified the type conversion here to reduce the unnecessary "noise" of the intermediate conversion step? I mean, is it *really* all that important, right here at this point in the code, to see and deal with the fact that `y` goes to `AnotherType` first before then going to `SomeType`?

Some would argue, at least in some circumstances, yes. But I think an equal argument can be made of many other circumstances that here, the simplification **actually aids in the readability of the code** by abstracting or hiding away such details, either in the language itself or in our own abstractions.

Undoubtedly, behind the scenes, somewhere, the intermediate conversion step is still happening. But if that detail is hidden from view here, we can just reason about getting `y` to type `SomeType` as a generic operation and hide the messy details.

While not a perfect analogy, what I'm going to argue throughout the rest of this chapter is that JS *implicit* coercion can be thought of as providing a similar aid to your code.

But, **and this is very important**, that is not an unbounded, absolute statement. There are definitely plenty of evils lurking around *implicit* coercion, that will harm your code much more than any potential readability improvements. Clearly, we have to learn how to avoid such constructs so we don't poison our code with all manner of bugs.

Many developers believe that if a mechanism can do some useful thing **A** but can also be abused or misused to do some awful thing **Z**, then we should throw out that mechanism altogether, just to be safe.

My encouragement to you is: don't settle for that. Don't "throw the baby out with the bathwater." Don't assume *implicit* coercion is all bad because all you think you've ever seen is its "bad parts." I think there are "good parts" here, and I want to help and inspire more of you to find and embrace them!

Implicitly: Strings <--> Numbers

Earlier in this chapter, we explored *explicitly* coercing between `string` and `number` values. Now, let's explore the same task but with *implicit* coercion approaches. But before we do, we have to examine some nuances of operations that will *implicitly* force coercion.

The `+` operator is overloaded to serve the purposes of both `number` addition and `string` concatenation. So how does JS know which type of operation you want to use? Consider:

```
var a = "42";
var b = "0";

var c = 42;
var d = 0;

a + b; // "420"
c + d; // 42
```

What's different that causes `"420"` vs `42`? It's a common misconception that the difference is whether one or both of the operands is a `string`, as that means `+` will assume `string` concatenation. While that's partially true, it's more complicated than that.

Consider:

```
var a = [1,2];
var b = [3,4];

a + b; // "1,23,4"
```

Neither of these operands is a `string`, but clearly they were both coerced to `string`s and then the `string` concatenation kicked in. So what's really going on?

(**Warning:** deeply nitty gritty spec-speak coming, so skip the next two paragraphs if that intimidates you!)

According to ES5 spec section 11.6.1, the `+` algorithm (when an `object` value is an operand) will concatenate if either operand is either already a `string`, or if the following steps produce a `string` representation. So, when `+` receives an `object` (including `array`) for either operand, it first calls the `ToPrimitive` abstract operation (section 9.1) on the value, which then calls the `[[DefaultValue]]` algorithm (section 8.12.8) with a context hint of `number`.

If you're paying close attention, you'll notice that this operation is now identical to how the `ToNumber` abstract operation handles `object`s (see the "`ToNumber`" section earlier). The `valueOf()` operation on the `array` will fail to produce a simple primitive, so it then falls to a `toString()` representation. The two `array`s thus become `"1,2"` and `"3,4"`, respectively. Now, `+` concatenates the two `string`s as you'd normally expect: `"1,23,4"`.

Let's set aside those messy details and go back to an earlier, simplified explanation: if either operand to `+` is a `string` (or becomes one with the above steps!), the operation will be `string` concatenation. Otherwise, it's always numeric addition.

Note: A commonly cited coercion gotcha is `[] + {}` vs. `{ } + []`, as those two expressions result, respectively, in `"[object Object]"` and `0`. There's more to it, though, and we cover those details in "Blocks" in Chapter 5.

What's that mean for *implicit* coercion?

You can coerce a `number` to a `string` simply by "adding" the `number` and the `""` empty `string`:

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

Tip: Numeric addition with the `+` operator is commutative, which means `2 + 3` is the same as `3 + 2`. String concatenation with `+` is obviously not generally commutative, but with the specific case of `""`, it's effectively commutative, as `a + ""` and `"" + a` will produce the same result.

It's extremely common/idiomatic to (*implicitly*) coerce `number` to `string` with a `+ ""` operation. In fact, interestingly, even some of the most vocal critics of *implicit* coercion still use that approach in their own code, instead of one of its *explicit* alternatives.

I think this is a great example of a useful form in *implicit* coercion, despite how frequently the mechanism gets criticized!

Comparing this *implicit* coercion of `a + ""` to our earlier example of `String(a)` *explicit* coercion, there's one additional quirk to be aware of. Because of how the `ToPrimitive` abstract operation works, `a + ""` invokes `valueOf()` on the `a` value, whose return value is then finally converted to a `string` via the internal `ToString` abstract operation. But `String(a)` just invokes `toString()` directly.

Both approaches ultimately result in a `string`, but if you're using an `object` instead of a regular primitive `number` value, you may not necessarily get the *same* `string` value!

Consider:

```
var a = {
  valueOf: function() { return 42; },
  toString: function() { return 4; }
};

a + "";           // "42"

String(a);      // "4"
```

Generally, this sort of gotcha won't bite you unless you're really trying to create confusing data structures and operations, but you should be careful if you're defining both your own `valueOf()` and `toString()` methods for some `object`, as how you coerce the value could affect the outcome.

What about the other direction? How can we *implicitly coerce* from `string` to `number`?

```
var a = "3.14";
var b = a - 0;

b; // 3.14
```

The `-` operator is defined only for numeric subtraction, so `a - 0` forces `a`'s value to be coerced to a `number`. While far less common, `a * 1` or `a / 1` would accomplish the same result, as those operators are also only defined for numeric operations.

What about `object` values with the `-` operator? Similar story as for `+` above:

```
var a = [3];
var b = [1];

a - b; // 2
```

Both `array` values have to become `number`s, but they end up first being coerced to `strings` (using the expected `toString()` serialization), and then are coerced to `number`s, for the `-` subtraction to perform on.

So, is *implicit* coercion of `string` and `number` values the ugly evil you've always heard horror stories about? I don't personally think so.

Compare `b = String(a)` (*explicit*) to `b = a + ""` (*implicit*). I think cases can be made for both approaches being useful in your code. Certainly `b = a + ""` is quite a bit more common in JS programs, proving its own utility regardless of *feelings* about the merits or hazards of *implicit* coercion in general.

Implicitly: Booleans --> Numbers

I think a case where *implicit* coercion can really shine is in simplifying certain types of complicated `boolean` logic into simple numeric addition. Of course, this is not a general-purpose technique, but a specific solution for specific cases.

Consider:

```
function onlyOne(a,b,c) {
    return (!((a && !b && !c) ||
              (!a && b && !c) || (!a && !b && c));
}

var a = true;
var b = false;

onlyOne( a, b, b );    // true
onlyOne( b, a, b );    // true

onlyOne( a, b, a );    // false
```

This `onlyOne(...)` utility should only return `true` if exactly one of the arguments is `true` / truthy. It's using *implicit* coercion on the truthy checks and *explicit* coercion on the others, including the final return value.

But what if we needed that utility to be able to handle four, five, or twenty flags in the same way? It's pretty difficult to imagine implementing code that would handle all those permutations of comparisons.

But here's where coercing the `boolean` values to `number`s (`0` or `1`, obviously) can greatly help:

```
function onlyOne() {
  var sum = 0;
  for (var i=0; i < arguments.length; i++) {
    // skip falsy values. same as treating
    // them as 0's, but avoids NaN's.
    if (arguments[i]) {
      sum += arguments[i];
    }
  }
  return sum == 1;
}

var a = true;
var b = false;

onlyOne( b, a );          // true
onlyOne( b, a, b, b, b ); // true

onlyOne( b, b );          // false
onlyOne( b, a, b, b, b, a ); // false
```

Note: Of course, instead of the `for` loop in `onlyOne(...)`, you could more tersely use the ES5 `reduce(..)` utility, but I didn't want to obscure the concepts.

What we're doing here is relying on the `1` for `true` /truthy coercions, and numerically adding them all up. `sum += arguments[i]` uses *implicit* coercion to make that happen. If one and only one value in the `arguments` list is `true`, then the numeric sum will be `1`, otherwise the sum will not be `1` and thus the desired condition is not met.

We could of course do this with *explicit* coercion instead:

```
function onlyOne() {
  var sum = 0;
  for (var i=0; i < arguments.length; i++) {
    sum += Number( !!arguments[i] );
  }
  return sum === 1;
}
```

We first use `!!arguments[i]` to force the coercion of the value to `true` or `false`. That's so you could pass non-`boolean` values in, like `onlyOne("42", 0)`, and it would still work as expected (otherwise you'd end up with `string` concatenation and the logic would be incorrect).

Once we're sure it's a `boolean`, we do another *explicit* coercion with `Number(..)` to make sure the value is `0` or `1`.

Is the *explicit* coercion form of this utility "better"? It does avoid the `NaN` trap as explained in the code comments. But, ultimately, it depends on your needs. I personally think the former version, relying on *implicit* coercion is more elegant (if you won't be passing `undefined` or `NaN`), and the *explicit* version is needlessly more verbose.

But as with almost everything we're discussing here, it's a judgment call.

Note: Regardless of *implicit* or *explicit* approaches, you could easily make `onlyTwo(..)` or `onlyFive(..)` variations by simply changing the final comparison from `1`, to `2` or `5`, respectively. That's drastically easier than adding a bunch of `&&` and `||` expressions. So, generally, coercion is very helpful in this case.

Implicitly: * --> Boolean

Now, let's turn our attention to *implicit* coercion to `boolean` values, as it's by far the most common and also by far the most potentially troublesome.

Remember, *implicit* coercion is what kicks in when you use a value in such a way that it forces the value to be converted. For numeric and `string` operations, it's fairly easy to see how the coercions can occur.

But, what sort of expression operations require/force (*implicitly*) a `boolean` coercion?

1. The test expression in an `if(..)` statement.
2. The test expression (second clause) in a `for(..;..;..)` header.
3. The test expression in `while(..)` and `do..while(..)` loops.
4. The test expression (first clause) in `? :` ternary expressions.
5. The left-hand operand (which serves as a test expression -- see below!) to the `||` ("logical or") and `&&` ("logical and") operators.

Any value used in these contexts that is not already a `boolean` will be *implicitly* coerced to a `boolean` using the rules of the `ToBoolean` abstract operation covered earlier in this chapter.

Let's look at some examples:

```

var a = 42;
var b = "abc";
var c;
var d = null;

if (a) {
    console.log( "yep" );           // yep
}

while (c) {
    console.log( "nope, never runs" );
}

c = d ? a : b;                   // "abc"

if ((a && d) || c) {
    console.log( "yep" );           // yep
}

```

In all these contexts, the non-`boolean` values are *implicitly coerced* to their `boolean` equivalents to make the test decisions.

Operators `||` and `&&`

It's quite likely that you have seen the `||` ("logical or") and `&&` ("logical and") operators in most or all other languages you've used. So it'd be natural to assume that they work basically the same in JavaScript as in other similar languages.

There's some very little known, but very important, nuance here.

In fact, I would argue these operators shouldn't even be called "logical _ operators", as that name is incomplete in describing what they do. If I were to give them a more accurate (if more clumsy) name, I'd call them "selector operators," or more completely, "operand selector operators."

Why? Because they don't actually result in a *logic* value (aka `boolean`) in JavaScript, as they do in some other languages.

So what *do* they result in? They result in the value of one (and only one) of their two operands. In other words, **they select one of the two operand's values**.

Quoting the ES5 spec from section 11.11:

The value produced by a `&&` or `||` operator is not necessarily of type Boolean. The value produced will always be the value of one of the two operand expressions.

Let's illustrate:

```
var a = 42;
var b = "abc";
var c = null;

a || b;          // 42
a && b;          // "abc"

c || b;          // "abc"
c && b;          // null
```

Wait, what!? Think about that. In languages like C and PHP, those expressions result in `true` or `false`, but in JS (and Python and Ruby, for that matter!), the result comes from the values themselves.

Both `||` and `&&` operators perform a `boolean` test on the **first operand** (`a` or `c`). If the operand is not already `boolean` (as it's not, here), a normal `ToBoolean` coercion occurs, so that the test can be performed.

For the `||` operator, if the test is `true`, the `||` expression results in the value of the **first operand** (`a` or `c`). If the test is `false`, the `||` expression results in the value of the **second operand** (`b`).

Inversely, for the `&&` operator, if the test is `true`, the `&&` expression results in the value of the **second operand** (`b`). If the test is `false`, the `&&` expression results in the value of the **first operand** (`a` or `c`).

The result of a `||` or `&&` expression is always the underlying value of one of the operands, **not** the (possibly coerced) result of the test. In `c && b`, `c` is `null`, and thus falsy. But the `&&` expression itself results in `null` (the value in `c`), not in the coerced `false` used in the test.

Do you see how these operators act as "operand selectors", now?

Another way of thinking about these operators:

```
a || b;
// roughly equivalent to:
a ? a : b;

a && b;
// roughly equivalent to:
a ? b : a;
```

Note: I call `a || b` "roughly equivalent" to `a ? a : b` because the outcome is identical, but there's a nuanced difference. In `a ? a : b`, if `a` was a more complex expression (like for instance one that might have side effects like calling a `function`, etc.), then the `a` expression would possibly be evaluated twice (if the first evaluation was truthy). By contrast, for `a || b`, the `a` expression is evaluated only once, and that value is used both for the coercive test as well as the result value (if appropriate). The same nuance applies to the `a && b` and `a ? b : a` expressions.

An extremely common and helpful usage of this behavior, which there's a good chance you may have used before and not fully understood, is:

```
function foo(a, b) {
  a = a || "hello";
  b = b || "world";

  console.log( a + " " + b );
}

foo();           // "hello world"
foo( "yeah", "yeah!" ); // "yeah yeah!"
```

The `a = a || "hello"` idiom (sometimes said to be JavaScript's version of the C# "null coalescing operator") acts to test `a` and if it has no value (or only an undesired falsy value), provides a backup default value (`"hello"`).

Be careful, though!

```
foo( "That's it!", "" ); // "That's it! world" <-- Oops!
```

See the problem? `""` as the second argument is a falsy value (see `ToBoolean` earlier in this chapter), so the `b = b || "world"` test fails, and the `"world"` default value is substituted, even though the intent probably was to have the explicitly passed `""` be the value assigned to `b`.

This `||` idiom is extremely common, and quite helpful, but you have to use it only in cases where *all falsy values* should be skipped. Otherwise, you'll need to be more explicit in your test, and probably use a `? :` ternary instead.

This *default value assignment* idiom is so common (and useful!) that even those who publicly and vehemently decry JavaScript coercion often use it in their own code!

What about `&&`?

There's another idiom that is quite a bit less commonly authored manually, but which is used by JS minifiers frequently. The `&&` operator "selects" the second operand if and only if the first operand tests as truthy, and this usage is sometimes called the "guard operator" (also see "Short Circuited" in Chapter 5) -- the first expression test "guards" the second expression:

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

var a = 42;

a && foo(); // 42
```

`foo()` gets called only because `a` tests as truthy. If that test failed, this `a && foo()` expression statement would just silently stop -- this is known as "short circuiting" -- and never call `foo()`.

Again, it's not nearly as common for people to author such things. Usually, they'd do `if (a) { foo(); }` instead. But JS minifiers choose `a && foo()` because it's much shorter. So, now, if you ever have to decipher such code, you'll know what it's doing and why.

OK, so `||` and `&&` have some neat tricks up their sleeve, as long as you're willing to allow the *implicit* coercion into the mix.

Note: Both the `a = b || "something"` and `a && b()` idioms rely on short circuiting behavior, which we cover in more detail in Chapter 5.

The fact that these operators don't actually result in `true` and `false` is possibly messing with your head a little bit by now. You're probably wondering how all your `if` statements and `for` loops have been working, if they've included compound logical expressions like `a && (b || c)`.

Don't worry! The sky is not falling. Your code is (probably) just fine. It's just that you probably never realized before that there was an *implicit* coercion to `boolean` going on **after** the compound expression was evaluated.

Consider:

```

var a = 42;
var b = null;
var c = "foo";

if (a && (b || c)) {
    console.log( "yep" );
}

```

This code still works the way you always thought it did, except for one subtle extra detail. The `a && (b || c)` expression *actually* results in `"foo"`, not `true`. So, the `if` statement *then* forces the `"foo"` value to coerce to a `boolean`, which of course will be `true`.

See? No reason to panic. Your code is probably still safe. But now you know more about how it does what it does.

And now you also realize that such code is using *implicit* coercion. If you're in the "avoid (implicit) coercion camp" still, you're going to need to go back and make all of those tests *explicit*:

```

if (!!a && (!!b || !!c)) {
    console.log( "yep" );
}

```

Good luck with that! ... Sorry, just teasing.

Symbol Coercion

Up to this point, there's been almost no observable outcome difference between *explicit* and *implicit* coercion -- only the readability of code has been at stake.

But ES6 Symbols introduce a gotcha into the coercion system that we need to discuss briefly. For reasons that go well beyond the scope of what we'll discuss in this book, *explicit* coercion of a `symbol` to a `string` is allowed, but *implicit* coercion of the same is disallowed and throws an error.

Consider:

```

var s1 = Symbol( "cool" );
String( s1 );                      // "Symbol(cool)"

var s2 = Symbol( "not cool" );
s2 + "";                           // TypeError

```

`symbol` values cannot coerce to `number` at all (throws an error either way), but strangely they can both *explicitly* and *implicitly* coerce to `boolean` (always `true`).

Consistency is always easier to learn, and exceptions are never fun to deal with, but we just need to be careful around the new ES6 `symbol` values and how we coerce them.

The good news: it's probably going to be exceedingly rare for you to need to coerce a `symbol` value. The way they're typically used (see Chapter 3) will probably not call for coercion on a normal basis.

Loose Equals vs. Strict Equals

Loose equals is the `==` operator, and strict equals is the `===` operator. Both operators are used for comparing two values for "equality," but the "loose" vs. "strict" indicates a **very important** difference in behavior between the two, specifically in how they decide "equality."

A very common misconception about these two operators is: "`==` checks values for equality and `===` checks both values and types for equality." While that sounds nice and reasonable, it's inaccurate. Countless well-respected JavaScript books and blogs have said exactly that, but unfortunately they're all *wrong*.

The correct description is: "`==` allows coercion in the equality comparison and `===` disallows coercion."

Equality Performance

Stop and think about the difference between the first (inaccurate) explanation and this second (accurate) one.

In the first explanation, it seems obvious that `===` is *doing more work* than `==`, because it has to *also* check the type. In the second explanation, `==` is the one *doing more work* because it has to follow through the steps of coercion if the types are different.

Don't fall into the trap, as many have, of thinking this has anything to do with performance, though, as if `==` is going to be slower than `===` in any relevant way. While it's measurable that coercion does take a *little bit* of processing time, it's mere microseconds (yes, that's millionths of a second!).

If you're comparing two values of the same types, `==` and `===` use the identical algorithm, and so other than minor differences in engine implementation, they should do the same work.

If you're comparing two values of different types, the performance isn't the important factor. What you should be asking yourself is: when comparing these two values, do I want coercion or not?

If you want coercion, use `==` loose equality, but if you don't want coercion, use `===` strict equality.

Note: The implication here then is that both `==` and `===` check the types of their operands. The difference is in how they respond if the types don't match.

Abstract Equality

The `==` operator's behavior is defined as "The Abstract Equality Comparison Algorithm" in section 11.9.3 of the ES5 spec. What's listed there is a comprehensive but simple algorithm that explicitly states every possible combination of types, and how the coercions (if necessary) should happen for each combination.

Warning: When (*implicit*) coercion is maligned as being too complicated and too flawed to be a *useful good part*, it is these rules of "abstract equality" that are being condemned. Generally, they are said to be too complex and too unintuitive for developers to practically learn and use, and that they are prone more to causing bugs in JS programs than to enabling greater code readability. I believe this is a flawed premise -- that you readers are competent developers who write (and read and understand!) algorithms (aka code) all day long. So, what follows is a plain exposition of the "abstract equality" in simple terms. But I implore you to also read the ES5 spec section 11.9.3. I think you'll be surprised at just how reasonable it is.

Basically, the first clause (11.9.3.1) says, if the two values being compared are of the same type, they are simply and naturally compared via Identity as you'd expect. For example, `42` is only equal to `42`, and `"abc"` is only equal to `"abc"`.

Some minor exceptions to normal expectation to be aware of:

- `NaN` is never equal to itself (see Chapter 2)
- `+0` and `-0` are equal to each other (see Chapter 2)

The final provision in clause 11.9.3.1 is for `==` loose equality comparison with `object s` (including `function s` and `array s`). Two such values are only *equal* if they are both references to *the exact same value*. No coercion occurs here.

Note: The `===` strict equality comparison is defined identically to 11.9.3.1, including the provision about two `object` values. It's a very little known fact that `==` and `===` **behave identically** in the case where two `object s` are being compared!

The rest of the algorithm in 11.9.3 specifies that if you use `==` loose equality to compare two values of different types, one or both of the values will need to be *implicitly* coerced. This coercion happens so that both values eventually end up as the same type, which can then directly be compared for equality using simple value Identity.

Note: The `!=` loose not-equality operation is defined exactly as you'd expect, in that it's literally the `==` operation comparison performed in its entirety, then the negation of the result. The same goes for the `!==` strict not-equality operation.

Comparing: `string s` to `number s`

To illustrate `==` coercion, let's first build off the `string` and `number` examples earlier in this chapter:

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

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

As we'd expect, `a === b` fails, because no coercion is allowed, and indeed the `42` and `"42"` values are different.

However, the second comparison `a == b` uses loose equality, which means that if the types happen to be different, the comparison algorithm will perform *implicit* coercion on one or both values.

But exactly what kind of coercion happens here? Does the `a` value of `42` become a `string`, or does the `b` value of `"42"` become a `number`?

In the ES5 spec, clauses 11.9.3.4-5 say:

1. If `Type(x)` is `Number` and `Type(y)` is `String`, return the result of the comparison `x == ToNumber(y)`.
2. If `Type(x)` is `String` and `Type(y)` is `Number`, return the result of the comparison `ToNumber(x) == y`.

Warning: The spec uses `Number` and `String` as the formal names for the types, while this book prefers `number` and `string` for the primitive types. Do not let the capitalization of `Number` in the spec confuse you for the `Number()` native function. For our purposes, the capitalization of the type name is irrelevant -- they have basically the same meaning.

Clearly, the spec says the `"42"` value is coerced to a `number` for the comparison. The *how* of that coercion has already been covered earlier, specifically with the `ToNumber` abstract operation. In this case, it's quite obvious then that the resulting two `42` values are equal.

Comparing: anything to `boolean`

One of the biggest gotchas with the *implicit* coercion of `==` loose equality pops up when you try to compare a value directly to `true` or `false`.

Consider:

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

a == b;    // false
```

Wait, what happened here!? We know that `"42"` is a truthy value (see earlier in this chapter). So, how come it's not `==` loose equal to `true`?

The reason is both simple and deceptively tricky. It's so easy to misunderstand, many JS developers never pay close enough attention to fully grasp it.

Let's again quote the spec, clauses 11.9.3.6-7:

1. If `Type(x)` is Boolean, return the result of the comparison `ToNumber(x) == y`.
2. If `Type(y)` is Boolean, return the result of the comparison `x == ToNumber(y)`.

Let's break that down. First:

```
var x = true;
var y = "42";

x == y; // false
```

The `Type(x)` is indeed `Boolean`, so it performs `ToNumber(x)`, which coerces `true` to `1`. Now, `1 == "42"` is evaluated. The types are still different, so (essentially recursively) we reconsult the algorithm, which just as above will coerce `"42"` to `42`, and `1 == 42` is clearly `false`.

Reverse it, and we still get the same outcome:

```
var x = "42";
var y = false;

x == y; // false
```

The `Type(y)` is `Boolean` this time, so `ToNumber(y)` yields `0`. `"42" == 0` recursively becomes `42 == 0`, which is of course `false`.

In other words, **the value "42" is neither `= true` nor `= false`**. At first, that statement might seem crazy. How can a value be neither truthy nor falsy?

But that's the problem! You're asking the wrong question, entirely. It's not your fault, really. Your brain is tricking you.

`"42"` is indeed truthy, but `"42" == true` **is not performing a boolean test/coercion** at all, no matter what your brain says. `"42"` *is not* being coerced to a `boolean` (`true`), but instead `true` is being coerced to a `1`, and then `"42"` is being coerced to `42`.

Whether we like it or not, `ToBoolean` is not even involved here, so the truthiness or falsiness of `"42"` is irrelevant to the `==` operation!

What *is* relevant is to understand how the `==` comparison algorithm behaves with all the different type combinations. As it regards a `boolean` value on either side of the `==`, a `boolean` **always coerces to a `number` first**.

If that seems strange to you, you're not alone. I personally would recommend to never, ever, under any circumstances, use `= true` or `= false`. Ever.

But remember, I'm only talking about `==` here. `== true` and `== false` wouldn't allow the coercion, so they're safe from this hidden `ToNumber` coercion.

Consider:

```

var a = "42";

// bad (will fail!):
if (a == true) {
    // ...
}

// also bad (will fail!):
if (a === true) {
    // ...
}

// good enough (works implicitly):
if (a) {
    // ...
}

// better (works explicitly):
if (!!a) {
    // ...
}

// also great (works explicitly):
if (Boolean( a )) {
    // ...
}

```

If you avoid ever using `== true` or `== false` (aka loose equality with `boolean s`) in your code, you'll never have to worry about this truthiness/falsiness mental gotcha.

Comparing: `null s to undefined s`

Another example of *implicit* coercion can be seen with `==` loose equality between `null` and `undefined` values. Yet again quoting the ES5 spec, clauses 11.9.3.2-3:

1. If `x` is `null` and `y` is `undefined`, return `true`.
2. If `x` is `undefined` and `y` is `null`, return `true`.

`null` and `undefined`, when compared with `==` loose equality, equate to (aka coerce to) each other (as well as themselves, obviously), and no other values in the entire language.

What this means is that `null` and `undefined` can be treated as indistinguishable for comparison purposes, if you use the `==` loose equality operator to allow their mutual *implicit* coercion.

```

var a = null;
var b;

a == b;          // true
a == null;       // true
b == null;       // true

a == false;      // false
b == false;      // false
a == "";         // false
b == "";         // false
a == 0;          // false
b == 0;          // false

```

The coercion between `null` and `undefined` is safe and predictable, and no other values can give false positives in such a check. I recommend using this coercion to allow `null` and `undefined` to be indistinguishable and thus treated as the same value.

For example:

```

var a = doSomething();

if (a == null) {
    // ...
}

```

The `a == null` check will pass only if `doSomething()` returns either `null` or `undefined`, and will fail with any other value, even other falsy values like `0`, `false`, and `""`.

The *explicit* form of the check, which disallows any such coercion, is (I think) unnecessarily much uglier (and perhaps a tiny bit less performant!):

```

var a = doSomething();

if (a === undefined || a === null) {
    // ...
}

```

In my opinion, the form `a == null` is yet another example where *implicit* coercion improves code readability, but does so in a reliably safe way.

Comparing: object s to non-object s

If an `object / function / array` is compared to a simple scalar primitive (`string`, `number`, or `boolean`), the ES5 spec says in clauses 11.9.3.8-9:

1. If Type(x) is either String or Number and Type(y) is Object, return the result of the comparison `x == ToPrimitive(y)`.
2. If Type(x) is Object and Type(y) is either String or Number, return the result of the comparison `ToPrimitive(x) == y`.

Note: You may notice that these clauses only mention `String` and `Number`, but not `Boolean`. That's because, as quoted earlier, clauses 11.9.3.6-7 take care of coercing any `Boolean` operand presented to a `Number` first.

Consider:

```
var a = 42;
var b = [ 42 ];

a == b;      // true
```

The `[42]` value has its `ToPrimitive` abstract operation called (see the "Abstract Value Operations" section earlier), which results in the `"42"` value. From there, it's just `42 == "42"`, which as we've already covered becomes `42 == 42`, so `a` and `b` are found to be coercively equal.

Tip: All the quirks of the `ToPrimitive` abstract operation that we discussed earlier in this chapter (`toString()`, `valueOf()`) apply here as you'd expect. This can be quite useful if you have a complex data structure that you want to define a custom `valueOf()` method on, to provide a simple value for equality comparison purposes.

In Chapter 3, we covered "unboxing," where an `object` wrapper around a primitive value (like from `new String("abc")`, for instance) is unwrapped, and the underlying primitive value (`"abc"`) is returned. This behavior is related to the `ToPrimitive` coercion in the `==` algorithm:

```
var a = "abc";
var b = Object( a );      // same as `new String( a )`

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

`a == b` is `true` because `b` is coerced (aka "unboxed," unwrapped) via `ToPrimitive` to its underlying `"abc"` simple scalar primitive value, which is the same as the value in `a`.

There are some values where this is not the case, though, because of other overriding rules in the `==` algorithm. Consider:

```

var a = null;
var b = Object( a );      // same as `Object()``
a == b;                  // false

var c = undefined;
var d = Object( c );      // same as `Object()``
c == d;                  // false

var e = NaN;
var f = Object( e );      // same as `new Number( e )`'
e == f;                  // false

```

The `null` and `undefined` values cannot be boxed -- they have no object wrapper equivalent -- so `Object(null)` is just like `Object()` in that both just produce a normal object.

`NaN` can be boxed to its `Number` object wrapper equivalent, but when `==` causes an unboxing, the `NaN == NaN` comparison fails because `NaN` is never equal to itself (see Chapter 2).

Edge Cases

Now that we've thoroughly examined how the *implicit* coercion of `==` loose equality works (in both sensible and surprising ways), let's try to call out the worst, craziest corner cases so we can see what we need to avoid to not get bitten with coercion bugs.

First, let's examine how modifying the built-in native prototypes can produce crazy results:

A Number By Any Other Value Would...

```

Number.prototype.valueOf = function() {
    return 3;
};

new Number( 2 ) == 3;      // true

```

Warning: `2 == 3` would not have fallen into this trap, because neither `2` nor `3` would have invoked the built-in `Number.prototype.valueOf()` method because both are already primitive `number` values and can be compared directly. However, `new Number(2)` must go through the `ToPrimitive` coercion, and thus invoke `valueOf()`.

Evil, huh? Of course it is. No one should ever do such a thing. The fact that you *can* do this is sometimes used as a criticism of coercion and `==`. But that's misdirected frustration. JavaScript is not *bad* because you can do such things, a developer is *bad if they do such*

things. Don't fall into the "my programming language should protect me from myself" fallacy.

Next, let's consider another tricky example, which takes the evil from the previous example to another level:

```
if (a == 2 && a == 3) {  
    // ...  
}
```

You might think this would be impossible, because `a` could never be equal to both `2` and `3` *at the same time*. But "at the same time" is inaccurate, since the first expression `a == 2` happens strictly *before* `a == 3`.

So, what if we make `a.valueOf()` have side effects each time it's called, such that the first time it returns `2` and the second time it's called it returns `3`? Pretty easy:

```
var i = 2;  
  
Number.prototype.valueOf = function() {  
    return i++;  
};  
  
var a = new Number( 42 );  
  
if (a == 2 && a == 3) {  
    console.log( "Yep, this happened." );  
}
```

Again, these are evil tricks. Don't do them. But also don't use them as complaints against coercion. Potential abuses of a mechanism are not sufficient evidence to condemn the mechanism. Just avoid these crazy tricks, and stick only with valid and proper usage of coercion.

False-y Comparisons

The most common complaint against *implicit* coercion in `==` comparisons comes from how falsy values behave surprisingly when compared to each other.

To illustrate, let's look at a list of the corner-cases around falsy value comparisons, to see which ones are reasonable and which are troublesome:

```

"0" == null;           // false
"0" == undefined;    // false
"0" == false;         // true -- UH OH!
"0" == NaN;           // false
"0" == 0;              // true
"0" == "";             // false

false == null;         // false
false == undefined;   // false
false == NaN;          // false
false == 0;             // true -- UH OH!
false == "";            // true -- UH OH!
false == [];            // true -- UH OH!
false == {};            // false

"" == null;            // false
"" == undefined;       // false
"" == NaN;             // false
"" == 0;                // true -- UH OH!
"" == [];               // true -- UH OH!
"" == {};               // false

0 == null;              // false
0 == undefined;         // false
0 == NaN;               // false
0 == [];                 // true -- UH OH!
0 == {};                  // false

```

In this list of 24 comparisons, 17 of them are quite reasonable and predictable. For example, we know that `""` and `NaN` are not at all equatable values, and indeed they don't coerce to be loose equals, whereas `"0"` and `0` are reasonably equatable and *do* coerce as loose equals.

However, seven of the comparisons are marked with "UH OH!" because as false positives, they are much more likely gotchas that could trip you up. `""` and `0` are definitely distinctly different values, and it's rare you'd want to treat them as equatable, so their mutual coercion is troublesome. Note that there aren't any false negatives here.

The Crazy Ones

We don't have to stop there, though. We can keep looking for even more troublesome coercions:

```
[] == ![];           // true
```

Oooo, that seems at a higher level of crazy, right!? Your brain may likely trick you that you're comparing a truthy to a falsy value, so the `true` result is surprising, as we know a value can never be truthy and falsy at the same time!

But that's not what's actually happening. Let's break it down. What do we know about the `!` unary operator? It explicitly coerces to a `boolean` using the `ToBoolean` rules (and it also flips the parity). So before `[] == ![]` is even processed, it's actually already translated to `[] == false`. We already saw that form in our above list (`false == []`), so its surprise result is *not new* to us.

How about other corner cases?

```
2 == [2];          // true
"" == [null];     // true
```

As we said earlier in our `ToNumber` discussion, the right-hand side `[2]` and `[null]` values will go through a `ToPrimitive` coercion so they can be more readily compared to the simple primitives (`2` and `""`, respectively) on the left-hand side. Since the `valueOf()` for `array` values just returns the `array` itself, coercion fails to stringify the `array`.

`[2]` will become `"2"`, which then is `ToNumber` coerced to `2` for the right-hand side value in the first comparison. `[null]` just straight becomes `""`.

So, `2 == 2` and `"" == ""` are completely understandable.

If your instinct is to still dislike these results, your frustration is not actually with coercion like you probably think it is. It's actually a complaint against the default `array` values' `ToPrimitive` behavior of coercing to a `string` value. More likely, you'd just wish that `[2].toString()` didn't return `"2"`, or that `[null].toString()` didn't return `""`.

But what exactly *should* these `string` coercions result in? I can't really think of any other appropriate `string` coercion of `[2]` than `"2"`, except perhaps `"[2]"` -- but that could be very strange in other contexts!

You could rightly make the case that since `String(null)` becomes `"null"`, then `String([null])` should also become `"null"`. That's a reasonable assertion. So, that's the real culprit.

Implicit coercion itself isn't the evil here. Even an *explicit* coercion of `[null]` to a `string` results in `""`. What's at odds is whether it's sensible at all for `array` values to stringify to the equivalent of their contents, and exactly how that happens. So, direct your frustration at the rules for `String([...])`, because that's where the craziness stems from. Perhaps there should be no stringification coercion of `array`s at all? But that would have lots of other downsides in other parts of the language.

Another famously cited gotcha:

```
0 == "\n";           // true
```

As we discussed earlier with empty `" "`, `"\n"` (or `" " " "` or any other whitespace combination) is coerced via `ToNumber`, and the result is `0`. What other `number` value would you expect whitespace to coerce to? Does it bother you that *explicit* `Number(" ")` yields `0`?

Really the only other reasonable `number` value that empty strings or whitespace strings could coerce to is the `NaN`. But would that *really* be better? The comparison `" " == NaN` would of course fail, but it's unclear that we'd have really *fixed* any of the underlying concerns.

The chances that a real-world JS program fails because `0 == "\n"` are awfully rare, and such corner cases are easy to avoid.

Type conversions **always** have corner cases, in any language -- nothing specific to coercion. The issues here are about second-guessing a certain set of corner cases (and perhaps rightly so!?), but that's not a salient argument against the overall coercion mechanism.

Bottom line: almost any crazy coercion between *normal values* that you're likely to run into (aside from intentionally tricky `valueOf()` or `toString()` hacks as earlier) will boil down to the short seven-item list of gotcha coercions we've identified above.

To contrast against these 24 likely suspects for coercion gotchas, consider another list like this:

```
42 == "43";                      // false
"foo" == 42;                      // false
"true" == true;                   // false

42 == "42";                      // true
"foo" == [ "foo" ];               // true
```

In these nonfalsy, noncorner cases (and there are literally an infinite number of comparisons we could put on this list), the coercion results are totally safe, reasonable, and explainable.

Sanity Check

OK, we've definitely found some crazy stuff when we've looked deeply into *implicit* coercion. No wonder that most developers claim coercion is evil and should be avoided, right!?

But let's take a step back and do a sanity check.

By way of magnitude comparison, we have a *list* of seven troublesome gotcha coercions, but we have *another list* of (at least 17, but actually infinite) coercions that are totally sane and explainable.

If you're looking for a textbook example of "throwing the baby out with the bathwater," this is it: discarding the entirety of coercion (the infinitely large list of safe and useful behaviors) because of a list of literally just seven gotchas.

The more prudent reaction would be to ask, "how can I use the countless *good parts* of coercion, but avoid the few *bad parts*?"

Let's look again at the *bad* list:

```
"0" == false;           // true -- UH OH!
false == 0;             // true -- UH OH!
false == "";            // true -- UH OH!
false == [];            // true -- UH OH!
"" == 0;                // true -- UH OH!
"" == [];              // true -- UH OH!
0 == [];               // true -- UH OH!
```

Four of the seven items on this list involve `== false` comparison, which we said earlier you should **always, always** avoid. That's a pretty easy rule to remember.

Now the list is down to three.

```
"" == 0;                // true -- UH OH!
"" == [];              // true -- UH OH!
0 == [];               // true -- UH OH!
```

Are these reasonable coercions you'd do in a normal JavaScript program? Under what conditions would they really happen?

I don't think it's terribly likely that you'd literally use `== []` in a `boolean` test in your program, at least not if you know what you're doing. You'd probably instead be doing `== ""` or `== 0`, like:

```
function doSomething(a) {
  if (a == "") {
    // ...
  }
}
```

You'd have an oops if you accidentally called `doSomething(0)` or `doSomething([])`. Another scenario:

```
function doSomething(a, b) {
  if (a == b) {
    // ...
  }
}
```

Again, this could break if you did something like `doSomething("", 0)` or `doSomething([], "")`.

So, while the situations *can* exist where these coercions will bite you, and you'll want to be careful around them, they're probably not super common on the whole of your code base.

Safely Using Implicit Coercion

The most important advice I can give you: examine your program and reason about what values can show up on either side of an `==` comparison. To effectively avoid issues with such comparisons, here's some heuristic rules to follow:

1. If either side of the comparison can have `true` or `false` values, don't ever, EVER use `==`.
2. If either side of the comparison can have `[]`, `""`, or `0` values, seriously consider not using `==`.

In these scenarios, it's almost certainly better to use `===` instead of `==`, to avoid unwanted coercion. Follow those two simple rules and pretty much all the coercion gotchas that could reasonably hurt you will effectively be avoided.

Being more explicit/verbose in these cases will save you from a lot of headaches.

The question of `==` vs. `===` is really appropriately framed as: should you allow coercion for a comparison or not?

There's lots of cases where such coercion can be helpful, allowing you to more tersely express some comparison logic (like with `null` and `undefined`, for example).

In the overall scheme of things, there's relatively few cases where *implicit* coercion is truly dangerous. But in those places, for safety sake, definitely use `===`.

Tip: Another place where coercion is guaranteed *not* to bite you is with the `typeof` operator. `typeof` is always going to return you one of seven strings (see Chapter 1), and none of them are the empty `""` string. As such, there's no case where checking the type of some value is going to run afoul of *implicit* coercion. `typeof x == "function"` is 100% as safe and reliable as `typeof x === "function"`. Literally, the spec says the algorithm will be

identical in this situation. So, don't just blindly use `==` everywhere simply because that's what your code tools tell you to do, or (worst of all) because you've been told in some book to **not think about it**. You own the quality of your code.

Is *implicit* coercion evil and dangerous? In a few cases, yes, but overwhelmingly, no.

Be a responsible and mature developer. Learn how to use the power of coercion (both *explicit* and *implicit*) effectively and safely. And teach those around you to do the same.

Here's a handy table made by Alex Dorey (@dorey on GitHub) to visualize a variety of comparisons:

	Not equal	Loose equality Often gives "false" positives like "1" is true; [] is "0"	Strict equality Mostly evaluates as one would expect.
true:	[]	[]	[]
false:	[]	[]	[]
1:	[]	[]	[]
0:	[]	[]	[]
-1:	[]	[]	[]
"true":	[]	[]	[]
"false":	[]	[]	[]
"1":	[]	[]	[]
"0":	[]	[]	[]
"-1":	[]	[]	[]
""	[]	[]	[]
null	[]	[]	[]
undefined	[]	[]	[]
Infinity	[]	[]	[]
-Infinity	[]	[]	[]
[]:	[]	[]	[]
{ }:	[]	[]	[]
[[]]:	[]	[]	[]
[0]:	[]	[]	[]
[1]:	[]	[]	[]
NaN	[]	[]	[]

Source: <https://github.com/dorey/JavaScript-Equality-Table>

Abstract Relational Comparison

While this part of *implicit* coercion often gets a lot less attention, it's important nonetheless to think about what happens with `a < b` comparisons (similar to how we just examined `a == b` in depth).

The "Abstract Relational Comparison" algorithm in ES5 section 11.8.5 essentially divides itself into two parts: what to do if the comparison involves both `string` values (second half), or anything else (first half).

Note: The algorithm is only defined for `a < b`. So, `a > b` is handled as `b < a`.

The algorithm first calls `ToPrimitive` coercion on both values, and if the return result of either call is not a `string`, then both values are coerced to `number` values using the `ToNumber` operation rules, and compared numerically.

For example:

```
var a = [ 42 ];
var b = [ "43" ];

a < b;      // true
b < a;      // false
```

Note: Similar caveats for `-0` and `Nan` apply here as they did in the `==` algorithm discussed earlier.

However, if both values are `string`s for the `<` comparison, simple lexicographic (natural alphabetic) comparison on the characters is performed:

```
var a = [ "42" ];
var b = [ "043" ];

a < b;      // false
```

`a` and `b` are *not* coerced to `number`s, because both of them end up as `string`s after the `ToPrimitive` coercion on the two `array`s. So, `"42"` is compared character by character to `"043"`, starting with the first characters `"4"` and `"0"`, respectively. Since `"0"` is lexicographically *less than* `"4"`, the comparison returns `false`.

The exact same behavior and reasoning goes for:

```
var a = [ 4, 2 ];
var b = [ 0, 4, 3 ];

a < b;      // false
```

Here, `a` becomes `"4,2"` and `b` becomes `"0,4,3"`, and those lexicographically compare identically to the previous snippet.

What about:

```
var a = { b: 42 };
var b = { b: 43 };

a < b; // ??
```

`a < b` is also `false`, because `a` becomes `[object Object]` and `b` becomes `[object Object]`, and so clearly `a` is not lexicographically less than `b`.

But strangely:

```
var a = { b: 42 };
var b = { b: 43 };

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

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

Why is `a == b` not `true`? They're the same `string` value (`"[object Object]"`), so it seems they should be equal, right? Nope. Recall the previous discussion about how `==` works with `object` references.

But then how are `a <= b` and `a >= b` resulting in `true`, if `a < b` and `a == b` and `a > b` are all `false`?

Because the spec says for `a <= b`, it will actually evaluate `b < a` first, and then negate that result. Since `b < a` is *also* `false`, the result of `a <= b` is `true`.

That's probably awfully contrary to how you might have explained what `<=` does up to now, which would likely have been the literal: "less than *or* equal to." JS more accurately considers `<=` as "not greater than" (`!(a > b)`, which JS treats as `!(b < a)`). Moreover, `a >= b` is explained by first considering it as `b <= a`, and then applying the same reasoning.

Unfortunately, there is no "strict relational comparison" as there is for equality. In other words, there's no way to prevent *implicit* coercion from occurring with relational comparisons like `a < b`, other than to ensure that `a` and `b` are of the same type explicitly before making the comparison.

Use the same reasoning from our earlier `==` vs. `===` sanity check discussion. If coercion is helpful and reasonably safe, like in a `42 < "43"` comparison, **use it**. On the other hand, if you need to be safe about a relational comparison, *explicitly coerce* the values first, before using `<` (or its counterparts).

```
var a = [ 42 ];
var b = "043";

a < b;                      // false -- string comparison!
Number( a ) < Number( b );   // true -- number comparison!
```

Review

In this chapter, we turned our attention to how JavaScript type conversions happen, called **coercion**, which can be characterized as either *explicit* or *implicit*.

Coercion gets a bad rap, but it's actually quite useful in many cases. An important task for the responsible JS developer is to take the time to learn all the ins and outs of coercion to decide which parts will help improve their code, and which parts they really should avoid.

Explicit coercion is code which is obvious that the intent is to convert a value from one type to another. The benefit is improvement in readability and maintainability of code by reducing confusion.

Implicit coercion is coercion that is "hidden" as a side-effect of some other operation, where it's not as obvious that the type conversion will occur. While it may seem that *implicit* coercion is the opposite of *explicit* and is thus bad (and indeed, many think so!), actually *implicit* coercion is also about improving the readability of code.

Especially for *implicit*, coercion must be used responsibly and consciously. Know why you're writing the code you're writing, and how it works. Strive to write code that others will easily be able to learn from and understand as well.

Chapter 5: Grammar

The last major topic we want to tackle is how JavaScript's language syntax works (aka its grammar). You may think you know how to write JS, but there's an awful lot of nuance to various parts of the language grammar that lead to confusion and misconception, so we want to dive into those parts and clear some things up.

Note: The term "grammar" may be a little less familiar to readers than the term "syntax." In many ways, they are similar terms, describing the *rules* for how the language works. There are nuanced differences, but they mostly don't matter for our discussion here. The grammar for JavaScript is a structured way to describe how the syntax (operators, keywords, etc.) fits together into well-formed, valid programs. In other words, discussing syntax without grammar would leave out a lot of the important details. So our focus here in this chapter is most accurately described as *grammar*, even though the raw syntax of the language is what developers directly interact with.

Statements & Expressions

It's fairly common for developers to assume that the term "statement" and "expression" are roughly equivalent. But here we need to distinguish between the two, because there are some very important differences in our JS programs.

To draw the distinction, let's borrow from terminology you may be more familiar with: the English language.

A "sentence" is one complete formation of words that expresses a thought. It's comprised of one or more "phrases," each of which can be connected with punctuation marks or conjunction words ("and," "or," etc). A phrase can itself be made up of smaller phrases. Some phrases are incomplete and don't accomplish much by themselves, while other phrases can stand on their own. These rules are collectively called the *grammar* of the English language.

And so it goes with JavaScript grammar. Statements are sentences, expressions are phrases, and operators are conjunctions/punctuation.

Every expression in JS can be evaluated down to a single, specific value result. For example:

```
var a = 3 * 6;
var b = a;
b;
```

In this snippet, `3 * 6` is an expression (evaluates to the value `18`). But `a` on the second line is also an expression, as is `b` on the third line. The `a` and `b` expressions both evaluate to the values stored in those variables at that moment, which also happens to be `18`.

Moreover, each of the three lines is a statement containing expressions. `var a = 3 * 6` and `var b = a` are called "declaration statements" because they each declare a variable (and optionally assign a value to it). The `a = 3 * 6` and `b = a` assignments (minus the `var`s) are called assignment expressions.

The third line contains just the expression `b`, but it's also a statement all by itself (though not a terribly interesting one!). This is generally referred to as an "expression statement."

Statement Completion Values

It's a fairly little known fact that statements all have completion values (even if that value is just `undefined`).

How would you even go about seeing the completion value of a statement?

The most obvious answer is to type the statement into your browser's developer console, because when you execute it, the console by default reports the completion value of the most recent statement it executed.

Let's consider `var b = a`. What's the completion value of that statement?

The `b = a` assignment expression results in the value that was assigned (`18` above), but the `var` statement itself results in `undefined`. Why? Because `var` statements are defined that way in the spec. If you put `var a = 42;` into your console, you'll see `undefined` reported back instead of `42`.

Note: Technically, it's a little more complex than that. In the ES5 spec, section 12.2 "Variable Statement," the `VariableDeclaration` algorithm actually *does* return a value (a `string` containing the name of the variable declared -- weird, huh!?), but that value is basically swallowed up (except for use by the `for..in` loop) by the `VariableStatement` algorithm, which forces an empty (aka `undefined`) completion value.

In fact, if you've done much code experimenting in your console (or in a JavaScript environment REPL -- read/evaluate/print/loop tool), you've probably seen `undefined` reported after many different statements, and perhaps never realized why or what that was.

Put simply, the console is just reporting the statement's completion value.

But what the console prints out for the completion value isn't something we can use inside our program. So how can we capture the completion value?

That's a much more complicated task. Before we explain *how*, let's explore *why* you would want to do that.

We need to consider other types of statement completion values. For example, any regular `{ ... }` block has a completion value of the completion value of its last contained statement/expression.

Consider:

```
var b;

if (true) {
    b = 4 + 38;
}
```

If you typed that into your console/REPL, you'd probably see `42` reported, since `42` is the completion value of the `if` block, which took on the completion value of its last assignment expression statement `b = 4 + 38`.

In other words, the completion value of a block is like an *implicit return* of the last statement value in the block.

Note: This is conceptually familiar in languages like CoffeeScript, which have implicit `return` values from `function`s that are the same as the last statement value in the function.

But there's an obvious problem. This kind of code doesn't work:

```
var a, b;

a = if (true) {
    b = 4 + 38;
};
```

We can't capture the completion value of a statement and assign it into another variable in any easy syntactic/grammatical way (at least not yet!).

So, what can we do?

Warning: For demo purposes only -- don't actually do the following in your real code!

We could use the much maligned `eval(...)` (sometimes pronounced "evil") function to capture this completion value.

```
var a, b;

a = eval( "if (true) { b = 4 + 38; }" );

a;    // 42
```

Yeeaaaahhhh. That's terribly ugly. But it works! And it illustrates the point that statement completion values are a real thing that can be captured not just in our console but in our programs.

There's a proposal for ES7 called "do expression." Here's how it might work:

```
var a, b;

a = do {
  if (true) {
    b = 4 + 38;
  }
};

a;    // 42
```

The `do { ... }` expression executes a block (with one or many statements in it), and the final statement completion value inside the block becomes the completion value of the `do` expression, which can then be assigned to `a` as shown.

The general idea is to be able to treat statements as expressions -- they can show up inside other statements -- without needing to wrap them in an inline function expression and perform an explicit `return ...`.

For now, statement completion values are not much more than trivia. But they're probably going to take on more significance as JS evolves, and hopefully `do { ... }` expressions will reduce the temptation to use stuff like `eval(...)`.

Warning: Repeating my earlier admonition: avoid `eval(...)`. Seriously. See the *Scope & Closures* title of this series for more explanation.

Expression Side Effects

Most expressions don't have side effects. For example:

```
var a = 2;
var b = a + 3;
```

The expression `a + 3` did not *itself* have a side effect, like for instance changing `a`. It had a result, which is `5`, and that result was assigned to `b` in the statement `b = a + 3`.

The most common example of an expression with (possible) side effects is a function call expression:

```
function foo() {
    a = a + 1;
}

var a = 1;
foo();           // result: `undefined`, side effect: changed `a`
```

There are other side-affecting expressions, though. For example:

```
var a = 42;
var b = a++;
```

The expression `a++` has two separate behaviors. *First*, it returns the current value of `a`, which is `42` (which then gets assigned to `b`). But *next*, it changes the value of `a` itself, incrementing it by one.

```
var a = 42;
var b = a++;

a;      // 43
b;      // 42
```

Many developers would mistakenly believe that `b` has value `43` just like `a` does. But the confusion comes from not fully considering the *when* of the side effects of the `++` operator.

The `++` increment operator and the `--` decrement operator are both unary operators (see Chapter 4), which can be used in either a postfix ("after") position or prefix ("before") position.

```
var a = 42;

a++;    // 42
a;      // 43

++a;    // 44
a;      // 44
```

When `++` is used in the prefix position as `++a`, its side effect (incrementing `a`) happens *before* the value is returned from the expression, rather than *after* as with `a++`.

Note: Would you think `++a++` was legal syntax? If you try it, you'll get a `ReferenceError` error, but why? Because side-effecting operators **require a variable reference** to target their side effects to. For `++a++`, the `a++` part is evaluated first (because of operator precedence -- see below), which gives back the value of `a` *before* the increment. But then it tries to evaluate `++42`, which (if you try it) gives the same `ReferenceError` error, since `++` can't have a side effect directly on a value like `42`.

It is sometimes mistakenly thought that you can encapsulate the *after* side effect of `a++` by wrapping it in a `()` pair, like:

```
var a = 42;
var b = (a++);

a;    // 43
b;    // 42
```

Unfortunately, `()` itself doesn't define a new wrapped expression that would be evaluated *after* the *after side effect* of the `a++` expression, as we might have hoped. In fact, even if it did, `a++` returns `42` first, and unless you have another expression that reevaluates `a` after the side effect of `++`, you're not going to get `43` from that expression, so `b` will not be assigned `43`.

There's an option, though: the `,` statement-series comma operator. This operator allows you to string together multiple standalone expression statements into a single statement:

```
var a = 42, b;
b = ( a++, a );

a;    // 43
b;    // 43
```

Note: The `(...)` around `a++, a` is required here. The reason is operator precedence, which we'll cover later in this chapter.

The expression `a++`, `a` means that the second `a` statement expression gets evaluated *after* the *after side effects* of the first `a++` statement expression, which means it returns the `43` value for assignment to `b`.

Another example of a side-effecting operator is `delete`. As we showed in Chapter 2, `delete` is used to remove a property from an `object` or a slot from an `array`. But it's usually just called as a standalone statement:

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

obj.a;          // 42
delete obj.a;  // true
obj.a;          // undefined
```

The result value of the `delete` operator is `true` if the requested operation is valid/allowable, or `false` otherwise. But the side effect of the operator is that it removes the property (or array slot).

Note: What do we mean by valid/allowable? Nonexistent properties, or properties that exist and are configurable (see Chapter 3 of the *this & Object Prototypes* title of this series) will return `true` from the `delete` operator. Otherwise, the result will be `false` or an error.

One last example of a side-effecting operator, which may at once be both obvious and nonobvious, is the `=` assignment operator.

Consider:

```
var a;

a = 42;      // 42
a;           // 42
```

It may not seem like `=` in `a = 42` is a side-effecting operator for the expression. But if we examine the result value of the `a = 42` statement, it's the value that was just assigned (`42`), so the assignment of that same value into `a` is essentially a side effect.

Tip: The same reasoning about side effects goes for the compound-assignment operators like `+=`, `-=`, etc. For example, `a = b += 2` is processed first as `b += 2` (which is `b = b + 2`), and the result of *that* `=` assignment is then assigned to `a`.

This behavior that an assignment expression (or statement) results in the assigned value is primarily useful for chained assignments, such as:

```
var a, b, c;
a = b = c = 42;
```

Here, `c = 42` is evaluated to `42` (with the side effect of assigning `42` to `c`), then `b = 42` is evaluated to `42` (with the side effect of assigning `42` to `b`), and finally `a = 42` is evaluated (with the side effect of assigning `42` to `a`).

Warning: A common mistake developers make with chained assignments is like `var a = b = 42`. While this looks like the same thing, it's not. If that statement were to happen without there also being a separate `var b` (somewhere in the scope) to formally declare `b`, then `var a = b = 42` would not declare `b` directly. Depending on `strict` mode, that would either throw an error or create an accidental global (see the *Scope & Closures* title of this series).

Another scenario to consider:

```
function vowels(str) {
  var matches;

  if (str) {
    // pull out all the vowels
    matches = str.match( /[aeiou]/g );

    if (matches) {
      return matches;
    }
  }
}

vowels( "Hello World" ); // ["e", "o", "o"]
```

This works, and many developers prefer such. But using an idiom where we take advantage of the assignment side effect, we can simplify by combining the two `if` statements into one:

```
function vowels(str) {
  var matches;

  // pull out all the vowels
  if (str && (matches = str.match( /[aeiou]/g ))) {
    return matches;
  }
}

vowels( "Hello World" ); // ["e", "o", "o"]
```

Note: The `(...)` around `matches = str.match..` is required. The reason is operator precedence, which we'll cover in the "Operator Precedence" section later in this chapter.

I prefer this shorter style, as I think it makes it clearer that the two conditionals are in fact related rather than separate. But as with most stylistic choices in JS, it's purely opinion which one is *better*.

Contextual Rules

There are quite a few places in the JavaScript grammar rules where the same syntax means different things depending on where/how it's used. This kind of thing can, in isolation, cause quite a bit of confusion.

We won't exhaustively list all such cases here, but just call out a few of the common ones.

{ ... } Curly Braces

There's two main places (and more coming as JS evolves!) that a pair of `{ ... }` curly braces will show up in your code. Let's take a look at each of them.

Object Literals

First, as an `object literal`:

```
// assume there's a `bar()` function defined

var a = {
  foo: bar()
};
```

How do we know this is an `object literal`? Because the `{ ... }` pair is a value that's getting assigned to `a`.

Note: The `a` reference is called an "l-value" (aka left-hand value) since it's the target of an assignment. The `{ ... }` pair is an "r-value" (aka right-hand value) since it's used *just* as a value (in this case as the source of an assignment).

Labels

What happens if we remove the `var a =` part of the above snippet?

```
// assume there's a `bar()` function defined

{
  foo: bar()
}
```

A lot of developers assume that the `{ ... }` pair is just a standalone `object` literal that doesn't get assigned anywhere. But it's actually entirely different.

Here, `{ ... }` is just a regular code block. It's not very idiomatic in JavaScript (much more so in other languages!) to have a standalone `{ ... }` block like that, but it's perfectly valid JS grammar. It can be especially helpful when combined with `let` block-scoping declarations (see the *Scope & Closures* title in this series).

The `{ ... }` code block here is functionally pretty much identical to the code block being attached to some statement, like a `for` / `while` loop, `if` conditional, etc.

But if it's a normal block of code, what's that bizarre looking `foo: bar()` syntax, and how is that legal?

It's because of a little known (and, frankly, discouraged) feature in JavaScript called "labeled statements." `foo` is a label for the statement `bar()` (which has omitted its trailing `; --` see "Automatic Semicolons" later in this chapter). But what's the point of a labeled statement?

If JavaScript had a `goto` statement, you'd theoretically be able to say `goto foo` and have execution jump to that location in code. `goto`s are usually considered terrible coding idioms as they make code much harder to understand (aka "spaghetti code"), so it's a *very good thing* that JavaScript doesn't have a general `goto`.

However, JS *does* support a limited, special form of `goto : labeled jumps`. Both the `continue` and `break` statements can optionally accept a specified label, in which case the program flow "jumps" kind of like a `goto`. Consider:

```
// `foo` labeled-loop
foo: for (var i=0; i<4; i++) {
    for (var j=0; j<4; j++) {
        // whenever the loops meet, continue outer loop
        if (j == i) {
            // jump to the next iteration of
            // the `foo` labeled-loop
            continue foo;
        }

        // skip odd multiples
        if ((j * i) % 2 == 1) {
            // normal (non-labeled) `continue` of inner loop
            continue;
        }

        console.log( i, j );
    }
}

// 1 0
// 2 0
// 2 1
// 3 0
// 3 2
```

Note: `continue foo` does not mean "go to the 'foo' labeled position to continue", but rather, "continue the loop that is labeled 'foo' with its next iteration." So, it's not *really* an arbitrary `goto`.

As you can see, we skipped over the odd-multiple `3 1` iteration, but the labeled-loop jump also skipped iterations `1 1` and `2 2`.

Perhaps a slightly more useful form of the labeled jump is with `break __` from inside an inner loop where you want to break out of the outer loop. Without a labeled `break`, this same logic could sometimes be rather awkward to write:

```
// `foo` labeled-loop
foo: for (var i=0; i<4; i++) {
    for (var j=0; j<4; j++) {
        if ((i * j) >= 3) {
            console.log( "stopping!", i, j );
            // break out of the `foo` labeled loop
            break foo;
        }

        console.log( i, j );
    }
}

// 0 0
// 0 1
// 0 2
// 0 3
// 1 0
// 1 1
// 1 2
// stopping! 1 3
```

Note: `break foo` does not mean "go to the 'foo' labeled position to continue," but rather, "break out of the loop/block that is labeled 'foo' and continue *after* it." Not exactly a `goto` in the traditional sense, huh?

The nonlabeled `break` alternative to the above would probably need to involve one or more functions, shared scope variable access, etc. It would quite likely be more confusing than labeled `break`, so here using a labeled `break` is perhaps the better option.

A label can apply to a non-loop block, but only `break` can reference such a non-loop label. You can do a labeled `break __` out of any labeled block, but you cannot `continue __` a non-loop label, nor can you do a non-labeled `break` out of a block.

```
function foo() {
    // `bar` labeled-block
    bar: {
        console.log( "Hello" );
        break bar;
        console.log( "never runs" );
    }
    console.log( "World" );
}

foo();
// Hello
// World
```

Labeled loops/blocks are extremely uncommon, and often frowned upon. It's best to avoid them if possible; for example using function calls instead of the loop jumps. But there are perhaps some limited cases where they might be useful. If you're going to use a labeled jump, make sure to document what you're doing with plenty of comments!

It's a very common belief that JSON is a proper subset of JS, so a string of JSON (like `{"a":42}` -- notice the quotes around the property name as JSON requires!) is thought to be a valid JavaScript program. **Not true!** Try putting `{"a":42}` into your JS console, and you'll get an error.

That's because statement labels cannot have quotes around them, so `"a"` is not a valid label, and thus `:` can't come right after it.

So, JSON is truly a subset of JS syntax, but JSON is not valid JS grammar by itself.

One extremely common misconception along these lines is that if you were to load a JS file into a `<script src=..>` tag that only has JSON content in it (like from an API call), the data would be read as valid JavaScript but just be inaccessible to the program. JSON-P (the practice of wrapping the JSON data in a function call, like `foo({"a":42})`) is usually said to solve this inaccessibility by sending the value to one of your program's functions.

Not true! The totally valid JSON value `{"a":42}` by itself would actually throw a JS error because it'd be interpreted as a statement block with an invalid label. But `foo({"a":42})` is valid JS because in it, `{"a":42}` is an `object` literal value being passed to `foo(..)`. So, properly said, **JSON-P makes JSON into valid JS grammar!**

Blocks

Another commonly cited JS gotcha (related to coercion -- see Chapter 4) is:

```
[ ] + { } ; // "[object Object]"  
{ } + [ ] ; // 0
```

This seems to imply the `+` operator gives different results depending on whether the first operand is the `[]` or the `{ }`. But that actually has nothing to do with it!

On the first line, `{ }` appears in the `+` operator's expression, and is therefore interpreted as an actual value (an empty `object`). Chapter 4 explained that `[]` is coerced to `""` and thus `{ }` is coerced to a `string` value as well: `"[object Object]"`.

But on the second line, `{ }` is interpreted as a standalone `{ }` empty block (which does nothing). Blocks don't need semicolons to terminate them, so the lack of one here isn't a problem. Finally, `+ []` is an expression that *explicitly coerces* (see Chapter 4) the `[]` to a `number`, which is the `0` value.

Object Destructuring

Starting with ES6, another place that you'll see `{ ... }` pairs showing up is with "destructuring assignments" (see the *ES6 & Beyond* title of this series for more info), specifically `object` destructuring. Consider:

```
function getData() {
  // ...
  return {
    a: 42,
    b: "foo"
  };
}

var { a, b } = getData();

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

As you can probably tell, `var { a, b } = ...` is a form of ES6 destructuring assignment, which is roughly equivalent to:

```
var res = getData();
var a = res.a;
var b = res.b;
```

Note: `{ a, b }` is actually ES6 destructuring shorthand for `{ a: a, b: b }`, so either will work, but it's expected that the shorter `{ a, b }` will become the preferred form.

Object destructuring with a `{ ... }` pair can also be used for named function arguments, which is sugar for this same sort of implicit object property assignment:

```
function foo({ a, b, c }) {
  // no need for:
  // var a = obj.a, b = obj.b, c = obj.c
  console.log( a, b, c );
}

foo( {
  c: [1, 2, 3],
  a: 42,
  b: "foo"
} ); // 42 "foo" [1, 2, 3]
```

So, the context we use `{ ... }` pairs in entirely determines what they mean, which illustrates the difference between syntax and grammar. It's very important to understand these nuances to avoid unexpected interpretations by the JS engine.

else if And Optional Blocks

It's a common misconception that JavaScript has an `else if` clause, because you can do:

```
if (a) {
    // ...
}
else if (b) {
    // ...
}
else {
    // ...
}
```

But there's a hidden characteristic of the JS grammar here: there is no `else if`. But `if` and `else` statements are allowed to omit the `{ }` around their attached block if they only contain a single statement. You've seen this many times before, undoubtedly:

```
if (a) doSomething( a );
```

Many JS style guides will insist that you always use `{ }` around a single statement block, like:

```
if (a) { doSomething( a ); }
```

However, the exact same grammar rule applies to the `else` clause, so the `else if` form you've likely always coded is *actually* parsed as:

```
if (a) {
    // ...
}
else {
    if (b) {
        // ...
    }
    else {
        // ...
    }
}
```

The `if (b) { ... } else { ... }` is a single statement that follows the `else`, so you can either put the surrounding `{ }` in or not. In other words, when you use `else if`, you're technically breaking that common style guide rule and just defining your `else` with a single `if` statement.

Of course, the `else if` idiom is extremely common and results in one less level of indentation, so it's attractive. Whichever way you do it, just call out explicitly in your own style guide/rules and don't assume things like `else if` are direct grammar rules.

Operator Precedence

As we covered in Chapter 4, JavaScript's version of `&&` and `||` are interesting in that they select and return one of their operands, rather than just resulting in `true` or `false`. That's easy to reason about if there are only two operands and one operator.

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

a && b;    // "foo"
a || b;    // 42
```

But what about when there's two operators involved, and three operands?

```
var a = 42;
var b = "foo";
var c = [1,2,3];

a && b || c; // ???
a || b && c; // ???
```

To understand what those expressions result in, we're going to need to understand what rules govern how the operators are processed when there's more than one present in an expression.

These rules are called "operator precedence."

I bet most readers feel they have a decent grasp on operator precedence. But as with everything else we've covered in this book series, we're going to poke and prod at that understanding to see just how solid it really is, and hopefully learn a few new things along the way.

Recall the example from above:

```
var a = 42, b;
b = ( a++, a );

a;    // 43
b;    // 43
```

But what would happen if we remove the `()` ?

```
var a = 42, b;
b = a++, a;

a;      // 43
b;      // 42
```

Wait! Why did that change the value assigned to `b` ?

Because the `,` operator has a lower precedence than the `=` operator. So, `b = a++, a` is interpreted as `(b = a++), a`. Because (as we explained earlier) `a++` has *after side effects*, the assigned value to `b` is the value `42` before the `++` changes `a`.

This is just a simple matter of needing to understand operator precedence. If you're going to use `,` as a statement-series operator, it's important to know that it actually has the lowest precedence. Every other operator will more tightly bind than `,` will.

Now, recall this example from above:

```
if (str && (matches = str.match( /[aeiou]/g ))) {
    // ...
}
```

We said the `()` around the assignment is required, but why? Because `&&` has higher precedence than `=`, so without the `()` to force the binding, the expression would instead be treated as `(str && matches) = str.match...`. But this would be an error, because the result of `(str && matches)` isn't going to be a variable, but instead a value (in this case `undefined`), and so it can't be the left-hand side of an `=` assignment!

OK, so you probably think you've got this operator precedence thing down.

Let's move on to a more complex example (which we'll carry throughout the next several sections of this chapter) to *really* test your understanding:

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

var d = a && b || c ? c || b ? a : c && b : a;

d;      // ??
```

OK, evil, I admit it. No one would write a string of expressions like that, right? *Probably* not, but we're going to use it to examine various issues around chaining multiple operators together, which *is* a very common task.

The result above is `42`. But that's not nearly as interesting as how we can figure out that answer without just plugging it into a JS program to let JavaScript sort it out.

Let's dig in.

The first question -- it may not have even occurred to you to ask -- is, does the first part (`a && b || c`) behave like `(a && b) || c` or like `a && (b || c)`? Do you know for certain? Can you even convince yourself they are actually different?

```
(false && true) || true;      // true
false && (true || true);    // false
```

So, there's proof they're different. But still, how does `false && true || true` behave? The answer:

```
false && true || true;      // true
(false && true) || true;    // true
```

So we have our answer. The `&&` operator is evaluated first and the `||` operator is evaluated second.

But is that just because of left-to-right processing? Let's reverse the order of operators:

```
true || false && false;      // true
(true || false) && false;    // false -- nope
true || (false && false);    // true -- winner, winner!
```

Now we've proved that `&&` is evaluated first and then `||`, and in this case that was actually counter to generally expected left-to-right processing.

So what caused the behavior? **Operator precedence**.

Every language defines its own operator precedence list. It's dismaying, though, just how uncommon it is that JS developers have read JS's list.

If you knew it well, the above examples wouldn't have tripped you up in the slightest, because you'd already know that `&&` is more precedent than `||`. But I bet a fair amount of readers had to think about it a little bit.

Note: Unfortunately, the JS spec doesn't really have its operator precedence list in a convenient, single location. You have to parse through and understand all the grammar rules. So we'll try to lay out the more common and useful bits here in a more convenient format. For a complete list of operator precedence, see "Operator Precedence" on the MDN site (* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precidence).

Short Circuited

In Chapter 4, we mentioned in a side note the "short circuiting" nature of operators like `&&` and `||`. Let's revisit that in more detail now.

For both `&&` and `||` operators, the right-hand operand will **not be evaluated** if the left-hand operand is sufficient to determine the outcome of the operation. Hence, the name "short circuited" (in that if possible, it will take an early shortcut out).

For example, with `a && b`, `b` is not evaluated if `a` is falsy, because the result of the `&&` operand is already certain, so there's no point in bothering to check `b`. Likewise, with `a || b`, if `a` is truthy, the result of the operand is already certain, so there's no reason to check `b`.

This short circuiting can be very helpful and is commonly used:

```
function doSomething(opts) {
  if (opts && opts.cool) {
    // ...
  }
}
```

The `opts` part of the `opts && opts.cool` test acts as sort of a guard, because if `opts` is unset (or is not an `object`), the expression `opts.cool` would throw an error. The `opts` test failing plus the short circuiting means that `opts.cool` won't even be evaluated, thus no error!

Similarly, you can use `||` short circuiting:

```
function doSomething(opts) {
  if (opts.cache || primeCache()) {
    // ...
  }
}
```

Here, we're checking for `opts.cache` first, and if it's present, we don't call the `primeCache()` function, thus avoiding potentially unnecessary work.

Tighter Binding

But let's turn our attention back to that earlier complex statement example with all the chained operators, specifically the `? :` ternary operator parts. Does the `? :` operator have more or less precedence than the `&&` and `||` operators?

```
a && b || c ? c || b ? a : c && b : a
```

Is that more like this:

```
a && b || (c ? c || (b ? a : c) && b : a)
```

or this?

```
(a && b || c) ? (c || b) ? a : (c && b) : a
```

The answer is the second one. But why?

Because `&&` is more precedent than `||`, and `||` is more precedent than `? :`.

So, the expression `(a && b || c)` is evaluated *first* before the `? :` it participates in. Another way this is commonly explained is that `&&` and `||` "bind more tightly" than `? :`. If the reverse was true, then `c ? c...` would bind more tightly, and it would behave (as the first choice) like `a && b || (c ? c..)`.

Associativity

So, the `&&` and `||` operators bind first, then the `? :` operator. But what about multiple operators of the same precedence? Do they always process left-to-right or right-to-left?

In general, operators are either left-associative or right-associative, referring to whether **grouping happens from the left or from the right**.

It's important to note that associativity is *not* the same thing as left-to-right or right-to-left processing.

But why does it matter whether processing is left-to-right or right-to-left? Because expressions can have side effects, like for instance with function calls:

```
var a = foo() && bar();
```

Here, `foo()` is evaluated first, and then possibly `bar()` depending on the result of the `foo()` expression. That definitely could result in different program behavior than if `bar()` was called before `foo()`.

But this behavior is *just* left-to-right processing (the default behavior in JavaScript!) -- it has nothing to do with the associativity of `&&`. In that example, since there's only one `&&` and thus no relevant grouping here, associativity doesn't even come into play.

But with an expression like `a && b && c`, grouping *will* happen implicitly, meaning that either `a && b` or `b && c` will be evaluated first.

Technically, `a && b && c` will be handled as `(a && b) && c`, because `&&` is left-associative (so is `||`, by the way). However, the right-associative alternative `a && (b && c)` behaves observably the same way. For the same values, the same expressions are evaluated in the same order.

Note: If hypothetically `&&` was right-associative, it would be processed the same as if you manually used `()` to create grouping like `a && (b && c)`. But that still **doesn't mean** that `c` would be processed before `b`. Right-associativity does **not** mean right-to-left evaluation, it means right-to-left **grouping**. Either way, regardless of the grouping/associativity, the strict ordering of evaluation will be `a`, then `b`, then `c` (aka left-to-right).

So it doesn't really matter that much that `&&` and `||` are left-associative, other than to be accurate in how we discuss their definitions.

But that's not always the case. Some operators would behave very differently depending on left-associativity vs. right-associativity.

Consider the `? :` ("ternary" or "conditional") operator:

```
a ? b : c ? d : e;
```

`? :` is right-associative, so which grouping represents how it will be processed?

- `a ? b : (c ? d : e)`
- `(a ? b : c) ? d : e`

The answer is `a ? b : (c ? d : e)`. Unlike with `&&` and `||` above, the right-associativity here actually matters, as `(a ? b : c) ? d : e` *will* behave differently for some (but not all!) combinations of values.

One such example:

```
true ? false : true ? true : true;           // false
true ? false : (true ? true : true);         // false
(true ? false : true) ? true : true;         // true
```

Even more nuanced differences lurk with other value combinations, even if the end result is the same. Consider:

```
true ? false : true ? true : false;          // false
true ? false : (true ? true : false);         // false
(true ? false : true) ? true : false;         // false
```

From that scenario, the same end result implies that the grouping is moot. However:

```
var a = true, b = false, c = true, d = true, e = false;

a ? b : (c ? d : e); // false, evaluates only `a` and `b`
(a ? b : c) ? d : e; // false, evaluates `a`, `b` AND `e`
```

So, we've clearly proved that `? :` is right-associative, and that it actually matters with respect to how the operator behaves if chained with itself.

Another example of right-associativity (grouping) is the `=` operator. Recall the chained assignment example from earlier in the chapter:

```
var a, b, c;

a = b = c = 42;
```

We asserted earlier that `a = b = c = 42` is processed by first evaluating the `c = 42` assignment, then `b = ...`, and finally `a = ...`. Why? Because of the right-associativity, which actually treats the statement like this: `a = (b = (c = 42))`.

Remember our running complex assignment expression example from earlier in the chapter?

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

var d = a && b || c ? c || b ? a : c && b : a;

d;           // 42
```

Armed with our knowledge of precedence and associativity, we should now be able to break down the code into its grouping behavior like this:

```
((a && b) || c) ? ((c || b) ? a : (c && b)) : a
```

Or, to present it indented if that's easier to understand:

```
(  
  (a && b)  
  ||  
  c  
)  
?  
(  
  (c || b)  
  ?  
  a  
  :  
  (c && b)  
)  
:  
a
```

Let's solve it now:

1. (a && b) is "foo".
2. "foo" || c is "foo".
3. For the first ? test, "foo" is truthy.
4. (c || b) is "foo".
5. For the second ? test, "foo" is truthy.
6. a is 42.

That's it, we're done! The answer is 42, just as we saw earlier. That actually wasn't so hard, was it?

Disambiguation

You should now have a much better grasp on operator precedence (and associativity) and feel much more comfortable understanding how code with multiple chained operators will behave.

But an important question remains: should we all write code understanding and perfectly relying on all the rules of operator precedence/associativity? Should we only use () manual grouping when it's necessary to force a different processing binding/order?

Or, on the other hand, should we recognize that even though such rules are *in fact* learnable, there's enough gotchas to warrant ignoring automatic precedence/associativity? If so, should we thus always use `()` manual grouping and remove all reliance on these automatic behaviors?

This debate is highly subjective, and heavily symmetrical to the debate in Chapter 4 over *implicit* coercion. Most developers feel the same way about both debates: either they accept both behaviors and code expecting them, or they discard both behaviors and stick to manual/explicit idioms.

Of course, I cannot answer this question definitively for the reader here anymore than I could in Chapter 4. But I've presented you the pros and cons, and hopefully encouraged enough deeper understanding that you can make informed rather than hype-driven decisions.

In my opinion, there's an important middle ground. We should mix both operator precedence/associativity *and* `()` manual grouping into our programs -- I argue the same way in Chapter 4 for healthy/safe usage of *implicit* coercion, but certainly don't endorse it exclusively without bounds.

For example, `if (a && b && c) ...` is perfectly OK to me, and I wouldn't do `if ((a && b) && c) ...` just to explicitly call out the associativity, because I think it's overly verbose.

On the other hand, if I needed to chain two `? :` conditional operators together, I'd certainly use `()` manual grouping to make it absolutely clear what my intended logic is.

Thus, my advice here is similar to that of Chapter 4: **use operator precedence/associativity where it leads to shorter and cleaner code, but use `()` manual grouping in places where it helps create clarity and reduce confusion.**

Automatic Semicolons

ASI (Automatic Semicolon Insertion) is when JavaScript assumes a `;` in certain places in your JS program even if you didn't put one there.

Why would it do that? Because if you omit even a single required `;` your program would fail. Not very forgiving. ASI allows JS to be tolerant of certain places where `;` aren't commonly thought to be necessary.

It's important to note that ASI will only take effect in the presence of a newline (aka line break). Semicolons are not inserted in the middle of a line.

Basically, if the JS parser parses a line where a parser error would occur (a missing expected `;`), and it can reasonably insert one, it does so. What's reasonable for insertion? Only if there's nothing but whitespace and/or comments between the end of some statement

and that line's newline/line break.

Consider:

```
var a = 42, b  
c;
```

Should JS treat the `c` on the next line as part of the `var` statement? It certainly would if a `,` had come anywhere (even another line) between `b` and `c`. But since there isn't one, JS assumes instead that there's an implied `;` (at the newline) after `b`. Thus, `c;` is left as a standalone expression statement.

Similarly:

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

That's still a valid program without error, because expression statements also accept ASI.

There's certain places where ASI is helpful, like for instance:

```
var a = 42;  
  
do {  
    // ..  
} while (a) // <-- ; expected here!  
a;
```

The grammar requires a `;` after a `do..while` loop, but not after `while` or `for` loops. But most developers don't remember that! So, ASI helpfully steps in and inserts one.

As we said earlier in the chapter, statement blocks do not require `;` termination, so ASI isn't necessary:

```
var a = 42;  
  
while (a) {  
    // ..  
} // <-- no ; expected here  
a;
```

The other major case where ASI kicks in is with the `break`, `continue`, `return`, and (ES6) `yield` keywords:

```
function foo(a) {
  if (!a) return;
  a *= 2;
  // ...
}
```

The `return` statement doesn't carry across the newline to the `a *= 2` expression, as ASI assumes the `;` terminating the `return` statement. Of course, `return` statements can easily break across multiple lines, just not when there's nothing after `return` but the newline/line break.

```
function foo(a) {
  return (
    a * 2 + 3 / 12
  );
}
```

Identical reasoning applies to `break`, `continue`, and `yield`.

Error Correction

One of the most hotly contested *religious wars* in the JS community (besides tabs vs. spaces) is whether to rely heavily/exclusively on ASI or not.

Most, but not all, semicolons are optional, but the two `;`s in the `for (...) ...` loop header are required.

On the pro side of this debate, many developers believe that ASI is a useful mechanism that allows them to write more terse (and more "beautiful") code by omitting all but the strictly required `;`s (which are very few). It is often asserted that ASI makes many `;`s optional, so a correctly written program *without them* is no different than a correctly written program *with them*.

On the con side of the debate, many other developers will assert that there are *too many* places that can be accidental gotchas, especially for newer, less experienced developers, where unintended `;`s being magically inserted change the meaning. Similarly, some developers will argue that if they omit a semicolon, it's a flat-out mistake, and they want their tools (linters, etc.) to catch it before the JS engine *corrects* the mistake under the covers.

Let me just share my perspective. A strict reading of the spec implies that ASI is an "error correction" routine. What kind of error, you may ask? Specifically, a **parser error**. In other words, in an attempt to have the parser fail less, ASI lets it be more tolerant.

But tolerant of what? In my view, the only way a **parser error** occurs is if it's given an incorrect/errored program to parse. So, while ASI is strictly correcting parser errors, the only way it can get such errors is if there were first program authoring errors -- omitting semicolons where the grammar rules require them.

So, to put it more bluntly, when I hear someone claim that they want to omit "optional semicolons," my brain translates that claim to "I want to write the most parser-broken program I can that will still work."

I find that to be a ludicrous position to take and the arguments of saving keystrokes and having more "beautiful code" to be weak at best.

Furthermore, I don't agree that this is the same thing as the spaces vs tabs debate -- that it's purely cosmetic -- but rather I believe it's a fundamental question of writing code that adheres to grammar requirements vs. code that relies on grammar exceptions to just barely skate through.

Another way of looking at it is that relying on ASI is essentially considering newlines to be significant "whitespace." Other languages like Python have true significant whitespace. But is it really appropriate to think of JavaScript as having significant newlines as it stands today?

My take: **use semicolons wherever you know they are "required," and limit your assumptions about ASI to a minimum.**

But don't just take my word for it. Back in 2012, creator of JavaScript Brendan Eich said (<http://brendaneich.com/2012/04/the-infernal-semicolon/>) the following:

The moral of this story: ASI is (formally speaking) a syntactic error correction procedure. If you start to code as if it were a universal significant-newline rule, you will get into trouble. ... I wish I had made newlines more significant in JS back in those ten days in May, 1995. ... Be careful not to use ASI as if it gave JS significant newlines.

Errors

Not only does JavaScript have different *subtypes* of errors (`TypeError` , `ReferenceError` , `SyntaxError` , etc.), but also the grammar defines certain errors to be enforced at compile time, as compared to all other errors that happen during runtime.

In particular, there have long been a number of specific conditions that should be caught and reported as "early errors" (during compilation). Any straight-up syntax error is an early error (e.g., `a = ,`), but also the grammar defines things that are syntactically valid but disallowed nonetheless.

Since execution of your code has not begun yet, these errors are not catchable with `try..catch`; they will just fail the parsing/compilation of your program.

Tip: There's no requirement in the spec about exactly how browsers (and developer tools) should report errors. So you may see variations across browsers in the following error examples, in what specific subtype of error is reported or what the included error message text will be.

One simple example is with syntax inside a regular expression literal. There's nothing wrong with the JS syntax here, but the invalid regex will throw an early error:

```
var a = /+foo/;           // Error!
```

The target of an assignment must be an identifier (or an ES6 destructuring expression that produces one or more identifiers), so a value like `42` in that position is illegal and can be reported right away:

```
var a;
42 = a;           // Error!
```

ES5's `strict` mode defines even more early errors. For example, in `strict` mode, function parameter names cannot be duplicated:

```
function foo(a,b,a) {}           // just fine

function bar(a,b,a) { "use strict"; } // Error!
```

Another `strict` mode early error is an object literal having more than one property of the same name:

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

  var a = {
    b: 42,
    b: 43
  };           // Error!
})();
```

Note: Semantically speaking, such errors aren't technically `syntax` errors but more `grammar` errors -- the above snippets are syntactically valid. But since there is no `GrammarError` type, some browsers use `SyntaxError` instead.

Using Variables Too Early

ES6 defines a (frankly confusingly named) new concept called the TDZ ("Temporal Dead Zone").

The TDZ refers to places in code where a variable reference cannot yet be made, because it hasn't reached its required initialization.

The most clear example of this is with ES6 `let` block-scoping:

```
{
  a = 2;          // ReferenceError!
  let a;
}
```

The assignment `a = 2` is accessing the `a` variable (which is indeed block-scoped to the `{ ... }` block) before it's been initialized by the `let a` declaration, so it's in the TDZ for `a` and throws an error.

Interestingly, while `typeof` has an exception to be safe for undeclared variables (see Chapter 1), no such safety exception is made for TDZ references:

```
{
  typeof a;    // undefined
  typeof b;    // ReferenceError! (TDZ)
  let b;
}
```

Function Arguments

Another example of a TDZ violation can be seen with ES6 default parameter values (see the *ES6 & Beyond* title of this series):

```
var b = 3;

function foo( a = 42, b = a + b + 5 ) {
  // ...
}
```

The `b` reference in the assignment would happen in the TDZ for the parameter `b` (not pull in the outer `b` reference), so it will throw an error. However, the `a` in the assignment is fine since by that time it's past the TDZ for parameter `a`.

When using ES6's default parameter values, the default value is applied to the parameter if you either omit an argument, or you pass an `undefined` value in its place:

```
function foo( a = 42, b = a + 1 ) {
    console.log( a, b );
}

foo();                // 42 43
foo( undefined );    // 42 43
foo( 5 );            // 5 6
foo( void 0, 7 );    // 42 7
foo( null );         // null 1
```

Note: `null` is coerced to a `0` value in the `a + 1` expression. See Chapter 4 for more info.

From the ES6 default parameter values perspective, there's no difference between omitting an argument and passing an `undefined` value. However, there is a way to detect the difference in some cases:

```
function foo( a = 42, b = a + 1 ) {
    console.log(
        arguments.length, a, b,
        arguments[0], arguments[1]
    );
}

foo();                // 0 42 43 undefined undefined
foo( 10 );           // 1 10 11 10 undefined
foo( 10, undefined );// 2 10 11 10 undefined
foo( 10, null );     // 2 10 null 10 null
```

Even though the default parameter values are applied to the `a` and `b` parameters, if no arguments were passed in those slots, the `arguments` array will not have entries.

Conversely, if you pass an `undefined` argument explicitly, an entry will exist in the `arguments` array for that argument, but it will be `undefined` and not (necessarily) the same as the default value that was applied to the named parameter for that same slot.

While ES6 default parameter values can create divergence between the `arguments` array slot and the corresponding named parameter variable, this same disjointedness can also occur in tricky ways in ES5:

```

function foo(a) {
  a = 42;
  console.log( arguments[0] );
}

foo( 2 );    // 42 (linked)
foo();        // undefined (not linked)

```

If you pass an argument, the `arguments` slot and the named parameter are linked to always have the same value. If you omit the argument, no such linkage occurs.

But in `strict` mode, the linkage doesn't exist regardless:

```

function foo(a) {
  "use strict";
  a = 42;
  console.log( arguments[0] );
}

foo( 2 );    // 2 (not linked)
foo();        // undefined (not linked)

```

It's almost certainly a bad idea to ever rely on any such linkage, and in fact the linkage itself is a leaky abstraction that's exposing an underlying implementation detail of the engine, rather than a properly designed feature.

Use of the `arguments` array has been deprecated (especially in favor of ES6 `... rest` parameters -- see the *ES6 & Beyond* title of this series), but that doesn't mean that it's all bad.

Prior to ES6, `arguments` is the only way to get an array of all passed arguments to pass along to other functions, which turns out to be quite useful. You can also mix named parameters with the `arguments` array and be safe, as long as you follow one simple rule: **never refer to a named parameter and its corresponding `arguments` slot at the same time**. If you avoid that bad practice, you'll never expose the leaky linkage behavior.

```

function foo(a) {
  console.log( a + arguments[1] ); // safe!
}

foo( 10, 32 );    // 42

```

try..finally

You're probably familiar with how the `try..catch` block works. But have you ever stopped to consider the `finally` clause that can be paired with it? In fact, were you aware that `try` only requires either `catch` or `finally`, though both can be present if needed.

The code in the `finally` clause *always* runs (no matter what), and it always runs right after the `try` (and `catch` if present) finish, before any other code runs. In one sense, you can kind of think of the code in a `finally` clause as being in a callback function that will always be called regardless of how the rest of the block behaves.

So what happens if there's a `return` statement inside a `try` clause? It obviously will return a value, right? But does the calling code that receives that value run before or after the `finally`?

```
function foo() {
  try {
    return 42;
  }
  finally {
    console.log( "Hello" );
  }

  console.log( "never runs" );
}

console.log( foo() );
// Hello
// 42
```

The `return 42` runs right away, which sets up the completion value from the `foo()` call. This action completes the `try` clause and the `finally` clause immediately runs next. Only then is the `foo()` function complete, so that its completion value is returned back for the `console.log(...)` statement to use.

The exact same behavior is true of a `throw` inside `try`:

```

function foo() {
  try {
    throw 42;
  }
  finally {
    console.log( "Hello" );
  }

  console.log( "never runs" );
}

console.log( foo() );
// Hello
// Uncaught Exception: 42

```

Now, if an exception is thrown (accidentally or intentionally) inside a `finally` clause, it will override as the primary completion of that function. If a previous `return` in the `try` block had set a completion value for the function, that value will be abandoned.

```

function foo() {
  try {
    return 42;
  }
  finally {
    throw "Oops!";
  }

  console.log( "never runs" );
}

console.log( foo() );
// Uncaught Exception: Oops!

```

It shouldn't be surprising that other nonlinear control statements like `continue` and `break` exhibit similar behavior to `return` and `throw`:

```

for (var i=0; i<10; i++) {
  try {
    continue;
  }
  finally {
    console.log( i );
  }
}
// 0 1 2 3 4 5 6 7 8 9

```

The `console.log(i)` statement runs at the end of the loop iteration, which is caused by the `continue` statement. However, it still runs before the `i++` iteration update statement, which is why the values printed are `0..9` instead of `1..10`.

Note: ES6 adds a `yield` statement, in generators (see the *Async & Performance* title of this series) which in some ways can be seen as an intermediate `return` statement. However, unlike a `return`, a `yield` isn't complete until the generator is resumed, which means a `try { .. yield .. }` has not completed. So an attached `finally` clause will not run right after the `yield` like it does with `return`.

A `return` inside a `finally` has the special ability to override a previous `return` from the `try` or `catch` clause, but only if `return` is explicitly called:

```
function foo() {
  try {
    return 42;
  }
  finally {
    // no `return ..` here, so no override
  }
}

function bar() {
  try {
    return 42;
  }
  finally {
    // override previous `return 42`
    return;
  }
}

function baz() {
  try {
    return 42;
  }
  finally {
    // override previous `return 42`
    return "Hello";
  }
}

foo();    // 42
bar();    // undefined
baz();    // "Hello"
```

Normally, the omission of `return` in a function is the same as `return;` or even `return undefined;`, but inside a `finally` block the omission of `return` does not act like an overriding `return undefined`; it just lets the previous `return` stand.

In fact, we can really up the craziness if we combine `finally` with labeled `break` (discussed earlier in the chapter):

```
function foo() {
  bar: {
    try {
      return 42;
    }
    finally {
      // break out of `bar` labeled block
      break bar;
    }
  }
  console.log( "Crazy" );

  return "Hello";
}

console.log( foo() );
// Crazy
// Hello
```

But... don't do this. Seriously. Using a `finally` + labeled `break` to effectively cancel a `return` is doing your best to create the most confusing code possible. I'd wager no amount of comments will redeem this code.

switch

Let's briefly explore the `switch` statement, a sort-of syntactic shorthand for an `if..else if..else..` statement chain.

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

As you can see, it evaluates `a` once, then matches the resulting value to each `case` expression (just simple value expressions here). If a match is found, execution will begin in that matched `case`, and will either go until a `break` is encountered or until the end of the `switch` block is found.

That much may not surprise you, but there are several quirks about `switch` you may not have noticed before.

First, the matching that occurs between the `a` expression and each `case` expression is identical to the `==` algorithm (see Chapter 4). Often times `switch` es are used with absolute values in `case` statements, as shown above, so strict matching is appropriate.

However, you may wish to allow coercive equality (aka `==`, see Chapter 4), and to do so you'll need to sort of "hack" the `switch` statement a bit:

```
var a = "42";

switch (true) {
  case a == 10:
    console.log( "10 or '10'" );
    break;
  case a == 42:
    console.log( "42 or '42'" );
    break;
  default:
    // never gets here
}
// 42 or '42'
```

This works because the `case` clause can have any expression (not just simple values), which means it will strictly match that expression's result to the test expression (`true`). Since `a == 42` results in `true` here, the match is made.

Despite `==`, the `switch` matching itself is still strict, between `true` and `true` here. If the `case` expression resulted in something that was truthy but not strictly `true` (see Chapter 4), it wouldn't work. This can bite you if you're for instance using a "logical operator" like `||` or `&&` in your expression:

```

var a = "hello world";
var b = 10;

switch (true) {
    case (a || b == 10):
        // never gets here
        break;
    default:
        console.log( "Oops" );
}
// Oops

```

Since the result of `(a || b == 10)` is `"hello world"` and not `true`, the strict match fails. In this case, the fix is to force the expression explicitly to be a `true` or `false`, such as `case !(a || b == 10):` (see Chapter 4).

Lastly, the `default` clause is optional, and it doesn't necessarily have to come at the end (although that's the strong convention). Even in the `default` clause, the same rules apply about encountering a `break` or not:

```

var a = 10;

switch (a) {
    case 1:
    case 2:
        // never gets here
    default:
        console.log( "default" );
    case 3:
        console.log( "3" );
        break;
    case 4:
        console.log( "4" );
}
// default
// 3

```

Note: As discussed previously about labeled `break`s, the `break` inside a `case` clause can also be labeled.

The way this snippet processes is that it passes through all the `case` clause matching first, finds no match, then goes back up to the `default` clause and starts executing. Since there's no `break` there, it continues executing in the already skipped over `case 3` block, before stopping once it hits that `break`.

While this sort of round-about logic is clearly possible in JavaScript, there's almost no chance that it's going to make for reasonable or understandable code. Be very skeptical if you find yourself wanting to create such circular logic flow, and if you really do, make sure you include plenty of code comments to explain what you're up to!

Review

JavaScript grammar has plenty of nuance that we as developers should spend a little more time paying closer attention to than we typically do. A little bit of effort goes a long way to solidifying your deeper knowledge of the language.

Statements and expressions have analogs in English language -- statements are like sentences and expressions are like phrases. Expressions can be pure/self-contained, or they can have side effects.

The JavaScript grammar layers semantic usage rules (aka context) on top of the pure syntax. For example, `{ }` pairs used in various places in your program can mean statement blocks, `object` literals, (ES6) destructuring assignments, or (ES6) named function arguments.

JavaScript operators all have well-defined rules for precedence (which ones bind first before others) and associativity (how multiple operator expressions are implicitly grouped). Once you learn these rules, it's up to you to decide if precedence/associativity are *too implicit* for their own good, or if they will aid in writing shorter, clearer code.

ASI (Automatic Semicolon Insertion) is a parser-error-correction mechanism built into the JS engine, which allows it under certain circumstances to insert an assumed `;` in places where it is required, was omitted, *and* where insertion fixes the parser error. The debate rages over whether this behavior implies that most `;` are optional (and can/should be omitted for cleaner code) or whether it means that omitting them is making mistakes that the JS engine merely cleans up for you.

JavaScript has several types of errors, but it's less known that it has two classifications for errors: "early" (compiler thrown, uncatchable) and "runtime" (`try..catch`able). All syntax errors are obviously early errors that stop the program before it runs, but there are others, too.

Function arguments have an interesting relationship to their formal declared named parameters. Specifically, the `arguments` array has a number of gotchas of leaky abstraction behavior if you're not careful. Avoid `arguments` if you can, but if you must use it, by all means avoid using the positional slot in `arguments` at the same time as using a named parameter for that same argument.

The `finally` clause attached to a `try` (or `try..catch`) offers some very interesting quirks in terms of execution processing order. Some of these quirks can be helpful, but it's possible to create lots of confusion, especially if combined with labeled blocks. As always, use `finally` to make code better and clearer, not more clever or confusing.

The `switch` offers some nice shorthand for `if..else if..` statements, but beware of many common simplifying assumptions about its behavior. There are several quirks that can trip you up if you're not careful, but there's also some neat hidden tricks that `switch` has up its sleeve!

Appendix A: Mixed Environment JavaScript

Beyond the core language mechanics we've fully explored in this book, there are several ways that your JS code can behave differently when it runs in the real world. If JS was executing purely inside an engine, it'd be entirely predictable based on nothing but the black-and-white of the spec. But JS pretty much always runs in the context of a hosting environment, which exposes your code to some degree of unpredictability.

For example, when your code runs alongside code from other sources, or when your code runs in different types of JS engines (not just browsers), there are some things that may behave differently.

We'll briefly explore some of these concerns.

Annex B (ECMAScript)

It's a little known fact that the official name of the language is ECMAScript (referring to the ECMA standards body that manages it). What then is "JavaScript"? JavaScript is the common trademark of the language, of course, but more appropriately, JavaScript is basically the browser implementation of the spec.

The official ECMAScript specification includes "Annex B," which discusses specific deviations from the official spec for the purposes of JS compatibility in browsers.

The proper way to consider these deviations is that they are only reliably present/valid if your code is running in a browser. If your code always runs in browsers, you won't see any observable difference. If not (like if it can run in node.js, Rhino, etc.), or you're not sure, tread carefully.

The main compatibility differences:

- Octal number literals are allowed, such as `0123` (decimal `83`) in non-`strict mode`.
- `window.escape(...)` and `window.unescape(...)` allow you to escape or unescape strings with `%`-delimited hexadecimal escape sequences. For example: `window.escape("foo=97&bar=3%")` produces `"%3Ffoo%3D97%25%26bar%3D3%25"`.
- `String.prototype.substr` is quite similar to `String.prototype.substring`, except that instead of the second parameter being the ending index (noninclusive), the second parameter is the `length` (number of characters to include).

Web ECMAScript

The Web ECMAScript specification (<http://javascript.spec.whatwg.org/>) covers the differences between the official ECMAScript specification and the current JavaScript implementations in browsers.

In other words, these items are "required" of browsers (to be compatible with each other) but are not (as of the time of writing) listed in the "Annex B" section of the official spec:

- `<!--` and `-->` are valid single-line comment delimiters.
- `String.prototype` additions for returning HTML-formatted strings: `anchor(..)` , `big(..)` , `blink(..)` , `bold(..)` , `fixed(..)` , `fontcolor(..)` , `fontsize(..)` , `italics(..)` , `link(..)` , `small(..)` , `strike(..)` , and `sub(..)`. **Note:** These are very rarely used in practice, and are generally discouraged in favor of other built-in DOM APIs or user-defined utilities.
- `RegExp` extensions: `RegExp.$1` .. `RegExp.$9` (match-groups) and `RegExp.lastMatch` / `RegExp["$&"]` (most recent match).
- `Function.prototype` additions: `Function.prototype.arguments` (aliases internal `arguments` object) and `Function.caller` (aliases internal `arguments.caller`). **Note:** `arguments` and thus `arguments.caller` are deprecated, so you should avoid using them if possible. That goes doubly so for these aliases -- don't use them!

Note: Some other minor and rarely used deviations are not included in our list here. See the external "Annex B" and "Web ECMAScript" documents for more detailed information as needed.

Generally speaking, all these differences are rarely used, so the deviations from the specification are not significant concerns. **Just be careful** if you rely on any of them.

Host Objects

The well-covered rules for how variables behave in JS have exceptions to them when it comes to variables that are auto-defined, or otherwise created and provided to JS by the environment that hosts your code (browser, etc.) -- so called, "host objects" (which include both built-in `object`s and `function`s).

For example:

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

typeof a;                                // "object" -- as expected
Object.prototype.toString.call( a );        // "[object HTMLDivElement]"

a.tagName;                                 // "DIV"
```

`a` is not just an `object`, but a special host object because it's a DOM element. It has a different internal `[[Class]]` value (`"HTMLDivElement"`) and comes with predefined (and often unchangeable) properties.

Another such quirk has already been covered, in the "Falsy Objects" section in Chapter 4: some objects can exist but when coerced to `boolean` they (confoundingly) will coerce to `false` instead of the expected `true`.

Other behavior variations with host objects to be aware of can include:

- not having access to normal `object` built-ins like `toString()`
- not being overwritable
- having certain predefined read-only properties
- having methods that cannot be `this`-overridden to other objects
- and more...

Host objects are critical to making our JS code work with its surrounding environment. But it's important to note when you're interacting with a host object and be careful assuming its behaviors, as they will quite often not conform to regular JS `object`s.

One notable example of a host object that you probably interact with regularly is the `console` object and its various functions (`log(..)`, `error(..)`, etc.). The `console` object is provided by the *hosting environment* specifically so your code can interact with it for various development-related output tasks.

In browsers, `console` hooks up to the developer tools' console display, whereas in node.js and other server-side JS environments, `console` is generally connected to the standard-output (`stdout`) and standard-error (`stderr`) streams of the JavaScript environment system process.

Global DOM Variables

You're probably aware that declaring a variable in the global scope (with or without `var`) creates not only a global variable, but also its mirror: a property of the same name on the `global` object (`window` in the browser).

But what may be less common knowledge is that (because of legacy browser behavior) creating DOM elements with `id` attributes creates global variables of those same names. For example:

```
<div id="foo"></div>
```

And:

```
if (typeof foo == "undefined") {  
    foo = 42;          // will never run  
}  
  
console.log( foo );      // HTML element
```

You're perhaps used to managing global variable tests (using `typeof` or `... in window` checks) under the assumption that only JS code creates such variables, but as you can see, the contents of your hosting HTML page can also create them, which can easily throw off your existence check logic if you're not careful.

This is yet one more reason why you should, if at all possible, avoid using global variables, and if you have to, use variables with unique names that won't likely collide. But you also need to make sure not to collide with the HTML content as well as any other code.

Native Prototypes

One of the most widely known and classic pieces of JavaScript *best practice* wisdom is: **never extend native prototypes.**

Whatever method or property name you come up with to add to `Array.prototype` that doesn't (yet) exist, if it's a useful addition and well-designed, and properly named, there's a strong chance it *could* eventually end up being added to the spec -- in which case your extension is now in conflict.

Here's a real example that actually happened to me that illustrates this point well.

I was building an embeddable widget for other websites, and my widget relied on jQuery (though pretty much any framework would have suffered this gotcha). It worked on almost every site, but we ran across one where it was totally broken.

After almost a week of analysis/debugging, I found that the site in question had, buried deep in one of its legacy files, code that looked like this:

```
// Netscape 4 doesn't have Array.push
Array.prototype.push = function(item) {
    this[this.length] = item;
};
```

Aside from the crazy comment (who cares about Netscape 4 anymore!?), this looks reasonable, right?

The problem is, `Array.prototype.push` was added to the spec sometime subsequent to this Netscape 4 era coding, but what was added is not compatible with this code. The standard `push(...)` allows multiple items to be pushed at once. This hacked one ignores the subsequent items.

Basically all JS frameworks have code that relies on `push(...)` with multiple elements. In my case, it was code around the CSS selector engine that was completely busted. But there could conceivably be dozens of other places susceptible.

The developer who originally wrote that `push(...)` hack had the right instinct to call it `push`, but didn't foresee pushing multiple elements. They were certainly acting in good faith, but they created a landmine that didn't go off until almost 10 years later when I unwittingly came along.

There's multiple lessons to take away on all sides.

First, don't extend the natives unless you're absolutely sure your code is the only code that will ever run in that environment. If you can't say that 100%, then extending the natives is dangerous. You must weigh the risks.

Next, don't unconditionally define extensions (because you can overwrite natives accidentally). In this particular example, had the code said this:

```
if (!Array.prototype.push) {
    // Netscape 4 doesn't have Array.push
    Array.prototype.push = function(item) {
        this[this.length] = item;
    };
}
```

The `if` statement guard would have only defined this hacked `push()` for JS environments where it didn't exist. In my case, that probably would have been OK. But even this approach is not without risk:

1. If the site's code (for some crazy reason!) was relying on a `push(...)` that ignored multiple items, that code would have been broken years ago when the standard `push(...)` was rolled out.

2. If any other library had come in and hacked in a `push(...)` ahead of this `if` guard, and it did so in an incompatible way, that would have broken the site at that time.

What that highlights is an interesting question that, frankly, doesn't get enough attention from JS developers: **Should you EVER rely on native built-in behavior** if your code is running in any environment where it's not the only code present?

The strict answer is **no**, but that's awfully impractical. Your code usually can't redefine its own private untouchable versions of all built-in behavior relied on. Even if you *could*, that's pretty wasteful.

So, should you feature-test for the built-in behavior as well as compliance-testing that it does what you expect? And what if that test fails -- should your code just refuse to run?

```
// don't trust Array.prototype.push
(function(){
    if (Array.prototype.push) {
        var a = [];
        a.push(1,2);
        if (a[0] === 1 && a[1] === 2) {
            // tests passed, safe to use!
            return;
        }
    }

    throw Error(
        "Array#push() is missing/broken!"
    );
})();
```

In theory, that sounds plausible, but it's also pretty impractical to design tests for every single built-in method.

So, what should we do? Should we *trust but verify* (feature- and compliance-test) **everything**? Should we just assume existence is compliance and let breakage (caused by others) bubble up as it will?

There's no great answer. The only fact that can be observed is that extending native prototypes is the only way these things bite you.

If you don't do it, and no one else does in the code in your application, you're safe. Otherwise, you should build in at least a little bit of skepticism, pessimism, and expectation of possible breakage.

Having a full set of unit/regression tests of your code that runs in all known environments is one way to surface some of these issues earlier, but it doesn't do anything to actually protect you from these conflicts.

Shims/Polyfills

It's usually said that the only safe place to extend a native is in an older (non-spec-compliant) environment, since that's unlikely to ever change -- new browsers with new spec features replace older browsers rather than amending them.

If you could see into the future, and know for sure what a future standard was going to be, like for `Array.prototype.foobar`, it'd be totally safe to make your own compatible version of it to use now, right?

```
if (!Array.prototype.foobar) {
    // silly, silly
    Array.prototype.foobar = function() {
        this.push( "foo", "bar" );
    };
}
```

If there's already a spec for `Array.prototype.foobar`, and the specified behavior is equal to this logic, you're pretty safe in defining such a snippet, and in that case it's generally called a "polyfill" (or "shim").

Such code is **very** useful to include in your code base to "patch" older browser environments that aren't updated to the newest specs. Using polyfills is a great way to create predictable code across all your supported environments.

Tip: ES5-Shim (<https://github.com/es-shims/es5-shim>) is a comprehensive collection of shims/polyfills for bringing a project up to ES5 baseline, and similarly, ES6-Shim (<https://github.com/es-shims/es6-shim>) provides shims for new APIs added as of ES6. While APIs can be shimmed/polyfilled, new syntax generally cannot. To bridge the syntactic divide, you'll want to also use an ES6-to-ES5 transpiler like Traceur (<https://github.com/google/traceur-compiler/wiki/Getting-Started>).

If there's likely a coming standard, and most discussions agree what it's going to be called and how it will operate, creating the ahead-of-time polyfill for future-facing standards compliance is called "prollyfill" (probably-fill).

The real catch is if some new standard behavior can't be (fully) polyfilled/prollyfilled.

There's debate in the community if a partial-polyfill for the common cases is acceptable (documenting the parts that cannot be polyfilled), or if a polyfill should be avoided if it purely can't be 100% compliant to the spec.

Many developers at least accept some common partial polyfills (like for instance `Object.create(...)`), because the parts that aren't covered are not parts they intend to use anyway.

Some developers believe that the `if` guard around a polyfill/shim should include some form of conformance test, replacing the existing method either if it's absent or fails the tests. This extra layer of compliance testing is sometimes used to distinguish "shim" (compliance tested) from "polyfill" (existence checked).

The only absolute take-away is that there is no absolute *right* answer here. Extending natives, even when done "safely" in older environments, is not 100% safe. The same goes for relying upon (possibly extended) natives in the presence of others' code.

Either should always be done with caution, defensive code, and lots of obvious documentation about the risks.

<script> S

Most browser-viewed websites/applications have more than one file that contains their code, and it's common to have a few or several `<script src=..></script>` elements in the page that load these files separately, and even a few inline-code `<script> .. </script>` elements as well.

But do these separate files/code snippets constitute separate programs or are they collectively one JS program?

The (perhaps surprising) reality is they act more like independent JS programs in most, but not all, respects.

The one thing they *share* is the single `global` object (`window` in the browser), which means multiple files can append their code to that shared namespace and they can all interact.

So, if one `script` element defines a global function `foo()`, when a second `script` later runs, it can access and call `foo()` just as if it had defined the function itself.

But global variable scope *hoisting* (see the *Scope & Closures* title of this series) does not occur across these boundaries, so the following code would not work (because `foo()`'s declaration isn't yet declared), regardless of if they are (as shown) inline `<script> .. </script>` elements or externally loaded `<script src=..></script>` files:

```
<script>foo();</script>

<script>
  function foo() { .. }
</script>
```

But either of these *would* work instead:

```
<script>
  foo();
  function foo() { .. }
</script>
```

Or:

```
<script>
  function foo() { .. }
</script>

<script>foo();</script>
```

Also, if an error occurs in a `script` element (inline or external), as a separate standalone JS program it will fail and stop, but any subsequent `script`s will run (still with the shared `global`) unimpeded.

You can create `script` elements dynamically from your code, and inject them into the DOM of the page, and the code in them will behave basically as if loaded normally in a separate file:

```
var greeting = "Hello World";

var el = document.createElement( "script" );

el.text = "function foo(){ alert( greeting );\n} setTimeout( foo, 1000 );";

document.body.appendChild( el );
```

Note: Of course, if you tried the above snippet but set `el.src` to some file URL instead of setting `el.text` to the code contents, you'd be dynamically creating an externally loaded `<script src=..></script>` element.

One difference between code in an inline code block and that same code in an external file is that in the inline code block, the sequence of characters `</script>` cannot appear together, as (regardless of where it appears) it would be interpreted as the end of the code block. So, beware of code like:

```
<script>
  var code = "<script>alert( 'Hello World' )</script>";
</script>
```

It looks harmless, but the `</script>` appearing inside the `string` literal will terminate the script block abnormally, causing an error. The most common workaround is:

```
"</sc" + "ript>";
```

Also, beware that code inside an external file will be interpreted in the character set (UTF-8, ISO-8859-8, etc.) the file is served with (or the default), but that same code in an inline `script` element in your HTML page will be interpreted by the character set of the page (or its default).

Warning: The `charset` attribute will not work on inline script elements.

Another deprecated practice with inline `script` elements is including HTML-style or X(HT)ML-style comments around inline code, like:

```
<script>
<!!--
alert( "Hello" );
//-->
</script>

<script>
<!!--//--><! [CDATA[//><! --
alert( "World" );
//--><!]]>
</script>
```

Both of these are totally unnecessary now, so if you're still doing that, stop it!

Note: Both `<!--` and `-->` (HTML-style comments) are actually specified as valid single-line comment delimiters (`var x = 2; <!-- valid comment and --> another valid line comment`) in JavaScript (see the "Web ECMAScript" section earlier), purely because of this old technique. But never use them.

Reserved Words

The ES5 spec defines a set of "reserved words" in Section 7.6.1 that cannot be used as standalone variable names. Technically, there are four categories: "keywords", "future reserved words", the `null` literal, and the `true` / `false` boolean literals.

Keywords are the obvious ones like `function` and `switch`. Future reserved words include things like `enum`, though many of the rest of them (`class`, `extends`, etc.) are all now actually used by ES6; there are other strict-mode only reserved words like `interface`.

StackOverflow user "art4theSould" creatively worked all these reserved words into a fun little poem (<http://stackoverflow.com/questions/26255/reserved-keywords-in-javascript/12114140#12114140>):

Let this long package float, Goto private class if short. While protected with debugger case, Continue volatile interface. Instanceof super synchronized throw, Extends final export throws.

Try import double enum?

- False, boolean, abstract function, Implements typeof transient break! Void static, default do, Switch int native new. Else, delete null public var In return for const, true, char ...Finally catch byte.

Note: This poem includes words that were reserved in ES3 (`byte` , `long` , etc.) that are no longer reserved as of ES5.

Prior to ES5, the reserved words also could not be property names or keys in object literals, but that restriction no longer exists.

So, this is not allowed:

```
var import = "42";
```

But this is allowed:

```
var obj = { import: "42" };
console.log( obj.import );
```

You should be aware though that some older browser versions (mainly older IE) weren't completely consistent on applying these rules, so there are places where using reserved words in object property name locations can still cause issues. Carefully test all supported browser environments.

Implementation Limits

The JavaScript spec does not place arbitrary limits on things such as the number of arguments to a function or the length of a string literal, but these limits exist nonetheless, because of implementation details in different engines.

For example:

```

function addAll() {
    var sum = 0;
    for (var i=0; i < arguments.length; i++) {
        sum += arguments[i];
    }
    return sum;
}

var nums = [];
for (var i=1; i < 100000; i++) {
    nums.push(i);
}

addAll( 2, 4, 6 );           // 12
addAll.apply( null, nums ); // should be: 499950000

```

In some JS engines, you'll get the correct `499950000` answer, but in others (like Safari 6.x), you'll get the error: "RangeError: Maximum call stack size exceeded."

Examples of other limits known to exist:

- maximum number of characters allowed in a string literal (not just a string value)
- size (bytes) of data that can be sent in arguments to a function call (aka stack size)
- number of parameters in a function declaration
- maximum depth of non-optimized call stack (i.e., with recursion): how long a chain of function calls from one to the other can be
- number of seconds a JS program can run continuously blocking the browser
- maximum length allowed for a variable name
- ...

It's not very common at all to run into these limits, but you should be aware that limits can and do exist, and importantly that they vary between engines.

Review

We know and can rely upon the fact that the JS language itself has one standard and is predictably implemented by all the modern browsers/engines. This is a very good thing!

But JavaScript rarely runs in isolation. It runs in an environment mixed in with code from third-party libraries, and sometimes it even runs in engines/environments that differ from those found in browsers.

Paying close attention to these issues improves the reliability and robustness of your code.

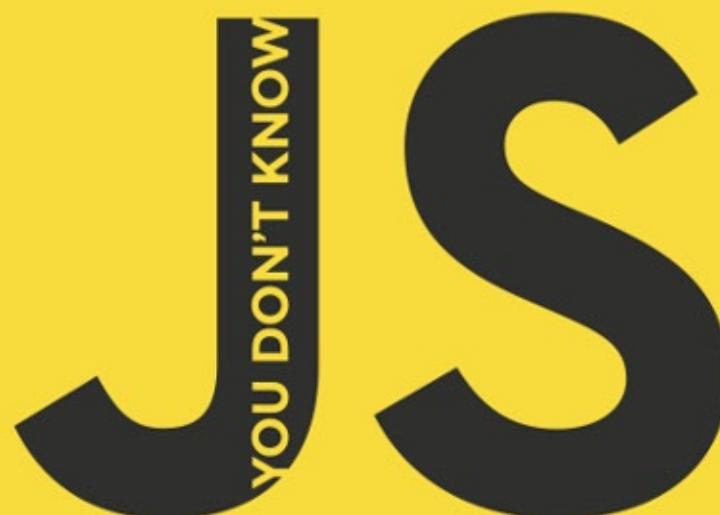
O'REILLY®

"The `this` keyword and prototypes are pivotal, because they
are foundational to doing real programming with JavaScript."

-NICK BERARDI, Senior Consultant, RDA Corporation

KYLE SIMPSON

this & OBJECT PROTOTYPES



Chapter 1: `this` Or That?

One of the most confused mechanisms in JavaScript is the `this` keyword. It's a special identifier keyword that's automatically defined in the scope of every function, but what exactly it refers to bedevils even seasoned JavaScript developers.

Any sufficiently *advanced* technology is indistinguishable from magic. -- Arthur C. Clarke

JavaScript's `this` mechanism isn't actually *that* advanced, but developers often paraphrase that quote in their own mind by inserting "complex" or "confusing", and there's no question that without lack of clear understanding, `this` can seem downright magical in *your* confusion.

Note: The word "this" is a terribly common pronoun in general discourse. So, it can be very difficult, especially verbally, to determine whether we are using "this" as a pronoun or using it to refer to the actual keyword identifier. For clarity, I will always use `this` to refer to the special keyword, and "this" or *this* or this otherwise.

Why `this` ?

If the `this` mechanism is so confusing, even to seasoned JavaScript developers, one may wonder why it's even useful? Is it more trouble than it's worth? Before we jump into the *how*, we should examine the *why*.

Let's try to illustrate the motivation and utility of `this` :

```
function identify() {
    return this.name.toUpperCase();
}

function speak() {
    var greeting = "Hello, I'm " + identify.call( this );
    console.log( greeting );
}

var me = {
    name: "Kyle"
};

var you = {
    name: "Reader"
};

identify.call( me ); // KYLE
identify.call( you ); // READER

speak.call( me ); // Hello, I'm KYLE
speak.call( you ); // Hello, I'm READER
```

If the *how* of this snippet confuses you, don't worry! We'll get to that shortly. Just set those questions aside briefly so we can look into the *why* more clearly.

This code snippet allows the `identify()` and `speak()` functions to be re-used against multiple `context` (`me` and `you`) objects, rather than needing a separate version of the function for each object.

Instead of relying on `this`, you could have explicitly passed in a context object to both `identify()` and `speak()`.

```
function identify(context) {
    return context.name.toUpperCase();
}

function speak(context) {
    var greeting = "Hello, I'm " + identify( context );
    console.log( greeting );
}

identify( you ); // READER
speak( me ); // Hello, I'm KYLE
```

However, the `this` mechanism provides a more elegant way of implicitly "passing along" an object reference, leading to cleaner API design and easier re-use.

The more complex your usage pattern is, the more clearly you'll see that passing context around as an explicit parameter is often messier than passing around a `this` context. When we explore objects and prototypes, you will see the helpfulness of a collection of functions being able to automatically reference the proper context object.

Confusions

We'll soon begin to explain how `this` actually works, but first we must dispel some misconceptions about how it *doesn't* actually work.

The name "this" creates confusion when developers try to think about it too literally. There are two meanings often assumed, but both are incorrect.

Itself

The first common temptation is to assume `this` refers to the function itself. That's a reasonable grammatical inference, at least.

Why would you want to refer to a function from inside itself? The most common reasons would be things like recursion (calling a function from inside itself) or having an event handler that can unbind itself when it's first called.

Developers new to JS's mechanisms often think that referencing the function as an object (all functions in JavaScript are objects!) lets you store *state* (values in properties) between function calls. While this is certainly possible and has some limited uses, the rest of the book will expound on many other patterns for *better* places to store state besides the function object.

But for just a moment, we'll explore that pattern, to illustrate how `this` doesn't let a function get a reference to itself like we might have assumed.

Consider the following code, where we attempt to track how many times a function (`foo`) was called:

```
function foo(num) {
    console.log("foo: " + num);

    // keep track of how many times `foo` is called
    this.count++;
}

foo.count = 0;

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        foo( i );
    }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// how many times was `foo` called?
console.log( foo.count ); // 0 -- WTF?
```

`foo.count` is *still* `0`, even though the four `console.log` statements clearly indicate `foo(...)` was in fact called four times. The frustration stems from a *too literal* interpretation of what `this` (in `this.count++`) means.

When the code executes `foo.count = 0`, indeed it's adding a property `count` to the function object `foo`. But for the `this.count` reference inside of the function, `this` is not in fact pointing *at all* to that function object, and so even though the property names are the same, the root objects are different, and confusion ensues.

Note: A responsible developer *should* ask at this point, "If I was incrementing a `count` property but it wasn't the one I expected, which `count` was I incrementing?" In fact, were she to dig deeper, she would find that she had accidentally created a global variable `count` (see Chapter 2 for *how that happened!*), and it currently has the value `NaN`. Of course, once she identifies this peculiar outcome, she then has a whole other set of questions: "How was it global, and why did it end up `NaN` instead of some proper count value?" (see Chapter 2).

Instead of stopping at this point and digging into why the `this` reference doesn't seem to be behaving as *expected*, and answering those tough but important questions, many developers simply avoid the issue altogether, and hack toward some other solution, such as creating another object to hold the `count` property:

```

function foo(num) {
  console.log("foo: " + num);

  // keep track of how many times `foo` is called
  data.count++;
}

var data = {
  count: 0
};

var i;

for (i=0; i<10; i++) {
  if (i > 5) {
    foo( i );
  }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// how many times was `foo` called?
console.log( data.count ); // 4

```

While it is true that this approach "solves" the problem, unfortunately it simply ignores the real problem -- lack of understanding what `this` means and how it works -- and instead falls back to the comfort zone of a more familiar mechanism: lexical scope.

Note: Lexical scope is a perfectly fine and useful mechanism; I am not belittling the use of it, by any means (see "Scope & Closures" title of this book series). But constantly *guessing* at how to use `this`, and usually being *wrong*, is not a good reason to retreat back to lexical scope and never learn *why* `this` eludes you.

To reference a function object from inside itself, `this` by itself will typically be insufficient. You generally need a reference to the function object via a lexical identifier (variable) that points at it.

Consider these two functions:

```
function foo() {
    foo.count = 4; // `foo` refers to itself
}

setTimeout( function(){
    // anonymous function (no name), cannot
    // refer to itself
}, 10 );
```

In the first function, called a "named function", `foo` is a reference that can be used to refer to the function from inside itself.

But in the second example, the function callback passed to `setTimeout(...)` has no name identifier (so called an "anonymous function"), so there's no proper way to refer to the function object itself.

Note: The old-school but now deprecated and frowned-upon `argumentscallee` reference inside a function *also* points to the function object of the currently executing function. This reference is typically the only way to access an anonymous function's object from inside itself. The best approach, however, is to avoid the use of anonymous functions altogether, at least for those which require a self-reference, and instead use a named function (expression). `argumentscallee` is deprecated and should not be used.

So another solution to our running example would have been to use the `foo` identifier as a function object reference in each place, and not use `this` at all, which *works*:

```
function foo(num) {
  console.log("foo: " + num);

  // keep track of how many times `foo` is called
  foo.count++;
}

foo.count = 0;

var i;

for (i=0; i<10; i++) {
  if (i > 5) {
    foo( i );
  }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// how many times was `foo` called?
console.log( foo.count ); // 4
```

However, that approach similarly side-steps *actual* understanding of `this` and relies entirely on the lexical scoping of variable `foo`.

Yet another way of approaching the issue is to force `this` to actually point at the `foo` function object:

```

function foo(num) {
  console.log("foo: " + num);

  // keep track of how many times `foo` is called
  // Note: `this` IS actually `foo` now, based on
  // how `foo` is called (see below)
  this.count++;
}

foo.count = 0;

var i;

for (i=0; i<10; i++) {
  if (i > 5) {
    // using `call(..)`, we ensure the `this`
    // points at the function object (`foo`) itself
    foo.call( foo, i );
  }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// how many times was `foo` called?
console.log( foo.count ); // 4

```

Instead of avoiding `this`, we embrace it. We'll explain in a little bit *how* such techniques work much more completely, so don't worry if you're still a bit confused!

Its Scope

The next most common misconception about the meaning of `this` is that it somehow refers to the function's scope. It's a tricky question, because in one sense there is some truth, but in the other sense, it's quite misguided.

To be clear, `this` does not, in any way, refer to a function's **lexical scope**. It is true that internally, scope is kind of like an object with properties for each of the available identifiers. But the scope "object" is not accessible to JavaScript code. It's an inner part of the *Engine's* implementation.

Consider code which attempts (and fails!) to cross over the boundary and use `this` to implicitly refer to a function's lexical scope:

```

function foo() {
  var a = 2;
  this.bar();
}

function bar() {
  console.log( this.a );
}

foo(); //undefined

```

There's more than one mistake in this snippet. While it may seem contrived, the code you see is a distillation of actual real-world code that has been exchanged in public community help forums. It's a wonderful (if not sad) illustration of just how misguided `this` assumptions can be.

Firstly, an attempt is made to reference the `bar()` function via `this.bar()`. It is almost certainly an *accident* that it works, but we'll explain the *how* of that shortly. The most natural way to have invoked `bar()` would have been to omit the leading `this.` and just make a lexical reference to the identifier.

However, the developer who writes such code is attempting to use `this` to create a bridge between the lexical scopes of `foo()` and `bar()`, so that `bar()` has access to the variable `a` in the inner scope of `foo()`. **No such bridge is possible.** You cannot use a `this` reference to look something up in a lexical scope. It is not possible.

Every time you feel yourself trying to mix lexical scope look-ups with `this`, remind yourself: *there is no bridge*.

What's `this` ?

Having set aside various incorrect assumptions, let us now turn our attention to how the `this` mechanism really works.

We said earlier that `this` is not an author-time binding but a runtime binding. It is contextual based on the conditions of the function's invocation. `this` binding has nothing to do with where a function is declared, but has instead everything to do with the manner in which the function is called.

When a function is invoked, an activation record, otherwise known as an execution context, is created. This record contains information about where the function was called from (the call-stack), *how* the function was invoked, what parameters were passed, etc. One of the

properties of this record is the `this` reference which will be used for the duration of that function's execution.

In the next chapter, we will learn to find a function's **call-site** to determine how its execution will bind `this`.

Review (TL;DR)

`this` binding is a constant source of confusion for the JavaScript developer who does not take the time to learn how the mechanism actually works. Guesses, trial-and-error, and blind copy-n-paste from Stack Overflow answers is not an effective or proper way to leverage *this* important `this` mechanism.

To learn `this`, you first have to learn what `this` is *not*, despite any assumptions or misconceptions that may lead you down those paths. `this` is neither a reference to the function itself, nor is it a reference to the function's *lexical* scope.

`this` is actually a binding that is made when a function is invoked, and *what* it references is determined entirely by the call-site where the function is called.

Chapter 2: this All Makes Sense Now!

In Chapter 1, we discarded various misconceptions about `this` and learned instead that `this` is a binding made for each function invocation, based entirely on its **call-site** (how the function is called).

Call-site

To understand `this` binding, we have to understand the call-site: the location in code where a function is called (**not where it's declared**). We must inspect the call-site to answer the question: what's `this` `this` a reference to?

Finding the call-site is generally: "go locate where a function is called from", but it's not always that easy, as certain coding patterns can obscure the *true* call-site.

What's important is to think about the **call-stack** (the stack of functions that have been called to get us to the current moment in execution). The call-site we care about is *in* the invocation *before* the currently executing function.

Let's demonstrate call-stack and call-site:

```

function baz() {
  // call-stack is: `baz`
  // so, our call-site is in the global scope

  console.log( "baz" );
  bar(); // <-- call-site for `bar`
}

function bar() {
  // call-stack is: `baz` -> `bar`
  // so, our call-site is in `baz`

  console.log( "bar" );
  foo(); // <-- call-site for `foo`
}

function foo() {
  // call-stack is: `baz` -> `bar` -> `foo`
  // so, our call-site is in `bar`

  console.log( "foo" );
}

baz(); // <-- call-site for `baz`

```

Take care when analyzing code to find the actual call-site (from the call-stack), because it's the only thing that matters for `this` binding.

Note: You can visualize a call-stack in your mind by looking at the chain of function calls in order, as we did with the comments in the above snippet. But this is painstaking and error-prone. Another way of seeing the call-stack is using a debugger tool in your browser. Most modern desktop browsers have built-in developer tools, which includes a JS debugger. In the above snippet, you could have set a breakpoint in the tools for the first line of the `foo()` function, or simply inserted the `debugger;` statement on that first line. When you run the page, the debugger will pause at this location, and will show you a list of the functions that have been called to get to that line, which will be your call stack. So, if you're trying to diagnose `this` binding, use the developer tools to get the call-stack, then find the second item from the top, and that will show you the real call-site.

Nothing But Rules

We turn our attention now to *how* the call-site determines where `this` will point during the execution of a function.

You must inspect the call-site and determine which of 4 rules applies. We will first explain each of these 4 rules independently, and then we will illustrate their order of precedence, if multiple rules *could* apply to the call-site.

Default Binding

The first rule we will examine comes from the most common case of function calls: standalone function invocation. Think of `this` as the default catch-all rule when none of the other rules apply.

Consider this code:

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

var a = 2;

foo(); // 2
```

The first thing to note, if you were not already aware, is that variables declared in the global scope, as `var a = 2` is, are synonymous with global-object properties of the same name. They're not copies of each other, they *are* each other. Think of it as two sides of the same coin.

Secondly, we see that when `foo()` is called, `this.a` resolves to our global variable `a`. Why? Because in this case, the *default binding* for `this` applies to the function call, and so points `this` at the global object.

How do we know that the *default binding* rule applies here? We examine the call-site to see how `foo()` is called. In our snippet, `foo()` is called with a plain, un-decorated function reference. None of the other rules we will demonstrate will apply here, so the *default binding* applies instead.

If `strict mode` is in effect, the global object is not eligible for the *default binding*, so the `this` is instead set to `undefined`.

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

  console.log( this.a );
}

var a = 2;

foo(); // TypeError: `this` is `undefined`
```

A subtle but important detail is: even though the overall `this` binding rules are entirely based on the call-site, the global object is **only** eligible for the *default binding* if the **contents** of `foo()` are **not** running in `strict mode`; the `strict mode` state of the call-site of `foo()` is irrelevant.

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

var a = 2;

(function(){
  "use strict";

  foo(); // 2
})();
```

Note: Intentionally mixing `strict mode` and `non-strict mode` together in your own code is generally frowned upon. Your entire program should probably either be **Strict** or **non-Strict**. However, sometimes you include a third-party library that has different **Strict**'ness than your own code, so care must be taken over these subtle compatibility details.

Implicit Binding

Another rule to consider is: does the call-site have a context object, also referred to as an owning or containing object, though *these* alternate terms could be slightly misleading.

Consider:

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

```
var obj = {  
    a: 2,  
    foo: foo  
};
```

```
obj.foo(); // 2
```

Firstly, notice the manner in which `foo()` is declared and then later added as a reference property onto `obj`. Regardless of whether `foo()` is initially declared on `obj`, or is added as a reference later (as this snippet shows), in neither case is the **function** really "owned" or "contained" by the `obj` object.

However, the call-site *uses* the `obj` context to **reference** the function, so you *could* say that the `obj` object "owns" or "contains" the **function reference** at the time the function is called.

Whatever you choose to call this pattern, at the point that `foo()` is called, it's preceded by an object reference to `obj`. When there is a context object for a function reference, the *implicit binding* rule says that it's *that* object which should be used for the function call's `this` binding.

Because `obj` is the `this` for the `foo()` call, `this.a` is synonymous with `obj.a`.

Only the top/last level of an object property reference chain matters to the call-site. For instance:

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

```
var obj2 = {  
    a: 42,  
    foo: foo  
};
```

```
var obj1 = {  
    a: 2,  
    obj2: obj2  
};
```

```
obj1.obj2.foo(); // 42
```

Implicitly Lost

One of the most common frustrations that `this` binding creates is when an *implicitly bound* function loses that binding, which usually means it falls back to the *default binding*, of either the global object or `undefined`, depending on `strict mode`.

Consider:

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

var obj = {
  a: 2,
  foo: foo
};

var bar = obj.foo; // function reference/alias!

var a = "oops, global"; // `a` also property on global object

bar(); // "oops, global"
```

Even though `bar` appears to be a reference to `obj.foo`, in fact, it's really just another reference to `foo` itself. Moreover, the call-site is what matters, and the call-site is `bar()`, which is a plain, un-decorated call and thus the *default binding* applies.

The more subtle, more common, and more unexpected way this occurs is when we consider passing a callback function:

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

function doFoo(fn) {
  // `fn` is just another reference to `foo`

  fn(); // <-- call-site!
}

var obj = {
  a: 2,
  foo: foo
};

var a = "oops, global"; // `a` also property on global object

doFoo( obj.foo ); // "oops, global"
```

Parameter passing is just an implicit assignment, and since we're passing a function, it's an implicit reference assignment, so the end result is the same as the previous snippet.

What if the function you're passing your callback to is not your own, but built-in to the language? No difference, same outcome.

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

var obj = {
  a: 2,
  foo: foo
};

var a = "oops, global"; // `a` also property on global object

setTimeout( obj.foo, 100 ); // "oops, global"
```

Think about this crude theoretical pseudo-implementation of `setTimeout()` provided as a built-in from the JavaScript environment:

```
function setTimeout(fn, delay) {
  // wait (somehow) for `delay` milliseconds
  fn(); // <-- call-site!
}
```

It's quite common that our function callbacks *lose* their `this` binding, as we've just seen. But another way that `this` can surprise us is when the function we've passed our callback to intentionally changes the `this` for the call. Event handlers in popular JavaScript libraries are quite fond of forcing your callback to have a `this` which points to, for instance, the DOM element that triggered the event. While that may sometimes be useful, other times it can be downright infuriating. Unfortunately, these tools rarely let you choose.

Either way the `this` is changed unexpectedly, you are not really in control of how your callback function reference will be executed, so you have no way (yet) of controlling the call-site to give your intended binding. We'll see shortly a way of "fixing" that problem by *fixing* the `this`.

Explicit Binding

With *implicit binding* as we just saw, we had to mutate the object in question to include a reference on itself to the function, and use this property function reference to indirectly (implicitly) bind `this` to the object.

But, what if you want to force a function call to use a particular object for the `this` binding, without putting a property function reference on the object?

"All" functions in the language have some utilities available to them (via their `[[Prototype]]` -- more on that later) which can be useful for this task. Specifically, functions have `call(..)` and `apply(..)` methods. Technically, JavaScript host environments sometimes provide functions which are special enough (a kind way of putting it!) that they do not have such functionality. But those are few. The vast majority of functions provided, and certainly all functions you will create, do have access to `call(..)` and `apply(..)`.

How do these utilities work? They both take, as their first parameter, an object to use for the `this`, and then invoke the function with that `this` specified. Since you are directly stating what you want the `this` to be, we call it *explicit binding*.

Consider:

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

var obj = {
  a: 2
};

foo.call( obj ); // 2
```

Invoking `foo` with *explicit binding* by `foo.call(..)` allows us to force its `this` to be `obj`.

If you pass a simple primitive value (of type `string`, `boolean`, or `number`) as the `this` binding, the primitive value is wrapped in its object-form (`new String(..)`, `new Boolean(..)`, or `new Number(..)`, respectively). This is often referred to as "boxing".

Note: With respect to `this` binding, `call(..)` and `apply(..)` are identical. They *do* behave differently with their additional parameters, but that's not something we care about presently.

Unfortunately, *explicit binding* alone still doesn't offer any solution to the issue mentioned previously, of a function "losing" its intended `this` binding, or just having it paved over by a framework, etc.

Hard Binding

But a variation pattern around *explicit binding* actually does the trick. Consider:

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

var obj = {
    a: 2
};

var bar = function() {
    foo.call( obj );
};

bar(); // 2
setTimeout( bar, 100 ); // 2

// `bar` hard binds `foo`'s `this` to `obj`
// so that it cannot be overriden
bar.call( window ); // 2
```

Let's examine how this variation works. We create a function `bar()` which, internally, manually calls `foo.call(obj)`, thereby forcibly invoking `foo` with `obj` binding for `this`. No matter how you later invoke the function `bar`, it will always manually invoke `foo` with `obj`. This binding is both explicit and strong, so we call it *hard binding*.

The most typical way to wrap a function with a *hard binding* creates a pass-thru of any arguments passed and any return value received:

```
function foo(something) {
    console.log( this.a, something );
    return this.a + something;
}

var obj = {
    a: 2
};

var bar = function() {
    return foo.apply( obj, arguments );
};

var b = bar( 3 ); // 2 3
console.log( b ); // 5
```

Another way to express this pattern is to create a re-usable helper:

```
function foo(something) {
  console.log( this.a, something );
  return this.a + something;
}

// simple `bind` helper
function bind(fn, obj) {
  return function() {
    return fn.apply( obj, arguments );
  };
}

var obj = {
  a: 2
};

var bar = bind( foo, obj );

var b = bar( 3 ); // 2 3
console.log( b ); // 5
```

Since *hard binding* is such a common pattern, it's provided with a built-in utility as of ES5:

`Function.prototype.bind`, and it's used like this:

```
function foo(something) {
  console.log( this.a, something );
  return this.a + something;
}

var obj = {
  a: 2
};

var bar = foo.bind( obj );

var b = bar( 3 ); // 2 3
console.log( b ); // 5
```

`bind(...)` returns a new function that is hard-coded to call the original function with the `this` context set as you specified.

Note: As of ES6, the hard-bound function produced by `bind(...)` has a `.name` property that derives from the original *target function*. For example: `bar = foo.bind(..)` should have a `bar.name` value of `"bound foo"`, which is the function call name that should show up in a stack trace.

API Call "Contexts"

Many libraries' functions, and indeed many new built-in functions in the JavaScript language and host environment, provide an optional parameter, usually called "context", which is designed as a work-around for you not having to use `bind(..)` to ensure your callback function uses a particular `this`.

For instance:

```
function foo(el) {
  console.log( el, this.id );
}

var obj = {
  id: "awesome"
};

// use `obj` as `this` for `foo(..)` calls
[1, 2, 3].forEach( foo, obj ); // 1 awesome 2 awesome 3 awesome
```

Internally, these various functions almost certainly use *explicit binding* via `call(..)` or `apply(..)`, saving you the trouble.

new Binding

The fourth and final rule for `this` binding requires us to re-think a very common misconception about functions and objects in JavaScript.

In traditional class-oriented languages, "constructors" are special methods attached to classes, that when the class is instantiated with a `new` operator, the constructor of that class is called. This usually looks something like:

```
something = new MyClass(..);
```

JavaScript has a `new` operator, and the code pattern to use it looks basically identical to what we see in those class-oriented languages; most developers assume that JavaScript's mechanism is doing something similar. However, there really is *no connection* to class-oriented functionality implied by `new` usage in JS.

First, let's re-define what a "constructor" in JavaScript is. In JS, constructors are **just functions** that happen to be called with the `new` operator in front of them. They are not attached to classes, nor are they instantiating a class. They are not even special types of functions. They're just regular functions that are, in essence, hijacked by the use of `new` in their invocation.

For example, the `Number(..)` function acting as a constructor, quoting from the ES5.1 spec:

15.7.2 The Number Constructor

When `Number` is called as part of a new expression it is a constructor: it initialises the newly created object.

So, pretty much any ol' function, including the built-in object functions like `Number(..)` (see Chapter 3) can be called with `new` in front of it, and that makes that function call a *constructor call*. This is an important but subtle distinction: there's really no such thing as "constructor functions", but rather construction calls *of* functions.

When a function is invoked with `new` in front of it, otherwise known as a constructor call, the following things are done automatically:

1. a brand new object is created (aka, constructed) out of thin air
2. *the newly constructed object is `[[Prototype]]`-linked*
3. the newly constructed object is set as the `this` binding for that function call
4. unless the function returns its own alternate **object**, the `new`-invoked function call will *automatically* return the newly constructed object.

Steps 1, 3, and 4 apply to our current discussion. We'll skip over step 2 for now and come back to it in Chapter 5.

Consider this code:

```
function foo(a) {  
    this.a = a;  
}  
  
var bar = new foo( 2 );  
console.log( bar.a ); // 2
```

By calling `foo(..)` with `new` in front of it, we've constructed a new object and set that new object as the `this` for the call of `foo(..)`. **So `new` is the final way that a function call's `this` can be bound.** We'll call this *new binding*.

Everything In Order

So, now we've uncovered the 4 rules for binding `this` in function calls. *All* you need to do is find the call-site and inspect it to see which rule applies. But, what if the call-site has multiple eligible rules? There must be an order of precedence to these rules, and so we will next demonstrate what order to apply the rules.

It should be clear that the *default binding* is the lowest priority rule of the 4. So we'll just set that one aside.

Which is more precedent, *implicit binding* or *explicit binding*? Let's test it:

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

var obj1 = {
  a: 2,
  foo: foo
};

var obj2 = {
  a: 3,
  foo: foo
};

obj1.foo(); // 2
obj2.foo(); // 3

obj1.foo.call( obj2 ); // 3
obj2.foo.call( obj1 ); // 2
```

So, *explicit binding* takes precedence over *implicit binding*, which means you should ask **first** if *explicit binding* applies before checking for *implicit binding*.

Now, we just need to figure out where *new binding* fits in the precedence.

```
function foo(something) {
  this.a = something;
}

var obj1 = {
  foo: foo
};

var obj2 = {};

obj1.foo( 2 );
console.log( obj1.a ); // 2

obj1.foo.call( obj2, 3 );
console.log( obj2.a ); // 3

var bar = new obj1.foo( 4 );
console.log( obj1.a ); // 2
console.log( bar.a ); // 4
```

OK, *new binding* is more precedent than *implicit binding*. But do you think *new binding* is more or less precedent than *explicit binding*?

Note: `new` and `call / apply` cannot be used together, so `new foo.call(obj1)` is not allowed, to test *new binding* directly against *explicit binding*. But we can still use a *hard binding* to test the precedence of the two rules.

Before we explore that in a code listing, think back to how *hard binding* physically works, which is that `Function.prototype.bind(...)` creates a new wrapper function that is hard-coded to ignore its own `this` binding (whatever it may be), and use a manual one we provide.

By that reasoning, it would seem obvious to assume that *hard binding* (which is a form of *explicit binding*) is more precedent than *new binding*, and thus cannot be overridden with `new`.

Let's check:

```
function foo(something) {
  this.a = something;
}

var obj1 = {};

var bar = foo.bind( obj1 );
bar( 2 );
console.log( obj1.a ); // 2

var baz = new bar( 3 );
console.log( obj1.a ); // 2
console.log( baz.a ); // 3
```

Whoa! `bar` is hard-bound against `obj1`, but `new bar(3)` did not change `obj1.a` to be `3` as we would have expected. Instead, the *hard bound* (to `obj1`) call to `bar(..)` is able to be overridden with `new`. Since `new` was applied, we got the newly created object back, which we named `baz`, and we see in fact that `baz.a` has the value `3`.

This should be surprising if you go back to our "fake" bind helper:

```
function bind(fn, obj) {
  return function() {
    fn.apply( obj, arguments );
  };
}
```

If you reason about how the helper's code works, it does not have a way for a `new` operator call to override the hard-binding to `obj` as we just observed.

But the built-in `Function.prototype.bind(..)` as of ES5 is more sophisticated, quite a bit so in fact. Here is the (slightly reformatted) polyfill provided by the MDN page for `bind(..)`:

```
if (!Function.prototype.bind) {
  Function.prototype.bind = function(oThis) {
    if (typeof this !== "function") {
      // closest thing possible to the ECMAScript 5
      // internal IsCallable function
      throw new TypeError( "Function.prototype.bind - what " +
        "is trying to be bound is not callable"
      );
    }

    var aArgs = Array.prototype.slice.call( arguments, 1 ),
        fToBind = this,
        fNOP = function(){},
        fBound = function(){
          return fToBind.apply(
            (
              this instanceof fNOP &&
              oThis ? this : oThis
            ),
            aArgs.concat( Array.prototype.slice.call( arguments ) )
          );
        }

        fNOP.prototype = this.prototype;
        fBound.prototype = new fNOP();

        return fBound;
  };
}
```

Note: The `bind(..)` polyfill shown above differs from the built-in `bind(..)` in ES5 with respect to hard-bound functions that will be used with `new` (see below for why that's useful). Because the polyfill cannot create a function without a `.prototype` as the built-in utility does, there's some nuanced indirection to approximate the same behavior. Tread carefully if you plan to use `new` with a hard-bound function and you rely on this polyfill.

The part that's allowing `new` overriding is:

```
this instanceof fNOP &&
oThis ? this : oThis

// ... and:

fNOP.prototype = this.prototype;
fBound.prototype = new fNOP();
```

We won't actually dive into explaining how this trickery works (it's complicated and beyond our scope here), but essentially the utility determines whether or not the hard-bound function has been called with `new` (resulting in a newly constructed object being its `this`), and if so, it uses *that* newly created `this` rather than the previously specified *hard binding* for `this`.

Why is `new` being able to override *hard binding* useful?

The primary reason for this behavior is to create a function (that can be used with `new` for constructing objects) that essentially ignores the `this` *hard binding* but which presets some or all of the function's arguments. One of the capabilities of `bind(..)` is that any arguments passed after the first `this` binding argument are defaulted as standard arguments to the underlying function (technically called "partial application", which is a subset of "currying").

For example:

```
function foo(p1,p2) {
  this.val = p1 + p2;
}

// using `null` here because we don't care about
// the `this` hard-binding in this scenario, and
// it will be overridden by the `new` call anyway!
var bar = foo.bind( null, "p1" );

var baz = new bar( "p2" );

baz.val; // p1p2
```

Determining `this`

Now, we can summarize the rules for determining `this` from a function call's call-site, in their order of precedence. Ask these questions in this order, and stop when the first rule applies.

1. Is the function called with `new` (**new binding**)? If so, `this` is the newly constructed object.

```
var bar = new foo()
```

2. Is the function called with `call` or `apply` (**explicit binding**), even hidden inside a `bind` *hard binding*? If so, `this` is the explicitly specified object.

```
var bar = foo.call( obj2 )
```

3. Is the function called with a context (**implicit binding**), otherwise known as an owning or containing object? If so, `this` is *that* context object.

```
var bar = obj1.foo()
```

4. Otherwise, default the `this` (**default binding**). If in `strict mode`, pick `undefined`, otherwise pick the `global` object.

```
var bar = foo()
```

That's it. That's *all it takes* to understand the rules of `this` binding for normal function calls. Well... almost.

Binding Exceptions

As usual, there are some *exceptions* to the "rules".

The `this`-binding behavior can in some scenarios be surprising, where you intended a different binding but you end up with binding behavior from the *default binding* rule (see previous).

Ignored `this`

If you pass `null` or `undefined` as a `this` binding parameter to `call`, `apply`, or `bind`, those values are effectively ignored, and instead the *default binding* rule applies to the invocation.

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

var a = 2;

foo.call( null ); // 2
```

Why would you intentionally pass something like `null` for a `this` binding?

It's quite common to use `apply(..)` for spreading out arrays of values as parameters to a function call. Similarly, `bind(..)` can curry parameters (pre-set values), which can be very helpful.

```

function foo(a,b) {
    console.log( "a:" + a + ", b:" + b );
}

// spreading out array as parameters
foo.apply( null, [2, 3] ); // a:2, b:3

// currying with `bind(..)`
var bar = foo.bind( null, 2 );
bar( 3 ); // a:2, b:3

```

Both these utilities require a `this` binding for the first parameter. If the functions in question don't care about `this`, you need a placeholder value, and `null` might seem like a reasonable choice as shown in this snippet.

Note: We don't cover it in this book, but ES6 has the `...` spread operator which will let you syntactically "spread out" an array as parameters without needing `apply(..)`, such as `foo(...[1,2])`, which amounts to `foo(1,2)` -- syntactically avoiding a `this` binding if it's unnecessary. Unfortunately, there's no ES6 syntactic substitute for currying, so the `this` parameter of the `bind(..)` call still needs attention.

However, there's a slight hidden "danger" in always using `null` when you don't care about the `this` binding. If you ever use that against a function call (for instance, a third-party library function that you don't control), and that function *does* make a `this` reference, the *default binding* rule means it might inadvertently reference (or worse, mutate!) the `global` object (`window` in the browser).

Obviously, such a pitfall can lead to a variety of *very difficult* to diagnose/track-down bugs.

Safer `this`

Perhaps a somewhat "safer" practice is to pass a specifically set up object for `this` which is guaranteed not to be an object that can create problematic side effects in your program. Borrowing terminology from networking (and the military), we can create a "DMZ" (de-militarized zone) object -- nothing more special than a completely empty, non-delegated (see Chapters 5 and 6) object.

If we always pass a DMZ object for ignored `this` bindings we don't think we need to care about, we're sure any hidden/unexpected usage of `this` will be restricted to the empty object, which insulates our program's `global` object from side-effects.

Since this object is totally empty, I personally like to give it the variable name `ø` (the lowercase mathematical symbol for the empty set). On many keyboards (like US-layout on Mac), this symbol is easily typed with `⌥ + ø` (option+ø). Some systems also let you set

up hotkeys for specific symbols. If you don't like the `ø` symbol, or your keyboard doesn't make that as easy to type, you can of course call it whatever you want.

Whatever you call it, the easiest way to set it up as **totally empty** is `Object.create(null)` (see Chapter 5). `Object.create(null)` is similar to `{ }`, but without the delegation to `Object.prototype`, so it's "more empty" than just `{ }`.

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

// our DMZ empty object
var ø = Object.create( null );

// spreading out array as parameters
foo.apply( ø, [2, 3] ); // a:2, b:3

// currying with `bind(..)`
var bar = foo.bind( ø, 2 );
bar( 3 ); // a:2, b:3
```

Not only functionally "safer", there's a sort of stylistic benefit to `ø`, in that it semantically conveys "I want the `this` to be empty" a little more clearly than `null` might. But again, name your DMZ object whatever you prefer.

Indirection

Another thing to be aware of is you can (intentionally or not!) create "indirect references" to functions, and in those cases, when that function reference is invoked, the *default binding* rule also applies.

One of the most common ways that *indirect references* occur is from an assignment:

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

var a = 2;
var o = { a: 3, foo: foo };
var p = { a: 4 };

o.foo(); // 3
(p.foo = o.foo)(); // 2
```

The *result value* of the assignment expression `p.foo = o.foo` is a reference to just the underlying function object. As such, the effective call-site is just `foo()`, not `p.foo()` or `o.foo()` as you might expect. Per the rules above, the *default binding* rule applies.

Reminder: regardless of how you get to a function invocation using the *default binding* rule, the `strict mode` status of the **contents** of the invoked function making the `this` reference -- not the function call-site -- determines the *default binding* value: either the `global` object if in non-`strict mode` or `undefined` if in `strict mode`.

Softening Binding

We saw earlier that *hard binding* was one strategy for preventing a function call falling back to the *default binding* rule inadvertently, by forcing it to be bound to a specific `this` (unless you use `new` to override it!). The problem is, *hard-binding* greatly reduces the flexibility of a function, preventing manual `this` override with either the *implicit binding* or even subsequent *explicit binding* attempts.

It would be nice if there was a way to provide a different default for *default binding* (not `global` or `undefined`), while still leaving the function able to be manually `this` bound via *implicit binding* or *explicit binding* techniques.

We can construct a so-called *soft binding* utility which emulates our desired behavior.

```
if (!Function.prototype.softBind) {
  Function.prototype.softBind = function(obj) {
    var fn = this,
      curried = [].slice.call( arguments, 1 ),
      bound = function bound() {
        return fn.apply(
          (!this ||
            (typeof window !== "undefined" &&
              this === window) ||
            (typeof global !== "undefined" &&
              this === global)
          ) ? obj : this,
          curried.concat.apply( curried, arguments )
        );
      };
    bound.prototype = Object.create( fn.prototype );
    return bound;
  };
}
```

The `softBind(...)` utility provided here works similarly to the built-in ES5 `bind(...)` utility, except with our *soft binding* behavior. It wraps the specified function in logic that checks the `this` at call-time and if it's `global` or `undefined`, uses a pre-specified alternate *default*

(`obj`). Otherwise the `this` is left untouched. It also provides optional currying (see the `bind(...)` discussion earlier).

Let's demonstrate its usage:

```
function foo() {
  console.log("name: " + this.name);
}

var obj = { name: "obj" },
    obj2 = { name: "obj2" },
    obj3 = { name: "obj3" };

var fooOBJ = foo.softBind( obj );

fooOBJ(); // name: obj

obj2.foo = foo.softBind(obj);
obj2.foo(); // name: obj2  <---- look!!!

fooOBJ.call( obj3 ); // name: obj3  <---- look!

setTimeout( obj2.foo, 10 ); // name: obj  <---- falls back to soft-binding
```

The soft-bound version of the `foo()` function can be manually `this`-bound to `obj2` or `obj3` as shown, but it falls back to `obj` if the *default binding* would otherwise apply.

Lexical `this`

Normal functions abide by the 4 rules we just covered. But ES6 introduces a special kind of function that does not use these rules: arrow-function.

Arrow-functions are signified not by the `function` keyword, but by the `=>` so called "fat arrow" operator. Instead of using the four standard `this` rules, arrow-functions adopt the `this` binding from the enclosing (function or global) scope.

Let's illustrate arrow-function lexical scope:

```
function foo() {
    // return an arrow function
    return (a) => {
        // `this` here is lexically adopted from `foo()`
        console.log( this.a );
    };
}

var obj1 = {
    a: 2
};

var obj2 = {
    a: 3
};

var bar = foo.call( obj1 );
bar.call( obj2 ); // 2, not 3!
```

The arrow-function created in `foo()` lexically captures whatever `foo()`'s `this` is at its call-time. Since `foo()` was `this`-bound to `obj1`, `bar` (a reference to the returned arrow-function) will also be `this`-bound to `obj1`. The lexical binding of an arrow-function cannot be overridden (even with `new`!).

The most common use-case will likely be in the use of callbacks, such as event handlers or timers:

```
function foo() {
    setTimeout(() => {
        // `this` here is lexically adopted from `foo()`
        console.log( this.a );
    }, 100);
}

var obj = {
    a: 2
};

foo.call( obj ); // 2
```

While arrow-functions provide an alternative to using `bind(..)` on a function to ensure its `this`, which can seem attractive, it's important to note that they essentially are disabling the traditional `this` mechanism in favor of more widely-understood lexical scoping. Pre-ES6, we already have a fairly common pattern for doing so, which is basically almost indistinguishable from the spirit of ES6 arrow-functions:

```
function foo() {
  var self = this; // lexical capture of `this`
  setTimeout( function(){
    console.log( self.a );
  }, 100 );
}

var obj = {
  a: 2
};

foo.call( obj ); // 2
```

While `self = this` and arrow-functions both seem like good "solutions" to not wanting to use `bind(...)`, they are essentially fleeing from `this` instead of understanding and embracing it.

If you find yourself writing `this`-style code, but most or all the time, you defeat the `this` mechanism with lexical `self = this` or arrow-function "tricks", perhaps you should either:

1. Use only lexical scope and forget the false pretense of `this`-style code.
2. Embrace `this`-style mechanisms completely, including using `bind(...)` where necessary, and try to avoid `self = this` and arrow-function "lexical this" tricks.

A program can effectively use both styles of code (lexical and `this`), but inside of the same function, and indeed for the same sorts of look-ups, mixing the two mechanisms is usually asking for harder-to-maintain code, and probably working too hard to be clever.

Review (TL;DR)

Determining the `this` binding for an executing function requires finding the direct call-site of that function. Once examined, four rules can be applied to the call-site, in *this* order of precedence:

1. Called with `new`? Use the newly constructed object.
2. Called with `call` or `apply` (or `bind`)? Use the specified object.
3. Called with a context object owning the call? Use that context object.
4. Default: `undefined` in `strict mode`, global object otherwise.

Be careful of accidental/unintentional invoking of the *default binding* rule. In cases where you want to "safely" ignore a `this` binding, a "DMZ" object like `o = Object.create(null)` is a good placeholder value that protects the `global` object from unintended side-effects.

Instead of the four standard binding rules, ES6 arrow-functions use lexical scoping for `this` binding, which means they adopt the `this` binding (whatever it is) from its enclosing function call. They are essentially a syntactic replacement of `self = this` in pre-ES6 coding.

Chapter 3: Objects

In Chapters 1 and 2, we explained how the `this` binding points to various objects depending on the call-site of the function invocation. But what exactly are objects, and why do we need to point to them? We will explore objects in detail in this chapter.

Syntax

Objects come in two forms: the declarative (literal) form, and the constructed form.

The literal syntax for an object looks like this:

```
var myObj = {  
    key: value  
    // ...  
};
```

The constructed form looks like this:

```
var myObj = new Object();  
myObj.key = value;
```

The constructed form and the literal form result in exactly the same sort of object. The only difference really is that you can add one or more key/value pairs to the literal declaration, whereas with constructed-form objects, you must add the properties one-by-one.

Note: It's extremely uncommon to use the "constructed form" for creating objects as just shown. You would pretty much always want to use the literal syntax form. The same will be true of most of the built-in objects (see below).

Type

Objects are the general building block upon which much of JS is built. They are one of the 6 primary types (called "language types" in the specification) in JS:

- `string`
- `number`
- `boolean`
- `null`

- `undefined`
- `object`

Note that the *simple primitives* (`string`, `number`, `boolean`, `null`, and `undefined`) are **not** themselves `objects`. `null` is sometimes referred to as an object type, but this misconception stems from a bug in the language which causes `typeof null` to return the string `"object"` incorrectly (and confusingly). In fact, `null` is its own primitive type.

It's a common mis-statement that "everything in JavaScript is an object". This is clearly not true.

By contrast, there *are* a few special object sub-types, which we can refer to as *complex primitives*.

`function` is a sub-type of object (technically, a "callable object"). Functions in JS are said to be "first class" in that they are basically just normal objects (with callable behavior semantics bolted on), and so they can be handled like any other plain object.

Arrays are also a form of objects, with extra behavior. The organization of contents in arrays is slightly more structured than for general objects.

Built-in Objects

There are several other object sub-types, usually referred to as built-in objects. For some of them, their names seem to imply they are directly related to their simple primitives counterparts, but in fact, their relationship is more complicated, which we'll explore shortly.

- `String`
- `Number`
- `Boolean`
- `Object`
- `Function`
- `Array`
- `Date`
- `RegExp`
- `Error`

These built-ins have the appearance of being actual types, even classes, if you rely on the similarity to other languages such as Java's `String` class.

But in JS, these are actually just built-in functions. Each of these built-in functions can be used as a constructor (that is, a function call with the `new` operator -- see Chapter 2), with the result being a newly *constructed* object of the sub-type in question. For instance:

```

var strPrimitive = "I am a string";
typeof strPrimitive;                                // "string"
strPrimitive instanceof String;                     // false

var strObject = new String("I am a string");
typeof strObject;                                  // "object"
strObject instanceof String;                      // true

// inspect the object sub-type
Object.prototype.toString.call(strObject);        // [object String]

```

We'll see in detail in a later chapter exactly how the `Object.prototype.toString...` bit works, but briefly, we can inspect the internal sub-type by borrowing the base default `toString()` method, and you can see it reveals that `strObject` is an object that was in fact created by the `String` constructor.

The primitive value `"I am a string"` is not an object, it's a primitive literal and immutable value. To perform operations on it, such as checking its length, accessing its individual character contents, etc, a `String` object is required.

Luckily, the language automatically coerces a `"string"` primitive to a `String` object when necessary, which means you almost never need to explicitly create the Object form. It is **strongly preferred** by the majority of the JS community to use the literal form for a value, where possible, rather than the constructed object form.

Consider:

```

var strPrimitive = "I am a string";

console.log(strPrimitive.length);                  // 13

console.log(strPrimitive.charAt(3));              // "m"

```

In both cases, we call a property or method on a string primitive, and the engine automatically coerces it to a `String` object, so that the property/method access works.

The same sort of coercion happens between the number literal primitive `42` and the `new Number(42)` object wrapper, when using methods like `42.359.toFixed(2)`. Likewise for `Boolean` objects from `"boolean"` primitives.

`null` and `undefined` have no object wrapper form, only their primitive values. By contrast, `Date` values can *only* be created with their constructed object form, as they have no literal form counter-part.

`Object`, `Array`, `Function`, and `RegExp` are all objects regardless of whether the literal or constructed form is used. The constructed form does offer, in some cases, more options in creation than the literal form counterpart. Since objects are created either way, the simpler literal form is almost universally preferred. **Only use the constructed form if you need the extra options.**

`Error` objects are rarely created explicitly in code, but usually created automatically when exceptions are thrown. They can be created with the constructed form `new Error(..)`, but it's often unnecessary.

Contents

As mentioned earlier, the contents of an object consist of values (any type) stored at specifically named *locations*, which we call properties.

It's important to note that while we say "contents" which implies that these values are *actually* stored inside the object, that's merely an appearance. The engine stores values in implementation-dependent ways, and may very well not store them *in* some object container. What *is* stored in the container are these property names, which act as pointers (technically, *references*) to where the values are stored.

Consider:

```
var myObject = {  
    a: 2  
};  
  
myObject.a;      // 2  
  
myObject["a"];   // 2
```

To access the value at the *location* `a` in `myObject`, we need to use either the `.` operator or the `[]` operator. The `.a` syntax is usually referred to as "property" access, whereas the `["a"]` syntax is usually referred to as "key" access. In reality, they both access the same *location*, and will pull out the same value, `2`, so the terms can be used interchangeably. We will use the most common term, "property access" from here on.

The main difference between the two syntaxes is that the `.` operator requires an `Identifier` compatible property name after it, whereas the `[".."]` syntax can take basically any UTF-8/unicode compatible string as the name for the property. To reference a property of the name "Super-Fun!", for instance, you would have to use the `["Super-Fun!"]` access syntax, as `Super-Fun!` is not a valid `Identifier` property name.

Also, since the `[...""]` syntax uses a string's **value** to specify the location, this means the program can programmatically build up the value of the string, such as:

```
var wantA = true;
var myObject = {
  a: 2
};

var idx;

if (wantA) {
  idx = "a";
}

// later

console.log( myObject[idx] ); // 2
```

In objects, property names are **always** strings. If you use any other value besides a `string` (primitive) as the property, it will first be converted to a string. This even includes numbers, which are commonly used as array indexes, so be careful not to confuse the use of numbers between objects and arrays.

```
var myObject = { };

myObject[true] = "foo";
myObject[3] = "bar";
myObject[myObject] = "baz";

myObject["true"];           // "foo"
myObject["3"];             // "bar"
myObject["[object Object]"]; // "baz"
```

Computed Property Names

The `myObject[...]` property access syntax we just described is useful if you need to use a computed expression value as the key name, like `myObject[prefix + name]`. But that's not really helpful when declaring objects using the object-literal syntax.

ES6 adds *computed property names*, where you can specify an expression, surrounded by a `[]` pair, in the key-name position of an object-literal declaration:

```
var prefix = "foo";

var myObject = {
  [prefix + "bar"]: "hello",
  [prefix + "baz"]: "world"
};

myObject["foobar"]; // hello
myObject["foobaz"]; // world
```

The most common usage of *computed property names* will probably be for ES6 `Symbol`s, which we will not be covering in detail in this book. In short, they're a new primitive data type which has an opaque unguessable value (technically a `string` value). You will be strongly discouraged from working with the *actual value* of a `Symbol` (which can theoretically be different between different JS engines), so the name of the `Symbol`, like `Symbol.Something` (just a made up name!), will be what you use:

```
var myObject = {
  [Symbol Something]: "hello world"
};
```

Property vs. Method

Some developers like to make a distinction when talking about a property access on an object, if the value being accessed happens to be a function. Because it's tempting to think of the function as *belonging* to the object, and in other languages, functions which belong to objects (aka, "classes") are referred to as "methods", it's not uncommon to hear, "method access" as opposed to "property access".

The specification makes this same distinction, interestingly.

Technically, functions never "belong" to objects, so saying that a function that just happens to be accessed on an object reference is automatically a "method" seems a bit of a stretch of semantics.

It *is* true that some functions have `this` references in them, and that *sometimes* these `this` references refer to the object reference at the call-site. But this usage really does not make that function any more a "method" than any other function, as `this` is dynamically bound at run-time, at the call-site, and thus its relationship to the object is indirect, at best.

Every time you access a property on an object, that is a **property access**, regardless of the type of value you get back. If you *happen* to get a function from that property access, it's not magically a "method" at that point. There's nothing special (outside of possible implicit `this` binding as explained earlier) about a function that comes from a property access.

For instance:

```
function foo() {
    console.log( "foo" );
}

var someFoo = foo;      // variable reference to `foo`

var myObject = {
    someFoo: foo
};

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

someFoo;              // function foo(){...}

myObject.someFoo;     // function foo(){...}
```

`someFoo` and `myObject.someFoo` are just two separate references to the same function, and neither implies anything about the function being special or "owned" by any other object. If `foo()` above was defined to have a `this` reference inside it, that `myObject.someFoo` *implicit binding* would be the **only** observable difference between the two references. Neither reference really makes sense to be called a "method".

Perhaps one could argue that a function *becomes a method*, not at definition time, but during run-time just for that invocation, depending on how it's called at its call-site (with an object reference context or not -- see Chapter 2 for more details). Even this interpretation is a bit of a stretch.

The safest conclusion is probably that "function" and "method" are interchangeable in JavaScript.

Note: ES6 adds a `super` reference, which is typically going to be used with `class` (see Appendix A). The way `super` behaves (static binding rather than late binding as `this`) gives further weight to the idea that a function which is `super` bound somewhere is more a "method" than "function". But again, these are just subtle semantic (and mechanical) nuances.

Even when you declare a function expression as part of the object-literal, that function doesn't magically *belong* more to the object -- still just multiple references to the same function object:

```

var myObject = {
  foo: function foo() {
    console.log( "foo" );
  }
};

var someFoo = myObject.foo;

someFoo;          // function foo(){...}

myObject.foo;     // function foo(){...}

```

Note: In Chapter 6, we will cover an ES6 short-hand for that `foo: function foo(){ ... }` declaration syntax in our object-literal.

Arrays

Arrays also use the `[]` access form, but as mentioned above, they have slightly more structured organization for how and where values are stored (though still no restriction on what *type* of values are stored). Arrays assume *numeric indexing*, which means that values are stored in locations, usually called *indices*, at non-negative integers, such as `0` and `42`.

```

var myArray = [ "foo", 42, "bar" ];

myArray.length;      // 3

myArray[0];          // "foo"

myArray[2];          // "bar"

```

Arrays are objects, so even though each index is a positive integer, you can *also* add properties onto the array:

```

var myArray = [ "foo", 42, "bar" ];

myArray.baz = "baz";

myArray.length;      // 3

myArray.baz;          // "baz"

```

Notice that adding named properties (regardless of `.` or `[]` operator syntax) does not change the reported `length` of the array.

You *could* use an array as a plain key/value object, and never add any numeric indices, but this is a bad idea because arrays have behavior and optimizations specific to their intended use, and likewise with plain objects. Use objects to store key/value pairs, and arrays to store values at numeric indices.

Be careful: If you try to add a property to an array, but the property name *looks* like a number, it will end up instead as a numeric index (thus modifying the array contents):

```
var myArray = [ "foo", 42, "bar" ];

myArray["3"] = "baz";

myArray.length; // 4

myArray[3]; // "baz"
```

Duplicating Objects

One of the most commonly requested features when developers newly take up the JavaScript language is how to duplicate an object. It would seem like there should just be a built-in `copy()` method, right? It turns out that it's a little more complicated than that, because it's not fully clear what, by default, should be the algorithm for the duplication.

For example, consider this object:

```
function anotherFunction() { /*...*/ }

var anotherObject = {
  c: true
};

var anotherArray = [];

var myObject = {
  a: 2,
  b: anotherObject, // reference, not a copy!
  c: anotherArray, // another reference!
  d: anotherFunction
};

anotherArray.push( anotherObject, myObject );
```

What exactly should be the representation of a *copy* of `myObject`?

Firstly, we should answer if it should be a *shallow* or *deep* copy. A *shallow copy* would end up with `a` on the new object as a copy of the value `2`, but `b`, `c`, and `d` properties as just references to the same places as the references in the original object. A *deep copy* would duplicate not only `myObject`, but `anotherObject` and `anotherArray`. But then we have issues that `anotherArray` has references to `anotherObject` and `myObject` in it, so those should also be duplicated rather than reference-preserved. Now we have an infinite circular duplication problem because of the circular reference.

Should we detect a circular reference and just break the circular traversal (leaving the deep element not fully duplicated)? Should we error out completely? Something in between?

Moreover, it's not really clear what "duplicating" a function would mean? There are some hacks like pulling out the `toString()` serialization of a function's source code (which varies across implementations and is not even reliable in all engines depending on the type of function being inspected).

So how do we resolve all these tricky questions? Various JS frameworks have each picked their own interpretations and made their own decisions. But which of these (if any) should JS adopt as *the standard*? For a long time, there was no clear answer.

One subset solution is that objects which are JSON-safe (that is, can be serialized to a JSON string and then re-parsed to an object with the same structure and values) can easily be *duplicated* with:

```
var newObj = JSON.parse( JSON.stringify( someObj ) );
```

Of course, that requires you to ensure your object is JSON safe. For some situations, that's trivial. For others, it's insufficient.

At the same time, a shallow copy is fairly understandable and has far less issues, so ES6 has now defined `Object.assign(...)` for this task. `Object.assign(...)` takes a *target* object as its first parameter, and one or more *source* objects as its subsequent parameters. It iterates over all the *enumerable* (see below), *owned keys (immediately present)* on the *source* object(s) and copies them (via `=` assignment only) to *target*. It also, helpfully, returns *target*, as you can see below:

```
var newObj = Object.assign( {}, myObject );

newObj.a;                      // 2
newObj.b === anotherObject;     // true
newObj.c === anotherArray;      // true
newObj.d === anotherFunction;   // true
```

Note: In the next section, we describe "property descriptors" (property characteristics) and show the use of `Object.defineProperty(..)`. The duplication that occurs for `Object.assign(..)` however is purely = style assignment, so any special characteristics of a property (like `writable`) on a source object **are not preserved** on the target object.

Property Descriptors

Prior to ES5, the JavaScript language gave no direct way for your code to inspect or draw any distinction between the characteristics of properties, such as whether the property was read-only or not.

But as of ES5, all properties are described in terms of a **property descriptor**.

Consider this code:

```
var myObject = {  
    a: 2  
};  
  
Object.getOwnPropertyDescriptor( myObject, "a" );  
// {  
//   value: 2,  
//   writable: true,  
//   enumerable: true,  
//   configurable: true  
// }
```

As you can see, the property descriptor (called a "data descriptor" since it's only for holding a data value) for our normal object property `a` is much more than just its `value` of `2`. It includes 3 other characteristics: `writable`, `enumerable`, and `configurable`.

While we can see what the default values for the property descriptor characteristics are when we create a normal property, we can use `Object.defineProperty(..)` to add a new property, or modify an existing one (if it's `configurable`!), with the desired characteristics.

For example:

```
var myObject = {};  
  
Object.defineProperty( myObject, "a", {  
    value: 2,  
    writable: true,  
    configurable: true,  
    enumerable: true  
} );  
  
myObject.a; // 2
```

Using `defineProperty(..)`, we added the plain, normal `a` property to `myObject` in a manually explicit way. However, you generally wouldn't use this manual approach unless you wanted to modify one of the descriptor characteristics from its normal behavior.

Writable

The ability for you to change the value of a property is controlled by `writable`.

Consider:

```
var myObject = {};  
  
Object.defineProperty( myObject, "a", {  
    value: 2,  
    writable: false, // not writable!  
    configurable: true,  
    enumerable: true  
} );  
  
myObject.a = 3;  
  
myObject.a; // 2
```

As you can see, our modification of the `value` silently failed. If we try in `strict mode`, we get an error:

```
"use strict";

var myObject = {};

Object.defineProperty( myObject, "a", {
    value: 2,
    writable: false, // not writable!
    configurable: true,
    enumerable: true
} );
myObject.a = 3; // TypeError
```

The `TypeError` tells us we cannot change a non-writable property.

Note: We will discuss getters/setters shortly, but briefly, you can observe that `writable:false` means a value cannot be changed, which is somewhat equivalent to if you defined a no-op setter. Actually, your no-op setter would need to throw a `TypeError` when called, to be truly conformant to `writable:false`.

Configurable

As long as a property is currently configurable, we can modify its descriptor definition, using the same `defineProperty(...)` utility.

```
var myObject = {  
    a: 2  
};  
  
myObject.a = 3;  
myObject.a; // 3  
  
Object.defineProperty( myObject, "a", {  
    value: 4,  
    writable: true,  
    configurable: false, // not configurable!  
    enumerable: true  
} );  
  
myObject.a; // 4  
myObject.a = 5;  
myObject.a; // 5  
  
Object.defineProperty( myObject, "a", {  
    value: 6,  
    writable: true,  
    configurable: true,  
    enumerable: true  
} ); // TypeError
```

The final `defineProperty(..)` call results in a `TypeError`, regardless of `strict mode`, if you attempt to change the descriptor definition of a non-configurable property. Be careful: as you can see, changing `configurable` to `false` is a **one-way action, and cannot be undone!**

Note: There's a nuanced exception to be aware of: even if the property is already `configurable:false`, `writable` can always be changed from `true` to `false` without error, but not back to `true` if already `false`.

Another thing `configurable:false` prevents is the ability to use the `delete` operator to remove an existing property.

```

var myObject = {
  a: 2
};

myObject.a;           // 2
delete myObject.a;
myObject.a;           // undefined

Object.defineProperty( myObject, "a", {
  value: 2,
  writable: true,
  configurable: false,
  enumerable: true
} );

myObject.a;           // 2
delete myObject.a;
myObject.a;           // 2

```

As you can see, the last `delete` call failed (silently) because we made the `a` property non-configurable.

`delete` is only used to remove object properties (which can be removed) directly from the object in question. If an object property is the last remaining *reference* to some object/function, and you `delete` it, that removes the reference and now that unreferenced object/function can be garbage collected. But, it is **not** proper to think of `delete` as a tool to free up allocated memory as it does in other languages (like C/C++). `delete` is just an object property removal operation -- nothing more.

Enumerable

The final descriptor characteristic we will mention here (there are two others, which we deal with shortly when we discuss getter/setters) is `enumerable`.

The name probably makes it obvious, but this characteristic controls if a property will show up in certain object-property enumerations, such as the `for...in` loop. Set to `false` to keep it from showing up in such enumerations, even though it's still completely accessible. Set to `true` to keep it present.

All normal user-defined properties are defaulted to `enumerable`, as this is most commonly what you want. But if you have a special property you want to hide from enumeration, set it to `enumerable:false`.

We'll demonstrate enumerability in much more detail shortly, so keep a mental bookmark on this topic.

Immutability

It is sometimes desired to make properties or objects that cannot be changed (either by accident or intentionally). ES5 adds support for handling that in a variety of different nuanced ways.

It's important to note that **all** of these approaches create shallow immutability. That is, they affect only the object and its direct property characteristics. If an object has a reference to another object (array, object, function, etc), the *contents* of that object are not affected, and remain mutable.

```
myImmutableObject.foo; // [1,2,3]
myImmutableObject.foo.push( 4 );
myImmutableObject.foo; // [1,2,3,4]
```

We assume in this snippet that `myImmutableObject` is already created and protected as immutable. But, to also protect the contents of `myImmutableObject.foo` (which is its own object -- array), you would also need to make `foo` immutable, using one or more of the following functionalities.

Note: It is not terribly common to create deeply entrenched immutable objects in JS programs. Special cases can certainly call for it, but as a general design pattern, if you find yourself wanting to *seal* or *freeze* all your objects, you may want to take a step back and reconsider your program design to be more robust to potential changes in objects' values.

Object Constant

By combining `writable:false` and `configurable:false`, you can essentially create a *constant* (cannot be changed, redefined or deleted) as an object property, like:

```
var myObject = {};

Object.defineProperty( myObject, "FAVORITE_NUMBER", {
    value: 42,
    writable: false,
    configurable: false
} );
```

Prevent Extensions

If you want to prevent an object from having new properties added to it, but otherwise leave the rest of the object's properties alone, call `Object.preventExtensions(..)`:

```
var myObject = {
  a: 2
};

Object.preventExtensions( myObject );

myObject.b = 3;
myObject.b; // undefined
```

In `non-strict mode`, the creation of `b` fails silently. In `strict mode`, it throws a `TypeError`.

Seal

`Object.seal(..)` creates a "sealed" object, which means it takes an existing object and essentially calls `Object.preventExtensions(..)` on it, but also marks all its existing properties as `configurable:false`.

So, not only can you not add any more properties, but you also cannot reconfigure or delete any existing properties (though you *can* still modify their values).

Freeze

`Object.freeze(..)` creates a frozen object, which means it takes an existing object and essentially calls `Object.seal(..)` on it, but it also marks all "data accessor" properties as `writable:false`, so that their values cannot be changed.

This approach is the highest level of immutability that you can attain for an object itself, as it prevents any changes to the object or to any of its direct properties (though, as mentioned above, the contents of any referenced other objects are unaffected).

You could "deep freeze" an object by calling `Object.freeze(..)` on the object, and then recursively iterating over all objects it references (which would have been unaffected thus far), and calling `Object.freeze(..)` on them as well. Be careful, though, as that could affect other (shared) objects you're not intending to affect.

[[Get]]

There's a subtle, but important, detail about how property accesses are performed.

Consider:

```
var myObject = {  
    a: 2  
};  
  
myObject.a; // 2
```

The `myObject.a` is a property access, but it doesn't *just* look in `myObject` for a property of the name `a`, as it might seem.

According to the spec, the code above actually performs a `[[Get]]` operation (kinda like a function call: `[[Get]]()`) on the `myObject`. The default built-in `[[Get]]` operation for an object *first* inspects the object for a property of the requested name, and if it finds it, it will return the value accordingly.

However, the `[[Get]]` algorithm defines other important behavior if it does *not* find a property of the requested name. We will examine in Chapter 5 what happens *next* (traversal of the `[[Prototype]]` chain, if any).

But one important result of this `[[Get]]` operation is that if it cannot through any means come up with a value for the requested property, it instead returns the value `undefined`.

```
var myObject = {  
    a: 2  
};  
  
myObject.b; // undefined
```

This behavior is different from when you reference *variables* by their identifier names. If you reference a variable that cannot be resolved within the applicable lexical scope look-up, the result is not `undefined` as it is for object properties, but instead a `ReferenceError` is thrown.

```
var myObject = {  
    a: undefined  
};  
  
myObject.a; // undefined  
  
myObject.b; // undefined
```

From a *value* perspective, there is no difference between these two references -- they both result in `undefined`. However, the `[[Get]]` operation underneath, though subtle at a glance, potentially performed a bit more "work" for the reference `myObject.b` than for the reference `myObject.a`.

Inspecting only the value results, you cannot distinguish whether a property exists and holds the explicit value `undefined`, or whether the property does *not* exist and `undefined` was the default return value after `[[Get]]` failed to return something explicitly. However, we will see shortly how you *can* distinguish these two scenarios.

[[Put]]

Since there's an internally defined `[[Get]]` operation for getting a value from a property, it should be obvious there's also a default `[[Put]]` operation.

It may be tempting to think that an assignment to a property on an object would just invoke `[[Put]]` to set or create that property on the object in question. But the situation is more nuanced than that.

When invoking `[[Put]]`, how it behaves differs based on a number of factors, including (most impactfully) whether the property is already present on the object or not.

If the property is present, the `[[Put]]` algorithm will roughly check:

1. Is the property an accessor descriptor (see "Getters & Setters" section below)? **If so, call the setter, if any.**
2. Is the property a data descriptor with `writable` of `false`? **If so, silently fail in non-strict mode, or throw `TypeError` in strict mode.**
3. Otherwise, set the value to the existing property as normal.

If the property is not yet present on the object in question, the `[[Put]]` operation is even more nuanced and complex. We will revisit this scenario in Chapter 5 when we discuss `[[Prototype]]` to give it more clarity.

Getters & Setters

The default `[[Put]]` and `[[Get]]` operations for objects completely control how values are set to existing or new properties, or retrieved from existing properties, respectively.

Note: Using future/advanced capabilities of the language, it may be possible to override the default `[[Get]]` or `[[Put]]` operations for an entire object (not just per property). This is beyond the scope of our discussion in this book, but will be covered later in the "You Don't Know JS" series.

ES5 introduced a way to override part of these default operations, not on an object level but a per-property level, through the use of getters and setters. Getters are properties which actually call a hidden function to retrieve a value. Setters are properties which actually call a hidden function to set a value.

When you define a property to have either a getter or a setter or both, its definition becomes an "accessor descriptor" (as opposed to a "data descriptor"). For accessor-descriptors, the `value` and `writable` characteristics of the descriptor are moot and ignored, and instead JS considers the `set` and `get` characteristics of the property (as well as `configurable` and `enumerable`).

Consider:

```
var myObject = {
    // define a getter for `a`
    get a() {
        return 2;
    }
};

Object.defineProperty(
    myObject,      // target
    "b",          // property name
    {             // descriptor
        // define a getter for `b`
        get: function(){ return this.a * 2 },
        // make sure `b` shows up as an object property
        enumerable: true
    }
);

myObject.a; // 2

myObject.b; // 4
```

Either through object-literal syntax with `get a() { .. }` or through explicit definition with `defineProperty(..)`, in both cases we created a property on the object that actually doesn't hold a value, but whose access automatically results in a hidden function call to the getter function, with whatever value it returns being the result of the property access.

```
var myObject = {
    // define a getter for `a`
    get a() {
        return 2;
    }
};

myObject.a = 3;

myObject.a; // 2
```

Since we only defined a getter for `a`, if we try to set the value of `a` later, the set operation won't throw an error but will just silently throw the assignment away. Even if there was a valid setter, our custom getter is hard-coded to return only `2`, so the set operation would be moot.

To make this scenario more sensible, properties should also be defined with setters, which override the default `[[Put]]` operation (aka, assignment), per-property, just as you'd expect. You will almost certainly want to always declare both getter and setter (having only one or the other often leads to unexpected/surprising behavior):

```
var myObject = {
    // define a getter for `a`
    get a() {
        return this._a_;
    },

    // define a setter for `a`
    set a(val) {
        this._a_ = val * 2;
    }
};

myObject.a = 2;

myObject.a; // 4
```

Note: In this example, we actually store the specified value `2` of the assignment (`[[Put]]` operation) into another variable `_a_`. The `_a_` name is purely by convention for this example and implies nothing special about its behavior -- it's a normal property like any other.

Existence

We showed earlier that a property access like `myobject.a` may result in an `undefined` value if either the explicit `undefined` is stored there or the `a` property doesn't exist at all. So, if the value is the same in both cases, how else do we distinguish them?

We can ask an object if it has a certain property *without* asking to get that property's value:

```
var myObject = {  
  a: 2  
};  
  
("a" in myObject);           // true  
("b" in myObject);           // false  
  
myObject.hasOwnProperty("a"); // true  
myObject.hasOwnProperty("b"); // false
```

The `in` operator will check to see if the property is *in* the object, or if it exists at any higher level of the `[[Prototype]]` chain object traversal (see Chapter 5). By contrast, `hasOwnProperty(..)` checks to see if *only* `myObject` has the property or not, and will *not* consult the `[[Prototype]]` chain. We'll come back to the important differences between these two operations in Chapter 5 when we explore `[[Prototype]]`s in detail.

`hasOwnProperty(..)` is accessible for all normal objects via delegation to `Object.prototype` (see Chapter 5). But it's possible to create an object that does not link to `Object.prototype` (via `Object.create(null)` -- see Chapter 5). In this case, a method call like `myObject.hasOwnProperty(..)` would fail.

In that scenario, a more robust way of performing such a check is

`Object.prototype.hasOwnProperty.call(myObject, "a")`, which borrows the base `hasOwnProperty(..)` method and uses *explicit this binding* (see Chapter 2) to apply it against our `myObject`.

Note: The `in` operator has the appearance that it will check for the existence of a *value* inside a container, but it actually checks for the existence of a property name. This difference is important to note with respect to arrays, as the temptation to try a check like `4 in [2, 4, 6]` is strong, but this will not behave as expected.

Enumeration

Previously, we explained briefly the idea of "enumerability" when we looked at the `enumerable` property descriptor characteristic. Let's revisit that and examine it in more close detail.

```
var myObject = { };

Object.defineProperty(
  myObject,
  "a",
  // make `a` enumerable, as normal
  { enumerable: true, value: 2 }
);

Object.defineProperty(
  myObject,
  "b",
  // make `b` NON-enumerable
  { enumerable: false, value: 3 }
);

myObject.b; // 3
("b" in myObject); // true
myObject.hasOwnProperty( "b" ); // true

// .....

for (var k in myObject) {
  console.log( k, myObject[k] );
}
// "a" 2
```

You'll notice that `myObject.b` in fact **exists** and has an accessible value, but it doesn't show up in a `for..in` loop (though, surprisingly, it **is** revealed by the `in` operator existence check). That's because "enumerable" basically means "will be included if the object's properties are iterated through".

Note: `for..in` loops applied to arrays can give somewhat unexpected results, in that the enumeration of an array will include not only all the numeric indices, but also any enumerable properties. It's a good idea to use `for..in` loops *only* on objects, and traditional `for` loops with numeric index iteration for the values stored in arrays.

Another way that enumerable and non-enumerable properties can be distinguished:

```

var myObject = { };

Object.defineProperty(
  myObject,
  "a",
  // make `a` enumerable, as normal
  { enumerable: true, value: 2 }
);

Object.defineProperty(
  myObject,
  "b",
  // make `b` non-enumerable
  { enumerable: false, value: 3 }
);

myObject.propertyIsEnumerable( "a" ); // true
myObject.propertyIsEnumerable( "b" ); // false

Object.keys( myObject ); // ["a"]
Object.getOwnPropertyNames( myObject ); // ["a", "b"]

```

`propertyIsEnumerable(..)` tests whether the given property name exists *directly* on the object and is also `enumerable:true`.

`Object.keys(..)` returns an array of all enumerable properties, whereas `Object.getOwnPropertyNames(..)` returns an array of *all* properties, enumerable or not.

Whereas `in` vs. `hasOwnProperty(..)` differ in whether they consult the `[[Prototype]]` chain or not, `Object.keys(..)` and `Object.getOwnPropertyNames(..)` both inspect *only* the direct object specified.

There's (currently) no built-in way to get a list of **all properties** which is equivalent to what the `in` operator test would consult (traversing all properties on the entire `[[Prototype]]` chain, as explained in Chapter 5). You could approximate such a utility by recursively traversing the `[[Prototype]]` chain of an object, and for each level, capturing the list from `Object.keys(..)` -- only enumerable properties.

Iteration

The `for..in` loop iterates over the list of enumerable properties on an object (including its `[[Prototype]]` chain). But what if you instead want to iterate over the values?

With numerically-indexed arrays, iterating over the values is typically done with a standard `for` loop, like:

```
var myArray = [1, 2, 3];

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

// 1 2 3
```

This isn't iterating over the values, though, but iterating over the indices, where you then use the index to reference the value, as `myArray[i]`.

ES5 also added several iteration helpers for arrays, including `forEach(..)`, `every(..)`, and `some(..)`. Each of these helpers accepts a function callback to apply to each element in the array, differing only in how they respectively respond to a return value from the callback.

`forEach(..)` will iterate over all values in the array, and ignores any callback return values. `every(..)` keeps going until the end or the callback returns a `false` (or "falsy") value, whereas `some(..)` keeps going until the end or the callback returns a `true` (or "truthy") value.

These special return values inside `every(..)` and `some(..)` act somewhat like a `break` statement inside a normal `for` loop, in that they stop the iteration early before it reaches the end.

If you iterate on an object with a `for..in` loop, you're also only getting at the values indirectly, because it's actually iterating only over the enumerable properties of the object, leaving you to access the properties manually to get the values.

Note: As contrasted with iterating over an array's indices in a numerically ordered way (`for` loop or other iterators), the order of iteration over an object's properties is **not guaranteed** and may vary between different JS engines. **Do not rely** on any observed ordering for anything that requires consistency among environments, as any observed agreement is unreliable.

But what if you want to iterate over the values directly instead of the array indices (or object properties)? Helpfully, ES6 adds a `for..of` loop syntax for iterating over arrays (and objects, if the object defines its own custom iterator):

```
var myArray = [ 1, 2, 3 ];

for (var v of myArray) {
    console.log( v );
}

// 1
// 2
// 3
```

The `for..of` loop asks for an iterator object (from a default internal function known as `@@iterator` in spec-speak) of the *thing* to be iterated, and the loop then iterates over the successive return values from calling that iterator object's `next()` method, once for each loop iteration.

Arrays have a built-in `@@iterator`, so `for..of` works easily on them, as shown. But let's manually iterate the array, using the built-in `@@iterator`, to see how it works:

```
var myArray = [ 1, 2, 3 ];
var it = myArray[Symbol.iterator]();

it.next(); // { value:1, done:false }
it.next(); // { value:2, done:false }
it.next(); // { value:3, done:false }
it.next(); // { done:true }
```

Note: We get at the `@@iterator` *internal property* of an object using an ES6 `Symbol`: `Symbol.iterator`. We briefly mentioned `Symbol` semantics earlier in the chapter (see "Computed Property Names"), so the same reasoning applies here. You'll always want to reference such special properties by `Symbol` name reference instead of by the special value it may hold. Also, despite the name's implications, `@@iterator` is **not the iterator object** itself, but a **function that returns** the iterator object -- a subtle but important detail!

As the above snippet reveals, the return value from an iterator's `next()` call is an object of the form `{ value: ... , done: ... }`, where `value` is the current iteration value, and `done` is a `boolean` that indicates if there's more to iterate.

Notice the value `3` was returned with a `done:false`, which seems strange at first glance. You have to call the `next()` a fourth time (which the `for..of` loop in the previous snippet automatically does) to get `done:true` and know you're truly done iterating. The reason for this quirk is beyond the scope of what we'll discuss here, but it comes from the semantics of ES6 generator functions.

While arrays do automatically iterate in `for..of` loops, regular objects **do not have a built-in `@@iterator`**. The reasons for this intentional omission are more complex than we will examine here, but in general it was better to not include some implementation that could prove troublesome for future types of objects.

It *is* possible to define your own default `@@iterator` for any object that you care to iterate over. For example:

```

var myObject = {
  a: 2,
  b: 3
};

Object.defineProperty( myObject, Symbol.iterator, {
  enumerable: false,
  writable: false,
  configurable: true,
  value: function() {
    var o = this;
    var idx = 0;
    var ks = Object.keys( o );
    return {
      next: function() {
        return {
          value: o[ks[idx++]],
          done: (idx > ks.length)
        };
      }
    };
  }
} );

// iterate `myObject` manually
var it = myObject[Symbol.iterator]();
it.next(); // { value:2, done:false }
it.next(); // { value:3, done:false }
it.next(); // { value:undefined, done:true }

// iterate `myObject` with `for..of`
for (var v of myObject) {
  console.log( v );
}
// 2
// 3

```

Note: We used `Object.defineProperty(...)` to define our custom `@@iterator` (mostly so we could make it non-enumerable), but using the `Symbol` as a *computed property name* (covered earlier in this chapter), we could have declared it directly, like `var myObject = { a:2, b:3, [Symbol.iterator]: function(){ /* .. */ } } .`

Each time the `for..of` loop calls `next()` on `myObject`'s iterator object, the internal pointer will advance and return back the next value from the object's properties list (see a previous note about iteration ordering on object properties/values).

The iteration we just demonstrated is a simple value-by-value iteration, but you can of course define arbitrarily complex iterations for your custom data structures, as you see fit. Custom iterators combined with ES6's `for..of` loop are a powerful new syntactic tool for

manipulating user-defined objects.

For example, a list of `Pixel` objects (with `x` and `y` coordinate values) could decide to order its iteration based on the linear distance from the `(0,0)` origin, or filter out points that are "too far away", etc. As long as your iterator returns the expected `{ value: ... }` return values from `next()` calls, and a `{ done: true }` after the iteration is complete, ES6's `for..of` can iterate over it.

In fact, you can even generate "infinite" iterators which never "finish" and always return a new value (such as a random number, an incremented value, a unique identifier, etc), though you probably will not use such iterators with an unbounded `for..of` loop, as it would never end and would hang your program.

```
var randoms = {
  [Symbol.iterator]: function() {
    return {
      next: function() {
        return { value: Math.random() };
      }
    };
  }
};

var randoms_pool = [];
for (var n of randoms) {
  randoms_pool.push( n );

  // don't proceed unbounded!
  if (randoms_pool.length === 100) break;
}
```

This iterator will generate random numbers "forever", so we're careful to only pull out 100 values so our program doesn't hang.

Review (TL;DR)

Objects in JS have both a literal form (such as `var a = { .. }`) and a constructed form (such as `var a = new Array(..)`). The literal form is almost always preferred, but the constructed form offers, in some cases, more creation options.

Many people mistakenly claim "everything in JavaScript is an object", but this is incorrect. Objects are one of the 6 (or 7, depending on your perspective) primitive types. Objects have sub-types, including `function`, and also can be behavior-specialized, like `[object Array]` as the internal label representing the array object sub-type.

Objects are collections of key/value pairs. The values can be accessed as properties, via `.propName` or `["propName"]` syntax. Whenever a property is accessed, the engine actually invokes the internal default `[[Get]]` operation (and `[[Put]]` for setting values), which not only looks for the property directly on the object, but which will traverse the `[[Prototype]]` chain (see Chapter 5) if not found.

Properties have certain characteristics that can be controlled through property descriptors, such as `writable` and `configurable`. In addition, objects can have their mutability (and that of their properties) controlled to various levels of immutability using

```
Object.preventExtensions(..) , Object.seal(..) , and Object.freeze(..) .
```

Properties don't have to contain values -- they can be "accessor properties" as well, with getters/setters. They can also be either `enumerable` or not, which controls if they show up in `for..in` loop iterations, for instance.

You can also iterate over **the values** in data structures (arrays, objects, etc) using the ES6 `for..of` syntax, which looks for either a built-in or custom `@@iterator` object consisting of a `next()` method to advance through the data values one at a time.

Chapter 4: Mixing (Up) "Class" Objects

Following our exploration of objects from the previous chapter, it's natural that we now turn our attention to "object oriented (OO) programming", with "classes". We'll first look at "class orientation" as a design pattern, before examining the mechanics of "classes": "instantiation", "inheritance" and "(relative) polymorphism".

We'll see that these concepts don't really map very naturally to the object mechanism in JS, and the lengths (mixins, etc.) many JavaScript developers go to overcome such challenges.

Note: This chapter spends quite a bit of time (the first half!) on heavy "objected oriented programming" theory. We eventually relate these ideas to real concrete JavaScript code in the second half, when we talk about "Mixins". But there's a lot of concept and pseudo-code to wade through first, so don't get lost -- just stick with it!

Class Theory

"Class/Inheritance" describes a certain form of code organization and architecture -- a way of modeling real world problem domains in our software.

OO or class oriented programming stresses that data intrinsically has associated behavior (of course, different depending on the type and nature of the data!) that operates on it, so proper design is to package up (aka, encapsulate) the data and the behavior together. This is sometimes called "data structures" in formal computer science.

For example, a series of characters that represents a word or phrase is usually called a "string". The characters are the data. But you almost never just care about the data, you usually want to *do things* with the data, so the behaviors that can apply *to* that data (calculating its length, appending data, searching, etc.) are all designed as methods of a `String` class.

Any given string is just an instance of this class, which means that it's a neatly collected packaging of both the character data and the functionality we can perform on it.

Classes also imply a way of *classifying* a certain data structure. The way we do this is to think about any given structure as a specific variation of a more general base definition.

Let's explore this classification process by looking at a commonly cited example. A *car* can be described as a specific implementation of a more general "class" of thing, called a *vehicle*.

We model this relationship in software with classes by defining a `Vehicle` class and a `Car` class.

The definition of `Vehicle` might include things like propulsion (engines, etc.), the ability to carry people, etc., which would all be the behaviors. What we define in `Vehicle` is all the stuff that is common to all (or most of) the different types of vehicles (the "planes, trains, and automobiles").

It might not make sense in our software to re-define the basic essence of "ability to carry people" over and over again for each different type of vehicle. Instead, we define that capability once in `Vehicle`, and then when we define `Car`, we simply indicate that it "inherits" (or "extends") the base definition from `Vehicle`. The definition of `Car` is said to specialize the general `Vehicle` definition.

While `Vehicle` and `Car` collectively define the behavior by way of methods, the data in an instance would be things like the unique VIN of a specific car, etc.

And thus, classes, inheritance, and instantiation emerge.

Another key concept with classes is "polymorphism", which describes the idea that a general behavior from a parent class can be overridden in a child class to give it more specifics. In fact, relative polymorphism lets us reference the base behavior from the overridden behavior.

Class theory strongly suggests that a parent class and a child class share the same method name for a certain behavior, so that the child overrides the parent (differentially). As we'll see later, doing so in your JavaScript code is opting into frustration and code brittleness.

"Class" Design Pattern

You may never have thought about classes as a "design pattern", since it's most common to see discussion of popular "OO Design Patterns", like "Iterator", "Observer", "Factory", "Singleton", etc. As presented this way, it's almost an assumption that OO classes are the lower-level mechanics by which we implement all (higher level) design patterns, as if OO is a given foundation for *all* (proper) code.

Depending on your level of formal education in programming, you may have heard of "procedural programming" as a way of describing code which only consists of procedures (aka, functions) calling other functions, without any higher abstractions. You may have been taught that classes were the *proper* way to transform procedural-style "spaghetti code" into well-formed, well-organized code.

Of course, if you have experience with "functional programming" (Monads, etc.), you know very well that classes are just one of several common design patterns. But for others, this may be the first time you've asked yourself if classes really are a fundamental foundation for code, or if they are an optional abstraction on top of code.

Some languages (like Java) don't give you the choice, so it's not very *optional* at all -- everything's a class. Other languages like C/C++ or PHP give you both procedural and class-oriented syntaxes, and it's left more to the developer's choice which style or mixture of styles is appropriate.

JavaScript "Classes"

Where does JavaScript fall in this regard? JS has had *some* class-like syntactic elements (like `new` and `instanceof`) for quite awhile, and more recently in ES6, some additions, like the `class` keyword (see Appendix A).

But does that mean JavaScript actually *has* classes? Plain and simple: **No.**

Since classes are a design pattern, you *can*, with quite a bit of effort (as we'll see throughout the rest of this chapter), implement approximations for much of classical class functionality. JS tries to satisfy the extremely pervasive *desire* to design with classes by providing seemingly class-like syntax.

While we may have a syntax that looks like classes, it's as if JavaScript mechanics are fighting against you using the *class design pattern*, because behind the curtain, the mechanisms that you build on are operating quite differently. Syntactic sugar and (extremely widely used) JS "Class" libraries go a long way toward hiding this reality from you, but sooner or later you will face the fact that the *classes* you have in other languages are not like the "classes" you're faking in JS.

What this boils down to is that classes are an optional pattern in software design, and you have the choice to use them in JavaScript or not. Since many developers have a strong affinity to class oriented software design, we'll spend the rest of this chapter exploring what it takes to maintain the illusion of classes with what JS provides, and the pain points we experience.

Class Mechanics

In many class-oriented languages, the "standard library" provides a "stack" data structure (`push`, `pop`, etc.) as a `Stack` class. This class would have an internal set of variables that stores the data, and it would have a set of publicly accessible behaviors ("methods")

provided by the class, which gives your code the ability to interact with the (hidden) data (adding & removing data, etc.).

But in such languages, you don't really operate directly on `stack` (unless making a **Static** class member reference, which is outside the scope of our discussion). The `stack` class is merely an abstract explanation of what *any* "stack" should do, but it's not itself a "stack". You must **instantiate** the `Stack` class before you have a concrete data structure *thing* to operate against.

Building

The traditional metaphor for "class" and "instance" based thinking comes from a building construction.

An architect plans out all the characteristics of a building: how wide, how tall, how many windows and in what locations, even what type of material to use for the walls and roof. She doesn't necessarily care, at this point, *where* the building will be built, nor does she care *how many* copies of that building will be built.

She also doesn't care very much about the contents of the building -- the furniture, wall paper, ceiling fans, etc. -- only what type of structure they will be contained by.

The architectural blue-prints she produces are only *plans* for a building. They don't actually constitute a building we can walk into and sit down. We need a builder for that task. A builder will take those plans and follow them, exactly, as he *builds* the building. In a very real sense, he is *copying* the intended characteristics from the plans to the physical building.

Once complete, the building is a physical instantiation of the blue-print plans, hopefully an essentially perfect *copy*. And then the builder can move to the open lot next door and do it all over again, creating yet another *copy*.

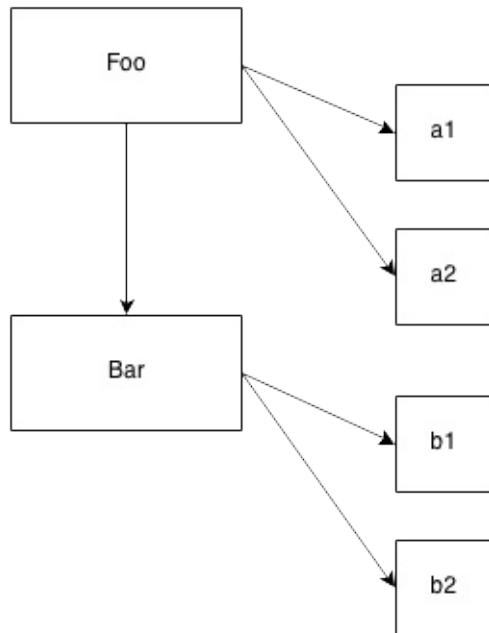
The relationship between building and blue-print is indirect. You can examine a blue-print to understand how the building was structured, for any parts where direct inspection of the building itself was insufficient. But if you want to open a door, you have to go to the building itself -- the blue-print merely has lines drawn on a page that *represent* where the door should be.

A class is a blue-print. To actually *get* an object we can interact with, we must build (aka, "instantiate") something from the class. The end result of such "construction" is an object, typically called an "instance", which we can directly call methods on and access any public data properties from, as necessary.

This object is a *copy* of all the characteristics described by the class.

You likely wouldn't expect to walk into a building and find, framed and hanging on the wall, a copy of the blue-prints used to plan the building, though the blue-prints are probably on file with a public records office. Similarly, you don't generally use an object instance to directly access and manipulate its class, but it is usually possible to at least determine *which class* an object instance comes from.

It's more useful to consider the direct relationship of a class to an object instance, rather than any indirect relationship between an object instance and the class it came from. **A class is instantiated into object form by a copy operation.**



As you can see, the arrows move from left to right, and from top to bottom, which indicates the copy operations that occur, both conceptually and physically.

Constructor

Instances of classes are constructed by a special method of the class, usually of the same name as the class, called a *constructor*. This method's explicit job is to initialize any information (state) the instance will need.

For example, consider this loose pseudo-code (invented syntax) for classes:

```

class CoolGuy {
    specialTrick = nothing

    CoolGuy( trick ) {
        specialTrick = trick
    }

    showOff() {
        output( "Here's my trick: ", specialTrick )
    }
}

```

To *make* a `CoolGuy` instance, we would call the class constructor:

```

Joe = new CoolGuy( "jumping rope" )

Joe.showOff() // Here's my trick: jumping rope

```

Notice that the `CoolGuy` class has a constructor `CoolGuy()`, which is actually what we call when we say `new CoolGuy(...)`. We get an object back (an instance of our class) from the constructor, and we can call the method `showoff()`, which prints out that particular `CoolGuy`'s special trick.

Obviously, jumping rope makes Joe a pretty cool guy.

The constructor of a class *belongs* to the class, almost universally with the same name as the class. Also, constructors pretty much always need to be called with `new` to let the language engine know you want to construct a *new* class instance.

Class Inheritance

In class-oriented languages, not only can you define a class which can be instantiated itself, but you can define another class that **inherits** from the first class.

The second class is often said to be a "child class" whereas the first is the "parent class". These terms obviously come from the metaphor of parents and children, though the metaphors here are a bit stretched, as you'll see shortly.

When a parent has a biological child, the genetic characteristics of the parent are copied into the child. Obviously, in most biological reproduction systems, there are two parents who co-equally contribute genes to the mix. But for the purposes of the metaphor, we'll assume just one parent.

Once the child exists, he or she is separate from the parent. The child was heavily influenced by the inheritance from his or her parent, but is unique and distinct. If a child ends up with red hair, that doesn't mean the parent's hair *was* or automatically *becomes* red.

In a similar way, once a child class is defined, it's separate and distinct from the parent class. The child class contains an initial copy of the behavior from the parent, but can then override any inherited behavior and even define new behavior.

It's important to remember that we're talking about parent and child **classes**, which aren't physical things. This is where the metaphor of parent and child gets a little confusing, because we actually should say that a parent class is like a parent's DNA and a child class is like a child's DNA. We have to make (aka "instantiate") a person out of each set of DNA to actually have a physical person to have a conversation with.

Let's set aside biological parents and children, and look at inheritance through a slightly different lens: different types of vehicles. That's one of the most canonical (and often groan-worthy) metaphors to understand inheritance.

Let's revisit the `Vehicle` and `car` discussion from earlier in this chapter. Consider this loose pseudo-code (invented syntax) for inherited classes:

```

class Vehicle {
    engines = 1

    ignition() {
        output( "Turning on my engine." )
    }

    drive() {
        ignition()
        output( "Steering and moving forward!" )
    }
}

class Car inherits Vehicle {
    wheels = 4

    drive() {
        inherited:drive()
        output( "Rolling on all ", wheels, " wheels!" )
    }
}

class SpeedBoat inherits Vehicle {
    engines = 2

    ignition() {
        output( "Turning on my ", engines, " engines." )
    }

    pilot() {
        inherited:drive()
        output( "Speeding through the water with ease!" )
    }
}

```

Note: For clarity and brevity, constructors for these classes have been omitted.

We define the `Vehicle` class to assume an engine, a way to turn on the ignition, and a way to drive around. But you wouldn't ever manufacture just a generic "vehicle", so it's really just an abstract concept at this point.

So then we define two specific kinds of vehicle: `Car` and `SpeedBoat`. They each inherit the general characteristics of `Vehicle`, but then they specialize the characteristics appropriately for each kind. A car needs 4 wheels, and a speed boat needs 2 engines, which means it needs extra attention to turn on the ignition of both engines.

Polymorphism

`car` defines its own `drive()` method, which overrides the method of the same name it inherited from `Vehicle`. But then, `car`'s `drive()` method calls `inherited:drive()`, which indicates that `car` can reference the original pre-overridden `drive()` it inherited.

`SpeedBoat`'s `pilot()` method also makes a reference to its inherited copy of `drive()`.

This technique is called "polymorphism", or "virtual polymorphism". More specifically to our current point, we'll call it "relative polymorphism".

Polymorphism is a much broader topic than we will exhaust here, but our current "relative" semantics refers to one particular aspect: the idea that any method can reference another method (of the same or different name) at a higher level of the inheritance hierarchy. We say "relative" because we don't absolutely define which inheritance level (aka, class) we want to access, but rather relatively reference it by essentially saying "look one level up".

In many languages, the keyword `super` is used, in place of this example's `inherited:`, which leans on the idea that a "super class" is the parent/ancestor of the current class.

Another aspect of polymorphism is that a method name can have multiple definitions at different levels of the inheritance chain, and these definitions are automatically selected as appropriate when resolving which methods are being called.

We see two occurrences of that behavior in our example above: `drive()` is defined in both `Vehicle` and `Car`, and `ignition()` is defined in both `Vehicle` and `SpeedBoat`.

Note: Another thing that traditional class-oriented languages give you via `super` is a direct way for the constructor of a child class to reference the constructor of its parent class. This is largely true because with real classes, the constructor belongs to the class. However, in JS, it's the reverse -- it's actually more appropriate to think of the "class" belonging to the constructor (the `Foo.prototype...` type references). Since in JS the relationship between child and parent exists only between the two `.prototype` objects of the respective constructors, the constructors themselves are not directly related, and thus there's no simple way to relatively reference one from the other (see Appendix A for ES6 `class` which "solves" this with `super`).

An interesting implication of polymorphism can be seen specifically with `ignition()`. Inside `pilot()`, a relative-polymorphic reference is made to (the inherited) `Vehicle`'s version of `drive()`. But that `drive()` references an `ignition()` method just by name (no relative reference).

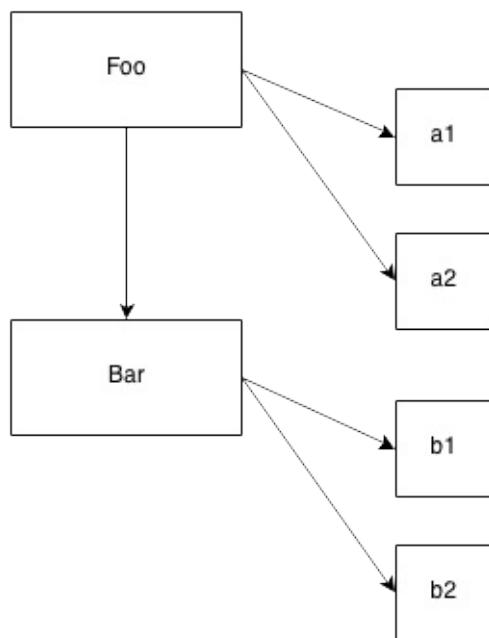
Which version of `ignition()` will the language engine use, the one from `Vehicle` or the one from `SpeedBoat`? **It uses the `SpeedBoat` version of `ignition()`.** If you were to instantiate `Vehicle` class itself, and then call its `drive()`, the language engine would instead just use `Vehicle`'s `ignition()` method definition.

Put another way, the definition for the method `ignition()` *polymorphs* (changes) depending on which class (level of inheritance) you are referencing an instance of.

This may seem like overly deep academic detail. But understanding these details is necessary to properly contrast similar (but distinct) behaviors in JavaScript's `[[Prototype]]` mechanism.

When classes are inherited, there is a way **for the classes themselves** (not the object instances created from them!) to *relatively* reference the class inherited from, and this relative reference is usually called `super`.

Remember this figure from earlier:



Notice how for both instantiation (`a1`, `a2`, `b1`, and `b2`) *and* inheritance (`Bar`), the arrows indicate a copy operation.

Conceptually, it would seem a child class `Bar` can access behavior in its parent class `Foo` using a relative polymorphic reference (aka, `super`). However, in reality, the child class is merely given a copy of the inherited behavior from its parent class. If the child "overrides" a method it inherits, both the original and overridden versions of the method are actually maintained, so that they are both accessible.

Don't let polymorphism confuse you into thinking a child class is linked to its parent class. A child class instead gets a copy of what it needs from the parent class. **Class inheritance implies copies.**

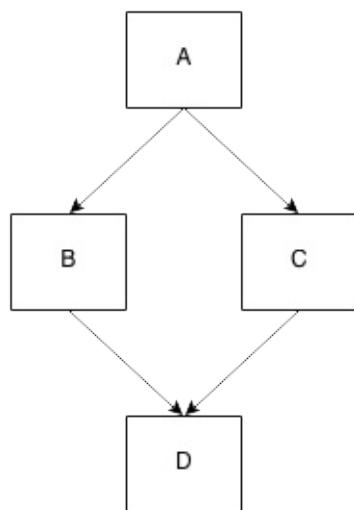
Multiple Inheritance

Recall our earlier discussion of parent(s) and children and DNA? We said that the metaphor was a bit weird because biologically most offspring come from two parents. If a class could inherit from two other classes, it would more closely fit the parent/child metaphor.

Some class-oriented languages allow you to specify more than one "parent" class to "inherit" from. Multiple-inheritance means that each parent class definition is copied into the child class.

On the surface, this seems like a powerful addition to class-orientation, giving us the ability to compose more functionality together. However, there are certainly some complicating questions that arise. If both parent classes provide a method called `drive()`, which version would a `drive()` reference in the child resolve to? Would you always have to manually specify which parent's `drive()` you meant, thus losing some of the gracefulness of polymorphic inheritance?

There's another variation, the so called "Diamond Problem", which refers to the scenario where a child class "D" inherits from two parent classes ("B" and "C"), and each of those in turn inherits from a common "A" parent. If "A" provides a method `drive()`, and both "B" and "C" override (polymorph) that method, when `D` references `drive()`, which version should it use (`B:drive()` or `C:drive()`)?



These complications go even much deeper than this quick glance. We address them here only so we can contrast to how JavaScript's mechanisms work.

JavaScript is simpler: it does not provide a native mechanism for "multiple inheritance". Many see this as a good thing, because the complexity savings more than make up for the "reduced" functionality. But this doesn't stop developers from trying to fake it in various ways, as we'll see next.

Mixins

JavaScript's object mechanism does not *automatically* perform copy behavior when you "inherit" or "instantiate". Plainly, there are no "classes" in JavaScript to instantiate, only objects. And objects don't get copied to other objects, they get *linked together* (more on that in Chapter 5).

Since observed class behaviors in other languages imply copies, let's examine how JS developers **fake** the *missing* copy behavior of classes in JavaScript: mixins. We'll look at two types of "mixin": **explicit** and **implicit**.

Explicit Mixins

Let's again revisit our `Vehicle` and `Car` example from before. Since JavaScript will not automatically copy behavior from `Vehicle` to `Car`, we can instead create a utility that manually copies. Such a utility is often called `extend(...)` by many libraries/frameworks, but we will call it `.mixin(..)` here for illustrative purposes.

```
// vastly simplified `mixin(..)` example:
function mixin( sourceObj, targetObj ) {
    for (var key in sourceObj) {
        // only copy if not already present
        if (!(key in targetObj)) {
            targetObj[key] = sourceObj[key];
        }
    }

    return targetObj;
}

var Vehicle = {
    engines: 1,

    ignition: function() {
        console.log( "Turning on my engine." );
    },

    drive: function() {
        this.ignition();
        console.log( "Steering and moving forward!" );
    }
};

var Car = mixin( Vehicle, {
    wheels: 4,

    drive: function() {
        Vehicle.drive.call( this );
        console.log( "Rolling on all " + this.wheels + " wheels!" );
    }
} );
```

Note: Subtly but importantly, we're not dealing with classes anymore, because there are no classes in JavaScript. `Vehicle` and `Car` are just objects that we make copies from and to, respectively.

`car` now has a copy of the properties and functions from `Vehicle`. Technically, functions are not actually duplicated, but rather *references* to the functions are copied. So, `car` now has a property called `ignition`, which is a copied reference to the `ignition()` function, as well as a property called `wheels` with the copied value of `1` from `Vehicle`.

`car` *already* had a `drive` property (function), so that property reference was not overridden (see the `if` statement in `mixin(..)` above).

"Polymorphism" Revisited

Let's examine this statement: `Vehicle.drive.call(this)`. This is what I call "explicit pseudo-polymorphism". Recall in our previous pseudo-code this line was `inherited:drive()`, which we called "relative polymorphism".

JavaScript does not have (prior to ES6; see Appendix A) a facility for relative polymorphism. So, **because both `car` and `Vehicle` had a function of the same name: `drive()`**, to distinguish a call to one or the other, we must make an absolute (not relative) reference. We explicitly specify the `Vehicle` object by name, and call the `drive()` function on it.

But if we said `Vehicle.drive()`, the `this` binding for that function call would be the `Vehicle` object instead of the `car` object (see Chapter 2), which is not what we want. So, instead we use `.call(this)` (Chapter 2) to ensure that `drive()` is executed in the context of the `car` object.

Note: If the function name identifier for `car.drive()` hadn't overlapped with (aka, "shadowed"; see Chapter 5) `Vehicle.drive()`, we wouldn't have been exercising "method polymorphism". So, a reference to `vehicle.drive()` would have been copied over by the `mixin(..)` call, and we could have accessed directly with `this.drive()`. The chosen identifier overlap **shadowing** is *why* we have to use the more complex *explicit pseudo-polymorphism* approach.

In class-oriented languages, which have relative polymorphism, the linkage between `car` and `Vehicle` is established once, at the top of the class definition, which makes for only one place to maintain such relationships.

But because of JavaScript's peculiarities, explicit pseudo-polymorphism (because of shadowing!) creates brittle manual/explicit linkage **in every single function where you need such a (pseudo-)polymorphic reference**. This can significantly increase the maintenance cost. Moreover, while explicit pseudo-polymorphism can emulate the behavior of "multiple inheritance", it only increases the complexity and brittleness.

The result of such approaches is usually more complex, harder-to-read, *and* harder-to-maintain code. **Explicit pseudo-polymorphism should be avoided wherever possible**, because the cost outweighs the benefit in most respects.

Mixing Copies

Recall the `mixin(..)` utility from above:

```
// vastly simplified `mixin()` example:  
function mixin( sourceObj, targetObj ) {  
    for (var key in sourceObj) {  
        // only copy if not already present  
        if (!(key in targetObj)) {  
            targetObj[key] = sourceObj[key];  
        }  
    }  
  
    return targetObj;  
}
```

Now, let's examine how `mixin(..)` works. It iterates over the properties of `sourceObj` (`Vehicle` in our example) and if there's no matching property of that name in `targetObj` (`Car` in our example), it makes a copy. Since we're making the copy after the initial object exists, we are careful to not copy over a target property.

If we made the copies first, before specifying the `Car` specific contents, we could omit this check against `targetObj`, but that's a little more clunky and less efficient, so it's generally less preferred:

```
// alternate mixin, less "safe" to overwrites  
function mixin( sourceObj, targetObj ) {  
    for (var key in sourceObj) {  
        targetObj[key] = sourceObj[key];  
    }  
  
    return targetObj;  
}  
  
var Vehicle = {  
    // ...  
};  
  
// first, create an empty object with  
// Vehicle's stuff copied in  
var Car = mixin( Vehicle, { } );  
  
// now copy the intended contents into Car  
mixin( {  
    wheels: 4,  
  
    drive: function() {  
        // ...  
    }  
}, Car );
```

Either approach, we have explicitly copied the non-overlapping contents of `Vehicle` into `car`. The name "mixin" comes from an alternate way of explaining the task: `car` has `Vehicle`'s contents **mixed-in**, just like you mix in chocolate chips into your favorite cookie dough.

As a result of the copy operation, `car` will operate somewhat separately from `Vehicle`. If you add a property onto `car`, it will not affect `Vehicle`, and vice versa.

Note: A few minor details have been skimmed over here. There are still some subtle ways the two objects can "affect" each other even after copying, such as if they both share a reference to a common object (such as an array).

Since the two objects also share references to their common functions, that means that **even manual copying of functions (aka, mixins) from one object to another doesn't actually emulate the real duplication from class to instance that occurs in class-oriented languages.**

JavaScript functions can't really be duplicated (in a standard, reliable way), so what you end up with instead is a **duplicated reference** to the same shared function object (functions are objects; see Chapter 3). If you modified one of the shared **function objects** (like `ignition()`) by adding properties on top of it, for instance, both `Vehicle` and `car` would be "affected" via the shared reference.

Explicit mixins are a fine mechanism in JavaScript. But they appear more powerful than they really are. Not much benefit is *actually* derived from copying a property from one object to another, **as opposed to just defining the properties twice**, once on each object. And that's especially true given the function-object reference nuance we just mentioned.

If you explicitly mix-in two or more objects into your target object, you can **partially emulate** the behavior of "multiple inheritance", but there's no direct way to handle collisions if the same method or property is being copied from more than one source. Some developers/libraries have come up with "late binding" techniques and other exotic work-arounds, but fundamentally these "tricks" are *usually* more effort (and lesser performance!) than the pay-off.

Take care only to use explicit mixins where it actually helps make more readable code, and avoid the pattern if you find it making code that's harder to trace, or if you find it creates unnecessary or unwieldy dependencies between objects.

If it starts to get harder to properly use mixins than before you used them, you should probably stop using mixins. In fact, if you have to use a complex library/utility to work out all these details, it might be a sign that you're going about it the harder way, perhaps unnecessarily. In Chapter 6, we'll try to distill a simpler way that accomplishes the desired outcomes without all the fuss.

Parasitic Inheritance

A variation on this explicit mixin pattern, which is both in some ways explicit and in other ways implicit, is called "parasitic inheritance", popularized mainly by Douglas Crockford.

Here's how it can work:

```
// "Traditional JS Class" `Vehicle`
function Vehicle() {
    this.engines = 1;
}
Vehicle.prototype.ignition = function() {
    console.log( "Turning on my engine." );
};
Vehicle.prototype.drive = function() {
    this.ignition();
    console.log( "Steering and moving forward!" );
};

// "Parasitic Class" `Car`
function Car() {
    // first, `car` is a `Vehicle`
    var car = new Vehicle();

    // now, let's modify our `car` to specialize it
    car.wheels = 4;

    // save a privileged reference to `Vehicle::drive()`
    var vehDrive = car.drive;

    // override `Vehicle::drive()`
    car.drive = function() {
        vehDrive.call( this );
        console.log( "Rolling on all " + this.wheels + " wheels!" );
    };
}

return car;
}

var myCar = new Car();

myCar.drive();
// Turning on my engine.
// Steering and moving forward!
// Rolling on all 4 wheels!
```

As you can see, we initially make a copy of the definition from the `Vehicle` "parent class" (object), then mixin our "child class" (object) definition (preserving privileged parent-class references as needed), and pass off this composed object `car` as our child instance.

Note: when we call `new car()`, a new object is created and referenced by `car`'s `this` reference (see Chapter 2). But since we don't use that object, and instead return our own `car` object, the initially created object is just discarded. So, `car()` could be called without the `new` keyword, and the functionality above would be identical, but without the wasted object creation/garbage-collection.

Implicit Mixins

Implicit mixins are closely related to *explicit pseudo-polymorphism* as explained previously. As such, they come with the same caveats and warnings.

Consider this code:

```
var Something = {
  cool: function() {
    this.greeting = "Hello World";
    this.count = this.count ? this.count + 1 : 1;
  }
};

Something.cool();
Something.greeting; // "Hello World"
Something.count; // 1

var Another = {
  cool: function() {
    // implicit mixin of `Something` to `Another`
    Something.cool.call( this );
  }
};

Another.cool();
Another.greeting; // "Hello World"
Another.count; // 1 (not shared state with `Something`)
```

With `Something.cool.call(this)`, which can happen either in a "constructor" call (most common) or in a method call (shown here), we essentially "borrow" the function `Something.cool()` and call it in the context of `Another` (via its `this` binding; see Chapter 2) instead of `Something`. The end result is that the assignments that `Something.cool()` makes are applied against the `Another` object rather than the `Something` object.

So, it is said that we "mixed in" `Something`'s behavior with (or into) `Another`.

While this sort of technique seems to take useful advantage of `this` rebinding functionality, it is the brittle `Something.cool.call(this)` call, which cannot be made into a relative (and thus more flexible) reference, that you should **heed with caution**. Generally, **avoid such**

constructs where possible to keep cleaner and more maintainable code.

Review (TL;DR)

Classes are a design pattern. Many languages provide syntax which enables natural class-oriented software design. JS also has a similar syntax, but it behaves **very differently** from what you're used to with classes in those other languages.

Classes mean copies.

When traditional classes are instantiated, a copy of behavior from class to instance occurs.
When classes are inherited, a copy of behavior from parent to child also occurs.

Polymorphism (having different functions at multiple levels of an inheritance chain with the same name) may seem like it implies a referential relative link from child back to parent, but it's still just a result of copy behavior.

JavaScript **does not automatically** create copies (as classes imply) between objects.

The mixin pattern (both explicit and implicit) is often used to *sort of* emulate class copy behavior, but this usually leads to ugly and brittle syntax like explicit pseudo-polymorphism (`otherObj.methodName.call(this, ...)`), which often results in harder to understand and maintain code.

Explicit mixins are also not exactly the same as class *copy*, since objects (and functions!) only have shared references duplicated, not the objects/functions duplicated themselves.
Not paying attention to such nuance is the source of a variety of gotchas.

In general, faking classes in JS often sets more landmines for future coding than solving present *real* problems.

Chapter 5: Prototypes

In Chapters 3 and 4, we mentioned the `[[Prototype]]` chain several times, but haven't said what exactly it is. We will now examine prototypes in detail.

Note: All of the attempts to emulate class-copy behavior, as described previously in Chapter 4, labeled as variations of "mixins", completely circumvent the `[[Prototype]]` chain mechanism we examine here in this chapter.

`[[Prototype]]`

Objects in JavaScript have an internal property, denoted in the specification as `[[Prototype]]`, which is simply a reference to another object. Almost all objects are given a non-`null` value for this property, at the time of their creation.

Note: We will see shortly that it *is* possible for an object to have an empty `[[Prototype]]` linkage, though this is somewhat less common.

Consider:

```
var myObject = {  
    a: 2  
};  
  
myObject.a; // 2
```

What is the `[[Prototype]]` reference used for? In Chapter 3, we examined the `[[Get]]` operation that is invoked when you reference a property on an object, such as `myObject.a`. For that default `[[Get]]` operation, the first step is to check if the object itself has a property `a` on it, and if so, it's used.

Note: ES6 Proxies are outside of our discussion scope in this book (will be covered in a later book in the series!), but everything we discuss here about normal `[[Get]]` and `[[Put]]` behavior does not apply if a `Proxy` is involved.

But it's what happens if a **isn't** present on `myObject` that brings our attention now to the `[[Prototype]]` link of the object.

The default `[[Get]]` operation proceeds to follow the `[[Prototype]]` link of the object if it cannot find the requested property on the object directly.

```
var anotherObject = {
  a: 2
};

// create an object linked to `anotherObject`
var myObject = Object.create( anotherObject );

myObject.a; // 2
```

Note: We will explain what `Object.create(..)` does, and how it operates, shortly. For now, just assume it creates an object with the `[[Prototype]]` linkage we're examining to the object specified.

So, we have `myObject` that is now `[[Prototype]]` linked to `anotherObject`. Clearly `myObject.a` doesn't actually exist, but nevertheless, the property access succeeds (being found on `anotherObject` instead) and indeed finds the value `2`.

But, if `a` weren't found on `anotherObject` either, its `[[Prototype]]` chain, if non-empty, is again consulted and followed.

This process continues until either a matching property name is found, or the `[[Prototype]]` chain ends. If no matching property is ever found by the end of the chain, the return result from the `[[Get]]` operation is `undefined`.

Similar to this `[[Prototype]]` chain look-up process, if you use a `for..in` loop to iterate over an object, any property that can be reached via its chain (and is also `enumerable` -- see Chapter 3) will be enumerated. If you use the `in` operator to test for the existence of a property on an object, `in` will check the entire chain of the object (regardless of *enumerability*).

```
var anotherObject = {
  a: 2
};

// create an object linked to `anotherObject`
var myObject = Object.create( anotherObject );

for (var k in myObject) {
  console.log("found: " + k);
}

// found: a

("a" in myObject); // true
```

So, the `[[Prototype]]` chain is consulted, one link at a time, when you perform property look-ups in various fashions. The look-up stops once the property is found or the chain ends.

Object.prototype

But *where* exactly does the `[[Prototype]]` chain "end"?

The top-end of every *normal* `[[Prototype]]` chain is the built-in `Object.prototype`. This object includes a variety of common utilities used all over JS, because all normal (built-in, not host-specific extension) objects in JavaScript "descend from" (aka, have at the top of their `[[Prototype]]` chain) the `Object.prototype` object.

Some utilities found here you may be familiar with include `.toString()` and `.valueOf()`. In Chapter 3, we introduced another: `.hasOwnProperty(..)`. And yet another function on `Object.prototype` you may not be familiar with, but which we'll address later in this chapter, is `.isPrototypeOf(..)`.

Setting & Shadowing Properties

Back in Chapter 3, we mentioned that setting properties on an object was more nuanced than just adding a new property to the object or changing an existing property's value. We will now revisit this situation more completely.

```
myObject.foo = "bar";
```

If the `myObject` object already has a normal data accessor property called `foo` directly present on it, the assignment is as simple as changing the value of the existing property.

If `foo` is not already present directly on `myObject`, the `[[Prototype]]` chain is traversed, just like for the `[[Get]]` operation. If `foo` is not found anywhere in the chain, the property `foo` is added directly to `myObject` with the specified value, as expected.

However, if `foo` is already present somewhere higher in the chain, nuanced (and perhaps surprising) behavior can occur with the `myObject.foo = "bar"` assignment. We'll examine that more in just a moment.

If the property name `foo` ends up both on `myObject` itself and at a higher level of the `[[Prototype]]` chain that starts at `myObject`, this is called *shadowing*. The `foo` property directly on `myObject` shadows any `foo` property which appears higher in the chain, because the `myObject.foo` look-up would always find the `foo` property that's lowest in the chain.

As we just hinted, shadowing `foo` on `myObject` is not as simple as it may seem. We will now examine three scenarios for the `myObject.foo = "bar"` assignment when `foo` is **not** already on `myObject` directly, but **is** at a higher level of `myObject`'s `[[Prototype]]` chain:

1. If a normal data accessor (see Chapter 3) property named `foo` is found anywhere

- higher on the `[[Prototype]]` chain, **and it's not marked as read-only** (`writable:false`) then a new property called `foo` is added directly to `myObject`, resulting in a **shadowed property**.
2. If a `foo` is found higher on the `[[Prototype]]` chain, but it's marked as **read-only** (`writable:false`), then both the setting of that existing property as well as the creation of the shadowed property on `myObject` **are disallowed**. If the code is running in `strict mode`, an error will be thrown. Otherwise, the setting of the property value will silently be ignored. Either way, **no shadowing occurs**.
 3. If a `foo` is found higher on the `[[Prototype]]` chain and it's a setter (see Chapter 3), then the setter will always be called. No `foo` will be added to (aka, shadowed on) `myObject`, nor will the `foo` setter be redefined.

Most developers assume that assignment of a property (`[[Put]]`) will always result in shadowing if the property already exists higher on the `[[Prototype]]` chain, but as you can see, that's only true in one (#1) of the three situations just described.

If you want to shadow `foo` in cases #2 and #3, you cannot use `=` assignment, but must instead use `Object.defineProperty(...)` (see Chapter 3) to add `foo` to `myObject`.

Note: Case #2 may be the most surprising of the three. The presence of a *read-only* property prevents a property of the same name being implicitly created (shadowed) at a lower level of a `[[Prototype]]` chain. The reason for this restriction is primarily to reinforce the illusion of class-inherited properties. If you think of the `foo` at a higher level of the chain as having been inherited (copied down) to `myObject`, then it makes sense to enforce the non-writable nature of that `foo` property on `myObject`. If you however separate the illusion from the fact, and recognize that no such inheritance copying *actually* occurred (see Chapters 4 and 5), it's a little unnatural that `myObject` would be prevented from having a `foo` property just because some other object had a non-writable `foo` on it. It's even stranger that this restriction only applies to `=` assignment, but is not enforced when using `Object.defineProperty(...)`.

Shadowing with **methods** leads to ugly *explicit pseudo-polymorphism* (see Chapter 4) if you need to delegate between them. Usually, shadowing is more complicated and nuanced than it's worth, **so you should try to avoid it if possible**. See Chapter 6 for an alternative design pattern, which among other things discourages shadowing in favor of cleaner alternatives.

Shadowing can even occur implicitly in subtle ways, so care must be taken if trying to avoid it. Consider:

```

var anotherObject = {
  a: 2
};

var myObject = Object.create( anotherObject );

anotherObject.a; // 2
myObject.a; // 2

anotherObject.hasOwnProperty( "a" ); // true
myObject.hasOwnProperty( "a" ); // false

myObject.a++; // oops, implicit shadowing!

anotherObject.a; // 2
myObject.a; // 3

myObject.hasOwnProperty( "a" ); // true

```

Though it may appear that `myObject.a++` should (via delegation) look-up and just increment the `anotherObject.a` property itself *in place*, instead the `++` operation corresponds to `myObject.a = myObject.a + 1`. The result is `[[Get]]` looking up `a` property via `[[Prototype]]` to get the current value `2` from `anotherObject.a`, incrementing the value by one, then `[[Put]]` assigning the `3` value to a new shadowed property `a` on `myObject`. Oops!

Be very careful when dealing with delegated properties that you modify. If you wanted to increment `anotherObject.a`, the only proper way is `anotherObject.a++`.

"Class"

At this point, you might be wondering: "Why does one object need to link to another object?" What's the real benefit? That is a very appropriate question to ask, but we must first understand what `[[Prototype]]` is **not** before we can fully understand and appreciate what it *is* and how it's useful.

As we explained in Chapter 4, in JavaScript, there are no abstract patterns/blueprints for objects called "classes" as there are in class-oriented languages. JavaScript **just** has objects.

In fact, JavaScript is **almost unique** among languages as perhaps the only language with the right to use the label "object oriented", because it's one of a very short list of languages where an object can be created directly, without a class at all.

In JavaScript, classes can't (being that they don't exist!) describe what an object can do. The object defines its own behavior directly. **There's just the object.**

"Class" Functions

There's a peculiar kind of behavior in JavaScript that has been shamelessly abused for years to *hack* something that *looks* like "classes". We'll examine this approach in detail.

The peculiar "sort-of class" behavior hinges on a strange characteristic of functions: all functions by default get a public, non-enumerable (see Chapter 3) property on them called `prototype`, which points at an otherwise arbitrary object.

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

Foo.prototype; // { }
```

This object is often called "Foo's prototype", because we access it via an unfortunately-named `Foo.prototype` property reference. However, that terminology is hopelessly destined to lead us into confusion, as we'll see shortly. Instead, I will call it "the object formerly known as Foo's prototype". Just kidding. How about: "object arbitrarily labeled 'Foo dot prototype'"?

Whatever we call it, what exactly is this object?

The most direct way to explain it is that each object created from calling `new Foo()` (see Chapter 2) will end up (somewhat arbitrarily) `[[Prototype]]`-linked to this "Foo dot prototype" object.

Let's illustrate:

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

var a = new Foo();

Object.getPrototypeOf(a) === Foo.prototype; // true
```

When `a` is created by calling `new Foo()`, one of the things (see Chapter 2 for all *four* steps) that happens is that `a` gets an internal `[[Prototype]]` link to the object that `Foo.prototype` is pointing at.

Stop for a moment and ponder the implications of that statement.

In class-oriented languages, multiple **copies** (aka, "instances") of a class can be made, like stamping something out from a mold. As we saw in Chapter 4, this happens because the process of instantiating (or inheriting from) a class means, "copy the behavior plan from that class into a physical object", and this is done again for each new instance.

But in JavaScript, there are no such copy-actions performed. You don't create multiple instances of a class. You can create multiple objects that `[[Prototype]]` *link* to a common object. But by default, no copying occurs, and thus these objects don't end up totally separate and disconnected from each other, but rather, quite **linked**.

`new Foo()` results in a new object (we called it `a`), and **that** new object `a` is internally `[[Prototype]]` linked to the `Foo.prototype` object.

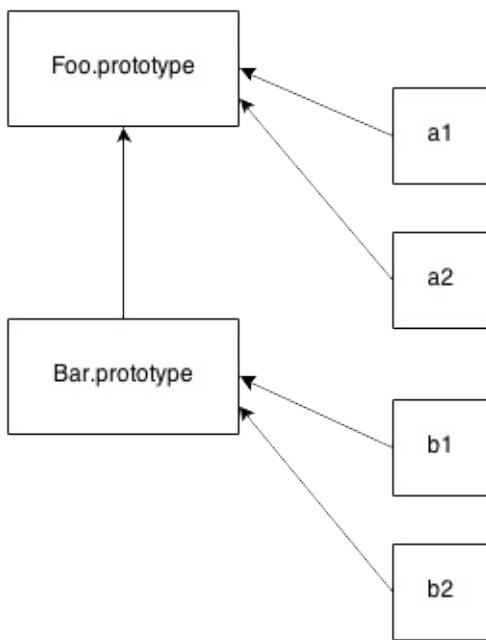
We end up with two objects, linked to each other. That's *it*. We didn't instantiate a class. We certainly didn't do any copying of behavior from a "class" into a concrete object. We just caused two objects to be linked to each other.

In fact, the secret, which eludes most JS developers, is that the `new Foo()` function calling had really almost nothing *direct* to do with the process of creating the link. **It was sort of an accidental side-effect.** `new Foo()` is an indirect, round-about way to end up with what we want: **a new object linked to another object**.

Can we get what we want in a more *direct* way? **Yes!** The hero is `Object.create(...)`. But we'll get to that in a little bit.

What's in a name?

In JavaScript, we don't make *copies* from one object ("class") to another ("instance"). We make *links* between objects. For the `[[Prototype]]` mechanism, visually, the arrows move from right to left, and from bottom to top.



This mechanism is often called "prototypal inheritance" (we'll explore the code in detail shortly), which is commonly said to be the dynamic-language version of "classical inheritance". It's an attempt to piggy-back on the common understanding of what "inheritance" means in the class-oriented world, but *tweak (read: pave over)* the understood semantics, to fit dynamic scripting.

The word "inheritance" has a very strong meaning (see Chapter 4), with plenty of mental precedent. Merely adding "prototypal" in front to distinguish the *actually nearly opposite* behavior in JavaScript has left in its wake nearly two decades of miry confusion.

I like to say that sticking "prototypal" in front of "inheritance" to drastically reverse its actual meaning is like holding an orange in one hand, an apple in the other, and insisting on calling the apple a "red orange". No matter what confusing label I put in front of it, that doesn't change the *fact* that one fruit is an apple and the other is an orange.

The better approach is to plainly call an apple an apple -- to use the most accurate and direct terminology. That makes it easier to understand both their similarities and their **many differences**, because we all have a simple, shared understanding of what "apple" means.

Because of the confusion and conflation of terms, I believe the label "prototypal inheritance" itself (and trying to mis-apply all its associated class-orientation terminology, like "class", "constructor", "instance", "polymorphism", etc) has done **more harm than good** in explaining how JavaScript's mechanism *really* works.

"Inheritance" implies a *copy* operation, and JavaScript doesn't copy object properties (natively, by default). Instead, JS creates a link between two objects, where one object can essentially *delegate* property/function access to another object. "Delegation" (see Chapter 6) is a much more accurate term for JavaScript's object-linking mechanism.

Another term which is sometimes thrown around in JavaScript is "differential inheritance". The idea here is that we describe an object's behavior in terms of what is *different* from a more general descriptor. For example, you explain that a car is a kind of vehicle, but one that has exactly 4 wheels, rather than re-describing all the specifics of what makes up a general vehicle (engine, etc).

If you try to think of any given object in JS as the sum total of all behavior that is *available* via delegation, and **in your mind you flatten** all that behavior into one tangible *thing*, then you can (sorta) see how "differential inheritance" might fit.

But just like with "prototypal inheritance", "differential inheritance" pretends that your mental model is more important than what is physically happening in the language. It overlooks the fact that object `B` is not actually differentially constructed, but is instead built with specific characteristics defined, alongside "holes" where nothing is defined. It is in these "holes" (gaps in, or lack of, definition) that delegation *can* take over and, on the fly, "fill them in" with delegated behavior.

The object is not, by native default, flattened into the single differential object, **through copying**, that the mental model of "differential inheritance" implies. As such, "differential inheritance" is just not as natural a fit for describing how JavaScript's `[[Prototype]]` mechanism actually works.

You *can choose* to prefer the "differential inheritance" terminology and mental model, as a matter of taste, but there's no denying the fact that it *only* fits the mental acrobatics in your mind, not the physical behavior in the engine.

"Constructors"

Let's go back to some earlier code:

```
function Foo() {  
    // ...  
}  
  
var a = new Foo();
```

What exactly leads us to think `Foo` is a "class"?

For one, we see the use of the `new` keyword, just like class-oriented languages do when they construct class instances. For another, it appears that we are in fact executing a *constructor* method of a class, because `Foo()` is actually a method that gets called, just like how a real class's constructor gets called when you instantiate that class.

To further the confusion of "constructor" semantics, the arbitrarily labeled `Foo.prototype` object has another trick up its sleeve. Consider this code:

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

Foo.prototype.constructor === Foo; // true

var a = new Foo();
a.constructor === Foo; // true
```

The `Foo.prototype` object by default (at declaration time on line 1 of the snippet!) gets a public, non-enumerable (see Chapter 3) property called `.constructor`, and this property is a reference back to the function (`Foo` in this case) that the object is associated with. Moreover, we see that object `a` created by the "constructor" call `new Foo()` *seems* to also have a property on it called `.constructor` which similarly points to "the function which created it".

Note: This is not actually true. `a` has no `.constructor` property on it, and though `a.constructor` does in fact resolve to the `Foo` function, "constructor" **does not actually mean** "was constructed by", as it appears. We'll explain this strangeness shortly.

Oh, yeah, also... by convention in the JavaScript world, "class"es are named with a capital letter, so the fact that it's `Foo` instead of `foo` is a strong clue that we intend it to be a "class". That's totally obvious to you, right!?

Note: This convention is so strong that many JS linters actually *complain* if you call `new` on a method with a lowercase name, or if we don't call `new` on a function that happens to start with a capital letter. That sort of boggles the mind that we struggle so much to get (fake) "class-orientation" *right* in JavaScript that we create linter rules to ensure we use capital letters, even though the capital letter doesn't mean **anything at all** to the JS engine.

Constructor Or Call?

In the above snippet, it's tempting to think that `Foo` is a "constructor", because we call it with `new` and we observe that it "constructs" an object.

In reality, `Foo` is no more a "constructor" than any other function in your program. Functions themselves are **not** constructors. However, when you put the `new` keyword in front of a normal function call, that makes that function call a "constructor call". In fact, `new` sort of hijacks any normal function and calls it in a fashion that constructs an object, **in addition to whatever else it was going to do**.

For example:

```
function NothingSpecial() {
    console.log( "Don't mind me!" );
}

var a = new NothingSpecial();
// "Don't mind me!"

a; // {}
```

`NothingSpecial` is just a plain old normal function, but when called with `new`, it *constructs* an object, almost as a side-effect, which we happen to assign to `a`. The `call` was a *constructor call*, but `NothingSpecial` is not, in and of itself, a *constructor*.

In other words, in JavaScript, it's most appropriate to say that a "constructor" is **any function called with the `new` keyword** in front of it.

Functions aren't constructors, but function calls are "constructor calls" if and only if `new` is used.

Mechanics

Are *those* the only common triggers for ill-fated "class" discussions in JavaScript?

Not quite. JS developers have strived to simulate as much as they can of class-orientation:

```
function Foo(name) {
    this.name = name;
}

Foo.prototype.myName = function() {
    return this.name;
};

var a = new Foo( "a" );
var b = new Foo( "b" );

a.myName(); // "a"
b.myName(); // "b"
```

This snippet shows two additional "class-orientation" tricks in play:

1. `this.name = name` : adds the `.name` property onto each object (`a` and `b`, respectively; see Chapter 2 about `this` binding), similar to how class instances encapsulate data values.

2. `Foo.prototype.myName = ...` : perhaps the more interesting technique, this adds a property (function) to the `Foo.prototype` object. Now, `a.myName()` works, but perhaps surprisingly. How?

In the above snippet, it's strongly tempting to think that when `a` and `b` are created, the properties/functions on the `Foo.prototype` object are *copied* over to each of `a` and `b` objects. **However, that's not what happens.**

At the beginning of this chapter, we explained the `[[Prototype]]` link, and how it provides the fall-back look-up steps if a property reference isn't found directly on an object, as part of the default `[[Get]]` algorithm.

So, by virtue of how they are created, `a` and `b` each end up with an internal `[[Prototype]]` linkage to `Foo.prototype`. When `myName` is not found on `a` or `b`, respectively, it's instead found (through delegation, see Chapter 6) on `Foo.prototype`.

"Constructor" Redux

Recall the discussion from earlier about the `.constructor` property, and how it *seems* like `a.constructor === Foo` being true means that `a` has an actual `.constructor` property on it, pointing at `Foo`? **Not correct.**

This is just unfortunate confusion. In actuality, the `.constructor` reference is also *delegated* up to `Foo.prototype`, which **happens to**, by default, have a `.constructor` that points at `Foo`.

It *seems* awfully convenient that an object `a` "constructed by" `Foo` would have access to a `.constructor` property that points to `Foo`. But that's nothing more than a false sense of security. It's a happy accident, almost tangentially, that `a.constructor` *happens* to point at `Foo` via this default `[[Prototype]]` delegation. There's actually several ways that the ill-fated assumption of `.constructor` meaning "was constructed by" can come back to bite you.

For one, the `.constructor` property on `Foo.prototype` is only there by default on the object created when `Foo` the function is declared. If you create a new object, and replace a function's default `.prototype` object reference, the new object will not by default magically get a `.constructor` on it.

Consider:

```

function Foo() { /* .. */ }

Foo.prototype = { /* .. */ }; // create a new prototype object

var a1 = new Foo();
a1.constructor === Foo; // false!
a1.constructor === Object; // true!

```

`Object(..)` didn't "construct" `a1` did it? It sure seems like `Foo()` "constructed" it. Many developers think of `Foo()` as doing the construction, but where everything falls apart is when you think "constructor" means "was constructed by", because by that reasoning, `a1.constructor` should be `Foo`, but it isn't!

What's happening? `a1` has no `.constructor` property, so it delegates up the `[[Prototype]]` chain to `Foo.prototype`. But that object doesn't have a `.constructor` either (like the default `Foo.prototype` object would have had!), so it keeps delegating, this time up to `Object.prototype`, the top of the delegation chain. *That* object indeed has a `.constructor` on it, which points to the built-in `Object(..)` function.

Misconception, busted.

Of course, you can add `.constructor` back to the `Foo.prototype` object, but this takes manual work, especially if you want to match native behavior and have it be non-enumerable (see Chapter 3).

For example:

```

function Foo() { /* .. */ }

Foo.prototype = { /* .. */ }; // create a new prototype object

// Need to properly "fix" the missing `constructor` property
// on the new object serving as `Foo.prototype`.
// See Chapter 3 for `defineProperty(..)`.

Object.defineProperty(Foo.prototype, "constructor", {
    enumerable: false,
    writable: true,
    configurable: true,
    value: Foo // point `constructor` at `Foo`
});
```

That's a lot of manual work to fix `.constructor`. Moreover, all we're really doing is perpetuating the misconception that "constructor" means "was constructed by". That's an expensive illusion.

The fact is, `.constructor` on an object arbitrarily points, by default, at a function who, reciprocally, has a reference back to the object -- a reference which it calls `.prototype`. The words "constructor" and "prototype" only have a loose default meaning that might or might not hold true later. The best thing to do is remind yourself, "constructor does not mean constructed by".

`.constructor` is not a magic immutable property. It *is* non-enumerable (see snippet above), but its value is writable (can be changed), and moreover, you can add or overwrite (intentionally or accidentally) a property of the name `constructor` on any object in any `[[Prototype]]` chain, with any value you see fit.

By virtue of how the `[[Get]]` algorithm traverses the `[[Prototype]]` chain, a `.constructor` property reference found anywhere may resolve quite differently than you'd expect.

See how arbitrary its meaning actually is?

The result? Some arbitrary object-property reference like `a1.constructor` cannot actually be *trusted* to be the assumed default function reference. Moreover, as we'll see shortly, just by simple omission, `a1.constructor` can even end up pointing somewhere quite surprising and insensible.

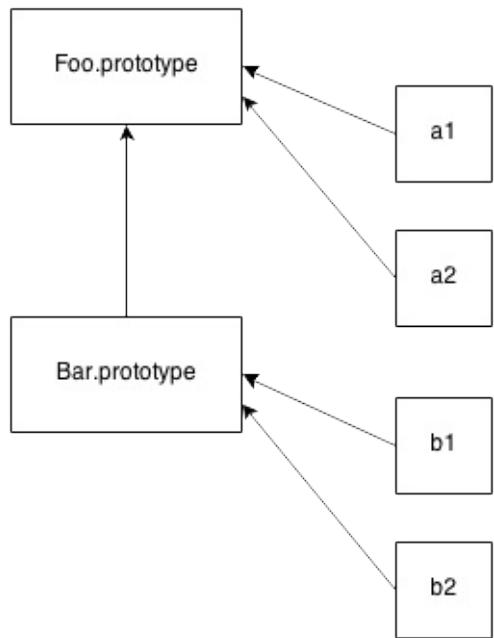
`.constructor` is extremely unreliable, and an unsafe reference to rely upon in your code.

Generally, such references should be avoided where possible.

"(Prototypal) Inheritance"

We've seen some approximations of "class" mechanics as typically hacked into JavaScript programs. But JavaScript "class"es would be rather hollow if we didn't have an approximation of "inheritance".

Actually, we've already seen the mechanism which is commonly called "prototypal inheritance" at work when `a` was able to "inherit from" `Foo.prototype`, and thus get access to the `myName()` function. But we traditionally think of "inheritance" as being a relationship between two "classes", rather than between "class" and "instance".



Recall this figure from earlier, which shows not only delegation from an object (aka, "instance") `a1` to object `Foo.prototype`, but from `Bar.prototype` to `Foo.prototype`, which somewhat resembles the concept of Parent-Child class inheritance. *Resembles*, except of course for the direction of the arrows, which show these are delegation links rather than copy operations.

And, here's the typical "prototype style" code that creates such links:

```

function Foo(name) {
    this.name = name;
}

Foo.prototype.myName = function() {
    return this.name;
};

function Bar(name,label) {
    Foo.call( this, name );
    this.label = label;
}

// here, we make a new `Bar.prototype`
// linked to `Foo.prototype`
Bar.prototype = Object.create( Foo.prototype );

// Beware! Now `Bar.prototype.constructor` is gone,
// and might need to be manually "fixed" if you're
// in the habit of relying on such properties!

Bar.prototype.myLabel = function() {
    return this.label;
};

var a = new Bar( "a", "obj a" );

a.myName(); // "a"
a.myLabel(); // "obj a"

```

Note: To understand why `this` points to `a` in the above code snippet, see Chapter 2.

The important part is `Bar.prototype = Object.create(Foo.prototype)`. `Object.create(...)` creates a "new" object out of thin air, and links that new object's internal `[[Prototype]]` to the object you specify (`Foo.prototype` in this case).

In other words, that line says: "make a *new* 'Bar dot prototype' object that's linked to 'Foo dot prototype'."

When `function Bar() { .. }` is declared, `Bar`, like any other function, has a `.prototype` link to its default object. But *that* object is not linked to `Foo.prototype` like we want. So, we create a *new* object that *is* linked as we want, effectively throwing away the original incorrectly-linked object.

Note: A common mis-conception/confusion here is that either of the following approaches would *also* work, but they do not work as you'd expect:

```
// doesn't work like you want!
Bar.prototype = Foo.prototype;

// works kinda like you want, but with
// side-effects you probably don't want :(
Bar.prototype = new Foo();
```

`Bar.prototype = Foo.prototype` doesn't create a new object for `Bar.prototype` to be linked to. It just makes `Bar.prototype` be another reference to `Foo.prototype`, which effectively links `Bar` directly to **the same object as** `Foo` links to: `Foo.prototype`. This means when you start assigning, like `Bar.prototype.myLabel = ...`, you're modifying **not a separate object** but *the shared* `Foo.prototype` object itself, which would affect any objects linked to `Foo.prototype`. This is almost certainly not what you want. If it *is* what you want, then you likely don't need `Bar` at all, and should just use only `Foo` and make your code simpler.

`Bar.prototype = new Foo()` **does in fact** create a new object which is duly linked to `Foo.prototype` as we'd want. But, it uses the `Foo(...)` "constructor call" to do it. If that function has any side-effects (such as logging, changing state, registering against other objects, **adding data properties to** `this`, etc.), those side-effects happen at the time of this linking (and likely against the wrong object!), rather than only when the eventual `Bar()` "descendants" are created, as would likely be expected.

So, we're left with using `Object.create(...)` to make a new object that's properly linked, but without having the side-effects of calling `Foo(...)`. The slight downside is that we have to create a new object, throwing the old one away, instead of modifying the existing default object we're provided.

It would be *nice* if there was a standard and reliable way to modify the linkage of an existing object. Prior to ES6, there's a non-standard and not fully-cross-browser way, via the `__proto__` property, which is settable. ES6 adds a `Object.setPrototypeOf(...)` helper utility, which does the trick in a standard and predictable way.

Compare the pre-ES6 and ES6-standardized techniques for linking `Bar.prototype` to `Foo.prototype`, side-by-side:

```
// pre-ES6
// throws away default existing `Bar.prototype`
Bar.prototype = Object.create( Foo.prototype );

// ES6+
// modifies existing `Bar.prototype`
Object.setPrototypeOf( Bar.prototype, Foo.prototype );
```

Ignoring the slight performance disadvantage (throwing away an object that's later garbage collected) of the `Object.create(...)` approach, it's a little bit shorter and may be perhaps a little easier to read than the ES6+ approach. But it's probably a syntactic wash either way.

Inspecting "Class" Relationships

What if you have an object like `a` and want to find out what object (if any) it delegates to? Inspecting an instance (just an object in JS) for its inheritance ancestry (delegation linkage in JS) is often called *introspection* (or *reflection*) in traditional class-oriented environments.

Consider:

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

Foo.prototype.blah = ...;

var a = new Foo();
```

How do we then introspect `a` to find out its "ancestry" (delegation linkage)? The first approach embraces the "class" confusion:

```
a instanceof Foo; // true
```

The `instanceof` operator takes a plain object as its left-hand operand and a **function** as its right-hand operand. The question `instanceof` answers is: **in the entire `[[Prototype]]` chain of `a`, does the object arbitrarily pointed to by `Foo.prototype` ever appear?**

Unfortunately, this means that you can only inquire about the "ancestry" of some object (`a`) if you have some **function** (`Foo`, with its attached `.prototype` reference) to test with. If you have two arbitrary objects, say `a` and `b`, and want to find out if *the objects* are related to each other through a `[[Prototype]]` chain, `instanceof` alone can't help.

Note: If you use the built-in `.bind(...)` utility to make a hard-bound function (see Chapter 2), the function created will not have a `.prototype` property. Using `instanceof` with such a function transparently substitutes the `.prototype` of the *target function* that the hard-bound function was created from.

It's fairly uncommon to use hard-bound functions as "constructor calls", but if you do, it will behave as if the original *target function* was invoked instead, which means that using `instanceof` with a hard-bound function also behaves according to the original function.

This snippet illustrates the ridiculousness of trying to reason about relationships between **two objects** using "class" semantics and `instanceof` :

```
// helper utility to see if `o1` is
// related to (delegates to) `o2`
function isRelatedTo(o1, o2) {
  function F(){}
  F.prototype = o2;
  return o1 instanceof F;
}

var a = {};
var b = Object.create( a );

isRelatedTo( b, a ); // true
```

Inside `isRelatedTo(..)` , we borrow a throw-away function `F` , reassign its `.prototype` to arbitrarily point to some object `o2` , then ask if `o1` is an "instance of" `F` . Obviously `o1` isn't *actually* inherited or descended or even constructed from `F` , so it should be clear why this kind of exercise is silly and confusing. **The problem comes down to the awkwardness of class semantics forced upon JavaScript**, in this case as revealed by the indirect semantics of `instanceof` .

The second, and much cleaner, approach to `[[Prototype]]` reflection is:

```
Foo.prototype.isPrototypeOf( a ); // true
```

Notice that in this case, we don't really care about (or even *need*) `Foo` , we just need an **object** (in our case, arbitrarily labeled `Foo.prototype`) to test against another **object**. The question `isPrototypeOf(..)` answers is: **in the entire `[[Prototype]]` chain of `a` , does `Foo.prototype` ever appear?**

Same question, and exact same answer. But in this second approach, we don't actually need the indirection of referencing a **function** (`Foo`) whose `.prototype` property will automatically be consulted.

We *just need two objects* to inspect a relationship between them. For example:

```
// Simply: does `b` appear anywhere in
// `c`'s [[Prototype]] chain?
b.isPrototypeOf( c );
```

Notice, this approach doesn't require a function ("class") at all. It just uses object references directly to `b` and `c`, and inquires about their relationship. In other words, our `isRelatedTo(...)` utility above is built-in to the language, and it's called `isPrototypeOf(..)`.

We can also directly retrieve the `[[Prototype]]` of an object. As of ES5, the standard way to do this is:

```
Object.getPrototypeOf( a );
```

And you'll notice that object reference is what we'd expect:

```
Object.getPrototypeOf( a ) === Foo.prototype; // true
```

Most browsers (not all!) have also long supported a non-standard alternate way of accessing the internal `[[Prototype]]`:

```
a.__proto__ === Foo.prototype; // true
```

The strange `__proto__` (not standardized until ES6!) property "magically" retrieves the internal `[[Prototype]]` of an object as a reference, which is quite helpful if you want to directly inspect (or even traverse: `__proto__.__proto__...`) the chain.

Just as we saw earlier with `.constructor`, `__proto__` doesn't actually exist on the object you're inspecting (`a` in our running example). In fact, it exists (non-enumerable; see Chapter 2) on the built-in `Object.prototype`, along with the other common utilities (`.toString()`, `.isPrototypeOf(..)`, etc).

Moreover, `__proto__` looks like a property, but it's actually more appropriate to think of it as a getter/setter (see Chapter 3).

Roughly, we could envision `__proto__` implemented (see Chapter 3 for object property definitions) like this:

```
Object.defineProperty( Object.prototype, "__proto__", {
    get: function() {
        return Object.getPrototypeOf( this );
    },
    set: function(o) {
        // setPrototypeOf(..) as of ES6
        Object.setPrototypeOf( this, o );
        return o;
    }
});
```

So, when we access (retrieve the value of) `a.__proto__`, it's like calling `a.__proto__()` (calling the getter function). *That* function call has `a` as its `this` even though the getter function exists on the `Object.prototype` object (see Chapter 2 for `this` binding rules), so it's just like saying `Object.getPrototypeOf(a)`.

`__proto__` is also a settable property, just like using ES6's `Object.setPrototypeOf(...)` shown earlier. However, generally you **should not change the `[[Prototype]]` of an existing object**.

There are some very complex, advanced techniques used deep in some frameworks that allow tricks like "subclassing" an `Array`, but this is commonly frowned on in general programming practice, as it usually leads to *much* harder to understand/maintain code.

Note: As of ES6, the `class` keyword will allow something that approximates "subclassing" of built-in's like `Array`. See Appendix A for discussion of the `class` syntax added in ES6.

The only other narrow exception (as mentioned earlier) would be setting the `[[Prototype]]` of a default function's `.prototype` object to reference some other object (besides `Object.prototype`). That would avoid replacing that default object entirely with a new linked object. Otherwise, **it's best to treat object `[[Prototype]]` linkage as a read-only characteristic** for ease of reading your code later.

Note: The JavaScript community unofficially coined a term for the double-underscore, specifically the leading one in properties like `__proto__`: "dunder". So, the "cool kids" in JavaScript would generally pronounce `__proto__` as "dunder proto".

Object Links

As we've now seen, the `[[Prototype]]` mechanism is an internal link that exists on one object which references some other object.

This linkage is (primarily) exercised when a property/method reference is made against the first object, and no such property/method exists. In that case, the `[[Prototype]]` linkage tells the engine to look for the property/method on the linked-to object. In turn, if that object cannot fulfill the look-up, its `[[Prototype]]` is followed, and so on. This series of links between objects forms what is called the "prototype chain".

Create() ing Links

We've thoroughly debunked why JavaScript's `[[Prototype]]` mechanism is **not** like *classes*, and we've seen how it instead creates **links** between proper objects.

What's the point of the `[[Prototype]]` mechanism? Why is it so common for JS developers to go to so much effort (emulating classes) in their code to wire up these linkages?

Remember we said much earlier in this chapter that `Object.create(..)` would be a hero? Now, we're ready to see how.

```
var foo = {  
    something: function() {  
        console.log( "Tell me something good..." );  
    }  
};  
  
var bar = Object.create( foo );  
  
bar.something(); // Tell me something good...
```

`Object.create(..)` creates a new object (`bar`) linked to the object we specified (`foo`), which gives us all the power (delegation) of the `[[Prototype]]` mechanism, but without any of the unnecessary complication of `new` functions acting as classes and constructor calls, confusing `.prototype` and `.constructor` references, or any of that extra stuff.

Note: `Object.create(null)` creates an object that has an empty (aka, `null`) `[[Prototype]]` linkage, and thus the object can't delegate anywhere. Since such an object has no prototype chain, the `instanceof` operator (explained earlier) has nothing to check, so it will always return `false`. These special empty- `[[Prototype]]` objects are often called "dictionaries" as they are typically used purely for storing data in properties, mostly because they have no possible surprise effects from any delegated properties/functions on the `[[Prototype]]` chain, and are thus purely flat data storage.

We don't *need* classes to create meaningful relationships between two objects. The only thing we should **really care about** is objects linked together for delegation, and `Object.create(..)` gives us that linkage without all the class cruft.

Object.create() Polyfilled

`Object.create(..)` was added in ES5. You may need to support pre-ES5 environments (like older IE's), so let's take a look at a simple **partial** polyfill for `Object.create(..)` that gives us the capability that we need even in those older JS environments:

```

if (!Object.create) {
    Object.create = function(o) {
        function F(){}
        F.prototype = o;
        return new F();
    };
}

```

This polyfill works by using a throw-away `F` function and overriding its `.prototype` property to point to the object we want to link to. Then we use `new F()` construction to make a new object that will be linked as we specified.

This usage of `Object.create(..)` is by far the most common usage, because it's the part that *can be* polyfilled. There's an additional set of functionality that the standard ES5 built-in `Object.create(..)` provides, which is **not polyfillable** for pre-ES5. As such, this capability is far-less commonly used. For completeness sake, let's look at that additional functionality:

```

var anotherObject = {
    a: 2
};

var myObject = Object.create( anotherObject, {
    b: {
        enumerable: false,
        writable: true,
        configurable: false,
        value: 3
    },
    c: {
        enumerable: true,
        writable: false,
        configurable: false,
        value: 4
    }
} );

myObject.hasOwnProperty( "a" ); // false
myObject.hasOwnProperty( "b" ); // true
myObject.hasOwnProperty( "c" ); // true

myObject.a; // 2
myObject.b; // 3
myObject.c; // 4

```

The second argument to `Object.create(..)` specifies property names to add to the newly created object, via declaring each new property's *property descriptor* (see Chapter 3). Because polyfilling property descriptors into pre-ES5 is not possible, this additional

functionality on `Object.create(...)` also cannot be polyfilled.

The vast majority of usage of `Object.create(...)` uses the polyfill-safe subset of functionality, so most developers are fine with using the **partial polyfill** in pre-ES5 environments.

Some developers take a much stricter view, which is that no function should be polyfilled unless it can be *fully* polyfilled. Since `Object.create(...)` is one of those partial-polyfill'able utilities, this narrower perspective says that if you need to use any of the functionality of `Object.create(...)` in a pre-ES5 environment, instead of polyfilling, you should use a custom utility, and stay away from using the name `Object.create` entirely. You could instead define your own utility, like:

```
function createAndLinkObject(o) {
  function F(){}
  F.prototype = o;
  return new F();
}

var anotherObject = {
  a: 2
};

var myObject = createAndLinkObject( anotherObject );

myObject.a; // 2
```

I do not share this strict opinion. I fully endorse the common partial-polyfill of `Object.create(...)` as shown above, and using it in your code even in pre-ES5. I'll leave it to you to make your own decision.

Links As Fallbacks?

It may be tempting to think that these links between objects *primarily* provide a sort of fallback for "missing" properties or methods. While that may be an observed outcome, I don't think it represents the right way of thinking about `[[Prototype]]`.

Consider:

```

var anotherObject = {
  cool: function() {
    console.log( "cool!" );
  }
};

var myObject = Object.create( anotherObject );

myObject.cool(); // "cool!"

```

That code will work by virtue of `[[Prototype]]`, but if you wrote it that way so that `anotherObject` was acting as a fallback **just in case** `myObject` couldn't handle some property/method that some developer may try to call, odds are that your software is going to be a bit more "magical" and harder to understand and maintain.

That's not to say there aren't cases where fallbacks are an appropriate design pattern, but it's not very common or idiomatic in JS, so if you find yourself doing so, you might want to take a step back and reconsider if that's really appropriate and sensible design.

Note: In ES6, an advanced functionality called `Proxy` is introduced which can provide something of a "method not found" type of behavior. `Proxy` is beyond the scope of this book, but will be covered in detail in a later book in the "*You Don't Know JS*" series.

Don't miss an important but nuanced point here.

Designing software where you intend for a developer to, for instance, call `myObject.cool()` and have that work even though there is no `cool()` method on `myObject` introduces some "magic" into your API design that can be surprising for future developers who maintain your software.

You can however design your API with less "magic" to it, but still take advantage of the power of `[[Prototype]]` linkage.

```

var anotherObject = {
  cool: function() {
    console.log( "cool!" );
  }
};

var myObject = Object.create( anotherObject );

myObject.doCool = function() {
  this.cool(); // internal delegation!
};

myObject.doCool(); // "cool!"

```

Here, we call `myObject.doCool()`, which is a method that *actually exists* on `myObject`, making our API design more explicit (less "magical"). *Internally*, our implementation follows the **delegation design pattern** (see Chapter 6), taking advantage of `[[Prototype]]` delegation to `anotherObject.cool()`.

In other words, delegation will tend to be less surprising/confusing if it's an internal implementation detail rather than plainly exposed in your API design. We will expound on **delegation** in great detail in the next chapter.

Review (TL;DR)

When attempting a property access on an object that doesn't have that property, the object's internal `[[Prototype]]` linkage defines where the `[[Get]]` operation (see Chapter 3) should look next. This cascading linkage from object to object essentially defines a "prototype chain" (somewhat similar to a nested scope chain) of objects to traverse for property resolution.

All normal objects have the built-in `object.prototype` as the top of the prototype chain (like the global scope in scope look-up), where property resolution will stop if not found anywhere prior in the chain. `toString()`, `valueOf()`, and several other common utilities exist on this `object.prototype` object, explaining how all objects in the language are able to access them.

The most common way to get two objects linked to each other is using the `new` keyword with a function call, which among its four steps (see Chapter 2), it creates a new object linked to another object.

The "another object" that the new object is linked to happens to be the object referenced by the arbitrarily named `.prototype` property of the function called with `new`. Functions called with `new` are often called "constructors", despite the fact that they are not actually instantiating a class as *constructors* do in traditional class-oriented languages.

While these JavaScript mechanisms can seem to resemble "class instantiation" and "class inheritance" from traditional class-oriented languages, the key distinction is that in JavaScript, no copies are made. Rather, objects end up linked to each other via an internal `[[Prototype]]` chain.

For a variety of reasons, not the least of which is terminology precedent, "inheritance" (and "prototypal inheritance") and all the other OO terms just do not make sense when considering how JavaScript *actually* works (not just applied to our forced mental models).

Instead, "delegation" is a more appropriate term, because these relationships are not *copies* but delegation **links**.

Chapter 6: Behavior Delegation

In Chapter 5, we addressed the `[[Prototype]]` mechanism in detail, and *why* it's confusing and inappropriate (despite countless attempts for nearly two decades) to describe it as "class" or "inheritance". We trudged through not only the fairly verbose syntax (`.prototype` littering the code), but the various gotchas (like surprising `.constructor` resolution or ugly pseudo-polymorphic syntax). We explored variations of the "mixin" approach, which many people use to attempt to smooth over such rough areas.

It's a common reaction at this point to wonder why it has to be so complex to do something seemingly so simple. Now that we've pulled back the curtain and seen just how dirty it all gets, it's not a surprise that most JS developers never dive this deep, and instead relegate such mess to a "class" library to handle it for them.

I hope by now you're not content to just gloss over and leave such details to a "black box" library. Let's now dig into how we *could and should* be thinking about the object `[[Prototype]]` mechanism in JS, in a **much simpler and more straightforward way** than the confusion of classes.

As a brief review of our conclusions from Chapter 5, the `[[Prototype]]` mechanism is an internal link that exists on one object which references another object.

This linkage is exercised when a property/method reference is made against the first object, and no such property/method exists. In that case, the `[[Prototype]]` linkage tells the engine to look for the property/method on the linked-to object. In turn, if that object cannot fulfill the look-up, its `[[Prototype]]` is followed, and so on. This series of links between objects forms what is called the "prototype chain".

In other words, the actual mechanism, the essence of what's important to the functionality we can leverage in JavaScript, is **all about objects being linked to other objects**.

That single observation is fundamental and critical to understanding the motivations and approaches for the rest of this chapter!

Towards Delegation-Oriented Design

To properly focus our thoughts on how to use `[[Prototype]]` in the most straightforward way, we must recognize that it represents a fundamentally different design pattern from classes (see Chapter 4).

Note: Some principles of class-oriented design are still very valid, so don't toss out everything you know (just most of it!). For example, *encapsulation* is quite powerful, and is compatible (though not as common) with delegation.

We need to try to change our thinking from the class/inheritance design pattern to the behavior delegation design pattern. If you have done most or all of your programming in your education/career thinking in classes, this may be uncomfortable or feel unnatural. You may need to try this mental exercise quite a few times to get the hang of this very different way of thinking.

I'm going to walk you through some theoretical exercises first, then we'll look side-by-side at a more concrete example to give you practical context for your own code.

Class Theory

Let's say we have several similar tasks ("XYZ", "ABC", etc) that we need to model in our software.

With classes, the way you design the scenario is: define a general parent (base) class like `Task`, defining shared behavior for all the "alike" tasks. Then, you define child classes `XYZ` and `ABC`, both of which inherit from `Task`, and each of which adds specialized behavior to handle their respective tasks.

Importantly, the class design pattern will encourage you that to get the most out of inheritance, you will want to employ method overriding (and polymorphism), where you override the definition of some general `Task` method in your `XYZ` task, perhaps even making use of `super` to call to the base version of that method while adding more behavior to it. **You'll likely find quite a few places** where you can "abstract" out general behavior to the parent class and specialize (override) it in your child classes.

Here's some loose pseudo-code for that scenario:

```
class Task {  
    id;  
  
    // constructor `Task()`  
    Task(ID) { id = ID; }  
    outputTask() { output( id ); }  
}  
  
class XYZ inherits Task {  
    label;  
  
    // constructor `XYZ()`  
    XYZ(ID,Label) { super( ID ); label = Label; }  
    outputTask() { super(); output( label ); }  
}  
  
class ABC inherits Task {  
    // ...  
}
```

Now, you can instantiate one or more **copies** of the `xyz` child class, and use those instance(s) to perform task "XYZ". These instances have **copies both** of the general `Task` defined behavior as well as the specific `xyz` defined behavior. Likewise, instances of the `ABC` class would have copies of the `Task` behavior and the specific `ABC` behavior. After construction, you will generally only interact with these instances (and not the classes), as the instances each have copies of all the behavior you need to do the intended task.

Delegation Theory

But now let's try to think about the same problem domain, but using *behavior delegation* instead of *classes*.

You will first define an **object** (not a class, nor a `function` as most JS's would lead you to believe) called `Task`, and it will have concrete behavior on it that includes utility methods that various tasks can use (read: *delegate to!*). Then, for each task ("XYZ", "ABC"), you define an **object** to hold that task-specific data/behavior. You **link** your task-specific object(s) to the `Task` utility object, allowing them to delegate to it when they need to.

Basically, you think about performing task "XYZ" as needing behaviors from two sibling/peer objects (`xyz` and `Task`) to accomplish it. But rather than needing to compose them together, via class copies, we can keep them in their separate objects, and we can allow `xyz` object to **delegate to** `Task` when needed.

Here's some simple code to suggest how you accomplish that:

```

var Task = {
    setID: function(ID) { this.id = ID; },
    outputID: function() { console.log( this.id ); }
};

// make `XYZ` delegate to `Task`
var XYZ = Object.create( Task );

XYZ.prepareTask = function(ID, Label) {
    this.setID( ID );
    this.label = Label;
};

XYZ.outputTaskDetails = function() {
    this.outputID();
    console.log( this.label );
};

// ABC = Object.create( Task );
// ABC ... = ...

```

In this code, `Task` and `XYZ` are not classes (or functions), they're **just objects**. `XYZ` is set up via `Object.create(...)` to `[[Prototype]]` delegate to the `Task` object (see Chapter 5).

As compared to class-orientation (aka, OO -- object-oriented), I call this style of code **"OLOO"** (objects-linked-to-other-objects). All we *really* care about is that the `XYZ` object delegates to the `Task` object (as does the `ABC` object).

In JavaScript, the `[[Prototype]]` mechanism links **objects** to other **objects**. There are no abstract mechanisms like "classes", no matter how much you try to convince yourself otherwise. It's like paddling a canoe upstream: you *can* do it, but you're *choosing* to go against the natural current, so it's obviously **going to be harder to get where you're going**.

Some other differences to note with **OLOO style code**:

1. Both `id` and `label` data members from the previous class example are data properties directly on `XYZ` (neither is on `Task`). In general, with `[[Prototype]]` delegation involved, **you want state to be on the delegators** (`XYZ`, `ABC`), not on the delegate (`Task`).
2. With the class design pattern, we intentionally named `outputTask` the same on both parent (`Task`) and child (`XYZ`), so that we could take advantage of overriding (polymorphism). In behavior delegation, we do the opposite: **we avoid if at all possible naming things the same** at different levels of the `[[Prototype]]` chain (called shadowing -- see Chapter 5), because having those name collisions creates awkward/brittle syntax to disambiguate references (see Chapter 4), and we want to avoid that if we can.

This design pattern calls for less of general method names which are prone to overriding and instead more of descriptive method names, *specific* to the type of behavior each object is doing. **This can actually create easier to understand/maintain code**, because the names of methods (not only at definition location but strewn throughout other code) are more obvious (self documenting).

3. `this.setID(ID);` inside of a method on the `xyz` object first looks on `xyz` for `setID(...)`, but since it doesn't find a method of that name on `xyz`, `[[Prototype]]` *delegation* means it can follow the link to `Task` to look for `setID(...)`, which it of course finds. Moreover, because of implicit call-site `this` binding rules (see Chapter 2), when `setID(...)` runs, even though the method was found on `Task`, the `this` binding for that function call is `xyz` exactly as we'd expect and want. We see the same thing with `this.outputID()` later in the code listing.

In other words, the general utility methods that exist on `Task` are available to us while interacting with `xyz`, because `xyz` can delegate to `Task`.

Behavior Delegation means: let some object (`xyz`) provide a delegation (to `Task`) for property or method references if not found on the object (`xyz`).

This is an *extremely powerful* design pattern, very distinct from the idea of parent and child classes, inheritance, polymorphism, etc. Rather than organizing the objects in your mind vertically, with Parents flowing down to Children, think of objects side-by-side, as peers, with any direction of delegation links between the objects as necessary.

Note: Delegation is more properly used as an internal implementation detail rather than exposed directly in the API design. In the above example, we don't necessarily *intend* with our API design for developers to call `xyz.setID()` (though we can, of course!). We sorta *hide* the delegation as an internal detail of our API, where `xyz.prepareTask(...)` delegates to `Task.setID(...)`. See the "Links As Fallbacks?" discussion in Chapter 5 for more detail.

Mutual Delegation (Disallowed)

You cannot create a *cycle* where two or more objects are mutually delegated (bi-directionally) to each other. If you make `B` linked to `A`, and then try to link `A` to `B`, you will get an error.

It's a shame (not terribly surprising, but mildly annoying) that this is disallowed. If you made a reference to a property/method which didn't exist in either place, you'd have an infinite recursion on the `[[Prototype]]` loop. But if all references were strictly present, then `B` could delegate to `A`, and vice versa, and it *could* work. This would mean you could use either object to delegate to the other, for various tasks. There are a few niche use-cases where this might be helpful.

But it's disallowed because engine implementors have observed that it's more performant to check for (and reject!) the infinite circular reference once at set-time rather than needing to have the performance hit of that guard check every time you look-up a property on an object.

Debugged

We'll briefly cover a subtle detail that can be confusing to developers. In general, the JS specification does not control how browser developer tools should represent specific values/structures to a developer, so each browser/engine is free to interpret such things as they see fit. As such, browsers/tools *don't always agree*. Specifically, the behavior we will now examine is currently observed only in Chrome's Developer Tools.

Consider this traditional "class constructor" style JS code, as it would appear in the *console* of Chrome Developer Tools:

```
function Foo() {}

var a1 = new Foo();

a1; // Foo {}
```

Let's look at the last line of that snippet: the output of evaluating the `a1` expression, which prints `Foo {}`. If you try this same code in Firefox, you will likely see `Object {}`. Why the difference? What do these outputs mean?

Chrome is essentially saying "`{}` is an empty object that was constructed by a function with name 'Foo'". Firefox is saying "`{}` is an empty object of general construction from Object". The subtle difference is that Chrome is actively tracking, as an *internal property*, the name of the actual function that did the construction, whereas other browsers don't track that additional information.

It would be tempting to attempt to explain this with JavaScript mechanisms:

```
function Foo() {}

var a1 = new Foo();

a1.constructor; // Foo(){}
a1.constructor.name; // "Foo"
```

So, is that how Chrome is outputting "Foo", by simply examining the object's `.constructor.name`? Confusingly, the answer is both "yes" and "no".

Consider this code:

```
function Foo() {}

var a1 = new Foo();

Foo.prototype.constructor = function Gotcha() {};

a1.constructor; // Gotcha(){}
a1.constructor.name; // "Gotcha"

a1; // Foo {}
```

Even though we change `a1.constructor.name` to legitimately be something else ("Gotcha"), Chrome's console still uses the "Foo" name.

So, it would appear the answer to previous question (does it use `.constructor.name` ?) is **no**, it must track it somewhere else, internally.

But, Not so fast! Let's see how this kind of behavior works with OOO-style code:

```
var Foo = {};

var a1 = Object.create( Foo );

a1; // Object {}

Object.defineProperty( Foo, "constructor", {
    enumerable: false,
    value: function Gotcha(){}
});

a1; // Gotcha {}
```

Ah-ha! **Gotcha!** Here, Chrome's console **did** find and use the `.constructor.name` . Actually, while writing this book, this exact behavior was identified as a bug in Chrome, and by the time you're reading this, it may have already been fixed. So you may instead have seen the corrected `a1; // Object {}` .

Aside from that bug, the internal tracking (apparently only for debug output purposes) of the "constructor name" that Chrome does (shown in the earlier snippets) is an intentional Chrome-only extension of behavior beyond what the JS specification calls for.

If you don't use a "constructor" to make your objects, as we've discouraged with OOO-style code here in this chapter, then you'll get objects that Chrome does *not* track an internal "constructor name" for, and such objects will correctly only be outputted as "Object {}", meaning "object generated from Object() construction".

Don't think this represents a drawback of OLOO-style coding. When you code with OLOO and behavior delegation as your design pattern, *who* "constructed" (that is, *which function* was called with `new`?) some object is an irrelevant detail. Chrome's specific internal "constructor name" tracking is really only useful if you're fully embracing "class-style" coding, but is moot if you're instead embracing OLOO delegation.

Mental Models Compared

Now that you can see a difference between "class" and "delegation" design patterns, at least theoretically, let's see the implications these design patterns have on the mental models we use to reason about our code.

We'll examine some more theoretical ("Foo", "Bar") code, and compare both ways (OO vs. OLOO) of implementing the code. The first snippet uses the classical ("prototypal") OO style:

```
function Foo(who) {
    this.me = who;
}

Foo.prototype.identify = function() {
    return "I am " + this.me;
};

function Bar(who) {
    Foo.call( this, who );
}

Bar.prototype = Object.create( Foo.prototype );

Bar.prototype.speak = function() {
    alert( "Hello, " + this.identify() + "." );
};

var b1 = new Bar( "b1" );
var b2 = new Bar( "b2" );

b1.speak();
b2.speak();
```

Parent class `Foo`, inherited by child class `Bar`, which is then instantiated twice as `b1` and `b2`. What we have is `b1` delegating to `Bar.prototype` which delegates to `Foo.prototype`. This should look fairly familiar to you, at this point. Nothing too ground-breaking going on.

Now, let's implement **the exact same functionality** using OLOO style code:

```

var Foo = {
    init: function(who) {
        this.me = who;
    },
    identify: function() {
        return "I am " + this.me;
    }
};

var Bar = Object.create( Foo );

Bar.speak = function() {
    alert( "Hello, " + this.identify() + "." );
};

var b1 = Object.create( Bar );
b1.init( "b1" );
var b2 = Object.create( Bar );
b2.init( "b2" );

b1.speak();
b2.speak();

```

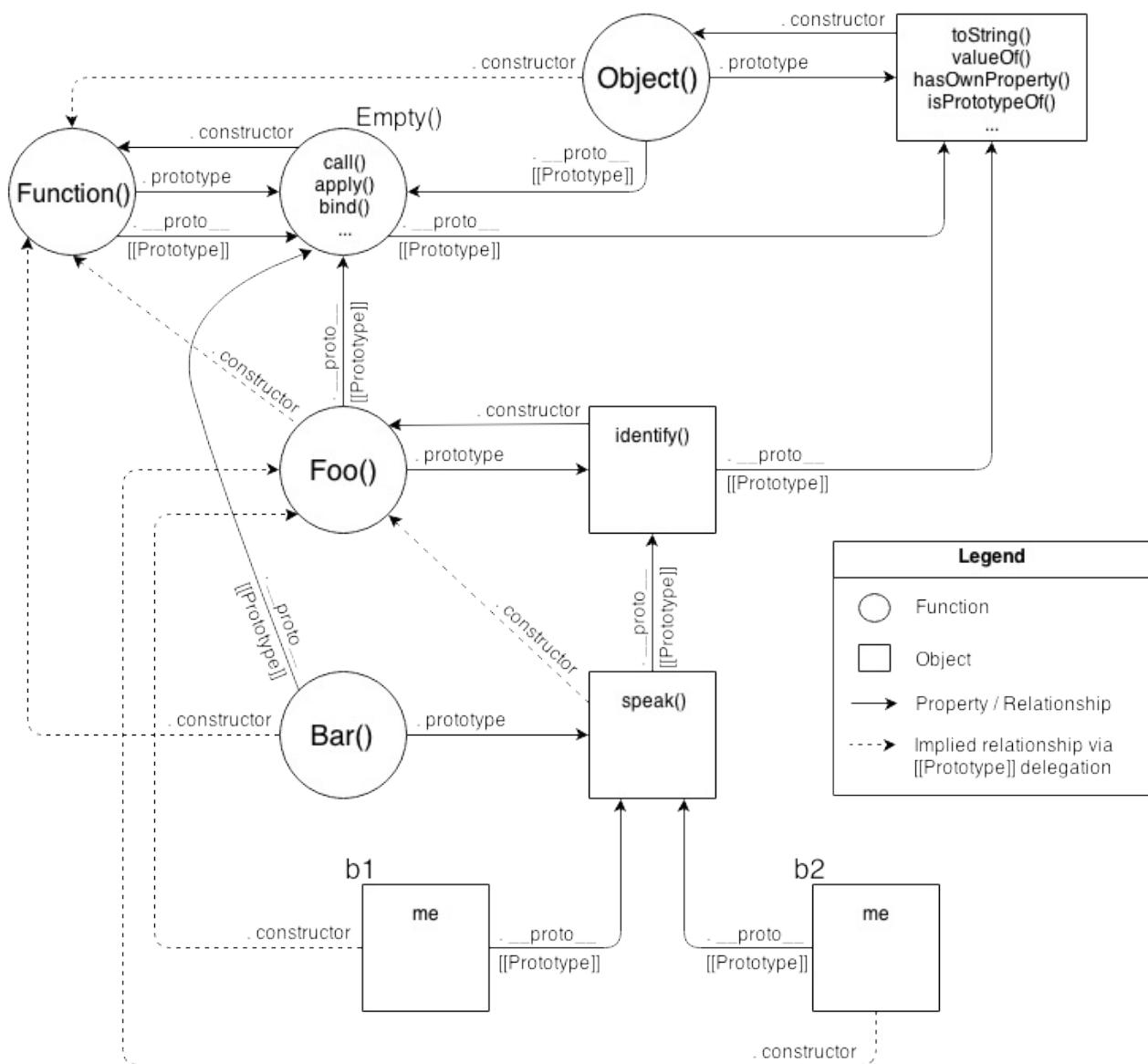
We take exactly the same advantage of `[[Prototype]]` delegation from `b1` to `Bar` to `Foo` as we did in the previous snippet between `b1`, `Bar.prototype`, and `Foo.prototype`. **We still have the same 3 objects linked together.**

But, importantly, we've greatly simplified *all the other stuff* going on, because now we just set up **objects** linked to each other, without needing all the cruft and confusion of things that look (but don't behave!) like classes, with constructors and prototypes and `new` calls.

Ask yourself: if I can get the same functionality with OOO style code as I do with "class" style code, but OOO is simpler and has less things to think about, **isn't OOO better?**

Let's examine the mental models involved between these two snippets.

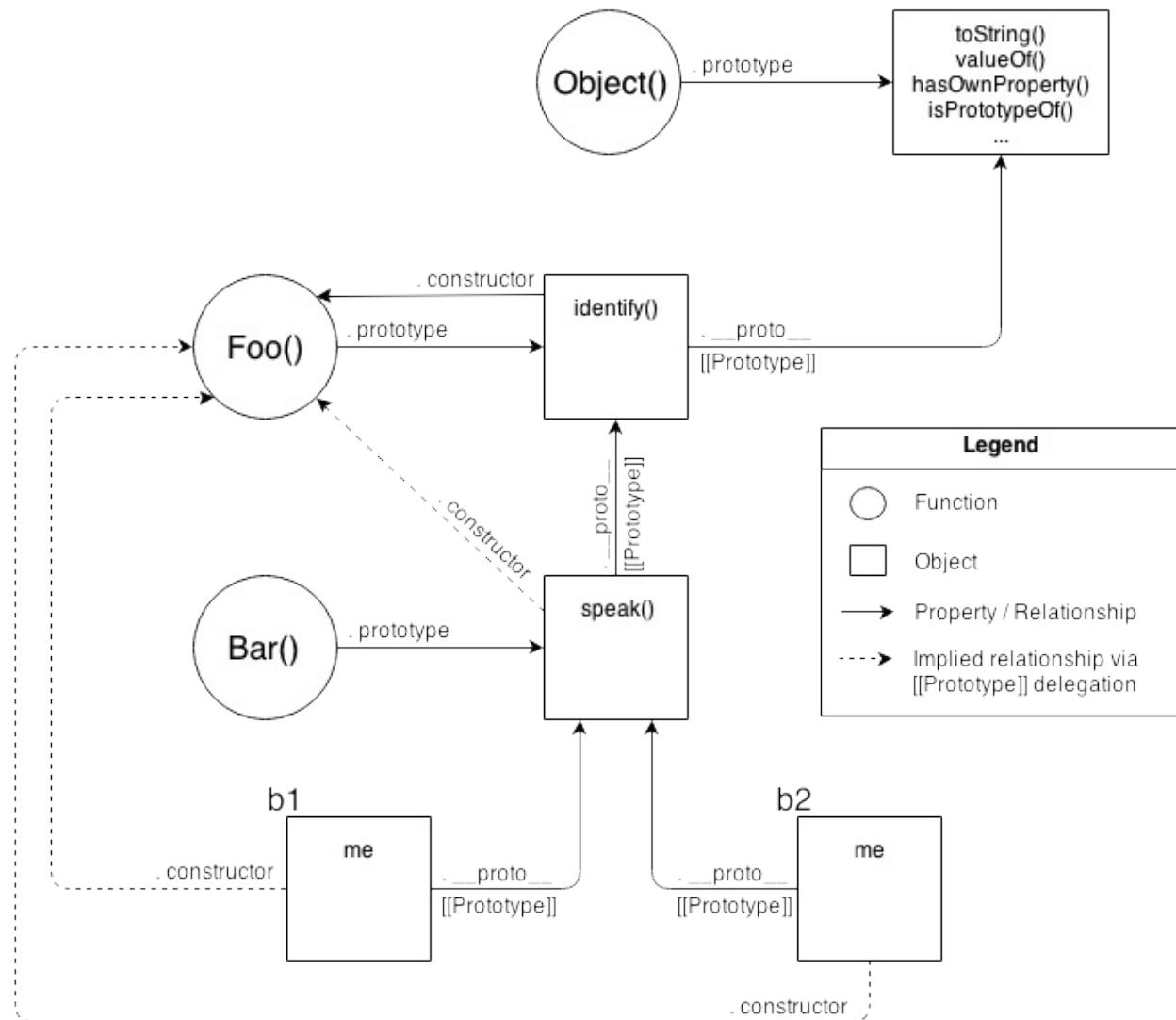
First, the class-style code snippet implies this mental model of entities and their relationships:



Actually, that's a little unfair/misleading, because it's showing a lot of extra detail that you don't *technically* need to know at all times (though you *do* need to understand it!). One take-away is that it's quite a complex series of relationships. But another take-away: if you spend the time to follow those relationship arrows around, **there's an amazing amount of internal consistency** in JS's mechanisms.

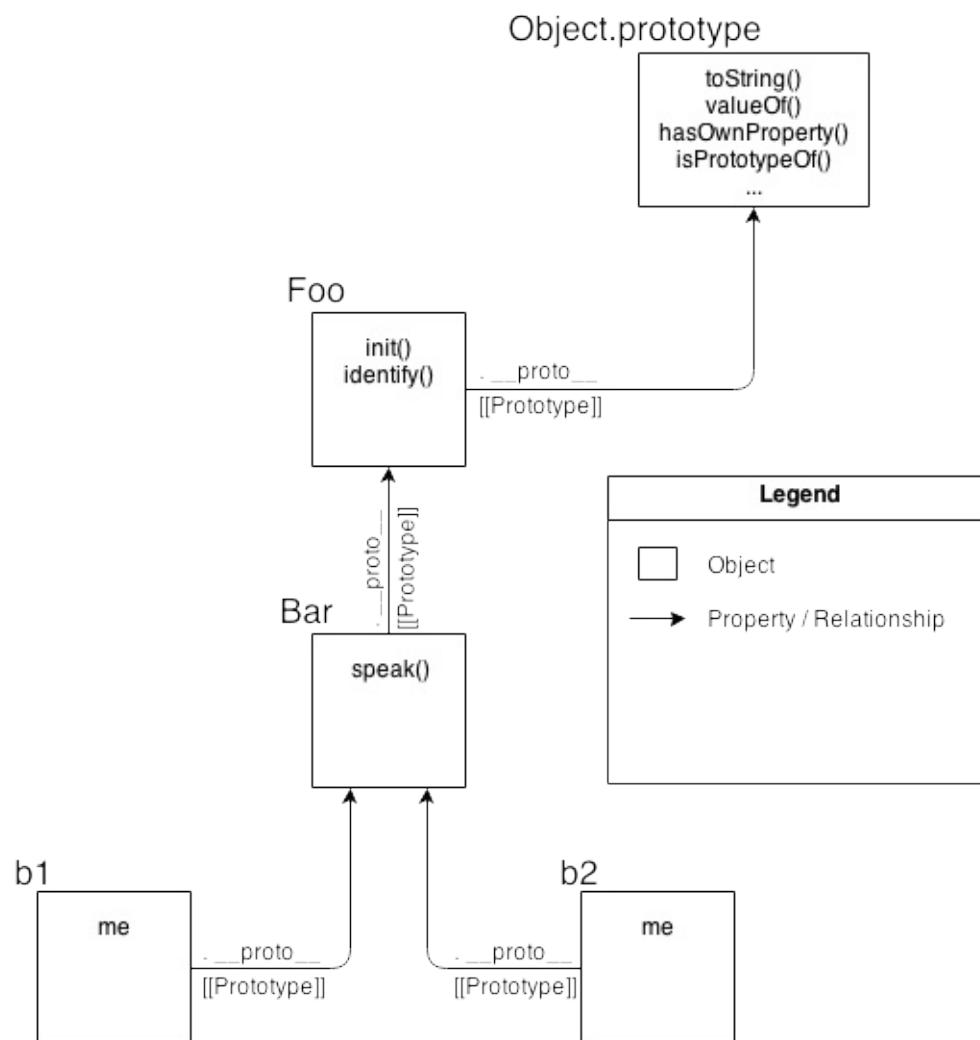
For instance, the ability of a JS function to access `call(..)`, `apply(..)`, and `bind(..)` (see Chapter 2) is because functions themselves are objects, and function-objects also have a `[[Prototype]]` linkage, to the `Function.prototype` object, which defines those default methods that any function-object can delegate to. JS can do those things, *and you can too!*.

OK, let's now look at a *slightly* simplified version of that diagram which is a little more "fair" for comparison -- it shows only the *relevant* entities and relationships.



Still pretty complex, eh? The dotted lines are depicting the implied relationships when you setup the "inheritance" between `Foo.prototype` and `Bar.prototype` and haven't yet fixed the **missing** `.constructor` property reference (see "Constructor Redux" in Chapter 5). Even with those dotted lines removed, the mental model is still an awful lot to juggle every time you work with object linkages.

Now, let's look at the mental model for OLOO-style code:



As you can see comparing them, it's quite obvious that OOO-style code has *vastly less stuff* to worry about, because OOO-style code embraces the **fact** that the only thing we ever really cared about was the **objects linked to other objects**.

All the other "class" cruft was a confusing and complex way of getting the same end result. Remove that stuff, and things get much simpler (without losing any capability).

Classes vs. Objects

We've just seen various theoretical explorations and mental models of "classes" vs. "behavior delegation". But, let's now look at more concrete code scenarios to show how'd you actually use these ideas.

We'll first examine a typical scenario in front-end web dev: creating UI widgets (buttons, drop-downs, etc.).

Widget "Classes"

Because you're probably still so used to the OO design pattern, you'll likely immediately think of this problem domain in terms of a parent class (perhaps called `Widget`) with all the common base widget behavior, and then child derived classes for specific widget types (like `Button`).

Note: We're going to use jQuery here for DOM and CSS manipulation, only because it's a detail we don't really care about for the purposes of our current discussion. None of this code cares which JS framework (jQuery, Dojo, YUI, etc), if any, you might solve such mundane tasks with.

Let's examine how we'd implement the "class" design in classic-style pure JS without any "class" helper library or syntax:

```

// Parent class
function Widget(width,height) {
    this.width = width || 50;
    this.height = height || 50;
    this.$elem = null;
}

Widget.prototype.render = function($where){
    if (this.$elem) {
        this.$elem.css( {
            width: this.width + "px",
            height: this.height + "px"
        } ).appendTo( $where );
    }
};

// Child class
function Button(width,height,label) {
    // "super" constructor call
    Widget.call( this, width, height );
    this.label = label || "Default";

    this.$elem = $( "<button>" ).text( this.label );
}

// make `Button` "inherit" from `Widget`
Button.prototype = Object.create( Widget.prototype );

// override base "inherited" `render(..)`
Button.prototype.render = function($where) {
    // "super" call
    Widget.prototype.render.call( this, $where );
    this.$elem.click( this.onClick.bind( this ) );
};

Button.prototype.onClick = function(evt) {
    console.log( "Button '" + this.label + "' clicked!" );
};

$( document ).ready( function(){
    var $body = $( document.body );
    var btn1 = new Button( 125, 30, "Hello" );
    var btn2 = new Button( 150, 40, "World" );

    btn1.render( $body );
    btn2.render( $body );
} );

```

OO design patterns tell us to declare a base `render(..)` in the parent class, then override it in our child class, but not to replace it per se, rather to augment the base functionality with button-specific behavior.

Notice the ugliness of *explicit pseudo-polymorphism* (see Chapter 4) with `Widget.call` and `Widget.prototype.render.call` references for faking "super" calls from the child "class" methods back up to the "parent" class base methods. Yuck.

ES6 class Sugar

We cover ES6 `class` syntax sugar in detail in Appendix A, but let's briefly demonstrate how we'd implement the same code using `class`:

```
class Widget {
    constructor(width,height) {
        this.width = width || 50;
        this.height = height || 50;
        this.$elem = null;
    }
    render($where){
        if (this.$elem) {
            this.$elem.css( {
                width: this.width + "px",
                height: this.height + "px"
            }).appendTo( $where );
        }
    }
}

class Button extends Widget {
    constructor(width,height,label) {
        super( width, height );
        this.label = label || "Default";
        this.$elem = $( "<button>" ).text( this.label );
    }
    render($where) {
        super.render( $where );
        this.$elem.click( this.onClick.bind( this ) );
    }
    onClick(evt) {
        console.log( "Button '" + this.label + "' clicked!" );
    }
}

$( document ).ready( function(){
    var $body = $( document.body );
    var btn1 = new Button( 125, 30, "Hello" );
    var btn2 = new Button( 150, 40, "World" );

    btn1.render( $body );
    btn2.render( $body );
});
```

Undoubtedly, a number of the syntax uglies of the previous classical approach have been smoothed over with ES6's `class`. The presence of a `super(...)` in particular seems quite nice (though when you dig into it, it's not all roses!).

Despite syntactic improvements, **these are not real classes**, as they still operate on top of the `[[Prototype]]` mechanism. They suffer from all the same mental-model mismatches we explored in Chapters 4, 5 and thus far in this chapter. Appendix A will expound on the ES6 `class` syntax and its implications in detail. We'll see why solving syntax hiccups doesn't substantially solve our class confusions in JS, though it makes a valiant effort masquerading as a solution!

Whether you use the classic prototypal syntax or the new ES6 sugar, you've still made a *choice* to model the problem domain (UI widgets) with "classes". And as the previous few chapters try to demonstrate, this *choice* in JavaScript is opting you into extra headaches and mental tax.

Delegating Widget Objects

Here's our simpler `Widget / Button` example, using **OLOO style delegation**:

```

var Widget = {
    init: function(width,height){
        this.width = width || 50;
        this.height = height || 50;
        this.$elem = null;
    },
    insert: function($where){
        if (this.$elem) {
            this.$elem.css( {
                width: this.width + "px",
                height: this.height + "px"
            }).appendTo( $where );
        }
    }
};

var Button = Object.create( Widget );

Button.setup = function(width,height,label){
    // delegated call
    this.init( width, height );
    this.label = label || "Default";

    this.$elem = $( "<button>" ).text( this.label );
};

Button.build = function($where) {
    // delegated call
    this.insert( $where );
    this.$elem.click( this.onClick.bind( this ) );
};

Button.onClick = function(evt) {
    console.log( "Button '" + this.label + "' clicked!" );
};

$( document ).ready( function(){
    var $body = $( document.body );

    var btn1 = Object.create( Button );
    btn1.setup( 125, 30, "Hello" );

    var btn2 = Object.create( Button );
    btn2.setup( 150, 40, "World" );

    btn1.build( $body );
    btn2.build( $body );
} );

```

With this OOO-style approach, we don't think of `Widget` as a parent and `Button` as a child. Rather, `Widget` is just an object and is sort of a utility collection that any specific type of widget might want to delegate to, and `Button` is also just a stand-alone object

(with a delegation link to `Widget`, of course!).

From a design pattern perspective, we **didn't** share the same method name `render(...)` in both objects, the way classes suggest, but instead we chose different names (`insert(...)` and `build(...)`) that were more descriptive of what task each does specifically. The *initialization* methods are called `init(...)` and `setup(...)`, respectively, for the same reasons.

Not only does this delegation design pattern suggest different and more descriptive names (rather than shared and more generic names), but doing so with OOOO happens to avoid the ugliness of the explicit pseudo-polymorphic calls (`Widget.call` and `Widget.prototype.render.call`), as you can see by the simple, relative, delegated calls to `this.init(...)` and `this.insert(...)`.

Syntactically, we also don't have any constructors, `.prototype` or `new` present, as they are, in fact, just unnecessary cruft.

Now, if you're paying close attention, you may notice that what was previously just one call (`var btn1 = new Button(...)`) is now two calls (`var btn1 = Object.create(Button)` and `btn1.setup(...)`). Initially this may seem like a drawback (more code).

However, even this is something that's a **pro of OOOO style code** as compared to classical prototype style code. How?

With class constructors, you are "forced" (not really, but strongly suggested) to do both construction and initialization in the same step. However, there are many cases where being able to do these two steps separately (as you do with OOOO!) is more flexible.

For example, let's say you create all your instances in a pool at the beginning of your program, but you wait to initialize them with specific setup until they are pulled from the pool and used. We showed the two calls happening right next to each other, but of course they can happen at very different times and in very different parts of our code, as needed.

OOOO supports *better* the principle of separation of concerns, where creation and initialization are not necessarily conflated into the same operation.

Simpler Design

In addition to OOOO providing ostensibly simpler (and more flexible!) code, behavior delegation as a pattern can actually lead to simpler code architecture. Let's examine one last example that illustrates how OOOO simplifies your overall design.

The scenario we'll examine is two controller objects, one for handling the login form of a web page, and another for actually handling the authentication (communication) with the server.

We'll need a utility helper for making the Ajax communication to the server. We'll use jQuery (though any framework would do fine), since it handles not only the Ajax for us, but it returns a promise-like answer so that we can listen for the response in our calling code with

```
.then(...).
```

Note: We don't cover Promises here, but we will cover them in a future title of the "*You Don't Know JS*" series.

Following the typical class design pattern, we'll break up the task into base functionality in a class called `Controller`, and then we'll derive two child classes, `LoginController` and `AuthController`, which both inherit from `Controller` and specialize some of those base behaviors.

```
// Parent class
function Controller() {
    this.errors = [];
}
Controller.prototype.showDialog = function(title,msg) {
    // display title & message to user in dialog
};
Controller.prototype.success = function(msg) {
    this.showDialog( "Success", msg );
};
Controller.prototype.failure = function(err) {
    this.errors.push( err );
    this.showDialog( "Error", err );
};
```

```
// Child class
function LoginController() {
    Controller.call( this );
}

// Link child class to parent
LoginController.prototype = Object.create( Controller.prototype );
LoginController.prototype.getUser = function() {
    return document.getElementById( "login_username" ).value;
};

LoginController.prototype.getPassword = function() {
    return document.getElementById( "login_password" ).value;
};

LoginController.prototype.validateEntry = function(user,pw) {
    user = user || this.getUser();
    pw = pw || this.getPassword();

    if (!(user && pw)) {
        return this.failure( "Please enter a username & password!" );
    }
    else if (pw.length < 5) {
        return this.failure( "Password must be 5+ characters!" );
    }

    // got here? validated!
    return true;
};

// Override to extend base `failure()`
LoginController.prototype.failure = function(err) {
    // "super" call
    Controller.prototype.failure.call( this, "Login invalid: " + err );
};
```

```

// Child class
function AuthController(login) {
    Controller.call( this );
    // in addition to inheritance, we also need composition
    this.login = login;
}

// Link child class to parent
AuthController.prototype = Object.create( Controller.prototype );
AuthController.prototype.server = function(url,data) {
    return $.ajax( {
        url: url,
        data: data
    } );
};

AuthController.prototype.checkAuth = function() {
    var user = this.login.getUser();
    var pw = this.login.getPassword();

    if (this.login.validateEntry( user, pw )) {
        this.server( "/check-auth",{
            user: user,
            pw: pw
        })
        .then( this.success.bind( this ) )
        .fail( this.failure.bind( this ) );
    }
};

// Override to extend base `success()`
AuthController.prototype.success = function() {
    // "super" call
    Controller.prototype.success.call( this, "Authenticated!" );
};

// Override to extend base `failure()`
AuthController.prototype.failure = function(err) {
    // "super" call
    Controller.prototype.failure.call( this, "Auth Failed: " + err );
};

```

```

var auth = new AuthController(
    // in addition to inheritance, we also need composition
    new LoginController()
);
auth.checkAuth();

```

We have base behaviors that all controllers share, which are `success(..)` , `failure(..)` and `showDialog(..)` . Our child classes `LoginController` and `AuthController` override `failure(..)` and `success(..)` to augment the default base class behavior. Also note that

`AuthController` needs an instance of `LoginController` to interact with the login form, so that becomes a member data property.

The other thing to mention is that we chose some *composition* to sprinkle in on top of the inheritance. `AuthController` needs to know about `LoginController`, so we instantiate it (`new LoginController()`) and keep a class member property called `this.login` to reference it, so that `AuthController` can invoke behavior on `LoginController`.

Note: There *might* have been a slight temptation to make `AuthController` inherit from `LoginController`, or vice versa, such that we had *virtual composition* through the inheritance chain. But this is a strongly clear example of what's wrong with class inheritance as *the* model for the problem domain, because neither `AuthController` nor `LoginController` are specializing base behavior of the other, so inheritance between them makes little sense except if classes are your only design pattern. Instead, we layered in some simple *composition* and now they can cooperate, while still both benefiting from the inheritance from the parent base `Controller`.

If you're familiar with class-oriented (OO) design, this should all look pretty familiar and natural.

De-class-ified

But, **do we really need to model this problem** with a parent `Controller` class, two child classes, **and some composition?** Is there a way to take advantage of OLOO-style behavior delegation and have a *much* simpler design? **Yes!**

```
var LoginController = {
    errors: [],
    getUser: function() {
        return document.getElementById( "login_username" ).value;
    },
    getPassword: function() {
        return document.getElementById( "login_password" ).value;
    },
    validateEntry: function(user,pw) {
        user = user || this.getUser();
        pw = pw || this.getPassword();

        if (!(user && pw)) {
            return this.failure( "Please enter a username & password!" );
        }
        else if (pw.length < 5) {
            return this.failure( "Password must be 5+ characters!" );
        }

        // got here? validated!
        return true;
    },
    showDialog: function(title,msg) {
        // display success message to user in dialog
    },
    failure: function(err) {
        this.errors.push( err );
        this.showDialog( "Error", "Login invalid: " + err );
    }
};
```

```
// Link `AuthController` to delegate to `LoginController`
var AuthController = Object.create( LoginController );

AuthController.errors = [];
AuthController.checkAuth = function() {
    var user = this.getUser();
    var pw = this.getPassword();

    if (this.validateEntry( user, pw )) {
        this.server( "/check-auth",{
            user: user,
            pw: pw
        } )
        .then( this.accepted.bind( this ) )
        .fail( this.rejected.bind( this ) );
    }
};

AuthController.server = function(url,data) {
    return $.ajax( {
        url: url,
        data: data
    } );
};

AuthController.accepted = function() {
    this.showDialog( "Success", "Authenticated!" )
};

AuthController.rejected = function(err) {
    this.failure( "Auth Failed: " + err );
};
```

Since `AuthController` is just an object (so is `LoginController`), we don't need to instantiate (like `new AuthController()`) to perform our task. All we need to do is:

```
AuthController.checkAuth();
```

Of course, with OOO, if you do need to create one or more additional objects in the delegation chain, that's easy, and still doesn't require anything like class instantiation:

```
var controller1 = Object.create( AuthController );
var controller2 = Object.create( AuthController );
```

With behavior delegation, `AuthController` and `LoginController` are **just objects**, *horizontal* peers of each other, and are not arranged or related as parents and children in class-orientation. We somewhat arbitrarily chose to have `AuthController` delegate to `LoginController` -- it would have been just as valid for the delegation to go the reverse direction.

The main takeaway from this second code listing is that we only have two entities (`LoginController` and `AuthController`), **not three** as before.

We didn't need a base `Controller` class to "share" behavior between the two, because delegation is a powerful enough mechanism to give us the functionality we need. We also, as noted before, don't need to instantiate our classes to work with them, because there are no classes, **just the objects themselves**. Furthermore, there's no need for *composition* as delegation gives the two objects the ability to cooperate *differentially* as needed.

Lastly, we avoided the polymorphism pitfalls of class-oriented design by not having the names `success(..)` and `failure(..)` be the same on both objects, which would have required ugly explicit pseudopolymorphism. Instead, we called them `accepted()` and `rejected(..)` on `AuthController` -- slightly more descriptive names for their specific tasks.

Bottom line: we end up with the same capability, but a (significantly) simpler design. That's the power of OOO-style code and the power of the *behavior delegation* design pattern.

Nicer Syntax

One of the nicer things that makes ES6's `class` so deceptively attractive (see Appendix A on why to avoid it!) is the short-hand syntax for declaring class methods:

```
class Foo {  
    methodName() { /* ... */ }  
}
```

We get to drop the word `function` from the declaration, which makes JS developers everywhere cheer!

And you may have noticed and been frustrated that the suggested OOO syntax above has lots of `function` appearances, which seems like a bit of a detractor to the goal of OOO simplification. **But it doesn't have to be that way!**

As of ES6, we can use *concise method declarations* in any object literal, so an object in OOO style can be declared this way (same short-hand sugar as with `class` body syntax):

```
var LoginController = {
  errors: [],
  getUser() { // Look ma, no `function`!
    // ...
  },
  getPassword() {
    // ...
  }
  // ...
};
```

About the only difference is that object literals will still require `,` comma separators between elements whereas `class` syntax doesn't. Pretty minor concession in the whole scheme of things.

Moreover, as of ES6, the clunkier syntax you use (like for the `AuthController` definition), where you're assigning properties individually and not using an object literal, can be re-written using an object literal (so that you can use concise methods), and you can just modify that object's `[[Prototype]]` with `Object.setPrototypeOf(..)`, like this:

```
// use nicer object literal syntax w/ concise methods!
var AuthController = {
  errors: [],
  checkAuth() {
    // ...
  },
  server(url,data) {
    // ...
  }
  // ...
};

// NOW, link `AuthController` to delegate to `LoginController`
Object.setPrototypeOf( AuthController, LoginController );
```

OLOO-style as of ES6, with concise methods, **is a lot friendlier** than it was before (and even then, it was much simpler and nicer than classical prototype-style code). **You don't have to opt for class** (complexity) to get nice clean object syntax!

Unlexical

There *is* one drawback to concise methods that's subtle but important to note. Consider this code:

```
var Foo = {
  bar() { /*...*/ },
  baz: function baz() { /*...*/ }
};
```

Here's the syntactic de-sugaring that expresses how that code will operate:

```
var Foo = {
  bar: function() { /*...*/ },
  baz: function baz() { /*...*/ }
};
```

See the difference? The `bar()` short-hand became an *anonymous function expression* (`function()...`) attached to the `bar` property, because the function object itself has no name identifier. Compare that to the manually specified *named function expression* (`function baz()...`) which has a lexical name identifier `baz` in addition to being attached to a `.baz` property.

So what? In the "Scope & Closures" title of this "You Don't Know JS" book series, we cover the three main downsides of *anonymous function expressions* in detail. We'll just briefly repeat them so we can compare to the concise method short-hand.

Lack of a `name` identifier on an anonymous function:

1. makes debugging stack traces harder
2. makes self-referencing (recursion, event (un)binding, etc) harder
3. makes code (a little bit) harder to understand

Items 1 and 3 don't apply to concise methods.

Even though the de-sugaring uses an *anonymous function expression* which normally would have no `name` in stack traces, concise methods are specified to set the internal `name` property of the function object accordingly, so stack traces should be able to use it (though that's implementation dependent so not guaranteed).

Item 2 is, unfortunately, **still a drawback to concise methods**. They will not have a lexical identifier to use as a self-reference. Consider:

```
var Foo = {
  bar: function(x) {
    if (x < 10) {
      return Foo.bar( x * 2 );
    }
    return x;
  },
  baz: function baz(x) {
    if (x < 10) {
      return baz( x * 2 );
    }
    return x;
  }
};
```

The manual `Foo.bar(x*2)` reference above kind of suffices in this example, but there are many cases where a function wouldn't necessarily be able to do that, such as cases where the function is being shared in delegation across different objects, using `this` binding, etc. You would want to use a real self-reference, and the function object's `name` identifier is the best way to accomplish that.

Just be aware of this caveat for concise methods, and if you run into such issues with lack of self-reference, make sure to forgo the concise method syntax **just for that declaration** in favor of the manual *named function expression* declaration form: `baz: function baz(){...}`.

Introspection

If you've spent much time with class oriented programming (either in JS or other languages), you're probably familiar with *type introspection*: inspecting an instance to find out what *kind* of object it is. The primary goal of *type introspection* with class instances is to reason about the structure/capabilities of the object based on *how it was created*.

Consider this code which uses `instanceof` (see Chapter 5) for introspecting on an object `a1` to infer its capability:

```

function Foo() {
    // ...
}
Foo.prototype.something = function(){
    // ...
}

var a1 = new Foo();

// later

if (a1 instanceof Foo) {
    a1.something();
}

```

Because `Foo.prototype` (not `Foo`!) is in the `[[Prototype]]` chain (see Chapter 5) of `a1`, the `instanceof` operator (confusingly) pretends to tell us that `a1` is an instance of the `Foo` "class". With this knowledge, we then assume that `a1` has the capabilities described by the `Foo` "class".

Of course, there is no `Foo` class, only a plain old normal function `Foo`, which happens to have a reference to an arbitrary object (`Foo.prototype`) that `a1` happens to be delegation-linked to. By its syntax, `instanceof` pretends to be inspecting the relationship between `a1` and `Foo`, but it's actually telling us whether `a1` and (the arbitrary object referenced by) `Foo.prototype` are related.

The semantic confusion (and indirection) of `instanceof` syntax means that to use `instanceof`-based introspection to ask if object `a1` is related to the capabilities object in question, you *have to* have a function that holds a reference to that object -- you can't just directly ask if the two objects are related.

Recall the abstract `Foo` / `Bar` / `b1` example from earlier in this chapter, which we'll abbreviate here:

```

function Foo() { /* ... */ }
Foo.prototype...

function Bar() { /* ... */ }
Bar.prototype = Object.create( Foo.prototype );

var b1 = new Bar( "b1" );

```

For *type introspection* purposes on the entities in that example, using `instanceof` and `.prototype` semantics, here are the various checks you might need to perform:

```
// relating `Foo` and `Bar` to each other
Bar.prototype instanceof Foo; // true
Object.getPrototypeOf( Bar.prototype ) === Foo.prototype; // true
Foo.prototype.isPrototypeOf( Bar.prototype ); // true

// relating `b1` to both `Foo` and `Bar`
b1 instanceof Foo; // true
b1 instanceof Bar; // true
Object.getPrototypeOf( b1 ) === Bar.prototype; // true
Foo.prototype.isPrototypeOf( b1 ); // true
Bar.prototype.isPrototypeOf( b1 ); // true
```

It's fair to say that some of that kinda sucks. For instance, intuitively (with classes) you might want to be able to say something like `Bar instanceof Foo` (because it's easy to mix up what "instance" means to think it includes "inheritance"), but that's not a sensible comparison in JS. You have to do `Bar.prototype instanceof Foo` instead.

Another common, but perhaps less robust, pattern for *type introspection*, which many devs seem to prefer over `instanceof`, is called "duck typing". This term comes from the adage, "if it looks like a duck, and it quacks like a duck, it must be a duck".

Example:

```
if (a1.something) {
  a1.something();
}
```

Rather than inspecting for a relationship between `a1` and an object that holds the delegatable `something()` function, we assume that the test for `a1.something` passing means `a1` has the capability to call `.something()` (regardless of if it found the method directly on `a1` or delegated to some other object). In and of itself, that assumption isn't so risky.

But "duck typing" is often extended to make **other assumptions about the object's capabilities** besides what's being tested, which of course introduces more risk (aka, brittle design) into the test.

One notable example of "duck typing" comes with ES6 Promises (which as an earlier note explained are not being covered in this book).

For various reasons, there's a need to determine if any arbitrary object reference *is a Promise*, but the way that test is done is to check if the object happens to have a `then()` function present on it. In other words, **if any object** happens to have a `then()` method, ES6 Promises will assume unconditionally that the object **is a "thenable"** and therefore will expect it to behave conformantly to all standard behaviors of Promises.

If you have any non-Promise object that happens for whatever reason to have a `then()` method on it, you are strongly advised to keep it far away from the ES6 Promise mechanism to avoid broken assumptions.

That example clearly illustrates the perils of "duck typing". You should only use such approaches sparingly and in controlled conditions.

Turning our attention once again back to OOO-style code as presented here in this chapter, *type introspection* turns out to be much cleaner. Let's recall (and abbreviate) the `Foo` / `Bar` / `b1` OOO example from earlier in the chapter:

```
var Foo = { /* ... */ };

var Bar = Object.create( Foo );
Bar...

var b1 = Object.create( Bar );
```

Using this OOO approach, where all we have are plain objects that are related via `[[Prototype]]` delegation, here's the quite simplified *type introspection* we might use:

```
// relating `Foo` and `Bar` to each other
Foo.isPrototypeOf( Bar ); // true
Object.getPrototypeOf( Bar ) === Foo; // true

// relating `b1` to both `Foo` and `Bar`
Foo.isPrototypeOf( b1 ); // true
Bar.isPrototypeOf( b1 ); // true
Object.getPrototypeOf( b1 ) === Bar; // true
```

We're not using `instanceof` anymore, because it's confusingly pretending to have something to do with classes. Now, we just ask the (informally stated) question, "are you a prototype of me?" There's no more indirection necessary with stuff like `Foo.prototype` or the painfully verbose `Foo.prototype.isPrototypeOf(...)`.

I think it's fair to say these checks are significantly less complicated/confusing than the previous set of introspection checks. **Yet again, we see that OOO is simpler than (but with all the same power of) class-style coding in JavaScript.**

Review (TL;DR)

Classes and inheritance are a design pattern you can *choose*, or *not choose*, in your software architecture. Most developers take for granted that classes are the only (proper) way to organize code, but here we've seen there's another less-commonly talked about pattern that's actually quite powerful: **behavior delegation**.

Behavior delegation suggests objects as peers of each other, which delegate amongst themselves, rather than parent and child class relationships. JavaScript's `[[Prototype]]` mechanism is, by its very designed nature, a behavior delegation mechanism. That means we can either choose to struggle to implement class mechanics on top of JS (see Chapters 4 and 5), or we can just embrace the natural state of `[[Prototype]]` as a delegation mechanism.

When you design code with objects only, not only does it simplify the syntax you use, but it can actually lead to simpler code architecture design.

OLOO (objects-linked-to-other-objects) is a code style which creates and relates objects directly without the abstraction of classes. OLOO quite naturally implements `[[Prototype]]` - based behavior delegation.

Appendix A: ES6 class

If there's any take-away message from the second half of this book (Chapters 4-6), it's that classes are an optional design pattern for code (not a necessary given), and that furthermore they are often quite awkward to implement in a `[[Prototype]]` language like JavaScript.

This awkwardness is *not* just about syntax, although that's a big part of it. Chapters 4 and 5 examined quite a bit of syntactic ugliness, from verbosity of `.prototype` references cluttering the code, to *explicit pseudo-polymorphism* (see Chapter 4) when you give methods the same name at different levels of the chain and try to implement a polymorphic reference from a lower-level method to a higher-level method. `.constructor` being wrongly interpreted as "was constructed by" and yet being unreliable for that definition is yet another syntactic ugly.

But the problems with class design are much deeper. Chapter 4 points out that classes in traditional class-oriented languages actually produce a *copy* action from parent to child to instance, whereas in `[[Prototype]]`, the action is **not** a copy, but rather the opposite -- a delegation link.

When compared to the simplicity of OOO-style code and behavior delegation (see Chapter 6), which embrace `[[Prototype]]` rather than hide from it, classes stand out as a sore thumb in JS.

class

But we *don't* need to re-argue that case again. I re-mention those issues briefly only so that you keep them fresh in your mind now that we turn our attention to the ES6 `class` mechanism. We'll demonstrate here how it works, and look at whether or not `class` does anything substantial to address any of those "class" concerns.

Let's revisit the `Widget / Button` example from Chapter 6:

```

class Widget {
    constructor(width,height) {
        this.width = width || 50;
        this.height = height || 50;
        this.$elem = null;
    }
    render($where){
        if (this.$elem) {
            this.$elem.css( {
                width: this.width + "px",
                height: this.height + "px"
            } ).appendTo( $where );
        }
    }
}

class Button extends Widget {
    constructor(width,height,label) {
        super( width, height );
        this.label = label || "Default";
        this.$elem = $( "<button>" ).text( this.label );
    }
    render($where) {
        super.render( $where );
        this.$elem.click( this.onClick.bind( this ) );
    }
    onClick(evt) {
        console.log( "Button '" + this.label + "' clicked!" );
    }
}

```

Beyond this syntax *looking* nicer, what problems does ES6 solve?

1. There's no more (well, sorta, see below!) references to `.prototype` cluttering the code.
2. `Button` is declared directly to "inherit from" (aka `extends`) `Widget`, instead of needing to use `Object.create(..)` to replace a `.prototype` object that's linked, or having to set with `__proto__` or `Object.setPrototypeOf(..)`.
3. `super(..)` now gives us a very helpful **relative polymorphism** capability, so that any method at one level of the chain can refer relatively one level up the chain to a method of the same name. This includes a solution to the note from Chapter 4 about the weirdness of constructors not belonging to their class, and so being unrelated -- `super()` works inside constructors exactly as you'd expect.
4. `class` literal syntax has no affordance for specifying properties (only methods). This might seem limiting to some, but it's expected that the vast majority of cases where a property (state) exists elsewhere but the end-chain "instances", this is usually a mistake and surprising (as it's state that's implicitly "shared" among all "instances"). So, one could say the `class` syntax is protecting you from mistakes.

5. `extends` lets you extend even built-in object (sub)types, like `Array` or `RegExp`, in a very natural way. Doing so without `class .. extends` has long been an exceedingly complex and frustrating task, one that only the most adept of framework authors have ever been able to accurately tackle. Now, it will be rather trivial!

In all fairness, those are some substantial solutions to many of the most obvious (syntactic) issues and surprises people have with classical prototype-style code.

class Gotchas

It's not all bubblegum and roses, though. There are still some deep and profoundly troubling issues with using "classes" as a design pattern in JS.

Firstly, the `class` syntax may convince you a new "class" mechanism exists in JS as of ES6. **Not so.** `class` is, mostly, just syntactic sugar on top of the existing `[[Prototype]]` (delegation!) mechanism.

That means `class` is not actually copying definitions statically at declaration time the way it does in traditional class-oriented languages. If you change/replace a method (on purpose or by accident) on the parent "class", the child "class" and/or instances will still be "affected", in that they didn't get copies at declaration time, they are all still using the live-delegation model based on `[[Prototype]]`:

```
class C {
  constructor() {
    this.num = Math.random();
  }
  rand() {
    console.log( "Random: " + this.num );
  }
}

var c1 = new C();
c1.rand(); // "Random: 0.4324299..."

C.prototype.rand = function() {
  console.log( "Random: " + Math.round( this.num * 1000 ) );
};

var c2 = new C();
c2.rand(); // "Random: 867"

c1.rand(); // "Random: 432" -- oops!!!
```

This only seems like reasonable behavior *if you already know* about the delegation nature of things, rather than expecting *copies* from "real classes". So the question to ask yourself is, why are you choosing `class` syntax for something fundamentally different from classes?

Doesn't the ES6 `class` syntax **just make it harder** to see and understand the difference between traditional classes and delegated objects?

`class` syntax *does not* provide a way to declare class member properties (only methods). So if you need to do that to track shared state among instances, then you end up going back to the ugly `.prototype` syntax, like this:

```
class C {
  constructor() {
    // make sure to modify the shared state,
    // not set a shadowed property on the
    // instances!
    C.prototype.count++;

    // here, `this.count` works as expected
    // via delegation
    console.log( "Hello: " + this.count );
  }
}

// add a property for shared state directly to
// prototype object
C.prototype.count = 0;

var c1 = new C();
// Hello: 1

var c2 = new C();
// Hello: 2

c1.count === 2; // true
c1.count === c2.count; // true
```

The biggest problem here is that it betrays the `class` syntax by exposing (leakage!) `.prototype` as an implementation detail.

But, we also still have the surprise gotcha that `this.count++` would implicitly create a separate shadowed `.count` property on both `c1` and `c2` objects, rather than updating the shared state. `class` offers us no consolation from that issue, except (presumably) to imply by lack of syntactic support that you shouldn't be doing that *at all*.

Moreover, accidental shadowing is still a hazard:

```

class C {
  constructor(id) {
    // oops, gotcha, we're shadowing `id()` method
    // with a property value on the instance
    this.id = id;
  }
  id() {
    console.log("Id: " + this.id);
  }
}

var c1 = new C("c1");
c1.id(); // TypeError -- `c1.id` is now the string "c1"

```

There's also some very subtle nuanced issues with how `super` works. You might assume that `super` would be bound in an analogous way to how `this` gets bound (see Chapter 2), which is that `super` would always be bound to one level higher than whatever the current method's position in the `[[Prototype]]` chain is.

However, for performance reasons (`this` binding is already expensive), `super` is not bound dynamically. It's bound sort of "statically", as declaration time. No big deal, right?

Ehh... maybe, maybe not. If you, like most JS devs, start assigning functions around to different objects (which came from `class` definitions), in various different ways, you probably won't be very aware that in all those cases, the `super` mechanism under the covers is having to be re-bound each time.

And depending on what sorts of syntactic approaches you take to these assignments, there may very well be cases where the `super` can't be properly bound (at least, not where you suspect), so you may (at time of writing, TC39 discussion is ongoing on the topic) have to manually bind `super` with `toMethod(...)` (kinda like you have to do `bind(...)` for `this` -- see Chapter 2).

You're used to being able to assign around methods to different objects to *automatically* take advantage of the dynamism of `this` via the *implicit binding* rule (see Chapter 2). But the same will likely not be true with methods that use `super`.

Consider what `super` should do here (against `D` and `E`):

```

class P {
    foo() { console.log( "P.foo" ); }
}

class C extends P {
    foo() {
        super();
    }
}

var c1 = new C();
c1.foo(); // "P.foo"

var D = {
    foo: function() { console.log( "D.foo" ); }
};

var E = {
    foo: C.prototype.foo
};

// Link E to D for delegation
Object.setPrototypeOf( E, D );

E.foo(); // "P.foo"

```

If you were thinking (quite reasonably!) that `super` would be bound dynamically at call-time, you might expect that `super()` would automatically recognize that `E` delegates to `D`, so `E.foo()` using `super()` should call to `D.foo()`.

Not so. For performance pragmatism reasons, `super` is not *late bound* (aka, dynamically bound) like `this` is. Instead it's derived at call-time from `[[HomeObject]].[[Prototype]]`, where `[[HomeObject]]` is statically bound at creation time.

In this particular case, `super()` is still resolving to `P.foo()`, since the method's `[[HomeObject]]` is still `C` and `C.[[Prototype]]` is `P`.

There will *probably* be ways to manually address such gotchas. Using `toMethod(..)` to bind/rebind a method's `[[HomeObject]]` (along with setting the `[[Prototype]]` of that object!) appears to work in this scenario:

```

var D = {
  foo: function() { console.log("D.foo"); }
};

// Link E to D for delegation
var E = Object.create( D );

// manually bind `foo`'s `[[HomeObject]]` as
// `E`, and `E.[[Prototype]]` is `D`, so thus
// `super()` is `D.foo()`
E.foo = C.prototype.foo.toMethod( E, "foo" );

E.foo(); // "D.foo"

```

Note: `toMethod(...)` clones the method, and takes `homeObject` as its first parameter (which is why we pass `E`), and the second parameter (optionally) sets a `name` for the new method (which keep at "foo").

It remains to be seen if there are other corner case gotchas that devs will run into beyond this scenario. Regardless, you will have to be diligent and stay aware of which places the engine automatically figures out `super` for you, and which places you have to manually take care of it. **Ugh!**

Static > Dynamic?

But the biggest problem of all about ES6 `class` is that all these various gotchas mean `class` sorta opts you into a syntax which seems to imply (like traditional classes) that once you declare a `class`, it's a static definition of a (future instantiated) thing. You completely lose sight of the fact that `c` is an object, a concrete thing, which you can directly interact with.

In traditional class-oriented languages, you never adjust the definition of a class later, so the class design pattern doesn't suggest such capabilities. But **one of the most powerful parts** of JS is that it *is* dynamic, and the definition of any object is (unless you make it immutable) a fluid and mutable *thing*.

`class` seems to imply you shouldn't do such things, by forcing you into the uglier `.prototype` syntax to do so, or forcing you to think about `super` gotchas, etc. It also offers *very little* support for any of the pitfalls that this dynamism can bring.

In other words, it's as if `class` is telling you: "dynamic is too hard, so it's probably not a good idea. Here's a static-looking syntax, so code your stuff statically."

What a sad commentary on JavaScript: **dynamic is too hard, let's pretend to be (but not actually be!) static.**

These are the reasons why ES6 `class` is masquerading as a nice solution to syntactic headaches, but it's actually muddying the waters further and making things worse for JS and for clear and concise understanding.

Note: If you use the `.bind(...)` utility to make a hard-bound function (see Chapter 2), the function created is not subclassable with ES6 `extend` like normal functions are.

Review (TL;DR)

`class` does a very good job of pretending to fix the problems with the class/inheritance design pattern in JS. But it actually does the opposite: **it hides many of the problems, and introduces other subtle but dangerous ones.**

`class` contributes to the ongoing confusion of "class" in JavaScript which has plagued the language for nearly two decades. In some respects, it asks more questions than it answers, and it feels in totality like a very unnatural fit on top of the elegant simplicity of the `[[Prototype]]` mechanism.

Bottom line: if ES6 `class` makes it harder to robustly leverage `[[Prototype]]`, and hides the most important nature of the JS object mechanism -- **the live delegation links between objects** -- shouldn't we see `class` as creating more troubles than it solves, and just relegate it to an anti-pattern?

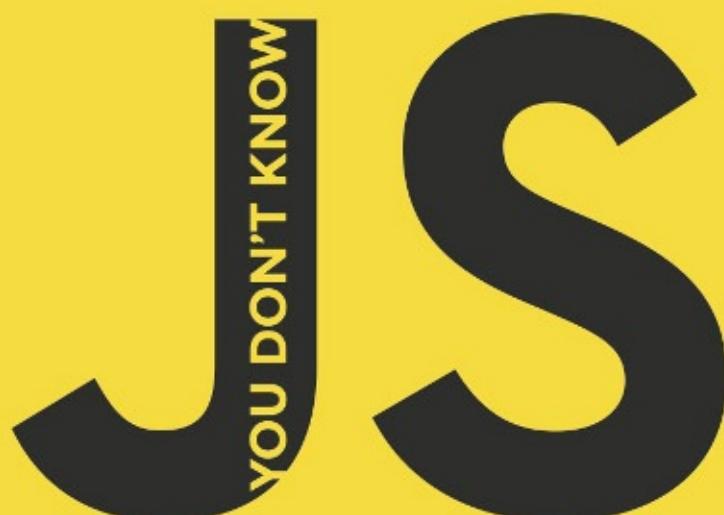
I can't really answer that question for you. But I hope this book has fully explored the issue at a deeper level than you've ever gone before, and has given you the information you need *to answer it yourself.*

O'REILLY®

"Kyle wrangles the asynchronous nature of JavaScript and shows you how to straighten it out with promises and generators."
—MARC GRABANSKI, CEO & UI Developer, Frontend Masters

KYLE SIMPSON

ASYNC & PERFORMANCE



One of the most important and yet often misunderstood parts of programming in a language like JavaScript is how to express and manipulate program behavior spread out over a period of time.

This is not just about what happens from the beginning of a `for` loop to the end of a `for` loop, which of course takes *some time* (microseconds to milliseconds) to complete. It's about what happens when part of your program runs *now*, and another part of your program runs *later* -- there's a gap between *now* and *later* where your program isn't actively executing.

Practically all nontrivial programs ever written (especially in JS) have in some way or another had to manage this gap, whether that be in waiting for user input, requesting data from a database or file system, sending data across the network and waiting for a response, or performing a repeated task at a fixed interval of time (like animation). In all these various ways, your program has to manage state across the gap in time. As they famously say in London (of the chasm between the subway door and the platform): "mind the gap."

In fact, the relationship between the *now* and *later* parts of your program is at the heart of asynchronous programming.

Asynchronous programming has been around since the beginning of JS, for sure. But most JS developers have never really carefully considered exactly how and why it crops up in their programs, or explored various *other* ways to handle it. The *good enough* approach has always been the humble callback function. Many to this day will insist that callbacks are more than sufficient.

But as JS continues to grow in both scope and complexity, to meet the ever-widening demands of a first-class programming language that runs in browsers and servers and every conceivable device in between, the pains by which we manage asynchrony are becoming increasingly crippling, and they cry out for approaches that are both more capable and more reasonable.

While this all may seem rather abstract right now, I assure you we'll tackle it more completely and concretely as we go on through this book. We'll explore a variety of emerging techniques for async JavaScript programming over the next several chapters.

But before we can get there, we're going to have to understand much more deeply what asynchrony is and how it operates in JS.

A Program in Chunks

You may write your JS program in one `.js` file, but your program is almost certainly comprised of several chunks, only one of which is going to execute *now*, and the rest of which will execute *later*. The most common unit of *chunk* is the `function`.

The problem most developers new to JS seem to have is that *later* doesn't happen strictly and immediately after *now*. In other words, tasks that cannot complete *now* are, by definition, going to complete asynchronously, and thus we will not have blocking behavior as you might intuitively expect or want.

Consider:

```
// ajax(..) is some arbitrary Ajax function given by a library
var data = ajax( "http://some.url.1" );

console.log( data );
// Oops! `data` generally won't have the Ajax results
```

You're probably aware that standard Ajax requests don't complete synchronously, which means the `ajax(..)` function does not yet have any value to return back to be assigned to `data` variable. If `ajax(..)` could block until the response came back, then the `data = ..` assignment would work fine.

But that's not how we do Ajax. We make an asynchronous Ajax request *now*, and we won't get the results back until *later*.

The simplest (but definitely not only, or necessarily even best!) way of "waiting" from *now* until *later* is to use a function, commonly called a **callback function**:

```
// ajax(..) is some arbitrary Ajax function given by a library
ajax( "http://some.url.1", function myCallbackFunction(data){

    console.log( data ); // Yay, I gots me some `data`!

});
```

Warning: You may have heard that it's possible to make synchronous Ajax requests. While that's technically true, you should never, ever do it, under any circumstances, because it locks the browser UI (buttons, menus, scrolling, etc.) and prevents any user interaction whatsoever. This is a terrible idea, and should always be avoided.

Before you protest in disagreement, no, your desire to avoid the mess of callbacks is *not* justification for blocking, synchronous Ajax.

For example, consider this code:

```

function now() {
    return 21;
}

function later() {
    answer = answer * 2;
    console.log( "Meaning of life:", answer );
}

var answer = now();

setTimeout( later, 1000 ); // Meaning of life: 42

```

There are two chunks to this program: the stuff that will run *now*, and the stuff that will run *later*. It should be fairly obvious what those two chunks are, but let's be super explicit:

Now:

```

function now() {
    return 21;
}

function later() { .. }

var answer = now();

setTimeout( later, 1000 );

```

Later:

```

answer = answer * 2;
console.log( "Meaning of life:", answer );

```

The *now* chunk runs right away, as soon as you execute your program. But `setTimeout(...)` also sets up an event (a timeout) to happen *later*, so the contents of the `later()` function will be executed at a later time (1,000 milliseconds from now).

Any time you wrap a portion of code into a `function` and specify that it should be executed in response to some event (timer, mouse click, Ajax response, etc.), you are creating a *later* chunk of your code, and thus introducing asynchrony to your program.

Async Console

There is no specification or set of requirements around how the `console.*` methods work -- they are not officially part of JavaScript, but are instead added to JS by the *hosting environment* (see the *Types & Grammar* title of this book series).

So, different browsers and JS environments do as they please, which can sometimes lead to confusing behavior.

In particular, there are some browsers and some conditions that `console.log(..)` does not actually immediately output what it's given. The main reason this may happen is because I/O is a very slow and blocking part of many programs (not just JS). So, it may perform better (from the page/UI perspective) for a browser to handle `console` I/O asynchronously in the background, without you perhaps even knowing that occurred.

A not terribly common, but possible, scenario where this could be *observable* (not from code itself but from the outside):

```
var a = {
  index: 1
};

// later
console.log( a ); // ??

// even later
a.index++;
```

We'd normally expect to see the `a` object be snapshotted at the exact moment of the `console.log(..)` statement, printing something like `{ index: 1 }`, such that in the next statement when `a.index++` happens, it's modifying something different than, or just strictly after, the output of `a`.

Most of the time, the preceding code will probably produce an object representation in your developer tools' console that's what you'd expect. But it's possible this same code could run in a situation where the browser felt it needed to defer the `console` I/O to the background, in which case it's *possible* that by the time the object is represented in the browser console, the `a.index++` has already happened, and it shows `{ index: 2 }`.

It's a moving target under what conditions exactly `console` I/O will be deferred, or even whether it will be observable. Just be aware of this possible asynchronicity in I/O in case you ever run into issues in debugging where objects have been modified *after* a `console.log(..)` statement and yet you see the unexpected modifications show up.

Note: If you run into this rare scenario, the best option is to use breakpoints in your JS debugger instead of relying on `console` output. The next best option would be to force a "snapshot" of the object in question by serializing it to a `string`, like with

```
JSON.stringify(...).
```

Event Loop

Let's make a (perhaps shocking) claim: despite clearly allowing asynchronous JS code (like the timeout we just looked at), up until recently (ES6), JavaScript itself has actually never had any direct notion of asynchrony built into it.

What!? That seems like a crazy claim, right? In fact, it's quite true. The JS engine itself has never done anything more than execute a single chunk of your program at any given moment, when asked to.

"Asked to." By whom? That's the important part!

The JS engine doesn't run in isolation. It runs inside a *hosting environment*, which is for most developers the typical web browser. Over the last several years (but by no means exclusively), JS has expanded beyond the browser into other environments, such as servers, via things like Node.js. In fact, JavaScript gets embedded into all kinds of devices these days, from robots to lightbulbs.

But the one common "thread" (that's a not-so-subtle asynchronous joke, for what it's worth) of all these environments is that they have a mechanism in them that handles executing multiple chunks of your program *over time*, at each moment invoking the JS engine, called the "event loop."

In other words, the JS engine has had no innate sense of *time*, but has instead been an on-demand execution environment for any arbitrary snippet of JS. It's the surrounding environment that has always *scheduled* "events" (JS code executions).

So, for example, when your JS program makes an Ajax request to fetch some data from a server, you set up the "response" code in a function (commonly called a "callback"), and the JS engine tells the hosting environment, "Hey, I'm going to suspend execution for now, but whenever you finish with that network request, and you have some data, please *call* this function *back*."

The browser is then set up to listen for the response from the network, and when it has something to give you, it schedules the callback function to be executed by inserting it into the *event loop*.

So what is the *event loop*?

Let's conceptualize it first through some fake-ish code:

```
// `eventLoop` is an array that acts as a queue (first-in, first-out)
var eventLoop = [ ];
var event;

// keep going "forever"
while (true) {
    // perform a "tick"
    if (eventLoop.length > 0) {
        // get the next event in the queue
        event = eventLoop.shift();

        // now, execute the next event
        try {
            event();
        }
        catch (err) {
            reportError(err);
        }
    }
}
```

This is, of course, vastly simplified pseudocode to illustrate the concepts. But it should be enough to help get a better understanding.

As you can see, there's a continuously running loop represented by the `while` loop, and each iteration of this loop is called a "tick." For each tick, if an event is waiting on the queue, it's taken off and executed. These events are your function callbacks.

It's important to note that `setTimeout(...)` doesn't put your callback on the event loop queue. What it does is set up a timer; when the timer expires, the environment places your callback into the event loop, such that some future tick will pick it up and execute it.

What if there are already 20 items in the event loop at that moment? Your callback waits. It gets in line behind the others -- there's not normally a path for preempting the queue and skipping ahead in line. This explains why `setTimeout(...)` timers may not fire with perfect temporal accuracy. You're guaranteed (roughly speaking) that your callback won't fire *before* the time interval you specify, but it can happen at or after that time, depending on the state of the event queue.

So, in other words, your program is generally broken up into lots of small chunks, which happen one after the other in the event loop queue. And technically, other events not related directly to your program can be interleaved within the queue as well.

Note: We mentioned "up until recently" in relation to ES6 changing the nature of where the event loop queue is managed. It's mostly a formal technicality, but ES6 now specifies how the event loop works, which means technically it's within the purview of the JS engine, rather

than just the *hosting environment*. One main reason for this change is the introduction of ES6 Promises, which we'll discuss in Chapter 3, because they require the ability to have direct, fine-grained control over scheduling operations on the event loop queue (see the discussion of `setTimeout(...)` in the "Cooperation" section).

Parallel Threading

It's very common to conflate the terms "async" and "parallel," but they are actually quite different. Remember, async is about the gap between *now* and *later*. But parallel is about things being able to occur simultaneously.

The most common tools for parallel computing are processes and threads. Processes and threads execute independently and may execute simultaneously: on separate processors, or even separate computers, but multiple threads can share the memory of a single process.

An event loop, by contrast, breaks its work into tasks and executes them in serial, disallowing parallel access and changes to shared memory. Parallelism and "serialism" can coexist in the form of cooperating event loops in separate threads.

The interleaving of parallel threads of execution and the interleaving of asynchronous events occur at very different levels of granularity.

For example:

```
function later() {
    answer = answer * 2;
    console.log("Meaning of life:", answer);
}
```

While the entire contents of `later()` would be regarded as a single event loop queue entry, when thinking about a thread this code would run on, there's actually perhaps a dozen different low-level operations. For example, `answer = answer * 2` requires first loading the current value of `answer`, then putting `2` somewhere, then performing the multiplication, then taking the result and storing it back into `answer`.

In a single-threaded environment, it really doesn't matter that the items in the thread queue are low-level operations, because nothing can interrupt the thread. But if you have a parallel system, where two different threads are operating in the same program, you could very likely have unpredictable behavior.

Consider:

```

var a = 20;

function foo() {
    a = a + 1;
}

function bar() {
    a = a * 2;
}

// ajax(..) is some arbitrary Ajax function given by a library
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );

```

In JavaScript's single-threaded behavior, if `foo()` runs before `bar()`, the result is that `a` has `42`, but if `bar()` runs before `foo()` the result in `a` will be `41`.

If JS events sharing the same data executed in parallel, though, the problems would be much more subtle. Consider these two lists of pseudocode tasks as the threads that could respectively run the code in `foo()` and `bar()`, and consider what happens if they are running at exactly the same time:

Thread 1 (`x` and `y` are temporary memory locations):

```

foo():
    a. load value of `a` in `X`
    b. store `1` in `Y`
    c. add `X` and `Y`, store result in `X`
    d. store value of `X` in `a`

```

Thread 2 (`x` and `y` are temporary memory locations):

```

bar():
    a. load value of `a` in `X`
    b. store `2` in `Y`
    c. multiply `X` and `Y`, store result in `X`
    d. store value of `X` in `a`

```

Now, let's say that the two threads are running truly in parallel. You can probably spot the problem, right? They use shared memory locations `x` and `y` for their temporary steps.

What's the end result in `a` if the steps happen like this?

```

1a (load value of `a` in `X`    ==> `20`)
2a (load value of `a` in `X`    ==> `20`)
1b (store `1` in `Y`    ==> `1`)
2b (store `2` in `Y`    ==> `2`)
1c (add `X` and `Y`, store result in `X`    ==> `22`)
1d (store value of `X` in `a`    ==> `22`)
2c (multiply `X` and `Y`, store result in `X`    ==> `44`)
2d (store value of `X` in `a`    ==> `44`)

```

The result in `a` will be `44`. But what about this ordering?

```

1a (load value of `a` in `X`    ==> `20`)
2a (load value of `a` in `X`    ==> `20`)
2b (store `2` in `Y`    ==> `2`)
1b (store `1` in `Y`    ==> `1`)
2c (multiply `X` and `Y`, store result in `X`    ==> `20`)
1c (add `X` and `Y`, store result in `X`    ==> `21`)
1d (store value of `X` in `a`    ==> `21`)
2d (store value of `X` in `a`    ==> `21`)

```

The result in `a` will be `21`.

So, threaded programming is very tricky, because if you don't take special steps to prevent this kind of interruption/interleaving from happening, you can get very surprising, nondeterministic behavior that frequently leads to headaches.

JavaScript never shares data across threads, which means *that* level of nondeterminism isn't a concern. But that doesn't mean JS is always deterministic. Remember earlier, where the relative ordering of `foo()` and `bar()` produces two different results (`41` or `42`)?

Note: It may not be obvious yet, but not all nondeterminism is bad. Sometimes it's irrelevant, and sometimes it's intentional. We'll see more examples of that throughout this and the next few chapters.

Run-to-Completion

Because of JavaScript's single-threading, the code inside of `foo()` (and `bar()`) is atomic, which means that once `foo()` starts running, the entirety of its code will finish before any of the code in `bar()` can run, or vice versa. This is called "run-to-completion" behavior.

In fact, the run-to-completion semantics are more obvious when `foo()` and `bar()` have more code in them, such as:

```

var a = 1;
var b = 2;

function foo() {
    a++;
    b = b * a;
    a = b + 3;
}

function bar() {
    b--;
    a = 8 + b;
    b = a * 2;
}

// ajax(..) is some arbitrary Ajax function given by a library
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );

```

Because `foo()` can't be interrupted by `bar()`, and `bar()` can't be interrupted by `foo()`, this program only has two possible outcomes depending on which starts running first -- if threading were present, and the individual statements in `foo()` and `bar()` could be interleaved, the number of possible outcomes would be greatly increased!

Chunk 1 is synchronous (happens *now*), but chunks 2 and 3 are asynchronous (happen *later*), which means their execution will be separated by a gap of time.

Chunk 1:

```

var a = 1;
var b = 2;

```

Chunk 2 (`foo()`):

```

a++;
b = b * a;
a = b + 3;

```

Chunk 3 (`bar()`):

```

b--;
a = 8 + b;
b = a * 2;

```

Chunks 2 and 3 may happen in either-first order, so there are two possible outcomes for this program, as illustrated here:

Outcome 1:

```
var a = 1;
var b = 2;

// foo()
a++;
b = b * a;
a = b + 3;

// bar()
b--;
a = 8 + b;
b = a * 2;

a; // 11
b; // 22
```

Outcome 2:

```
var a = 1;
var b = 2;

// bar()
b--;
a = 8 + b;
b = a * 2;

// foo()
a++;
b = b * a;
a = b + 3;

a; // 183
b; // 180
```

Two outcomes from the same code means we still have nondeterminism! But it's at the function (event) ordering level, rather than at the statement ordering level (or, in fact, the expression operation ordering level) as it is with threads. In other words, it's *more deterministic* than threads would have been.

As applied to JavaScript's behavior, this function-ordering nondeterminism is the common term "race condition," as `foo()` and `bar()` are racing against each other to see which runs first. Specifically, it's a "race condition" because you cannot predict reliably how `a` and `b` will turn out.

Note: If there was a function in JS that somehow did not have run-to-completion behavior, we could have many more possible outcomes, right? It turns out ES6 introduces just such a thing (see Chapter 4 "Generators"), but don't worry right now, we'll come back to that!

Concurrency

Let's imagine a site that displays a list of status updates (like a social network news feed) that progressively loads as the user scrolls down the list. To make such a feature work correctly, (at least) two separate "processes" will need to be executing *simultaneously* (i.e., during the same window of time, but not necessarily at the same instant).

Note: We're using "process" in quotes here because they aren't true operating system-level processes in the computer science sense. They're virtual processes, or tasks, that represent a logically connected, sequential series of operations. We'll simply prefer "process" over "task" because terminology-wise, it will match the definitions of the concepts we're exploring.

The first "process" will respond to `onscroll` events (making Ajax requests for new content) as they fire when the user has scrolled the page further down. The second "process" will receive Ajax responses back (to render content onto the page).

Obviously, if a user scrolls fast enough, you may see two or more `onscroll` events fired during the time it takes to get the first response back and process, and thus you're going to have `onscroll` events and Ajax response events firing rapidly, interleaved with each other.

Concurrency is when two or more "processes" are executing simultaneously over the same period, regardless of whether their individual constituent operations happen *in parallel* (at the same instant on separate processors or cores) or not. You can think of concurrency then as "process"-level (or task-level) parallelism, as opposed to operation-level parallelism (separate-processor threads).

Note: Concurrency also introduces an optional notion of these "processes" interacting with each other. We'll come back to that later.

For a given window of time (a few seconds worth of a user scrolling), let's visualize each independent "process" as a series of events/operations:

"Process" 1 (`onscroll` events):

```
onscroll, request 1
onscroll, request 2
onscroll, request 3
onscroll, request 4
onscroll, request 5
onscroll, request 6
onscroll, request 7
```

"Process" 2 (Ajax response events):

```
response 1
response 2
response 3
response 4
response 5
response 6
response 7
```

It's quite possible that an `onscroll` event and an Ajax response event could be ready to be processed at exactly the same *moment*. For example let's visualize these events in a timeline:

```
onscroll, request 1
onscroll, request 2      response 1
onscroll, request 3      response 2
response 3
onscroll, request 4
onscroll, request 5
onscroll, request 6      response 4
onscroll, request 7
response 6
response 5
response 7
```

But, going back to our notion of the event loop from earlier in the chapter, JS is only going to be able to handle one event at a time, so either `onscroll, request 2` is going to happen first or `response 1` is going to happen first, but they cannot happen at literally the same moment. Just like kids at a school cafeteria, no matter what crowd they form outside the doors, they'll have to merge into a single line to get their lunch!

Let's visualize the interleaving of all these events onto the event loop queue.

Event Loop Queue:

```

onscroll, request 1    <--- Process 1 starts
onscroll, request 2
response 1             <--- Process 2 starts
onscroll, request 3
response 2
response 3
onscroll, request 4
onscroll, request 5
onscroll, request 6
response 4
onscroll, request 7    <--- Process 1 finishes
response 6
response 5
response 7             <--- Process 2 finishes

```

"Process 1" and "Process 2" run concurrently (task-level parallel), but their individual events run sequentially on the event loop queue.

By the way, notice how `response 6` and `response 5` came back out of expected order?

The single-threaded event loop is one expression of concurrency (there are certainly others, which we'll come back to later).

Noninteracting

As two or more "processes" are interleaving their steps/events concurrently within the same program, they don't necessarily need to interact with each other if the tasks are unrelated. **If they don't interact, nondeterminism is perfectly acceptable.**

For example:

```

var res = {};

function foo(results) {
  res.foo = results;
}

function bar(results) {
  res.bar = results;
}

// ajax(..) is some arbitrary Ajax function given by a library
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );

```

`foo()` and `bar()` are two concurrent "processes," and it's nondeterminate which order they will be fired in. But we've constructed the program so it doesn't matter what order they fire in, because they act independently and as such don't need to interact.

This is not a "race condition" bug, as the code will always work correctly, regardless of the ordering.

Interaction

More commonly, concurrent "processes" will by necessity interact, indirectly through scope and/or the DOM. When such interaction will occur, you need to coordinate these interactions to prevent "race conditions," as described earlier.

Here's a simple example of two concurrent "processes" that interact because of implied ordering, which is only *sometimes broken*:

```
var res = [];

function response(data) {
    res.push( data );
}

// ajax(..) is some arbitrary Ajax function given by a library
ajax( "http://some.url.1", response );
ajax( "http://some.url.2", response );
```

The concurrent "processes" are the two `response()` calls that will be made to handle the Ajax responses. They can happen in either-first order.

Let's assume the expected behavior is that `res[0]` has the results of the `"http://some.url.1"` call, and `res[1]` has the results of the `"http://some.url.2"` call. Sometimes that will be the case, but sometimes they'll be flipped, depending on which call finishes first. There's a pretty good likelihood that this nondeterminism is a "race condition" bug.

Note: Be extremely wary of assumptions you might tend to make in these situations. For example, it's not uncommon for a developer to observe that `"http://some.url.2"` is "always" much slower to respond than `"http://some.url.1"`, perhaps by virtue of what tasks they're doing (e.g., one performing a database task and the other just fetching a static file), so the observed ordering seems to always be as expected. Even if both requests go to the same server, and *it* intentionally responds in a certain order, there's no *real* guarantee of what order the responses will arrive back in the browser.

So, to address such a race condition, you can coordinate ordering interaction:

```
var res = [];

function response(data) {
    if (data.url == "http://some.url.1") {
        res[0] = data;
    }
    else if (data.url == "http://some.url.2") {
        res[1] = data;
    }
}

// ajax(..) is some arbitrary Ajax function given by a library
ajax( "http://some.url.1", response );
ajax( "http://some.url.2", response );
```

Regardless of which Ajax response comes back first, we inspect the `data.url` (assuming one is returned from the server, of course!) to figure out which position the response data should occupy in the `res` array. `res[0]` will always hold the `"http://some.url.1"` results and `res[1]` will always hold the `"http://some.url.2"` results. Through simple coordination, we eliminated the "race condition" nondeterminism.

The same reasoning from this scenario would apply if multiple concurrent function calls were interacting with each other through the shared DOM, like one updating the contents of a `<div>` and the other updating the style or attributes of the `<div>` (e.g., to make the DOM element visible once it has content). You probably wouldn't want to show the DOM element before it had content, so the coordination must ensure proper ordering interaction.

Some concurrency scenarios are *always broken* (not just *sometimes*) without coordinated interaction. Consider:

```

var a, b;

function foo(x) {
    a = x * 2;
    baz();
}

function bar(y) {
    b = y * 2;
    baz();
}

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

// ajax(..) is some arbitrary Ajax function given by a library
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );

```

In this example, whether `foo()` or `bar()` fires first, it will always cause `baz()` to run too early (either `a` or `b` will still be `undefined`), but the second invocation of `baz()` will work, as both `a` and `b` will be available.

There are different ways to address such a condition. Here's one simple way:

```

var a, b;

function foo(x) {
    a = x * 2;
    if (a && b) {
        baz();
    }
}

function bar(y) {
    b = y * 2;
    if (a && b) {
        baz();
    }
}

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

// ajax(..) is some arbitrary Ajax function given by a library
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );

```

The `if (a && b)` conditional around the `baz()` call is traditionally called a "gate," because we're not sure what order `a` and `b` will arrive, but we wait for both of them to get there before we proceed to open the gate (call `baz()`).

Another concurrency interaction condition you may run into is sometimes called a "race," but more correctly called a "latch." It's characterized by "only the first one wins" behavior. Here, nondeterminism is acceptable, in that you are explicitly saying it's OK for the "race" to the finish line to have only one winner.

Consider this broken code:

```
var a;

function foo(x) {
  a = x * 2;
  baz();
}

function bar(x) {
  a = x / 2;
  baz();
}

function baz() {
  console.log(a);
}

// ajax(..) is some arbitrary Ajax function given by a library
ajax("http://some.url.1", foo);
ajax("http://some.url.2", bar);
```

Whichever one (`foo()` or `bar()`) fires last will not only overwrite the assigned `a` value from the other, but it will also duplicate the call to `baz()` (likely undesired).

So, we can coordinate the interaction with a simple latch, to let only the first one through:

```

var a;

function foo(x) {
  if (a == undefined) {
    a = x * 2;
    baz();
  }
}

function bar(x) {
  if (a == undefined) {
    a = x / 2;
    baz();
  }
}

function baz() {
  console.log(a);
}

// ajax(...) is some arbitrary Ajax function given by a library
ajax("http://some.url.1", foo);
ajax("http://some.url.2", bar);

```

The `if (a == undefined)` conditional allows only the first of `foo()` or `bar()` through, and the second (and indeed any subsequent) calls would just be ignored. There's just no virtue in coming in second place!

Note: In all these scenarios, we've been using global variables for simplistic illustration purposes, but there's nothing about our reasoning here that requires it. As long as the functions in question can access the variables (via scope), they'll work as intended. Relying on lexically scoped variables (see the *Scope & Closures* title of this book series), and in fact global variables as in these examples, is one obvious downside to these forms of concurrency coordination. As we go through the next few chapters, we'll see other ways of coordination that are much cleaner in that respect.

Cooperation

Another expression of concurrency coordination is called "cooperative concurrency." Here, the focus isn't so much on interacting via value sharing in scopes (though that's obviously still allowed!). The goal is to take a long-running "process" and break it up into steps or batches so that other concurrent "processes" have a chance to interleave their operations into the event loop queue.

For example, consider an Ajax response handler that needs to run through a long list of results to transform the values. We'll use `Array#map(...)` to keep the code shorter:

```
var res = [];

// `response(...)` receives array of results from the Ajax call
function response(data) {
    // add onto existing `res` array
    res = res.concat(
        // make a new transformed array with all `data` values doubled
        data.map( function(val){
            return val * 2;
        })
    );
}

// ajax(...) is some arbitrary Ajax function given by a library
ajax( "http://some.url.1", response );
ajax( "http://some.url.2", response );
```

If `"http://some.url.1"` gets its results back first, the entire list will be mapped into `res` all at once. If it's a few thousand or less records, this is not generally a big deal. But if it's say 10 million records, that can take a while to run (several seconds on a powerful laptop, much longer on a mobile device, etc.).

While such a "process" is running, nothing else in the page can happen, including no other `response(...)` calls, no UI updates, not even user events like scrolling, typing, button clicking, and the like. That's pretty painful.

So, to make a more cooperatively concurrent system, one that's friendlier and doesn't hog the event loop queue, you can process these results in asynchronous batches, after each one "yielding" back to the event loop to let other waiting events happen.

Here's a very simple approach:

```

var res = [];

// `response(...)` receives array of results from the Ajax call
function response(data) {
    // let's just do 1000 at a time
    var chunk = data.splice( 0, 1000 );

    // add onto existing `res` array
    res = res.concat(
        // make a new transformed array with all `chunk` values doubled
        chunk.map( function(val){
            return val * 2;
        })
    );

    // anything left to process?
    if (data.length > 0) {
        // async schedule next batch
        setTimeout( function(){
            response( data );
        }, 0 );
    }
}

// ajax(...) is some arbitrary Ajax function given by a library
ajax( "http://some.url.1", response );
ajax( "http://some.url.2", response );

```

We process the data set in maximum-sized chunks of 1,000 items. By doing so, we ensure a short-running "process," even if that means many more subsequent "processes," as the interleaving onto the event loop queue will give us a much more responsive (performant) site/app.

Of course, we're not interaction-coordinating the ordering of any of these "processes," so the order of results in `res` won't be predictable. If ordering was required, you'd need to use interaction techniques like those we discussed earlier, or ones we will cover in later chapters of this book.

We use the `setTimeout(..0)` (hack) for async scheduling, which basically just means "stick this function at the end of the current event loop queue."

Note: `setTimeout(..0)` is not technically inserting an item directly onto the event loop queue. The timer will insert the event at its next opportunity. For example, two subsequent `setTimeout(..0)` calls would not be strictly guaranteed to be processed in call order, so it is possible to see various conditions like timer drift where the ordering of such events isn't predictable. In Node.js, a similar approach is `process.nextTick(..)`. Despite how

convenient (and usually more performant) it would be, there's not a single direct way (at least yet) across all environments to ensure async event ordering. We cover this topic in more detail in the next section.

Jobs

As of ES6, there's a new concept layered on top of the event loop queue, called the "Job queue." The most likely exposure you'll have to it is with the asynchronous behavior of Promises (see Chapter 3).

Unfortunately, at the moment it's a mechanism without an exposed API, and thus demonstrating it is a bit more convoluted. So we're going to have to just describe it conceptually, such that when we discuss async behavior with Promises in Chapter 3, you'll understand how those actions are being scheduled and processed.

So, the best way to think about this that I've found is that the "Job queue" is a queue hanging off the end of every tick in the event loop queue. Certain async-implied actions that may occur during a tick of the event loop will not cause a whole new event to be added to the event loop queue, but will instead add an item (aka Job) to the end of the current tick's Job queue.

It's kinda like saying, "oh, here's this other thing I need to do *later*, but make sure it happens right away before anything else can happen."

Or, to use a metaphor: the event loop queue is like an amusement park ride, where once you finish the ride, you have to go to the back of the line to ride again. But the Job queue is like finishing the ride, but then cutting in line and getting right back on.

A Job can also cause more Jobs to be added to the end of the same queue. So, it's theoretically possible that a Job "loop" (a Job that keeps adding another Job, etc.) could spin indefinitely, thus starving the program of the ability to move on to the next event loop tick. This would conceptually be almost the same as just expressing a long-running or infinite loop (like `while (true) ...`) in your code.

Jobs are kind of like the spirit of the `setTimeout(..0)` hack, but implemented in such a way as to have a much more well-defined and guaranteed ordering: **later, but as soon as possible**.

Let's imagine an API for scheduling Jobs (directly, without hacks), and call it `schedule(..)`. Consider:

```

console.log( "A" );

setTimeout( function(){
    console.log( "B" );
}, 0 );

// theoretical "Job API"
schedule( function(){
    console.log( "C" );

    schedule( function(){
        console.log( "D" );
    } );
} );

```

You might expect this to print out `A B C D`, but instead it would print out `A C D B`, because the Jobs happen at the end of the current event loop tick, and the timer fires to schedule for the *next* event loop tick (if available!).

In Chapter 3, we'll see that the asynchronous behavior of Promises is based on Jobs, so it's important to keep clear how that relates to event loop behavior.

Statement Ordering

The order in which we express statements in our code is not necessarily the same order as the JS engine will execute them. That may seem like quite a strange assertion to make, so we'll just briefly explore it.

But before we do, we should be crystal clear on something: the rules/grammar of the language (see the *Types & Grammar* title of this book series) dictate a very predictable and reliable behavior for statement ordering from the program point of view. So what we're about to discuss are **not things you should ever be able to observe** in your JS program.

Warning: If you are ever able to *observe* compiler statement reordering like we're about to illustrate, that'd be a clear violation of the specification, and it would unquestionably be due to a bug in the JS engine in question -- one which should promptly be reported and fixed! But it's vastly more common that you *suspect* something crazy is happening in the JS engine, when in fact it's just a bug (probably a "race condition"!) in your own code -- so look there first, and again and again. The JS debugger, using breakpoints and stepping through code line by line, will be your most powerful tool for sniffing out such bugs in *your code*.

Consider:

```
var a, b;

a = 10;
b = 30;

a = a + 1;
b = b + 1;

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

This code has no expressed asynchrony to it (other than the rare `console async` I/O discussed earlier!), so the most likely assumption is that it would process line by line in top-down fashion.

But it's *possible* that the JS engine, after compiling this code (yes, JS is compiled -- see the *Scope & Closures* title of this book series!) might find opportunities to run your code faster by rearranging (safely) the order of these statements. Essentially, as long as you can't observe the reordering, anything's fair game.

For example, the engine might find it's faster to actually execute the code like this:

```
var a, b;

a = 10;
a++;

b = 30;
b++;

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

Or this:

```
var a, b;

a = 11;
b = 31;

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

Or even:

```
// because `a` and `b` aren't used anymore, we can
// inline and don't even need them!
console.log( 42 ); // 42
```

In all these cases, the JS engine is performing safe optimizations during its compilation, as the end *observable* result will be the same.

But here's a scenario where these specific optimizations would be unsafe and thus couldn't be allowed (of course, not to say that it's not optimized at all):

```
var a, b;

a = 10;
b = 30;

// we need `a` and `b` in their preincremented state!
console.log( a * b ); // 300

a = a + 1;
b = b + 1;

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

Other examples where the compiler reordering could create observable side effects (and thus must be disallowed) would include things like any function call with side effects (even and especially getter functions), or ES6 Proxy objects (see the *ES6 & Beyond* title of this book series).

Consider:

```
function foo() {
    console.log( b );
    return 1;
}

var a, b, c;

// ES5.1 getter literal syntax
c = {
    get bar() {
        console.log( a );
        return 1;
    }
};

a = 10;
b = 30;

a += foo();           // 30
b += c.bar;          // 11

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

If it weren't for the `console.log(...)` statements in this snippet (just used as a convenient form of observable side effect for the illustration), the JS engine would likely have been free, if it wanted to (who knows if it would!?), to reorder the code to:

```
// ...  
  
a = 10 + foo();  
b = 30 + c.bar;  
  
// ...
```

While JS semantics thankfully protect us from the *observable* nightmares that compiler statement reordering would seem to be in danger of, it's still important to understand just how tenuous a link there is between the way source code is authored (in top-down fashion) and the way it runs after compilation.

Compiler statement reordering is almost a micro-metaphor for concurrency and interaction. As a general concept, such awareness can help you understand async JS code flow issues better.

Review

A JavaScript program is (practically) always broken up into two or more chunks, where the first chunk runs *now* and the next chunk runs *later*, in response to an event. Even though the program is executed chunk-by-chunk, all of them share the same access to the program scope and state, so each modification to state is made on top of the previous state.

Whenever there are events to run, the *event loop* runs until the queue is empty. Each iteration of the event loop is a "tick." User interaction, IO, and timers enqueue events on the event queue.

At any given moment, only one event can be processed from the queue at a time. While an event is executing, it can directly or indirectly cause one or more subsequent events.

Concurrency is when two or more chains of events interleave over time, such that from a high-level perspective, they appear to be running *simultaneously* (even though at any given moment only one event is being processed).

It's often necessary to do some form of interaction coordination between these concurrent "processes" (as distinct from operating system processes), for instance to ensure ordering or to prevent "race conditions." These "processes" can also *cooperate* by breaking themselves into smaller chunks and to allow other "process" interleaving.

In Chapter 1, we explored the terminology and concepts around asynchronous programming in JavaScript. Our focus is on understanding the single-threaded (one-at-a-time) event loop queue that drives all "events" (async function invocations). We also explored various ways that concurrency patterns explain the relationships (if any!) between *simultaneously* running chains of events, or "processes" (tasks, function calls, etc.).

All our examples in Chapter 1 used the function as the individual, indivisible unit of operations, whereby inside the function, statements run in predictable order (above the compiler level!), but at the function-ordering level, events (aka async function invocations) can happen in a variety of orders.

In all these cases, the function is acting as a "callback," because it serves as the target for the event loop to "call back into" the program, whenever that item in the queue is processed.

As you no doubt have observed, callbacks are by far the most common way that asynchrony in JS programs is expressed and managed. Indeed, the callback is the most fundamental async pattern in the language.

Countless JS programs, even very sophisticated and complex ones, have been written upon no other async foundation than the callback (with of course the concurrency interaction patterns we explored in Chapter 1). The callback function is the async work horse for JavaScript, and it does its job respectably.

Except... callbacks are not without their shortcomings. Many developers are excited by the *promise* (pun intended!) of better async patterns. But it's impossible to effectively use any abstraction if you don't understand what it's abstracting, and why.

In this chapter, we will explore a couple of those in depth, as motivation for why more sophisticated async patterns (explored in subsequent chapters of this book) are necessary and desired.

Continuations

Let's go back to the async callback example we started with in Chapter 1, but let me slightly modify it to illustrate a point:

```
// A
ajax( "...", function(...){
    // C
} );
// B
```

// A and // B represent the first half of the program (aka the *now*), and // c marks the second half of the program (aka the *later*). The first half executes right away, and then there's a "pause" of indeterminate length. At some future moment, if the Ajax call completes, then the program will pick up where it left off, and *continue* with the second half.

In other words, the callback function wraps or encapsulates the *continuation* of the program.

Let's make the code even simpler:

```
// A
setTimeout( function(){
  // C
}, 1000 );
// B
```

Stop for a moment and ask yourself how you'd describe (to someone else less informed about how JS works) the way that program behaves. Go ahead, try it out loud. It's a good exercise that will help my next points make more sense.

Most readers just now probably thought or said something to the effect of: "Do A, then set up a timeout to wait 1,000 milliseconds, then once that fires, do C." How close was your rendition?

You might have caught yourself and self-edited to: "Do A, setup the timeout for 1,000 milliseconds, then do B, then after the timeout fires, do C." That's more accurate than the first version. Can you spot the difference?

Even though the second version is more accurate, both versions are deficient in explaining this code in a way that matches our brains to the code, and the code to the JS engine. The disconnect is both subtle and monumental, and is at the very heart of understanding the shortcomings of callbacks as async expression and management.

As soon as we introduce a single continuation (or several dozen as many programs do!) in the form of a callback function, we have allowed a divergence to form between how our brains work and the way the code will operate. Any time these two diverge (and this is by far not the only place that happens, as I'm sure you know!), we run into the inevitable fact that our code becomes harder to understand, reason about, debug, and maintain.

Sequential Brain

I'm pretty sure most of you readers have heard someone say (even made the claim yourself), "I'm a multitasker." The effects of trying to act as a multitasker range from humorous (e.g., the silly patting-head-rubbing-stomach kids' game) to mundane (chewing

gum while walking) to downright dangerous (texting while driving).

But are we multitaskers? Can we really do two conscious, intentional actions at once and think/reason about both of them at exactly the same moment? Does our highest level of brain functionality have parallel multithreading going on?

The answer may surprise you: **probably not.**

That's just not really how our brains appear to be set up. We're much more single taskers than many of us (especially A-type personalities!) would like to admit. We can really only think about one thing at any given instant.

I'm not talking about all our involuntary, subconscious, automatic brain functions, such as heart beating, breathing, and eyelid blinking. Those are all vital tasks to our sustained life, but we don't intentionally allocate any brain power to them. Thankfully, while we obsess about checking social network feeds for the 15th time in three minutes, our brain carries on in the background (threads!) with all those important tasks.

We're instead talking about whatever task is at the forefront of our minds at the moment. For me, it's writing the text in this book right now. Am I doing any other higher level brain function at exactly this same moment? Nope, not really. I get distracted quickly and easily -- a few dozen times in these last couple of paragraphs!

When we *fake* multitasking, such as trying to type something at the same time we're talking to a friend or family member on the phone, what we're actually most likely doing is acting as fast context switchers. In other words, we switch back and forth between two or more tasks in rapid succession, *simultaneously* progressing on each task in tiny, fast little chunks. We do it so fast that to the outside world it appears as if we're doing these things *in parallel*.

Does that sound suspiciously like async evented concurrency (like the sort that happens in JS) to you?! If not, go back and read Chapter 1 again!

In fact, one way of simplifying (i.e., abusing) the massively complex world of neurology into something I can remotely hope to discuss here is that our brains work kinda like the event loop queue.

If you think about every single letter (or word) I type as a single async event, in just this sentence alone there are several dozen opportunities for my brain to be interrupted by some other event, such as from my senses, or even just my random thoughts.

I don't get interrupted and pulled to another "process" at every opportunity that I could be (thankfully -- or this book would never be written!). But it happens often enough that I feel my own brain is nearly constantly switching to various different contexts (aka "processes"). And that's an awful lot like how the JS engine would probably feel.

Doing Versus Planning

OK, so our brains can be thought of as operating in single-threaded event loop queue like ways, as can the JS engine. That sounds like a good match.

But we need to be more nuanced than that in our analysis. There's a big, observable difference between how we plan various tasks, and how our brains actually operate those tasks.

Again, back to the writing of this text as my metaphor. My rough mental outline plan here is to keep writing and writing, going sequentially through a set of points I have ordered in my thoughts. I don't plan to have any interruptions or nonlinear activity in this writing. But yet, my brain is nevertheless switching around all the time.

Even though at an operational level our brains are async evented, we seem to plan out tasks in a sequential, synchronous way. "I need to go to the store, then buy some milk, then drop off my dry cleaning."

You'll notice that this higher level thinking (planning) doesn't seem very async evented in its formulation. In fact, it's kind of rare for us to deliberately think solely in terms of events. Instead, we plan things out carefully, sequentially (A then B then C), and we assume to an extent a sort of temporal blocking that forces B to wait on A, and C to wait on B.

When a developer writes code, they are planning out a set of actions to occur. If they're any good at being a developer, they're **carefully planning** it out. "I need to set `z` to the value of `x`, and then `x` to the value of `y`," and so forth.

When we write out synchronous code, statement by statement, it works a lot like our errands to-do list:

```
// swap `x` and `y` (via temp variable `z`)
z = x;
x = y;
y = z;
```

These three assignment statements are synchronous, so `x = y` waits for `z = x` to finish, and `y = z` in turn waits for `x = y` to finish. Another way of saying it is that these three statements are temporally bound to execute in a certain order, one right after the other. Thankfully, we don't need to be bothered with any async evented details here. If we did, the code gets a lot more complex, quickly!

So if synchronous brain planning maps well to synchronous code statements, how well do our brains do at planning out asynchronous code?

It turns out that how we express asynchrony (with callbacks) in our code doesn't map very well at all to that synchronous brain planning behavior.

Can you actually imagine having a line of thinking that plans out your to-do errands like this?

"I need to go to the store, but on the way I'm sure I'll get a phone call, so 'Hi, Mom', and while she starts talking, I'll be looking up the store address on GPS, but that'll take a second to load, so I'll turn down the radio so I can hear Mom better, then I'll realize I forgot to put on a jacket and it's cold outside, but no matter, keep driving and talking to Mom, and then the seatbelt ding reminds me to buckle up, so 'Yes, Mom, I am wearing my seatbelt, I always do!'. Ah, finally the GPS got the directions, now..."

As ridiculous as that sounds as a formulation for how we plan our day out and think about what to do and in what order, nonetheless it's exactly how our brains operate at a functional level. Remember, that's not multitasking, it's just fast context switching.

The reason it's difficult for us as developers to write async evented code, especially when all we have is the callback to do it, is that stream of consciousness thinking/planning is unnatural for most of us.

We think in step-by-step terms, but the tools (callbacks) available to us in code are not expressed in a step-by-step fashion once we move from synchronous to asynchronous.

And **that** is why it's so hard to accurately author and reason about async JS code with callbacks: because it's not how our brain planning works.

Note: The only thing worse than not knowing why some code breaks is not knowing why it worked in the first place! It's the classic "house of cards" mentality: "it works, but not sure why, so nobody touch it!" You may have heard, "Hell is other people" (Sartre), and the programmer meme twist, "Hell is other people's code." I believe truly: "Hell is not understanding my own code." And callbacks are one main culprit.

Nested/Chained Callbacks

Consider:

```

listen( "click", function handler(evt){
    setTimeout( function request(){
        ajax( "http://some.url.1", function response(text){
            if (text == "hello") {
                handler();
            }
            else if (text == "world") {
                request();
            }
        } );
    }, 500) ;
} );

```

There's a good chance code like that is recognizable to you. We've got a chain of three functions nested together, each one representing a step in an asynchronous series (task, "process").

This kind of code is often called "callback hell," and sometimes also referred to as the "pyramid of doom" (for its sideways-facing triangular shape due to the nested indentation).

But "callback hell" actually has almost nothing to do with the nesting/indentation. It's a far deeper problem than that. We'll see how and why as we continue through the rest of this chapter.

First, we're waiting for the "click" event, then we're waiting for the timer to fire, then we're waiting for the Ajax response to come back, at which point it might do it all again.

At first glance, this code may seem to map its asynchrony naturally to sequential brain planning.

First (*now*), we:

```

listen( "...", function handler(...){
    // ...
} );

```

Then *later*, we:

```

setTimeout( function request(...){
    // ...
}, 500) ;

```

Then still *later*, we:

```
ajax( "...", function response(...){
    // ...
} );
```

And finally (most *later*), we:

```
if ( ... ) {
    // ...
}
else ...
```

But there's several problems with reasoning about this code linearly in such a fashion.

First, it's an accident of the example that our steps are on subsequent lines (1, 2, 3, and 4...). In real async JS programs, there's often a lot more noise cluttering things up, noise that we have to deftly maneuver past in our brains as we jump from one function to the next. Understanding the async flow in such callback-laden code is not impossible, but it's certainly not natural or easy, even with lots of practice.

But also, there's something deeper wrong, which isn't evident just in that code example. Let me make up another scenario (pseudocode-ish) to illustrate it:

```
doA( function(){
    doB();

    doC( function(){
        doD();
    } )

    doE();
} );

doF();
```

While the experienced among you will correctly identify the true order of operations here, I'm betting it is more than a little confusing at first glance, and takes some concerted mental cycles to arrive at. The operations will happen in this order:

- doA()
- doF()
- doB()
- doC()
- doE()
- doD()

Did you get that right the very first time you glanced at the code?

OK, some of you are thinking I was unfair in my function naming, to intentionally lead you astray. I swear I was just naming in top-down appearance order. But let me try again:

```
doA( function(){  
    doC();  
  
    doD( function(){  
        doF();  
    } )  
  
    doE();  
} );  
  
doB();
```

Now, I've named them alphabetically in order of actual execution. But I still bet, even with experience now in this scenario, tracing through the `A -> B -> C -> D -> E -> F` order doesn't come natural to many if any of you readers. Certainly your eyes do an awful lot of jumping up and down the code snippet, right?

But even if that all comes natural to you, there's still one more hazard that could wreak havoc. Can you spot what it is?

What if `doA(...)` or `doD(...)` aren't actually async, the way we obviously assumed them to be? Uh oh, now the order is different. If they're both sync (and maybe only sometimes, depending on the conditions of the program at the time), the order is now `A -> C -> D -> F -> E -> B`.

That sound you just heard faintly in the background is the sighs of thousands of JS developers who just had a face-in-hands moment.

Is nesting the problem? Is that what makes it so hard to trace the async flow? That's part of it, certainly.

But let me rewrite the previous nested event/timeout/Ajax example without using nesting:

```

listen( "click", handler );

function handler() {
    setTimeout( request, 500 );
}

function request(){
    ajax( "http://some.url.1", response );
}

function response(text){
    if (text == "hello") {
        handler();
    }
    else if (text == "world") {
        request();
    }
}

```

This formulation of the code is not hardly as recognizable as having the nesting/indentation woes of its previous form, and yet it's every bit as susceptible to "callback hell." Why?

As we go to linearly (sequentially) reason about this code, we have to skip from one function, to the next, to the next, and bounce all around the code base to "see" the sequence flow. And remember, this is simplified code in sort of best-case fashion. We all know that real async JS program code bases are often fantastically more jumbled, which makes such reasoning orders of magnitude more difficult.

Another thing to notice: to get steps 2, 3, and 4 linked together so they happen in succession, the only affordance callbacks alone gives us is to hardcode step 2 into step 1, step 3 into step 2, step 4 into step 3, and so on. The hardcoded isn't necessarily a bad thing, if it really is a fixed condition that step 2 should always lead to step 3.

But the hardcoded definitely makes the code a bit more brittle, as it doesn't account for anything going wrong that might cause a deviation in the progression of steps. For example, if step 2 fails, step 3 never gets reached, nor does step 2 retry, or move to an alternate error handling flow, and so on.

All of these issues are things you *can* manually hardcode into each step, but that code is often very repetitive and not reusable in other steps or in other async flows in your program.

Even though our brains might plan out a series of tasks in a sequential type of way (this, then this, then this), the evented nature of our brain operation makes recovery/retry/forking of flow control almost effortless. If you're out running errands, and you realize you left a

shopping list at home, it doesn't end the day because you didn't plan that ahead of time. Your brain routes around this hiccup easily: you go home, get the list, then head right back out to the store.

But the brittle nature of manually hardcoded callbacks (even with hardcoded error handling) is often far less graceful. Once you end up specifying (aka pre-planning) all the various eventualities/paths, the code becomes so convoluted that it's hard to ever maintain or update it.

That is what "callback hell" is all about! The nesting/indentation are basically a side show, a red herring.

And as if all that's not enough, we haven't even touched what happens when two or more chains of these callback continuations are happening *simultaneously*, or when the third step branches out into "parallel" callbacks with gates or latches, or... OMG, my brain hurts, how about yours!?

Are you catching the notion here that our sequential, blocking brain planning behaviors just don't map well onto callback-oriented async code? That's the first major deficiency to articulate about callbacks: they express asynchrony in code in ways our brains have to fight just to keep in sync with (pun intended!).

Trust Issues

The mismatch between sequential brain planning and callback-driven async JS code is only part of the problem with callbacks. There's something much deeper to be concerned about.

Let's once again revisit the notion of a callback function as the continuation (aka the second half) of our program:

```
// A
ajax( "...", function(...){
  // C
} );
// B
```

// A and // B happen *now*, under the direct control of the main JS program. But // c gets deferred to happen *later*, and under the control of another party -- in this case, the ajax(...) function. In a basic sense, that sort of hand-off of control doesn't regularly cause lots of problems for programs.

But don't be fooled by its infrequency that this control switch isn't a big deal. In fact, it's one of the worst (and yet most subtle) problems about callback-driven design. It revolves around the idea that sometimes `ajax(...)` (i.e., the "party" you hand your callback continuation to) is not a function that you wrote, or that you directly control. Many times it's a utility provided by some third party.

We call this "inversion of control," when you take part of your program and give over control of its execution to another third party. There's an unspoken "contract" that exists between your code and the third-party utility -- a set of things you expect to be maintained.

Tale of Five Callbacks

It might not be terribly obvious why this is such a big deal. Let me construct an exaggerated scenario to illustrate the hazards of trust at play.

Imagine you're a developer tasked with building out an ecommerce checkout system for a site that sells expensive TVs. You already have all the various pages of the checkout system built out just fine. On the last page, when the user clicks "confirm" to buy the TV, you need to call a third-party function (provided say by some analytics tracking company) so that the sale can be tracked.

You notice that they've provided what looks like an async tracking utility, probably for the sake of performance best practices, which means you need to pass in a callback function. In this continuation that you pass in, you will have the final code that charges the customer's credit card and displays the thank you page.

This code might look like:

```
analytics.trackPurchase( purchaseData, function(){
    chargeCreditCard();
    displayThankyouPage();
});
```

Easy enough, right? You write the code, test it, everything works, and you deploy to production. Everyone's happy!

Six months go by and no issues. You've almost forgotten you even wrote that code. One morning, you're at a coffee shop before work, casually enjoying your latte, when you get a panicked call from your boss insisting you drop the coffee and rush into work right away.

When you arrive, you find out that a high-profile customer has had his credit card charged five times for the same TV, and he's understandably upset. Customer service has already issued an apology and processed a refund. But your boss demands to know how this could possibly have happened. "Don't we have tests for stuff like this!?"

You don't even remember the code you wrote. But you dig back in and start trying to find out what could have gone awry.

After digging through some logs, you come to the conclusion that the only explanation is that the analytics utility somehow, for some reason, called your callback five times instead of once. Nothing in their documentation mentions anything about this.

Frustrated, you contact customer support, who of course is as astonished as you are. They agree to escalate it to their developers, and promise to get back to you. The next day, you receive a lengthy email explaining what they found, which you promptly forward to your boss.

Apparently, the developers at the analytics company had been working on some experimental code that, under certain conditions, would retry the provided callback once per second, for five seconds, before failing with a timeout. They had never intended to push that into production, but somehow they did, and they're totally embarrassed and apologetic. They go into plenty of detail about how they've identified the breakdown and what they'll do to ensure it never happens again. Yadda, yadda.

What's next?

You talk it over with your boss, but he's not feeling particularly comfortable with the state of things. He insists, and you reluctantly agree, that you can't trust *them* anymore (that's what bit you), and that you'll need to figure out how to protect the checkout code from such a vulnerability again.

After some tinkering, you implement some simple ad hoc code like the following, which the team seems happy with:

```
var tracked = false;

analytics.trackPurchase( purchaseData, function(){
    if (!tracked) {
        tracked = true;
        chargeCreditCard();
        displayThankyouPage();
    }
});
```

Note: This should look familiar to you from Chapter 1, because we're essentially creating a latch to handle if there happen to be multiple concurrent invocations of our callback.

But then one of your QA engineers asks, "what happens if they never call the callback?" Oops. Neither of you had thought about that.

You begin to chase down the rabbit hole, and think of all the possible things that could go wrong with them calling your callback. Here's roughly the list you come up with of ways the analytics utility could misbehave:

- Call the callback too early (before it's been tracked)
- Call the callback too late (or never)
- Call the callback too few or too many times (like the problem you encountered!)
- Fail to pass along any necessary environment/parameters to your callback
- Swallow any errors/exceptions that may happen
- ...

That should feel like a troubling list, because it is. You're probably slowly starting to realize that you're going to have to invent an awful lot of ad hoc logic **in each and every single callback** that's passed to a utility you're not positive you can trust.

Now you realize a bit more completely just how hellish "callback hell" is.

Not Just Others' Code

Some of you may be skeptical at this point whether this is as big a deal as I'm making it out to be. Perhaps you don't interact with truly third-party utilities much if at all. Perhaps you use versioned APIs or self-host such libraries, so that its behavior can't be changed out from underneath you.

So, contemplate this: can you even *really* trust utilities that you do theoretically control (in your own code base)?

Think of it this way: most of us agree that at least to some extent we should build our own internal functions with some defensive checks on the input parameters, to reduce/prevent unexpected issues.

Overly trusting of input:

```
function addNumbers(x,y) {
    // + is overloaded with coercion to also be
    // string concatenation, so this operation
    // isn't strictly safe depending on what's
    // passed in.
    return x + y;
}

addNumbers( 21, 21 );      // 42
addNumbers( 21, "21" );    // "2121"
```

Defensive against untrusted input:

```

function addNumbers(x,y) {
    // ensure numerical input
    if (typeof x != "number" || typeof y != "number") {
        throw Error( "Bad parameters" );
    }

    // if we get here, + will safely do numeric addition
    return x + y;
}

addNumbers( 21, 21 );      // 42
addNumbers( 21, "21" );    // Error: "Bad parameters"

```

Or perhaps still safe but friendlier:

```

function addNumbers(x,y) {
    // ensure numerical input
    x = Number( x );
    y = Number( y );

    // + will safely do numeric addition
    return x + y;
}

addNumbers( 21, 21 );      // 42
addNumbers( 21, "21" );    // 42

```

However you go about it, these sorts of checks/normalizations are fairly common on function inputs, even with code we theoretically entirely trust. In a crude sort of way, it's like the programming equivalent of the geopolitical principle of "Trust But Verify."

So, doesn't it stand to reason that we should do the same thing about composition of async function callbacks, not just with truly external code but even with code we know is generally "under our own control"? **Of course we should.**

But callbacks don't really offer anything to assist us. We have to construct all that machinery ourselves, and it often ends up being a lot of boilerplate/overhead that we repeat for every single async callback.

The most troublesome problem with callbacks is *inversion of control* leading to a complete breakdown along all those trust lines.

If you have code that uses callbacks, especially but not exclusively with third-party utilities, and you're not already applying some sort of mitigation logic for all these *inversion of control* trust issues, your code *has* bugs in it right now even though they may not have bitten you yet. Latent bugs are still bugs.

Hell indeed.

Trying to Save Callbacks

There are several variations of callback design that have attempted to address some (not all!) of the trust issues we've just looked at. It's a valiant, but doomed, effort to save the callback pattern from imploding on itself.

For example, regarding more graceful error handling, some API designs provide for split callbacks (one for the success notification, one for the error notification):

```
function success(data) {
    console.log( data );
}

function failure(err) {
    console.error( err );
}

ajax( "http://some.url.1", success, failure );
```

In APIs of this design, often the `failure()` error handler is optional, and if not provided it will be assumed you want the errors swallowed. Ugh.

Note: This split-callback design is what the ES6 Promise API uses. We'll cover ES6 Promises in much more detail in the next chapter.

Another common callback pattern is called "error-first style" (sometimes called "Node style," as it's also the convention used across nearly all Node.js APIs), where the first argument of a single callback is reserved for an error object (if any). If success, this argument will be empty/falsy (and any subsequent arguments will be the success data), but if an error result is being signaled, the first argument is set/truthy (and usually nothing else is passed):

```
function response(err,data) {
    // error?
    if (err) {
        console.error( err );
    }
    // otherwise, assume success
    else {
        console.log( data );
    }
}

ajax( "http://some.url.1", response );
```

In both of these cases, several things should be observed.

First, it has not really resolved the majority of trust issues like it may appear. There's nothing about either callback that prevents or filters unwanted repeated invocations. Moreover, things are worse now, because you may get both success and error signals, or neither, and you still have to code around either of those conditions.

Also, don't miss the fact that while it's a standard pattern you can employ, it's definitely more verbose and boilerplate-ish without much reuse, so you're going to get weary of typing all that out for every single callback in your application.

What about the trust issue of never being called? If this is a concern (and it probably should be!), you likely will need to set up a timeout that cancels the event. You could make a utility (proof-of-concept only shown) to help you with that:

```
function timeoutify(fn,delay) {
  var intv = setTimeout( function(){
    intv = null;
    fn( new Error( "Timeout!" ) );
  }, delay )
;

  return function() {
    // timeout hasn't happened yet?
    if (intv) {
      clearTimeout( intv );
      fn.apply( this, [ null ].concat( [].slice.call( arguments ) ) );
    }
  };
}
```

Here's how you use it:

```
// using "error-first style" callback design
function foo(err,data) {
  if (err) {
    console.error( err );
  }
  else {
    console.log( data );
  }
}

ajax( "http://some.url.1", timeoutify( foo, 500 ) );
```

Another trust issue is being called "too early." In application-specific terms, this may actually involve being called before some critical task is complete. But more generally, the problem is evident in utilities that can either invoke the callback you provide *now* (synchronously), or *later* (asynchronously).

This nondeterminism around the sync-or-async behavior is almost always going to lead to very difficult to track down bugs. In some circles, the fictional insanity-inducing monster named Zalgo is used to describe the sync/async nightmares. "Don't release Zalgo!" is a common cry, and it leads to very sound advice: always invoke callbacks asynchronously, even if that's "right away" on the next turn of the event loop, so that all callbacks are predictably async.

Note: For more information on Zalgo, see Oren Golan's "Don't Release Zalgo!" (<https://github.com/oren/oren.github.io/blob/master/posts/zalgo.md>) and Isaac Z. Schlueter's "Designing APIs for Asynchrony" (<http://blog.izs.me/post/59142742143/designing-apis-for-asynchrony>).

Consider:

```
function result(data) {
    console.log( a );
}

var a = 0;

ajax( "...pre-cached-url...", result );
a++;
```

Will this code print `0` (sync callback invocation) or `1` (async callback invocation)?

Depends... on the conditions.

You can see just how quickly the unpredictability of Zalgo can threaten any JS program. So the silly-sounding "never release Zalgo" is actually incredibly common and solid advice. Always be asyncing.

What if you don't know whether the API in question will always execute async? You could invent a utility like this `asyncify(..)` proof-of-concept:

```

function asyncify(fn) {
  var orig_fn = fn,
    intv = setTimeout( function(){
      intv = null;
      if (fn) fn();
    }, 0 )
  ;

  fn = null;

  return function() {
    // firing too quickly, before `intv` timer has fired to
    // indicate async turn has passed?
    if (intv) {
      fn = orig_fn.bind.apply(
        orig_fn,
        // add the wrapper's `this` to the `bind(..)`
        // call parameters, as well as currying any
        // passed in parameters
        [this].concat( [].slice.call( arguments ) )
      );
    }
    // already async
    else {
      // invoke original function
      orig_fn.apply( this, arguments );
    }
  };
}

```

You use `asyncify(..)` like this:

```

function result(data) {
  console.log( a );
}

var a = 0;

ajax( "..pre-cached-url..", asyncify( result ) );
a++;

```

Whether the Ajax request is in the cache and resolves to try to call the callback right away, or must be fetched over the wire and thus complete later asynchronously, this code will always output `1` instead of `0` -- `result(..)` cannot help but be invoked asynchronously, which means the `a++` has a chance to run before `result(..)` does.

Yay, another trust issued "solved"! But it's inefficient, and yet again more bloated boilerplate to weigh your project down.

That's just the story, over and over again, with callbacks. They can do pretty much anything you want, but you have to be willing to work hard to get it, and oftentimes this effort is much more than you can or should spend on such code reasoning.

You might find yourself wishing for built-in APIs or other language mechanics to address these issues. Finally ES6 has arrived on the scene with some great answers, so keep reading!

Review

Callbacks are the fundamental unit of asynchrony in JS. But they're not enough for the evolving landscape of async programming as JS matures.

First, our brains plan things out in sequential, blocking, single-threaded semantic ways, but callbacks express asynchronous flow in a rather nonlinear, nonsequential way, which makes reasoning properly about such code much harder. Bad to reason about code is bad code that leads to bad bugs.

We need a way to express asynchrony in a more synchronous, sequential, blocking manner, just like our brains do.

Second, and more importantly, callbacks suffer from *inversion of control* in that they implicitly give control over to another party (often a third-party utility not in your control!) to invoke the *continuation* of your program. This control transfer leads us to a troubling list of trust issues, such as whether the callback is called more times than we expect.

Inventing ad hoc logic to solve these trust issues is possible, but it's more difficult than it should be, and it produces clunkier and harder to maintain code, as well as code that is likely insufficiently protected from these hazards until you get visibly bitten by the bugs.

We need a generalized solution to **all of the trust issues**, one that can be reused for as many callbacks as we create without all the extra boilerplate overhead.

We need something better than callbacks. They've served us well to this point, but the *future* of JavaScript demands more sophisticated and capable async patterns. The subsequent chapters in this book will dive into those emerging evolutions.

In Chapter 2, we identified two major categories of deficiencies with using callbacks to express program asynchrony and manage concurrency: lack of sequentiality and lack of trustability. Now that we understand the problems more intimately, it's time we turn our attention to patterns that can address them.

The issue we want to address first is the *inversion of control*, the trust that is so fragilely held and so easily lost.

Recall that we wrap up the *continuation* of our program in a callback function, and hand that callback over to another party (potentially even external code) and just cross our fingers that it will do the right thing with the invocation of the callback.

We do this because we want to say, "here's what happens *later*, after the current step finishes."

But what if we could uninvert that *inversion of control*? What if instead of handing the continuation of our program to another party, we could expect it to return us a capability to know when its task finishes, and then our code could decide what to do next?

This paradigm is called **Promises**.

Promises are starting to take the JS world by storm, as developers and specification writers alike desperately seek to untangle the insanity of callback hell in their code/design. In fact, most new async APIs being added to JS/DOM platform are being built on Promises. So it's probably a good idea to dig in and learn them, don't you think!?

Note: The word "immediately" will be used frequently in this chapter, generally to refer to some Promise resolution action. However, in essentially all cases, "immediately" means in terms of the Job queue behavior (see Chapter 1), not in the strictly synchronous *now* sense.

What Is a Promise?

When developers decide to learn a new technology or pattern, usually their first step is "Show me the code!" It's quite natural for us to just jump in feet first and learn as we go.

But it turns out that some abstractions get lost on the APIs alone. Promises are one of those tools where it can be painfully obvious from how someone uses it whether they understand what it's for and about versus just learning and using the API.

So before I show the Promise code, I want to fully explain what a Promise really is conceptually. I hope this will then guide you better as you explore integrating Promise theory into your own async flow.

With that in mind, let's look at two different analogies for what a Promise *is*.

Future Value

Imagine this scenario: I walk up to the counter at a fast-food restaurant, and place an order for a cheeseburger. I hand the cashier \$1.47. By placing my order and paying for it, I've made a request for a *value* back (the cheeseburger). I've started a transaction.

But often, the cheeseburger is not immediately available for me. The cashier hands me something in place of my cheeseburger: a receipt with an order number on it. This order number is an IOU ("I owe you") *promise* that ensures that eventually, I should receive my cheeseburger.

So I hold onto my receipt and order number. I know it represents my *future cheeseburger*, so I don't need to worry about it anymore -- aside from being hungry!

While I wait, I can do other things, like send a text message to a friend that says, "Hey, can you come join me for lunch? I'm going to eat a cheeseburger."

I am reasoning about my *future cheeseburger* already, even though I don't have it in my hands yet. My brain is able to do this because it's treating the order number as a placeholder for the cheeseburger. The placeholder essentially makes the value *time independent*. It's a **future value**.

Eventually, I hear, "Order 113!" and I gleefully walk back up to the counter with receipt in hand. I hand my receipt to the cashier, and I take my cheeseburger in return.

In other words, once my *future value* was ready, I exchanged my value-promise for the value itself.

But there's another possible outcome. They call my order number, but when I go to retrieve my cheeseburger, the cashier regretfully informs me, "I'm sorry, but we appear to be all out of cheeseburgers." Setting aside the customer frustration of this scenario for a moment, we can see an important characteristic of *future values*: they can either indicate a success or failure.

Every time I order a cheeseburger, I know that I'll either get a cheeseburger eventually, or I'll get the sad news of the cheeseburger shortage, and I'll have to figure out something else to eat for lunch.

Note: In code, things are not quite as simple, because metaphorically the order number may never be called, in which case we're left indefinitely in an unresolved state. We'll come back to dealing with that case later.

Values Now and Later

This all might sound too mentally abstract to apply to your code. So let's be more concrete.

However, before we can introduce how Promises work in this fashion, we're going to derive in code that we already understand -- callbacks! -- how to handle these *future values*.

When you write code to reason about a value, such as performing math on a `number`, whether you realize it or not, you've been assuming something very fundamental about that value, which is that it's a concrete *now* value already:

```
var x, y = 2;

console.log( x + y ); // NaN <-- because `x` isn't set yet
```

The `x + y` operation assumes both `x` and `y` are already set. In terms we'll expound on shortly, we assume the `x` and `y` values are already *resolved*.

It would be nonsense to expect that the `+` operator by itself would somehow be magically capable of detecting and waiting around until both `x` and `y` are resolved (aka ready), only then to do the operation. That would cause chaos in the program if different statements finished *now* and others finished *later*, right?

How could you possibly reason about the relationships between two statements if either one (or both) of them might not be finished yet? If statement 2 relies on statement 1 being finished, there are just two outcomes: either statement 1 finished right *now* and everything proceeds fine, or statement 1 didn't finish yet, and thus statement 2 is going to fail.

If this sort of thing sounds familiar from Chapter 1, good!

Let's go back to our `x + y` math operation. Imagine if there was a way to say, "Add `x` and `y`, but if either of them isn't ready yet, just wait until they are. Add them as soon as you can."

Your brain might have just jumped to callbacks. OK, so...

```

function add(getX, getY, cb) {
  var x, y;
  getX( function(xVal){
    x = xVal;
    // both are ready?
    if (y != undefined) {
      cb( x + y ); // send along sum
    }
  });
  getY( function(yVal){
    y = yVal;
    // both are ready?
    if (x != undefined) {
      cb( x + y ); // send along sum
    }
  });
}

// `fetchX()` and `fetchY()` are sync or async
// functions
add( fetchX, fetchY, function(sum){
  console.log( sum ); // that was easy, huh?
} );

```

Take just a moment to let the beauty (or lack thereof) of that snippet sink in (whistles patiently).

While the ugliness is undeniable, there's something very important to notice about this async pattern.

In that snippet, we treated `x` and `y` as future values, and we express an operation `add(...)` that (from the outside) does not care whether `x` or `y` or both are available right away or not. In other words, it normalizes the *now* and *later*, such that we can rely on a predictable outcome of the `add(...)` operation.

By using an `add(...)` that is temporally consistent -- it behaves the same across *now* and *later* times -- the async code is much easier to reason about.

To put it more plainly: to consistently handle both *now* and *later*, we make both of them *later*: all operations become async.

Of course, this rough callbacks-based approach leaves much to be desired. It's just a first tiny step toward realizing the benefits of reasoning about *future values* without worrying about the time aspect of when it's available or not.

Promise Value

We'll definitely go into a lot more detail about Promises later in the chapter -- so don't worry if some of this is confusing -- but let's just briefly glimpse at how we can express the `x + y` example via `Promise`s:

```
function add(xPromise, yPromise) {
    // `Promise.all([ .. ])` takes an array of promises,
    // and returns a new promise that waits on them
    // all to finish
    return Promise.all( [xPromise, yPromise] )

    // when that promise is resolved, let's take the
    // received `X` and `Y` values and add them together.
    .then( function(values){
        // `values` is an array of the messages from the
        // previously resolved promises
        return values[0] + values[1];
    } );
}

// `fetchX()` and `fetchY()` return promises for
// their respective values, which may be ready
// *now* or *later*.
add( fetchX(), fetchY() )

// we get a promise back for the sum of those
// two numbers.
// now we chain-call `then(..)` to wait for the
// resolution of that returned promise.
.then( function(sum){
    console.log( sum ); // that was easier!
} );
```

There are two layers of Promises in this snippet.

`fetchX()` and `fetchY()` are called directly, and the values they return (promises!) are passed into `add(..)`. The underlying values those promises represent may be ready *now* or *later*, but each promise normalizes the behavior to be the same regardless. We reason about `x` and `y` values in a time-independent way. They are *future values*.

The second layer is the promise that `add(..)` creates (via `Promise.all([..])`) and returns, which we wait on by calling `then(..)`. When the `add(..)` operation completes, our `sum` *future value* is ready and we can print it out. We hide inside of `add(..)` the logic for waiting on the `x` and `y` *future values*.

Note: Inside `add(..)`, the `Promise.all([..])` call creates a promise (which is waiting on `promiseX` and `promiseY` to resolve). The chained call to `.then(..)` creates another promise, which the `return values[0] + values[1]` line immediately resolves (with the result of the addition). Thus, the `then(..)` call we chain off the end of the `add(..)` call -- at the

end of the snippet -- is actually operating on that second promise returned, rather than the first one created by `Promise.all([...])`. Also, though we are not chaining off the end of that second `then(...)`, it too has created another promise, had we chosen to observe/use it. This Promise chaining stuff will be explained in much greater detail later in this chapter.

Just like with cheeseburger orders, it's possible that the resolution of a Promise is rejection instead of fulfillment. Unlike a fulfilled Promise, where the value is always programmatic, a rejection value -- commonly called a "rejection reason" -- can either be set directly by the program logic, or it can result implicitly from a runtime exception.

With Promises, the `then(...)` call can actually take two functions, the first for fulfillment (as shown earlier), and the second for rejection:

```
add( fetchX(), fetchY() )
  .then(
    // fulfillment handler
    function(sum) {
      console.log( sum );
    },
    // rejection handler
    function(err) {
      console.error( err ); // bummer!
    }
);
```

If something went wrong getting `x` or `y`, or something somehow failed during the addition, the promise that `add(...)` returns is rejected, and the second callback error handler passed to `then(...)` will receive the rejection value from the promise.

Because Promises encapsulate the time-dependent state -- waiting on the fulfillment or rejection of the underlying value -- from the outside, the Promise itself is time-independent, and thus Promises can be composed (combined) in predictable ways regardless of the timing or outcome underneath.

Moreover, once a Promise is resolved, it stays that way forever -- it becomes an *immutable value* at that point -- and can then be *observed* as many times as necessary.

Note: Because a Promise is externally immutable once resolved, it's now safe to pass that value around to any party and know that it cannot be modified accidentally or maliciously. This is especially true in relation to multiple parties observing the resolution of a Promise. It is not possible for one party to affect another party's ability to observe Promise resolution. Immutability may sound like an academic topic, but it's actually one of the most fundamental and important aspects of Promise design, and shouldn't be casually passed over.

That's one of the most powerful and important concepts to understand about Promises. With a fair amount of work, you could ad hoc create the same effects with nothing but ugly callback composition, but that's not really an effective strategy, especially because you have to do it over and over again.

Promises are an easily repeatable mechanism for encapsulating and composing *future values*.

Completion Event

As we just saw, an individual Promise behaves as a *future value*. But there's another way to think of the resolution of a Promise: as a flow-control mechanism -- a temporal this-then-that -- for two or more steps in an asynchronous task.

Let's imagine calling a function `foo(...)` to perform some task. We don't know about any of its details, nor do we care. It may complete the task right away, or it may take a while.

We just simply need to know when `foo(...)` finishes so that we can move on to our next task. In other words, we'd like a way to be notified of `foo(...)`'s completion so that we can *continue*.

In typical JavaScript fashion, if you need to listen for a notification, you'd likely think of that in terms of events. So we could reframe our need for notification as a need to listen for a *completion* (or *continuation*) event emitted by `foo(...)`.

Note: Whether you call it a "completion event" or a "continuation event" depends on your perspective. Is the focus more on what happens with `foo(...)`, or what happens *after* `foo(...)` finishes? Both perspectives are accurate and useful. The event notification tells us that `foo(...)` has *completed*, but also that it's OK to *continue* with the next step. Indeed, the callback you pass to be called for the event notification is itself what we've previously called a *continuation*. Because *completion event* is a bit more focused on the `foo(...)`, which more has our attention at present, we slightly favor *completion event* for the rest of this text.

With callbacks, the "notification" would be our callback invoked by the task (`foo(...)`). But with Promises, we turn the relationship around, and expect that we can listen for an event from `foo(...)`, and when notified, proceed accordingly.

First, consider some pseudocode:

```

foo(x) {
    // start doing something that could take a while
}

foo( 42 )

on (foo "completion") {
    // now we can do the next step!
}

on (foo "error") {
    // oops, something went wrong in `foo(..)`
}

```

We call `foo(..)` and then we set up two event listeners, one for `"completion"` and one for `"error"` -- the two possible *final* outcomes of the `foo(..)` call. In essence, `foo(..)` doesn't even appear to be aware that the calling code has subscribed to these events, which makes for a very nice *separation of concerns*.

Unfortunately, such code would require some "magic" of the JS environment that doesn't exist (and would likely be a bit impractical). Here's the more natural way we could express that in JS:

```

function foo(x) {
    // start doing something that could take a while

    // make a `listener` event notification
    // capability to return

    return listener;
}

var evt = foo( 42 );

evt.on( "completion", function(){
    // now we can do the next step!
} );

evt.on( "failure", function(err){
    // oops, something went wrong in `foo(..)`
} );

```

`foo(..)` expressly creates an event subscription capability to return back, and the calling code receives and registers the two event handlers against it.

The inversion from normal callback-oriented code should be obvious, and it's intentional. Instead of passing the callbacks to `foo(...)`, it returns an event capability we call `evt`, which receives the callbacks.

But if you recall from Chapter 2, callbacks themselves represent an *inversion of control*. So inverting the callback pattern is actually an *inversion of inversion*, or an *uninversion of control* -- restoring control back to the calling code where we wanted it to be in the first place.

One important benefit is that multiple separate parts of the code can be given the event listening capability, and they can all independently be notified of when `foo(...)` completes to perform subsequent steps after its completion:

```
var evt = foo( 42 );  
  
// let `bar(..)` listen to `foo(..)`'s completion  
bar( evt );  
  
// also, let `baz(..)` listen to `foo(..)`'s completion  
baz( evt );
```

Uninversion of control enables a nicer *separation of concerns*, where `bar(..)` and `baz(..)` don't need to be involved in how `foo(..)` is called. Similarly, `foo(..)` doesn't need to know or care that `bar(..)` and `baz(..)` exist or are waiting to be notified when `foo(..)` completes.

Essentially, this `evt` object is a neutral third-party negotiation between the separate concerns.

Promise "Events"

As you may have guessed by now, the `evt` event listening capability is an analogy for a Promise.

In a Promise-based approach, the previous snippet would have `foo(..)` creating and returning a `Promise` instance, and that promise would then be passed to `bar(..)` and `baz(..)`.

Note: The Promise resolution "events" we listen for aren't strictly events (though they certainly behave like events for these purposes), and they're not typically called "completion" or "error". Instead, we use `then(..)` to register a "then" event. Or perhaps more precisely, `then(..)` registers "fulfillment" and/or "rejection" event(s), though we don't see those terms used explicitly in the code.

Consider:

```
function foo(x) {
  // start doing something that could take a while

  // construct and return a promise
  return new Promise( function(resolve,reject){
    // eventually, call `resolve(..)` or `reject(..)`,
    // which are the resolution callbacks for
    // the promise.
  } );
}

var p = foo( 42 );

bar( p );

baz( p );
```

Note: The pattern shown with `new Promise(function(...){ ... })` is generally called the **"revealing constructor"**. The function passed in is executed immediately (not async deferred, as callbacks to `then(...)` are), and it's provided two parameters, which in this case we've named `resolve` and `reject`. These are the resolution functions for the promise.

`resolve(..)` generally signals fulfillment, and `reject(..)` signals rejection.

You can probably guess what the internals of `bar(..)` and `baz(..)` might look like:

```
function bar(fooPromise) {
  // listen for `foo(..)` to complete
  fooPromise.then(
    function(){
      // `foo(..)` has now finished, so
      // do `bar(..)`'s task
    },
    function(){
      // oops, something went wrong in `foo(..)`
    }
  );
}

// ditto for `baz(..)`
```

Promise resolution doesn't necessarily need to involve sending along a message, as it did when we were examining Promises as *future values*. It can just be a flow-control signal, as used in the previous snippet.

Another way to approach this is:

```

function bar() {
  // `foo(..)` has definitely finished, so
  // do `bar(..)`'s task
}

function oopsBar() {
  // oops, something went wrong in `foo(..)`,
  // so `bar(..)` didn't run
}

// ditto for `baz()` and `oopsBaz()`

var p = foo( 42 );

p.then( bar, oopsBar );

p.then( baz, oopsBaz );

```

Note: If you've seen Promise-based coding before, you might be tempted to believe that the last two lines of that code could be written as `p.then(...).then(...)`, using chaining, rather than `p.then(...); p.then(...)`. That would have an entirely different behavior, so be careful! The difference might not be clear right now, but it's actually a different async pattern than we've seen thus far: splitting/forking. Don't worry! We'll come back to this point later in this chapter.

Instead of passing the `p` promise to `bar(..)` and `baz(..)`, we use the promise to control when `bar(..)` and `baz(..)` will get executed, if ever. The primary difference is in the error handling.

In the first snippet's approach, `bar(..)` is called regardless of whether `foo(..)` succeeds or fails, and it handles its own fallback logic if it's notified that `foo(..)` failed. The same is true for `baz(..)`, obviously.

In the second snippet, `bar(..)` only gets called if `foo(..)` succeeds, and otherwise `oopsBar(..)` gets called. Ditto for `baz(..)`.

Neither approach is *correct* per se. There will be cases where one is preferred over the other.

In either case, the promise `p` that comes back from `foo(..)` is used to control what happens next.

Moreover, the fact that both snippets end up calling `then(..)` twice against the same promise `p` illustrates the point made earlier, which is that Promises (once resolved) retain their same resolution (fulfillment or rejection) forever, and can subsequently be observed as many times as necessary.

Whenever `p` is resolved, the next step will always be the same, both *now* and *later*.

Thenable Duck Typing

In Promises-land, an important detail is how to know for sure if some value is a genuine Promise or not. Or more directly, is it a value that will behave like a Promise?

Given that Promises are constructed by the `new Promise(..)` syntax, you might think that `p instanceof Promise` would be an acceptable check. But unfortunately, there are a number of reasons that's not totally sufficient.

Mainly, you can receive a Promise value from another browser window (iframe, etc.), which would have its own Promise different from the one in the current window/frame, and that check would fail to identify the Promise instance.

Moreover, a library or framework may choose to vend its own Promises and not use the native ES6 `Promise` implementation to do so. In fact, you may very well be using Promises with libraries in older browsers that have no `Promise` at all.

When we discuss Promise resolution processes later in this chapter, it will become more obvious why a non-genuine-but-Promise-like value would still be very important to be able to recognize and assimilate. But for now, just take my word for it that it's a critical piece of the puzzle.

As such, it was decided that the way to recognize a Promise (or something that behaves like a Promise) would be to define something called a "thenable" as any object or function which has a `then(..)` method on it. It is assumed that any such value is a Promise-conforming thenable.

The general term for "type checks" that make assumptions about a value's "type" based on its shape (what properties are present) is called "duck typing" -- "If it looks like a duck, and quacks like a duck, it must be a duck" (see the *Types & Grammar* title of this book series). So the duck typing check for a thenable would roughly be:

```

if (
  p !== null &&
  (
    typeof p === "object" ||
    typeof p === "function"
  ) &&
  typeof p.then === "function"
) {
  // assume it's a thenable!
}
else {
  // not a thenable
}

```

Yuck! Setting aside the fact that this logic is a bit ugly to implement in various places, there's something deeper and more troubling going on.

If you try to fulfill a Promise with any object/function value that happens to have a `then(...)` function on it, but you weren't intending it to be treated as a Promise/thenable, you're out of luck, because it will automatically be recognized as thenable and treated with special rules (see later in the chapter).

This is even true if you didn't realize the value has a `then(...)` on it. For example:

```

var o = { then: function(){ } };

// make `v` be `[[Prototype]]`-linked to `o`
var v = Object.create( o );

v.someStuff = "cool";
v.otherStuff = "not so cool";

v.hasOwnProperty( "then" );           // false

```

`v` doesn't look like a Promise or thenable at all. It's just a plain object with some properties on it. You're probably just intending to send that value around like any other object.

But unknown to you, `v` is also `[[Prototype]]`-linked (see the *this & Object Prototypes* title of this book series) to another object `o`, which happens to have a `then(...)` on it. So the thenable duck typing checks will think and assume `v` is a thenable. Uh oh.

It doesn't even need to be something as directly intentional as that:

```
Object.prototype.then = function(){};
Array.prototype.then = function(){};

var v1 = { hello: "world" };
var v2 = [ "Hello", "World" ];
```

Both `v1` and `v2` will be assumed to be thenables. You can't control or predict if any other code accidentally or maliciously adds `then(..)` to `Object.prototype`, `Array.prototype`, or any of the other native prototypes. And if what's specified is a function that doesn't call either of its parameters as callbacks, then any Promise resolved with such a value will just silently hang forever! Crazy.

Sound implausible or unlikely? Perhaps.

But keep in mind that there were several well-known non-Promise libraries preexisting in the community prior to ES6 that happened to already have a method on them called `then(..)`. Some of those libraries chose to rename their own methods to avoid collision (that sucks!). Others have simply been relegated to the unfortunate status of "incompatible with Promise-based coding" in reward for their inability to change to get out of the way.

The standards decision to hijack the previously nonreserved -- and completely general-purpose sounding -- `then` property name means that no value (or any of its delegates), either past, present, or future, can have a `then(..)` function present, either on purpose or by accident, or that value will be confused for a thenable in Promises systems, which will probably create bugs that are really hard to track down.

Warning: I do not like how we ended up with duck typing of thenables for Promise recognition. There were other options, such as "branding" or even "anti-branding"; what we got seems like a worst-case compromise. But it's not all doom and gloom. Thenable duck typing can be helpful, as we'll see later. Just beware that thenable duck typing can be hazardous if it incorrectly identifies something as a Promise that isn't.

Promise Trust

We've now seen two strong analogies that explain different aspects of what Promises can do for our async code. But if we stop there, we've missed perhaps the single most important characteristic that the Promise pattern establishes: trust.

Whereas the *future values* and *completion events* analogies play out explicitly in the code patterns we've explored, it won't be entirely obvious why or how Promises are designed to solve all of the *inversion of control* trust issues we laid out in the "Trust Issues" section of

Chapter 2. But with a little digging, we can uncover some important guarantees that restore the confidence in async coding that Chapter 2 tore down!

Let's start by reviewing the trust issues with callbacks-only coding. When you pass a callback to a utility `foo(...)`, it might:

- Call the callback too early
- Call the callback too late (or never)
- Call the callback too few or too many times
- Fail to pass along any necessary environment/parameters
- swallow any errors/exceptions that may happen

The characteristics of Promises are intentionally designed to provide useful, repeatable answers to all these concerns.

Calling Too Early

Primarily, this is a concern of whether code can introduce Zalgo-like effects (see Chapter 2), where sometimes a task finishes synchronously and sometimes asynchronously, which can lead to race conditions.

Promises by definition cannot be susceptible to this concern, because even an immediately fulfilled Promise (like `new Promise(function(resolve){ resolve(42); })`) cannot be *observed* synchronously.

That is, when you call `then(...)` on a Promise, even if that Promise was already resolved, the callback you provide to `then(...)` will **always** be called asynchronously (for more on this, refer back to "Jobs" in Chapter 1).

No more need to insert your own `setTimeout(..., 0)` hacks. Promises prevent Zalgo automatically.

Calling Too Late

Similar to the previous point, a Promise's `then(...)` registered observation callbacks are automatically scheduled when either `resolve(...)` or `reject(...)` are called by the Promise creation capability. Those scheduled callbacks will predictably be fired at the next asynchronous moment (see "Jobs" in Chapter 1).

It's not possible for synchronous observation, so it's not possible for a synchronous chain of tasks to run in such a way to in effect "delay" another callback from happening as expected. That is, when a Promise is resolved, all `then(...)` registered callbacks on it will be called, in

order, immediately at the next asynchronous opportunity (again, see "Jobs" in Chapter 1), and nothing that happens inside of one of those callbacks can affect/delay the calling of the other callbacks.

For example:

```
p.then( function(){
  p.then( function(){
    console.log( "C" );
  } );
  console.log( "A" );
} );
p.then( function(){
  console.log( "B" );
} );
// A B C
```

Here, `"C"` cannot interrupt and precede `"B"`, by virtue of how Promises are defined to operate.

Promise Scheduling Quirks

It's important to note, though, that there are lots of nuances of scheduling where the relative ordering between callbacks chained off two separate Promises is not reliably predictable.

If two promises `p1` and `p2` are both already resolved, it should be true that `p1.then(..); p2.then(..)` would end up calling the callback(s) for `p1` before the ones for `p2`. But there are subtle cases where that might not be true, such as the following:

```

var p3 = new Promise( function(resolve,reject){
    resolve( "B" );
} );

var p1 = new Promise( function(resolve,reject){
    resolve( p3 );
} );

var p2 = new Promise( function(resolve,reject){
    resolve( "A" );
} );

p1.then( function(v){
    console.log( v );
} );

p2.then( function(v){
    console.log( v );
} );

// A B  <-- not  B A  as you might expect

```

We'll cover this more later, but as you can see, `p1` is resolved not with an immediate value, but with another promise `p3` which is itself resolved with the value `"B"`. The specified behavior is to *unwrap* `p3` into `p1`, but asynchronously, so `p1`'s callback(s) are *behind* `p2`'s callback(s) in the asynchronous Job queue (see Chapter 1).

To avoid such nuanced nightmares, you should never rely on anything about the ordering/scheduling of callbacks across Promises. In fact, a good practice is not to code in such a way where the ordering of multiple callbacks matters at all. Avoid that if you can.

Never Calling the Callback

This is a very common concern. It's addressable in several ways with Promises.

First, nothing (not even a JS error) can prevent a Promise from notifying you of its resolution (if it's resolved). If you register both fulfillment and rejection callbacks for a Promise, and the Promise gets resolved, one of the two callbacks will always be called.

Of course, if your callbacks themselves have JS errors, you may not see the outcome you expect, but the callback will in fact have been called. We'll cover later how to be notified of an error in your callback, because even those don't get swallowed.

But what if the Promise itself never gets resolved either way? Even that is a condition that Promises provide an answer for, using a higher level abstraction called a "race":

```
// a utility for timing out a Promise
function timeoutPromise(delay) {
  return new Promise( function(resolve,reject){
    setTimeout( function(){
      reject( "Timeout!" );
    }, delay );
  } );
}

// setup a timeout for `foo()`
Promise.race( [
  foo(),                  // attempt `foo()`
  timeoutPromise( 3000 )   // give it 3 seconds
] )
.then(
  function(){
    // `foo(..)` fulfilled in time!
  },
  function(err){
    // either `foo()` rejected, or it just
    // didn't finish in time, so inspect
    // `err` to know which
  }
);

```

There are more details to consider with this Promise timeout pattern, but we'll come back to it later.

Importantly, we can ensure a signal as to the outcome of `foo()`, to prevent it from hanging our program indefinitely.

Calling Too Few or Too Many Times

By definition, *one* is the appropriate number of times for the callback to be called. The "too few" case would be zero calls, which is the same as the "never" case we just examined.

The "too many" case is easy to explain. Promises are defined so that they can only be resolved once. If for some reason the Promise creation code tries to call `resolve(..)` or `reject(..)` multiple times, or tries to call both, the Promise will accept only the first resolution, and will silently ignore any subsequent attempts.

Because a Promise can only be resolved once, any `then(..)` registered callbacks will only ever be called once (each).

Of course, if you register the same callback more than once, (e.g., `p.then(f); p.then(f);`), it'll be called as many times as it was registered. The guarantee that a response function is called only once does not prevent you from shooting yourself in the foot.

Failing to Pass Along Any Parameters/Environment

Promises can have, at most, one resolution value (fulfillment or rejection).

If you don't explicitly resolve with a value either way, the value is `undefined`, as is typical in JS. But whatever the value, it will always be passed to all registered (and appropriate: fulfillment or rejection) callbacks, either *now* or in the future.

Something to be aware of: If you call `resolve(...)` or `reject(...)` with multiple parameters, all subsequent parameters beyond the first will be silently ignored. Although that might seem a violation of the guarantee we just described, it's not exactly, because it constitutes an invalid usage of the Promise mechanism. Other invalid usages of the API (such as calling `resolve(...)` multiple times) are similarly *protected*, so the Promise behavior here is consistent (if not a tiny bit frustrating).

If you want to pass along multiple values, you must wrap them in another single value that you pass, such as an `array` or an `object`.

As for environment, functions in JS always retain their closure of the scope in which they're defined (see the *Scope & Closures* title of this series), so they of course would continue to have access to whatever surrounding state you provide. Of course, the same is true of callbacks-only design, so this isn't a specific augmentation of benefit from Promises -- but it's a guarantee we can rely on nonetheless.

Swallowing Any Errors/Exceptions

In the base sense, this is a restatement of the previous point. If you reject a Promise with a `reason` (aka error message), that value is passed to the rejection callback(s).

But there's something much bigger at play here. If at any point in the creation of a Promise, or in the observation of its resolution, a JS exception error occurs, such as a `TypeError` or `ReferenceError`, that exception will be caught, and it will force the Promise in question to become rejected.

For example:

```

var p = new Promise( function(resolve,reject){
    foo.bar();      // `foo` is not defined, so error!
    resolve( 42 );  // never gets here :(
} );


p.then(  

    function fulfilled(){  

        // never gets here :(  

    },  

    function rejected(err){  

        // `err` will be a `TypeError` exception object  

        // from the `foo.bar()` line.  

    }  

);


```

The JS exception that occurs from `foo.bar()` becomes a Promise rejection that you can catch and respond to.

This is an important detail, because it effectively solves another potential Zalgo moment, which is that errors could create a synchronous reaction whereas nonerrors would be asynchronous. Promises turn even JS exceptions into asynchronous behavior, thereby reducing the race condition chances greatly.

But what happens if a Promise is fulfilled, but there's a JS exception error during the observation (in a `then(...)` registered callback)? Even those aren't lost, but you may find how they're handled a bit surprising, until you dig in a little deeper:

```

var p = new Promise( function(resolve,reject){
    resolve( 42 );
} );


p.then(  

    function fulfilled(msg){  

        foo.bar();  

        console.log( msg );    // never gets here :(  

    },  

    function rejected(err){  

        // never gets here either :(  

    }  

);


```

Wait, that makes it seem like the exception from `foo.bar()` really did get swallowed. Never fear, it didn't. But something deeper is wrong, which is that we've failed to listen for it. The `p.then(...)` call itself returns another promise, and it's *that* promise that will be rejected with the `TypeError` exception.

Why couldn't it just call the error handler we have defined there? Seems like a logical behavior on the surface. But it would violate the fundamental principle that Promises are **immutable** once resolved. `p` was already fulfilled to the value `42`, so it can't later be changed to a rejection just because there's an error in observing `p`'s resolution.

Besides the principle violation, such behavior could wreak havoc, if say there were multiple `then(...)` registered callbacks on the promise `p`, because some would get called and others wouldn't, and it would be very opaque as to why.

Trustable Promise?

There's one last detail to examine to establish trust based on the Promise pattern.

You've no doubt noticed that Promises don't get rid of callbacks at all. They just change where the callback is passed to. Instead of passing a callback to `foo(...)`, we get *something* (ostensibly a genuine Promise) back from `foo(...)`, and we pass the callback to that *something* instead.

But why would this be any more trustable than just callbacks alone? How can we be sure the *something* we get back is in fact a trustable Promise? Isn't it basically all just a house of cards where we can trust only because we already trusted?

One of the most important, but often overlooked, details of Promises is that they have a solution to this issue as well. Included with the native ES6 `Promise` implementation is `Promise.resolve(...)`.

If you pass an immediate, non-Promise, non-thenable value to `Promise.resolve(...)`, you get a promise that's fulfilled with that value. In other words, these two promises `p1` and `p2` will behave basically identically:

```
var p1 = new Promise( function(resolve,reject){
  resolve( 42 );
} );

var p2 = Promise.resolve( 42 );
```

But if you pass a genuine Promise to `Promise.resolve(...)`, you just get the same promise back:

```
var p1 = Promise.resolve( 42 );

var p2 = Promise.resolve( p1 );

p1 === p2; // true
```

Even more importantly, if you pass a non-Promise thenable value to `Promise.resolve(..)`, it will attempt to unwrap that value, and the unwrapping will keep going until a concrete final non-Promise-like value is extracted.

Recall our previous discussion of thenables?

Consider:

```
var p = {
  then: function(cb) {
    cb( 42 );
  }
};

// this works OK, but only by good fortune
p
.then(
  function fulfilled(val){
    console.log( val ); // 42
  },
  function rejected(err){
    // never gets here
  }
);
```

This `p` is a thenable, but it's not a genuine Promise. Luckily, it's reasonable, as most will be. But what if you got back instead something that looked like:

```
var p = {
  then: function(cb,errcb) {
    cb( 42 );
    errcb( "evil laugh" );
  }
};

p
.then(
  function fulfilled(val){
    console.log( val ); // 42
  },
  function rejected(err){
    // oops, shouldn't have run
    console.log( err ); // evil laugh
  }
);
```

This `p` is a thenable but it's not so well behaved of a promise. Is it malicious? Or is it just ignorant of how Promises should work? It doesn't really matter, to be honest. In either case, it's not trustable as is.

Nonetheless, we can pass either of these versions of `p` to `Promise.resolve(..)`, and we'll get the normalized, safe result we'd expect:

```
Promise.resolve( p )
  .then(
    function fulfilled(val){
      console.log( val ); // 42
    },
    function rejected(err){
      // never gets here
    }
  );

```

`Promise.resolve(..)` will accept any thenable, and will unwrap it to its non-thenable value. But you get back from `Promise.resolve(..)` a real, genuine Promise in its place, **one that you can trust**. If what you passed in is already a genuine Promise, you just get it right back, so there's no downside at all to filtering through `Promise.resolve(..)` to gain trust.

So let's say we're calling a `foo(..)` utility and we're not sure we can trust its return value to be a well-behaving Promise, but we know it's at least a thenable. `Promise.resolve(..)` will give us a trustable Promise wrapper to chain off of:

```
// don't just do this:
foo( 42 )
  .then( function(v){
    console.log( v );
  } );

// instead, do this:
Promise.resolve( foo( 42 ) )
  .then( function(v){
    console.log( v );
  } );

```

Note: Another beneficial side effect of wrapping `Promise.resolve(..)` around any function's return value (thenable or not) is that it's an easy way to normalize that function call into a well-behaving async task. If `foo(42)` returns an immediate value sometimes, or a Promise other times, `Promise.resolve(foo(42))` makes sure it's always a Promise result. And avoiding Zalgo makes for much better code.

Trust Built

Hopefully the previous discussion now fully "resolves" (pun intended) in your mind why the Promise is trustable, and more importantly, why that trust is so critical in building robust, maintainable software.

Can you write async code in JS without trust? Of course you can. We JS developers have been coding async with nothing but callbacks for nearly two decades.

But once you start questioning just how much you can trust the mechanisms you build upon to actually be predictable and reliable, you start to realize callbacks have a pretty shaky trust foundation.

Promises are a pattern that augments callbacks with trustable semantics, so that the behavior is more reasonable and more reliable. By uninverting the *inversion of control* of callbacks, we place the control with a trustable system (Promises) that was designed specifically to bring sanity to our async.

Chain Flow

We've hinted at this a couple of times already, but Promises are not just a mechanism for a single-step *this-then-that* sort of operation. That's the building block, of course, but it turns out we can string multiple Promises together to represent a sequence of async steps.

The key to making this work is built on two behaviors intrinsic to Promises:

- Every time you call `then(...)` on a Promise, it creates and returns a new Promise, which we can *chain* with.
- Whatever value you return from the `then(...)` call's fulfillment callback (the first parameter) is automatically set as the fulfillment of the *chained* Promise (from the first point).

Let's first illustrate what that means, and *then* we'll derive how that helps us create async sequences of flow control. Consider the following:

```

var p = Promise.resolve( 21 );

var p2 = p.then( function(v){
    console.log( v );      // 21

    // fulfill `p2` with value `42`
    return v * 2;
} );

// chain off `p2`
p2.then( function(v){
    console.log( v );      // 42
} );

```

By returning `v * 2` (i.e., `42`), we fulfill the `p2` promise that the first `then(...)` call created and returned. When `p2`'s `then(...)` call runs, it's receiving the fulfillment from the `return v * 2` statement. Of course, `p2.then(..)` creates yet another promise, which we could have stored in a `p3` variable.

But it's a little annoying to have to create an intermediate variable `p2` (or `p3`, etc.). Thankfully, we can easily just chain these together:

```

var p = Promise.resolve( 21 );

p
.then( function(v){
    console.log( v );      // 21

    // fulfill the chained promise with value `42`
    return v * 2;
} )
// here's the chained promise
.then( function(v){
    console.log( v );      // 42
} );

```

So now the first `then(..)` is the first step in an async sequence, and the second `then(..)` is the second step. This could keep going for as long as you needed it to extend. Just keep chaining off a previous `then(..)` with each automatically created Promise.

But there's something missing here. What if we want step 2 to wait for step 1 to do something asynchronous? We're using an immediate `return` statement, which immediately fulfills the chained promise.

The key to making a Promise sequence truly async capable at every step is to recall how `Promise.resolve(..)` operates when what you pass to it is a Promise or thenable instead of a final value. `Promise.resolve(..)` directly returns a received genuine Promise, or it

unwraps the value of a received thenable -- and keeps going recursively while it keeps unwrapping thenables.

The same sort of unwrapping happens if you `return` a thenable or Promise from the fulfillment (or rejection) handler. Consider:

```
var p = Promise.resolve( 21 );

p.then( function(v){
    console.log( v );      // 21

    // create a promise and return it
    return new Promise( function(resolve,reject){
        // fulfill with value `42`
        resolve( v * 2 );
    } );
} )
.then( function(v){
    console.log( v );      // 42
} );
```

Even though we wrapped `42` up in a promise that we returned, it still got unwrapped and ended up as the resolution of the chained promise, such that the second `then(..)` still received `42`. If we introduce asynchrony to that wrapping promise, everything still nicely works the same:

```
var p = Promise.resolve( 21 );

p.then( function(v){
    console.log( v );      // 21

    // create a promise to return
    return new Promise( function(resolve,reject){
        // introduce asynchrony!
        setTimeout( function(){
            // fulfill with value `42`
            resolve( v * 2 );
        }, 100 );
    } );
} )
.then( function(v){
    // runs after the 100ms delay in the previous step
    console.log( v );      // 42
} );
```

That's incredibly powerful! Now we can construct a sequence of however many async steps we want, and each step can delay the next step (or not!), as necessary.

Of course, the value passing from step to step in these examples is optional. If you don't return an explicit value, an implicit `undefined` is assumed, and the promises still chain together the same way. Each Promise resolution is thus just a signal to proceed to the next step.

To further the chain illustration, let's generalize a delay-Promise creation (without resolution messages) into a utility we can reuse for multiple steps:

```
function delay(time) {
  return new Promise( function(resolve,reject){
    setTimeout( resolve, time );
  } );
}

delay( 100 ) // step 1
.then( function STEP2(){
  console.log( "step 2 (after 100ms)" );
  return delay( 200 );
} )
.then( function STEP3(){
  console.log( "step 3 (after another 200ms)" );
} )
.then( function STEP4(){
  console.log( "step 4 (next Job)" );
  return delay( 50 );
} )
.then( function STEP5(){
  console.log( "step 5 (after another 50ms)" );
} )
...
}
```

Calling `delay(200)` creates a promise that will fulfill in 200ms, and then we return that from the first `then(...)` fulfillment callback, which causes the second `then(...)`'s promise to wait on that 200ms promise.

Note: As described, technically there are two promises in that interchange: the 200ms-delay promise and the chained promise that the second `then(...)` chains from. But you may find it easier to mentally combine these two promises together, because the Promise mechanism automatically merges their states for you. In that respect, you could think of `return delay(200)` as creating a promise that replaces the earlier-returned chained promise.

To be honest, though, sequences of delays with no message passing isn't a terribly useful example of Promise flow control. Let's look at a scenario that's a little more practical.

Instead of timers, let's consider making Ajax requests:

```
// assume an `ajax( {url}, {callback} )` utility

// Promise-aware ajax
function request(url) {
  return new Promise( function(resolve,reject){
    // the `ajax(..)` callback should be our
    // promise's `resolve(..)` function
    ajax( url, resolve );
  } );
}
```

We first define a `request(..)` utility that constructs a promise to represent the completion of the `ajax(..)` call:

```
request( "http://some.url.1/" )
.then( function(response1){
  return request( "http://some.url.2/?v=" + response1 );
} )
.then( function(response2){
  console.log( response2 );
} );
```

Note: Developers commonly encounter situations in which they want to do Promise-aware async flow control with utilities that are not themselves Promise-enabled (like `ajax(..)` here, which expects a callback). Although the native ES6 `Promise` mechanism doesn't automatically solve this pattern for us, practically all Promise libraries do. They usually call this process "lifting" or "promisifying" or some variation thereof. We'll come back to this technique later.

Using the Promise-returning `request(..)`, we create the first step in our chain implicitly by calling it with the first URL, and chain off that returned promise with the first `then(..)`.

Once `response1` comes back, we use that value to construct a second URL, and make a second `request(..)` call. That second `request(..)` promise is `return`ed so that the third step in our async flow control waits for that Ajax call to complete. Finally, we print `response2` once it returns.

The Promise chain we construct is not only a flow control that expresses a multistep async sequence, but it also acts as a message channel to propagate messages from step to step.

What if something went wrong in one of the steps of the Promise chain? An error/exception is on a per-Promise basis, which means it's possible to catch such an error at any point in the chain, and that catching acts to sort of "reset" the chain back to normal operation at that point:

```
// step 1:  
request( "http://some.url.1/" )  
  
// step 2:  
.then( function(response1){  
    foo.bar(); // undefined, error!  
  
    // never gets here  
    return request( "http://some.url.2/?v=" + response1 );  
} )  
  
// step 3:  
.then(  
    function fulfilled(response2){  
        // never gets here  
    },  
    // rejection handler to catch the error  
    function rejected(err){  
        console.log( err ); // `TypeError` from `foo.bar()`'s error  
        return 42;  
    }  
)  
  
// step 4:  
.then( function(msg){  
    console.log( msg ); // 42  
} );
```

When the error occurs in step 2, the rejection handler in step 3 catches it. The return value (`42` in this snippet), if any, from that rejection handler fulfills the promise for the next step (4), such that the chain is now back in a fulfillment state.

Note: As we discussed earlier, when returning a promise from a fulfillment handler, it's unwrapped and can delay the next step. That's also true for returning promises from rejection handlers, such that if the `return 42` in step 3 instead returned a promise, that promise could delay step 4. A thrown exception inside either the fulfillment or rejection handler of a `then(...)` call causes the next (chained) promise to be immediately rejected with that exception.

If you call `then(...)` on a promise, and you only pass a fulfillment handler to it, an assumed rejection handler is substituted:

```

var p = new Promise( function(resolve,reject){
    reject( "Oops" );
} );

var p2 = p.then(
    function fulfilled(){
        // never gets here
    }
    // assumed rejection handler, if omitted or
    // any other non-function value passed
    // function(err) {
    //     throw err;
    // }
);

);

```

As you can see, the assumed rejection handler simply rethrows the error, which ends up forcing `p2` (the chained promise) to reject with the same error reason. In essence, this allows the error to continue propagating along a Promise chain until an explicitly defined rejection handler is encountered.

Note: We'll cover more details of error handling with Promises a little later, because there are other nuanced details to be concerned about.

If a proper valid function is not passed as the fulfillment handler parameter to `then(..)`, there's also a default handler substituted:

```

var p = Promise.resolve( 42 );

p.then(
    // assumed fulfillment handler, if omitted or
    // any other non-function value passed
    // function(v) {
    //     return v;
    // }
    null,
    function rejected(err){
        // never gets here
    }
);

```

As you can see, the default fulfillment handler simply passes whatever value it receives along to the next step (Promise).

Note: The `then(null,function(err){ .. })` pattern -- only handling rejections (if any) but letting fulfillments pass through -- has a shortcut in the API: `catch(function(err){ .. })`. We'll cover `catch(..)` more fully in the next section.

Let's review briefly the intrinsic behaviors of Promises that enable chaining flow control:

- A `then(...)` call against one Promise automatically produces a new Promise to return from the call.
- Inside the fulfillment/rejection handlers, if you return a value or an exception is thrown, the new returned (chainable) Promise is resolved accordingly.
- If the fulfillment or rejection handler returns a Promise, it is unwrapped, so that whatever its resolution is will become the resolution of the chained Promise returned from the current `then(...)`.

While chaining flow control is helpful, it's probably most accurate to think of it as a side benefit of how Promises compose (combine) together, rather than the main intent. As we've discussed in detail several times already, Promises normalize asynchrony and encapsulate time-dependent value state, and *that* is what lets us chain them together in this useful way.

Certainly, the sequential expressiveness of the chain (this-then-this-then-this...) is a big improvement over the tangled mess of callbacks as we identified in Chapter 2. But there's still a fair amount of boilerplate (`then(...)` and `function(){ ... }`) to wade through. In the next chapter, we'll see a significantly nicer pattern for sequential flow control expressivity, with generators.

Terminology: Resolve, Fulfill, and Reject

There's some slight confusion around the terms "resolve," "fulfill," and "reject" that we need to clear up, before you get too much deeper into learning about Promises. Let's first consider the `Promise(..)` constructor:

```
var p = new Promise( function(X,Y){  
    // X() for fulfillment  
    // Y() for rejection  
} );
```

As you can see, two callbacks (here labeled `x` and `y`) are provided. The first is *usually* used to mark the Promise as fulfilled, and the second *always* marks the Promise as rejected. But what's the "usually" about, and what does that imply about accurately naming those parameters?

Ultimately, it's just your user code and the identifier names aren't interpreted by the engine to mean anything, so it doesn't *technically* matter; `foo(..)` and `bar(..)` are equally functional. But the words you use can affect not only how you are thinking about the code, but how other developers on your team will think about it. Thinking wrongly about carefully orchestrated async code is almost surely going to be worse than the spaghetti-callback alternatives.

So it actually does kind of matter what you call them.

The second parameter is easy to decide. Almost all literature uses `reject(..)` as its name, and because that's exactly (and only!) what it does, that's a very good choice for the name. I'd strongly recommend you always use `reject(..)`.

But there's a little more ambiguity around the first parameter, which in Promise literature is often labeled `resolve(..)`. That word is obviously related to "resolution," which is what's used across the literature (including this book) to describe setting a final value/state to a Promise. We've already used "resolve the Promise" several times to mean either fulfilling or rejecting the Promise.

But if this parameter seems to be used to specifically fulfill the Promise, why shouldn't we call it `fulfill(..)` instead of `resolve(..)` to be more accurate? To answer that question, let's also take a look at two of the `Promise` API methods:

```
var fulfilledPr = Promise.resolve( 42 );
var rejectedPr = Promise.reject( "Oops" );
```

`Promise.resolve(..)` creates a Promise that's resolved to the value given to it. In this example, `42` is a normal, non-Promise, non-thenable value, so the fulfilled promise `fulfilledPr` is created for the value `42`. `Promise.reject("Oops")` creates the rejected promise `rejectedPr` for the reason `"Oops"`.

Let's now illustrate why the word "resolve" (such as in `Promise.resolve(..)`) is unambiguous and indeed more accurate, if used explicitly in a context that could result in either fulfillment or rejection:

```
var rejectedTh = {
  then: function(resolved, rejected) {
    rejected( "Oops" );
  }
};

var rejectedPr = Promise.resolve( rejectedTh );
```

As we discussed earlier in this chapter, `Promise.resolve(..)` will return a received genuine Promise directly, or unwrap a received thenable. If that thenable unwrapping reveals a rejected state, the Promise returned from `Promise.resolve(..)` is in fact in that same rejected state.

So `Promise.resolve(..)` is a good, accurate name for the API method, because it can actually result in either fulfillment or rejection.

The first callback parameter of the `Promise(..)` constructor will unwrap either a thenable (identically to `Promise.resolve(..)`) or a genuine Promise:

```
var rejectedPr = new Promise( function(resolve,reject){
    // resolve this promise with a rejected promise
    resolve( Promise.reject( "Oops" ) );
} );

rejectedPr.then(
    function fulfilled(){
        // never gets here
    },
    function rejected(err){
        console.log( err );      // "Oops"
    }
);
```

It should be clear now that `resolve(..)` is the appropriate name for the first callback parameter of the `Promise(..)` constructor.

Warning: The previously mentioned `reject(..)` does **not** do the unwrapping that `resolve(..)` does. If you pass a Promise/thenable value to `reject(..)`, that untouched value will be set as the rejection reason. A subsequent rejection handler would receive the actual Promise/thenable you passed to `reject(..)`, not its underlying immediate value.

But now let's turn our attention to the callbacks provided to `then(..)`. What should they be called (both in literature and in code)? I would suggest `fulfilled(..)` and `rejected(..)`:

```
function fulfilled(msg) {
    console.log( msg );
}

function rejected(err) {
    console.error( err );
}

p.then(
    fulfilled,
    rejected
);
```

In the case of the first parameter to `then(..)`, it's unambiguously always the fulfillment case, so there's no need for the duality of "resolve" terminology. As a side note, the ES6 specification uses `onFulfilled(..)` and `onRejected(..)` to label these two callbacks, so they are accurate terms.

Error Handling

We've already seen several examples of how Promise rejection -- either intentional through calling `reject(..)` or accidental through JS exceptions -- allows saner error handling in asynchronous programming. Let's circle back though and be explicit about some of the details that we glossed over.

The most natural form of error handling for most developers is the synchronous `try..catch` construct. Unfortunately, it's synchronous-only, so it fails to help in async code patterns:

```
function foo() {
  setTimeout( function(){
    baz.bar();
  }, 100 );
}

try {
  foo();
  // later throws global error from `baz.bar()`
}
catch (err) {
  // never gets here
}
```

`try..catch` would certainly be nice to have, but it doesn't work across async operations. That is, unless there's some additional environmental support, which we'll come back to with generators in Chapter 4.

In callbacks, some standards have emerged for patterned error handling, most notably the "error-first callback" style:

```

function foo(cb) {
  setTimeout( function(){
    try {
      var x = baz.bar();
      cb( null, x ); // success!
    }
    catch (err) {
      cb( err );
    }
  }, 100 );
}

foo( function(err, val){
  if (err) {
    console.error( err ); // bummer :(
  }
  else {
    console.log( val );
  }
} );

```

Note: The `try..catch` here works only from the perspective that the `baz.bar()` call will either succeed or fail immediately, synchronously. If `baz.bar()` was itself its own async completing function, any async errors inside it would not be catchable.

The callback we pass to `foo(..)` expects to receive a signal of an error by the reserved first parameter `err`. If present, error is assumed. If not, success is assumed.

This sort of error handling is technically *async capable*, but it doesn't compose well at all. Multiple levels of error-first callbacks woven together with these ubiquitous `if` statement checks inevitably will lead you to the perils of callback hell (see Chapter 2).

So we come back to error handling in Promises, with the rejection handler passed to `then(..)`. Promises don't use the popular "error-first callback" design style, but instead use "split callbacks" style; there's one callback for fulfillment and one for rejection:

```

var p = Promise.reject( "Oops" );

p.then(
  function fulfilled(){
    // never gets here
  },
  function rejected(err){
    console.log( err ); // "Oops"
  }
);

```

While this pattern of error handling makes fine sense on the surface, the nuances of Promise error handling are often a fair bit more difficult to fully grasp.

Consider:

```
var p = Promise.resolve( 42 );  
  
p.then(  
  function fulfilled(msg){  
    // numbers don't have string functions,  
    // so will throw an error  
    console.log( msg.toLowerCase() );  
  },  
  function rejected(err){  
    // never gets here  
  }  
);
```

If the `msg.toLowerCase()` legitimately throws an error (it does!), why doesn't our error handler get notified? As we explained earlier, it's because *that* error handler is for the `p` promise, which has already been fulfilled with value `42`. The `p` promise is immutable, so the only promise that can be notified of the error is the one returned from `p.then(...)`, which in this case we don't capture.

That should paint a clear picture of why error handling with Promises is error-prone (pun intended). It's far too easy to have errors swallowed, as this is very rarely what you'd intend.

Warning: If you use the Promise API in an invalid way and an error occurs that prevents proper Promise construction, the result will be an immediately thrown exception, **not a rejected Promise**. Some examples of incorrect usage that fail Promise construction: `new Promise(null)`, `Promise.all()`, `Promise.race(42)`, and so on. You can't get a rejected Promise if you don't use the Promise API validly enough to actually construct a Promise in the first place!

Pit of Despair

Jeff Atwood noted years ago: programming languages are often set up in such a way that by default, developers fall into the "pit of despair" (<http://blog.codinghorror.com/falling-into-the-pit-of-success/>) -- where accidents are punished -- and that you have to try harder to get it right. He implored us to instead create a "pit of success," where by default you fall into expected (successful) action, and thus would have to try hard to fail.

Promise error handling is unquestionably "pit of despair" design. By default, it assumes that you want any error to be swallowed by the Promise state, and if you forget to observe that state, the error silently languishes/dies in obscurity -- usually despair.

To avoid losing an error to the silence of a forgotten/discarded Promise, some developers have claimed that a "best practice" for Promise chains is to always end your chain with a final `catch(..)`, like:

```
var p = Promise.resolve( 42 );

p.then(
  function fulfilled(msg){
    // numbers don't have string functions,
    // so will throw an error
    console.log( msg.toLowerCase() );
  }
)
.catch( handleErrors );
```

Because we didn't pass a rejection handler to the `then(..)`, the default handler was substituted, which simply propagates the error to the next promise in the chain. As such, both errors that come into `p`, and errors that come *after* `p` in its resolution (like the `msg.toLowerCase()` one) will filter down to the final `handleErrors(..)`.

Problem solved, right? Not so fast!

What happens if `handleErrors(..)` itself also has an error in it? Who catches that? There's still yet another unattended promise: the one `catch(..)` returns, which we don't capture and don't register a rejection handler for.

You can't just stick another `catch(..)` on the end of that chain, because it too could fail. The last step in any Promise chain, whatever it is, always has the possibility, even decreasingly so, of dangling with an uncaught error stuck inside an unobserved Promise.

Sound like an impossible conundrum yet?

Uncaught Handling

It's not exactly an easy problem to solve completely. There are other ways to approach it which many would say are *better*.

Some Promise libraries have added methods for registering something like a "global unhandled rejection" handler, which would be called instead of a globally thrown error. But their solution for how to identify an error as "uncaught" is to have an arbitrary-length timer, say 3 seconds, running from time of rejection. If a Promise is rejected but no error handler is registered before the timer fires, then it's assumed that you won't ever be registering a handler, so it's "uncaught."

In practice, this has worked well for many libraries, as most usage patterns don't typically call for significant delay between Promise rejection and observation of that rejection. But this pattern is troublesome because 3 seconds is so arbitrary (even if empirical), and also because there are indeed some cases where you want a Promise to hold on to its rejectedness for some indefinite period of time, and you don't really want to have your "uncaught" handler called for all those false positives (not-yet-handled "uncaught errors").

Another more common suggestion is that Promises should have a `done(..)` added to them, which essentially marks the Promise chain as "done." `done(..)` doesn't create and return a Promise, so the callbacks passed to `done(..)` are obviously not wired up to report problems to a chained Promise that doesn't exist.

So what happens instead? It's treated as you might usually expect in uncaught error conditions: any exception inside a `done(..)` rejection handler would be thrown as a global uncaught error (in the developer console, basically):

```
var p = Promise.resolve( 42 );

p.then(
  function fulfilled(msg){
    // numbers don't have string functions,
    // so will throw an error
    console.log( msg.toLowerCase() );
  }
)
.done( null, handleErrors );

// if `handleErrors(..)` caused its own exception, it would
// be thrown globally here
```

This might sound more attractive than the never-ending chain or the arbitrary timeouts. But the biggest problem is that it's not part of the ES6 standard, so no matter how good it sounds, at best it's a lot longer way off from being a reliable and ubiquitous solution.

Are we just stuck, then? Not entirely.

Browsers have a unique capability that our code does not have: they can track and know for sure when any object gets thrown away and garbage collected. So, browsers can track Promise objects, and whenever they get garbage collected, if they have a rejection in them, the browser knows for sure this was a legitimate "uncaught error," and can thus confidently know it should report it to the developer console.

Note: At the time of this writing, both Chrome and Firefox have early attempts at that sort of "uncaught rejection" capability, though support is incomplete at best.

However, if a Promise doesn't get garbage collected -- it's exceedingly easy for that to accidentally happen through lots of different coding patterns -- the browser's garbage collection sniffing won't help you know and diagnose that you have a silently rejected Promise laying around.

Is there any other alternative? Yes.

Pit of Success

The following is just theoretical, how Promises *could* be someday changed to behave. I believe it would be far superior to what we currently have. And I think this change would be possible even post-ES6 because I don't think it would break web compatibility with ES6 Promises. Moreover, it can be polyfilled/prollyfilled in, if you're careful. Let's take a look:

- Promises could default to reporting (to the developer console) any rejection, on the next Job or event loop tick, if at that exact moment no error handler has been registered for the Promise.
- For the cases where you want a rejected Promise to hold onto its rejected state for an indefinite amount of time before observing, you could call `defer()`, which suppresses automatic error reporting on that Promise.

If a Promise is rejected, it defaults to noisily reporting that fact to the developer console (instead of defaulting to silence). You can opt out of that reporting either implicitly (by registering an error handler before rejection), or explicitly (with `defer()`). In either case, *you* control the false positives.

Consider:

```
var p = Promise.reject( "Oops" ).defer();

// `foo(..)` is Promise-aware
foo( 42 )
.then(
  function fulfilled(){
    return p;
  },
  function rejected(err){
    // handle `foo(..)` error
  }
);
...
...
```

When we create `p`, we know we're going to wait a while to use/observe its rejection, so we call `defer()` -- thus no global reporting. `defer()` simply returns the same promise, for chaining purposes.

The promise returned from `foo(...)` gets an error handler attached *right away*, so it's implicitly opted out and no global reporting for it occurs either.

But the promise returned from the `then(...)` call has no `defer()` or error handler attached, so if it rejects (from inside either resolution handler), then *it* will be reported to the developer console as an uncaught error.

This design is a pit of success. By default, all errors are either handled or reported -- what almost all developers in almost all cases would expect. You either have to register a handler or you have to intentionally opt out, and indicate you intend to defer error handling until *later*; you're opting for the extra responsibility in just that specific case.

The only real danger in this approach is if you `defer()` a Promise but then fail to actually ever observe/handle its rejection.

But you had to intentionally call `defer()` to opt into that pit of despair -- the default was the pit of success -- so there's not much else we could do to save you from your own mistakes.

I think there's still hope for Promise error handling (post-ES6). I hope the powers that be will rethink the situation and consider this alternative. In the meantime, you can implement this yourself (a challenging exercise for the reader!), or use a *smarter* Promise library that does so for you!

Note: This exact model for error handling/reporting is implemented in my `asynquence` Promise abstraction library, which will be discussed in Appendix A of this book.

Promise Patterns

We've already implicitly seen the sequence pattern with Promise chains (this-then-this-then-that flow control) but there are lots of variations on asynchronous patterns that we can build as abstractions on top of Promises. These patterns serve to simplify the expression of async flow control -- which helps make our code more reasonable and more maintainable -- even in the most complex parts of our programs.

Two such patterns are codified directly into the native ES6 `Promise` implementation, so we get them for free, to use as building blocks for other patterns.

Promise.all([..])

In an async sequence (Promise chain), only one async task is being coordinated at any given moment -- step 2 strictly follows step 1, and step 3 strictly follows step 2. But what about doing two or more steps concurrently (aka "in parallel")?

In classic programming terminology, a "gate" is a mechanism that waits on two or more parallel/concurrent tasks to complete before continuing. It doesn't matter what order they finish in, just that all of them have to complete for the gate to open and let the flow control through.

In the Promise API, we call this pattern `all([...])`.

Say you wanted to make two Ajax requests at the same time, and wait for both to finish, regardless of their order, before making a third Ajax request. Consider:

```
// `request(..)` is a Promise-aware Ajax utility,
// like we defined earlier in the chapter

var p1 = request( "http://some.url.1/" );
var p2 = request( "http://some.url.2/" );

Promise.all( [p1,p2] )
.then( function(msgs){
    // both `p1` and `p2` fulfill and pass in
    // their messages here
    return request(
        "http://some.url.3/?v=" + msgs.join(",")
    );
})
.then( function(msg){
    console.log( msg );
} );
```

`Promise.all([...])` expects a single argument, an `array`, consisting generally of Promise instances. The promise returned from the `Promise.all([...])` call will receive a fulfillment message (`msgs` in this snippet) that is an `array` of all the fulfillment messages from the passed in promises, in the same order as specified (regardless of fulfillment order).

Note: Technically, the `array` of values passed into `Promise.all([...])` can include Promises, thenables, or even immediate values. Each value in the list is essentially passed through `Promise.resolve(..)` to make sure it's a genuine Promise to be waited on, so an immediate value will just be normalized into a Promise for that value. If the `array` is empty, the main Promise is immediately fulfilled.

The main promise returned from `Promise.all([...])` will only be fulfilled if and when all its constituent promises are fulfilled. If any one of those promises instead is rejected, the main `Promise.all([...])` promise is immediately rejected, discarding all results from any other promises.

Remember to always attach a rejection/error handler to every promise, even and especially the one that comes back from `Promise.all([...])`.

Promise.race([..])

While `Promise.all([..])` coordinates multiple Promises concurrently and assumes all are needed for fulfillment, sometimes you only want to respond to the "first Promise to cross the finish line," letting the other Promises fall away.

This pattern is classically called a "latch," but in Promises it's called a "race."

Warning: While the metaphor of "only the first across the finish line wins" fits the behavior well, unfortunately "race" is kind of a loaded term, because "race conditions" are generally taken as bugs in programs (see Chapter 1). Don't confuse `Promise.race([..])` with "race condition."

`Promise.race([..])` also expects a single `array` argument, containing one or more Promises, thenables, or immediate values. It doesn't make much practical sense to have a race with immediate values, because the first one listed will obviously win -- like a foot race where one runner starts at the finish line!

Similar to `Promise.all([..])`, `Promise.race([..])` will fulfill if and when any Promise resolution is a fulfillment, and it will reject if and when any Promise resolution is a rejection.

Warning: A "race" requires at least one "runner," so if you pass an empty `array`, instead of immediately resolving, the main `race([..])` Promise will never resolve. This is a footgun! ES6 should have specified that it either fulfills, rejects, or just throws some sort of synchronous error. Unfortunately, because of precedence in Promise libraries predating ES6 `Promise`, they had to leave this gotcha in there, so be careful never to send in an empty `array`.

Let's revisit our previous concurrent Ajax example, but in the context of a race between `p1` and `p2`:

```
// `request(..)` is a Promise-aware Ajax utility,
// like we defined earlier in the chapter

var p1 = request( "http://some.url.1/" );
var p2 = request( "http://some.url.2/" );

Promise.race( [p1,p2] )
.then( function(msg){
    // either `p1` or `p2` will win the race
    return request(
        "http://some.url.3/?v=" + msg
    );
} )
.then( function(msg){
    console.log( msg );
} );
```

Because only one promise wins, the fulfillment value is a single message, not an `array` as it was for `Promise.all([...])`.

Timeout Race

We saw this example earlier, illustrating how `Promise.race([...])` can be used to express the "promise timeout" pattern:

```
// `foo()` is a Promise-aware function

// `timeoutPromise(..)`, defined earlier, returns
// a Promise that rejects after a specified delay

// setup a timeout for `foo()`
Promise.race([
    foo(),                      // attempt `foo()`
    timeoutPromise( 3000 )       // give it 3 seconds
])
.then(
    function(){
        // `foo(..)` fulfilled in time!
    },
    function(err){
        // either `foo()` rejected, or it just
        // didn't finish in time, so inspect
        // `err` to know which
    }
);
```

This timeout pattern works well in most cases. But there are some nuances to consider, and frankly they apply to both `Promise.race([...])` and `Promise.all([...])` equally.

"Finally"

The key question to ask is, "What happens to the promises that get discarded/ignored?" We're not asking that question from the performance perspective -- they would typically end up garbage collection eligible -- but from the behavioral perspective (side effects, etc.). Promises cannot be canceled -- and shouldn't be as that would destroy the external immutability trust discussed in the "Promise Uncancelable" section later in this chapter -- so they can only be silently ignored.

But what if `foo()` in the previous example is reserving some sort of resource for usage, but the timeout fires first and causes that promise to be ignored? Is there anything in this pattern that proactively frees the reserved resource after the timeout, or otherwise cancels any side effects it may have had? What if all you wanted was to log the fact that `foo()` timed out?

Some developers have proposed that Promises need a `finally(...)` callback registration, which is always called when a Promise resolves, and allows you to specify any cleanup that may be necessary. This doesn't exist in the specification at the moment, but it may come in ES7+. We'll have to wait and see.

It might look like:

```
var p = Promise.resolve( 42 );

p.then( something )
  .finally( cleanup )
  .then( another )
  .finally( cleanup );
```

Note: In various Promise libraries, `finally(...)` still creates and returns a new Promise (to keep the chain going). If the `cleanup(...)` function were to return a Promise, it would be linked into the chain, which means you could still have the unhandled rejection issues we discussed earlier.

In the meantime, we could make a static helper utility that lets us observe (without interfering) the resolution of a Promise:

```
// polyfill-safe guard check
if (!Promise.observe) {
  Promise.observe = function(pr,cb) {
    // side-observe `pr`'s resolution
    pr.then(
      function fulfilled(msg){
        // schedule callback async (as Job)
        Promise.resolve( msg ).then( cb );
      },
      function rejected(err){
        // schedule callback async (as Job)
        Promise.resolve( err ).then( cb );
      }
    );
    // return original promise
    return pr;
  };
}
```

Here's how we'd use it in the timeout example from before:

```
Promise.race([
  Promise.observe(
    foo(), // attempt `foo()`
    function cleanup(msg){
      // clean up after `foo()`, even if it
      // didn't finish before the timeout
    }
  ),
  timeoutPromise( 3000 ) // give it 3 seconds
])
```

This `Promise.observe(..)` helper is just an illustration of how you could observe the completions of Promises without interfering with them. Other Promise libraries have their own solutions. Regardless of how you do it, you'll likely have places where you want to make sure your Promises aren't *just* silently ignored by accident.

Variations on `all([..])` and `race([..])`

While native ES6 Promises come with built-in `Promise.all([..])` and `Promise.race([..])`, there are several other commonly used patterns with variations on those semantics:

- `none([..])` is like `all([..])`, but fulfillments and rejections are transposed. All Promises need to be rejected -- rejections become the fulfillment values and vice versa.
- `any([..])` is like `all([..])`, but it ignores any rejections, so only one needs to

fulfill instead of *all* of them.

- `first([...])` is like a race with `any([...])`, which is that it ignores any rejections and fulfills as soon as the first Promise fulfills.
- `last([...])` is like `first([...])`, but only the latest fulfillment wins.

Some Promise abstraction libraries provide these, but you could also define them yourself using the mechanics of Promises, `race([...])` and `all([...])`.

For example, here's how we could define `first([...])`:

```
// polyfill-safe guard check
if (!Promise.first) {
  Promise.first = function(prs) {
    return new Promise( function(resolve,reject){
      // loop through all promises
      prs.forEach( function(pr){
        // normalize the value
        Promise.resolve( pr )
        // whichever one fulfills first wins, and
        // gets to resolve the main promise
        .then( resolve );
      } );
    } );
  };
}
```

Note: This implementation of `first(...)` does not reject if all its promises reject; it simply hangs, much like a `Promise.race([])` does. If desired, you could add additional logic to track each promise rejection and if all reject, call `reject()` on the main promise. We'll leave that as an exercise for the reader.

Concurrent Iterations

Sometimes you want to iterate over a list of Promises and perform some task against all of them, much like you can do with synchronous `array s` (e.g., `forEach(...)`, `map(...)`, `some(...)`, and `every(...)`). If the task to perform against each Promise is fundamentally synchronous, these work fine, just as we used `forEach(...)` in the previous snippet.

But if the tasks are fundamentally asynchronous, or can/should otherwise be performed concurrently, you can use `async` versions of these utilities as provided by many libraries.

For example, let's consider an asynchronous `map(...)` utility that takes an `array` of values (could be Promises or anything else), plus a function (`task`) to perform against each.

`map(...)` itself returns a promise whose fulfillment value is an `array` that holds (in the same mapping order) the `async` fulfillment value from each task:

```

if (!Promise.map) {
  Promise.map = function(vals,cb) {
    // new promise that waits for all mapped promises
    return Promise.all(
      // note: regular array `map(..)` , turns
      // the array of values into an array of
      // promises
      vals.map( function(val){
        // replace `val` with a new promise that
        // resolves after `val` is async mapped
        return new Promise( function(resolve){
          cb( val, resolve );
        } );
      } )
    );
  };
}

```

Note: In this implementation of `map(..)`, you can't signal async rejection, but if a synchronous exception/error occurs inside of the mapping callback (`cb(..)`), the main `Promise.map(..)` returned promise would reject.

Let's illustrate using `map(..)` with a list of Promises (instead of simple values):

```

var p1 = Promise.resolve( 21 );
var p2 = Promise.resolve( 42 );
var p3 = Promise.reject( "Oops" );

// double values in list even if they're in
// Promises
Promise.map( [p1,p2,p3], function(pr,done){
  // make sure the item itself is a Promise
  Promise.resolve( pr )
  .then(
    // extract value as `v`
    function(v){
      // map fulfillment `v` to new value
      done( v * 2 );
    },
    // or, map to promise rejection message
    done
  );
} )
.then( function(vals){
  console.log( vals ); // [42,84,"Oops"]
} );

```

Promise API Recap

Let's review the ES6 `Promise` API that we've already seen unfold in bits and pieces throughout this chapter.

Note: The following API is native only as of ES6, but there are specification-compliant polyfills (not just extended Promise libraries) which can define `Promise` and all its associated behavior so that you can use native Promises even in pre-ES6 browsers. One such polyfill is "Native Promise Only" (<http://github.com/getify/native-promise-only>), which I wrote!

new Promise(..) Constructor

The *revealing constructor* `Promise(..)` must be used with `new`, and must be provided a function callback that is synchronously/immediately called. This function is passed two function callbacks that act as resolution capabilities for the promise. We commonly label these `resolve(..)` and `reject(..)`:

```
var p = new Promise( function(resolve,reject){
    // `resolve(..)` to resolve/fulfill the promise
    // `reject(..)` to reject the promise
} );
```

`reject(..)` simply rejects the promise, but `resolve(..)` can either fulfill the promise or reject it, depending on what it's passed. If `resolve(..)` is passed an immediate, non-Promise, non-thenable value, then the promise is fulfilled with that value.

But if `resolve(..)` is passed a genuine Promise or thenable value, that value is unwrapped recursively, and whatever its final resolution/state is will be adopted by the promise.

Promise.resolve(..) and Promise.reject(..)

A shortcut for creating an already-rejected Promise is `Promise.reject(..)`, so these two promises are equivalent:

```
var p1 = new Promise( function(resolve,reject){
    reject( "Oops" );
} );

var p2 = Promise.reject( "Oops" );
```

`Promise.resolve(..)` is usually used to create an already-fulfilled Promise in a similar way to `Promise.reject(..)`. However, `Promise.resolve(..)` also unwraps thenable values (as discussed several times already). In that case, the Promise returned adopts the final resolution of the thenable you passed in, which could either be fulfillment or rejection:

```

var fulfilledTh = {
    then: function(cb) { cb( 42 ); }
};

var rejectedTh = {
    then: function(cb,errCb) {
        errCb( "Oops" );
    }
};

var p1 = Promise.resolve( fulfilledTh );
var p2 = Promise.resolve( rejectedTh );

// `p1` will be a fulfilled promise
// `p2` will be a rejected promise

```

And remember, `Promise.resolve(..)` doesn't do anything if what you pass is already a genuine Promise; it just returns the value directly. So there's no overhead to calling `Promise.resolve(..)` on values that you don't know the nature of, if one happens to already be a genuine Promise.

then(..) and catch(..)

Each Promise instance (**not** the `Promise` API namespace) has `then(..)` and `catch(..)` methods, which allow registering of fulfillment and rejection handlers for the Promise. Once the Promise is resolved, one or the other of these handlers will be called, but not both, and it will always be called asynchronously (see "Jobs" in Chapter 1).

`then(..)` takes one or two parameters, the first for the fulfillment callback, and the second for the rejection callback. If either is omitted or is otherwise passed as a non-function value, a default callback is substituted respectively. The default fulfillment callback simply passes the message along, while the default rejection callback simply rethrows (propagates) the error reason it receives.

`catch(..)` takes only the rejection callback as a parameter, and automatically substitutes the default fulfillment callback, as just discussed. In other words, it's equivalent to `then(null,..)`:

```

p.then( fulfilled );

p.then( fulfilled, rejected );

p.catch( rejected ); // or `p.then( null, rejected )`

```

`then(...)` and `catch(...)` also create and return a new promise, which can be used to express Promise chain flow control. If the fulfillment or rejection callbacks have an exception thrown, the returned promise is rejected. If either callback returns an immediate, non-Promise, non-thenable value, that value is set as the fulfillment for the returned promise. If the fulfillment handler specifically returns a promise or thenable value, that value is unwrapped and becomes the resolution of the returned promise.

Promise.all([..]) and Promise.race([..])

The static helpers `Promise.all([..])` and `Promise.race([..])` on the ES6 `Promise` API both create a Promise as their return value. The resolution of that promise is controlled entirely by the array of promises that you pass in.

For `Promise.all([..])`, all the promises you pass in must fulfill for the returned promise to fulfill. If any promise is rejected, the main returned promise is immediately rejected, too (discarding the results of any of the other promises). For fulfillment, you receive an `array` of all the passed in promises' fulfillment values. For rejection, you receive just the first promise rejection reason value. This pattern is classically called a "gate": all must arrive before the gate opens.

For `Promise.race([..])`, only the first promise to resolve (fulfillment or rejection) "wins," and whatever that resolution is becomes the resolution of the returned promise. This pattern is classically called a "latch": first one to open the latch gets through. Consider:

```
var p1 = Promise.resolve( 42 );
var p2 = Promise.resolve( "Hello World" );
var p3 = Promise.reject( "Oops" );

Promise.race( [p1,p2,p3] )
.then( function(msg){
    console.log( msg );           // 42
} );

Promise.all( [p1,p2,p3] )
.catch( function(err){
    console.error( err );        // "Oops"
} );

Promise.all( [p1,p2] )
.then( function(msgs){
    console.log( msgs );         // [42,"Hello World"]
} );
```

Warning: Be careful! If an empty `array` is passed to `Promise.all([..])`, it will fulfill immediately, but `Promise.race([..])` will hang forever and never resolve.

The ES6 `Promise` API is pretty simple and straightforward. It's at least good enough to serve the most basic of async cases, and is a good place to start when rearranging your code from callback hell to something better.

But there's a whole lot of async sophistication that apps often demand which Promises themselves will be limited in addressing. In the next section, we'll dive into those limitations as motivations for the benefit of Promise libraries.

Promise Limitations

Many of the details we'll discuss in this section have already been alluded to in this chapter, but we'll just make sure to review these limitations specifically.

Sequence Error Handling

We covered Promise-flavored error handling in detail earlier in this chapter. The limitations of how Promises are designed -- how they chain, specifically -- creates a very easy pitfall where an error in a Promise chain can be silently ignored accidentally.

But there's something else to consider with Promise errors. Because a Promise chain is nothing more than its constituent Promises wired together, there's no entity to refer to the entire chain as a single *thing*, which means there's no external way to observe any errors that may occur.

If you construct a Promise chain that has no error handling in it, any error anywhere in the chain will propagate indefinitely down the chain, until observed (by registering a rejection handler at some step). So, in that specific case, having a reference to the *last* promise in the chain is enough (`p` in the following snippet), because you can register a rejection handler there, and it will be notified of any propagated errors:

```
// `foo(..)`, `STEP2(..)` and `STEP3(..)` are
// all promise-aware utilities

var p = foo( 42 )
  .then( STEP2 )
  .then( STEP3 );
```

Although it may seem sneakily confusing, `p` here doesn't point to the first promise in the chain (the one from the `foo(42)` call), but instead from the last promise, the one that comes from the `then(STEP3)` call.

Also, no step in the promise chain is observably doing its own error handling. That means that you could then register a rejection error handler on `p`, and it would be notified if any errors occur anywhere in the chain:

```
p.catch( handleErrors );
```

But if any step of the chain in fact does its own error handling (perhaps hidden/abstracted away from what you can see), your `handleErrors(..)` won't be notified. This may be what you want -- it was, after all, a "handled rejection" -- but it also may *not* be what you want. The complete lack of ability to be notified (of "already handled" rejection errors) is a limitation that restricts capabilities in some use cases.

It's basically the same limitation that exists with a `try..catch` that can catch an exception and simply swallow it. So this isn't a limitation **unique to Promises**, but it *is* something we might wish to have a workaround for.

Unfortunately, many times there is no reference kept for the intermediate steps in a Promise-chain sequence, so without such references, you cannot attach error handlers to reliably observe the errors.

Single Value

Promises by definition only have a single fulfillment value or a single rejection reason. In simple examples, this isn't that big of a deal, but in more sophisticated scenarios, you may find this limiting.

The typical advice is to construct a values wrapper (such as an `object` or `array`) to contain these multiple messages. This solution works, but it can be quite awkward and tedious to wrap and unwrap your messages with every single step of your Promise chain.

Splitting Values

Sometimes you can take this as a signal that you could/should decompose the problem into two or more Promises.

Imagine you have a utility `foo(..)` that produces two values (`x` and `y`) asynchronously:

```

function getY(x) {
  return new Promise( function(resolve,reject){
    setTimeout( function(){
      resolve( (3 * x) - 1 );
    }, 100 );
  } );
}

function foo(bar,baz) {
  var x = bar * baz;

  return getY( x )
    .then( function(y){
      // wrap both values into container
      return [x,y];
    } );
}

foo( 10, 20 )
.then( function(msgs){
  var x = msgs[0];
  var y = msgs[1];

  console.log( x, y ); // 200 599
} );

```

First, let's rearrange what `foo(..)` returns so that we don't have to wrap `x` and `y` into a single `array` value to transport through one Promise. Instead, we can wrap each value into its own promise:

```

function foo(bar,baz) {
  var x = bar * baz;

  // return both promises
  return [
    Promise.resolve( x ),
    getY( x )
  ];
}

Promise.all(
  foo( 10, 20 )
)
.then( function(msgs){
  var x = msgs[0];
  var y = msgs[1];

  console.log( x, y );
} );

```

Is an `array` of promises really better than an `array` of values passed through a single promise? Syntactically, it's not much of an improvement.

But this approach more closely embraces the Promise design theory. It's now easier in the future to refactor to split the calculation of `x` and `y` into separate functions. It's cleaner and more flexible to let the calling code decide how to orchestrate the two promises -- using `Promise.all([...])` here, but certainly not the only option -- rather than to abstract such details away inside of `foo(...)`.

Unwrap/Spread Arguments

The `var x = ...` and `var y = ...` assignments are still awkward overhead. We can employ some functional trickery (hat tip to Reginald Braithwaite, @raganwald on Twitter) in a helper utility:

```
function spread(fn) {
  return Function.prototype.bind.call( fn, null );
}

Promise.all(
  foo( 10, 20 )
)
.then(
  spread( function(x,y){
    console.log( x, y );      // 200 599
  } )
)
```

That's a bit nicer! Of course, you could inline the functional magic to avoid the extra helper:

```
Promise.all(
  foo( 10, 20 )
)
.then( Function.prototype.bind(
  function(x,y){
    console.log( x, y );      // 200 599
  },
  null
) );
```

These tricks may be neat, but ES6 has an even better answer for us: destructuring. The array destructuring assignment form looks like this:

```
Promise.all(  
  foo( 10, 20 )  
)  
.then( function(msgs){  
  var [x,y] = msgs;  
  
  console.log( x, y );    // 200 599  
} );
```

But best of all, ES6 offers the array parameter destructuring form:

```
Promise.all(  
  foo( 10, 20 )  
)  
.then( function([x,y]){  
  console.log( x, y );    // 200 599  
} );
```

We've now embraced the one-value-per-Promise mantra, but kept our supporting boilerplate to a minimum!

Note: For more information on ES6 destructuring forms, see the *ES6 & Beyond* title of this series.

Single Resolution

One of the most intrinsic behaviors of Promises is that a Promise can only be resolved once (fulfillment or rejection). For many async use cases, you're only retrieving a value once, so this works fine.

But there's also a lot of async cases that fit into a different model -- one that's more akin to events and/or streams of data. It's not clear on the surface how well Promises can fit into such use cases, if at all. Without a significant abstraction on top of Promises, they will completely fall short for handling multiple value resolution.

Imagine a scenario where you might want to fire off a sequence of async steps in response to a stimulus (like an event) that can in fact happen multiple times, like a button click.

This probably won't work the way you want:

```
// `click(..)` binds the `click` event to a DOM element
// `request(..)` is the previously defined Promise-aware Ajax

var p = new Promise( function(resolve,reject){
    click( "#mybtn", resolve );
} );

p.then( function(evt){
    var btnID = evt.currentTarget.id;
    return request( "http://some.url.1/?id=" + btnID );
} )
.then( function(text){
    console.log( text );
} );
```

The behavior here only works if your application calls for the button to be clicked just once. If the button is clicked a second time, the `p` promise has already been resolved, so the second `resolve(..)` call would be ignored.

Instead, you'd probably need to invert the paradigm, creating a whole new Promise chain for each event firing:

```
click( "#mybtn", function(evt){
    var btnID = evt.currentTarget.id;

    request( "http://some.url.1/?id=" + btnID )
    .then( function(text){
        console.log( text );
    } );
} );
```

This approach will work in that a whole new Promise sequence will be fired off for each `"click"` event on the button.

But beyond just the ugliness of having to define the entire Promise chain inside the event handler, this design in some respects violates the idea of separation of concerns/capabilities (SoC). You might very well want to define your event handler in a different place in your code from where you define the *response* to the event (the Promise chain). That's pretty awkward to do in this pattern, without helper mechanisms.

Note: Another way of articulating this limitation is that it'd be nice if we could construct some sort of "observable" that we can subscribe a Promise chain to. There are libraries that have created these abstractions (such as RxJS -- <http://rxjs.codeplex.com/>), but the abstractions can seem so heavy that you can't even see the nature of Promises anymore. Such heavy

abstraction brings important questions to mind such as whether (sans Promises) these mechanisms are as *trustable* as Promises themselves have been designed to be. We'll revisit the "Observable" pattern in Appendix B.

Inertia

One concrete barrier to starting to use Promises in your own code is all the code that currently exists which is not already Promise-aware. If you have lots of callback-based code, it's far easier to just keep coding in that same style.

"A code base in motion (with callbacks) will remain in motion (with callbacks) unless acted upon by a smart, Promises-aware developer."

Promises offer a different paradigm, and as such, the approach to the code can be anywhere from just a little different to, in some cases, radically different. You have to be intentional about it, because Promises will not just naturally shake out from the same ol' ways of doing code that have served you well thus far.

Consider a callback-based scenario like the following:

```
function foo(x,y,cb) {
  ajax(
    "http://some.url.1/?x=" + x + "&y=" + y,
    cb
  );
}

foo( 11, 31, function(err,text) {
  if (err) {
    console.error( err );
  }
  else {
    console.log( text );
  }
});
```

Is it immediately obvious what the first steps are to convert this callback-based code to Promise-aware code? Depends on your experience. The more practice you have with it, the more natural it will feel. But certainly, Promises don't just advertise on the label exactly how to do it -- there's no one-size-fits-all answer -- so the responsibility is up to you.

As we've covered before, we definitely need an Ajax utility that is Promise-aware instead of callback-based, which we could call `request(...)`. You can make your own, as we have already. But the overhead of having to manually define Promise-aware wrappers for every

callback-based utility makes it less likely you'll choose to refactor to Promise-aware coding at all.

Promises offer no direct answer to that limitation. Most Promise libraries do offer a helper, however. But even without a library, imagine a helper like this:

```
// polyfill-safe guard check
if (!Promise.wrap) {
  Promise.wrap = function(fn) {
    return function() {
      var args = [].slice.call( arguments );
      return new Promise( function(resolve,reject){
        fn.apply(
          null,
          args.concat( function(err,v){
            if (err) {
              reject( err );
            }
            else {
              resolve( v );
            }
          } )
        );
      } );
    };
  };
}
```

OK, that's more than just a tiny trivial utility. However, although it may look a bit intimidating, it's not as bad as you'd think. It takes a function that expects an error-first style callback as its last parameter, and returns a new one that automatically creates a Promise to return, and substitutes the callback for you, wired up to the Promise fulfillment/rejection.

Rather than waste too much time talking about *how* this `Promise.wrap(..)` helper works, let's just look at how we use it:

```
var request = Promise.wrap( ajax );

request( "http://some.url.1/" )
.then( ... )
...
```

Wow, that was pretty easy!

`Promise.wrap(...)` does **not** produce a Promise. It produces a function that will produce Promises. In a sense, a Promise-producing function could be seen as a "Promise factory." I propose "promisory" as the name for such a thing ("Promise" + "factory").

The act of wrapping a callback-expecting function to be a Promise-aware function is sometimes referred to as "lifting" or "promisifying". But there doesn't seem to be a standard term for what to call the resultant function other than a "lifted function", so I like "promisory" better as I think it's more descriptive.

Note: Promisory isn't a made-up term. It's a real word, and its definition means to contain or convey a promise. That's exactly what these functions are doing, so it turns out to be a pretty perfect terminology match!

So, `Promise.wrap.ajax()` produces an `ajax(...)` promisory we call `request(...)`, and that promisory produces Promises for Ajax responses.

If all functions were already promisories, we wouldn't need to make them ourselves, so the extra step is a tad bit of a shame. But at least the wrapping pattern is (usually) repeatable so we can put it into a `Promise.wrap(...)` helper as shown to aid our promise coding.

So back to our earlier example, we need a promisory for both `ajax(...)` and `foo(...)`:

```
// make a promisory for `ajax(..)`  
var request = Promise.wrap( ajax );  
  
// refactor `foo(..)`, but keep it externally  
// callback-based for compatibility with other  
// parts of the code for now -- only use  
// `request(..)`'s promise internally.  
function foo(x,y,cb) {  
    request(  
        "http://some.url.1/?x=" + x + "&y=" + y  
    )  
    .then(  
        function fulfilled(text){  
            cb( null, text );  
        },  
        cb  
    );  
}  
  
// now, for this code's purposes, make a  
// promisory for `foo(..)`  
var betterFoo = Promise.wrap( foo );  
  
// and use the promisory  
betterFoo( 11, 31 )  
.then(  
    function fulfilled(text){  
        console.log( text );  
    },  
    function rejected(err){  
        console.error( err );  
    }  
);
```

Of course, while we're refactoring `foo(..)` to use our new `request(..)` promisory, we could just make `foo(..)` a promisory itself, instead of remaining callback-based and needing to make and use the subsequent `betterFoo(..)` promisory. This decision just depends on whether `foo(..)` needs to stay callback-based compatible with other parts of the code base or not.

Consider:

```
// `foo(..)` is now also a promisory because it
// delegates to the `request(..)` promisory
function foo(x,y) {
  return request(
    "http://some.url.1/?x=" + x + "&y=" + y
  );
}

foo( 11, 31 )
.then( ... )
..
```

While ES6 Promises don't natively ship with helpers for such promisory wrapping, most libraries provide them, or you can make your own. Either way, this particular limitation of Promises is addressable without too much pain (certainly compared to the pain of callback hell!).

Promise Uncancelable

Once you create a Promise and register a fulfillment and/or rejection handler for it, there's nothing external you can do to stop that progression if something else happens to make that task moot.

Note: Many Promise abstraction libraries provide facilities to cancel Promises, but this is a terrible idea! Many developers wish Promises had natively been designed with external cancellation capability, but the problem is that it would let one consumer/observer of a Promise affect some other consumer's ability to observe that same Promise. This violates the future-value's trustability (external immutability), but moreover is the embodiment of the "action at a distance" anti-pattern

(http://en.wikipedia.org/wiki/Action_at_a_distance_%28computer_programming%29).

Regardless of how useful it seems, it will actually lead you straight back into the same nightmares as callbacks.

Consider our Promise timeout scenario from earlier:

```
var p = foo( 42 );

Promise.race( [
  p,
  timeoutPromise( 3000 )
] )
.then(
  doSomething,
  handleError
);

p.then( function(){
  // still happens even in the timeout case :(
} );
```

The "timeout" was external to the promise `p`, so `p` itself keeps going, which we probably don't want.

One option is to invasively define your resolution callbacks:

```
var OK = true;

var p = foo( 42 );

Promise.race( [
  p,
  timeoutPromise( 3000 )
  .catch( function(err){
    OK = false;
    throw err;
  } )
] )
.then(
  doSomething,
  handleError
);

p.then( function(){
  if (OK) {
    // only happens if no timeout! :)
  }
} );
```

This is ugly. It works, but it's far from ideal. Generally, you should try to avoid such scenarios.

But if you can't, the ugliness of this solution should be a clue that *cancellation* is a functionality that belongs at a higher level of abstraction on top of Promises. I'd recommend you look to Promise abstraction libraries for assistance rather than hacking it yourself.

Note: My `asynquence` Promise abstraction library provides just such an abstraction and an `abort()` capability for the sequence, all of which will be discussed in Appendix A.

A single Promise is not really a flow-control mechanism (at least not in a very meaningful sense), which is exactly what *cancellation* refers to; that's why Promise cancellation would feel awkward.

By contrast, a chain of Promises taken collectively together -- what I like to call a "sequence" -- *is* a flow control expression, and thus it's appropriate for cancellation to be defined at that level of abstraction.

No individual Promise should be cancelable, but it's sensible for a *sequence* to be cancelable, because you don't pass around a sequence as a single immutable value like you do with a Promise.

Promise Performance

This particular limitation is both simple and complex.

Comparing how many pieces are moving with a basic callback-based async task chain versus a Promise chain, it's clear Promises have a fair bit more going on, which means they are naturally at least a tiny bit slower. Think back to just the simple list of trust guarantees that Promises offer, as compared to the ad hoc solution code you'd have to layer on top of callbacks to achieve the same protections.

More work to do, more guards to protect, means that Promises *are* slower as compared to naked, untrustable callbacks. That much is obvious, and probably simple to wrap your brain around.

But how much slower? Well... that's actually proving to be an incredibly difficult question to answer absolutely, across the board.

Frankly, it's kind of an apples-to-oranges comparison, so it's probably the wrong question to ask. You should actually compare whether an ad-hoc callback system with all the same protections manually layered in is faster than a Promise implementation.

If Promises have a legitimate performance limitation, it's more that they don't really offer a line-item choice as to which trustability protections you want/need or not -- you get them all, always.

Nevertheless, if we grant that a Promise is generally a *little bit slower* than its non-Promise, non-trustable callback equivalent -- assuming there are places where you feel you can justify the lack of trustability -- does that mean that Promises should be avoided across the board, as if your entire application is driven by nothing but must-be-utterly-the-fastest code possible?

Sanity check: if your code is legitimately like that, **is JavaScript even the right language for such tasks?** JavaScript can be optimized to run applications very performantly (see Chapter 5 and Chapter 6). But is obsessing over tiny performance tradeoffs with Promises, in light of all the benefits they offer, *really* appropriate?

Another subtle issue is that Promises make *everything* async, which means that some immediately (synchronously) complete steps still defer advancement of the next step to a Job (see Chapter 1). That means that it's possible that a sequence of Promise tasks could complete ever-so-slightly slower than the same sequence wired up with callbacks.

Of course, the question here is this: are these potential slips in tiny fractions of performance *worth* all the other articulated benefits of Promises we've laid out across this chapter?

My take is that in virtually all cases where you might think Promise performance is slow enough to be concerned, it's actually an anti-pattern to optimize away the benefits of Promise trustability and composability by avoiding them altogether.

Instead, you should default to using them across the code base, and then profile and analyze your application's hot (critical) paths. Are Promises *really* a bottleneck, or are they just a theoretical slowdown? Only *then*, armed with actual valid benchmarks (see Chapter 6) is it responsible and prudent to factor out the Promises in just those identified critical areas.

Promises are a little slower, but in exchange you're getting a lot of trustability, non-Zalgo predictability, and composability built in. Maybe the limitation is not actually their performance, but your lack of perception of their benefits?

Review

Promises are awesome. Use them. They solve the *inversion of control* issues that plague us with callbacks-only code.

They don't get rid of callbacks, they just redirect the orchestration of those callbacks to a trustable intermediary mechanism that sits between us and another utility.

Promise chains also begin to address (though certainly not perfectly) a better way of expressing async flow in sequential fashion, which helps our brains plan and maintain async JS code better. We'll see an even better solution to *that* problem in the next chapter!

In Chapter 2, we identified two key drawbacks to expressing async flow control with callbacks:

- Callback-based async doesn't fit how our brain plans out steps of a task.
- Callbacks aren't trustable or composable because of *inversion of control*.

In Chapter 3, we detailed how Promises uninvert the *inversion of control* of callbacks, restoring trustability/composability.

Now we turn our attention to expressing async flow control in a sequential, synchronous-looking fashion. The "magic" that makes it possible is ES6 **generators**.

Breaking Run-to-Completion

In Chapter 1, we explained an expectation that JS developers almost universally rely on in their code: once a function starts executing, it runs until it completes, and no other code can interrupt and run in between.

As bizarre as it may seem, ES6 introduces a new type of function that does not behave with the run-to-completion behavior. This new type of function is called a "generator."

To understand the implications, let's consider this example:

```
var x = 1;

function foo() {
    x++;
    bar();           // <-- what about this line?
    console.log("x:", x);
}

function bar() {
    x++;
}

foo();           // x: 3
```

In this example, we know for sure that `bar()` runs in between `x++` and `console.log(x)`. But what if `bar()` wasn't there? Obviously the result would be `2` instead of `3`.

Now let's twist your brain. What if `bar()` wasn't present, but it could still somehow run between the `x++` and `console.log(x)` statements? How would that be possible?

In **preemptive** multithreaded languages, it would essentially be possible for `bar()` to "interrupt" and run at exactly the right moment between those two statements. But JS is not preemptive, nor is it (currently) multithreaded. And yet, a **cooperative** form of this

"interruption" (concurrency) is possible, if `foo()` itself could somehow indicate a "pause" at that part in the code.

Note: I use the word "cooperative" not only because of the connection to classical concurrency terminology (see Chapter 1), but because as you'll see in the next snippet, the ES6 syntax for indicating a pause point in code is `yield` -- suggesting a politely *cooperative* yielding of control.

Here's the ES6 code to accomplish such cooperative concurrency:

```
var x = 1;

function *foo() {
  x++;
  yield; // pause!
  console.log("x:", x);
}

function bar() {
  x++;
}
```

Note: You will likely see most other JS documentation/code that will format a generator declaration as `function* foo() { .. }` instead of as I've done here with `function *foo() { .. }` -- the only difference being the stylistic positioning of the `*`. The two forms are functionally/syntactically identical, as is a third `function*foo() { .. }` (no space) form. There are arguments for both styles, but I basically prefer `function *foo..` because it then matches when I reference a generator in writing with `*foo()`. If I said only `foo()`, you wouldn't know as clearly if I was talking about a generator or a regular function. It's purely a stylistic preference.

Now, how can we run the code in that previous snippet such that `bar()` executes at the point of the `yield` inside of `*foo()`?

```
// construct an iterator `it` to control the generator
var it = foo();

// start `foo()` here!
it.next();
x;                      // 2
bar();                  // 3
x;                      // x: 3
it.next();
```

OK, there's quite a bit of new and potentially confusing stuff in those two code snippets, so we've got plenty to wade through. But before we explain the different mechanics/syntax with ES6 generators, let's walk through the behavior flow:

1. The `it = foo()` operation does *not* execute the `*foo()` generator yet, but it merely constructs an *iterator* that will control its execution. More on *iterators* in a bit.
2. The first `it.next()` starts the `*foo()` generator, and runs the `x++` on the first line of `*foo()`.
3. `*foo()` pauses at the `yield` statement, at which point that first `it.next()` call finishes. At the moment, `*foo()` is still running and active, but it's in a paused state.
4. We inspect the value of `x`, and it's now `2`.
5. We call `bar()`, which increments `x` again with `x++`.
6. We inspect the value of `x` again, and it's now `3`.
7. The final `it.next()` call resumes the `*foo()` generator from where it was paused, and runs the `console.log(..)` statement, which uses the current value of `x` of `3`.

Clearly, `*foo()` started, but did *not* run-to-completion -- it paused at the `yield`. We resumed `*foo()` later, and let it finish, but that wasn't even required.

So, a generator is a special kind of function that can start and stop one or more times, and doesn't necessarily ever have to finish. While it won't be terribly obvious yet why that's so powerful, as we go throughout the rest of this chapter, that will be one of the fundamental building blocks we use to construct generators-as-async-flow-control as a pattern for our code.

Input and Output

A generator function is a special function with the new processing model we just alluded to. But it's still a function, which means it still has some basic tenets that haven't changed -- namely, that it still accepts arguments (aka "input"), and that it can still return a value (aka "output"):

```
function *foo(x,y) {
  return x * y;
}

var it = foo( 6, 7 );

var res = it.next();

res.value;           // 42
```

We pass in the arguments `6` and `7` to `*foo(..)` as the parameters `x` and `y`, respectively. And `*foo(..)` returns the value `42` back to the calling code.

We now see a difference with how the generator is invoked compared to a normal function. `foo(6, 7)` obviously looks familiar. But subtly, the `*foo(..)` generator hasn't actually run yet as it would have with a function.

Instead, we're just creating an *iterator* object, which we assign to the variable `it`, to control the `*foo(..)` generator. Then we call `it.next()`, which instructs the `*foo(..)` generator to advance from its current location, stopping either at the next `yield` or end of the generator.

The result of that `next(..)` call is an object with a `value` property on it holding whatever value (if anything) was returned from `*foo(..)`. In other words, `yield` caused a value to be sent out from the generator during the middle of its execution, kind of like an intermediate `return`.

Again, it won't be obvious yet why we need this whole indirect *iterator* object to control the generator. We'll get there, I *promise*.

Iteration Messaging

In addition to generators accepting arguments and having return values, there's even more powerful and compelling input/output messaging capability built into them, via `yield` and `next(..)`.

Consider:

```
function *foo(x) {
  var y = x * (yield);
  return y;
}

var it = foo( 6 );

// start `foo(..)`
it.next();

var res = it.next( 7 );

res.value;           // 42
```

First, we pass in `6` as the parameter `x`. Then we call `it.next()`, and it starts up `*foo(..)`.

Inside `*foo(...)`, the `var y = x ...` statement starts to be processed, but then it runs across a `yield` expression. At that point, it pauses `*foo(...)` (in the middle of the assignment statement!), and essentially requests the calling code to provide a result value for the `yield` expression. Next, we call `it.next(7)`, which is passing the `7` value back in to be that result of the paused `yield` expression.

So, at this point, the assignment statement is essentially `var y = 6 * 7`. Now, `return y` returns that `42` value back as the result of the `it.next(7)` call.

Notice something very important but also easily confusing, even to seasoned JS developers: depending on your perspective, there's a mismatch between the `yield` and the `next(...)` call. In general, you're going to have one more `next(...)` call than you have `yield` statements -- the preceding snippet has one `yield` and two `next(...)` calls.

Why the mismatch?

Because the first `next(...)` always starts a generator, and runs to the first `yield`. But it's the second `next(...)` call that fulfills the first paused `yield` expression, and the third `next(...)` would fulfill the second `yield`, and so on.

Tale of Two Questions

Actually, which code you're thinking about primarily will affect whether there's a perceived mismatch or not.

Consider only the generator code:

```
var y = x * (yield);
return y;
```

This **first** `yield` is basically *asking a question*: "What value should I insert here?"

Who's going to answer that question? Well, the **first** `next()` has already run to get the generator up to this point, so obviously *it* can't answer the question. So, the **second** `next(...)` call must answer the question *posed by the first yield*.

See the mismatch -- second-to-first?

But let's flip our perspective. Let's look at it not from the generator's point of view, but from the iterator's point of view.

To properly illustrate this perspective, we also need to explain that messages can go in both directions -- `yield ..` as an expression can send out messages in response to `next(...)` calls, and `next(...)` can send values to a paused `yield` expression. Consider this slightly adjusted code:

```

function *foo(x) {
    var y = x * (yield "Hello"); // <-- yield a value!
    return y;
}

var it = foo( 6 );

var res = it.next(); // first `next()`, don't pass anything
res.value;           // "Hello"

res = it.next( 7 ); // pass `7` to waiting `yield`
res.value;           // 42

```

`yield ..` and `next(..)` pair together as a two-way message passing system **during the execution of the generator**.

So, looking only at the *iterator* code:

```

var res = it.next(); // first `next()`, don't pass anything
res.value;           // "Hello"

res = it.next( 7 ); // pass `7` to waiting `yield`
res.value;           // 42

```

Note: We don't pass a value to the first `next()` call, and that's on purpose. Only a paused `yield` could accept such a value passed by a `next(..)`, and at the beginning of the generator when we call the first `next()`, there **is no paused `yield`** to accept such a value. The specification and all compliant browsers just silently **discard** anything passed to the first `next()`. It's still a bad idea to pass a value, as you're just creating silently "failing" code that's confusing. So, always start a generator with an argument-free `next()`.

The first `next()` call (with nothing passed to it) is basically *asking a question*: "What *next* value does the `*foo(..)` generator have to give me?" And who answers this question? The first `yield "hello"` expression.

See? No mismatch there.

Depending on *who* you think about asking the question, there is either a mismatch between the `yield` and `next(..)` calls, or not.

But wait! There's still an extra `next()` compared to the number of `yield` statements. So, that final `it.next(7)` call is again asking the question about what *next* value the generator will produce. But there's no more `yield` statements left to answer, is there? So who answers?

The `return` statement answers the question!

And if there **is no** `return` in your generator -- `return` is certainly not any more required in generators than in regular functions -- there's always an assumed/implicit `return;` (aka `return undefined;`), which serves the purpose of default answering the question *posed by the final `it.next(7)` call.*

These questions and answers -- the two-way message passing with `yield` and `next(...)` -- are quite powerful, but it's not obvious at all how these mechanisms are connected to async flow control. We're getting there!

Multiple Iterators

It may appear from the syntactic usage that when you use an *iterator* to control a generator, you're controlling the declared generator function itself. But there's a subtlety that's easy to miss: each time you construct an *iterator*, you are implicitly constructing an instance of the generator which that *iterator* will control.

You can have multiple instances of the same generator running at the same time, and they can even interact:

```
function *foo() {
  var x = yield 2;
  z++;
  var y = yield (x * z);
  console.log(x, y, z);
}

var z = 1;

var it1 = foo();
var it2 = foo();

var val1 = it1.next().value;           // 2 <-- yield 2
var val2 = it2.next().value;           // 2 <-- yield 2

val1 = it1.next(val2 * 10).value;      // 40 <-- x:20, z:2
val2 = it2.next(val1 * 5).value;       // 600 <-- x:200, z:3

it1.next(val2 / 2);                  // y:300
                                    // 20 300 3
it2.next(val1 / 4);                  // y:10
                                    // 200 10 3
```

Warning: The most common usage of multiple instances of the same generator running concurrently is not such interactions, but when the generator is producing its own values without input, perhaps from some independently connected resource. We'll talk more about value production in the next section.

Let's briefly walk through the processing:

1. Both instances of `*foo()` are started at the same time, and both `next()` calls reveal a value of `2` from the `yield 2` statements, respectively.
2. `val2 * 10` is `2 * 10`, which is sent into the first generator instance `it1`, so that `x` gets value `20`. `z` is incremented from `1` to `2`, and then `20 * 2` is yielded out, setting `val1` to `40`.
3. `val1 * 5` is `40 * 5`, which is sent into the second generator instance `it2`, so that `x` gets value `200`. `z` is incremented again, from `2` to `3`, and then `200 * 3` is yielded out, setting `val2` to `600`.
4. `val2 / 2` is `600 / 2`, which is sent into the first generator instance `it1`, so that `y` gets value `300`, then printing out `20 300 3` for its `x y z` values, respectively.
5. `val1 / 4` is `40 / 4`, which is sent into the second generator instance `it2`, so that `y` gets value `10`, then printing out `200 10 3` for its `x y z` values, respectively.

That's a "fun" example to run through in your mind. Did you keep it straight?

Interleaving

Recall this scenario from the "Run-to-completion" section of Chapter 1:

```
var a = 1;
var b = 2;

function foo() {
    a++;
    b = b * a;
    a = b + 3;
}

function bar() {
    b--;
    a = 8 + b;
    b = a * 2;
}
```

With normal JS functions, of course either `foo()` can run completely first, or `bar()` can run completely first, but `foo()` cannot interleave its individual statements with `bar()`. So, there are only two possible outcomes to the preceding program.

However, with generators, clearly interleaving (even in the middle of statements!) is possible:

```

var a = 1;
var b = 2;

function *foo() {
    a++;
    yield;
    b = b * a;
    a = (yield b) + 3;
}

function *bar() {
    b--;
    yield;
    a = (yield 8) + b;
    b = a * (yield 2);
}

```

Depending on what respective order the *iterators* controlling `*foo()` and `*bar()` are called, the preceding program could produce several different results. In other words, we can actually illustrate (in a sort of fake-ish way) the theoretical "threaded race conditions" circumstances discussed in Chapter 1, by interleaving the two generator interations over the same shared variables.

First, let's make a helper called `step(...)` that controls an *iterator*:

```

function step(gen) {
    var it = gen();
    var last;

    return function() {
        // whatever is `yield`ed out, just
        // send it right back in the next time!
        last = it.next( last ).value;
    };
}

```

`step(...)` initializes a generator to create its `it` *iterator*, then returns a function which, when called, advances the *iterator* by one step. Additionally, the previously `yield` ed out value is sent right back in at the *next* step. So, `yield 8` will just become `8` and `yield b` will just be `b` (whatever it was at the time of `yield`).

Now, just for fun, let's experiment to see the effects of interleaving these different chunks of `*foo()` and `*bar()`. We'll start with the boring base case, making sure `*foo()` totally finishes before `*bar()` (just like we did in Chapter 1):

```
// make sure to reset `a` and `b`
a = 1;
b = 2;

var s1 = step( foo );
var s2 = step( bar );

// run `*foo()` completely first
s1();
s1();
s1();

// now run `*bar()`
s2();
s2();
s2();
s2();

console.log( a, b );    // 11 22
```

The end result is `11` and `22`, just as it was in the Chapter 1 version. Now let's mix up the interleaving ordering and see how it changes the final values of `a` and `b`:

```
// make sure to reset `a` and `b`
a = 1;
b = 2;

var s1 = step( foo );
var s2 = step( bar );

s2();      // b--;
s2();      // yield 8
s1();      // a++;
s2();      // a = 8 + b;
           // yield 2
s1();      // b = b * a;
           // yield b
s1();      // a = b + 3;
s2();      // b = a * 2;
```

Before I tell you the results, can you figure out what `a` and `b` are after the preceding program? No cheating!

```
console.log( a, b );    // 12 18
```

Note: As an exercise for the reader, try to see how many other combinations of results you can get back rearranging the order of the `s1()` and `s2()` calls. Don't forget you'll always need three `s1()` calls and four `s2()` calls. Recall the discussion earlier about matching `next()` with `yield` for the reasons why.

You almost certainly won't want to intentionally create *this* level of interleaving confusion, as it creates incredibly difficult to understand code. But the exercise is interesting and instructive to understand more about how multiple generators can run concurrently in the same shared scope, because there will be places where this capability is quite useful.

We'll discuss generator concurrency in more detail at the end of this chapter.

Generator'ing Values

In the previous section, we mentioned an interesting use for generators, as a way to produce values. This is **not** the main focus in this chapter, but we'd be remiss if we didn't cover the basics, especially because this use case is essentially the origin of the name: generators.

We're going to take a slight diversion into the topic of *iterators* for a bit, but we'll circle back to how they relate to generators and using a generator to *generate* values.

Producers and Iterators

Imagine you're producing a series of values where each value has a definable relationship to the previous value. To do this, you're going to need a stateful producer that remembers the last value it gave out.

You can implement something like that straightforwardly using a function closure (see the *Scope & Closures* title of this series):

```

var gimmeSomething = (function(){
  var nextVal;

  return function(){
    if (nextVal === undefined) {
      nextVal = 1;
    }
    else {
      nextVal = (3 * nextVal) + 6;
    }

    return nextVal;
  };
})();

gimmeSomething();          // 1
gimmeSomething();          // 9
gimmeSomething();          // 33
gimmeSomething();          // 105

```

Note: The `nextVal` computation logic here could have been simplified, but conceptually, we don't want to calculate the *next value* (aka `nextVal`) until the *next* `gimmeSomething()` call happens, because in general that could be a resource-leaky design for producers of more persistent or resource-limited values than simple `number`s.

Generating an arbitrary number series isn't a terribly realistic example. But what if you were generating records from a data source? You could imagine much the same code.

In fact, this task is a very common design pattern, usually solved by iterators. An *iterator* is a well-defined interface for stepping through a series of values from a producer. The JS interface for iterators, as it is in most languages, is to call `next()` each time you want the next value from the producer.

We could implement the standard *iterator* interface for our number series producer:

```

var something = (function(){
  var nextVal;

  return {
    // needed for `for..of` loops
    [Symbol.iterator]: function(){ return this; },

    // standard iterator interface method
    next: function(){
      if (nextVal === undefined) {
        nextVal = 1;
      }
      else {
        nextVal = (3 * nextVal) + 6;
      }

      return { done:false, value:nextVal };
    }
  };
})();

something.next().value;          // 1
something.next().value;          // 9
something.next().value;          // 33
something.next().value;          // 105

```

Note: We'll explain why we need the `[Symbol.iterator]: ...` part of this code snippet in the "Iterables" section. Syntactically though, two ES6 features are at play. First, the `[..]` syntax is called a *computed property name* (see the *this & Object Prototypes* title of this series). It's a way in an object literal definition to specify an expression and use the result of that expression as the name for the property. Next, `symbol.iterator` is one of ES6's predefined special `symbol` values (see the *ES6 & Beyond* title of this book series).

The `next()` call returns an object with two properties: `done` is a `boolean` value signaling the *iterator's* complete status; `value` holds the iteration value.

ES6 also adds the `for..of` loop, which means that a standard *iterator* can automatically be consumed with native loop syntax:

```

for (var v of something) {
  console.log( v );

  // don't let the loop run forever!
  if (v > 500) {
    break;
  }
}

// 1 9 33 105 321 969

```

Note: Because our `something iterator` always returns `done:false`, this `for..of` loop would run forever, which is why we put the `break` conditional in. It's totally OK for iterators to be never-ending, but there are also cases where the `iterator` will run over a finite set of values and eventually return a `done:true`.

The `for..of` loop automatically calls `next()` for each iteration -- it doesn't pass any values in to the `next()` -- and it will automatically terminate on receiving a `done:true`. It's quite handy for looping over a set of data.

Of course, you could manually loop over iterators, calling `next()` and checking for the `done:true` condition to know when to stop:

```
for (
  var ret;
  (ret = something.next()) && !ret.done;
) {
  console.log( ret.value );

  // don't let the loop run forever!
  if (ret.value > 500) {
    break;
  }
}
// 1 9 33 105 321 969
```

Note: This manual `for` approach is certainly uglier than the ES6 `for..of` loop syntax, but its advantage is that it affords you the opportunity to pass in values to the `next(..)` calls if necessary.

In addition to making your own `iterators`, many built-in data structures in JS (as of ES6), like `array`s, also have default `iterators`:

```
var a = [1,3,5,7,9];

for (var v of a) {
  console.log( v );
}
// 1 3 5 7 9
```

The `for..of` loop asks `a` for its `iterator`, and automatically uses it to iterate over `a`'s values.

Note: It may seem a strange omission by ES6, but regular `object`s intentionally do not come with a default `iterator` the way `array`s do. The reasons go deeper than we will cover here. If all you want is to iterate over the properties of an object (with no particular guarantee of ordering), `Object.keys(..)` returns an `array`, which can then be used like `for (var k of`

`Object.keys(obj)) { ... }` Such a `for..of` loop over an object's keys would be similar to a `for..in` loop, except that `Object.keys(...)` does not include properties from the `[[Prototype]]` chain while `for..in` does (see the *this & Object Prototypes* title of this series).

Iterables

The `something` object in our running example is called an *iterator*, as it has the `next()` method on its interface. But a closely related term is *iterable*, which is an `object` that **contains** an *iterator* that can iterate over its values.

As of ES6, the way to retrieve an *iterator* from an *iterable* is that the *iterable* must have a function on it, with the name being the special ES6 symbol value `Symbol.iterator`. When this function is called, it returns an *iterator*. Though not required, generally each call should return a fresh new *iterator*.

`a` in the previous snippet is an *iterable*. The `for..of` loop automatically calls its `Symbol.iterator` function to construct an *iterator*. But we could of course call the function manually, and use the *iterator* it returns:

```
var a = [1, 3, 5, 7, 9];

var it = a[Symbol.iterator]();

it.next().value;    // 1
it.next().value;    // 3
it.next().value;    // 5
..
```

In the previous code listing that defined `something`, you may have noticed this line:

```
[Symbol.iterator]: function(){ return this; }
```

That little bit of confusing code is making the `something` value -- the interface of the `something` *iterator* -- also an *iterable*; it's now both an *iterable* and an *iterator*. Then, we pass `something` to the `for..of` loop:

```
for (var v of something) {
  ...
}
```

The `for..of` loop expects `something` to be an *iterable*, so it looks for and calls its `Symbol.iterator` function. We defined that function to simply `return this`, so it just gives itself back, and the `for..of` loop is none the wiser.

Generator Iterator

Let's turn our attention back to generators, in the context of *iterators*. A generator can be treated as a producer of values that we extract one at a time through an *iterator* interface's `next()` calls.

So, a generator itself is not technically an *iterable*, though it's very similar -- when you execute the generator, you get an *iterator* back:

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

var it = foo();
```

We can implement the `something` infinite number series producer from earlier with a generator, like this:

```
function *something() {
    var nextVal;

    while (true) {
        if (nextVal === undefined) {
            nextVal = 1;
        }
        else {
            nextVal = (3 * nextVal) + 6;
        }

        yield nextVal;
    }
}
```

Note: A `while..true` loop would normally be a very bad thing to include in a real JS program, at least if it doesn't have a `break` or `return` in it, as it would likely run forever, synchronously, and block/lock-up the browser UI. However, in a generator, such a loop is generally totally OK if it has a `yield` in it, as the generator will pause at each iteration, `yield`ing back to the main program and/or to the event loop queue. To put it glibly, "generators put the `while..true` back in JS programming!"

That's a fair bit cleaner and simpler, right? Because the generator pauses at each `yield`, the state (scope) of the function `*something()` is kept around, meaning there's no need for the closure boilerplate to preserve variable state across calls.

Not only is it simpler code -- we don't have to make our own `iterator` interface -- it actually is more reasonable code, because it more clearly expresses the intent. For example, the `while..true` loop tells us the generator is intended to run forever -- to keep *generating* values as long as we keep asking for them.

And now we can use our shiny new `*something()` generator with a `for..of` loop, and you'll see it works basically identically:

```
for (var v of something()) {
    console.log( v );

    // don't let the loop run forever!
    if (v > 500) {
        break;
    }
}

// 1 9 33 105 321 969
```

But don't skip over `for (var v of something()) ...`! We didn't just reference `something` as a value like in earlier examples, but instead called the `*something()` generator to get its *iterator* for the `for..of` loop to use.

If you're paying close attention, two questions may arise from this interaction between the generator and the loop:

- Why couldn't we say `for (var v of something) ...`? Because `something` here is a generator, which is not an *iterable*. We have to call `something()` to construct a producer for the `for..of` loop to iterate over.
- The `something()` call produces an *iterator*, but the `for..of` loop wants an *iterable*, right? Yep. The generator's *iterator* also has a `Symbol.iterator` function on it, which basically does a `return this`, just like the `something iterable` we defined earlier. In other words, a generator's *iterator* is also an *iterable*!

Stopping the Generator

In the previous example, it would appear the *iterator* instance for the `*something()` generator was basically left in a suspended state forever after the `break` in the loop was called.

But there's a hidden behavior that takes care of that for you. "Abnormal completion" (i.e., "early termination") of the `for..of` loop -- generally caused by a `break`, `return`, or an uncaught exception -- sends a signal to the generator's *iterator* for it to terminate.

Note: Technically, the `for..of` loop also sends this signal to the *iterator* at the normal completion of the loop. For a generator, that's essentially a moot operation, as the generator's *iterator* had to complete first so the `for..of` loop completed. However, custom *iterators* might desire to receive this additional signal from `for..of` loop consumers.

While a `for..of` loop will automatically send this signal, you may wish to send the signal manually to an *iterator*; you do this by calling `return(...)`.

If you specify a `try..finally` clause inside the generator, it will always be run even when the generator is externally completed. This is useful if you need to clean up resources (database connections, etc.):

```
function *something() {
  try {
    var nextVal;

    while (true) {
      if (nextVal === undefined) {
        nextVal = 1;
      }
      else {
        nextVal = (3 * nextVal) + 6;
      }

      yield nextVal;
    }
  }
  // cleanup clause
  finally {
    console.log( "cleaning up!" );
  }
}
```

The earlier example with `break` in the `for..of` loop will trigger the `finally` clause. But you could instead manually terminate the generator's *iterator* instance from the outside with `return(...)`:

```
var it = something();
for (var v of it) {
    console.log( v );

    // don't let the loop run forever!
    if (v > 500) {
        console.log(
            // complete the generator's iterator
            it.return( "Hello World" ).value
        );
        // no `break` needed here
    }
}
// 1 9 33 105 321 969
// cleaning up!
// Hello World
```

When we call `it.return(..)`, it immediately terminates the generator, which of course runs the `finally` clause. Also, it sets the returned `value` to whatever you passed in to `return(..)`, which is how `"Hello World"` comes right back out. We also don't need to include a `break` now because the generator's *iterator* is set to `done:true`, so the `for..of` loop will terminate on its next iteration.

Generators owe their namesake mostly to this *consuming produced values* use. But again, that's just one of the uses for generators, and frankly not even the main one we're concerned with in the context of this book.

But now that we more fully understand some of the mechanics of how they work, we can *next* turn our attention to how generators apply to async concurrency.

Iterating Generators Asynchronously

What do generators have to do with async coding patterns, fixing problems with callbacks, and the like? Let's get to answering that important question.

We should revisit one of our scenarios from Chapter 3. Let's recall the callback approach:

```

function foo(x,y,cb) {
  ajax(
    "http://some.url.1/?x=" + x + "&y=" + y,
    cb
  );
}

foo( 11, 31, function(err,text) {
  if (err) {
    console.error( err );
  }
  else {
    console.log( text );
  }
} );

```

If we wanted to express this same task flow control with a generator, we could do:

```

function foo(x,y) {
  ajax(
    "http://some.url.1/?x=" + x + "&y=" + y,
    function(err,data){
      if (err) {
        // throw an error into `*main()`
        it.throw( err );
      }
      else {
        // resume `*main()` with received `data`
        it.next( data );
      }
    }
  );
}

function *main() {
  try {
    var text = yield foo( 11, 31 );
    console.log( text );
  }
  catch (err) {
    console.error( err );
  }
}

var it = main();

// start it all up!
it.next();

```

At first glance, this snippet is longer, and perhaps a little more complex looking, than the callback snippet before it. But don't let that impression get you off track. The generator snippet is actually **much** better! But there's a lot going on for us to explain.

First, let's look at this part of the code, which is the most important:

```
var text = yield foo( 11, 31 );
console.log( text );
```

Think about how that code works for a moment. We're calling a normal function `foo(..)` and we're apparently able to get back the `text` from the Ajax call, even though it's asynchronous.

How is that possible? If you recall the beginning of Chapter 1, we had almost identical code:

```
var data = ajax( "..url 1.." );
console.log( data );
```

And that code didn't work! Can you spot the difference? It's the `yield` used in a generator.

That's the magic! That's what allows us to have what appears to be blocking, synchronous code, but it doesn't actually block the whole program; it only pauses/blocks the code in the generator itself.

In `yield foo(11,31)`, first the `foo(11,31)` call is made, which returns nothing (aka `undefined`), so we're making a call to request data, but we're actually then doing `yield undefined`. That's OK, because the code is not currently relying on a `yield` ed value to do anything interesting. We'll revisit this point later in the chapter.

We're not using `yield` in a message passing sense here, only in a flow control sense to pause/block. Actually, it will have message passing, but only in one direction, after the generator is resumed.

So, the generator pauses at the `yield`, essentially asking the question, "what value should I return to assign to the variable `text`?" Who's going to answer that question?

Look at `foo(..)`. If the Ajax request is successful, we call:

```
it.next( data );
```

That's resuming the generator with the response data, which means that our paused `yield` expression receives that value directly, and then as it restarts the generator code, that value gets assigned to the local variable `text`.

Pretty cool, huh?

Take a step back and consider the implications. We have totally synchronous-looking code inside the generator (other than the `yield` keyword itself), but hidden behind the scenes, inside of `foo(..)`, the operations can complete asynchronously.

That's huge! That's a nearly perfect solution to our previously stated problem with callbacks not being able to express asynchrony in a sequential, synchronous fashion that our brains can relate to.

In essence, we are abstracting the asynchrony away as an implementation detail, so that we can reason synchronously/sequentially about our flow control: "Make an Ajax request, and when it finishes print out the response." And of course, we just expressed two steps in the flow control, but this same capability extends without bounds, to let us express however many steps we need to.

Tip: This is such an important realization, just go back and read the last three paragraphs again to let it sink in!

Synchronous Error Handling

But the preceding generator code has even more goodness to `yield` to us. Let's turn our attention to the `try..catch` inside the generator:

```
try {
  var text = yield foo( 11, 31 );
  console.log( text );
}
catch (err) {
  console.error( err );
}
```

How does this work? The `foo(..)` call is asynchronously completing, and doesn't `try..catch` fail to catch asynchronous errors, as we looked at in Chapter 3?

We already saw how the `yield` lets the assignment statement pause to wait for `foo(..)` to finish, so that the completed response can be assigned to `text`. The awesome part is that this `yield` pausing also allows the generator to `catch` an error. We throw that error into the generator with this part of the earlier code listing:

```
if (err) {
  // throw an error into `*main()`
  it.throw( err );
}
```

The `yield`-pause nature of generators means that not only do we get synchronous-looking `return` values from `async` function calls, but we can also synchronously `catch` errors from those `async` function calls!

So we've seen we can throw errors *into* a generator, but what about throwing errors *out of* a generator? Exactly as you'd expect:

```
function *main() {
  var x = yield "Hello World";

  yield x.toLowerCase();    // cause an exception!
}

var it = main();

it.next().value;           // Hello World

try {
  it.next( 42 );
}
catch (err) {
  console.error( err );   // TypeError
}
```

Of course, we could have manually thrown an error with `throw ...` instead of causing an exception.

We can even `catch` the same error that we `throw(...)` into the generator, essentially giving the generator a chance to handle it but if it doesn't, the *iterator* code must handle it:

```
function *main() {
  var x = yield "Hello World";

  // never gets here
  console.log( x );
}

var it = main();

it.next();

try {
  // will `*main()` handle this error? we'll see!
  it.throw( "Oops" );
}
catch (err) {
  // nope, didn't handle it!
  console.error( err );           // Oops
}
```

Synchronous-looking error handling (via `try..catch`) with `async` code is a huge win for readability and reason-ability.

Generators + Promises

In our previous discussion, we showed how generators can be iterated asynchronously, which is a huge step forward in sequential reason-ability over the spaghetti mess of callbacks. But we lost something very important: the trustability and composability of Promises (see Chapter 3)!

Don't worry -- we can get that back. The best of all worlds in ES6 is to combine generators (synchronous-looking `async` code) with Promises (trustable and composable).

But how?

Recall from Chapter 3 the Promise-based approach to our running Ajax example:

```
function foo(x,y) {
  return request(
    "http://some.url.1/?x=" + x + "&y=" + y
  );
}

foo( 11, 31 )
.then(
  function(text){
    console.log( text );
  },
  function(err){
    console.error( err );
  }
);
```

In our earlier generator code for the running Ajax example, `foo(..)` returned nothing (`undefined`), and our *iterator* control code didn't care about that `yield` ed value.

But here the Promise-aware `foo(..)` returns a promise after making the Ajax call. That suggests that we could construct a promise with `foo(..)` and then `yield` it from the generator, and then the *iterator* control code would receive that promise.

But what should the *iterator* do with the promise?

It should listen for the promise to resolve (fulfillment or rejection), and then either resume the generator with the fulfillment message or throw an error into the generator with the rejection reason.

Let me repeat that, because it's so important. The natural way to get the most out of Promises and generators is **to yield a Promise**, and wire that Promise to control the generator's *iterator*.

Let's give it a try! First, we'll put the Promise-aware `foo(...)` together with the generator

```
*main() :  
  
function foo(x,y) {  
    return request(  
        "http://some.url.1/?x=" + x + "&y=" + y  
    );  
}  
  
function *main() {  
    try {  
        var text = yield foo( 11, 31 );  
        console.log( text );  
    }  
    catch (err) {  
        console.error( err );  
    }  
}
```

The most powerful revelation in this refactor is that the code inside `*main()` **did not have to change at all!** Inside the generator, whatever values are `yield`ed out is just an opaque implementation detail, so we're not even aware it's happening, nor do we need to worry about it.

But how are we going to run `*main()` now? We still have some of the implementation plumbing work to do, to receive and wire up the `yield`ed promise so that it resumes the generator upon resolution. We'll start by trying that manually:

```
var it = main();  
  
var p = it.next().value;  
  
// wait for the `p` promise to resolve  
p.then(  
    function(text){  
        it.next( text );  
    },  
    function(err){  
        it.throw( err );  
    }  
)
```

Actually, that wasn't so painful at all, was it?

This snippet should look very similar to what we did earlier with the manually wired generator controlled by the error-first callback. Instead of an `if (err) { it.throw...}`, the promise already splits fulfillment (success) and rejection (failure) for us, but otherwise the *iterator* control is identical.

Now, we've glossed over some important details.

Most importantly, we took advantage of the fact that we knew that `*main()` only had one Promise-aware step in it. What if we wanted to be able to Promise-drive a generator no matter how many steps it has? We certainly don't want to manually write out the Promise chain differently for each generator! What would be much nicer is if there was a way to repeat (aka "loop" over) the iteration control, and each time a Promise comes out, wait on its resolution before continuing.

Also, what if the generator throws out an error (intentionally or accidentally) during the `it.next(...)` call? Should we quit, or should we `catch` it and send it right back in?

Similarly, what if we `it.throw(...)` a Promise rejection into the generator, but it's not handled, and comes right back out?

Promise-Aware Generator Runner

The more you start to explore this path, the more you realize, "wow, it'd be great if there was just some utility to do it for me." And you're absolutely correct. This is such an important pattern, and you don't want to get it wrong (or exhaust yourself repeating it over and over), so your best bet is to use a utility that is specifically designed to *run* Promise-`yield`ing generators in the manner we've illustrated.

Several Promise abstraction libraries provide just such a utility, including my `asynquence` library and its `runner(...)`, which will be discussed in Appendix A of this book.

But for the sake of learning and illustration, let's just define our own standalone utility that we'll call `run(...)`:

```

// thanks to Benjamin Gruenbaum (@benjammingr on GitHub) for
// big improvements here!
function run(gen) {
  var args = [].slice.call( arguments, 1 ), it;

  // initialize the generator in the current context
  it = gen.apply( this, args );

  // return a promise for the generator completing
  return Promise.resolve()
    .then( function handleNext(value){
      // run to the next yielded value
      var next = it.next( value );

      return (function handleResult(next){
        // generator has completed running?
        if (next.done) {
          return next.value;
        }
        // otherwise keep going
        else {
          return Promise.resolve( next.value )
            .then(
              // resume the async loop on
              // success, sending the resolved
              // value back into the generator
              handleNext,
              // if `value` is a rejected
              // promise, propagate error back
              // into the generator for its own
              // error handling
              function handleErr(err) {
                return Promise.resolve(
                  it.throw( err )
                )
                .then( handleResult );
              }
            );
        }
      })(next);
    } );
}

```

As you can see, it's a quite a bit more complex than you'd probably want to author yourself, and you especially wouldn't want to repeat this code for each generator you use. So, a utility/library helper is definitely the way to go. Nevertheless, I encourage you to spend a few minutes studying that code listing to get a better sense of how to manage the generator+Promise negotiation.

How would you use `run(..)` with `*main()` in our *running* Ajax example?

```
function *main() {
  // ...
}

run( main );
```

That's it! The way we wired `run(..)`, it will automatically advance the generator you pass to it, asynchronously until completion.

Note: The `run(..)` we defined returns a promise which is wired to resolve once the generator is complete, or receive an uncaught exception if the generator doesn't handle it. We don't show that capability here, but we'll come back to it later in the chapter.

ES7: `async` and `await` ?

The preceding pattern -- generators yielding Promises that then control the generator's *iterator* to advance it to completion -- is such a powerful and useful approach, it would be nicer if we could do it without the clutter of the library utility helper (aka `run(..)`).

There's probably good news on that front. At the time of this writing, there's early but strong support for a proposal for more syntactic addition in this realm for the post-ES6, ES7-ish timeframe. Obviously, it's too early to guarantee the details, but there's a pretty decent chance it will shake out similar to the following:

```
function foo(x,y) {
  return request(
    "http://some.url.1/?x=" + x + "&y=" + y
  );
}

async function main() {
  try {
    var text = await foo( 11, 31 );
    console.log( text );
  }
  catch (err) {
    console.error( err );
  }
}

main();
```

As you can see, there's no `run(...)` call (meaning no need for a library utility!) to invoke and drive `main()` -- it's just called as a normal function. Also, `main()` isn't declared as a generator function anymore; it's a new kind of function: `async function`. And finally, instead of `yield`ing a Promise, we `await` for it to resolve.

The `async function` automatically knows what to do if you `await` a Promise -- it will pause the function (just like with generators) until the Promise resolves. We didn't illustrate it in this snippet, but calling an `async` function like `main()` automatically returns a promise that's resolved whenever the function finishes completely.

Tip: The `async / await` syntax should look very familiar to readers with experience in C#, because it's basically identical.

The proposal essentially codifies support for the pattern we've already derived, into a syntactic mechanism: combining Promises with sync-looking flow control code. That's the best of both worlds combined, to effectively address practically all of the major concerns we outlined with callbacks.

The mere fact that such a ES7-ish proposal already exists and has early support and enthusiasm is a major vote of confidence in the future importance of this `async` pattern.

Promise Concurrency in Generators

So far, all we've demonstrated is a single-step `async` flow with Promises+generators. But real-world code will often have many `async` steps.

If you're not careful, the sync-looking style of generators may lull you into complacency with how you structure your `async` concurrency, leading to suboptimal performance patterns. So we want to spend a little time exploring the options.

Imagine a scenario where you need to fetch data from two different sources, then combine those responses to make a third request, and finally print out the last response. We explored a similar scenario with Promises in Chapter 3, but let's reconsider it in the context of generators.

Your first instinct might be something like:

```
function *foo() {
  var r1 = yield request( "http://some.url.1" );
  var r2 = yield request( "http://some.url.2" );

  var r3 = yield request(
    "http://some.url.3/?v=" + r1 + "," + r2
  );

  console.log( r3 );
}

// use previously defined `run(..)` utility
run( foo );
```

This code will work, but in the specifics of our scenario, it's not optimal. Can you spot why?

Because the `r1` and `r2` requests can -- and for performance reasons, *should* -- run concurrently, but in this code they will run sequentially; the `"http://some.url.2"` URL isn't Ajax fetched until after the `"http://some.url.1"` request is finished. These two requests are independent, so the better performance approach would likely be to have them run at the same time.

But how exactly would you do that with a generator and `yield`? We know that `yield` is only a single pause point in the code, so you can't really do two pauses at the same time.

The most natural and effective answer is to base the async flow on Promises, specifically on their capability to manage state in a time-independent fashion (see "Future Value" in Chapter 3).

The simplest approach:

```

function *foo() {
  // make both requests "in parallel"
  var p1 = request( "http://some.url.1" );
  var p2 = request( "http://some.url.2" );

  // wait until both promises resolve
  var r1 = yield p1;
  var r2 = yield p2;

  var r3 = yield request(
    "http://some.url.3/?v=" + r1 + "," + r2
  );

  console.log( r3 );
}

// use previously defined `run(..)` utility
run( foo );

```

Why is this different from the previous snippet? Look at where the `yield` is and is not. `p1` and `p2` are promises for Ajax requests made concurrently (aka "in parallel"). It doesn't matter which one finishes first, because promises will hold onto their resolved state for as long as necessary.

Then we use two subsequent `yield` statements to wait for and retrieve the resolutions from the promises (into `r1` and `r2`, respectively). If `p1` resolves first, the `yield p1` resumes first then waits on the `yield p2` to resume. If `p2` resolves first, it will just patiently hold onto that resolution value until asked, but the `yield p1` will hold on first, until `p1` resolves.

Either way, both `p1` and `p2` will run concurrently, and both have to finish, in either order, before the `r3 = yield request...` Ajax request will be made.

If that flow control processing model sounds familiar, it's basically the same as what we identified in Chapter 3 as the "gate" pattern, enabled by the `Promise.all([..])` utility. So, we could also express the flow control like this:

```

function *foo() {
  // make both requests "in parallel," and
  // wait until both promises resolve
  var results = yield Promise.all( [
    request( "http://some.url.1" ),
    request( "http://some.url.2" )
  ] );

  var r1 = results[0];
  var r2 = results[1];

  var r3 = yield request(
    "http://some.url.3/?v=" + r1 + "," + r2
  );

  console.log( r3 );
}

// use previously defined `run(..)` utility
run( foo );

```

Note: As we discussed in Chapter 3, we can even use ES6 destructuring assignment to simplify the `var r1 = .. var r2 = ..` assignments, with `var [r1,r2] = results`.

In other words, all of the concurrency capabilities of Promises are available to us in the generator+Promise approach. So in any place where you need more than sequential this-then-that async flow control steps, Promises are likely your best bet.

Promises, Hidden

As a word of stylistic caution, be careful about how much Promise logic you include **inside your generators**. The whole point of using generators for asynchrony in the way we've described is to create simple, sequential, sync-looking code, and to hide as much of the details of asynchrony away from that code as possible.

For example, this might be a cleaner approach:

```
// note: normal function, not generator
function bar(url1,url2) {
  return Promise.all( [
    request( url1 ),
    request( url2 )
  ] );
}

function *foo() {
  // hide the Promise-based concurrency details
  // inside `bar(..)`
  var results = yield bar(
    "http://some.url.1",
    "http://some.url.2"
  );

  var r1 = results[0];
  var r2 = results[1];

  var r3 = yield request(
    "http://some.url.3/?v=" + r1 + "," + r2
  );

  console.log( r3 );
}

// use previously defined `run(..)` utility
run( foo );
```

Inside `*foo()`, it's cleaner and clearer that all we're doing is just asking `bar(..)` to get us some `results`, and we'll `yield`-wait on that to happen. We don't have to care that under the covers a `Promise.all([..])` Promise composition will be used to make that happen.

We treat asynchrony, and indeed Promises, as an implementation detail.

Hiding your Promise logic inside a function that you merely call from your generator is especially useful if you're going to do a sophisticated series flow-control. For example:

```
function bar() {
  Promise.all( [
    baz( .. )
    .then( .. ),
    Promise.race( [ .. ] )
  ] )
  .then( .. )
}
```

That kind of logic is sometimes required, and if you dump it directly inside your generator(s), you've defeated most of the reason why you would want to use generators in the first place. We *should* intentionally abstract such details away from our generator code so that they don't clutter up the higher level task expression.

Beyond creating code that is both functional and performant, you should also strive to make code that is as reasonable and maintainable as possible.

Note: Abstraction is not *always* a healthy thing for programming -- many times it can increase complexity in exchange for terseness. But in this case, I believe it's much healthier for your generator+Promise async code than the alternatives. As with all such advice, though, pay attention to your specific situations and make proper decisions for you and your team.

Generator Delegation

In the previous section, we showed calling regular functions from inside a generator, and how that remains a useful technique for abstracting away implementation details (like async Promise flow). But the main drawback of using a normal function for this task is that it has to behave by the normal function rules, which means it cannot pause itself with `yield` like a generator can.

It may then occur to you that you might try to call one generator from another generator, using our `run(..)` helper, such as:

```
function *foo() {
  var r2 = yield request( "http://some.url.2" );
  var r3 = yield request( "http://some.url.3/?v=" + r2 );

  return r3;
}

function *bar() {
  var r1 = yield request( "http://some.url.1" );

  // "delegating" to `*foo()` via `run(..)`
  var r3 = yield run( foo );

  console.log( r3 );
}

run( bar );
```

We run `*foo()` inside of `*bar()` by using our `run(..)` utility again. We take advantage here of the fact that the `run(..)` we defined earlier returns a promise which is resolved when its generator is run to completion (or errors out), so if we `yield` out to a `run(..)` instance the promise from another `run(..)` call, it automatically pauses `*bar()` until `*foo()` finishes.

But there's an even better way to integrate calling `*foo()` into `*bar()`, and it's called `yield`-delegation. The special syntax for `yield`-delegation is: `yield * __` (notice the extra `*`). Before we see it work in our previous example, let's look at a simpler scenario:

```
function *foo() {
  console.log(```*foo()` starting`);
  yield 3;
  yield 4;
  console.log(````*foo()` finished`);
}

function *bar() {
  yield 1;
  yield 2;
  yield *foo();    // `yield`-delegation!
  yield 5;
}

var it = bar();

it.next().value;    // 1
it.next().value;    // 2
it.next().value;    // ```*foo()` starting
                    // 3
it.next().value;    // 4
it.next().value;    // ```*foo()` finished
                    // 5
```

Note: Similar to a note earlier in the chapter where I explained why I prefer `function *foo()`... instead of `function* foo() ...`, I also prefer -- differing from most other documentation on the topic -- to say `yield *foo()` instead of `yield* foo()`. The placement of the `*` is purely stylistic and up to your best judgment. But I find the consistency of styling attractive.

How does the `yield *foo()` delegation work?

First, calling `foo()` creates an *iterator* exactly as we've already seen. Then, `yield *` delegates/transfers the *iterator* instance control (of the present `*bar()` generator) over to this other `*foo()` *iterator*.

So, the first two `it.next()` calls are controlling `*bar()`, but when we make the third `it.next()` call, now `*foo()` starts up, and now we're controlling `*foo()` instead of `*bar()`. That's why it's called delegation -- `*bar()` delegated its iteration control to `*foo()`.

As soon as the `it iterator` control exhausts the entire `*foo() iterator`, it automatically returns to controlling `*bar()`.

So now back to the previous example with the three sequential Ajax requests:

```
function *foo() {
  var r2 = yield request( "http://some.url.2" );
  var r3 = yield request( "http://some.url.3/?v=" + r2 );

  return r3;
}

function *bar() {
  var r1 = yield request( "http://some.url.1" );

  // "delegating" to `*foo()` via `yield*`
  var r3 = yield *foo();

  console.log( r3 );
}

run( bar );
```

The only difference between this snippet and the version used earlier is the use of `yield *foo()` instead of the previous `yield run(foo)`.

Note: `yield *` yields iteration control, not generator control; when you invoke the `*foo()` generator, you're now `yield`-delegating to its `iterator`. But you can actually `yield`-delegate to any `iterable`; `yield *[1,2,3]` would consume the default `iterator` for the `[1,2,3]` array value.

Why Delegation?

The purpose of `yield`-delegation is mostly code organization, and in that way is symmetrical with normal function calling.

Imagine two modules that respectively provide methods `foo()` and `bar()`, where `bar()` calls `foo()`. The reason the two are separate is generally because the proper organization of code for the program calls for them to be in separate functions. For example, there may be cases where `foo()` is called standalone, and other places where `bar()` calls `foo()`.

For all these exact same reasons, keeping generators separate aids in program readability, maintenance, and debuggability. In that respect, `yield *` is a syntactic shortcut for manually iterating over the steps of `*foo()` while inside of `*bar()`.

Such manual approach would be especially complex if the steps in `*foo()` were asynchronous, which is why you'd probably need to use that `run(..)` utility to do it. And as we've shown, `yield *foo()` eliminates the need for a sub-instance of the `run(..)` utility (like `run(foo)`).

Delegating Messages

You may wonder how this `yield`-delegation works not just with *iterator* control but with the two-way message passing. Carefully follow the flow of messages in and out, through the `yield`-delegation:

```

function *foo() {
  console.log( "inside `*foo()`:", yield "B" );

  console.log( "inside `*foo()`:", yield "C" );

  return "D";
}

function *bar() {
  console.log( "inside `*bar()`:", yield "A" );

  // `yield`-delegation!
  console.log( "inside `*bar()`:", yield *foo() );

  console.log( "inside `*bar()`:", yield "E" );

  return "F";
}

var it = bar();

console.log( "outside:", it.next().value );
// outside: A

console.log( "outside:", it.next( 1 ).value );
// inside `*bar()`: 1
// outside: B

console.log( "outside:", it.next( 2 ).value );
// inside `*foo()`: 2
// outside: C

console.log( "outside:", it.next( 3 ).value );
// inside `*foo()`: 3
// inside `*bar()`: D
// outside: E

console.log( "outside:", it.next( 4 ).value );
// inside `*bar()`: 4
// outside: F

```

Pay particular attention to the processing steps after the `it.next(3)` call:

1. The `3` value is passed (through the `yield`-delegation in `*bar()`) into the waiting `yield "C"` expression inside of `*foo()`.
2. `*foo()` then calls `return "D"`, but this value doesn't get returned all the way back to the outside `it.next(3)` call.
3. Instead, the `"D"` value is sent as the result of the waiting `yield *foo()` expression inside of `*bar()` -- this `yield`-delegation expression has essentially been paused

while all of `*foo()` was exhausted. So `"D"` ends up inside of `*bar()` for it to print out.

4. `yield "E"` is called inside of `*bar()`, and the `"E"` value is yielded to the outside as the result of the `it.next(3)` call.

From the perspective of the external `iterator` (`it`), it doesn't appear any differently between controlling the initial generator or a delegated one.

In fact, `yield`-delegation doesn't even have to be directed to another generator; it can just be directed to a non-generator, general `iterable`. For example:

```
function *bar() {
  console.log(`inside `*bar()``, yield "A" );
  // `yield`-delegation to a non-generator!
  console.log(`inside `*bar()``, yield *[ "B", "C", "D" ] );
  console.log(`inside `*bar()``, yield "E" );
  return "F";
}

var it = bar();

console.log(`outside:`, it.next().value );
// outside: A

console.log(`outside:`, it.next( 1 ).value );
// inside `*bar()``: 1
// outside: B

console.log(`outside:`, it.next( 2 ).value );
// outside: C

console.log(`outside:`, it.next( 3 ).value );
// outside: D

console.log(`outside:`, it.next( 4 ).value );
// inside `*bar()``: undefined
// outside: E

console.log(`outside:`, it.next( 5 ).value );
// inside `*bar()``: 5
// outside: F
```

Notice the differences in where the messages were received/reported between this example and the one previous.

Most strikingly, the default `array iterator` doesn't care about any messages sent in via `next(...)` calls, so the values `2`, `3`, and `4` are essentially ignored. Also, because that `iterator` has no explicit `return` value (unlike the previously used `*foo()`), the `yield *` expression gets an `undefined` when it finishes.

Exceptions Delegated, Too!

In the same way that `yield`-delegation transparently passes messages through in both directions, errors/exceptions also pass in both directions:

```
function *foo() {
  try {
    yield "B";
  }
  catch (err) {
    console.log( "error caught inside `*foo()`:", err );
  }

  yield "C";

  throw "D";
}

function *bar() {
  yield "A";

  try {
    yield *foo();
  }
  catch (err) {
    console.log( "error caught inside `*bar()`:", err );
  }

  yield "E";

  yield *baz();

  // note: can't get here!
  yield "G";
}

function *baz() {
  throw "F";
}

var it = bar();

console.log( "outside:", it.next().value );
// outside: A
```

```

console.log( "outside:", it.next( 1 ).value );
// outside: B

console.log( "outside:", it.throw( 2 ).value );
// error caught inside `*foo()`: 2
// outside: C

console.log( "outside:", it.next( 3 ).value );
// error caught inside `*bar()`: D
// outside: E

try {
  console.log( "outside:", it.next( 4 ).value );
}
catch (err) {
  console.log( "error caught outside:", err );
}
// error caught outside: F

```

Some things to note from this snippet:

1. When we call `it.throw(2)`, it sends the error message `2` into `*bar()`, which delegates that to `*foo()`, which then `catches` it and handles it gracefully. Then, the `yield "C"` sends `"C"` back out as the return `value` from the `it.throw(2)` call.
2. The `"D"` value that's next `thrown` from inside `*foo()` propagates out to `*bar()`, which `catches` it and handles it gracefully. Then the `yield "E"` sends `"E"` back out as the return `value` from the `it.next(3)` call.
3. Next, the exception `thrown` from `*baz()` isn't caught in `*bar()` -- though we did `catch` it outside -- so both `*baz()` and `*bar()` are set to a completed state. After this snippet, you would not be able to get the `"G"` value out with any subsequent `next(...)` call(s) -- they will just return `undefined` for `value`.

Delegating Asynchrony

Let's finally get back to our earlier `yield`-delegation example with the multiple sequential Ajax requests:

```

function *foo() {
  var r2 = yield request( "http://some.url.2" );
  var r3 = yield request( "http://some.url.3/?v=" + r2 );

  return r3;
}

function *bar() {
  var r1 = yield request( "http://some.url.1" );

  var r3 = yield *foo();

  console.log( r3 );
}

run( bar );

```

Instead of calling `yield run(foo)` inside of `*bar()`, we just call `yield *foo()`.

In the previous version of this example, the Promise mechanism (controlled by `run(..)`) was used to transport the value from `return r3` in `*foo()` to the local variable `r3` inside `*bar()`. Now, that value is just returned back directly via the `yield *` mechanics.

Otherwise, the behavior is pretty much identical.

Delegating "Recursion"

Of course, `yield`-delegation can keep following as many delegation steps as you wire up. You could even use `yield`-delegation for async-capable generator "recursion" -- a generator `yield`-delegating to itself:

```

function *foo(val) {
  if (val > 1) {
    // generator recursion
    val = yield *foo( val - 1 );
  }

  return yield request( "http://some.url/?v=" + val );
}

function *bar() {
  var r1 = yield *foo( 3 );
  console.log( r1 );
}

run( bar );

```

Note: Our `run(..)` utility could have been called with `run(foo, 3)`, because it supports additional parameters being passed along to the initialization of the generator. However, we used a parameter-free `*bar()` here to highlight the flexibility of `yield *`.

What processing steps follow from that code? Hang on, this is going to be quite intricate to describe in detail:

1. `run(bar)` starts up the `*bar()` generator.
2. `foo(3)` creates an *iterator* for `*foo(..)` and passes `3` as its `val` parameter.
3. Because `3 > 1`, `foo(2)` creates another *iterator* and passes in `2` as its `val` parameter.
4. Because `2 > 1`, `foo(1)` creates yet another *iterator* and passes in `1` as its `val` parameter.
5. `1 > 1` is `false`, so we next call `request(..)` with the `1` value, and get a promise back for that first Ajax call.
6. That promise is `yield` ed out, which comes back to the `*foo(2)` generator instance.
7. The `yield *` passes that promise back out to the `*foo(3)` generator instance. Another `yield *` passes the promise out to the `*bar()` generator instance. And yet again another `yield *` passes the promise out to the `run(..)` utility, which will wait on that promise (for the first Ajax request) to proceed.
8. When the promise resolves, its fulfillment message is sent to resume `*bar()`, which passes through the `yield *` into the `*foo(3)` instance, which then passes through the `yield *` to the `*foo(2)` generator instance, which then passes through the `yield *` to the normal `yield` that's waiting in the `*foo(3)` generator instance.
9. That first call's Ajax response is now immediately `return` ed from the `*foo(3)` generator instance, which sends that value back as the result of the `yield *` expression in the `*foo(2)` instance, and assigned to its local `val` variable.
10. Inside `*foo(2)`, a second Ajax request is made with `request(..)`, whose promise is `yield` ed back to the `*foo(1)` instance, and then `yield *` propagates all the way out to `run(..)` (step 7 again). When the promise resolves, the second Ajax response propagates all the way back into the `*foo(2)` generator instance, and is assigned to its local `val` variable.
11. Finally, the third Ajax request is made with `request(..)`, its promise goes out to `run(..)`, and then its resolution value comes all the way back, which is then `return` ed so that it comes back to the waiting `yield *` expression in `*bar()`.

Phew! A lot of crazy mental juggling, huh? You might want to read through that a few more times, and then go grab a snack to clear your head!

Generator Concurrency

As we discussed in both Chapter 1 and earlier in this chapter, two simultaneously running "processes" can cooperatively interleave their operations, and many times this can *yield* (pun intended) very powerful asynchrony expressions.

Frankly, our earlier examples of concurrency interleaving of multiple generators showed how to make it really confusing. But we hinted that there's places where this capability is quite useful.

Recall a scenario we looked at in Chapter 1, where two different simultaneous Ajax response handlers needed to coordinate with each other to make sure that the data communication was not a race condition. We slotted the responses into the `res` array like this:

```
function response(data) {
  if (data.url == "http://some.url.1") {
    res[0] = data;
  }
  else if (data.url == "http://some.url.2") {
    res[1] = data;
  }
}
```

But how can we use multiple generators concurrently for this scenario?

```
// `request(..)` is a Promise-aware Ajax utility

var res = [];

function *reqData(url) {
  res.push(
    yield request( url )
  );
}
```

Note: We're going to use two instances of the `*reqData(..)` generator here, but there's no difference to running a single instance of two different generators; both approaches are reasoned about identically. We'll see two different generators coordinating in just a bit.

Instead of having to manually sort out `res[0]` and `res[1]` assignments, we'll use coordinated ordering so that `res.push(..)` properly slots the values in the expected and predictable order. The expressed logic thus should feel a bit cleaner.

But how will we actually orchestrate this interaction? First, let's just do it manually, with Promises:

```
var it1 = reqData( "http://some.url.1" );
var it2 = reqData( "http://some.url.2" );

var p1 = it1.next().value;
var p2 = it2.next().value;

p1
.then( function(data){
  it1.next( data );
  return p2;
} )
.then( function(data){
  it2.next( data );
} );
```

*reqData(..)'s two instances are both started to make their Ajax requests, then paused with `yield`. Then we choose to resume the first instance when `p1` resolves, and then `p2`'s resolution will restart the second instance. In this way, we use Promise orchestration to ensure that `res[0]` will have the first response and `res[1]` will have the second response.

But frankly, this is awfully manual, and it doesn't really let the generators orchestrate themselves, which is where the true power can lie. Let's try it a different way:

```
// `request(..)` is a Promise-aware Ajax utility

var res = [];

function *reqData(url) {
  var data = yield request( url );

  // transfer control
  yield;

  res.push( data );
}

var it1 = reqData( "http://some.url.1" );
var it2 = reqData( "http://some.url.2" );

var p1 = it1.next().value;
var p2 = it2.next().value;

p1.then( function(data){
  it1.next( data );
} );

p2.then( function(data){
  it2.next( data );
} );

Promise.all( [p1,p2] )
.then( function(){
  it1.next();
  it2.next();
} );

```

OK, this is a bit better (though still manual!), because now the two instances of `*reqData(..)` run truly concurrently, and (at least for the first part) independently.

In the previous snippet, the second instance was not given its data until after the first instance was totally finished. But here, both instances receive their data as soon as their respective responses come back, and then each instance does another `yield` for control transfer purposes. We then choose what order to resume them in the `Promise.all([..])` handler.

What may not be as obvious is that this approach hints at an easier form for a reusable utility, because of the symmetry. We can do even better. Let's imagine using a utility called `runAll(..)`:

```
// `request(..)` is a Promise-aware Ajax utility

var res = [];

runAll(
  function*(){
    var p1 = request( "http://some.url.1" );

    // transfer control
    yield;

    res.push( yield p1 );
  },
  function*(){
    var p2 = request( "http://some.url.2" );

    // transfer control
    yield;

    res.push( yield p2 );
  }
);

```

Note: We're not including a code listing for `runAll(..)` as it is not only long enough to bog down the text, but is an extension of the logic we've already implemented in `run(..)` earlier. So, as a good supplementary exercise for the reader, try your hand at evolving the code from `run(..)` to work like the imagined `runAll(..)`. Also, my `asynquence` library provides a previously mentioned `runner(..)` utility with this kind of capability already built in, and will be discussed in Appendix A of this book.

Here's how the processing inside `runAll(..)` would operate:

1. The first generator gets a promise for the first Ajax response from `"http://some.url.1"`, then `yield`s control back to the `runAll(..)` utility.
2. The second generator runs and does the same for `"http://some.url.2"`, `yield`ing control back to the `runAll(..)` utility.
3. The first generator resumes, and then `yield`s out its promise `p1`. The `runAll(..)` utility does the same in this case as our previous `run(..)`, in that it waits on that promise to resolve, then resumes the same generator (no control transfer!). When `p1` resolves, `runAll(..)` resumes the first generator again with that resolution value, and then `res[0]` is given its value. When the first generator then finishes, that's an implicit transfer of control.
4. The second generator resumes, `yield`s out its promise `p2`, and waits for it to resolve. Once it does, `runAll(..)` resumes the second generator with that value, and `res[1]` is set.

In this running example, we use an outer variable called `res` to store the results of the two different Ajax responses -- that's our concurrency coordination making that possible.

But it might be quite helpful to further extend `runAll(..)` to provide an inner variable space for the multiple generator instances to *share*, such as an empty object we'll call `data` below. Also, it could take non-Promise values that are `yield`ed and hand them off to the next generator.

Consider:

```
// `request(..)` is a Promise-aware Ajax utility

runAll(
  function*(data){
    data.res = [];

    // transfer control (and message pass)
    var url1 = yield "http://some.url.2";

    var p1 = request( url1 ); // "http://some.url.1"

    // transfer control
    yield;

    data.res.push( yield p1 );
  },
  function*(data){
    // transfer control (and message pass)
    var url2 = yield "http://some.url.1";

    var p2 = request( url2 ); // "http://some.url.2"

    // transfer control
    yield;

    data.res.push( yield p2 );
  }
);
```

In this formulation, the two generators are not just coordinating control transfer, but actually communicating with each other, both through `data.res` and the `yield`ed messages that trade `url1` and `url2` values. That's incredibly powerful!

Such realization also serves as a conceptual base for a more sophisticated asynchrony technique called CSP (Communicating Sequential Processes), which we will cover in Appendix B of this book.

Thunks

So far, we've made the assumption that `yield`ing a Promise from a generator -- and having that Promise resume the generator via a helper utility like `run(...)` -- was the best possible way to manage asynchrony with generators. To be clear, it is.

But we skipped over another pattern that has some mildly widespread adoption, so in the interest of completeness we'll take a brief look at it.

In general computer science, there's an old pre-JS concept called a "thunk." Without getting bogged down in the historical nature, a narrow expression of a thunk in JS is a function that -- without any parameters -- is wired to call another function.

In other words, you wrap a function definition around function call -- with any parameters it needs -- to *defer* the execution of that call, and that wrapping function is a thunk. When you later execute the thunk, you end up calling the original function.

For example:

```
function foo(x,y) {
  return x + y;
}

function fooThunk() {
  return foo( 3, 4 );
}

// later

console.log( fooThunk() );    // 7
```

So, a synchronous thunk is pretty straightforward. But what about an async thunk? We can essentially extend the narrow thunk definition to include it receiving a callback.

Consider:

```

function foo(x,y,cb) {
  setTimeout( function(){
    cb( x + y );
  }, 1000 );
}

function fooThunk(cb) {
  foo( 3, 4, cb );
}

// later

fooThunk( function(sum){
  console.log( sum );           // 7
} );

```

As you can see, `fooThunk(..)` only expects a `cb(..)` parameter, as it already has values `3` and `4` (for `x` and `y`, respectively) pre-specified and ready to pass to `foo(..)`. A thunk is just waiting around patiently for the last piece it needs to do its job: the callback.

You don't want to make thunks manually, though. So, let's invent a utility that does this wrapping for us.

Consider:

```

function thunkify(fn) {
  var args = [].slice.call( arguments, 1 );
  return function(cb) {
    args.push( cb );
    return fn.apply( null, args );
  };
}

var fooThunk = thunkify( foo, 3, 4 );

// later

fooThunk( function(sum) {
  console.log( sum );           // 7
} );

```

Tip: Here we assume that the original (`foo(..)`) function signature expects its callback in the last position, with any other parameters coming before it. This is a pretty ubiquitous "standard" for async JS function standards. You might call it "callback-last style." If for some reason you had a need to handle "callback-first style" signatures, you would just make a utility that used `args.unshift(..)` instead of `args.push(..)`.

The preceding formulation of `thunkify(...)` takes both the `foo(...)` function reference, and any parameters it needs, and returns back the thunk itself (`fooThunk(...)`). However, that's not the typical approach you'll find to thunks in JS.

Instead of `thunkify(...)` making the thunk itself, typically -- if not perplexingly -- the `thunkify(...)` utility would produce a function that produces thunks.

Uhhhh... yeah.

Consider:

```
function thunkify(fn) {
  return function() {
    var args = [].slice.call( arguments );
    return function(cb) {
      args.push( cb );
      return fn.apply( null, args );
    };
  };
}
```

The main difference here is the extra `return function() { .. }` layer. Here's how its usage differs:

```
var whatIsThis = thunkify( foo );

var fooThunk = whatIsThis( 3, 4 );

// later

fooThunk( function(sum) {
  console.log( sum );           // 7
} );
```

Obviously, the big question this snippet implies is what is `whatIsThis` properly called? It's not the thunk, it's the thing that will produce thunks from `foo(...)` calls. It's kind of like a "factory" for "thunks." There doesn't seem to be any kind of standard agreement for naming such a thing.

So, my proposal is "thunkory" ("thunk" + "factory"). So, `thunkify(...)` produces a thunkory, and a thunkory produces thunks. That reasoning is symmetric to my proposal for "promisory" in Chapter 3:

```

var fooThunkory = thunkify( foo );

var fooThunk1 = fooThunkory( 3, 4 );
var fooThunk2 = fooThunkory( 5, 6 );

// later

fooThunk1( function(sum) {
    console.log( sum );           // 7
} );

fooThunk2( function(sum) {
    console.log( sum );           // 11
} );

```

Note: The running `foo(..)` example expects a style of callback that's not "error-first style." Of course, "error-first style" is much more common. If `foo(..)` had some sort of legitimate error-producing expectation, we could change it to expect and use an error-first callback. None of the subsequent `thunkify(..)` machinery cares what style of callback is assumed. The only difference in usage would be `fooThunk1(function(err,sum){...})`.

Exposing the thunkory method -- instead of how the earlier `thunkify(..)` hides this intermediary step -- may seem like unnecessary complication. But in general, it's quite useful to make thunkories at the beginning of your program to wrap existing API methods, and then be able to pass around and call those thunkories when you need thunks. The two distinct steps preserve a cleaner separation of capability.

To illustrate:

```

// cleaner:
var fooThunkory = thunkify( foo );

var fooThunk1 = fooThunkory( 3, 4 );
var fooThunk2 = fooThunkory( 5, 6 );

// instead of:
var fooThunk1 = thunkify( foo, 3, 4 );
var fooThunk2 = thunkify( foo, 5, 6 );

```

Regardless of whether you like to deal with the thunkories explicitly or not, the usage of thunks `fooThunk1(..)` and `fooThunk2(..)` remains the same.

s/promise/thunk/

So what's all this thunk stuff have to do with generators?

Comparing thunks to promises generally: they're not directly interchangeable as they're not equivalent in behavior. Promises are vastly more capable and trustable than bare thunks.

But in another sense, they both can be seen as a request for a value, which may be async in its answering.

Recall from Chapter 3 we defined a utility for promisifying a function, which we called

`Promise.wrap(...)` -- we could have called it `promisify(...)`, too! This Promise-wrapping utility doesn't produce Promises; it produces promisories that in turn produce Promises. This is completely symmetric to the thunkories and thunks presently being discussed.

To illustrate the symmetry, let's first alter the running `foo(..)` example from earlier to assume an "error-first style" callback:

```
function foo(x,y,cb) {
  setTimeout( function(){
    // assume `cb(..)` as "error-first style"
    cb( null, x + y );
  }, 1000 );
}
```

Now, we'll compare using `thunkify(..)` and `promisify(..)` (aka `Promise.wrap(..)` from Chapter 3):

```
// symmetrical: constructing the question asker
var fooThunkory = thunkify( foo );
var fooPromisory = promisify( foo );

// symmetrical: asking the question
var fooThunk = fooThunkory( 3, 4 );
var fooPromise = fooPromisory( 3, 4 );

// get the thunk answer
fooThunk( function(err,sum){
  if (err) {
    console.error( err );
  }
  else {
    console.log( sum );           // 7
  }
} );

// get the promise answer
fooPromise
.then(
  function(sum){
    console.log( sum );           // 7
  },
  function(err){
    console.error( err );
  }
);

```

Both the thunkory and the promisory are essentially asking a question (for a value), and respectively the thunk `fooThunk` and promise `fooPromise` represent the future answers to that question. Presented in that light, the symmetry is clear.

With that perspective in mind, we can see that generators which `yield` Promises for asynchrony could instead `yield` thunks for asynchrony. All we'd need is a smarter `run(...)` utility (like from before) that can not only look for and wire up to a `yield` ed Promise but also to provide a callback to a `yield` ed thunk.

Consider:

```
function *foo() {
  var val = yield request( "http://some.url.1" );
  console.log( val );
}

run( foo );
```

In this example, `request(..)` could either be a promisory that returns a promise, or a thunkory that returns a thunk. From the perspective of what's going on inside the generator code logic, we don't care about that implementation detail, which is quite powerful!

So, `request(..)` could be either:

```
// promisory `request(..)` (see Chapter 3)
var request = Promise.wrap( ajax );

// vs.

// thunkory `request(..)`
var request = thunkify( ajax );
```

Finally, as a thunk-aware patch to our earlier `run(..)` utility, we would need logic like this:

```
// ...
// did we receive a thunk back?
else if (typeof next.value == "function") {
    return new Promise( function(resolve,reject){
        // call the thunk with an error-first callback
        next.value( function(err,msg) {
            if (err) {
                reject( err );
            }
            else {
                resolve( msg );
            }
        } );
    } )
    .then(
        handleNext,
        function handleErr(err) {
            return Promise.resolve(
                it.throw( err )
            )
            .then( handleResult );
        }
    );
}
```

Now, our generators can either call promisories to `yield` Promises, or call thunkories to `yield` thunks, and in either case, `run(..)` would handle that value and use it to wait for the completion to resume the generator.

Symmetry wise, these two approaches look identical. However, we should point out that's true only from the perspective of Promises or thunks representing the future value continuation of a generator.

From the larger perspective, thunks do not in and of themselves have hardly any of the trustability or composable guarantees that Promises are designed with. Using a thunk as a stand-in for a Promise in this particular generator asynchrony pattern is workable but should be seen as less than ideal when compared to all the benefits that Promises offer (see Chapter 3).

If you have the option, prefer `yield pr` rather than `yield th`. But there's nothing wrong with having a `run(...)` utility which can handle both value types.

Note: The `runner(...)` utility in my `asynquence` library, which will be discussed in Appendix A, handles `yield`s of Promises, thunks and `asynquence` sequences.

Pre-ES6 Generators

You're hopefully convinced now that generators are a very important addition to the async programming toolbox. But it's a new syntax in ES6, which means you can't just polyfill generators like you can Promises (which are just a new API). So what can we do to bring generators to our browser JS if we don't have the luxury of ignoring pre-ES6 browsers?

For all new syntax extensions in ES6, there are tools -- the most common term for them is transpilers, for trans-compilers -- which can take your ES6 syntax and transform it into equivalent (but obviously uglier!) pre-ES6 code. So, generators can be transpiled into code that will have the same behavior but work in ES5 and below.

But how? The "magic" of `yield` doesn't obviously sound like code that's easy to transpile. We actually hinted at a solution in our earlier discussion of closure-based *iterators*.

Manual Transformation

Before we discuss the transpilers, let's derive how manual transpilation would work in the case of generators. This isn't just an academic exercise, because doing so will actually help further reinforce how they work.

Consider:

```
// `request(..)` is a Promise-aware Ajax utility

function *foo(url) {
  try {
    console.log("requesting:", url);
    var val = yield request(url);
    console.log(val);
  }
  catch (err) {
    console.log("Oops:", err);
    return false;
  }
}

var it = foo("http://some.url.1");
```

The first thing to observe is that we'll still need a normal `foo()` function that can be called, and it will still need to return an *iterator*. So, let's sketch out the non-generator transformation:

```
function foo(url) {

  // ...

  // make and return an iterator
  return {
    next: function(v) {
      // ...
    },
    throw: function(e) {
      // ...
    }
  };
}

var it = foo("http://some.url.1");
```

The next thing to observe is that a generator does its "magic" by suspending its scope/state, but we can emulate that with function closure (see the *Scope & Closures* title of this series). To understand how to write such code, we'll first annotate different parts of our generator with state values:

```
// `request(..)` is a Promise-aware Ajax utility

function *foo(url) {
  // STATE *1*

  try {
    console.log("requesting:", url);
    var TMP1 = request(url);

    // STATE *2*
    var val = yield TMP1;
    console.log(val);
  }
  catch (err) {
    // STATE *3*
    console.log("Oops:", err);
    return false;
  }
}
```

Note: For more accurate illustration, we split up the `val = yield request..` statement into two parts, using the temporary `TMP1` variable. `request(..)` happens in state `*1*`, and the assignment of its completion value to `val` happens in state `*2*`. We'll get rid of that intermediate `TMP1` when we convert the code to its non-generator equivalent.

In other words, `*1*` is the beginning state, `*2*` is the state if the `request(..)` succeeds, and `*3*` is the state if the `request(..)` fails. You can probably imagine how any extra `yield` steps would just be encoded as extra states.

Back to our transpiled generator, let's define a variable `state` in the closure we can use to keep track of the state:

```
function foo(url) {
  // manage generator state
  var state;

  // ...
}
```

Now, let's define an inner function called `process(..)` inside the closure which handles each state, using a `switch` statement:

```
// `request(..)` is a Promise-aware Ajax utility

function foo(url) {
  // manage generator state
  var state;

  // generator-wide variable declarations
  var val;

  function process(v) {
    switch (state) {
      case 1:
        console.log("requesting:", url);
        return request(url);
      case 2:
        val = v;
        console.log(val);
        return;
      case 3:
        var err = v;
        console.log("Oops:", err);
        return false;
    }
  }

  // ...
}

// ...
}
```

Each state in our generator is represented by its own `case` in the `switch` statement. `process(..)` will be called each time we need to process a new state. We'll come back to how that works in just a moment.

For any generator-wide variable declarations (`val`), we move those to a `var` declaration outside of `process(..)` so they can survive multiple calls to `process(..)`. But the "block scoped" `err` variable is only needed for the `*3*` state, so we leave it in place.

In state `*1*`, instead of `yield request(..)`, we did `return request(..)`. In terminal state `*2*`, there was no explicit `return`, so we just do a `return;` which is the same as `return undefined`. In terminal state `*3*`, there was a `return false`, so we preserve that.

Now we need to define the code in the *iterator* functions so they call `process(..)` appropriately:

```
function foo(url) {
  // manage generator state
  var state;

  // generator-wide variable declarations
  var val;
```

```

function process(v) {
    switch (state) {
        case 1:
            console.log( "requesting:", url );
            return request( url );
        case 2:
            val = v;
            console.log( val );
            return;
        case 3:
            var err = v;
            console.log( "Oops:", err );
            return false;
    }
}

// make and return an iterator
return {
    next: function(v) {
        // initial state
        if (!state) {
            state = 1;
            return {
                done: false,
                value: process()
            };
        }
        // yield resumed successfully
        else if (state == 1) {
            state = 2;
            return {
                done: true,
                value: process( v )
            };
        }
        // generator already completed
        else {
            return {
                done: true,
                value: undefined
            };
        }
    },
    "throw": function(e) {
        // the only explicit error handling is in
        // state *1*
        if (state == 1) {
            state = 3;
            return {
                done: true,
                value: process( e )
            };
        }
    }
}

```

```

        }
        // otherwise, an error won't be handled,
        // so just throw it right back out
        else {
            throw e;
        }
    );
}
}

```

How does this code work?

1. The first call to the *iterator's* `next()` call would move the generator from the uninitialized state to state `1`, and then call `process()` to handle that state. The return value from `request(..)`, which is the promise for the Ajax response, is returned back as the `value` property from the `next()` call.
2. If the Ajax request succeeds, the second call to `next(..)` should send in the Ajax response value, which moves our state to `2`. `process(..)` is again called (this time with the passed in Ajax response value), and the `value` property returned from `next(..)` will be `undefined`.
3. However, if the Ajax request fails, `throw(..)` should be called with the error, which would move the state from `1` to `3` (instead of `2`). Again `process(..)` is called, this time with the error value. That `case` returns `false`, which is set as the `value` property returned from the `throw(..)` call.

From the outside -- that is, interacting only with the *iterator* -- this `foo(..)` normal function works pretty much the same as the `*foo(..)` generator would have worked. So we've effectively "transpiled" our ES6 generator to pre-ES6 compatibility!

We could then manually instantiate our generator and control its iterator -- calling `var it = foo(..)` and `it.next(..)` and such -- or better, we could pass it to our previously defined `run(..)` utility as `run(foo, "..")`.

Automatic Transpilation

The preceding exercise of manually deriving a transformation of our ES6 generator to pre-ES6 equivalent teaches us how generators work conceptually. But that transformation was really intricate and very non-portable to other generators in our code. It would be quite impractical to do this work by hand, and would completely obviate all the benefit of generators.

But luckily, several tools already exist that can automatically convert ES6 generators to things like what we derived in the previous section. Not only do they do the heavy lifting work for us, but they also handle several complications that we glossed over.

One such tool is regenerator (<https://facebook.github.io/regenerator/>), from the smart folks at Facebook.

If we use regenerator to transpile our previous generator, here's the code produced (at the time of this writing):

```
// `request(..)` is a Promise-aware Ajax utility

var foo = regeneratorRuntime.mark(function foo(url) {
    var val;

    return regeneratorRuntime.wrap(function foo$(context$1$0) {
        while (1) switch (context$1$0.prev = context$1$0.next) {
            case 0:
                context$1$0.prev = 0;
                console.log("requesting:", url);
                context$1$0.next = 4;
                return request(url);
            case 4:
                val = context$1$0.sent;
                console.log(val);
                context$1$0.next = 12;
                break;
            case 8:
                context$1$0.prev = 8;
                context$1$0.t0 = context$1$0.catch(0);
                console.log("Oops:", context$1$0.t0);
                return context$1$0.abrupt("return", false);
            case 12:
            case "end":
                return context$1$0.stop();
        }
    }, foo, this, [[0, 8]]);
});
```

There's some obvious similarities here to our manual derivation, such as the `switch` / `case` statements, and we even see `val` pulled out of the closure just as we did.

Of course, one trade-off is that regenerator's transpilation requires a helper library `regeneratorRuntime` that holds all the reusable logic for managing a general generator / iterator. A lot of that boilerplate looks different than our version, but even then, the concepts can be seen, like with `context$1$0.next = 4` keeping track of the next state for the generator.

The main takeaway is that generators are not restricted to only being useful in ES6+ environments. Once you understand the concepts, you can employ them throughout your code, and use tools to transform the code to be compatible with older environments.

This is more work than just using a `Promise` API polyfill for pre-ES6 Promises, but the effort is totally worth it, because generators are so much better at expressing async flow control in a reasonable, sensible, synchronous-looking, sequential fashion.

Once you get hooked on generators, you'll never want to go back to the hell of async spaghetti callbacks!

Review

Generators are a new ES6 function type that does not run-to-completion like normal functions. Instead, the generator can be paused in mid-completion (entirely preserving its state), and it can later be resumed from where it left off.

This pause/resume interchange is cooperative rather than preemptive, which means that the generator has the sole capability to pause itself, using the `yield` keyword, and yet the *iterator* that controls the generator has the sole capability (via `next(...)`) to resume the generator.

The `yield` / `next(..)` duality is not just a control mechanism, it's actually a two-way message passing mechanism. A `yield ..` expression essentially pauses waiting for a value, and the next `next(..)` call passes a value (or implicit `undefined`) back to that paused `yield` expression.

The key benefit of generators related to async flow control is that the code inside a generator expresses a sequence of steps for the task in a naturally sync/sequential fashion. The trick is that we essentially hide potential asynchrony behind the `yield` keyword -- moving the asynchrony to the code where the generator's *iterator* is controlled.

In other words, generators preserve a sequential, synchronous, blocking code pattern for async code, which lets our brains reason about the code much more naturally, addressing one of the two key drawbacks of callback-based async.

This book so far has been all about how to leverage asynchrony patterns more effectively. But we haven't directly addressed why asynchrony really matters to JS. The most obvious explicit reason is **performance**.

For example, if you have two Ajax requests to make, and they're independent, but you need to wait on them both to finish before doing the next task, you have two options for modeling that interaction: serial and concurrent.

You could make the first request and wait to start the second request until the first finishes. Or, as we've seen both with promises and generators, you could make both requests "in parallel," and express the "gate" to wait on both of them before moving on.

Clearly, the latter is usually going to be more performant than the former. And better performance generally leads to better user experience.

It's even possible that asynchrony (interleaved concurrency) can improve just the perception of performance, even if the overall program still takes the same amount of time to complete. User perception of performance is every bit -- if not more! -- as important as actual measurable performance.

We want to now move beyond localized asynchrony patterns to talk about some bigger picture performance details at the program level.

Note: You may be wondering about micro-performance issues like if `a++` or `++a` is faster. We'll look at those sorts of performance details in the next chapter on "Benchmarking & Tuning."

Web Workers

If you have processing-intensive tasks but you don't want them to run on the main thread (which may slow down the browser/UI), you might have wished that JavaScript could operate in a multithreaded manner.

In Chapter 1, we talked in detail about how JavaScript is single threaded. And that's still true. But a single thread isn't the only way to organize the execution of your program.

Imagine splitting your program into two pieces, and running one of those pieces on the main UI thread, and running the other piece on an entirely separate thread.

What kinds of concerns would such an architecture bring up?

For one, you'd want to know if running on a separate thread meant that it ran in parallel (on systems with multiple CPUs/cores) such that a long-running process on that second thread would **not** block the main program thread. Otherwise, "virtual threading" wouldn't be of much

benefit over what we already have in JS with async concurrency.

And you'd want to know if these two pieces of the program have access to the same shared scope/resources. If they do, then you have all the questions that multithreaded languages (Java, C++, etc.) deal with, such as needing cooperative or preemptive locking (mutexes, etc.). That's a lot of extra work, and shouldn't be undertaken lightly.

Alternatively, you'd want to know how these two pieces could "communicate" if they couldn't share scope/resources.

All these are great questions to consider as we explore a feature added to the web platform circa HTML5 called "Web Workers." This is a feature of the browser (aka host environment) and actually has almost nothing to do with the JS language itself. That is, JavaScript does not *currently* have any features that support threaded execution.

But an environment like your browser can easily provide multiple instances of the JavaScript engine, each on its own thread, and let you run a different program in each thread. Each of those separate threaded pieces of your program is called a "(Web) Worker." This type of parallelism is called "task parallelism," as the emphasis is on splitting up chunks of your program to run in parallel.

From your main JS program (or another Worker), you instantiate a Worker like so:

```
var w1 = new Worker( "http://some.url.1/mycoolworker.js" );
```

The URL should point to the location of a JS file (not an HTML page!) which is intended to be loaded into a Worker. The browser will then spin up a separate thread and let that file run as an independent program in that thread.

Note: The kind of Worker created with such a URL is called a "Dedicated Worker." But instead of providing a URL to an external file, you can also create an "Inline Worker" by providing a Blob URL (another HTML5 feature); essentially it's an inline file stored in a single (binary) value. However, Blobs are beyond the scope of what we'll discuss here.

Workers do not share any scope or resources with each other or the main program -- that would bring all the nightmares of threaded programming to the forefront -- but instead have a basic event messaging mechanism connecting them.

The `w1` Worker object is an event listener and trigger, which lets you subscribe to events sent by the Worker as well as send events to the Worker.

Here's how to listen for events (actually, the fixed `"message"` event):

```
w1.addEventListener( "message", function(evt){  
    // evt.data  
} );
```

And you can send the `"message"` event to the Worker:

```
w1.postMessage( "something cool to say" );
```

Inside the Worker, the messaging is totally symmetrical:

```
// "mycoolworker.js"  
  
addEventListener( "message", function(evt){  
    // evt.data  
} );  
  
postMessage( "a really cool reply" );
```

Notice that a dedicated Worker is in a one-to-one relationship with the program that created it. That is, the `"message"` event doesn't need any disambiguation here, because we're sure that it could only have come from this one-to-one relationship -- either it came from the Worker or the main page.

Usually the main page application creates the Workers, but a Worker can instantiate its own child Worker(s) -- known as subworkers -- as necessary. Sometimes this is useful to delegate such details to a sort of "master" Worker that spawns other Workers to process parts of a task. Unfortunately, at the time of this writing, Chrome still does not support subworkers, while Firefox does.

To kill a Worker immediately from the program that created it, call `terminate()` on the Worker object (like `w1` in the previous snippets). Abruptly terminating a Worker thread does not give it any chance to finish up its work or clean up any resources. It's akin to you closing a browser tab to kill a page.

If you have two or more pages (or multiple tabs with the same page!) in the browser that try to create a Worker from the same file URL, those will actually end up as completely separate Workers. Shortly, we'll discuss a way to "share" a Worker.

Note: It may seem like a malicious or ignorant JS program could easily perform a denial-of-service attack on a system by spawning hundreds of Workers, seemingly each with their own thread. While it's true that it's somewhat of a guarantee that a Worker will end up on a separate thread, this guarantee is not unlimited. The system is free to decide how many actual threads/CPUs/cores it really wants to create. There's no way to predict or guarantee

how many you'll have access to, though many people assume it's at least as many as the number of CPUs/cores available. I think the safest assumption is that there's at least one other thread besides the main UI thread, but that's about it.

Worker Environment

Inside the Worker, you do not have access to any of the main program's resources. That means you cannot access any of its global variables, nor can you access the page's DOM or other resources. Remember: it's a totally separate thread.

You can, however, perform network operations (Ajax, WebSockets) and set timers. Also, the Worker has access to its own copy of several important global variables/features, including `navigator`, `location`, `JSON`, and `applicationCache`.

You can also load extra JS scripts into your Worker, using `importScripts(...)`:

```
// inside the Worker
importScripts( "foo.js", "bar.js" );
```

These scripts are loaded synchronously, which means the `importScripts(...)` call will block the rest of the Worker's execution until the file(s) are finished loading and executing.

Note: There have also been some discussions about exposing the `<canvas>` API to Workers, which combined with having canvases be Transferables (see the "Data Transfer" section), would allow Workers to perform more sophisticated off-thread graphics processing, which can be useful for high-performance gaming (WebGL) and other similar applications. Although this doesn't exist yet in any browsers, it's likely to happen in the near future.

What are some common uses for Web Workers?

- Processing intensive math calculations
- Sorting large data sets
- Data operations (compression, audio analysis, image pixel manipulations, etc.)
- High-traffic network communications

Data Transfer

You may notice a common characteristic of most of those uses, which is that they require a large amount of information to be transferred across the barrier between threads using the event mechanism, perhaps in both directions.

In the early days of Workers, serializing all data to a string value was the only option. In addition to the speed penalty of the two-way serializations, the other major negative was that the data was being copied, which meant a doubling of memory usage (and the subsequent churn of garbage collection).

Thankfully, we now have a few better options.

If you pass an object, a so-called "Structured Cloning Algorithm"

(https://developer.mozilla.org/en-US/docs/Web/Guide/API/DOM/The_structured_clone_algorithm)

is used to copy/duplicate the object on the other side. This algorithm is fairly sophisticated and can even handle duplicating objects with circular references. The to-string/from-string performance penalty is not paid, but we still have duplication of memory using this approach. There is support for this in IE10 and above, as well as all the other major browsers.

An even better option, especially for larger data sets, is "Transferable Objects"

(<http://updates.html5rocks.com/2011/12/Transferable-Objects-Lightning-Fast>).

What happens is that the object's "ownership" is transferred, but the data itself is not moved. Once you transfer away an object to a Worker, it's empty or inaccessible in the originating location -- that eliminates the hazards of threaded programming over a shared scope. Of course, transfer of ownership can go in both directions.

There really isn't much you need to do to opt into a Transferable Object; any data structure that implements the Transferable interface (<https://developer.mozilla.org/en-US/docs/Web/API/Transferable>) will automatically be transferred this way (support Firefox & Chrome).

For example, typed arrays like `Uint8Array` (see the *ES6 & Beyond* title of this series) are "Transferables." This is how you'd send a Transferable Object using `postMessage(...)`:

```
// `foo` is a `Uint8Array` for instance
postMessage( foo.buffer, [ foo.buffer ] );
```

The first parameter is the raw buffer and the second parameter is a list of what to transfer.

Browsers that don't support Transferable Objects simply degrade to structured cloning, which means performance reduction rather than outright feature breakage.

Shared Workers

If your site or app allows for loading multiple tabs of the same page (a common feature), you may very well want to reduce the resource usage of their system by preventing duplicate dedicated Workers; the most common limited resource in this respect is a socket network connection, as browsers limit the number of simultaneous connections to a single host. Of course, limiting multiple connections from a client also eases your server resource requirements.

In this case, creating a single centralized Worker that all the page instances of your site or app can *share* is quite useful.

That's called a `SharedWorker`, which you create like so (support for this is limited to Firefox and Chrome):

```
var w1 = new SharedWorker( "http://some.url.1/mycoolworker.js" );
```

Because a shared Worker can be connected to or from more than one program instance or page on your site, the Worker needs a way to know which program a message comes from. This unique identification is called a "port" -- think network socket ports. So the calling program must use the `port` object of the Worker for communication:

```
w1.port.addEventListener( "message", handleMessages );  
  
// ...  
  
w1.port.postMessage( "something cool" );
```

Also, the port connection must be initialized, as:

```
w1.port.start();
```

Inside the shared Worker, an extra event must be handled: `"connect"`. This event provides the port `object` for that particular connection. The most convenient way to keep multiple connections separate is to use closure (see *Scope & Closures* title of this series) over the `port`, as shown next, with the event listening and transmitting for that connection defined inside the handler for the `"connect"` event:

```
// inside the shared Worker
addEventListener( "connect", function(evt){
    // the assigned port for this connection
    var port = evt.ports[0];

    port.addEventListener( "message", function(evt){
        // ...

        port.postMessage( ... );

        // ...
    } );

    // initialize the port connection
    port.start();
} );
```

Other than that difference, shared and dedicated Workers have the same capabilities and semantics.

Note: Shared Workers survive the termination of a port connection if other port connections are still alive, whereas dedicated Workers are terminated whenever the connection to their initiating program is terminated.

Polyfilling Web Workers

Web Workers are very attractive performance-wise for running JS programs in parallel. However, you may be in a position where your code needs to run in older browsers that lack support. Because Workers are an API and not a syntax, they can be polyfilled, to an extent.

If a browser doesn't support Workers, there's simply no way to fake multithreading from the performance perspective. Iframes are commonly thought of to provide a parallel environment, but in all modern browsers they actually run on the same thread as the main page, so they're not sufficient for faking parallelism.

As we detailed in Chapter 1, JS's asynchronicity (not parallelism) comes from the event loop queue, so you can force faked Workers to be asynchronous using timers (`setTimeout(...)`, etc.). Then you just need to provide a polyfill for the Worker API. There are some listed here (<https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-Browser-Polyfills#web-workers>), but frankly none of them look great.

I've written a sketch of a polyfill for `Worker` [here](#) (<https://gist.github.com/getify/1b26accb1a09aa53ad25>). It's basic, but it should get the job done for simple `Worker` support, given that the two-way messaging works correctly as well

as `"onerror"` handling. You could probably also extend it with more features, such as `terminate()` or faked Shared Workers, as you see fit.

Note: You can't fake synchronous blocking, so this polyfill just disallows use of `importScripts(...)`. Another option might have been to parse and transform the Worker's code (once Ajax loaded) to handle rewriting to some asynchronous form of an `importScripts(...)` polyfill, perhaps with a promise-aware interface.

SIMD

Single instruction, multiple data (SIMD) is a form of "data parallelism," as contrasted to "task parallelism" with Web Workers, because the emphasis is not really on program logic chunks being parallelized, but rather multiple bits of data being processed in parallel.

With SIMD, threads don't provide the parallelism. Instead, modern CPUs provide SIMD capability with "vectors" of numbers -- think: type specialized arrays -- as well as instructions that can operate in parallel across all the numbers; these are low-level operations leveraging instruction-level parallelism.

The effort to expose SIMD capability to JavaScript is primarily spearheaded by Intel (<https://01.org/node/1495>), namely by Mohammad Haghigat (at the time of this writing), in cooperation with Firefox and Chrome teams. SIMD is on an early standards track with a good chance of making it into a future revision of JavaScript, likely in the ES7 timeframe.

SIMD JavaScript proposes to expose short vector types and APIs to JS code, which on those SIMD-enabled systems would map the operations directly through to the CPU equivalents, with fallback to non-parallelized operation "shims" on non-SIMD systems.

The performance benefits for data-intensive applications (signal analysis, matrix operations on graphics, etc.) with such parallel math processing are quite obvious!

Early proposal forms of the SIMD API at the time of this writing look like this:

```
var v1 = SIMD.float32x4( 3.14159, 21.0, 32.3, 55.55 );
var v2 = SIMD.float32x4( 2.1, 3.2, 4.3, 5.4 );

var v3 = SIMD.int32x4( 10, 101, 1001, 10001 );
var v4 = SIMD.int32x4( 10, 20, 30, 40 );

SIMD.float32x4.mul( v1, v2 );      // [ 6.597339, 67.2, 138.89, 299.97 ]
SIMD.int32x4.add( v3, v4 );       // [ 20, 121, 1031, 10041 ]
```

Shown here are two different vector data types, 32-bit floating-point numbers and 32-bit integer numbers. You can see that these vectors are sized exactly to four 32-bit elements, as this matches the SIMD vector sizes (128-bit) available in most modern CPUs. It's also possible we may see an `x8` (or larger!) version of these APIs in the future.

Besides `mul()` and `add()`, many other operations are likely to be included, such as `sub()`, `div()`, `abs()`, `neg()`, `sqrt()`, `reciprocal()`, `reciprocalSqrt()` (arithmetic), `shuffle()` (rearrange vector elements), `and()`, `or()`, `xor()`, `not()` (logical), `equal()`, `greaterThan()`, `lessThan()` (comparison), `shiftLeft()`, `shiftRightLogical()`, `shiftRightArithmetic()` (shifts), `fromFloat32x4()`, and `fromInt32x4()` (conversions).

Note: There's an official "prolyfill" (hopeful, expectant, future-leaning polyfill) for the SIMD functionality available (https://github.com/johnmccutchan/ecmascript_simd), which illustrates a lot more of the planned SIMD capability than we've illustrated in this section.

asm.js

"asm.js" (<http://asmjs.org/>) is a label for a highly optimizable subset of the JavaScript language. By carefully avoiding certain mechanisms and patterns that are *hard* to optimize (garbage collection, coercion, etc.), asm.js-styled code can be recognized by the JS engine and given special attention with aggressive low-level optimizations.

Distinct from other program performance mechanisms discussed in this chapter, asm.js isn't necessarily something that needs to be adopted into the JS language specification. There *is* an asm.js specification (<http://asmjs.org/spec/latest/>), but it's mostly for tracking an agreed upon set of candidate inferences for optimization rather than a set of requirements of JS engines.

There's not currently any new syntax being proposed. Instead, asm.js suggests ways to recognize existing standard JS syntax that conforms to the rules of asm.js and let engines implement their own optimizations accordingly.

There's been some disagreement between browser vendors over exactly how asm.js should be activated in a program. Early versions of the asm.js experiment required a `"use asm";` pragma (similar to strict mode's `"use strict";`) to help clue the JS engine to be looking for asm.js optimization opportunities and hints. Others have asserted that asm.js should just be a set of heuristics that engines automatically recognize without the author having to do anything extra, meaning that existing programs could theoretically benefit from asm.js-style optimizations without doing anything special.

How to Optimize with asm.js

The first thing to understand about asm.js optimizations is around types and coercion (see the *Types & Grammar* title of this series). If the JS engine has to track multiple different types of values in a variable through various operations, so that it can handle coercions between types as necessary, that's a lot of extra work that keeps the program optimization suboptimal.

Note: We're going to use asm.js-style code here for illustration purposes, but be aware that it's not commonly expected that you'll author such code by hand. asm.js is more intended to a compilation target from other tools, such as Emscripten (<https://github.com/kripken/emscripten/wiki>). It's of course possible to write your own asm.js code, but that's usually a bad idea because the code is very low level and managing it can be very time consuming and error prone. Nevertheless, there may be cases where you'd want to hand tweak your code for asm.js optimization purposes.

There are some "tricks" you can use to hint to an asm.js-aware JS engine what the intended type is for variables/operations, so that it can skip these coercion tracking steps.

For example:

```
var a = 42;  
// ...  
var b = a;
```

In that program, the `b = a` assignment leaves the door open for type divergence in variables. However, it could instead be written as:

```
var a = 42;  
// ...  
var b = a | 0;
```

Here, we've used the `|` ("binary OR") with value `0`, which has no effect on the value other than to make sure it's a 32-bit integer. That code run in a normal JS engine works just fine, but when run in an asm.js-aware JS engine it *can* signal that `b` should always be treated as a 32-bit integer, so the coercion tracking can be skipped.

Similarly, the addition operation between two variables can be restricted to a more performant integer addition (instead of floating point):

```
(a + b) | 0
```

Again, the asm.js-aware JS engine can see that hint and infer that the `+` operation should be 32-bit integer addition because the end result of the whole expression would automatically be 32-bit integer conformed anyway.

asm.js Modules

One of the biggest detractors to performance in JS is around memory allocation, garbage collection, and scope access. asm.js suggests one of the ways around these issues is to declare a more formalized asm.js "module" -- do not confuse these with ES6 modules; see the *ES6 & Beyond* title of this series.

For an asm.js module, you need to explicitly pass in a tightly conformed namespace -- this is referred to in the spec as `stdlib`, as it should represent standard libraries needed -- to import necessary symbols, rather than just using globals via lexical scope. In the base case, the `window` object is an acceptable `stdlib` object for asm.js module purposes, but you could and perhaps should construct an even more restricted one.

You also must declare a "heap" -- which is just a fancy term for a reserved spot in memory where variables can already be used without asking for more memory or releasing previously used memory -- and pass that in, so that the asm.js module won't need to do anything that would cause memory churn; it can just use the pre-reserved space.

A "heap" is likely a typed `ArrayBuffer`, such as:

```
var heap = new ArrayBuffer( 0x10000 );      // 64k heap
```

Using that pre-reserved 64k of binary space, an asm.js module can store and retrieve values in that buffer without any memory allocation or garbage collection penalties. For example, the `heap` buffer could be used inside the module to back an array of 64-bit float values like this:

```
var arr = new Float64Array( heap );
```

OK, so let's make a quick, silly example of an asm.js-styled module to illustrate how these pieces fit together. We'll define a `foo(...)` that takes a start (`x`) and end (`y`) integer for a range, and calculates all the inner adjacent multiplications of the values in the range, and then finally averages those values together:

```

function fooASM(stdlib,foreign,heap) {
    "use asm";

    var arr = new stdlib.Int32Array( heap );

    function foo(x,y) {
        x = x | 0;
        y = y | 0;

        var i = 0;
        var p = 0;
        var sum = 0;
        var count = ((y|0) - (x|0)) | 0;

        // calculate all the inner adjacent multiplications
        for (i = x | 0;
             (i | 0) < (y | 0);
             p = (p + 8) | 0, i = (i + 1) | 0
        ) {
            // store result
            arr[ p >> 3 ] = (i * (i + 1)) | 0;
        }

        // calculate average of all intermediate values
        for (i = 0, p = 0;
             (i | 0) < (count | 0);
             p = (p + 8) | 0, i = (i + 1) | 0
        ) {
            sum = (sum + arr[ p >> 3 ]) | 0;
        }

        return +(sum / count);
    }

    return {
        foo: foo
    };
}

var heap = new ArrayBuffer( 0x1000 );
var foo = fooASM( window, null, heap ).foo;

foo( 10, 20 );           // 233

```

Note: This asm.js example is hand authored for illustration purposes, so it doesn't represent the same code that would be produced from a compilation tool targeting asm.js. But it does show the typical nature of asm.js code, especially the type hinting and use of the `heap` buffer for temporary variable storage.

The first call to `fooASM(...)` is what sets up our `asm.js` module with its `heap` allocation. The result is a `foo(...)` function we can call as many times as necessary. Those `foo(...)` calls should be specially optimized by an `asm.js`-aware JS engine. Importantly, the preceding code is completely standard JS and would run just fine (without special optimization) in a non-`asm.js` engine.

Obviously, the nature of restrictions that make `asm.js` code so optimizable reduces the possible uses for such code significantly. `asm.js` won't necessarily be a general optimization set for any given JS program. Instead, it's intended to provide an optimized way of handling specialized tasks such as intensive math operations (e.g., those used in graphics processing for games).

Review

The first four chapters of this book are based on the premise that `async` coding patterns give you the ability to write more performant code, which is generally a very important improvement. But `async` behavior only gets you so far, because it's still fundamentally bound to a single event loop thread.

So in this chapter we've covered several program-level mechanisms for improving performance even further.

Web Workers let you run a JS file (aka program) in a separate thread using `async` events to message between the threads. They're wonderful for offloading long-running or resource-intensive tasks to a different thread, leaving the main UI thread more responsive.

SIMD proposes to map CPU-level parallel math operations to JavaScript APIs for high-performance data-parallel operations, like number processing on large data sets.

Finally, `asm.js` describes a small subset of JavaScript that avoids the hard-to-optimize parts of JS (like garbage collection and coercion) and lets the JS engine recognize and run such code through aggressive optimizations. `asm.js` could be hand authored, but that's extremely tedious and error prone, akin to hand authoring assembly language (hence the name). Instead, the main intent is that `asm.js` would be a good target for cross-compilation from other highly optimized program languages -- for example, Emscripten (<https://github.com/kripken/emscripten/wiki>) transpiling C/C++ to JavaScript.

While not covered explicitly in this chapter, there are even more radical ideas under very early discussion for JavaScript, including approximations of direct threaded functionality (not just hidden behind data structure APIs). Whether that happens explicitly, or we just see more parallelism creep into JS behind the scenes, the future of more optimized program-level performance in JS looks really *promising*.

As the first four chapters of this book were all about performance as a coding pattern (asynchrony and concurrency), and Chapter 5 was about performance at the macro program architecture level, this chapter goes after the topic of performance at the micro level, focusing on single expressions/statements.

One of the most common areas of curiosity -- indeed, some developers can get quite obsessed about it -- is in analyzing and testing various options for how to write a line or chunk of code, and which one is faster.

We're going to look at some of these issues, but it's important to understand from the outset that this chapter is **not** about feeding the obsession of micro-performance tuning, like whether some given JS engine can run `++a` faster than `a++`. The more important goal of this chapter is to figure out what kinds of JS performance matter and which ones don't, *and how to tell the difference*.

But even before we get there, we need to explore how to most accurately and reliably test JS performance, because there's tons of misconceptions and myths that have flooded our collective cult knowledge base. We've got to sift through all that junk to find some clarity.

Benchmarking

OK, time to start dispelling some misconceptions. I'd wager the vast majority of JS developers, if asked to benchmark the speed (execution time) of a certain operation, would initially go about it something like this:

```
var start = (new Date()).getTime();    // or `Date.now()`

// do some operation

var end = (new Date()).getTime();

console.log("Duration:", (end - start));
```

Raise your hand if that's roughly what came to your mind. Yep, I thought so. There's a lot wrong with this approach, but don't feel bad; **we've all been there**.

What did that measurement tell you, exactly? Understanding what it does and doesn't say about the execution time of the operation in question is key to learning how to appropriately benchmark performance in JavaScript.

If the duration reported is `0`, you may be tempted to believe that it took less than a millisecond. But that's not very accurate. Some platforms don't have single millisecond precision, but instead only update the timer in larger increments. For example, older

versions of windows (and thus IE) had only 15ms precision, which means the operation has to take at least that long for anything other than `0` to be reported!

Moreover, whatever duration is reported, the only thing you really know is that the operation took approximately that long on that exact single run. You have near-zero confidence that it will always run at that speed. You have no idea if the engine or system had some sort of interference at that exact moment, and that at other times the operation could run faster.

What if the duration reported is `4`? Are you more sure it took about four milliseconds? Nope. It might have taken less time, and there may have been some other delay in getting either `start` or `end` timestamps.

More troublingly, you also don't know that the circumstances of this operation test aren't overly optimistic. It's possible that the JS engine figured out a way to optimize your isolated test case, but in a more real program such optimization would be diluted or impossible, such that the operation would run slower than your test.

So... what do we know? Unfortunately, with those realizations stated, **we know very little**. Something of such low confidence isn't even remotely good enough to build your determinations on. Your "benchmark" is basically useless. And worse, it's dangerous in that it implies false confidence, not just to you but also to others who don't think critically about the conditions that led to those results.

Repetition

"OK," you now say, "Just put a loop around it so the whole test takes longer." If you repeat an operation 100 times, and that whole loop reportedly takes a total of 137ms, then you can just divide by 100 and get an average duration of 1.37ms for each operation, right?

Well, not exactly.

A straight mathematical average by itself is definitely not sufficient for making judgments about performance which you plan to extrapolate to the breadth of your entire application. With a hundred iterations, even a couple of outliers (high or low) can skew the average, and then when you apply that conclusion repeatedly, you even further inflate the skew beyond credulity.

Instead of just running for a fixed number of iterations, you can instead choose to run the loop of tests until a certain amount of time has passed. That might be more reliable, but how do you decide how long to run? You might guess that it should be some multiple of how long your operation should take to run once. Wrong.

Actually, the length of time to repeat across should be based on the accuracy of the timer you're using, specifically to minimize the chances of inaccuracy. The less precise your timer, the longer you need to run to make sure you've minimized the error percentage. A 15ms timer is pretty bad for accurate benchmarking; to minimize its uncertainty (aka "error rate") to less than 1%, you need to run each cycle of test iterations for 750ms. A 1ms timer only needs a cycle to run for 50ms to get the same confidence.

But then, that's just a single sample. To be sure you're factoring out the skew, you'll want lots of samples to average across. You'll also want to understand something about just how slow the worst sample is, how fast the best sample is, how far apart those best and worse cases were, and so on. You'll want to know not just a number that tells you how fast something ran, but also to have some quantifiable measure of how trustable that number is.

Also, you probably want to combine these different techniques (as well as others), so that you get the best balance of all the possible approaches.

That's all bare minimum just to get started. If you've been approaching performance benchmarking with anything less serious than what I just glossed over, well... "you don't know: proper benchmarking."

Benchmark.js

Any relevant and reliable benchmark should be based on statistically sound practices. I am not going to write a chapter on statistics here, so I'll hand wave around some terms: standard deviation, variance, margin of error. If you don't know what those terms really mean -- I took a stats class back in college and I'm still a little fuzzy on them -- you are not actually qualified to write your own benchmarking logic.

Luckily, smart folks like John-David Dalton and Mathias Bynens do understand these concepts, and wrote a statistically sound benchmarking tool called Benchmark.js (<http://benchmarkjs.com/>). So I can end the suspense by simply saying: "just use that tool."

I won't repeat their whole documentation for how Benchmark.js works; they have fantastic API Docs (<http://benchmarkjs.com/docs>) you should read. Also there are some great (<http://calendar.perfplanet.com/2010/bulletproof-javascript-benchmarks/>) writeups (<http://monsur.hossa.in/2012/12/11/benchmarkjs.html>) on more of the details and methodology.

But just for quick illustration purposes, here's how you could use Benchmark.js to run a quick performance test:

```

function foo() {
    // operation(s) to test
}

var bench = new Benchmark(
    "foo test",           // test name
    foo,                  // function to test (just contents)
    {
        // ...            // optional extra options (see docs)
    }
);

bench.hz;                // number of operations per second
bench.stats.moe;         // margin of error
bench.stats.variance;    // variance across samples
// ...

```

There's *lots* more to learn about using Benchmark.js besides this glance I'm including here. But the point is that it's handling all of the complexities of setting up a fair, reliable, and valid performance benchmark for a given piece of JavaScript code. If you're going to try to test and benchmark your code, this library is the first place you should turn.

We're showing here the usage to test a single operation like X, but it's fairly common that you want to compare X to Y. This is easy to do by simply setting up two different tests in a "Suite" (a Benchmark.js organizational feature). Then, you run them head-to-head, and compare the statistics to conclude whether X or Y was faster.

Benchmark.js can of course be used to test JavaScript in a browser (see the "jsPerf.com" section later in this chapter), but it can also run in non-browser environments (Node.js, etc.).

One largely untapped potential use-case for Benchmark.js is to use it in your Dev or QA environments to run automated performance regression tests against critical path parts of your application's JavaScript. Similar to how you might run unit test suites before deployment, you can also compare the performance against previous benchmarks to monitor if you are improving or degrading application performance.

Setup/Teardown

In the previous code snippet, we glossed over the "extra options" `{ ... }` object. But there are two options we should discuss: `setup` and `teardown`.

These two options let you define functions to be called before and after your test case runs.

It's incredibly important to understand that your `setup` and `teardown` code **does not run for each test iteration**. The best way to think about it is that there's an outer loop (repeating cycles), and an inner loop (repeating test iterations). `setup` and `teardown` are run at the

beginning and end of each *outer* loop (aka cycle) iteration, but not inside the inner loop.

Why does this matter? Let's imagine you have a test case that looks like this:

```
a = a + "w";
b = a.charAt( 1 );
```

Then, you set up your test `setup` as follows:

```
var a = "x";
```

Your temptation is probably to believe that `a` is starting out as `"x"` for each test iteration.

But it's not! It's starting `a` at `"x"` for each test cycle, and then your repeated `+ "w"` concatenations will be making a larger and larger `a` value, even though you're only ever accessing the character `"w"` at the `1` position.

Where this most commonly bites you is when you make side effect changes to something like the DOM, like appending a child element. You may think your parent element is set as empty each time, but it's actually getting lots of elements added, and that can significantly sway the results of your tests.

Context Is King

Don't forget to check the context of a particular performance benchmark, especially a comparison between X and Y tasks. Just because your test reveals that X is faster than Y doesn't mean that the conclusion "X is faster than Y" is actually relevant.

For example, let's say a performance test reveals that X runs 10,000,000 operations per second, and Y runs at 8,000,000 operations per second. You could claim that Y is 20% slower than X, and you'd be mathematically correct, but your assertion doesn't hold as much water as you'd think.

Let's think about the results more critically: 10,000,000 operations per second is 10,000 operations per millisecond, and 10 operations per microsecond. In other words, a single operation takes 0.1 microseconds, or 100 nanoseconds. It's hard to fathom just how small 100ns is, but for comparison, it's often cited that the human eye isn't generally capable of distinguishing anything less than 100ms, which is one million times slower than the 100ns speed of the X operation.

Even recent scientific studies showing that maybe the brain can process as quick as 13ms (about 8x faster than previously asserted) would mean that X is still running 125,000 times faster than the human brain can perceive a distinct thing happening. **X is going really, really fast.**

But more importantly, let's talk about the difference between X and Y, the 2,000,000 operations per second difference. If X takes 100ns, and Y takes 80ns, the difference is 20ns, which in the best case is still one 650-thousandth of the interval the human brain can perceive.

What's my point? **None of this performance difference matters, at all!**

But wait, what if this operation is going to happen a whole bunch of times in a row? Then the difference could add up, right?

OK, so what we're asking then is, how likely is it that operation X is going to be run over and over again, one right after the other, and that this has to happen 650,000 times just to get a sliver of a hope the human brain could perceive it. More likely, it'd have to happen 5,000,000 to 10,000,000 times together in a tight loop to even approach relevance.

While the computer scientist in you might protest that this is possible, the louder voice of realism in you should sanity check just how likely or unlikely that really is. Even if it is relevant in rare occasions, it's irrelevant in most situations.

The vast majority of your benchmark results on tiny operations -- like the `++x` vs `x++` myth -- **are just totally bogus** for supporting the conclusion that X should be favored over Y on a performance basis.

Engine Optimizations

You simply cannot reliably extrapolate that if X was 10 microseconds faster than Y in your isolated test, that means X is always faster than Y and should always be used. That's not how performance works. It's vastly more complicated.

For example, let's imagine (purely hypothetical) that you test some microperformance behavior such as comparing:

```

var twelve = "12";
var foo = "foo";

// test 1
var X1 = parseInt( twelve );
var X2 = parseInt( foo );

// test 2
var Y1 = Number( twelve );
var Y2 = Number( foo );

```

If you understand what `parseInt(..)` does compared to `Number(..)`, you might intuit that `parseInt(..)` potentially has "more work" to do, especially in the `foo` case. Or you might intuit that they should have the same amount of work to do in the `foo` case, as both should be able to stop at the first character `"f"`.

Which intuition is correct? I honestly don't know. But I'll make the case it doesn't matter what your intuition is. What might the results be when you test it? Again, I'm making up a pure hypothetical here, I haven't actually tried, nor do I care.

Let's pretend the test comes back that `x` and `y` are statistically identical. Have you then confirmed your intuition about the `"f"` character thing? Nope.

It's possible in our hypothetical that the engine might recognize that the variables `twelve` and `foo` are only being used in one place in each test, and so it might decide to inline those values. Then it may realize that `Number("12")` can just be replaced by `12`. And maybe it comes to the same conclusion with `parseInt(..)`, or maybe not.

Or an engine's dead-code removal heuristic could kick in, and it could realize that variables `x` and `y` aren't being used, so declaring them is irrelevant, so it doesn't end up doing anything at all in either test.

And all that's just made with the mindset of assumptions about a single test run. Modern engines are fantastically more complicated than what we're intuiting here. They do all sorts of tricks, like tracing and tracking how a piece of code behaves over a short period of time, or with a particularly constrained set of inputs.

What if the engine optimizes a certain way because of the fixed input, but in your real program you give more varied input and the optimization decisions shake out differently (or not at all!)? Or what if the engine kicks in optimizations because it sees the code being run tens of thousands of times by the benchmarking utility, but in your real program it will only run a hundred times in near proximity, and under those conditions the engine determines the optimizations are not worth it?

And all those optimizations we just hypothesized about might happen in our constrained test but maybe the engine wouldn't do them in a more complex program (for various reasons). Or it could be reversed -- the engine might not optimize such trivial code but may be more inclined to optimize it more aggressively when the system is already more taxed by a more sophisticated program.

The point I'm trying to make is that you really don't know for sure exactly what's going on under the covers. All the guesses and hypothesis you can muster don't amount to hardly anything concrete for really making such decisions.

Does that mean you can't really do any useful testing? **Definitely not!**

What this boils down to is that testing *not real* code gives you *not real* results. In so much as is possible and practical, you should test actual real, non-trivial snippets of your code, and under as best of real conditions as you can actually hope to. Only then will the results you get have a chance to approximate reality.

Microbenchmarks like `++x` vs `x++` are so incredibly likely to be bogus, we might as well just flatly assume them as such.

jsPerf.com

While Benchmark.js is useful for testing the performance of your code in whatever JS environment you're running, it cannot be stressed enough that you need to compile test results from lots of different environments (desktop browsers, mobile devices, etc.) if you want to have any hope of reliable test conclusions.

For example, Chrome on a high-end desktop machine is not likely to perform anywhere near the same as Chrome mobile on a smartphone. And a smartphone with a full battery charge is not likely to perform anywhere near the same as a smartphone with 2% battery life left, when the device is starting to power down the radio and processor.

If you want to make assertions like "X is faster than Y" in any reasonable sense across more than just a single environment, you're going to need to actually test as many of those real world environments as possible. Just because Chrome executes some X operation faster than Y doesn't mean that all browsers do. And of course you also probably will want to cross-reference the results of multiple browser test runs with the demographics of your users.

There's an awesome website for this purpose called jsPerf (<http://jsperf.com>). It uses the Benchmark.js library we talked about earlier to run statistically accurate and reliable tests, and makes the test on an openly available URL that you can pass around to others.

Each time a test is run, the results are collected and persisted with the test, and the cumulative test results are graphed on the page for anyone to see.

When creating a test on the site, you start out with two test cases to fill in, but you can add as many as you need. You also have the ability to set up `setup` code that is run at the beginning of each test cycle and `teardown` code run at the end of each cycle.

Note: A trick for doing just one test case (if you're benchmarking a single approach instead of a head-to-head) is to fill in the second test input boxes with placeholder text on first creation, then edit the test and leave the second test blank, which will delete it. You can always add more test cases later.

You can define the initial page setup (importing libraries, defining utility helper functions, declaring variables, etc.). There are also options for defining setup and teardown behavior if needed -- consult the "Setup/Teardown" section in the Benchmark.js discussion earlier.

Sanity Check

jsPerf is a fantastic resource, but there's an awful lot of tests published that when you analyze them are quite flawed or bogus, for any of a variety of reasons as outlined so far in this chapter.

Consider:

```
// Case 1
var x = [];
for (var i=0; i<10; i++) {
    x[i] = "x";
}

// Case 2
var x = [];
for (var i=0; i<10; i++) {
    x[x.length] = "x";
}

// Case 3
var x = [];
for (var i=0; i<10; i++) {
    x.push( "x" );
}
```

Some observations to ponder about this test scenario:

- It's extremely common for devs to put their own loops into test cases, and they forget that Benchmark.js already does all the repetition you need. There's a really strong

chance that the `for` loops in these cases are totally unnecessary noise.

- The declaring and initializing of `x` is included in each test case, possibly unnecessarily. Recall from earlier that if `x = []` were in the `setup` code, it wouldn't actually be run before each test iteration, but instead once at the beginning of each cycle. That means `x` would continue growing quite large, not just the size `10` implied by the `for` loops.

So is the intent to make sure the tests are constrained only to how the JS engine behaves with very small arrays (size `10`)? That *could* be the intent, but if it is, you have to consider if that's not focusing far too much on nuanced internal implementation details.

On the other hand, does the intent of the test embrace the context that the arrays will actually be growing quite large? Is the JS engines' behavior with larger arrays relevant and accurate when compared with the intended real world usage?

- Is the intent to find out how much `x.length` or `x.push(..)` add to the performance of the operation to append to the `x` array? OK, that might be a valid thing to test. But then again, `push(..)` is a function call, so of course it's going to be slower than `[..]` access. Arguably, cases 1 and 2 are fairer than case 3.

Here's another example that illustrates a common apples-to-oranges flaw:

```
// Case 1
var x = ["John", "Albert", "Sue", "Frank", "Bob"];
x.sort();

// Case 2
var x = ["John", "Albert", "Sue", "Frank", "Bob"];
x.sort( function mySort(a,b){
    if (a < b) return -1;
    if (a > b) return 1;
    return 0;
} );
```

Here, the obvious intent is to find out how much slower the custom `mySort(..)` comparator is than the built-in default comparator. But by specifying the function `mysort(..)` as inline function expression, you've created an unfair/bogus test. Here, the second case is not only testing a custom user JS function, **but it's also testing creating a new function expression for each iteration.**

Would it surprise you to find out that if you run a similar test but update it to isolate only for creating an inline function expression versus using a pre-declared function, the inline function expression creation can be from 2% to 20% slower!?

Unless your intent with this test *is* to consider the inline function expression creation "cost," a better/fairer test would put `mysort(...)`'s declaration in the page setup -- don't put it in the test `setup` as that's unnecessary redeclaration for each cycle -- and simply reference it by name in the test case: `x.sort(mysort)`.

Building on the previous example, another pitfall is in opaquely avoiding or adding "extra work" to one test case that creates an apples-to-oranges scenario:

```
// Case 1
var x = [12, -14, 0, 3, 18, 0, 2.9];
x.sort();

// Case 2
var x = [12, -14, 0, 3, 18, 0, 2.9];
x.sort( function mySort(a,b){
    return a - b;
} );
```

Setting aside the previously mentioned inline function expression pitfall, the second case's `mysort(...)` works in this case because you have provided it numbers, but would have of course failed with strings. The first case doesn't throw an error, but it actually behaves differently and has a different outcome! It should be obvious, but: **a different outcome between two test cases almost certainly invalidates the entire test!**

But beyond the different outcomes, in this case, the built in `sort(...)`'s comparator is actually doing "extra work" that `mySort()` does not, in that the built-in one coerces the compared values to strings and does lexicographic comparison. The first snippet results in `[-14, 0, 0, 12, 18, 2.9, 3]` while the second snippet results (likely more accurately based on intent) in `[-14, 0, 0, 2.9, 3, 12, 18]`.

So that test is unfair because it's not actually doing the same task between the cases. Any results you get are bogus.

These same pitfalls can even be much more subtle:

```
// Case 1
var x = false;
var y = x ? 1 : 2;

// Case 2
var x;
var y = x ? 1 : 2;
```

Here, the intent might be to test the performance impact of the coercion to a Boolean that the `? :` operator will do if the `x` expression is not already a Boolean (see the *Types & Grammar* title of this book series). So, you're apparently OK with the fact that there is extra work to do the coercion in the second case.

The subtle problem? You're setting `x`'s value in the first case and not setting it in the other, so you're actually doing work in the first case that you're not doing in the second. To eliminate any potential (albeit minor) skew, try:

```
// Case 1
var x = false;
var y = x ? 1 : 2;

// Case 2
var x = undefined;
var y = x ? 1 : 2;
```

Now there's an assignment in both cases, so the thing you want to test -- the coercion of `x` or not -- has likely been more accurately isolated and tested.

Writing Good Tests

Let me see if I can articulate the bigger point I'm trying to make here.

Good test authoring requires careful analytical thinking about what differences exist between two test cases and whether the differences between them are *intentional* or *unintentional*.

Intentional differences are of course normal and OK, but it's too easy to create unintentional differences that skew your results. You have to be really, really careful to avoid that skew. Moreover, you may intend a difference but it may not be obvious to other readers of your test what your intent was, so they may doubt (or trust!) your test incorrectly. How do you fix that?

Write better, clearer tests. But also, take the time to document (using the jsPerf.com "Description" field and/or code comments) exactly what the intent of your test is, even to the nuanced detail. Call out the intentional differences, which will help others and your future self to better identify unintentional differences that could be skewing the test results.

Isolate things which aren't relevant to your test by pre-declaring them in the page or test setup settings so they're outside the timed parts of the test.

Instead of trying to narrow in on a tiny snippet of your real code and benchmarking just that piece out of context, tests and benchmarks are better when they include a larger (while still relevant) context. Those tests also tend to run slower, which means any differences you spot are more relevant in context.

Microperformance

OK, until now we've been dancing around various microperformance issues and generally looking disfavorably upon obsessing about them. I want to take just a moment to address them directly.

The first thing you need to get more comfortable with when thinking about performance benchmarking your code is that the code you write is not always the code the engine actually runs. We briefly looked at that topic back in Chapter 1 when we discussed statement reordering by the compiler, but here we're going to suggest the compiler can sometimes decide to run different code than you wrote, not just in different orders but different in substance.

Let's consider this piece of code:

```
var foo = 41;

(function(){
  (function(){
    (function(baz){
      var bar = foo + baz;
      // ...
    })(1);
  })();
})();
```

You may think about the `foo` reference in the innermost function as needing to do a three-level scope lookup. We covered in the *Scope & Closures* title of this book series how lexical scope works, and the fact that the compiler generally caches such lookups so that referencing `foo` from different scopes doesn't really practically "cost" anything extra.

But there's something deeper to consider. What if the compiler realizes that `foo` isn't referenced anywhere else but that one location, and it further notices that the value never is anything except the `41` as shown?

Isn't it quite possible and acceptable that the JS compiler could decide to just remove the `foo` variable entirely, and *inline* the value, such as this:

```
(function(){
  (function(){
    (function(baz){
      var bar = 41 + baz;
      // ...
    })(1);
  })();
})();
```

Note: Of course, the compiler could probably also do a similar analysis and rewrite with the `baz` variable here, too.

When you begin to think about your JS code as being a hint or suggestion to the engine of what to do, rather than a literal requirement, you realize that a lot of the obsession over discrete syntactic minutia is most likely unfounded.

Another example:

```
function factorial(n) {
  if (n < 2) return 1;
  return n * factorial( n - 1 );
}

factorial( 5 );           // 120
```

Ah, the good ol' fashioned "factorial" algorithm! You might assume that the JS engine will run that code mostly as is. And to be honest, it might -- I'm not really sure.

But as an anecdote, the same code expressed in C and compiled with advanced optimizations would result in the compiler realizing that the call `factorial(5)` can just be replaced with the constant value `120`, eliminating the function and call entirely!

Moreover, some engines have a practice called "unrolling recursion," where it can realize that the recursion you've expressed can actually be done "easier" (i.e., more optimally) with a loop. It's possible the preceding code could be *rewritten* by a JS engine to run as:

```

function factorial(n) {
    if (n < 2) return 1;

    var res = 1;
    for (var i=n; i>1; i--) {
        res *= i;
    }
    return res;
}

factorial( 5 );           // 120

```

Now, let's imagine that in the earlier snippet you had been worried about whether `n * factorial(n-1)` or `n *= factorial(--n)` runs faster. Maybe you even did a performance benchmark to try to figure out which was better. But you miss the fact that in the bigger context, the engine may not run either line of code because it may unroll the recursion!

Speaking of `--`, `--n` versus `n--` is often cited as one of those places where you can optimize by choosing the `--n` version, because theoretically it requires less effort down at the assembly level of processing.

That sort of obsession is basically nonsense in modern JavaScript. That's the kind of thing you should be letting the engine take care of. You should write the code that makes the most sense. Compare these three `for` loops:

```

// Option 1
for (var i=0; i<10; i++) {
    console.log( i );
}

// Option 2
for (var i=0; i<10; ++i) {
    console.log( i );
}

// Option 3
for (var i=-1; ++i<10; ) {
    console.log( i );
}

```

Even if you have some theory where the second or third option is more performant than the first option by a tiny bit, which is dubious at best, the third loop is more confusing because you have to start with `-1` for `i` to account for the fact that `++i` pre-increment is used. And the difference between the first and second options is really quite irrelevant.

It's entirely possible that a JS engine may see a place where `i++` is used and realize that it can safely replace it with the `+i` equivalent, which means your time spent deciding which one to pick was completely wasted and the outcome moot.

Here's another common example of silly microperformance obsession:

```
var x = [ .. ];

// Option 1
for (var i=0; i < x.length; i++) {
    // ...
}

// Option 2
for (var i=0, len = x.length; i < len; i++) {
    // ...
}
```

The theory here goes that you should cache the length of the `x` array in the variable `len`, because ostensibly it doesn't change, to avoid paying the price of `x.length` being consulted for each iteration of the loop.

If you run performance benchmarks around `x.length` usage compared to caching it in a `len` variable, you'll find that while the theory sounds nice, in practice any measured differences are statistically completely irrelevant.

In fact, in some engines like v8, it can be shown (<http://mrale.ph/blog/2014/12/24/array-length-caching.html>) that you could make things slightly worse by pre-caching the length instead of letting the engine figure it out for you. Don't try to outsmart your JavaScript engine, you'll probably lose when it comes to performance optimizations.

Not All Engines Are Alike

The different JS engines in various browsers can all be "spec compliant" while having radically different ways of handling code. The JS specification doesn't require anything performance related -- well, except ES6's "Tail Call Optimization" covered later in this chapter.

The engines are free to decide that one operation will receive its attention to optimize, perhaps trading off for lesser performance on another operation. It can be very tenuous to find an approach for an operation that always runs faster in all browsers.

There's a movement among some in the JS dev community, especially those who work with Node.js, to analyze the specific internal implementation details of the v8 JavaScript engine and make decisions about writing JS code that is tailored to take best advantage of how v8

works. You can actually achieve a surprisingly high degree of performance optimization with such endeavors, so the payoff for the effort can be quite high.

Some commonly cited examples

(<https://github.com/petkaantonov/bluebird/wiki/Optimization-killers>) for v8:

- Don't pass the `arguments` variable from one function to any other function, as such "leakage" slows down the function implementation.
- Isolate a `try..catch` in its own function. Browsers struggle with optimizing any function with a `try..catch` in it, so moving that construct to its own function means you contain the de-optimization harm while letting the surrounding code be optimizable.

But rather than focus on those tips specifically, let's sanity check the v8-only optimization approach in a general sense.

Are you genuinely writing code that only needs to run in one JS engine? Even if your code is entirely intended for Node.js *right now*, is the assumption that v8 will *always* be the used JS engine reliable? Is it possible that someday a few years from now, there's another server-side JS platform besides Node.js that you choose to run your code on? What if what you optimized for before is now a much slower way of doing that operation on the new engine?

Or what if your code always stays running on v8 from here on out, but v8 decides at some point to change the way some set of operations works such that what used to be fast is now slow, and vice versa?

These scenarios aren't just theoretical, either. It used to be that it was faster to put multiple string values into an array and then call `join("")` on the array to concatenate the values than to just use `+` concatenation directly with the values. The historical reason for this is nuanced, but it has to do with internal implementation details about how string values were stored and managed in memory.

As a result, "best practice" advice at the time disseminated across the industry suggesting developers always use the array `join(...)` approach. And many followed.

Except, somewhere along the way, the JS engines changed approaches for internally managing strings, and specifically put in optimizations for `+` concatenation. They didn't slow down `join(...)` per se, but they put more effort into helping `+` usage, as it was still quite a bit more widespread.

Note: The practice of standardizing or optimizing some particular approach based mostly on its existing widespread usage is often called (metaphorically) "paving the cowpath."

Once that new approach to handling strings and concatenation took hold, unfortunately all the code out in the wild that was using array `join(...)` to concatenate strings was then sub-optimal.

Another example: at one time, the Opera browser differed from other browsers in how it handled the boxing/unboxing of primitive wrapper objects (see the *Types & Grammar* title of this book series). As such, their advice to developers was to use a `String` object instead of the primitive `string` value if properties like `length` or methods like `charAt(..)` needed to be accessed. This advice may have been correct for Opera at the time, but it was literally completely opposite for other major contemporary browsers, as they had optimizations specifically for the `string` primitives and not their object wrapper counterparts.

I think these various gotchas are at least possible, if not likely, for code even today. So I'm very cautious about making wide ranging performance optimizations in my JS code based purely on engine implementation details, **especially if those details are only true of a single engine.**

The reverse is also something to be wary of: you shouldn't necessarily change a piece of code to work around one engine's difficulty with running a piece of code in an acceptably performant way.

Historically, IE has been the brunt of many such frustrations, given that there have been plenty of scenarios in older IE versions where it struggled with some performance aspect that other major browsers of the time seemed not to have much trouble with. The string concatenation discussion we just had was actually a real concern back in the IE6 and IE7 days, where it was possible to get better performance out of `join(..)` than `+`.

But it's troublesome to suggest that just one browser's trouble with performance is justification for using a code approach that quite possibly could be sub-optimal in all other browsers. Even if the browser in question has a large market share for your site's audience, it may be more practical to write the proper code and rely on the browser to update itself with better optimizations eventually.

"There is nothing more permanent than a temporary hack." Chances are, the code you write now to work around some performance bug will probably outlive the performance bug in the browser itself.

In the days when a browser only updated once every five years, that was a tougher call to make. But as it stands now, browsers across the board are updating at a much more rapid interval (though obviously the mobile world still lags), and they're all competing to optimize web features better and better.

If you run across a case where a browser *does* have a performance wart that others don't suffer from, make sure to report it to them through whatever means you have available. Most browsers have open public bug trackers suitable for this purpose.

Tip: I'd only suggest working around a performance issue in a browser if it was a really drastic show-stopper, not just an annoyance or frustration. And I'd be very careful to check that the performance hack didn't have noticeable negative side effects in another browser.

Big Picture

Instead of worrying about all these microperformance nuances, we should instead be looking at big-picture types of optimizations.

How do you know what's big picture or not? You have to first understand if your code is running on a critical path or not. If it's not on the critical path, chances are your optimizations are not worth much.

Ever heard the admonition, "that's premature optimization!"? It comes from a famous quote from Donald Knuth: "premature optimization is the root of all evil.". Many developers cite this quote to suggest that most optimizations are "premature" and are thus a waste of effort. The truth is, as usual, more nuanced.

Here is Knuth's quote, in context:

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of **noncritical** parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that **critical** 3%. [emphasis added]

(http://web.archive.org/web/20130731202547/http://pplab.snu.ac.kr/courses/adv_pl05/papers/p261-knuth.pdf, Computing Surveys, Vol 6, No 4, December 1974)

I believe it's a fair paraphrasing to say that Knuth *meant*: "non-critical path optimization is the root of all evil." So the key is to figure out if your code is on the critical path -- you should optimize it! -- or not.

I'd even go so far as to say this: no amount of time spent optimizing critical paths is wasted, no matter how little is saved; but no amount of optimization on noncritical paths is justified, no matter how much is saved.

If your code is on the critical path, such as a "hot" piece of code that's going to be run over and over again, or in UX critical places where users will notice, like an animation loop or CSS style updates, then you should spare no effort in trying to employ relevant, measurably significant optimizations.

For example, consider a critical path animation loop that needs to coerce a string value to a number. There are of course multiple ways to do that (see the *Types & Grammar* title of this book series), but which one if any is the fastest?

```
var x = "42";      // need number `42`  
  
// Option 1: let implicit coercion automatically happen  
var y = x / 2;  
  
// Option 2: use `parseInt(..)`  
var y = parseInt( x, 0 ) / 2;  
  
// Option 3: use `Number(..)`  
var y = Number( x ) / 2;  
  
// Option 4: use `+` unary operator  
var y = +x / 2;  
  
// Option 5: use `|` unary operator  
var y = (x | 0) / 2;
```

Note: I will leave it as an exercise to the reader to set up a test if you're interested in examining the minute differences in performance among these options.

When considering these different options, as they say, "One of these things is not like the others." `parseInt(..)` does the job, but it also does a lot more -- it parses the string rather than just coercing. You can probably guess, correctly, that `parseInt(..)` is a slower option, and you should probably avoid it.

Of course, if `x` can ever be a value that **needs parsing**, such as `"42px"` (like from a CSS style lookup), then `parseInt(..)` really is the only suitable option!

`Number(..)` is also a function call. From a behavioral perspective, it's identical to the `+` unary operator option, but it may in fact be a little slower, requiring more machinery to execute the function. Of course, it's also possible that the JS engine recognizes this behavioral symmetry and just handles the inlining of `Number(..)`'s behavior (aka `+x`) for you!

But remember, obsessing about `+x` versus `x | 0` is in most cases likely a waste of effort. This is a microperformance issue, and one that you shouldn't let dictate/upgrade the readability of your program.

While performance is very important in critical paths of your program, it's not the only factor. Among several options that are roughly similar in performance, readability should be another important concern.

Tail Call Optimization (TCO)

As we briefly mentioned earlier, ES6 includes a specific requirement that ventures into the world of performance. It's related to a specific form of optimization that can occur with function calls: *tail call optimization*.

Briefly, a "tail call" is a function call that appears at the "tail" of another function, such that after the call finishes, there's nothing left to do (except perhaps return its result value).

For example, here's a non-recursive setup with tail calls:

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

function bar(y) {
    return foo(y + 1);    // tail call
}

function baz() {
    return 1 + bar(40);   // not tail call
}

baz();                // 42
```

`foo(y+1)` is a tail call in `bar(...)` because after `foo(...)` finishes, `bar(...)` is also finished except in this case returning the result of the `foo(..)` call. However, `bar(40)` is *not* a tail call because after it completes, its result value must be added to `1` before `baz()` can return it.

Without getting into too much nitty-gritty detail, calling a new function requires an extra amount of reserved memory to manage the call stack, called a "stack frame." So the preceding snippet would generally require a stack frame for each of `baz()`, `bar(..)`, and `foo(..)` all at the same time.

However, if a TCO-capable engine can realize that the `foo(y+1)` call is in *tail position* meaning `bar(..)` is basically complete, then when calling `foo(..)`, it doesn't need to create a new stack frame, but can instead reuse the existing stack frame from `bar(..)`. That's not only faster, but it also uses less memory.

That sort of optimization isn't a big deal in a simple snippet, but it becomes a *much bigger deal* when dealing with recursion, especially if the recursion could have resulted in hundreds or thousands of stack frames. With TCO the engine can perform all those calls with a single stack frame!

Recursion is a hairy topic in JS because without TCO, engines have had to implement arbitrary (and different!) limits to how deep they will let the recursion stack get before they stop it, to prevent running out of memory. With TCO, recursive functions with *tail position* calls can essentially run unbounded, because there's never any extra usage of memory!

Consider that recursive `factorial(...)` from before, but rewritten to make it TCO friendly:

```
function factorial(n) {
  function fact(n, res) {
    if (n < 2) return res;

    return fact( n - 1, n * res );
  }

  return fact( n, 1 );
}

factorial( 5 );           // 120
```

This version of `factorial(...)` is still recursive, but it's also optimizable with TCO, because both inner `fact(...)` calls are in *tail position*.

Note: It's important to note that TCO only applies if there's actually a tail call. If you write recursive functions without tail calls, the performance will still fall back to normal stack frame allocation, and the engines' limits on such recursive call stacks will still apply. Many recursive functions can be rewritten as we just showed with `factorial(...)`, but it takes careful attention to detail.

One reason that ES6 requires engines to implement TCO rather than leaving it up to their discretion is because the *lack of TCO* actually tends to reduce the chances that certain algorithms will be implemented in JS using recursion, for fear of the call stack limits.

If the lack of TCO in the engine would just gracefully degrade to slower performance in all cases, it wouldn't probably have been something that ES6 needed to *require*. But because the lack of TCO can actually make certain programs impractical, it's more an important feature of the language than just a hidden implementation detail.

ES6 guarantees that from now on, JS developers will be able to rely on this optimization across all ES6+ compliant browsers. That's a win for JS performance!

Review

Effectively benchmarking performance of a piece of code, especially to compare it to another option for that same code to see which approach is faster, requires careful attention to detail.

Rather than rolling your own statistically valid benchmarking logic, just use the Benchmark.js library, which does that for you. But be careful about how you author tests, because it's far too easy to construct a test that seems valid but that's actually flawed -- even tiny differences can skew the results to be completely unreliable.

It's important to get as many test results from as many different environments as possible to eliminate hardware/device bias. jsPerf.com is a fantastic website for crowdsourcing performance benchmark test runs.

Many common performance tests unfortunately obsess about irrelevant microperformance details like `x++` versus `++x`. Writing good tests means understanding how to focus on big picture concerns, like optimizing on the critical path, and avoiding falling into traps like different JS engines' implementation details.

Tail call optimization (TCO) is a required optimization as of ES6 that will make some recursive patterns practical in JS where they would have been impossible otherwise. TCO allows a function call in the *tail position* of another function to execute without needing any extra resources, which means the engine no longer needs to place arbitrary restrictions on call stack depth for recursive algorithms.

Chapters 1 and 2 went into quite a bit of detail about typical asynchronous programming patterns and how they're commonly solved with callbacks. But we also saw why callbacks are fatally limited in capability, which led us to Chapters 3 and 4, with Promises and generators offering a much more solid, trustable, and reasonable base to build your asynchrony on.

I referenced my own asynchronous library *asynquence* (<http://github.com/getify/asynquence>) -- "async" + "sequence" = "asynquence" -- several times in this book, and I want to now briefly explain how it works and why its unique design is important and helpful.

In the next appendix, we'll explore some advanced async patterns, but you'll probably want a library to make those palatable enough to be useful. We'll use *asynquence* to express those patterns, so you'll want to spend a little time here getting to know the library first.

asynquence is obviously not the only option for good async coding; certainly there are many great libraries in this space. But *asynquence* provides a unique perspective by combining the best of all these patterns into a single library, and moreover is built on a single basic abstraction: the (async) sequence.

My premise is that sophisticated JS programs often need bits and pieces of various different asynchronous patterns woven together, and this is usually left entirely up to each developer to figure out. Instead of having to bring in two or more different async libraries that focus on different aspects of asynchrony, *asynquence* unifies them into variated sequence steps, with just one core library to learn and deploy.

I believe the value is strong enough with *asynquence* to make async flow control programming with Promise-style semantics super easy to accomplish, so that's why we'll exclusively focus on that library here.

To begin, I'll explain the design principles behind *asynquence*, and then we'll illustrate how its API works with code examples.

Sequences, Abstraction Design

Understanding *asynquence* begins with understanding a fundamental abstraction: any series of steps for a task, whether they separately are synchronous or asynchronous, can be collectively thought of as a "sequence". In other words, a sequence is a container that represents a task, and is comprised of individual (potentially async) steps to complete that task.

Each step in the sequence is controlled under the covers by a Promise (see Chapter 3). That is, every step you add to a sequence implicitly creates a Promise that is wired to the previous end of the sequence. Because of the semantics of Promises, every single step

advancement in a sequence is asynchronous, even if you synchronously complete the step.

Moreover, a sequence will always proceed linearly from step to step, meaning that step 2 always comes after step 1 finishes, and so on.

Of course, a new sequence can be forked off an existing sequence, meaning the fork only occurs once the main sequence reaches that point in the flow. Sequences can also be combined in various ways, including having one sequence subsumed by another sequence at a particular point in the flow.

A sequence is kind of like a Promise chain. However, with Promise chains, there is no "handle" to grab that references the entire chain. Whichever Promise you have a reference to only represents the current step in the chain plus any other steps hanging off it. Essentially, you cannot hold a reference to a Promise chain unless you hold a reference to the first Promise in the chain.

There are many cases where it turns out to be quite useful to have a handle that references the entire sequence collectively. The most important of those cases is with sequence abort/cancel. As we covered extensively in Chapter 3, Promises themselves should never be able to be canceled, as this violates a fundamental design imperative: external immutability.

But sequences have no such immutability design principle, mostly because sequences are not passed around as future-value containers that need immutable value semantics. So sequences are the proper level of abstraction to handle abort/cancel behavior. *asynquence* sequences can be `abort()` ed at any time, and the sequence will stop at that point and not go for any reason.

There's plenty more reasons to prefer a sequence abstraction on top of Promise chains, for flow control purposes.

First, Promise chaining is a rather manual process -- one that can get pretty tedious once you start creating and chaining Promises across a wide swath of your programs -- and this tedium can act counterproductively to dissuade the developer from using Promises in places where they are quite appropriate.

Abstractions are meant to reduce boilerplate and tedium, so the sequence abstraction is a good solution to this problem. With Promises, your focus is on the individual step, and there's little assumption that you will keep the chain going. With sequences, the opposite approach is taken, assuming the sequence will keep having more steps added indefinitely.

This abstraction complexity reduction is especially powerful when you start thinking about higher-order Promise patterns (beyond `race([...])` and `all([...])`).

For example, in the middle of a sequence, you may want to express a step that is conceptually like a `try..catch` in that the step will always result in success, either the intended main success resolution or a positive nonerror signal for the caught error. Or, you might want to express a step that is like a retry/until loop, where it keeps trying the same step over and over until success occurs.

These sorts of abstractions are quite nontrivial to express using only Promise primitives, and doing so in the middle of an existing Promise chain is not pretty. But if you abstract your thinking to a sequence, and consider a step as a wrapper around a Promise, that step wrapper can hide such details, freeing you to think about the flow control in the most sensible way without being bothered by the details.

Second, and perhaps more importantly, thinking of async flow control in terms of steps in a sequence allows you to abstract out the details of what types of asynchronicity are involved with each individual step. Under the covers, a Promise will always control the step, but above the covers, that step can look either like a continuation callback (the simple default), or like a real Promise, or as a run-to-completion generator, or ... Hopefully, you get the picture.

Third, sequences can more easily be twisted to adapt to different modes of thinking, such as event-, stream-, or reactive-based coding. `asynquence` provides a pattern I call "reactive sequences" (which we'll cover later) as a variation on the "reactive observable" ideas in RxJS ("Reactive Extensions"), that lets a repeatable event fire off a new sequence instance each time. Promises are one-shot-only, so it's quite awkward to express repetitious asynchrony with Promises alone.

Another alternate mode of thinking inverts the resolution/control capability in a pattern I call "iterable sequences". Instead of each individual step internally controlling its own completion (and thus advancement of the sequence), the sequence is inverted so the advancement control is through an external iterator, and each step in the *iterable sequence* just responds to the `next(...)` *iterator* control.

We'll explore all of these different variations as we go throughout the rest of this appendix, so don't worry if we ran over those bits far too quickly just now.

The takeaway is that sequences are a more powerful and sensible abstraction for complex asynchrony than just Promises (Promise chains) or just generators, and `asynquence` is designed to express that abstraction with just the right level of sugar to make async programming more understandable and more enjoyable.

asynquence API

To start off, the way you create a sequence (an `asynquence` instance) is with the `ASQ(. .)` function. An `ASQ()` call with no parameters creates an empty initial sequence, whereas passing one or more values or functions to `ASQ(. .)` sets up the sequence with each argument representing the initial steps of the sequence.

Note: For the purposes of all code examples here, I will use the `asynquence` top-level identifier in global browser usage: `ASQ`. If you include and use `asynquence` through a module system (browser or server), you of course can define whichever symbol you prefer, and `asynquence` won't care!

Many of the API methods discussed here are built into the core of `asynquence`, but others are provided through including the optional "contrib" plug-ins package. See the documentation for `asynquence` for whether a method is built in or defined via plug-in:
<http://github.com/getify/asynquence>

Steps

If a function represents a normal step in the sequence, that function is invoked with the first parameter being the continuation callback, and any subsequent parameters being any messages passed on from the previous step. The step will not complete until the continuation callback is called. Once it's called, any arguments you pass to it will be sent along as messages to the next step in the sequence.

To add an additional normal step to the sequence, call `then(. .)` (which has essentially the exact same semantics as the `ASQ(. .)` call):

```

ASQ(
    // step 1
    function(done){
        setTimeout( function(){
            done( "Hello" );
        }, 100 );
    },
    // step 2
    function(done,greeting) {
        setTimeout( function(){
            done( greeting + " World" );
        }, 100 );
    }
)
// step 3
.then( function(done,msg){
    setTimeout( function(){
        done( msg.toUpperCase() );
    }, 100 );
} )
// step 4
.then( function(done,msg){
    console.log( msg );           // HELLO WORLD
} );

```

Note: Though the name `then(...)` is identical to the native Promises API, this `then(...)` is different. You can pass as few or as many functions or values to `then(...)` as you'd like, and each is taken as a separate step. There's no two-callback fulfilled/rejected semantics involved.

Unlike with Promises, where to chain one Promise to the next you have to create and `return` that Promise from a `then(...)` fulfillment handler, with `asynquence`, all you need to do is call the continuation callback -- I always call it `done()` but you can name it whatever suits you -- and optionally pass it completion messages as arguments.

Each step defined by `then(...)` is assumed to be asynchronous. If you have a step that's synchronous, you can either just call `done(...)` right away, or you can use the simpler `val(...)` step helper:

```
// step 1 (sync)
ASQ( function(done){
    done( "Hello" );      // manually synchronous
} )
// step 2 (sync)
.val( function(greeting){
    return greeting + " World";
} )
// step 3 (async)
.then( function(done,msg){
    setTimeout( function(){
        done( msg.toUpperCase() );
    }, 100 );
} )
// step 4 (sync)
.val( function(msg){
    console.log( msg );
} );
```

As you can see, `.val(...)`-invoked steps don't receive a continuation callback, as that part is assumed for you -- and the parameter list is less cluttered as a result! To send a message along to the next step, you simply use `return`.

Think of `.val(...)` as representing a synchronous "value-only" step, which is useful for synchronous value operations, logging, and the like.

Errors

One important difference with `asynquence` compared to Promises is with error handling.

With Promises, each individual Promise (step) in a chain can have its own independent error, and each subsequent step has the ability to handle the error or not. The main reason for this semantic comes (again) from the focus on individual Promises rather than on the chain (sequence) as a whole.

I believe that most of the time, an error in one part of a sequence is generally not recoverable, so the subsequent steps in the sequence are moot and should be skipped. So, by default, an error at any step of a sequence throws the entire sequence into error mode, and the rest of the normal steps are ignored.

If you *do* need to have a step where its error is recoverable, there are several different API methods that can accommodate, such as `try(..)` -- previously mentioned as a kind of `try..catch` step -- or `until(..)` -- a retry loop that keeps attempting the step until it succeeds or you manually `break()` the loop. `asynquence` even has `pThen(..)` and

`pCatch(...)` methods, which work identically to how normal Promise `then(...)` and `catch(...)` work (see Chapter 3), so you can do localized mid-sequence error handling if you so choose.

The point is, you have both options, but the more common one in my experience is the default. With Promises, to get a chain of steps to ignore all steps once an error occurs, you have to take care not to register a rejection handler at any step; otherwise, that error gets swallowed as handled, and the sequence may continue (perhaps unexpectedly). This kind of desired behavior is a bit awkward to properly and reliably handle.

To register a sequence error notification handler, `asynquence` provides an `or(...)` sequence method, which also has an alias of `onerror(...)`. You can call this method anywhere in the sequence, and you can register as many handlers as you'd like. That makes it easy for multiple different consumers to listen in on a sequence to know if it failed or not; it's kind of like an error event handler in that respect.

Just like with Promises, all JS exceptions become sequence errors, or you can programmatically signal a sequence error:

```
var sq = ASQ( function(done){
    setTimeout( function(){
        // signal an error for the sequence
        done.fail( "Oops" );
    }, 100 );
} )
.then( function(done){
    // will never get here
} )
.or( function(err){
    console.log( err );           // Oops
} )
.then( function(done){
    // won't get here either
} );

// later

sq.or( function(err){
    console.log( err );           // Oops
} );
```

Another really important difference with error handling in `asynquence` compared to native Promises is the default behavior of "unhandled exceptions". As we discussed at length in Chapter 3, a rejected Promise without a registered rejection handler will just silently hold (aka swallow) the error; you have to remember to always end a chain with a final `catch(...)`.

In *asynquence*, the assumption is reversed.

If an error occurs on a sequence, and it **at that moment** has no error handlers registered, the error is reported to the `console`. In other words, unhandled rejections are by default always reported so as not to be swallowed and missed.

As soon as you register an error handler against a sequence, it opts that sequence out of such reporting, to prevent duplicate noise.

There may, in fact, be cases where you want to create a sequence that may go into the error state before you have a chance to register the handler. This isn't common, but it can happen from time to time.

In those cases, you can also **opt a sequence instance out** of error reporting by calling `defer()` on the sequence. You should only opt out of error reporting if you are sure that you're going to eventually handle such errors:

```
var sq1 = ASQ( function(done){
    doesnt.Exist();           // will throw exception to console
} );

var sq2 = ASQ( function(done){
    doesnt.Exist();           // will throw only a sequence error
} )
// opt-out of error reporting
.defer();

setTimeout( function(){
    sq1.or( function(err){
        console.log( err );   // ReferenceError
    } );

    sq2.or( function(err){
        console.log( err );   // ReferenceError
    } );
}, 100 );

// ReferenceError (from sq1)
```

This is better error handling behavior than Promises themselves have, because it's the Pit of Success, not the Pit of Failure (see Chapter 3).

Note: If a sequence is piped into (aka subsumed by) another sequence -- see "Combining Sequences" for a complete description -- then the source sequence is opted out of error reporting, but now the target sequence's error reporting or lack thereof must be considered.

Parallel Steps

Not all steps in your sequences will have just a single (async) task to perform; some will need to perform multiple steps "in parallel" (concurrently). A step in a sequence in which multiple substeps are processing concurrently is called a `gate(..)` -- there's an `all(..)` alias if you prefer -- and is directly symmetric to native `Promise.all([...])`.

If all the steps in the `gate(..)` complete successfully, all success messages will be passed to the next sequence step. If any of them generate errors, the whole sequence immediately goes into an error state.

Consider:

```
ASQ( function(done){
    setTimeout( done, 100 );
} )
.gate(
    function(done){
        setTimeout( function(){
            done( "Hello" );
        }, 100 );
    },
    function(done){
        setTimeout( function(){
            done( "World", "!" );
        }, 100 );
    }
)
.val( function(msg1,msg2){
    console.log( msg1 );      // Hello
    console.log( msg2 );      // [ "World", "!" ]
} );
```

For illustration, let's compare that example to native Promises:

```

new Promise( function(resolve,reject){
    setTimeout( resolve, 100 );
} )
.then( function(){
    return Promise.all( [
        new Promise( function(resolve,reject){
            setTimeout( function(){
                resolve( "Hello" );
            }, 100 );
        } ),
        new Promise( function(resolve,reject){
            setTimeout( function(){
                // note: we need a [ ] array here
                resolve( [ "World", "!" ] );
            }, 100 );
        } )
    ] );
} )
.then( function(msgs){
    console.log( msgs[0] ); // Hello
    console.log( msgs[1] ); // [ "World", "!" ]
} );

```

Yuck. Promises require a lot more boilerplate overhead to express the same asynchronous flow control. That's a great illustration of why the *asynquence* API and abstraction make dealing with Promise steps a lot nicer. The improvement only goes higher the more complex your asynchrony is.

Step Variations

There are several variations in the contrib plug-ins on *asynquence*'s `gate(..)` step type that can be quite helpful:

- `any(..)` is like `gate(..)`, except just one segment has to eventually succeed to proceed on the main sequence.
- `first(..)` is like `any(..)`, except as soon as any segment succeeds, the main sequence proceeds (ignoring subsequent results from other segments).
- `race(..)` (symmetric with `Promise.race(..)`) is like `first(..)`, except the main sequence proceeds as soon as any segment completes (either success or failure).
- `last(..)` is like `any(..)`, except only the latest segment to complete successfully sends its message(s) along to the main sequence.
- `none(..)` is the inverse of `gate(..)`: the main sequence proceeds only if all the segments fail (with all segment error message(s) transposed as success message(s) and vice versa).

Let's first define some helpers to make illustration cleaner:

```
function success1(done) {
    setTimeout( function(){
        done( 1 );
    }, 100 );
}

function success2(done) {
    setTimeout( function(){
        done( 2 );
    }, 100 );
}

function failure3(done) {
    setTimeout( function(){
        done.fail( 3 );
    }, 100 );
}

function output(msg) {
    console.log( msg );
}
```

Now, let's demonstrate these `gate(..)` step variations:

```

ASQ().race(
  failure3,
  success1
)
.or( output );           // 3

ASQ().any(
  success1,
  failure3,
  success2
)
.val( function(){
  var args = [].slice.call( arguments );
  console.log(
    args      // [ 1, undefined, 2 ]
  );
} );
}

ASQ().first(
  failure3,
  success1,
  success2
)
.val( output );           // 1

ASQ().last(
  failure3,
  success1,
  success2
)
.val( output );           // 2

ASQ().none(
  failure3
)
.val( output )           // 3
.none(
  failure3
  success1
)
.or( output );           // 1

```

Another step variation is `map(..)`, which lets you asynchronously map elements of an array to different values, and the step doesn't proceed until all the mappings are complete.

`map(..)` is very similar to `gate(..)`, except it gets the initial values from an array instead of from separately specified functions, and also because you define a single function callback to operate on each value:

```

function double(x,done) {
    setTimeout( function(){
        done( x * 2 );
    }, 100 );
}

ASQ().map( [1,2,3], double )
.val( output );           // [2,4,6]

```

Also, `map(...)` can receive either of its parameters (the array or the callback) from messages passed from the previous step:

```

function plusOne(x,done) {
    setTimeout( function(){
        done( x + 1 );
    }, 100 );
}

ASQ( [1,2,3] )
.map( double )           // message `[1,2,3]` comes in
.map( plusOne )          // message `[2,4,6]` comes in
.val( output );          // [3,5,7]

```

Another variation is `waterfall(..)`, which is kind of like a mixture between `gate(..)`'s message collection behavior but `then(..)`'s sequential processing.

Step 1 is first executed, then the success message from step 1 is given to step 2, and then both success messages go to step 3, and then all three success messages go to step 4, and so on, such that the messages sort of collect and cascade down the "waterfall".

Consider:

```

function double(done) {
    var args = [].slice.call( arguments, 1 );
    console.log( args );

    setTimeout( function(){
        done( args[args.length - 1] * 2 );
    }, 100 );
}

ASQ( 3 )
.waterval(
    double,           // [ 3 ]
    double,           // [ 6 ]
    double,           // [ 6, 12 ]
    double           // [ 6, 12, 24 ]
)
.val( function(){
    var args = [].slice.call( arguments );
    console.log( args ); // [ 6, 12, 24, 48 ]
} );

```

If at any point in the "waterfall" an error occurs, the whole sequence immediately goes into an error state.

Error Tolerance

Sometimes you want to manage errors at the step level and not let them necessarily send the whole sequence into the error state. *asynquence* offers two step variations for that purpose.

`try(..)` attempts a step, and if it succeeds, the sequence proceeds as normal, but if the step fails, the failure is turned into a success message formated as `{ catch: .. }` with the error message(s) filled in:

```

ASQ()
.try( success1 )
.val( output )           // 1
.try( failure3 )
.val( output )           // { catch: 3 }
.or( function(err){
    // never gets here
} );

```

You could instead set up a retry loop using `until(..)`, which tries the step and if it fails, retries the step again on the next event loop tick, and so on.

This retry loop can continue indefinitely, but if you want to break out of the loop, you can call the `break()` flag on the completion trigger, which sends the main sequence into an error state:

```
var count = 0;

ASQ( 3 )
.until( double )
.val( output ) // 6
.until( function(done){
    count++;

    setTimeout( function(){
        if (count < 5) {
            done.fail();
        }
        else {
            // break out of the `until(..)` retry loop
            done.break( "Oops" );
        }
    }, 100 );
} )
.or( output ); // Oops
```

Promise-Style Steps

If you would prefer to have, inline in your sequence, Promise-style semantics like Promises' `then(..)` and `catch(..)` (see Chapter 3), you can use the `pThen` and `pCatch` plug-ins:

```
ASQ( 21 )
.pThen( function(msg){
    return msg * 2;
} )
.pThen( output ) // 42
.pThen( function(){
    // throw an exception
    doesnt.Exist();
} )
.pCatch( function(err){
    // caught the exception (rejection)
    console.log( err ); // ReferenceError
} )
.val( function(){
    // main sequence is back in a
    // success state because previous
    // exception was caught by
    // `pCatch(..)`
} );
```

`pThen(...)` and `pCatch(...)` are designed to run in the sequence, but behave as if it was a normal Promise chain. As such, you can either resolve genuine Promises or *asynquence* sequences from the "fulfillment" handler passed to `pThen(...)` (see Chapter 3).

Forking Sequences

One feature that can be quite useful about Promises is that you can attach multiple `then(...)` handler registrations to the same promise, effectively "forking" the flow-control at that promise:

```
var p = Promise.resolve( 21 );

// fork 1 (from `p`)
p.then( function(msg){
    return msg * 2;
} )
.then( function(msg){
    console.log( msg );           // 42
} )

// fork 2 (from `p`)
p.then( function(msg){
    console.log( msg );           // 21
} );
```

The same "forking" is easy in *asynquence* with `fork()`:

```
var sq = ASQ(...).then(...).then(...);

var sq2 = sq.fork();

// fork 1
sq.then(...);

// fork 2
sq2.then(...);
```

Combining Sequences

The reverse of `fork()` ing, you can combine two sequences by subsuming one into another, using the `seq(...)` instance method:

```

var sq = ASQ( function(done){
    setTimeout( function(){
        done( "Hello World" );
    }, 200 );
} );

ASQ( function(done){
    setTimeout( done, 100 );
} )
// subsume `sq` sequence into this sequence
.seq( sq )
.val( function(msg){
    console.log( msg );           // Hello World
} )

```

`seq(..)` can either accept a sequence itself, as shown here, or a function. If a function, it's expected that the function when called will return a sequence, so the preceding code could have been done with:

```

// ...
.seq( function(){
    return sq;
} )
// ...

```

Also, that step could instead have been accomplished with a `pipe(..)`:

```

// ...
.then( function(done){
    // pipe `sq` into the `done` continuation callback
    sq.pipe( done );
} )
// ...

```

When a sequence is subsumed, both its success message stream and its error stream are piped in.

Note: As mentioned in an earlier note, piping (manually with `pipe(..)` or automatically with `seq(..)`) opts the source sequence out of error-reporting, but doesn't affect the error reporting status of the target sequence.

Value and Error Sequences

If any step of a sequence is just a normal value, that value is just mapped to that step's completion message:

```
var sq = ASQ( 42 );

sq.val( function(msg){
    console.log( msg );           // 42
} );
```

If you want to make a sequence that's automatically errored:

```
var sq = ASQ.failed( "Oops" );

ASQ()
.seq( sq )
.val( function(msg){
    // won't get here
} )
.or( function(err){
    console.log( err );          // Oops
} );
```

You also may want to automatically create a delayed-value or a delayed-error sequence. Using the `after` and `failAfter` contrib plug-ins, this is easy:

```
var sq1 = ASQ.after( 100, "Hello", "World" );
var sq2 = ASQ.failAfter( 100, "Oops" );

sq1.val( function(msg1,msg2){
    console.log( msg1, msg2 );      // Hello World
} );

sq2.or( function(err){
    console.log( err );            // Oops
} );
```

You can also insert a delay in the middle of a sequence using `after(..)`:

```
ASQ( 42 )
// insert a delay into the sequence
.after( 100 )
.val( function(msg){
    console.log( msg );           // 42
} );
```

Promises and Callbacks

I think `asynquence` sequences provide a lot of value on top of native Promises, and for the most part you'll find it more pleasant and more powerful to work at that level of abstraction. However, integrating `asynquence` with other non-`asynquence` code will be a reality.

You can easily subsume a promise (e.g., `thenable` -- see Chapter 3) into a sequence using the `promise(...)` instance method:

```
var p = Promise.resolve( 42 );

ASQ()
.promise( p )           // could also: `function(){ return p; }`
.val( function(msg){
    console.log( msg );   // 42
} );
```

And to go the opposite direction and fork/vend a promise from a sequence at a certain step, use the `toPromise` contrib plug-in:

```
var sq = ASQ.after( 100, "Hello World" );

sq.toPromise()
// this is a standard promise chain now
.then( function(msg){
    return msg.toUpperCase();
} )
.then( function(msg){
    console.log( msg );      // HELLO WORLD
} );
```

To adapt `asynquence` to systems using callbacks, there are several helper facilities. To automatically generate an "error-first style" callback from your sequence to wire into a callback-oriented utility, use `errfcb`:

```

var sq = ASQ( function(done){
    // note: expecting "error-first style" callback
    someAsyncFuncWithCB( 1, 2, done.errfcb )
} )
.val( function(msg){
    // ..
} )
.or( function(err){
    // ..
} );

// note: expecting "error-first style" callback
anotherAsyncFuncWithCB( 1, 2, sq.errfcb() );

```

You also may want to create a sequence-wrapped version of a utility -- compare to "promisory" in Chapter 3 and "thunkory" in Chapter 4 -- and *asynquence* provides `ASQ.wrap(..)` for that purpose:

```

var coolUtility = ASQ.wrap( someAsyncFuncWithCB );

coolUtility( 1, 2 )
.val( function(msg){
    // ..
} )
.or( function(err){
    // ..
} );

```

Note: For the sake of clarity (and for fun!), let's coin yet another term, for a sequence-producing function that comes from `ASQ.wrap(..)`, like `coolUtility` here. I propose "sequory" ("sequence" + "factory").

Iterable Sequences

The normal paradigm for a sequence is that each step is responsible for completing itself, which is what advances the sequence. Promises work the same way.

The unfortunate part is that sometimes you need external control over a Promise/step, which leads to awkward "capability extraction".

Consider this Promises example:

```

var domready = new Promise( function(resolve,reject){
    // don't want to put this here, because
    // it belongs logically in another part
    // of the code
    document.addEventListener( "DOMContentLoaded", resolve );
} );

// ...

domready.then( function(){
    // DOM is ready!
} );

```

The "capability extraction" anti-pattern with Promises looks like this:

```

var ready;

var domready = new Promise( function(resolve,reject){
    // extract the `resolve()` capability
    ready = resolve;
} );

// ...

domready.then( function(){
    // DOM is ready!
} );

// ...

document.addEventListener( "DOMContentLoaded", ready );

```

Note: This anti-pattern is an awkward code smell, in my opinion, but some developers like it, for reasons I can't grasp.

asynquence offers an inverted sequence type I call "iterable sequences", which externalizes the control capability (it's quite useful in use cases like the `domready`):

```
// note: `domready` here is an *iterator* that
// controls the sequence
var domready = ASQ iterable();

// ...

domready.val( function(){
    // DOM is ready
} );

// ...

document.addEventListener( "DOMContentLoaded", domready.next );
```

There's more to iterable sequences than what we see in this scenario. We'll come back to them in Appendix B.

Running Generators

In Chapter 4, we derived a utility called `run(...)` which can run generators to completion, listening for `yield` ed Promises and using them to `async resume` the generator. `asynquence` has just such a utility built in, called `runner(..)`.

Let's first set up some helpers for illustration:

```
function doublePr(x) {
    return new Promise( function(resolve,reject){
        setTimeout( function(){
            resolve( x * 2 );
        }, 100 );
    } );
}

function doubleSeq(x) {
    return ASQ( function(done){
        setTimeout( function(){
            done( x * 2 )
        }, 100 );
    } );
}
```

Now, we can use `runner(..)` as a step in the middle of a sequence:

```

ASQ( 10, 11 )
.runner( function*(token){
    var x = token.messages[0] + token.messages[1];

    // yield a real promise
    x = yield doublePr( x );

    // yield a sequence
    x = yield doubleSeq( x );

    return x;
} )
.val( function(msg){
    console.log( msg );           // 84
} );

```

Wrapped Generators

You can also create a self-packaged generator -- that is, a normal function that runs your specified generator and returns a sequence for its completion -- by `ASQ.wrap(...)` ing it:

```

var foo = ASQ.wrap( function*(token){
    var x = token.messages[0] + token.messages[1];

    // yield a real promise
    x = yield doublePr( x );

    // yield a sequence
    x = yield doubleSeq( x );

    return x;
}, { gen: true } );

// ...

foo( 8, 9 )
.val( function(msg){
    console.log( msg );           // 68
} );

```

There's a lot more awesome that `runner(..)` is capable of, but we'll come back to that in Appendix B.

Review

asynquence is a simple abstraction -- a sequence is a series of (async) steps -- on top of Promises, aimed at making working with various asynchronous patterns much easier, without any compromise in capability.

There are other goodies in the *asynquence* core API and its contrib plug-ins beyond what we saw in this appendix, but we'll leave that as an exercise for the reader to go check the rest of the capabilities out.

You've now seen the essence and spirit of *asynquence*. The key take away is that a sequence is comprised of steps, and those steps can be any of dozens of different variations on Promises, or they can be a generator-run, or... The choice is up to you, you have all the freedom to weave together whatever async flow control logic is appropriate for your tasks. No more library switching to catch different async patterns.

If these *asynquence* snippets have made sense to you, you're now pretty well up to speed on the library; it doesn't take that much to learn, actually!

If you're still a little fuzzy on how it works (or why!), you'll want to spend a little more time examining the previous examples and playing around with *asynquence* yourself, before going on to the next appendix. Appendix B will push *asynquence* into several more advanced and powerful async patterns.

Appendix B: Advanced Async Patterns

Appendix A introduced the `asynquence` library for sequence-oriented async flow control, primarily based on Promises and generators.

Now we'll explore other advanced asynchronous patterns built on top of that existing understanding and functionality, and see how `asynquence` makes those sophisticated async techniques easy to mix and match in our programs without needing lots of separate libraries.

Iterable Sequences

We introduced `asynquence`'s iterable sequences in the previous appendix, but we want to revisit them in more detail.

To refresh, recall:

```
var domready = ASQ.iterable();

// ...

domready.val( function(){
    // DOM is ready
} );

// ...

document.addEventListener( "DOMContentLoaded", domready.next );
```

Now, let's define a sequence of multiple steps as an iterable sequence:

```
var steps = ASQ iterable();

steps
  .then( function STEP1(x){
    return x * 2;
  })
  .then( function STEP2(x){
    return x + 3;
  })
  .then( function STEP3(x){
    return x * 4;
  });

steps.next().value;      // 16
steps.next().value;      // 19
steps.next().value;      // 76
steps.next().done;       // true
```

As you can see, an iterable sequence is a standard-compliant *iterator* (see Chapter 4). So, it can be iterated with an ES6 `for..of` loop, just like a generator (or any other *iterable*) can:

```
var steps = ASQ iterable();

steps
  .then( function STEP1(){ return 2; } )
  .then( function STEP2(){ return 4; } )
  .then( function STEP3(){ return 6; } )
  .then( function STEP4(){ return 8; } )
  .then( function STEP5(){ return 10; } );

for (var v of steps) {
  console.log( v );
}
// 2 4 6 8 10
```

Beyond the event triggering example shown in the previous appendix, iterable sequences are interesting because in essence they can be seen as a stand-in for generators or Promise chains, but with even more flexibility.

Consider a multiple Ajax request example -- we've seen the same scenario in Chapters 3 and 4, both as a Promise chain and as a generator, respectively -- expressed as an iterable sequence:

```
// sequence-aware ajax
var request = ASQ.wrap( ajax );

ASQ( "http://some.url.1" )
.runner(
  ASQ iterable()

  .then( function STEP1(token){
    var url = token.messages[0];
    return request( url );
  } )

  .then( function STEP2(resp){
    return ASQ().gate(
      request( "http://some.url.2/?v=" + resp ),
      request( "http://some.url.3/?v=" + resp )
    );
  } )

  .then( function STEP3(r1,r2){ return r1 + r2; } )
)
.val( function(msg){
  console.log( msg );
} );
```

The iterable sequence expresses a sequential series of (sync or async) steps that looks awfully similar to a Promise chain -- in other words, it's much cleaner looking than just plain nested callbacks, but not quite as nice as the `yield`-based sequential syntax of generators.

But we pass the iterable sequence into `ASQ#runner(...)`, which runs it to completion the same as if it was a generator. The fact that an iterable sequence behaves essentially the same as a generator is notable for a couple of reasons.

First, iterable sequences are kind of a pre-ES6 equivalent to a certain subset of ES6 generators, which means you can either author them directly (to run anywhere), or you can author ES6 generators and transpile/convert them to iterable sequences (or Promise chains for that matter!).

Thinking of an `async-run-to-completion` generator as just syntactic sugar for a Promise chain is an important recognition of their isomorphic relationship.

Before we move on, we should note that the previous snippet could have been expressed in `asynquence` as:

```

ASQ( "http://some.url.1" )
  .seq( /*STEP 1*/ request )
  .seq( function STEP2(resp){
    return ASQ().gate(
      request( "http://some.url.2/?v=" + resp ),
      request( "http://some.url.3/?v=" + resp )
    );
  } )
  .val( function STEP3(r1,r2){ return r1 + r2; } )
  .val( function(msg){
    console.log( msg );
  } );

```

Moreover, step 2 could have even been expressed as:

```

.gate(
  function STEP2a(done,resp) {
    request( "http://some.url.2/?v=" + resp )
    .pipe( done );
  },
  function STEP2b(done,resp) {
    request( "http://some.url.3/?v=" + resp )
    .pipe( done );
  }
)

```

So, why would we go to the trouble of expressing our flow control as an iterable sequence in a `ASQ#runner(..)` step, when it seems like a simpler/flatter `asyquence` chain does the job well?

Because the iterable sequence form has an important trick up its sleeve that gives us more capability. Read on.

Extending Iterable Sequences

Generators, normal `asyquence` sequences, and Promise chains, are all **eagerly evaluated** -- whatever flow control is expressed initially *is* the fixed flow that will be followed.

However, iterable sequences are **lazily evaluated**, which means that during execution of the iterable sequence, you can extend the sequence with more steps if desired.

Note: You can only append to the end of an iterable sequence, not inject into the middle of the sequence.

Let's first look at a simpler (synchronous) example of that capability to get familiar with it:

```
function double(x) {
  x *= 2;

  // should we keep extending?
  if (x < 500) {
    isq.then( double );
  }

  return x;
}

// setup single-step iterable sequence
var isq = ASQ iterable().then( double );

for (var v = 10, ret;
     (ret = isq.next( v )) && !ret.done;
  ) {
  v = ret.value;
  console.log( v );
}
```

The iterable sequence starts out with only one defined step (`isq.then(double)`), but the sequence keeps extending itself under certain conditions (`x < 500`). Both `asynquence` sequences and Promise chains technically *can* do something similar, but we'll see in a little bit why their capability is insufficient.

Though this example is rather trivial and could otherwise be expressed with a `while` loop in a generator, we'll consider more sophisticated cases.

For instance, you could examine the response from an Ajax request and if it indicates that more data is needed, you conditionally insert more steps into the iterable sequence to make the additional request(s). Or you could conditionally add a value-formatting step to the end of your Ajax handling.

Consider:

```

var steps = ASQ iterable()

.then( function STEP1(token){
  var url = token.messages[0].url;

  // was an additional formatting step provided?
  if (token.messages[0].format) {
    steps.then( token.messages[0].format );
  }

  return request( url );
} )

.then( function STEP2(resp){
  // add another Ajax request to the sequence?
  if (/x1/.test( resp )) {
    steps.then( function STEP5(text){
      return request(
        "http://some.url.4/?v=" + text
      );
    } );
  }
}

return ASQ().gate(
  request( "http://some.url.2/?v=" + resp ),
  request( "http://some.url.3/?v=" + resp )
);
} )

.then( function STEP3(r1,r2){ return r1 + r2; } );

```

You can see in two different places where we conditionally extend `steps` with `steps.then(...)`. And to run this `steps` iterable sequence, we just wire it into our main program flow with an `asynquence` sequence (called `main` here) using `ASQ#runner(..)`:

```

var main = ASQ( {
  url: "http://some.url.1",
  format: function STEP4(text){
    return text.toUpperCase();
  }
} )
.runner( steps )
.val( function(msg){
  console.log( msg );
} );

```

Can the flexibility (conditional behavior) of the `steps` iterable sequence be expressed with a generator? Kind of, but we have to rearrange the logic in a slightly awkward way:

```

function *steps(token) {
    // **STEP 1**
    var resp = yield request( token.messages[0].url );

    // **STEP 2**
    var rvals = yield ASQ().gate(
        request( "http://some.url.2/?v=" + resp ),
        request( "http://some.url.3/?v=" + resp )
    );

    // **STEP 3**
    var text = rvals[0] + rvals[1];

    // **STEP 4**
    // was an additional formatting step provided?
    if (token.messages[0].format) {
        text = yield token.messages[0].format( text );
    }

    // **STEP 5**
    // need another Ajax request added to the sequence?
    if (/foobar/.test( resp )) {
        text = yield request(
            "http://some.url.4/?v=" + text
        );
    }

    return text;
}

// note: `*steps()` can be run by the same `ASQ` sequence
// as `steps` was previously

```

Setting aside the already identified benefits of the sequential, synchronous-looking syntax of generators (see Chapter 4), the `steps` logic had to be reordered in the `*steps()` generator form, to fake the dynamicism of the extendable iterable sequence `steps`.

What about expressing the functionality with Promises or sequences, though? You *can* do something like this:

```

var steps = something( ... )
.then( ... )
.then( function(...){
    // ...

    // extending the chain, right?
    steps = steps.then( ... );

    // ...
})
.then( ... );

```

The problem is subtle but important to grasp. So, consider trying to wire up our `steps` Promise chain into our main program flow -- this time expressed with Promises instead of `asynquence`:

```

var main = Promise.resolve( {
    url: "http://some.url.1",
    format: function STEP4(text){
        return text.toUpperCase();
    }
} )
.then( function(...){
    return steps;           // hint!
} )
.val( function(msg){
    console.log( msg );
} );

```

Can you spot the problem now? Look closely!

There's a race condition for sequence `steps` ordering. When you `return steps`, at that moment `steps` *might* be the originally defined promise chain, or it might now point to the extended promise chain via the `steps = steps.then(...)` call, depending on what order things happen.

Here are the two possible outcomes:

- If `steps` is still the original promise chain, once it's later "extended" by `steps = steps.then(...)`, that extended promise on the end of the chain is **not** considered by the `main` flow, as it's already tapped the `steps` chain. This is the unfortunately limiting **eager evaluation**.
- If `steps` is already the extended promise chain, it works as we expect in that the extended promise is what `main` taps.

Other than the obvious fact that a race condition is intolerable, the first case is the concern; it illustrates **eager evaluation** of the promise chain. By contrast, we easily extended the iterable sequence without such issues, because iterable sequences are **lazily evaluated**.

The more dynamic you need your flow control, the more iterable sequences will shine.

Tip: Check out more information and examples of iterable sequences on the *asynquence* site (<https://github.com/getify/asynquence/blob/master/README.md#iterable-sequences>).

Event Reactive

It should be obvious from (at least!) Chapter 3 that Promises are a very powerful tool in your async toolbox. But one thing that's clearly lacking is in their capability to handle streams of events, as a Promise can only be resolved once. And frankly, this exact same weakness is true of plain *asynquence* sequences, as well.

Consider a scenario where you want to fire off a series of steps every time a certain event is fired. A single Promise or sequence cannot represent all occurrences of that event. So, you have to create a whole new Promise chain (or sequence) for *each* event occurrence, such as:

```
listener.on( "foobar", function(data){  
  
    // create a new event handling promise chain  
    new Promise( function(resolve,reject){  
        // ..  
    } )  
    .then( .. )  
    .then( .. );  
  
} );
```

The base functionality we need is present in this approach, but it's far from a desirable way to express our intended logic. There are two separate capabilities conflated in this paradigm: the event listening, and responding to the event; separation of concerns would implore us to separate out these capabilities.

The carefully observant reader will see this problem as somewhat symmetrical to the problems we detailed with callbacks in Chapter 2; it's kind of an inversion of control problem.

Imagine uninverting this paradigm, like so:

```

var observable = listener.on( "foobar" );

// later
observable
  .then( ... )
  .then( ... );

// elsewhere
observable
  .then( ... )
  .then( ... );

```

The `observable` value is not exactly a Promise, but you can *observe* it much like you can observe a Promise, so it's closely related. In fact, it can be observed many times, and it will send out notifications every time its event (`"foobar"`) occurs.

Tip: This pattern I've just illustrated is a **massive simplification** of the concepts and motivations behind reactive programming (aka RP), which has been implemented/expounded upon by several great projects and languages. A variation on RP is functional reactive programming (FRP), which refers to applying functional programming techniques (immutability, referential integrity, etc.) to streams of data. "Reactive" refers to spreading this functionality out over time in response to events. The interested reader should consider studying "Reactive Observables" in the fantastic "Reactive Extensions" library ("RxJS" for JavaScript) by Microsoft (<http://rxjs.codeplex.com/>); it's much more sophisticated and powerful than I've just shown. Also, Andre Staltz has an excellent write-up (<https://gist.github.com/staltz/868e7e9bc2a7b8c1f754>) that pragmatically lays out RP in concrete examples.

ES7 Observables

At the time of this writing, there's an early ES7 proposal for a new data type called "Observable" (<https://github.com/jhusain/asyncgenerator#introducing-observable>), which in spirit is similar to what we've laid out here, but is definitely more sophisticated.

The notion of this kind of Observable is that the way you "subscribe" to the events from a stream is to pass in a generator -- actually the *iterator* is the interested party -- whose `next(...)` method will be called for each event.

You could imagine it sort of like this:

```
// `someEventStream` is a stream of events, like from
// mouse clicks, and the like.

var observer = new Observer( someEventStream, function*(){
  while (var evt = yield) {
    console.log( evt );
  }
});
```

The generator you pass in will `yield` pause the `while` loop waiting for the next event. The *iterator* attached to the generator instance will have its `next(..)` called each time `someEventStream` has a new event published, and so that event data will resume your generator/*iterator* with the `evt` data.

In the subscription to events functionality here, it's the *iterator* part that matters, not the generator. So conceptually you could pass in practically any iterable, including `ASQ iterable()` iterable sequences.

Interestingly, there are also proposed adapters to make it easy to construct Observables from certain types of streams, such as `fromEvent(..)` for DOM events. If you look at a suggested implementation of `fromEvent(..)` in the earlier linked ES7 proposal, it looks an awful lot like the `ASQ.react(..)` we'll see in the next section.

Of course, these are all early proposals, so what shakes out may very well look/behave differently than shown here. But it's exciting to see the early alignments of concepts across different libraries and language proposals!

Reactive Sequences

With that crazy brief summary of Observables (and F/RP) as our inspiration and motivation, I will now illustrate an adaptation of a small subset of "Reactive Observables," which I call "Reactive Sequences."

First, let's start with how to create an Observable, using an `asynquence` plug-in utility called `react(..)`:

```
var observable = ASQ.react( function setup(next){
  listener.on( "foobar", next );
} );
```

Now, let's see how to define a sequence that "reacts" -- in F/RP, this is typically called "subscribing" -- to that `observable`:

```
observable
.seq( ... )
.then( ... )
.val( ... );
```

So, you just define the sequence by chaining off the Observable. That's easy, huh?

In F/RP, the stream of events typically channels through a set of functional transforms, like `scan(..)`, `map(..)`, `reduce(..)`, and so on. With reactive sequences, each event channels through a new instance of the sequence. Let's look at a more concrete example:

```
ASQ.react( function setup(next){
  document.getElementById( "mybtn" )
    .addEventListener( "click", next, false );
} )
.seq( function(evt){
  var btnID = evt.target.id;
  return request(
    "http://some.url.1/?id=" + btnID
  );
} )
.val( function(text){
  console.log( text );
} );
```

The "reactive" portion of the reactive sequence comes from assigning one or more event handlers to invoke the event trigger (calling `next(..)`).

The "sequence" portion of the reactive sequence is exactly like the sequences we've already explored: each step can be whatever asynchronous technique makes sense, from continuation callback to Promise to generator.

Once you set up a reactive sequence, it will continue to initiate instances of the sequence as long as the events keep firing. If you want to stop a reactive sequence, you can call `stop()`.

If a reactive sequence is `stop()`'d, you likely want the event handler(s) to be unregistered as well; you can register a teardown handler for this purpose:

```

var sq = ASQ.react( function setup(next,registerTeardown){
  var btn = document.getElementById( "mybtn" );

  btn.addEventListener( "click", next, false );

  // will be called once `sq.stop()` is called
  registerTeardown( function(){
    btn.removeEventListener( "click", next, false );
  } );
} )
.seq( ... )
.then( ... )
.val( ... );

// later
sq.stop();

```

Note: The `this` binding reference inside the `setup(..)` handler is the same `sq` reactive sequence, so you can use the `this` reference to add to the reactive sequence definition, call methods like `stop()`, and so on.

Here's an example from the Node.js world, using reactive sequences to handle incoming HTTP requests:

```

var server = http.createServer();
server.listen(8000);

// reactive observer
var request = ASQ.react( function setup(next,registerTeardown){
  server.addListener( "request", next );
  server.addListener( "close", this.stop );

  registerTeardown( function(){
    server.removeListener( "request", next );
    server.removeListener( "close", request.stop );
  } );
});

// respond to requests
request
.seq( pullFromDatabase )
.val( function(data,res){
  res.end( data );
} );

// node teardown
process.on( "SIGINT", request.stop );

```

The `next(..)` trigger can also adapt to node streams easily, using `onStream(..)` and `unStream(..)`:

```
ASQ.react( function setup(next){
  var fstream = fs.createReadStream( "/some/file" );

  // pipe the stream's "data" event to `next(..)`
  next.onStream( fstream );

  // listen for the end of the stream
  fstream.on( "end", function(){
    next.unStream( fstream );
  } );
} )
.seq( ... )
.then( ... )
.val( ... );
```

You can also use sequence combinations to compose multiple reactive sequence streams:

```
var sq1 = ASQ.react( ... ).seq( ... ).then( ... );
var sq2 = ASQ.react( ... ).seq( ... ).then( ... );

var sq3 = ASQ.react( ... )
.gate(
  sq1,
  sq2
)
.then( ... );
```

The main takeaway is that `ASQ.react(..)` is a lightweight adaptation of F/RP concepts, enabling the wiring of an event stream to a sequence, hence the term "reactive sequence." Reactive sequences are generally capable enough for basic reactive uses.

Note: Here's an example of using `ASQ.react(..)` in managing UI state (<http://jsbin.com/rozipaki/6/edit?js,output>), and another example of handling HTTP request/response streams with `ASQ.react(..)` (<https://gist.github.com/getify/bba5ec0de9d6047b720e>).

Generator Coroutine

Hopefully Chapter 4 helped you get pretty familiar with ES6 generators. In particular, we want to revisit the "Generator Concurrency" discussion, and push it even further.

We imagined a `runAll(...)` utility that could take two or more generators and run them concurrently, letting them cooperatively `yield` control from one to the next, with optional message passing.

In addition to being able to run a single generator to completion, the `ASQ#runner(...)` we discussed in Appendix A is a similar implementation of the concepts of `runAll(...)`, which can run multiple generators concurrently to completion.

So let's see how we can implement the concurrent Ajax scenario from Chapter 4:

```

ASQ(
  "http://some.url.2"
)
.runner(
  function*(token){
    // transfer control
    yield token;

    var url1 = token.messages[0]; // "http://some.url.1"

    // clear out messages to start fresh
    token.messages = [];

    var p1 = request( url1 );

    // transfer control
    yield token;

    token.messages.push( yield p1 );
  },
  function*(token){
    var url2 = token.messages[0]; // "http://some.url.2"

    // message pass and transfer control
    token.messages[0] = "http://some.url.1";
    yield token;

    var p2 = request( url2 );

    // transfer control
    yield token;

    token.messages.push( yield p2 );

    // pass along results to next sequence step
    return token.messages;
  }
)
.val( function(res){
  // `res[0]` comes from "http://some.url.1"
  // `res[1]` comes from "http://some.url.2"
} );

```

The main differences between `ASQ#runner(..)` and `runAll(..)` are as follows:

- Each generator (coroutine) is provided an argument we call `token`, which is the special value to `yield` when you want to explicitly transfer control to the next coroutine.
- `token.messages` is an array that holds any messages passed in from the previous sequence step. It's also a data structure that you can use to share messages between coroutines.

- `yield` ing a Promise (or sequence) value does not transfer control, but instead pauses the coroutine processing until that value is ready.
- The last `return` ed or `yield` ed value from the coroutine processing run will be forward passed to the next step in the sequence.

It's also easy to layer helpers on top of the base `ASQ#runner(...)` functionality to suit different uses.

State Machines

One example that may be familiar to many programmers is state machines. You can, with the help of a simple cosmetic utility, create an easy-to-express state machine processor.

Let's imagine such a utility. We'll call it `state(...)`, and will pass it two arguments: a state value and a generator that handles that state. `state(...)` will do the dirty work of creating and returning an adapter generator to pass to `ASQ#runner(...)`.

Consider:

```
function state(val, handler) {
  // make a coroutine handler for this state
  return function*(token) {
    // state transition handler
    function transition(to) {
      token.messages[0] = to;
    }

    // set initial state (if none set yet)
    if (token.messages.length < 1) {
      token.messages[0] = val;
    }

    // keep going until final state (false) is reached
    while (token.messages[0] !== false) {
      // current state matches this handler?
      if (token.messages[0] === val) {
        // delegate to state handler
        yield *handler(transition);
      }

      // transfer control to another state handler?
      if (token.messages[0] !== false) {
        yield token;
      }
    }
  };
}
```

If you look closely, you'll see that `state(..)` returns back a generator that accepts a `token`, and then it sets up a `while` loop that will run until the state machine reaches its final state (which we arbitrarily pick as the `false` value); that's exactly the kind of generator we want to pass to `ASQ#runner(..)`!

We also arbitrarily reserve the `token.messages[0]` slot as the place where the current state of our state machine will be tracked, which means we can even seed the initial state as the value passed in from the previous step in the sequence.

How do we use the `state(..)` helper along with `ASQ#runner(..)`?

```

var prevState;

ASQ(
  /* optional: initial state value */
  2
)
// run our state machine
// transitions: 2 -> 3 -> 1 -> 3 -> false
.runner(
  // state `1` handler
  state( 1, function *stateOne(transition){
    console.log( "in state 1" );

    prevState = 1;
    yield transition( 3 );      // goto state `3`
  } ),

  // state `2` handler
  state( 2, function *stateTwo(transition){
    console.log( "in state 2" );

    prevState = 2;
    yield transition( 3 );      // goto state `3`
  } ),

  // state `3` handler
  state( 3, function *stateThree(transition){
    console.log( "in state 3" );

    if (prevState === 2) {
      prevState = 3;
      yield transition( 1 ); // goto state `1`
    }
    // all done!
    else {
      yield "That's all folks!";

      prevState = 3;
      yield transition( false ); // terminal state
    }
  } )
)
// state machine complete, so move on
.val( function(msg){
  console.log( msg );      // That's all folks!
} );

```

It's important to note that the `*stateOne(..)`, `*stateTwo(..)`, and `*stateThree(..)` generators themselves are reinvoked each time that state is entered, and they finish when you `transition(..)` to another value. While not shown here, of course these state

generator handlers can be asynchronously paused by `yield`ing Promises/sequences/thunks.

The underneath hidden generators produced by the `state(...)` helper and actually passed to `ASQ#runner(...)` are the ones that continue to run concurrently for the length of the state machine, and each of them handles cooperatively `yield`ing control to the next, and so on.

Note: See this "ping pong" example (<http://jsbin.com/qutabu/1/edit?js,output>) for more illustration of using cooperative concurrency with generators driven by `ASQ#runner(...)`.

Communicating Sequential Processes (CSP)

"Communicating Sequential Processes" (CSP) was first described by C. A. R. Hoare in a 1978 academic paper (<http://dl.acm.org/citation.cfm?doid=359576.359585>), and later in a 1985 book (<http://www.usingcsp.com/>) of the same name. CSP describes a formal method for concurrent "processes" to interact (aka "communicate") during processing.

You may recall that we examined concurrent "processes" back in Chapter 1, so our exploration of CSP here will build upon that understanding.

Like most great concepts in computer science, CSP is heavily steeped in academic formalism, expressed as a process algebra. However, I suspect symbolic algebra theorems won't make much practical difference to the reader, so we will want to find some other way of wrapping our brains around CSP.

I will leave much of the formal description and proof of CSP to Hoare's writing, and to many other fantastic writings since. Instead, we will try to just briefly explain the idea of CSP in as un-academic and hopefully intuitively understandable a way as possible.

Message Passing

The core principle in CSP is that all communication/interaction between otherwise independent processes must be through formal message passing. Perhaps counter to your expectations, CSP message passing is described as a synchronous action, where the sender process and the receiver process have to mutually be ready for the message to be passed.

How could such synchronous messaging possibly be related to asynchronous programming in JavaScript?

The concreteness of relationship comes from the nature of how ES6 generators are used to produce synchronous-looking actions that under the covers can indeed either be synchronous or (more likely) asynchronous.

In other words, two or more concurrently running generators can appear to synchronously message each other while preserving the fundamental asynchrony of the system because each generator's code is paused (aka "blocked") waiting on resumption of an asynchronous action.

How does this work?

Imagine a generator (aka "process") called "A" that wants to send a message to generator "B." First, "A" `yield s` the message (thus pausing "A") to be sent to "B." When "B" is ready and takes the message, "A" is then resumed (unblocked).

Symmetrically, imagine a generator "A" that wants a message **from** "B." "A" `yield s` its request (thus pausing "A") for the message from "B," and once "B" sends a message, "A" takes the message and is resumed.

One of the more popular expressions of this CSP message passing theory comes from ClojureScript's `core.async` library, and also from the `go` language. These take on CSP embody the described communication semantics in a conduit that is opened between processes called a "channel."

Note: The term *channel* is used in part because there are modes in which more than one value can be sent at once into the "buffer" of the channel; this is similar to what you may think of as a stream. We won't go into depth about it here, but it can be a very powerful technique for managing streams of data.

In the simplest notion of CSP, a channel that we create between "A" and "B" would have a method called `take(...)` for blocking to receive a value, and a method called `put(...)` for blocking to send a value.

This might look like:

```

var ch = channel();

function *foo() {
    var msg = yield take( ch );

    console.log( msg );
}

function *bar() {
    yield put( ch, "Hello World" );

    console.log( "message sent" );
}

run( foo );
run( bar );
// Hello World
// "message sent"

```

Compare this structured, synchronous(-looking) message passing interaction to the informal and unstructured message sharing that `ASQ#runner(..)` provides through the `token.messages` array and cooperative `yield`ing. In essence, `yield put(..)` is a single operation that both sends the value and pauses execution to transfer control, whereas in earlier examples we did those as separate steps.

Moreover, CSP stresses that you don't really explicitly "transfer control," but rather you design your concurrent routines to block expecting either a value received from the channel, or to block expecting to try to send a message on the channel. The blocking around receiving or sending messages is how you coordinate sequencing of behavior between the coroutines.

Note: Fair warning: this pattern is very powerful but it's also a little mind twisting to get used to at first. You will want to practice this a bit to get used to this new way of thinking about coordinating your concurrency.

There are several great libraries that have implemented this flavor of CSP in JavaScript, most notably "js-csp" (<https://github.com/ubolonton/js-csp>), which James Long (<http://twitter.com/jlongster>) forked (<https://github.com/jlongster/js-csp>) and has written extensively about (<http://jlongster.com/Taming-the-Asynchronous-Beast-with-CSP-in-JavaScript>). Also, it cannot be stressed enough how amazing the many writings of David Nolen (<http://twitter.com/swannodette>) are on the topic of adapting ClojureScript's go-style core.async CSP into JS generators (<http://swannodette.github.io/2013/08/24/es6-generators-and-csp>).

asynquence CSP emulation

Because we've been discussing async patterns here in the context of my `asynquence` library, you might be interested to see that we can fairly easily add an emulation layer on top of `ASQ#runner(...)` generator handling as a nearly perfect porting of the CSP API and behavior. This emulation layer ships as an optional part of the "asynquence-contrib" package alongside `asynquence`.

Very similar to the `state(...)` helper from earlier, `ASQ.csp.go(...)` takes a generator -- in go/core.async terms, it's known as a goroutine -- and adapts it to use with `ASQ#runner(...)` by returning a new generator.

Instead of being passed a `token`, your goroutine receives an initially created channel (`ch` below) that all goroutines in this run will share. You can create more channels (which is often quite helpful!) with `ASQ.csp.chan(...)`.

In CSP, we model all asynchrony in terms of blocking on channel messages, rather than blocking waiting for a Promise/sequence/thunk to complete.

So, instead of `yield`ing the Promise returned from `request(..)`, `request(..)` should return a channel that you `take(..)` a value from. In other words, a single-value channel is roughly equivalent in this context/usage to a Promise/sequence.

Let's first make a channel-aware version of `request(..)`:

```
function request(url) {
  var ch = ASQ.csp.channel();
  ajax( url ).then( function(content){
    // `putAsync(..)` is a version of `put(..)` that
    // can be used outside of a generator. It returns
    // a promise for the operation's completion. We
    // don't use that promise here, but we could if
    // we needed to be notified when the value had
    // been `take(..)`n.
    ASQ.csp.putAsync( ch, content );
  } );
  return ch;
}
```

From Chapter 3, "promisory" is a Promise-producing utility, "thunkory" from Chapter 4 is a thunk-producing utility, and finally, in Appendix A we invented "sequory" for a sequence-producing utility.

Naturally, we need to coin a symmetric term here for a channel-producing utility. So let's unsurprisingly call it a "chanory" ("channel" + "factory"). As an exercise for the reader, try your hand at defining a `channelify(..)` utility similar to `Promise.wrap(..)` / `promisify(..)` (Chapter 3), `thunkify(..)` (Chapter 4), and `ASQ.wrap(..)` (Appendix A).

Now consider the concurrent Ajax example using `asyquence`-flavored CSP:

```
ASQ()
.runner(
  ASQ.csp.go( function*(ch){
    yield ASQ.csp.put( ch, "http://some.url.2" );

    var url1 = yield ASQ.csp.take( ch );
    // "http://some.url.1"

    var res1 = yield ASQ.csp.take( request( url1 ) );

    yield ASQ.csp.put( ch, res1 );
  } ),
  ASQ.csp.go( function*(ch){
    var url2 = yield ASQ.csp.take( ch );
    // "http://some.url.2"

    yield ASQ.csp.put( ch, "http://some.url.1" );

    var res2 = yield ASQ.csp.take( request( url2 ) );
    var res1 = yield ASQ.csp.take( ch );

    // pass along results to next sequence step
    ch.buffer_size = 2;
    ASQ.csp.put( ch, res1 );
    ASQ.csp.put( ch, res2 );
  } )
)
.val( function(res1,res2){
  // `res1` comes from "http://some.url.1"
  // `res2` comes from "http://some.url.2"
} );
```

The message passing that trades the URL strings between the two goroutines is pretty straightforward. The first goroutine makes an Ajax request to the first URL, and that response is put onto the `ch` channel. The second goroutine makes an Ajax request to the second URL, then gets the first response `res1` off the `ch` channel. At that point, both responses `res1` and `res2` are completed and ready.

If there are any remaining values in the `ch` channel at the end of the goroutine run, they will be passed along to the next step in the sequence. So, to pass out message(s) from the final goroutine, `put(...)` them into `ch`. As shown, to avoid the blocking of those final `put(..)`s, we switch `ch` into buffering mode by setting its `buffer_size` to `2` (default: `0`).

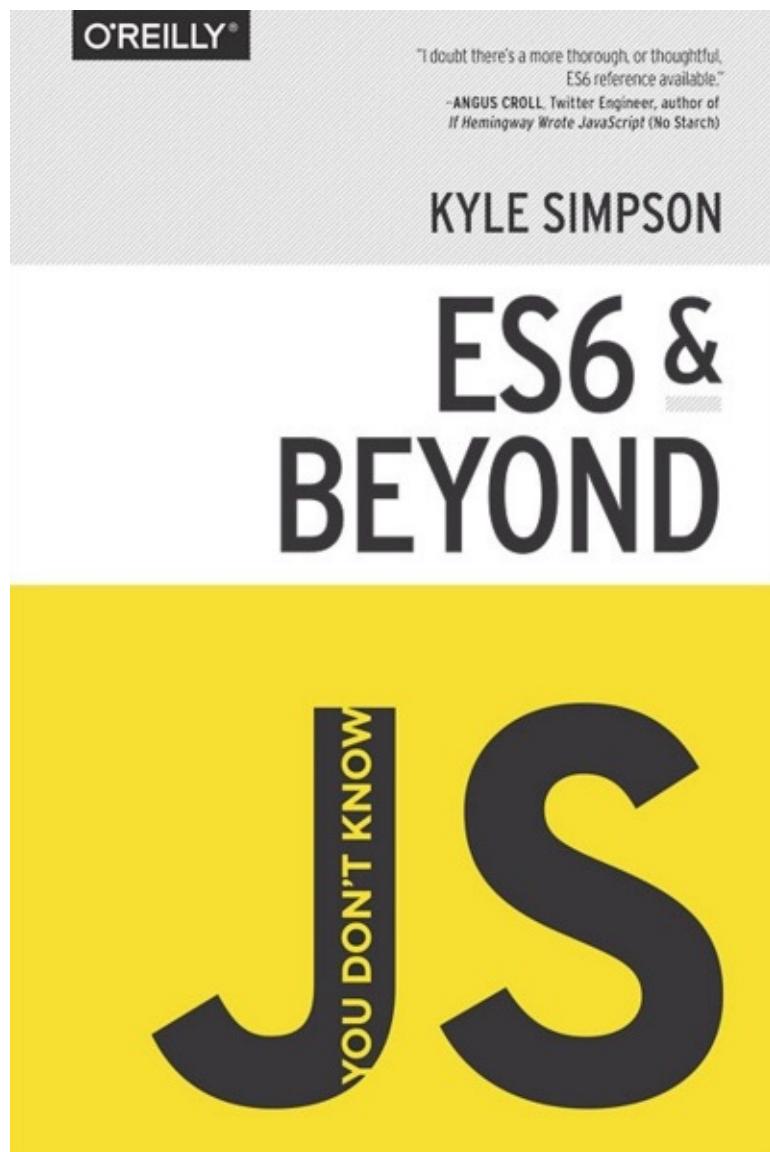
Note: See many more examples of using `asyquence`-flavored CSP here (<https://gist.github.com/getify/e0d04f1f5aa24b1947ae>).

Review

Promises and generators provide the foundational building blocks upon which we can build much more sophisticated and capable asynchrony.

`asynquence` has utilities for implementing *iterable sequences*, *reactive sequences* (aka "Observables"), *concurrent coroutines*, and even *CSP goroutines*.

Those patterns, combined with the continuation-callback and Promise capabilities, gives `asynquence` a powerful mix of different asynchronous functionalities, all integrated in one clean async flow control abstraction: the sequence.



Chapter 1: ES? Now & Future

Before you dive into this book, you should have a solid working proficiency over JavaScript up to the most recent standard (at the time of this writing), which is commonly called *ES5* (technically ES 5.1). Here, we plan to talk squarely about the upcoming *ES6*, as well as cast our vision beyond to understand how JS will evolve moving forward.

If you are still looking for confidence with JavaScript, I highly recommend you read the other titles in this series first:

- *Up & Going*: Are you new to programming and JS? This is the roadmap you need to consult as you start your learning journey.
- *Scope & Closures*: Did you know that JS lexical scope is based on compiler (not interpreter!) semantics? Can you explain how closures are a direct result of lexical scope and functions as values?
- *this & Object Prototypes*: Can you recite the four simple rules for how `this` is bound? Have you been muddling through fake "classes" in JS instead of adopting the simpler "behavior delegation" design pattern? Ever heard of *objects linked to other objects* (OLOO)?
- *Types & Grammar*: Do you know the built-in types in JS, and more importantly, do you know how to properly and safely use coercion between types? How comfortable are you with the nuances of JS grammar/syntax?
- *Async & Performance*: Are you still using callbacks to manage your asynchrony? Can you explain what a promise is and why/how it solves "callback hell"? Do you know how to use generators to improve the legibility of async code? What exactly constitutes mature optimization of JS programs and individual operations?

If you've already read all those titles and you feel pretty comfortable with the topics they cover, it's time we dive into the evolution of JS to explore all the changes coming not only soon but farther over the horizon.

Unlike ES5, ES6 is not just a modest set of new APIs added to the language. It incorporates a whole slew of new syntactic forms, some of which may take quite a bit of getting used to. There's also a variety of new organization forms and new API helpers for various data types.

ES6 is a radical jump forward for the language. Even if you think you know JS in ES5, ES6 is full of new stuff you *don't know yet*, so get ready! This book explores all the major themes of ES6 that you need to get up to speed on, and even gives you a glimpse of future features coming down the track that you should be aware of.

Warning: All code in this book assumes an ES6+ environment. At the time of this writing, ES6 support varies quite a bit in browsers and JS environments (like Node.js), so your mileage may vary.

Versioning

The JavaScript standard is referred to officially as "ECMAScript" (abbreviated "ES"), and up until just recently has been versioned entirely by ordinal number (i.e., "5" for "5th edition").

The earliest versions, ES1 and ES2, were not widely known or implemented. ES3 was the first widespread baseline for JavaScript, and constitutes the JavaScript standard for browsers like IE6-8 and older Android 2.x mobile browsers. For political reasons beyond what we'll cover here, the ill-fated ES4 never came about.

In 2009, ES5 was officially finalized (later ES5.1 in 2011), and settled as the widespread standard for JS for the modern revolution and explosion of browsers, such as Firefox, Chrome, Opera, Safari, and many others.

Leading up to the expected *next* version of JS (slipped from 2013 to 2014 and then 2015), the obvious and common label in discourse has been ES6.

However, late into the ES6 specification timeline, suggestions have surfaced that versioning may in the future switch to a year-based schema, such as ES2016 (aka ES7) to refer to whatever version of the specification is finalized before the end of 2016. Some disagree, but ES6 will likely maintain its dominant mindshare over the late-change substitute ES2015. However, ES2016 may in fact signal the new year-based schema.

It has also been observed that the pace of JS evolution is much faster even than single-year versioning. As soon as an idea begins to progress through standards discussions, browsers start prototyping the feature, and early adopters start experimenting with the code.

Usually well before there's an official stamp of approval, a feature is de facto standardized by virtue of this early engine/tooling prototyping. So it's also valid to consider the future of JS versioning to be per-feature rather than per-arbitrary-collection-of-major-features (as it is now) or even per-year (as it may become).

The takeaway is that the version labels stop being as important, and JavaScript starts to be seen more as an evergreen, living standard. The best way to cope with this is to stop thinking about your code base as being "ES6-based," for instance, and instead consider it feature by feature for support.

Transpiling

Made even worse by the rapid evolution of features, a problem arises for JS developers who at once may both strongly desire to use new features while at the same time being slapped with the reality that their sites/apps may need to support older browsers without such support.

The way ES5 appears to have played out in the broader industry, the typical mindset was that code bases waited to adopt ES5 until most if not all pre-ES5 environments had fallen out of their support spectrum. As a result, many are just recently (at the time of this writing) starting to adopt things like `strict` mode, which landed in ES5 over five years ago.

It's widely considered to be a harmful approach for the future of the JS ecosystem to wait around and trail the specification by so many years. All those responsible for evolving the language desire for developers to begin basing their code on the new features and patterns as soon as they stabilize in specification form and browsers have a chance to implement them.

So how do we resolve this seeming contradiction? The answer is tooling, specifically a technique called *transpiling* (transformation + compiling). Roughly, the idea is to use a special tool to transform your ES6 code into equivalent (or close!) matches that work in ES5 environments.

For example, consider shorthand property definitions (see "Object Literal Extensions" in Chapter 2). Here's the ES6 form:

```
var foo = [1,2,3];

var obj = {
  foo      // means `foo: foo`
};

obj.foo;    // [1,2,3]
```

But (roughly) here's how that transpiles:

```
var foo = [1,2,3];

var obj = {
  foo: foo
};

obj.foo;    // [1,2,3]
```

This is a minor but pleasant transformation that lets us shorten the `foo: foo` in an object literal declaration to just `foo`, if the names are the same.

Transpilers perform these transformations for you, usually in a build workflow step similar to how you perform linting, minification, and other similar operations.

Shims/Polyfills

Not all new ES6 features need a transpiler. Polyfills (aka shims) are a pattern for defining equivalent behavior from a newer environment into an older environment, when possible. Syntax cannot be polyfilled, but APIs often can be.

For example, `Object.is(...)` is a new utility for checking strict equality of two values but without the nuanced exceptions that `==` has for `NaN` and `-0` values. The polyfill for `Object.is(...)` is pretty easy:

```
if (!Object.is) {
  Object.is = function(v1, v2) {
    // test for `~-0`
    if (v1 === 0 && v2 === 0) {
      return 1 / v1 === 1 / v2;
    }
    // test for `NaN`
    if (v1 !== v1) {
      return v2 !== v2;
    }
    // everything else
    return v1 === v2;
  };
}
```

Tip: Pay attention to the outer `if` statement guard wrapped around the polyfill. This is an important detail, which means the snippet only defines its fallback behavior for older environments where the API in question isn't already defined; it would be very rare that you'd want to overwrite an existing API.

There's a great collection of ES6 shims called "ES6 Shim" (<https://github.com/paulmillr/es6-shim/>) that you should definitely adopt as a standard part of any new JS project!

It is assumed that JS will continue to evolve constantly, with browsers rolling out support for features continually rather than in large chunks. So the best strategy for keeping updated as it evolves is to just introduce polyfill shims into your code base, and a transpiler step into your build workflow, right now and get used to that new reality.

If you decide to keep the status quo and just wait around for all browsers without a feature supported to go away before you start using the feature, you're always going to be way behind. You'll sadly be missing out on all the innovations designed to make writing JavaScript more effective, efficient, and robust.

Review

ES6 (some may try to call it ES2015) is just landing as of the time of this writing, and it has lots of new stuff you need to learn!

But it's even more important to shift your mindset to align with the new way that JavaScript is going to evolve. It's not just waiting around for years for some official document to get a vote of approval, as many have done in the past.

Now, JavaScript features land in browsers as they become ready, and it's up to you whether you'll get on the train early or whether you'll be playing costly catch-up games years from now.

Whatever labels that future JavaScript adopts, it's going to move a lot quicker than it ever has before. Transpilers and shims/polyfills are important tools to keep you on the forefront of where the language is headed.

If there's any narrative important to understand about the new reality for JavaScript, it's that all JS developers are strongly implored to move from the trailing edge of the curve to the leading edge. And learning ES6 is where that all starts!

Chapter 2: Syntax

If you've been writing JS for any length of time, odds are the syntax is pretty familiar to you. There are certainly many quirks, but overall it's a fairly reasonable and straightforward syntax that draws many similarities from other languages.

However, ES6 adds quite a few new syntactic forms that take some getting used to. In this chapter, we'll tour through them to find out what's in store.

Tip: At the time of this writing, some of the features discussed in this book have been implemented in various browsers (Firefox, Chrome, etc.), but some have only been partially implemented and many others have not been implemented at all. Your experience may be mixed trying these examples directly. If so, try them out with transpilers, as most of these features are covered by those tools. ES6Fiddle (<http://www.es6fiddle.net/>) is a great, easy-to-use playground for trying out ES6, as is the online REPL for the Babel transpiler (<http://babeljs.io/repl/>).

Block-Scope Declarations

You're probably aware that the fundamental unit of variable scoping in JavaScript has always been the `function`. If you needed to create a block of scope, the most prevalent way to do so other than a regular function declaration was the immediately invoked function expression (IIFE). For example:

```
var a = 2;

(function IIFE(){
  var a = 3;
  console.log( a );    // 3
})();

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

let Declarations

However, we can now create declarations that are bound to any block, called (unsurprisingly) *block scoping*. This means all we need is a pair of `{ ... }` to create a scope. Instead of using `var`, which always declares variables attached to the enclosing function (or global, if top level) scope, use `let`:

```
var a = 2;

{
  let a = 3;
  console.log( a );    // 3
}

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

It's not very common or idiomatic thus far in JS to use a standalone `{ ... }` block, but it's always been valid. And developers from other languages that have *block scoping* will readily recognize that pattern.

I believe this is the best way to create block-scoped variables, with a dedicated `{ ... }` block. Moreover, you should always put the `let` declaration(s) at the very top of that block. If you have more than one to declare, I'd recommend using just one `let`.

Stylistically, I even prefer to put the `let` on the same line as the opening `{`, to make it clearer that this block is only for the purpose of declaring the scope for those variables.

```
{   let a = 2, b, c;
    // ...
}
```

Now, that's going to look strange and it's not likely going to match the recommendations given in most other ES6 literature. But I have reasons for my madness.

There's another experimental (not standardized) form of the `let` declaration called the `let -block`, which looks like:

```
let (a = 2, b, c) {
  // ...
}
```

That form is what I call *explicit* block scoping, whereas the `let ..` declaration form that mirrors `var` is more *implicit*, as it kind of hijacks whatever `{ ... }` pair it's found in. Generally developers find *explicit* mechanisms a bit more preferable than *implicit* mechanisms, and I claim this is one of those cases.

If you compare the previous two snippet forms, they're very similar, and in my opinion both qualify stylistically as *explicit* block scoping. Unfortunately, the `let (...) { .. }` form, the most *explicit* of the options, was not adopted in ES6. That may be revisited post-ES6, but for now the former option is our best bet, I think.

To reinforce the *implicit* nature of `let ..` declarations, consider these usages:

```
let a = 2;

if (a > 1) {
    let b = a * 3;
    console.log( b );           // 6

    for (let i = a; i <= b; i++) {
        let j = i + 10;
        console.log( j );
    }
    // 12 13 14 15 16

    let c = a + b;
    console.log( c );           // 8
}
```

Quick quiz without looking back at that snippet: which variable(s) exist only inside the `if` statement, and which variable(s) exist only inside the `for` loop?

The answers: the `if` statement contains `b` and `c` block-scoped variables, and the `for` loop contains `i` and `j` block-scoped variables.

Did you have to think about it for a moment? Does it surprise you that `i` isn't added to the enclosing `if` statement scope? That mental pause and questioning -- I call it a "mental tax" -- comes from the fact that this `let` mechanism is not only new to us, but it's also *implicit*.

There's also hazard in the `let c = ..` declaration appearing so far down in the scope. Unlike traditional `var`-declared variables, which are attached to the entire enclosing function scope regardless of where they appear, `let` declarations attach to the block scope but are not initialized until they appear in the block.

Accessing a `let`-declared variable earlier than its `let ..` declaration/initialization causes an error, whereas with `var` declarations the ordering doesn't matter (except stylistically).

Consider:

```
{
    console.log( a );      // undefined
    console.log( b );      // ReferenceError!

    var a;
    let b;
}
```

Warning: This `ReferenceError` from accessing too-early `let`-declared references is technically called a *Temporal Dead Zone (TDZ)* error -- you're accessing a variable that's been declared but not yet initialized. This will not be the only time we see TDZ errors -- they crop up in several places in ES6. Also, note that "initialized" doesn't require explicitly assigning a value in your code, as `let b;` is totally valid. A variable that's not given an assignment at declaration time is assumed to have been assigned the `undefined` value, so `let b;` is the same as `let b = undefined;`. Explicit assignment or not, you cannot access `b` until the `let b` statement is run.

One last gotcha: `typeof` behaves differently with TDZ variables than it does with undeclared (or declared!) variables. For example:

```
{
  // `a` is not declared
  if (typeof a === "undefined") {
    console.log( "cool" );
  }

  // `b` is declared, but in its TDZ
  if (typeof b === "undefined") {           // ReferenceError!
    ...
  }

  ...
}

let b;
}
```

The `a` is not declared, so `typeof` is the only safe way to check for its existence or not. But `typeof b` throws the TDZ error because farther down in the code there happens to be a `let b` declaration. Oops.

Now it should be clearer why I insist that `let` declarations should all be at the top of their scope. That totally avoids the accidental errors of accessing too early. It also makes it more *explicit* when you look at the start of a block, any block, what variables it contains.

Your blocks (`if` statements, `while` loops, etc.) don't have to share their original behavior with scoping behavior.

This explicitness on your part, which is up to you to maintain with discipline, will save you lots of refactor headaches and footguns down the line.

Note: For more information on `let` and block scoping, see Chapter 3 of the *Scope & Closures* title of this series.

let + for

The only exception I'd make to the preference for the *explicit* form of `let` declaration blocking is a `let` that appears in the header of a `for` loop. The reason may seem nuanced, but I believe it to be one of the more important ES6 features.

Consider:

```
var funcs = [];

for (let i = 0; i < 5; i++) {
  funcs.push( function(){
    console.log( i );
  });
}

funcs[3]();           // 3
```

The `let i` in the `for` header declares an `i` not just for the `for` loop itself, but it redeclares a new `i` for each iteration of the loop. That means that closures created inside the loop iteration close over those per-iteration variables the way you'd expect.

If you tried that same snippet but with `var i` in the `for` loop header, you'd get `5` instead of `3`, because there'd only be one `i` in the outer scope that was closed over, instead of a new `i` for each iteration's function to close over.

You could also have accomplished the same thing slightly more verbosely:

```
var funcs = [];

for (var i = 0; i < 5; i++) {
  let j = i;
  funcs.push( function(){
    console.log( j );
  });
}

funcs[3]();           // 3
```

Here, we forcibly create a new `j` for each iteration, and then the closure works the same way. I prefer the former approach; that extra special capability is why I endorse the `for (let ...) ...` form. It could be argued it's somewhat more *implicit*, but it's *explicit* enough, and useful enough, for my tastes.

`let` also works the same way with `for..in` and `for..of` loops (see "for..of Loops").

const Declarations

There's one other form of block-scoped declaration to consider: the `const`, which creates *constants*.

What exactly is a constant? It's a variable that's read-only after its initial value is set.

Consider:

```
{
  const a = 2;
  console.log( a );    // 2

  a = 3;              // TypeError!
}
```

You are not allowed to change the value the variable holds once it's been set, at declaration time. A `const` declaration must have an explicit initialization. If you wanted a *constant* with the `undefined` value, you'd have to declare `const a = undefined` to get it.

Constants are not a restriction on the value itself, but on the variable's assignment of that value. In other words, the value is not frozen or immutable because of `const`, just the assignment of it. If the value is complex, such as an object or array, the contents of the value can still be modified:

```
{
  const a = [1,2,3];
  a.push( 4 );
  console.log( a );    // [1,2,3,4]

  a = 42;              // TypeError!
}
```

The `a` variable doesn't actually hold a constant array; rather, it holds a constant reference to the array. The array itself is freely mutable.

Warning: Assigning an object or array as a constant means that value will not be able to be garbage collected until that constant's lexical scope goes away, as the reference to the value can never be unset. That may be desirable, but be careful if it's not your intent!

Essentially, `const` declarations enforce what we've stylistically signaled with our code for years, where we declared a variable name of all uppercase letters and assigned it some literal value that we took care never to change. There's no enforcement on a `var` assignment, but there is now with a `const` assignment, which can help you catch unintended changes.

`const` can be used with variable declarations of `for`, `for..in`, and `for..of` loops (see "for..of Loops"). However, an error will be thrown if there's any attempt to reassign, such as the typical `i++` clause of a `for` loop.

const Or Not

There's some rumored assumptions that a `const` could be more optimizable by the JS engine in certain scenarios than a `let` or `var` would be. Theoretically, the engine more easily knows the variable's value/type will never change, so it can eliminate some possible tracking.

Whether `const` really helps here or this is just our own fantasies and intuitions, the much more important decision to make is if you intend constant behavior or not. Remember: one of the most important roles for source code is to communicate clearly, not only to you, but your future self and other code collaborators, what your intent is.

Some developers prefer to start out every variable declaration as a `const` and then relax a declaration back to a `let` if it becomes necessary for its value to change in the code. This is an interesting perspective, but it's not clear that it genuinely improves the readability or reason-ability of code.

It's not really a *protection*, as many believe, because any later developer who wants to change a value of a `const` can just blindly change `const` to `let` on the declaration. At best, it protects accidental change. But again, other than our intuitions and sensibilities, there doesn't appear to be objective and clear measure of what constitutes "accidents" or prevention thereof. Similar mindsets exist around type enforcement.

My advice: to avoid potentially confusing code, only use `const` for variables that you're intentionally and obviously signaling will not change. In other words, don't *rely on* `const` for code behavior, but instead use it as a tool for signaling intent, when intent can be signaled clearly.

Block-scoped Functions

Starting with ES6, function declarations that occur inside of blocks are now specified to be scoped to that block. Prior to ES6, the specification did not call for this, but many implementations did it anyway. So now the specification meets reality.

Consider:

```
{
  foo();           // works!

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

foo();           // ReferenceError
```

The `foo()` function is declared inside the `{ ... }` block, and as of ES6 is block-scoped there. So it's not available outside that block. But also note that it is "hoisted" within the block, as opposed to `let` declarations, which suffer the TDZ error trap mentioned earlier.

Block-scoping of function declarations could be a problem if you've ever written code like this before, and relied on the old legacy non-block-scoped behavior:

```
if (something) {
  function foo() {
    console.log( "1" );
  }
}
else {
  function foo() {
    console.log( "2" );
  }
}

foo();           // ??
```

In pre-ES6 environments, `foo()` would print `"2"` regardless of the value of `something`, because both function declarations were hoisted out of the blocks, and the second one always wins.

In ES6, that last line throws a `ReferenceError`.

Spread/Rest

ES6 introduces a new `...` operator that's typically referred to as the *spread* or *rest* operator, depending on where/how it's used. Let's take a look:

```
function foo(x,y,z) {
    console.log( x, y, z );
}

foo( ...[1,2,3] );           // 1 2 3
```

When `...` is used in front of an array (actually, any *iterable*, which we cover in Chapter 3), it acts to "spread" it out into its individual values.

You'll typically see that usage as is shown in that previous snippet, when spreading out an array as a set of arguments to a function call. In this usage, `...` acts to give us a simpler syntactic replacement for the `apply(..)` method, which we would typically have used pre-ES6 as:

```
foo.apply( null, [1,2,3] );           // 1 2 3
```

But `...` can be used to spread out/expand a value in other contexts as well, such as inside another array declaration:

```
var a = [2,3,4];
var b = [ 1, ...a, 5 ];

console.log( b );                  // [1,2,3,4,5]
```

In this usage, `...` is basically replacing `concat(..)`, as it behaves like `[1].concat(a, [5])` here.

The other common usage of `...` can be seen as essentially the opposite; instead of spreading a value out, the `...` gathers a set of values together into an array. Consider:

```
function foo(x, y, ...z) {
    console.log( x, y, z );
}

foo( 1, 2, 3, 4, 5 );           // 1 2 [3,4,5]
```

The `...z` in this snippet is essentially saying: "gather the *rest* of the arguments (if any) into an array called `z`." Because `x` was assigned `1`, and `y` was assigned `2`, the rest of the arguments `3`, `4`, and `5` were gathered into `z`.

Of course, if you don't have any named parameters, the `...` gathers all arguments:

```
function foo( ...args ) {  
    console.log( args );  
}  
  
foo( 1, 2, 3, 4, 5);           // [1,2,3,4,5]
```

Note: The `...args` in the `foo(..)` function declaration is usually called "rest parameters," because you're collecting the rest of the parameters. I prefer "gather," because it's more descriptive of what it does rather than what it contains.

The best part about this usage is that it provides a very solid alternative to using the long-since-deprecated `arguments` array -- actually, it's not really an array, but an array-like object. Because `args` (or whatever you call it -- a lot of people prefer `r` or `rest`) is a real array, we can get rid of lots of silly pre-ES6 tricks we jumped through to make `arguments` into something we can treat as an array.

Consider:

```
// doing things the new ES6 way
function foo(...args) {
    // `args` is already a real array

    // discard first element in `args`
    args.shift();

    // pass along all of `args` as arguments
    // to `console.log(..)`
    console.log( ...args );
}

// doing things the old-school pre-ES6 way
function bar() {
    // turn `arguments` into a real array
    var args = Array.prototype.slice.call( arguments );

    // add some elements on the end
    args.push( 4, 5 );

    // filter out odd numbers
    args = args.filter( function(v){
        return v % 2 == 0;
    } );

    // pass along all of `args` as arguments
    // to `foo(..)`
    foo.apply( null, args );
}

bar( 0, 1, 2, 3 );                                // 2 4
```

The `...args` in the `foo(..)` function declaration gathers arguments, and the `...args` in the `console.log(..)` call spreads them out. That's a good illustration of the symmetric but opposite uses of the `...` operator.

Besides the `...` usage in a function declaration, there's another case where `...` is used for gathering values, and we'll look at it in the "Too Many, Too Few, Just Enough" section later in this chapter.

Default Parameter Values

Perhaps one of the most common idioms in JavaScript relates to setting a default value for a function parameter. The way we've done this for years should look quite familiar:

```

function foo(x,y) {
  x = x || 11;
  y = y || 31;

  console.log( x + y );
}

foo();           // 42
foo( 5, 6 );    // 11
foo( 5 );       // 36
foo( null, 6 ); // 17

```

Of course, if you've used this pattern before, you know that it's both helpful and a little bit dangerous, if for example you need to be able to pass in what would otherwise be considered a falsy value for one of the parameters. Consider:

```
foo( 0, 42 );      // 53 <-- Oops, not 42
```

Why? Because the `0` is falsy, and so the `x || 11` results in `11`, not the directly passed in `0`.

To fix this gotcha, some people will instead write the check more verbosely like this:

```

function foo(x,y) {
  x = (x !== undefined) ? x : 11;
  y = (y !== undefined) ? y : 31;

  console.log( x + y );
}

foo( 0, 42 );      // 42
foo( undefined, 6 ); // 17

```

Of course, that means that any value except `undefined` can be directly passed in. However, `undefined` will be assumed to signal, "I didn't pass this in." That works great unless you actually need to be able to pass `undefined` in.

In that case, you could test to see if the argument is actually omitted, by it actually not being present in the `arguments` array, perhaps like this:

```

function foo(x,y) {
  x = (0 in arguments) ? x : 11;
  y = (1 in arguments) ? y : 31;

  console.log( x + y );
}

foo( 5 );           // 36
foo( 5, undefined ); // NaN

```

But how would you omit the first `x` argument without the ability to pass in any kind of value (not even `undefined`) that signals "I'm omitting this argument"?

`foo(,5)` is tempting, but it's invalid syntax. `foo.apply(null,[,5])` seems like it should do the trick, but `apply(..)`'s quirks here mean that the arguments are treated as `[undefined,5]`, which of course doesn't omit.

If you investigate further, you'll find you can only omit arguments on the end (i.e., righthand side) by simply passing fewer arguments than "expected," but you cannot omit arguments in the middle or at the beginning of the arguments list. It's just not possible.

There's a principle applied to JavaScript's design here that is important to remember: `undefined` means *missing*. That is, there's no difference between `undefined` and *missing*, at least as far as function arguments go.

Note: There are, confusingly, other places in JS where this particular design principle doesn't apply, such as for arrays with empty slots. See the *Types & Grammar* title of this series for more information.

With all this in mind, we can now examine a nice helpful syntax added as of ES6 to streamline the assignment of default values to missing arguments:

```

function foo(x = 11, y = 31) {
  console.log( x + y );
}

foo();           // 42
foo( 5, 6 );     // 11
foo( 0, 42 );    // 42

foo( 5 );        // 36
foo( 5, undefined ); // 36 <-- `undefined` is missing
foo( 5, null );   // 5 <-- null coerces to `0`

foo( undefined, 6 ); // 17 <-- `undefined` is missing
foo( null, 6 );    // 6 <-- null coerces to `0`

```

Notice the results and how they imply both subtle differences and similarities to the earlier approaches.

`x = 11` in a function declaration is more like `x !== undefined ? x : 11` than the much more common idiom `x || 11`, so you'll need to be careful in converting your pre-ES6 code to this ES6 default parameter value syntax.

Note: A rest/gather parameter (see "Spread/Rest") cannot have a default value. So, while `function foo(...vals=[1,2,3]) { }` might seem an intriguing capability, it's not valid syntax. You'll need to continue to apply that sort of logic manually if necessary.

Default Value Expressions

Function default values can be more than just simple values like `31`; they can be any valid expression, even a function call:

```
function bar(val) {
  console.log("bar called!");
  return y + val;
}

function foo(x = y + 3, z = bar(x)) {
  console.log(x, z);
}

var y = 5;
foo();                                // "bar called"
                                         // 8 13
foo( 10 );                            // "bar called"
                                         // 10 15
y = 6;
foo( undefined, 10 );                  // 9 10
```

As you can see, the default value expressions are lazily evaluated, meaning they're only run if and when they're needed -- that is, when a parameter's argument is omitted or is `undefined`.

It's a subtle detail, but the formal parameters in a function declaration are in their own scope (think of it as a scope bubble wrapped around just the `(...)` of the function declaration), not in the function body's scope. That means a reference to an identifier in a default value expression first matches the formal parameters' scope before looking to an outer scope. See the *Scope & Closures* title of this series for more information.

Consider:

```
var w = 1, z = 2;

function foo( x = w + 1, y = x + 1, z = z + 1 ) {
    console.log( x, y, z );
}

foo();           // ReferenceError
```

The `w` in the `w + 1` default value expression looks for `w` in the formal parameters' scope, but does not find it, so the outer scope's `w` is used. Next, The `x` in the `x + 1` default value expression finds `x` in the formal parameters' scope, and luckily `x` has already been initialized, so the assignment to `y` works fine.

However, the `z` in `z + 1` finds `z` as a not-yet-initialized-at-that-moment parameter variable, so it never tries to find the `z` from the outer scope.

As we mentioned in the "let Declarations" section earlier in this chapter, ES6 has a TDZ, which prevents a variable from being accessed in its uninitialized state. As such, the `z + 1` default value expression throws a TDZ `ReferenceError` error.

Though it's not necessarily a good idea for code clarity, a default value expression can even be an inline function expression call -- commonly referred to as an immediately invoked function expression (IIFE):

```
function foo( x =
    (function(v){ return v + 11; })( 31 )
) {
    console.log( x );
}

foo();           // 42
```

There will very rarely be any cases where an IIFE (or any other executed inline function expression) will be appropriate for default value expressions. If you find yourself tempted to do this, take a step back and reevaluate!

Warning: If the IIFE had tried to access the `x` identifier and had not declared its own `x`, this would also have been a TDZ error, just as discussed before.

The default value expression in the previous snippet is an IIFE in that in the sense that it's a function that's executed right inline, via `(31)`. If we had left that part off, the default value assigned to `x` would have just been a function reference itself, perhaps like a default callback. There will probably be cases where that pattern will be quite useful, such as:

```
function ajax(url, cb = function(){}) {
    // ...
}

ajax( "http://some.url.1" );
```

In this case, we essentially want to default `cb` to be a no-op empty function call if not otherwise specified. The function expression is just a function reference, not a function call itself (no invoking `()` on the end of it), which accomplishes that goal.

Since the early days of JS, there's been a little-known but useful quirk available to us:

`Function.prototype` is itself an empty no-op function. So, the declaration could have been `cb = Function.prototype` and saved the inline function expression creation.

Destructuring

ES6 introduces a new syntactic feature called *destructuring*, which may be a little less confusing if you instead think of it as *structured assignment*. To understand this meaning, consider:

```
function foo() {
    return [1,2,3];
}

var tmp = foo(),
    a = tmp[0], b = tmp[1], c = tmp[2];

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

As you can see, we created a manual assignment of the values in the array that `foo()` returns to individual variables `a`, `b`, and `c`, and to do so we (unfortunately) needed the `tmp` variable.

Similarly, we can do the following with objects:

```

function bar() {
  return {
    x: 4,
    y: 5,
    z: 6
  };
}

var tmp = bar(),
  x = tmp.x, y = tmp.y, z = tmp.z;

console.log( x, y, z );           // 4 5 6

```

The `tmp.x` property value is assigned to the `x` variable, and likewise for `tmp.y` to `y` and `tmp.z` to `z`.

Manually assigning indexed values from an array or properties from an object can be thought of as *structured assignment*. ES6 adds a dedicated syntax for *destructuring*, specifically *array destructuring* and *object destructuring*. This syntax eliminates the need for the `tmp` variable in the previous snippets, making them much cleaner. Consider:

```

var [ a, b, c ] = foo();
var { x: x, y: y, z: z } = bar();

console.log( a, b, c );           // 1 2 3
console.log( x, y, z );           // 4 5 6

```

You're likely more accustomed to seeing syntax like `[a,b,c]` on the righthand side of an `=` assignment, as the value being assigned.

Destructuring symmetrically flips that pattern, so that `[a,b,c]` on the lefthand side of the `=` assignment is treated as a kind of "pattern" for decomposing the righthand side array value into separate variable assignments.

Similarly, `{ x: x, y: y, z: z }` specifies a "pattern" to decompose the object value from `bar()` into separate variable assignments.

Object Property Assignment Pattern

Let's dig into that `{ x: x, ... }` syntax from the previous snippet. If the property name being matched is the same as the variable you want to declare, you can actually shorten the syntax:

```
var { x, y, z } = bar();

console.log( x, y, z );           // 4 5 6
```

Pretty cool, right?

But is `{ x, ... }` leaving off the `x:` part or leaving off the `: x` part? We're actually leaving off the `x:` part when we use the shorter syntax. That may not seem like an important detail, but you'll understand its importance in just a moment.

If you can write the shorter form, why would you ever write out the longer form? Because that longer form actually allows you to assign a property to a different variable name, which can sometimes be quite useful:

```
var { x: bam, y: baz, z: bap } = bar();

console.log( bam, baz, bap );      // 4 5 6
console.log( x, y, z );           // ReferenceError
```

There's a subtle but super-important quirk to understand about this variation of the object destructuring form. To illustrate why it can be a gotcha you need to be careful of, let's consider the "pattern" of how normal object literals are specified:

```
var X = 10, Y = 20;

var o = { a: X, b: Y };

console.log( o.a, o.b );          // 10 20
```

In `{ a: X, b: Y }`, we know that `a` is the object property, and `X` is the source value that gets assigned to it. In other words, the syntactic pattern is `target: source`, or more obviously, `property-alias: value`. We intuitively understand this because it's the same as `= assignment`, where the pattern is `target = source`.

However, when you use object destructuring assignment -- that is, putting the `{ ... }` object literal-looking syntax on the lefthand side of the `=` operator -- you invert that `target: source` pattern.

Recall:

```
var { x: bam, y: baz, z: bap } = bar();
```

The syntactic pattern here is `source: target` (or `value: variable-alias`). `x: bam` means the `x` property is the source value and `bam` is the target variable to assign to. In other words, object literals are `target <- source`, and object destructuring assignments are `source --> target`. See how that's flipped?

There's another way to think about this syntax though, which may help ease the confusion. Consider:

```
var aa = 10, bb = 20;

var o = { x: aa, y: bb };
var { x: AA, y: BB } = o;

console.log( AA, BB );           // 10 20
```

In the `{ x: aa, y: bb }` line, the `x` and `y` represent the object properties. In the `{ x: AA, y: BB }` line, the `x` and the `y` also represent the object properties.

Recall how earlier I asserted that `{ x, ... }` was leaving off the `x:` part? In those two lines, if you erase the `x:` and `y:` parts in that snippet, you're left only with `aa, bb` and `AA, BB`, which in effect -- only conceptually, not actually -- are assignments from `aa` to `AA` and from `bb` to `BB`.

So, that symmetry may help to explain why the syntactic pattern was intentionally flipped for this ES6 feature.

Note: I would have preferred the syntax to be `{ AA: x, BB: y }` for the destructuring assignment, as that would have preserved consistency of the more familiar `target: source` pattern for both usages. Alas, I'm having to train my brain for the inversion, as some readers may also have to do.

Not Just Declarations

So far, we've used destructuring assignment with `var` declarations (of course, they could also use `let` and `const`), but destructuring is a general assignment operation, not just a declaration.

Consider:

```
var a, b, c, x, y, z;

[a,b,c] = foo();
( { x, y, z } = bar() );

console.log( a, b, c );           // 1 2 3
console.log( x, y, z );           // 4 5 6
```

The variables can already be declared, and then the destructuring only does assignments, exactly as we've already seen.

Note: For the object destructuring form specifically, when leaving off a `var` / `let` / `const` declarator, we had to surround the whole assignment expression in `()`, because otherwise the `{ ... }` on the lefthand side as the first element in the statement is taken to be a block statement instead of an object.

In fact, the assignment expressions (`a`, `y`, etc.) don't actually need to be just variable identifiers. Anything that's a valid assignment expression is allowed. For example:

```
var o = {};

[o.a, o.b, o.c] = foo();
( { x: o.x, y: o.y, z: o.z } = bar() );

console.log( o.a, o.b, o.c );           // 1 2 3
console.log( o.x, o.y, o.z );           // 4 5 6
```

You can even use computed property expressions in the destructuring. Consider:

```
var which = "x",
    o = {};

( { [which]: o[which] } = bar() );

console.log( o.x );                  // 4
```

The `[which]:` part is the computed property, which results in `x` -- the property to destructure from the object in question as the source of the assignment. The `o[which]` part is just a normal object key reference, which equates to `o.x` as the target of the assignment.

You can use the general assignments to create object mappings/transformations, such as:

```
var o1 = { a: 1, b: 2, c: 3 },
o2 = {};

( { a: o2.x, b: o2.y, c: o2.z } = o1 );

console.log( o2.x, o2.y, o2.z );      // 1 2 3
```

Or you can map an object to an array, such as:

```
var o1 = { a: 1, b: 2, c: 3 },
a2 = [];

( { a: a2[0], b: a2[1], c: a2[2] } = o1 );

console.log( a2 );                  // [1,2,3]
```

Or the other way around:

```
var a1 = [ 1, 2, 3 ],
o2 = {};

[ o2.a, o2.b, o2.c ] = a1;

console.log( o2.a, o2.b, o2.c );    // 1 2 3
```

Or you could reorder one array to another:

```
var a1 = [ 1, 2, 3 ],
a2 = [];

[ a2[2], a2[0], a2[1] ] = a1;

console.log( a2 );                // [2,3,1]
```

You can even solve the traditional "swap two variables" task without a temporary variable:

```
var x = 10, y = 20;

[ y, x ] = [ x, y ];

console.log( x, y );              // 20 10
```

Warning: Be careful: you shouldn't mix in declaration with assignment unless you want all of the assignment expressions *also* to be treated as declarations. Otherwise, you'll get syntax errors. That's why in the earlier example I had to do `var a2 = []` separately from the `[`

`a2[0], ...] = ...` destructuring assignment. It wouldn't make any sense to try `var [a2[0], ...] = ...`, because `a2[0]` isn't a valid declaration identifier; it also obviously couldn't implicitly create a `var a2 = []` declaration to use.

Repeated Assignments

The object destructuring form allows a source property (holding any value type) to be listed multiple times. For example:

```
var { a: X, a: Y } = { a: 1 };

X;    // 1
Y;    // 1
```

That also means you can both destructure a sub-object/array property and also capture the sub-object/array's value itself. Consider:

```
var { a: { x: X, x: Y }, a } = { a: { x: 1 } };

X;    // 1
Y;    // 1
a;    // { x: 1 }

( { a: X, a: Y, a: [ Z ] } = { a: [ 1 ] } );

X.push( 2 );
Y[0] = 10;

X;    // [10, 2]
Y;    // [10, 2]
Z;    // 1
```

A word of caution about destructuring: it may be tempting to list destructuring assignments all on a single line as has been done thus far in our discussion. However, it's a much better idea to spread destructuring assignment patterns over multiple lines, using proper indentation -- much like you would in JSON or with an object literal value -- for readability sake.

```
// harder to read:  
var { a: { b: [ c, d ], e: { f } }, g } = obj;  
  
// better:  
var {  
  a: {  
    b: [ c, d ],  
    e: { f }  
  },  
  g  
} = obj;
```

Remember: **the purpose of destructuring is not just less typing, but more declarative readability.**

Destructuring Assignment Expressions

The assignment expression with object or array destructuring has as its completion value the full righthand object/array value. Consider:

```
var o = { a:1, b:2, c:3 },  
       a, b, c, p;  
  
p = { a, b, c } = o;  
  
console.log( a, b, c );           // 1 2 3  
p === o;                         // true
```

In the previous snippet, `p` was assigned the `o` object reference, not one of the `a`, `b`, or `c` values. The same is true of array destructuring:

```
var o = [1,2,3],  
       a, b, c, p;  
  
p = [ a, b, c ] = o;  
  
console.log( a, b, c );           // 1 2 3  
p === o;                         // true
```

By carrying the object/array value through as the completion, you can chain destructuring assignment expressions together:

```

var o = { a:1, b:2, c:3 },
    p = [4,5,6],
    a, b, c, x, y, z;

( {a} = {b,c} = o );
[x,y] = [z] = p;

console.log( a, b, c );           // 1 2 3
console.log( x, y, z );           // 4 5 6

```

Too Many, Too Few, Just Enough

With both array destructuring assignment and object destructuring assignment, you do not have to assign all the values that are present. For example:

```

var [,b] = foo();
var { x, z } = bar();

console.log( b, x, z );           // 2 4 6

```

The `1` and `3` values that came back from `foo()` are discarded, as is the `5` value from `bar()`.

Similarly, if you try to assign more values than are present in the value you're destructuring/decomposing, you get graceful fallback to `undefined`, as you'd expect:

```

var [,,c,d] = foo();
var { w, z } = bar();

console.log( c, z );           // 3 6
console.log( d, w );           // undefined undefined

```

This behavior follows symmetrically from the earlier stated "`undefined` is missing" principle.

We examined the `...` operator earlier in this chapter, and saw that it can sometimes be used to spread an array value out into its separate values, and sometimes it can be used to do the opposite: to gather a set of values together into an array.

In addition to the gather/rest usage in function declarations, `...` can perform the same behavior in destructuring assignments. To illustrate, let's recall a snippet from earlier in this chapter:

```
var a = [2,3,4];
var b = [ 1, ...a, 5 ];

console.log( b );           // [1,2,3,4,5]
```

Here we see that `...a` is spreading `a` out, because it appears in the array `[...]` value position. If `...a` appears in an array destructuring position, it performs the gather behavior:

```
var a = [2,3,4];
var [ b, ...c ] = a;

console.log( b, c );        // 2 [3,4]
```

The `var [...] = a` destructuring assignment spreads `a` out to be assigned to the pattern described inside the `[...]`. The first part names `b` for the first value in `a` (`2`). But then `...c` gathers the rest of the values (`3` and `4`) into an array and calls it `c`.

Note: We've seen how `...` works with arrays, but what about with objects? It's not an ES6 feature, but see Chapter 8 for discussion of a possible "beyond ES6" feature where `...` works with spreading or gathering objects.

Default Value Assignment

Both forms of destructuring can offer a default value option for an assignment, using the `=` syntax similar to the default function argument values discussed earlier.

Consider:

```
var [ a = 3, b = 6, c = 9, d = 12 ] = foo();
var { x = 5, y = 10, z = 15, w = 20 } = bar();

console.log( a, b, c, d );           // 1 2 3 12
console.log( x, y, z, w );          // 4 5 6 20
```

You can combine the default value assignment with the alternative assignment expression syntax covered earlier. For example:

```
var { x, y, z, w: ww = 20 } = bar();

console.log( x, y, z, ww );         // 4 5 6 20
```

Be careful about confusing yourself (or other developers who read your code) if you use an object or array as the default value in a destructuring. You can create some really hard to understand code:

```
var x = 200, y = 300, z = 100;
var o1 = { x: { y: 42 }, z: { y: z } };

( { y: x = { y: y } } = o1 );
( { z: y = { y: z } } = o1 );
( { x: z = { y: x } } = o1 );
```

Can you tell from that snippet what values `x`, `y`, and `z` have at the end? Takes a moment of pondering, I would imagine. I'll end the suspense:

```
console.log( x.y, y.y, z.y );           // 300 100 42
```

The takeaway here: destructuring is great and can be very useful, but it's also a sharp sword that can cause injury (to someone's brain) if used unwisely.

Nested Destructuring

If the values you're destructuring have nested objects or arrays, you can destructure those nested values as well:

```
var a1 = [ 1, [2, 3, 4], 5 ];
var o1 = { x: { y: { z: 6 } } };

var [ a, [ b, c, d ], e ] = a1;
var { x: { y: { z: w } } } = o1;

console.log( a, b, c, d, e );           // 1 2 3 4 5
console.log( w );                      // 6
```

Nested destructuring can be a simple way to flatten out object namespaces. For example:

```
var App = {
  model: {
    User: function(){ ... }
  }
};

// instead of:
// var User = App.model.User;

var { model: { User } } = App;
```

Destructuring Parameters

In the following snippet, can you spot the assignment?

```
function foo(x) {
  console.log( x );
}

foo( 42 );
```

The assignment is kinda hidden: `42` (the argument) is assigned to `x` (the parameter) when `foo(42)` is executed. If parameter/argument pairing is an assignment, then it stands to reason that it's an assignment that could be destructured, right? Of course!

Consider array destructuring for parameters:

```
function foo( [ x, y ] ) {
  console.log( x, y );
}

foo( [ 1, 2 ] );           // 1 2
foo( [ 1 ] );             // 1 undefined
foo( [] );                // undefined undefined
```

Object destructuring for parameters works, too:

```
function foo( { x, y } ) {
  console.log( x, y );
}

foo( { y: 1, x: 2 } );      // 2 1
foo( { y: 42 } );          // undefined 42
foo( {} );                 // undefined undefined
```

This technique is an approximation of named arguments (a long requested feature for JS!), in that the properties on the object map to the destructured parameters of the same names. That also means that we get optional parameters (in any position) for free, as you can see leaving off the `x` "parameter" worked as we'd expect.

Of course, all the previously discussed variations of destructuring are available to us with parameter destructuring, including nested destructuring, default values, and more.

Destructuring also mixes fine with other ES6 function parameter capabilities, like default parameter values and rest/gather parameters.

Consider these quick illustrations (certainly not exhaustive of the possible variations):

```

function f1([ x=2, y=3, z ]) { ... }
function f2([ x, y, ...z ], w) { ... }
function f3([ x, y, ...z ], ...w) { ... }

function f4({ x: X, y }) { ... }
function f5({ x: X = 10, y = 20 }) { ... }
function f6({ x = 10 } = {}, { y } = { y: 10 }) { ... }

```

Let's take one example from this snippet and examine it, for illustration purposes:

```

function f3([ x, y, ...z ], ...w) {
    console.log( x, y, z, w );
}

f3();                                // undefined undefined []
f3( [1,2,3,4], 5, 6 );                // 1 2 [3,4] [5,6]

```

There are two `...` operators in use here, and they're both gathering values in arrays (`z` and `w`), though `...z` gathers from the rest of the values left over in the first array argument, while `...w` gathers from the rest of the main arguments left over after the first.

Destructuring Defaults + Parameter Defaults

There's one subtle point you should be particularly careful to notice -- the difference in behavior between a destructuring default value and a function parameter default value. For example:

```

function f6({ x = 10 } = {}, { y } = { y: 10 }) {
    console.log( x, y );
}

f6();                                // 10 10

```

At first, it would seem that we've declared a default value of `10` for both the `x` and `y` parameters, but in two different ways. However, these two different approaches will behave differently in certain cases, and the difference is awfully subtle.

Consider:

```
f6( {}, {} );                      // 10 undefined
```

Wait, why did that happen? It's pretty clear that named parameter `x` is defaulting to `10` if not passed as a property of that same name in the first argument's object.

But what about `y` being `undefined`? The `{ y: 10 }` value is an object as a function parameter default value, not a destructuring default value. As such, it only applies if the second argument is not passed at all, or is passed as `undefined`.

In the previous snippet, we are passing a second argument (`{}`), so the default `{ y: 10 }` value is not used, and the `{ y }` destructuring occurs against the passed in `{}` empty object value.

Now, compare `{ y } = { y: 10 }` to `{ x = 10 } = {}`.

For the `x`'s form usage, if the first function argument is omitted or `undefined`, the `{}` empty object default applies. Then, whatever value is in the first argument position -- either the default `{}` or whatever you passed in -- is destructured with the `{ x = 10 }`, which checks to see if an `x` property is found, and if not found (or `undefined`), the `10` default value is applied to the `x` named parameter.

Deep breath. Read back over those last few paragraphs a couple of times. Let's review via code:

```
function f6({ x = 10 } = {}, { y } = { y: 10 }) {
  console.log(x, y);
}

f6();                                // 10 10
f6( undefined, undefined );          // 10 10
f6( {}, undefined );                // 10 10

f6( {}, {} );                      // 10 undefined
f6( undefined, {} );                // 10 undefined

f6( { x: 2 }, { y: 3 } );          // 2 3
```

It would generally seem that the defaulting behavior of the `x` parameter is probably the more desirable and sensible case compared to that of `y`. As such, it's important to understand why and how `{ x = 10 } = {}` form is different from `{ y } = { y: 10 }` form.

If that's still a bit fuzzy, go back and read it again, and play with this yourself. Your future self will thank you for taking the time to get this very subtle gotcha nuance detail straight.

Nested Defaults: Destructured and Restructured

Although it may at first be difficult to grasp, an interesting idiom emerges for setting defaults for a nested object's properties: using object destructuring along with what I'd call *restructuring*.

Consider a set of defaults in a nested object structure, like the following:

```
// taken from: http://es-discourse.com/t/partial-default-arguments/120/7

var defaults = {
  options: {
    remove: true,
    enable: false,
    instance: {}
  },
  log: {
    warn: true,
    error: true
  }
};
```

Now, let's say that you have an object called `config`, which has some of these applied, but perhaps not all, and you'd like to set all the defaults into this object in the missing spots, but not override specific settings already present:

```
var config = {
  options: {
    remove: false,
    instance: null
  }
};
```

You can of course do so manually, as you might have done in the past:

```
config.options = config.options || {};
config.options.remove = (config.options.remove !== undefined) ?
  config.options.remove : defaults.options.remove;
config.options.enable = (config.options.enable !== undefined) ?
  config.options.enable : defaults.options.enable;
...
```

Yuck.

Others may prefer the assign-overwrite approach to this task. You might be tempted by the ES6 `Object.assign(..)` utility (see Chapter 6) to clone the properties first from `defaults` and then overwritten with the cloned properties from `config`, as so:

```
config = Object.assign( {}, defaults, config );
```

That looks way nicer, huh? But there's a major problem! `Object.assign(..)` is shallow, which means when it copies `defaults.options`, it just copies that object reference, not deep cloning that object's properties to a `config.options` object. `Object.assign(..)` would need

to be applied (sort of "recursively") at all levels of your object's tree to get the deep cloning you're expecting.

Note: Many JS utility libraries/frameworks provide their own option for deep cloning of an object, but those approaches and their gotchas are beyond our scope to discuss here.

So let's examine if ES6 object destructuring with defaults can help at all:

```
config.options = config.options || {};
config.log = config.log || {};
({
  options: {
    remove: config.options.remove = defaults.options.remove,
    enable: config.options.enable = defaults.options.enable,
    instance: config.options.instance = defaults.options.instance
  } = {},
  log: {
    warn: config.log.warn = defaults.log.warn,
    error: config.log.error = defaults.log.error
  } = {}
} = config);
```

Not as nice as the false promise of `object.assign(..)` (being that it's shallow only), but it's better than the manual approach by a fair bit, I think. It is still unfortunately verbose and repetitive, though.

The previous snippet's approach works because I'm hacking the destructuring and defaults mechanism to do the property `== undefined` checks and assignment decisions for me. It's a trick in that I'm destructuring `config` (see the `= config` at the end of the snippet), but I'm reassigning all the destructured values right back into `config`, with the `config.options.enable` assignment references.

Still too much, though. Let's see if we can make anything better.

The following trick works best if you know that all the various properties you're destructuring are uniquely named. You can still do it even if that's not the case, but it's not as nice -- you'll have to do the destructuring in stages, or create unique local variables as temporary aliases.

If we fully destructure all the properties into top-level variables, we can then immediately restructure to reconstitute the original nested object structure.

But all those temporary variables hanging around would pollute scope. So, let's use block scoping (see "Block-Scoped Declarations" earlier in this chapter) with a general `{ }` enclosing block:

```
// merge `defaults` into `config`
{
  // destructure (with default value assignments)
  let {
    options: {
      remove = defaults.options.remove,
      enable = defaults.options.enable,
      instance = defaults.options.instance
    } = {},
    log: {
      warn = defaults.log.warn,
      error = defaults.log.error
    } = {}
  } = config;

  // restructure
  config = {
    options: { remove, enable, instance },
    log: { warn, error }
  };
}
```

That seems a fair bit nicer, huh?

Note: You could also accomplish the scope enclosure with an arrow IIFE instead of the general `{ }` block and `let` declarations. Your destructuring assignments/defaults would be in the parameter list and your restructuring would be the `return` statement in the function body.

The `{ warn, error }` syntax in the restructuring part may look new to you; that's called "concise properties" and we cover it in the next section!

Object Literal Extensions

ES6 adds a number of important convenience extensions to the humble `{ .. }` object literal.

Concise Properties

You're certainly familiar with declaring object literals in this form:

```
var x = 2, y = 3,
  o = {
    x: x,
    y: y
  };

```

If it's always felt redundant to say `x: x` all over, there's good news. If you need to define a property that is the same name as a lexical identifier, you can shorten it from `x: x` to `x`. Consider:

```
var x = 2, y = 3,
  o = {
    x,
    y
  };

```

Concise Methods

In a similar spirit to concise properties we just examined, functions attached to properties in object literals also have a concise form, for convenience.

The old way:

```
var o = {
  x: function(){
    // ..
  },
  y: function(){
    // ..
  }
}

```

And as of ES6:

```
var o = {
  x() {
    // ..
  },
  y() {
    // ..
  }
}

```

Warning: While `x() { .. }` seems to just be shorthand for `x: function(){ .. }`, concise methods have special behaviors that their older counterparts don't; specifically, the allowance for `super` (see "Object `super`" later in this chapter).

Generators (see Chapter 4) also have a concise method form:

```
var o = {
  *foo() { .. }
};
```

Concisely Unnamed

While that convenience shorthand is quite attractive, there's a subtle gotcha to be aware of. To illustrate, let's examine pre-ES6 code like the following, which you might try to refactor to use concise methods:

```
function runSomething(o) {
  var x = Math.random(),
      y = Math.random();

  return o.something( x, y );
}

runSomething( {
  something: function something(x,y) {
    if (x > y) {
      // recursively call with `x`
      // and `y` swapped
      return something( y, x );
    }

    return y - x;
  }
} );
```

This obviously silly code just generates two random numbers and subtracts the smaller from the bigger. But what's important here isn't what it does, but rather how it's defined. Let's focus on the object literal and function definition, as we see here:

```
runSomething( {
  something: function something(x,y) {
    // ..
  }
} );
```

Why do we say both `something:` and `function something`? Isn't that redundant? Actually, no, both are needed for different purposes. The property `something` is how we can call `o.something(...)`, sort of like its public name. But the second `something` is a lexical name to refer to the function from inside itself, for recursion purposes.

Can you see why the line `return something(y,x)` needs the name `something` to refer to the function? There's no lexical name for the object, such that it could have said `return o.something(y,x)` or something of that sort.

That's actually a pretty common practice when the object literal does have an identifying name, such as:

```
var controller = {
  makeRequest: function(...){
    // ...
    controller.makeRequest(...);
  }
};
```

Is this a good idea? Perhaps, perhaps not. You're assuming that the name `controller` will always point to the object in question. But it very well may not -- the `makeRequest(...)` function doesn't control the outer code and so can't force that to be the case. This could come back to bite you.

Others prefer to use `this` to define such things:

```
var controller = {
  makeRequest: function(...){
    // ...
    this.makeRequest(...);
  }
};
```

That looks fine, and should work if you always invoke the method as `controller.makeRequest(...)`. But you now have a `this` binding gotcha if you do something like:

```
btn.addEventListener( "click", controller.makeRequest, false );
```

Of course, you can solve that by passing `controller.makeRequest.bind(controller)` as the handler reference to bind the event to. But yuck -- it isn't very appealing.

Or what if your inner `this.makeRequest(..)` call needs to be made from a nested function? You'll have another `this` binding hazard, which people will often solve with the hacky `var self = this`, such as:

```
var controller = {
  makeRequest: function(...){
    var self = this;

    btn.addEventListener( "click", function(){
      // ..
      self.makeRequest(..);
    }, false );
  }
};
```

More yuck.

Note: For more information on `this` binding rules and gotchas, see Chapters 1-2 of the *this & Object Prototypes* title of this series.

OK, what does all this have to do with concise methods? Recall our `something(..)` method definition:

```
runSomething( {
  something: function something(x,y) {
    // ..
  }
});
```

The second `something` here provides a super convenient lexical identifier that will always point to the function itself, giving us the perfect reference for recursion, event binding/unbinding, and so on -- no messing around with `this` or trying to use an untrustable object reference.

Great!

So, now we try to refactor that function reference to this ES6 concise method form:

```

runSomething( {
    something(x,y) {
        if (x > y) {
            return something( y, x );
        }

        return y - x;
    }
} );

```

Seems fine at first glance, except this code will break. The `return something(..)` call will not find a `something` identifier, so you'll get a `ReferenceError`. Oops. But why?

The above ES6 snippet is interpreted as meaning:

```

runSomething( {
    something: function(x,y){
        if (x > y) {
            return something( y, x );
        }

        return y - x;
    }
} );

```

Look closely. Do you see the problem? The concise method definition implies `something: function(x,y)`. See how the second `something` we were relying on has been omitted? In other words, concise methods imply anonymous function expressions.

Yeah, yuck.

Note: You may be tempted to think that `=>` arrow functions are a good solution here, but they're equally insufficient, as they're also anonymous function expressions. We'll cover them in "Arrow Functions" later in this chapter.

The partially redeeming news is that our `something(x,y)` concise method won't be totally anonymous. See "Function Names" in Chapter 7 for information about ES6 function name inference rules. That won't help us for our recursion, but it helps with debugging at least.

So what are we left to conclude about concise methods? They're short and sweet, and a nice convenience. But you should only use them if you're never going to need them to do recursion or event binding/unbinding. Otherwise, stick to your old-school `something: function something(..)` method definitions.

A lot of your methods are probably going to benefit from concise method definitions, so that's great news! Just be careful of the few where there's an un-naming hazard.

ES5 Getter/Setter

Technically, ES5 defined getter/setter literals forms, but they didn't seem to get used much, mostly due to the lack of transpilers to handle that new syntax (the only major new syntax added in ES5, really). So while it's not a new ES6 feature, we'll briefly refresh on that form, as it's probably going to be much more useful with ES6 going forward.

Consider:

```
var o = {
  __id: 10,
  get id() { return this.__id++; },
  set id(v) { this.__id = v; }
}

o.id;          // 10
o.id;          // 11
o.id = 20;
o.id;          // 20

// and:
o.__id;        // 21
o.__id;        // 21 -- still!
```

These getter and setter literal forms are also present in classes; see Chapter 3.

Warning: It may not be obvious, but the setter literal must have exactly one declared parameter; omitting it or listing others is illegal syntax. The single required parameter *can* use destructuring and defaults (e.g., `set id({ id: v = 0 }) { ... }`), but the gather/rest `... is not allowed (set id(...v) { ... }).`

Computed Property Names

You've probably been in a situation like the following snippet, where you have one or more property names that come from some sort of expression and thus can't be put into the object literal:

```
var prefix = "user_";

var o = {
  baz: function(...){ ... }
};

o[ prefix + "foo" ] = function(...){ ... };
o[ prefix + "bar" ] = function(...){ ... };
...
```

ES6 adds a syntax to the object literal definition which allows you to specify an expression that should be computed, whose result is the property name assigned. Consider:

```
var prefix = "user_";

var o = {
  baz: function(...){ ... },
  [prefix + "foo": function(...){ ... },
  [prefix + "bar": function(...){ ... }
  ...
};
```

Any valid expression can appear inside the `[...]` that sits in the property name position of the object literal definition.

Probably the most common use of computed property names will be with `Symbol`s (which we cover in "Symbols" later in this chapter), such as:

```
var o = {
  [Symbol.toStringTag]: "really cool thing",
  ...
};
```

`Symbol.toStringTag` is a special built-in value, which we evaluate with the `[...]` syntax, so we can assign the `"really cool thing"` value to the special property name.

Computed property names can also appear as the name of a concise method or a concise generator:

```
var o = {
  ["f" + "oo"](): { ... } // computed concise method
  *[["b" + "ar"](): { ... } // computed concise generator
};
```

Setting `[[Prototype]]`

We won't cover prototypes in detail here, so for more information, see the *this & Object Prototypes* title of this series.

Sometimes it will be helpful to assign the `[[Prototype]]` of an object at the same time you're declaring its object literal. The following has been a nonstandard extension in many JS engines for a while, but is standardized as of ES6:

```
var o1 = {
  // ...
};

var o2 = {
  __proto__: o1,
  // ...
};
```

`o2` is declared with a normal object literal, but it's also `[[Prototype]]`-linked to `o1`. The `__proto__` property name here can also be a string `"__proto__"`, but note that it *cannot* be the result of a computed property name (see the previous section).

`__proto__` is controversial, to say the least. It's a decades-old proprietary extension to JS that is finally standardized, somewhat begrudgingly it seems, in ES6. Many developers feel it shouldn't ever be used. In fact, it's in "Annex B" of ES6, which is the section that lists things JS feels it has to standardize for compatibility reasons only.

Warning: Though I'm narrowly endorsing `__proto__` as a key in an object literal definition, I definitely do not endorse using it in its object property form, like `o.__proto__`. That form is both a getter and setter (again for compatibility reasons), but there are definitely better options. See the *this & Object Prototypes* title of this series for more information.

For setting the `[[Prototype]]` of an existing object, you can use the ES6 utility `Object.setPrototypeOf(..)`. Consider:

```
var o1 = {
  // ...
};

var o2 = {
  // ...
};

Object.setPrototypeOf( o2, o1 );
```

Note: We'll discuss `Object` again in Chapter 6. " `Object.setPrototypeOf(..)` Static Function" provides additional details on `Object.setPrototypeOf(..)`. Also see " `Object.assign(..)` Static Function" for another form that relates `o2` prototypically to `o1`.

Object super

`super` is typically thought of as being only related to classes. However, due to JS's classless-objects-with-prototypes nature, `super` is equally effective, and nearly the same in behavior, with plain objects' concise methods.

Consider:

```
var o1 = {
  foo() {
    console.log( "o1:foo" );
  }
};

var o2 = {
  foo() {
    super.foo();
    console.log( "o2:foo" );
  }
};

Object.setPrototypeOf( o2, o1 );

o2.foo();           // o1:foo
                  // o2:foo
```

Warning: `super` is only allowed in concise methods, not regular function expression properties. It also is only allowed in `super.xxx` form (for property/method access), not in `super()` form.

The `super` reference in the `o2.foo()` method is locked statically to `o2`, and specifically to the `[[Prototype]]` of `o2`. `super` here would basically be `Object.getPrototypeOf(o2)` -- resolves to `o1` of course -- which is how it finds and calls `o1.foo()`.

For complete details on `super`, see "Classes" in Chapter 3.

Template Literals

At the very outset of this section, I'm going to have to call out the name of this ES6 feature as being awfully... misleading, depending on your experiences with what the word *template* means.

Many developers think of templates as being reusable renderable pieces of text, such as the capability provided by most template engines (Mustache, Handlebars, etc.). ES6's use of the word *template* would imply something similar, like a way to declare inline template literals that can be re-rendered. However, that's not at all the right way to think about this feature.

So, before we go on, I'm renaming to what it should have been called: *interpolated string literals* (or *interpoliterals* for short).

You're already well aware of declaring string literals with `"` or `'` delimiters, and you also know that these are not *smart strings* (as some languages have), where the contents would be parsed for interpolation expressions.

However, ES6 introduces a new type of string literal, using the ``` backtick as the delimiter. These string literals allow basic string interpolation expressions to be embedded, which are then automatically parsed and evaluated.

Here's the old pre-ES6 way:

```
var name = "Kyle";  
  
var greeting = "Hello " + name + "!";  
  
console.log( greeting );           // "Hello Kyle!"  
console.log( typeof greeting );    // "string"
```

Now, consider the new ES6 way:

```
var name = "Kyle";  
  
var greeting = `Hello ${name}!`;  
  
console.log( greeting );           // "Hello Kyle!"  
console.log( typeof greeting );    // "string"
```

As you can see, we used the ``...`` around a series of characters, which are interpreted as a string literal, but any expressions of the form `${..}` are parsed and evaluated inline immediately. The fancy term for such parsing and evaluating is *interpolation* (much more accurate than templating).

The result of the interpolated string literal expression is just a plain old normal string, assigned to the `greeting` variable.

Warning: `typeof greeting == "string"` illustrates why it's important not to think of these entities as special template values, as you cannot assign the unevaluated form of the literal to something and reuse it. The ``...`` string literal is more like an IIFE in the sense that it's automatically evaluated inline. The result of a ``...`` string literal is, simply, just a string.

One really nice benefit of interpolated string literals is they are allowed to split across multiple lines:

```
var text =
`Now is the time for all good men
to come to the aid of their
country!`;

console.log( text );
// Now is the time for all good men
// to come to the aid of their
// country!
```

The line breaks (newlines) in the interpolated string literal were preserved in the string value.

Unless appearing as explicit escape sequences in the literal value, the value of the `\r` carriage return character (code point `U+000D`) or the value of the `\r\n` carriage return + line feed sequence (code points `U+000D` and `U+000A`) are both normalized to a `\n` line feed character (code point `U+000A`). Don't worry though; this normalization is rare and would likely only happen if copy-pasting text into your JS file.

Interpolated Expressions

Any valid expression is allowed to appear inside `${...}` in an interpolated string literal, including function calls, inline function expression calls, and even other interpolated string literals!

Consider:

```
function upper(s) {
    return s.toUpperCase();
}

var who = "reader";

var text =
`A very ${upper( "warm" )} welcome
to all of you ${upper( `#${who}s` )}!`;

console.log( text );
// A very WARM welcome
// to all of you READERS!
```

Here, the inner ``${who}s`` interpolated string literal was a little bit nicer convenience for us when combining the `who` variable with the `"s"` string, as opposed to `who + "s"`. There will be cases that nesting interpolated string literals is helpful, but be wary if you find yourself doing that kind of thing often, or if you find yourself nesting several levels deep.

If that's the case, the odds are good that your string value production could benefit from some abstractions.

Warning: As a word of caution, be very careful about the readability of your code with such new found power. Just like with default value expressions and destructuring assignment expressions, just because you *can* do something doesn't mean you *should* do it. Never go so overboard with new ES6 tricks that your code becomes more clever than you or your other team members.

Expression Scope

One quick note about the scope that is used to resolve variables in expressions. I mentioned earlier that an interpolated string literal is kind of like an IIFE, and it turns out thinking about it like that explains the scoping behavior as well.

Consider:

```
function foo(str) {
  var name = "foo";
  console.log( str );
}

function bar() {
  var name = "bar";
  foo(`Hello from ${name}!`);
}

var name = "global";

bar();           // "Hello from bar!"
```

At the moment the ``...`` string literal is expressed, inside the `bar()` function, the scope available to it finds `bar()`'s `name` variable with value `"bar"`. Neither the global `name` nor `foo(...)`'s `name` matter. In other words, an interpolated string literal is just lexically scoped where it appears, not dynamically scoped in any way.

Tagged Template Literals

Again, renaming the feature for sanity sake: *tagged string literals*.

To be honest, this is one of the cooler tricks that ES6 offers. It may seem a little strange, and perhaps not all that generally practical at first. But once you've spent some time with it, tagged string literals may just surprise you in their usefulness.

For example:

```

function foo(strings, ...values) {
    console.log( strings );
    console.log( values );
}

var desc = "awesome";

foo`Everything is ${desc}!`;
// [ "Everything is ", "!" ]
// [ "awesome" ]

```

Let's take a moment to consider what's happening in the previous snippet. First, the most jarring thing that jumps out is `foo`Everything...``. That doesn't look like anything we've seen before. What is it?

It's essentially a special kind of function call that doesn't need the `(...)`. The *tag* -- the `foo` part before the ``...`` string literal -- is a function value that should be called. Actually, it can be any expression that results in a function, even a function call that returns another function, like:

```

function bar() {
    return function foo(strings, ...values) {
        console.log( strings );
        console.log( values );
    }
}

var desc = "awesome";

bar()`Everything is ${desc}!`;
// [ "Everything is ", "!" ]
// [ "awesome" ]

```

But what gets passed to the `foo(..)` function when invoked as a tag for a string literal?

The first argument -- we called it `strings` -- is an array of all the plain strings (the stuff between any interpolated expressions). We get two values in the `strings` array:

`"Everything is " and "!"`.

For convenience sake in our example, we then gather up all subsequent arguments into an array called `values` using the `...` gather/rest operator (see the "Spread/Rest" section earlier in this chapter), though you could of course have left them as individual named parameters following the `strings` parameter.

The argument(s) gathered into our `values` array are the results of the already-evaluated interpolation expressions found in the string literal. So obviously the only element in `values` in our example is `"awesome"`.

You can think of these two arrays as: the values in `values` are the separators if you were to splice them in between the values in `strings`, and then if you joined everything together, you'd get the complete interpolated string value.

A tagged string literal is like a processing step after the interpolation expressions are evaluated but before the final string value is compiled, allowing you more control over generating the string from the literal.

Typically, the string literal tag function (`foo(..)` in the previous snippets) should compute an appropriate string value and return it, so that you can use the tagged string literal as a value just like untagged string literals:

```
function tag(strings, ...values) {
  return strings.reduce( function(s,v,idx){
    return s + (idx > 0 ? values[idx-1] : "") + v;
  }, "" );
}

var desc = "awesome";

var text = tag`Everything is ${desc}!`;
console.log( text );           // Everything is awesome!
```

In this snippet, `tag(..)` is a pass-through operation, in that it doesn't perform any special modifications, but just uses `reduce(..)` to loop over and splice/interleave `strings` and `values` together the same way an untagged string literal would have done.

So what are some practical uses? There are many advanced ones that are beyond our scope to discuss here. But here's a simple idea that formats numbers as U.S. dollars (sort of like basic localization):

```

function dollabillsyall(strings, ...values) {
  return strings.reduce( function(s,v,idx){
    if (idx > 0) {
      if (typeof values[idx-1] == "number") {
        // look, also using interpolated
        // string literals!
        s += `$$${values[idx-1].toFixed( 2 )}`;
      }
      else {
        s += values[idx-1];
      }
    }

    return s + v;
  }, "");
}

var amt1 = 11.99,
  amt2 = amt1 * 1.08,
  name = "Kyle";

var text = dollabillsyall
`Thanks for your purchase, ${name}! Your
product cost was ${amt1}, which with tax
comes out to ${amt2}.`

console.log( text );
// Thanks for your purchase, Kyle! Your
// product cost was $11.99, which with tax
// comes out to $12.95.

```

If a `number` value is encountered in the `values` array, we put `"$"` in front of it and format it to two decimal places with `toFixed(2)`. Otherwise, we let the value pass-through untouched.

Raw Strings

In the previous snippets, our tag functions receive the first argument we called `strings`, which is an array. But there's an additional bit of data included: the raw unprocessed versions of all the strings. You can access those raw string values using the `.raw` property, like this:

```

function showraw(strings, ...values) {
  console.log( strings );
  console.log( strings.raw );
}

showraw`Hello\nWorld`;
// [ "Hello
// World" ]
// [ "Hello\nWorld" ]

```

The raw version of the value preserves the raw escaped `\n` sequence (the `\` and the `n` are separate characters), while the processed version considers it a single newline character. However, the earlier mentioned line-ending normalization is applied to both values.

ES6 comes with a built-in function that can be used as a string literal tag: `String.raw(...)`. It simply passes through the raw versions of the `strings` values:

```

console.log(`Hello\nWorld`);
// Hello
// World

console.log(String.raw`Hello\nWorld`);
// Hello\nWorld

String.raw`Hello\nWorld`.length;
// 12

```

Other uses for string literal tags included special processing for internationalization, localization, and more!

Arrow Functions

We've touched on `this` binding complications with functions earlier in this chapter, and they're covered at length in the *this & Object Prototypes* title of this series. It's important to understand the frustrations that `this`-based programming with normal functions brings, because that is the primary motivation for the new ES6 `=>` arrow function feature.

Let's first illustrate what an arrow function looks like, as compared to normal functions:

```

function foo(x,y) {
    return x + y;
}

// versus

var foo = (x,y) => x + y;

```

The arrow function definition consists of a parameter list (of zero or more parameters, and surrounding `(...)` if there's not exactly one parameter), followed by the `=>` marker, followed by a function body.

So, in the previous snippet, the arrow function is just the `(x,y) => x + y` part, and that function reference happens to be assigned to the variable `foo`.

The body only needs to be enclosed by `{ ... }` if there's more than one expression, or if the body consists of a non-expression statement. If there's only one expression, and you omit the surrounding `{ ... }`, there's an implied `return` in front of the expression, as illustrated in the previous snippet.

Here's some other arrow function variations to consider:

```

var f1 = () => 12;
var f2 = x => x * 2;
var f3 = (x,y) => {
    var z = x * 2 + y;
    y++;
    x *= 3;
    return (x + y + z) / 2;
};

```

Arrow functions are *always* function expressions; there is no arrow function declaration. It also should be clear that they are anonymous function expressions -- they have no named reference for the purposes of recursion or event binding/unbinding -- though "Function Names" in Chapter 7 will describe ES6's function name inference rules for debugging purposes.

Note: All the capabilities of normal function parameters are available to arrow functions, including default values, destructuring, rest parameters, and so on.

Arrow functions have a nice, shorter syntax, which makes them on the surface very attractive for writing terser code. Indeed, nearly all literature on ES6 (other than the titles in this series) seems to immediately and exclusively adopt the arrow function as "the new function."

It is telling that nearly all examples in discussion of arrow functions are short single statement utilities, such as those passed as callbacks to various utilities. For example:

```
var a = [1,2,3,4,5];
a = a.map( v => v * 2 );
console.log( a );           // [2,4,6,8,10]
```

In those cases, where you have such inline function expressions, and they fit the pattern of computing a quick calculation in a single statement and returning that result, arrow functions indeed look to be an attractive and lightweight alternative to the more verbose `function` keyword and syntax.

Most people tend to *ooh and aah* at nice terse examples like that, as I imagine you just did!

However, I would caution you that it would seem to me somewhat a misapplication of this feature to use arrow function syntax with otherwise normal, multistatement functions, especially those that would otherwise be naturally expressed as function declarations.

Recall the `dollabillsyall(..)` string literal tag function from earlier in this chapter -- let's change it to use `=>` syntax:

```
var dollabillsyall = (strings, ...values) =>
  strings.reduce( (s,v,idx) => {
    if (idx > 0) {
      if (typeof values[idx-1] == "number") {
        // look, also using interpolated
        // string literals!
        s += `$$\{values[idx-1].toFixed( 2 )}\`;
      }
      else {
        s += values[idx-1];
      }
    }

    return s + v;
  }, "" );
```

In this example, the only modifications I made were the removal of `function`, `return`, and some `{ .. }`, and then the insertion of `=>` and a `var`. Is this a significant improvement in the readability of the code? Meh.

I'd actually argue that the lack of `return` and outer `{ .. }` partially obscures the fact that the `reduce(..)` call is the only statement in the `dollabillsyall(..)` function and that its result is the intended result of the call. Also, the trained eye that is so used to hunting for the

word `function` in code to find scope boundaries now needs to look for the `=>` marker, which can definitely be harder to find in the thick of the code.

While not a hard-and-fast rule, I'd say that the readability gains from `=>` arrow function conversion are inversely proportional to the length of the function being converted. The longer the function, the less `=>` helps; the shorter the function, the more `=>` can shine.

I think it's probably more sensible and reasonable to adopt `=>` for the places in code where you do need short inline function expressions, but leave your normal-length main functions as is.

Not Just Shorter Syntax, But `this`

Most of the popular attention toward `=>` has been on saving those precious keystrokes by dropping `function`, `return`, and `{ ... }` from your code.

But there's a big detail we've skipped over so far. I said at the beginning of the section that `=>` functions are closely related to `this` binding behavior. In fact, `=>` arrow functions are *primarily designed* to alter `this` behavior in a specific way, solving a particular and common pain point with `this`-aware coding.

The saving of keystrokes is a red herring, a misleading sideshow at best.

Let's revisit another example from earlier in this chapter:

```
var controller = {
  makeRequest: function(...){
    var self = this;

    btn.addEventListener( "click", function(){
      // ...
      self.makeRequest(...);
    }, false );
  }
};
```

We used the `var self = this` hack, and then referenced `self.makeRequest(..)`, because inside the callback function we're passing to `addEventListener(..)`, the `this` binding will not be the same as it is in `makeRequest(..)` itself. In other words, because `this` bindings are dynamic, we fall back to the predictability of lexical scope via the `self` variable.

Herein we finally can see the primary design characteristic of `=>` arrow functions. Inside arrow functions, the `this` binding is not dynamic, but is instead lexical. In the previous snippet, if we used an arrow function for the callback, `this` will be predictably what we wanted it to be.

Consider:

```
var controller = {
  makeRequest: function(...){
    btn.addEventListener( "click", () => {
      // ...
      this.makeRequest(...);
    }, false );
  }
};
```

Lexical `this` in the arrow function callback in the previous snippet now points to the same value as in the enclosing `makeRequest(...)` function. In other words, `=>` is a syntactic stand-in for `var self = this`.

In cases where `var self = this` (or, alternatively, a function `.bind(this)` call) would normally be helpful, `=>` arrow functions are a nicer alternative operating on the same principle. Sounds great, right?

Not quite so simple.

If `=>` replaces `var self = this` or `.bind(this)` and it helps, guess what happens if you use `=>` with a `this`-aware function that *doesn't* need `var self = this` to work? You might be able to guess that it's going to mess things up. Yeah.

Consider:

```
var controller = {
  makeRequest: (... )=> {
    // ...
    this.helper(...);
  },
  helper: (... )=> {
    // ...
  }
};

controller.makeRequest(...);
```

Although we invoke as `controller.makeRequest(...)`, the `this.helper` reference fails, because `this` here doesn't point to `controller` as it normally would. Where does it point? It lexically inherits `this` from the surrounding scope. In this previous snippet, that's the global scope, where `this` points to the global object. Ugh.

In addition to lexical `this`, arrow functions also have lexical `arguments` -- they don't have their own `arguments` array but instead inherit from their parent -- as well as lexical `super` and `new.target` (see "Classes" in Chapter 3).

So now we can conclude a more nuanced set of rules for when `=>` is appropriate and not:

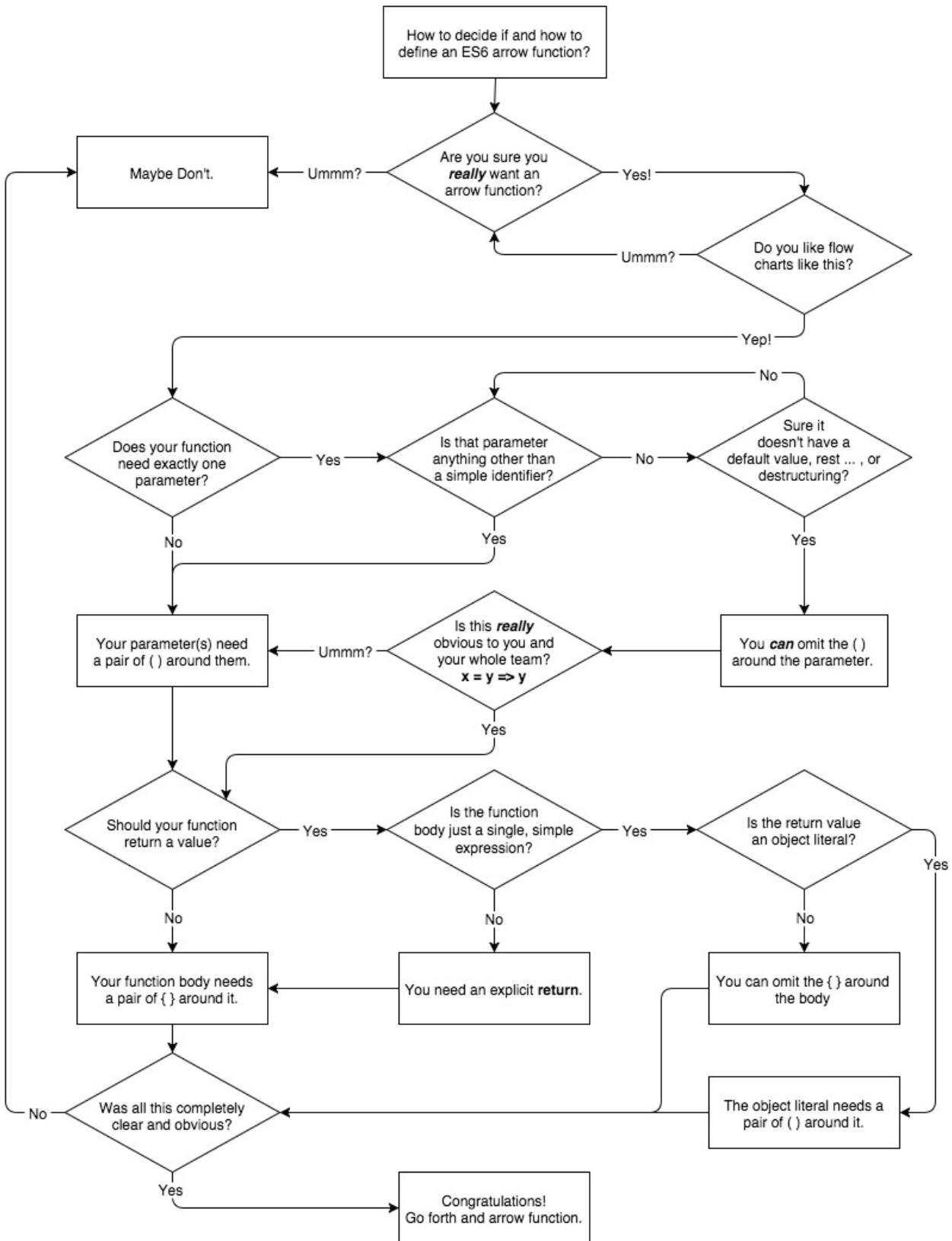
- If you have a short, single-statement inline function expression, where the only statement is a `return` of some computed value, *and* that function doesn't already make a `this` reference inside it, *and* there's no self-reference (recursion, event binding/unbinding), *and* you don't reasonably expect the function to ever be that way, you can probably safely refactor it to be an `=>` arrow function.
- If you have an inner function expression that's relying on a `var self = this` hack or a `.bind(this)` call on it in the enclosing function to ensure proper `this` binding, that inner function expression can probably safely become an `=>` arrow function.
- If you have an inner function expression that's relying on something like `var args = Array.prototype.slice.call(arguments)` in the enclosing function to make a lexical copy of `arguments`, that inner function expression can probably safely become an `=>` arrow function.
- For everything else -- normal function declarations, longer multistatement function expressions, functions that need a lexical name identifier self-reference (recursion, etc.), and any other function that doesn't fit the previous characteristics -- you should probably avoid `=>` function syntax.

Bottom line: `=>` is about lexical binding of `this`, `arguments`, and `super`. These are intentional features designed to fix some common problems, not bugs, quirks, or mistakes in ES6.

Don't believe any hype that `=>` is primarily, or even mostly, about fewer keystrokes. Whether you save keystrokes or waste them, you should know exactly what you are intentionally doing with every character typed.

Tip: If you have a function that for any of these articulated reasons is not a good match for an `=>` arrow function, but it's being declared as part of an object literal, recall from "Concise Methods" earlier in this chapter that there's another option for shorter function syntax.

If you prefer a visual decision chart for how/why to pick an arrow function:



for..of Loops

Joining the `for` and `for..in` loops from the JavaScript we're all familiar with, ES6 adds a `for..of` loop, which loops over the set of values produced by an *iterator*.

The value you loop over with `for..of` must be an *iterable*, or it must be a value which can be coerced/boxed to an object (see the *Types & Grammar* title of this series) that is an iterable. An iterable is simply an object that is able to produce an iterator, which the loop then uses.

Let's compare `for..of` to `for..in` to illustrate the difference:

```
var a = ["a", "b", "c", "d", "e"];

for (var idx in a) {
    console.log( idx );
}

// 0 1 2 3 4

for (var val of a) {
    console.log( val );
}

// "a" "b" "c" "d" "e"
```

As you can see, `for..in` loops over the keys/indexes in the `a` array, while `for..of` loops over the values in `a`.

Here's the pre-ES6 version of the `for..of` from that previous snippet:

```
var a = ["a", "b", "c", "d", "e"],
    k = Object.keys( a );

for (var val, i = 0; i < k.length; i++) {
    val = a[ k[i] ];
    console.log( val );
}

// "a" "b" "c" "d" "e"
```

And here's the ES6 but non-`for..of` equivalent, which also gives a glimpse at manually iterating an iterator (see "Iterators" in Chapter 3):

```
var a = ["a", "b", "c", "d", "e"];

for (var val, ret, it = a[Symbol.iterator]();
     (ret = it.next()) && !ret.done;
) {
    val = ret.value;
    console.log( val );
}

// "a" "b" "c" "d" "e"
```

Under the covers, the `for..of` loop asks the iterable for an iterator (using the built-in `Symbol.iterator`; see "Well-Known Symbols" in Chapter 7), then it repeatedly calls the iterator and assigns its produced value to the loop iteration variable.

Standard built-in values in JavaScript that are by default iterables (or provide them) include:

- Arrays
- Strings
- Generators (see Chapter 3)
- Collections / TypedArrays (see Chapter 5)

Warning: Plain objects are not by default suitable for `for..of` looping. That's because they don't have a default iterator, which is intentional, not a mistake. However, we won't go any further into those nuanced reasonings here. In "Iterators" in Chapter 3, we'll see how to define iterators for our own objects, which lets `for..of` loop over any object to get a set of values we define.

Here's how to loop over the characters in a primitive string:

```
for (var c of "hello") {
    console.log( c );
}
// "h" "e" "l" "l" "o"
```

The `"hello"` primitive string value is coerced/boxed to the `String` object wrapper equivalent, which is an iterable by default.

In `for (XYZ of ABC)...`, the `XYZ` clause can either be an assignment expression or a declaration, identical to that same clause in `for` and `for..in` loops. So you can do stuff like this:

```
var o = {};

for (o.a of [1,2,3]) {
    console.log( o.a );
}
// 1 2 3

for ({x: o.a} of [ {x: 1}, {x: 2}, {x: 3} ]) {
    console.log( o.a );
}
// 1 2 3
```

`for..of` loops can be prematurely stopped, just like other loops, with `break`, `continue`, `return` (if in a function), and thrown exceptions. In any of these cases, the iterator's `return(..)` function is automatically called (if one exists) to let the iterator perform cleanup tasks, if necessary.

Note: See "Iterators" in Chapter 3 for more complete coverage on iterables and iterators.

Regular Expressions

Let's face it: regular expressions haven't changed much in JS in a long time. So it's a great thing that they've finally learned a couple of new tricks in ES6. We'll briefly cover the additions here, but the overall topic of regular expressions is so dense that you'll need to turn to chapters/books dedicated to it (of which there are many!) if you need a refresher.

Unicode Flag

We'll cover the topic of Unicode in more detail in "Unicode" later in this chapter. Here, we'll just look briefly at the new `u` flag for ES6+ regular expressions, which turns on Unicode matching for that expression.

JavaScript strings are typically interpreted as sequences of 16-bit characters, which correspond to the characters in the *Basic Multilingual Plane (BMP)* (http://en.wikipedia.org/wiki/Plane_%28Unicode%29). But there are many UTF-16 characters that fall outside this range, and so strings may have these multibyte characters in them.

Prior to ES6, regular expressions could only match based on BMP characters, which means that those extended characters were treated as two separate characters for matching purposes. This is often not ideal.

So, as of ES6, the `u` flag tells a regular expression to process a string with the interpretation of Unicode (UTF-16) characters, such that such an extended character will be matched as a single entity.

Warning: Despite the name implication, "UTF-16" doesn't strictly mean 16 bits. Modern Unicode uses 21 bits, and standards like UTF-8 and UTF-16 refer roughly to how many bits are used in the representation of a character.

An example (straight from the ES6 specification): (the musical symbol G-clef) is Unicode point U+1D11E (0x1D11E).

If this character appears in a regular expression pattern (like `//`), the standard BMP interpretation would be that it's two separate characters (0xD834 and 0xDD1E) to match with. But the new ES6 Unicode-aware mode means that `//u` (or the escaped Unicode form `/\u{1D11E}/u`) will match `""` in a string as a single matched character.

You might be wondering why this matters? In non-Unicode BMP mode, the pattern is treated as two separate characters, but would still find the match in a string with the `""` character in it, as you can see if you try:

```
//.test( "-clef" );           // true
```

The length of the match is what matters. For example:

```
/^.-clef/.test( "-clef" );      // false
/^.-clef/u.test( "-clef" );    // true
```

The `^.-clef` in the pattern says to match only a single character at the beginning before the normal `"-clef"` text. In standard BMP mode, the match fails (two characters), but with `u` Unicode mode flagged on, the match succeeds (one character).

It's also important to note that `u` makes quantifiers like `+` and `*` apply to the entire Unicode code point as a single character, not just the *lower surrogate* (aka rightmost half of the symbol) of the character. The same goes for Unicode characters appearing in character classes, like `/[-]/u`.

Note: There's plenty more nitty-gritty details about `u` behavior in regular expressions, which Mathias Bynens (<https://twitter.com/mathias>) has written extensively about (<https://mathiasbynens.be/notes/es6-unicode-regex>).

Sticky Flag

Another flag mode added to ES6 regular expressions is `y`, which is often called "sticky mode." *Sticky* essentially means the regular expression has a virtual anchor at its beginning that keeps it rooted to matching at only the position indicated by the regular expression's `lastIndex` property.

To illustrate, let's consider two regular expressions, the first without sticky mode and the second with:

```

var re1 = /foo/,
    str = "++foo++";

re1.lastIndex;          // 0
re1.test( str );       // true
re1.lastIndex;          // 0 -- not updated

re1.lastIndex = 4;
re1.test( str );       // true -- ignored `lastIndex`
re1.lastIndex;          // 4 -- not updated

```

Three things to observe about this snippet:

- `test(..)` doesn't pay any attention to `lastIndex`'s value, and always just performs its match from the beginning of the input string.
- Because our pattern does not have a `^` start-of-input anchor, the search for `"foo"` is free to move ahead through the whole string looking for a match.
- `lastIndex` is not updated by `test(..)`.

Now, let's try a sticky mode regular expression:

```

var re2 = /foo/y,           // <-- notice the `y` sticky flag
    str = "++foo++";

re2.lastIndex;          // 0
re2.test( str );       // false -- "foo" not found at `0`
re2.lastIndex;          // 0

re2.lastIndex = 2;
re2.test( str );       // true
re2.lastIndex;          // 5 -- updated to after previous match

re2.test( str );       // false
re2.lastIndex;          // 0 -- reset after previous match failure

```

And so our new observations about sticky mode:

- `test(..)` uses `lastIndex` as the exact and only position in `str` to look to make a match. There is no moving ahead to look for the match -- it's either there at the `lastIndex` position or not.
- If a match is made, `test(..)` updates `lastIndex` to point to the character immediately following the match. If a match fails, `test(..)` resets `lastIndex` back to `0`.

Normal non-sticky patterns that aren't otherwise `^`-rooted to the start-of-input are free to move ahead in the input string looking for a match. But sticky mode restricts the pattern to matching just at the position of `lastIndex`.

As I suggested at the beginning of this section, another way of looking at this is that `y` implies a virtual anchor at the beginning of the pattern that is relative (aka constrains the start of the match) to exactly the `lastIndex` position.

Warning: In previous literature on the topic, it has alternatively been asserted that this behavior is like `y` implying a `^` (start-of-input) anchor in the pattern. This is inaccurate. We'll explain in further detail in "Anchored Sticky" later.

Sticky Positioning

It may seem strangely limiting that to use `y` for repeated matches, you have to manually ensure `lastIndex` is in the exact right position, as it has no move-ahead capability for matching.

Here's one possible scenario: if you know that the match you care about is always going to be at a position that's a multiple of a number (e.g., `0`, `10`, `20`, etc.), you can just construct a limited pattern matching what you care about, but then manually set `lastIndex` each time before match to those fixed positions.

Consider:

```
var re = /f...y,
    str = "foo      far      fad";

str.match( re );          // ["foo"]

re.lastIndex = 10;
str.match( re );          // ["far"]

re.lastIndex = 20;
str.match( re );          // ["fad"]
```

However, if you're parsing a string that isn't formatted in fixed positions like that, figuring out what to set `lastIndex` to before each match is likely going to be untenable.

There's a saving nuance to consider here. `y` requires that `lastIndex` be in the exact position for a match to occur. But it doesn't strictly require that *you* manually set `lastIndex`.

Instead, you can construct your expressions in such a way that they capture in each main match everything before and after the thing you care about, up to right before the next thing you'll care to match.

Because `lastIndex` will set to the next character beyond the end of a match, if you've matched everything up to that point, `lastIndex` will always be in the correct position for the `y` pattern to start from the next time.

Warning: If you can't predict the structure of the input string in a sufficiently patterned way like that, this technique may not be suitable and you may not be able to use `y`.

Having structured string input is likely the most practical scenario where `y` will be capable of performing repeated matching throughout a string. Consider:

```
var re = /\d+\.\s(.*)\:(\s|$)/y
str = "1. foo 2. bar 3. baz";

str.match( re );           // [ "1. foo ", "foo" ]

re.lastIndex;             // 7 -- correct position!
str.match( re );           // [ "2. bar ", "bar" ]

re.lastIndex;             // 14 -- correct position!
str.match( re );           // [ "3. baz", "baz" ]
```

This works because I knew something ahead of time about the structure of the input string: there is always a numeral prefix like `"1. "` before the desired match (`"foo"`, etc.), and either a space after it, or the end of the string (`$` anchor). So the regular expression I constructed captures all of that in each main match, and then I use a matching group `()` so that the stuff I really care about is separated out for convenience.

After the first match (`"1. foo "`), the `lastIndex` is `7`, which is already the position needed to start the next match, for `"2. bar "`, and so on.

If you're going to use `y` sticky mode for repeated matches, you'll probably want to look for opportunities to have `lastIndex` automatically positioned as we've just demonstrated.

Sticky Versus Global

Some readers may be aware that you can emulate something like this `lastIndex`-relative matching with the `g` global match flag and the `exec(...)` method, as so:

```

var re = /o+./g,           // <-- look, `g`!
str = "foot book more";

re.exec( str );           // ["oot"]
re.lastIndex;             // 4

re.exec( str );           // ["ook"]
re.lastIndex;             // 9

re.exec( str );           // ["or"]
re.lastIndex;             // 13

re.exec( str );           // null -- no more matches!
re.lastIndex;             // 0 -- starts over now!

```

While it's true that `g` pattern matches with `exec(...)` start their matching from `lastIndex`'s current value, and also update `lastIndex` after each match (or failure), this is not the same thing as `y`'s behavior.

Notice in the previous snippet that `"ook"`, located at position `6`, was matched and found by the second `exec(...)` call, even though at the time, `lastIndex` was `4` (from the end of the previous match). Why? Because as we said earlier, non-sticky matches are free to move ahead in their matching. A sticky mode expression would have failed here, because it would not be allowed to move ahead.

In addition to perhaps undesired move-ahead matching behavior, another downside to just using `g` instead of `y` is that `g` changes the behavior of some matching methods, like `str.match(re)`.

Consider:

```

var re = /o+./g,           // <-- look, `g`!
str = "foot book more";

str.match( re );           // ["oot","ook","or"]

```

See how all the matches were returned at once? Sometimes that's OK, but sometimes that's not what you want.

The `y` sticky flag will give you one-at-a-time progressive matching with utilities like `test(...)` and `match(...)`. Just make sure the `lastIndex` is always in the right position for each match!

Anchored Sticky

As we warned earlier, it's inaccurate to think of sticky mode as implying a pattern starts with `^`. The `^` anchor has a distinct meaning in regular expressions, which is *not altered* by sticky mode. `^` is an anchor that *always* refers to the beginning of the input, and *is not* in any way relative to `lastIndex`.

Besides poor/inaccurate documentation on this topic, the confusion is unfortunately strengthened further because an older pre-ES6 experiment with sticky mode in Firefox *did* make `^` relative to `lastIndex`, so that behavior has been around for years.

ES6 elected not to do it that way. `^` in a pattern means start-of-input absolutely and only.

As a consequence, a pattern like `/^foo/y` will always and only find a "foo" match at the beginning of a string, *if it's allowed to match there*. If `lastIndex` is not `0`, the match will fail. Consider:

```
var re = /^foo/y,
    str = "foo";

re.test( str );           // true
re.test( str );           // false
re.lastIndex;             // 0 -- reset after failure

re.lastIndex = 1;
re.test( str );           // false -- failed for positioning
re.lastIndex;             // 0 -- reset after failure
```

Bottom line: `y` plus `^` plus `lastIndex > 0` is an incompatible combination that will always cause a failed match.

Note: While `y` does not alter the meaning of `^` in any way, the `m` multiline mode *does*, such that `^` means start-of-input *or* start of text after a newline. So, if you combine `y` and `m` flags together for a pattern, you can find multiple `^`-rooted matches in a string. But remember: because it's `y` sticky, you'll have to make sure `lastIndex` is pointing at the correct new line position (likely by matching to the end of the line) each subsequent time, or no subsequent matches will be made.

Regular Expression flags

Prior to ES6, if you wanted to examine a regular expression object to see what flags it had applied, you needed to parse them out -- ironically, probably with another regular expression -- from the content of the `source` property, such as:

```
var re = /foo/ig;

re.toString();           // "/foo/ig"

var flags = re.toString().match( /\(([gim]*$)/ )[1];

flags;                 // "ig"
```

As of ES6, you can now get these values directly, with the new `flags` property:

```
var re = /foo/ig;

re.flags;             // "gi"
```

It's a small nuance, but the ES6 specification calls for the expression's flags to be listed in this order: `"gimuy"`, regardless of what order the original pattern was specified with. That's the reason for the difference between `/ig` and `"gi"`.

No, the order of flags specified or listed doesn't matter.

Another tweak from ES6 is that the `RegExp(..)` constructor is now `flags`-aware if you pass it an existing regular expression:

```
var re1 = /foo*/y;
re1.source;           // "foo*"
re1.flags;            // "y"

var re2 = new RegExp( re1 );
re2.source;           // "foo*"
re2.flags;            // "y"

var re3 = new RegExp( re1, "ig" );
re3.source;           // "foo*"
re3.flags;            // "gi"
```

Prior to ES6, the `re3` construction would throw an error, but as of ES6 you can override the flags when duplicating.

Number Literal Extensions

Prior to ES5, number literals looked like the following -- the octal form was not officially specified, only allowed as an extension that browsers had come to de facto agreement on:

```
var dec = 42,
    oct = 052,
    hex = 0x2a;
```

Note: Though you are specifying a number in different bases, the number's mathematical value is what is stored, and the default output interpretation is always base-10. The three variables in the previous snippet all have the `42` value stored in them.

To further illustrate that `052` was a nonstandard form extension, consider:

```
Number( "42" );           // 42
Number( "052" );          // 52
Number( "0x2a" );         // 42
```

ES5 continued to permit the browser-extended octal form (including such inconsistencies), except that in strict mode, the octal literal (`052`) form is disallowed. This restriction was done mainly because many developers had the habit (from other languages) of seemingly innocuously prefixing otherwise base-10 numbers with `0`'s for code alignment purposes, and then running into the accidental fact that they'd changed the number value entirely!

ES6 continues the legacy of changes/variations to how number literals outside base-10 numbers can be represented. There's now an official octal form, an amended hexadecimal form, and a brand-new binary form. For web compatibility reasons, the old octal `052` form will continue to be legal (though unspecified) in non-strict mode, but should really never be used anymore.

Here are the new ES6 number literal forms:

```
var dec = 42,
    oct = 0o52,           // or `0052` :(
    hex = 0x2a,            // or `0X2a` :/
    bin = 0b101010;        // or `0B101010` :/
```

The only decimal form allowed is base-10. Octal, hexadecimal, and binary are all integer forms.

And the string representations of these forms are all able to be coerced/converted to their number equivalent:

```
Number( "42" );           // 42
Number( "0o52" );          // 42
Number( "0x2a" );          // 42
Number( "0b101010" );       // 42
```

Though not strictly new to ES6, it's a little-known fact that you can actually go the opposite direction of conversion (well, sort of):

```
var a = 42;

a.toString();           // "42" -- also `a.toString( 10 )`
a.toString( 8 );       // "52"
a.toString( 16 );      // "2a"
a.toString( 2 );       // "101010"
```

In fact, you can represent a number this way in any base from `2` to `36`, though it'd be rare that you'd go outside the standard bases: 2, 8, 10, and 16.

Unicode

Let me just say that this section is not an exhaustive everything-you-ever-wanted-to-know-about-Unicode resource. I want to cover what you need to know that's *changing* for Unicode in ES6, but we won't go much deeper than that. Mathias Bynens (<http://twitter.com/mathias>) has written/spoken extensively and brilliantly about JS and Unicode (see <https://mathiasbynens.be/notes/javascript-unicode> and <http://fluentconf.com/javascript-html-2015/public/content/2015/02/18-javascript-loves-unicode>).

The Unicode characters that range from `0x0000` to `0xFFFF` contain all the standard printed characters (in various languages) that you're likely to have seen or interacted with. This group of characters is called the *Basic Multilingual Plane (BMP)*. The BMP even contains fun symbols like this cool snowman: ☃ (U+2603).

There are lots of other extended Unicode characters beyond this BMP set, which range up to `0x10FFFF`. These symbols are often referred to as *astral* symbols, as that's the name given to the set of 16 *planes* (e.g., layers/groupings) of characters beyond the BMP. Examples of astral symbols include (U+1D11E) and (U+1F4A9).

Prior to ES6, JavaScript strings could specify Unicode characters using Unicode escaping, such as:

```
var snowman = "\u2603";
console.log( snowman );           // ☃"
```

However, the `\uxxxx` Unicode escaping only supports four hexadecimal characters, so you can only represent the BMP set of characters in this way. To represent an astral character using Unicode escaping prior to ES6, you need to use a *surrogate pair* -- basically two

specially calculated Unicode-escaped characters side by side, which JS interprets together as a single astral character:

```
var gclef = "\uD834\uDD1E";
console.log( gclef );           // ""
```

As of ES6, we now have a new form for Unicode escaping (in strings and regular expressions), called Unicode *code point escaping*:

```
var gclef = "\u{1D11E}";
console.log( gclef );           // ""
```

As you can see, the difference is the presence of the `{ }` in the escape sequence, which allows it to contain any number of hexadecimal characters. Because you only need six to represent the highest possible code point value in Unicode (i.e., 0x10FFFF), this is sufficient.

Unicode-Aware String Operations

By default, JavaScript string operations and methods are not sensitive to astral symbols in string values. So, they treat each BMP character individually, even the two surrogate halves that make up an otherwise single astral character. Consider:

```
var snowman = "\u2603";
snowman.length;                  // 1

var gclef = "\u{1D11E}";
gclef.length;                   // 2
```

So, how do we accurately calculate the length of such a string? In this scenario, the following trick will work:

```
var gclef = "\u{1D11E}";
[...gclef].length;               // 1
Array.from( gclef ).length;     // 1
```

Recall from the " `for..of` Loops" section earlier in this chapter that ES6 strings have built-in iterators. This iterator happens to be Unicode-aware, meaning it will automatically output an astral symbol as a single value. We take advantage of that using the `...` spread operator in an array literal, which creates an array of the string's symbols. Then we just inspect the length of that resultant array. ES6's `Array.from(..)` does basically the same thing as `[...xyz]`, but we'll cover that utility in detail in Chapter 6.

Warning: It should be noted that constructing and exhausting an iterator just to get the length of a string is quite expensive on performance, relatively speaking, compared to what a theoretically optimized native utility/property would do.

Unfortunately, the full answer is not as simple or straightforward. In addition to the surrogate pairs (which the string iterator takes care of), there are special Unicode code points that behave in other special ways, which is much harder to account for. For example, there's a set of code points that modify the previous adjacent character, known as *Combining Diacritical Marks*.

Consider these two string outputs:

```
console.log( s1 );           // "é"
console.log( s2 );           // "é"
```

They look the same, but they're not! Here's how we created `s1` and `s2`:

```
var s1 = "\xE9",
    s2 = "e\u0301";
```

As you can probably guess, our previous `length` trick doesn't work with `s2`:

```
[...s1].length;           // 1
[...s2].length;           // 2
```

So what can we do? In this case, we can perform a *Unicode normalization* on the value before inquiring about its length, using the ES6 `String#normalize(..)` utility (which we'll cover more in Chapter 6):

```
var s1 = "\xE9",
    s2 = "e\u0301";

s1.normalize().length;      // 1
s2.normalize().length;      // 1

s1 === s2;                 // false
s1 === s2.normalize();     // true
```

Essentially, `normalize(..)` takes a sequence like `"e\u0301"` and normalizes it to `"\xE9"`. Normalization can even combine multiple adjacent combining marks if there's a suitable Unicode character they combine to:

```

var s1 = "o\u0302\u0300",
    s2 = s1.normalize(),
    s3 = "\u0300"; // ó

s1.length;           // 3
s2.length;           // 1
s3.length;           // 1

s2 === s3;           // true

```

Unfortunately, normalization isn't fully perfect here, either. If you have multiple combining marks modifying a single character, you may not get the length count you'd expect, because there may not be a single defined normalized character that represents the combination of all the marks. For example:

```

var s1 = "e\u0301\u0330"; // é

console.log( s1 ); // "é"

s1.normalize().length; // 2

```

The further you go down this rabbit hole, the more you realize that it's difficult to get one precise definition for "length." What we see visually rendered as a single character -- more precisely called a *grapheme* -- doesn't always strictly relate to a single "character" in the program processing sense.

Tip: If you want to see just how deep this rabbit hole goes, check out the "Grapheme Cluster Boundaries" algorithm (http://www.Unicode.org/reports/tr29/#Grapheme_Cluster_Boundaries).

Character Positioning

Similar to length complications, what does it actually mean to ask, "what is the character at position 2?" The naive pre-ES6 answer comes from `charAt(..)`, which will not respect the atomicity of an astral character, nor will it take into account combining marks.

Consider:

```

var s1 = "abc\u0301d",
    s2 = "ab\u0107d",
    s3 = "ab\u{1d49e}d";

console.log( s1 );           // "abćd"
console.log( s2 );           // "abćd"
console.log( s3 );           // "abd"

s1.charCodeAt( 2 );          // "c"
s2.charCodeAt( 2 );          // "ć"
s3.charCodeAt( 2 );          // "" <-- unprintable surrogate
s3.charCodeAt( 3 );          // "" <-- unprintable surrogate

```

So, is ES6 giving us a Unicode-aware version of `charAt(..)`? Unfortunately, no. At the time of this writing, there's a proposal for such a utility that's under consideration for post-ES6.

But with what we explored in the previous section (and of course with the limitations noted thereof!), we can hack an ES6 answer:

```

var s1 = "abc\u0301d",
    s2 = "ab\u0107d",
    s3 = "ab\u{1d49e}d";

[...s1.normalize()][2];        // "ć"
[...s2.normalize()][2];        // "ć"
[...s3.normalize()][2];        // ""

```

Warning: Reminder of an earlier warning: constructing and exhausting an iterator each time you want to get at a single character is... not very ideal, performance wise. Let's hope we get a built-in and optimized utility for this soon, post-ES6.

What about a Unicode-aware version of the `charCodeAt(..)` utility? ES6 gives us

`codePointAt(..)`:

```

var s1 = "abc\u0301d",
    s2 = "ab\u0107d",
    s3 = "ab\u{1d49e}d";

s1.normalize().codePointAt( 2 ).toString( 16 );
// "107"

s2.normalize().codePointAt( 2 ).toString( 16 );
// "107"

s3.normalize().codePointAt( 2 ).toString( 16 );
// "1d49e"

```

What about the other direction? A Unicode-aware version of `String.fromCharCode(..)` is ES6's `String.fromCodePoint(..)`:

```
String.fromCodePoint( 0x107 );           // "ć"
String.fromCodePoint( 0x1d49e );         // ""
```

So wait, can we just combine `String.fromCodePoint(..)` and `codePointAt(..)` to get a better version of a Unicode-aware `charAt(..)` from earlier? Yep!

```
var s1 = "abc\u0301d",
    s2 = "ab\u0107d",
    s3 = "ab\u{1d49e}d";

String.fromCodePoint( s1.normalize().codePointAt( 2 ) );
// "ć"

String.fromCodePoint( s2.normalize().codePointAt( 2 ) );
// "ć"

String.fromCodePoint( s3.normalize().codePointAt( 2 ) );
// ""
```

There's quite a few other string methods we haven't addressed here, including `toUpperCase()`, `toLowerCase()`, `substring(..)`, `indexOf(..)`, `slice(..)`, and a dozen others. None of these have been changed or augmented for full Unicode awareness, so you should be very careful -- probably just avoid them! -- when working with strings containing astral symbols.

There are also several string methods that use regular expressions for their behavior, like `replace(..)` and `match(..)`. Thankfully, ES6 brings Unicode awareness to regular expressions, as we covered in "Unicode Flag" earlier in this chapter.

OK, there we have it! JavaScript's Unicode string support is significantly better over pre-ES6 (though still not perfect) with the various additions we've just covered.

Unicode Identifier Names

Unicode can also be used in identifier names (variables, properties, etc.). Prior to ES6, you could do this with Unicode-escapes, like:

```
var \u03A9 = 42;
// same as: var Ω = 42;
```

As of ES6, you can also use the earlier explained code point escape syntax:

```
var \u{2B400} = 42;  
  
// same as: var = 42;
```

There's a complex set of rules around exactly which Unicode characters are allowed. Furthermore, some are allowed only if they're not the first character of the identifier name.

Note: Mathias Bynens has a great post (<https://mathiasbynens.be/notes/javascript-identifiers-es6>) on all the nitty-gritty details.

The reasons for using such unusual characters in identifier names are rather rare and academic. You typically won't be best served by writing code that relies on these esoteric capabilities.

Symbols

With ES6, for the first time in quite a while, a new primitive type has been added to JavaScript: the `symbol`. Unlike the other primitive types, however, symbols don't have a literal form.

Here's how you create a symbol:

```
var sym = Symbol( "some optional description" );  
  
typeof sym;           // "symbol"
```

Some things to note:

- You cannot and should not use `new` with `Symbol(...)`. It's not a constructor, nor are you producing an object.
- The parameter passed to `Symbol(...)` is optional. If passed, it should be a string that gives a friendly description for the symbol's purpose.
- The `typeof` output is a new value (`"symbol"`) that is the primary way to identify a symbol.

The description, if provided, is solely used for the stringification representation of the symbol:

```
sym.toString();        // "Symbol(some optional description)"
```

Similar to how primitive string values are not instances of `String`, symbols are also not instances of `Symbol`. If, for some reason, you want to construct a boxed wrapper object form of a symbol value, you can do the following:

```
sym instanceof Symbol;           // false

var symObj = Object( sym );
symObj instanceof Symbol;       // true

symObj.valueOf() === sym;       // true
```

Note: `symObj` in this snippet is interchangeable with `sym`; either form can be used in all places symbols are utilized. There's not much reason to use the boxed wrapper object form (`symObj`) instead of the primitive form (`sym`). Keeping with similar advice for other primitives, it's probably best to prefer `sym` over `symObj`.

The internal value of a symbol itself -- referred to as its `name` -- is hidden from the code and cannot be obtained. You can think of this symbol value as an automatically generated, unique (within your application) string value.

But if the value is hidden and unobtainable, what's the point of having a symbol at all?

The main point of a symbol is to create a string-like value that can't collide with any other value. So, for example, consider using a symbol as a constant representing an event name:

```
const EVT_LOGIN = Symbol( "event.login" );
```

You'd then use `EVT_LOGIN` in place of a generic string literal like `"event.login"`:

```
evthub.listen( EVT_LOGIN, function(data){
  // ...
});
```

The benefit here is that `EVT_LOGIN` holds a value that cannot be duplicated (accidentally or otherwise) by any other value, so it is impossible for there to be any confusion of which event is being dispatched or handled.

Note: Under the covers, the `evthub` utility assumed in the previous snippet would almost certainly be using the symbol value from the `EVT_LOGIN` argument directly as the property/key in some internal object (hash) that tracks event handlers. If `evthub` instead needed to use the symbol value as a real string, it would need to explicitly coerce with `String(...)` or `toString()`, as implicit string coercion of symbols is not allowed.

You may use a symbol directly as a property name/key in an object, such as a special property that you want to treat as hidden or meta in usage. It's important to know that although you intend to treat it as such, it is not *actually* a hidden or untouchable property.

Consider this module that implements the *singleton* pattern behavior -- that is, it only allows itself to be created once:

```
const INSTANCE = Symbol( "instance" );

function HappyFace() {
  if (HappyFace[INSTANCE]) return HappyFace[INSTANCE];

  function smile() { ... }

  return HappyFace[INSTANCE] = {
    smile: smile
  };
}

var me = HappyFace(),
  you = HappyFace();

me === you;           // true
```

The `INSTANCE` symbol value here is a special, almost hidden, meta-like property stored statically on the `HappyFace()` function object.

It could alternatively have been a plain old property like `__instance`, and the behavior would have been identical. The usage of a symbol simply improves the metaprogramming style, keeping this `INSTANCE` property set apart from any other normal properties.

Symbol Registry

One mild downside to using symbols as in the last few examples is that the `EVT_LOGIN` and `INSTANCE` variables had to be stored in an outer scope (perhaps even the global scope), or otherwise somehow stored in a publicly available location, so that all parts of the code that need to use the symbols can access them.

To aid in organizing code with access to these symbols, you can create symbol values with the *global symbol registry*. For example:

```
const EVT_LOGIN = Symbol.for( "event.login" );

console.log( EVT_LOGIN );           // Symbol(event.login)
```

And:

```
function HappyFace() {
  const INSTANCE = Symbol.for( "instance" );

  if (HappyFace[INSTANCE]) return HappyFace[INSTANCE];

  // ...

  return HappyFace[INSTANCE] = { ... };
}
```

`Symbol.for(...)` looks in the global symbol registry to see if a symbol is already stored with the provided description text, and returns it if so. If not, it creates one to return. In other words, the global symbol registry treats symbol values, by description text, as singletons themselves.

But that also means that any part of your application can retrieve the symbol from the registry using `Symbol.for(...)`, as long as the matching description name is used.

Ironically, symbols are basically intended to replace the use of *magic strings* (arbitrary string values given special meaning) in your application. But you precisely use *magic* description string values to uniquely identify/locate them in the global symbol registry!

To avoid accidental collisions, you'll probably want to make your symbol descriptions quite unique. One easy way of doing that is to include prefix/context/namespacing information in them.

For example, consider a utility such as the following:

```
function extractValues(str) {
  var key = Symbol.for( "extractValues.parse" ),
    re = extractValues[key] ||
        /[^\&]+?=([^\&]+?)(?:=&|$)/g,
    values = [], match;

  while (match = re.exec( str )) {
    values.push( match[1] );
  }

  return values;
}
```

We use the magic string value `"extractValues.parse"` because it's quite unlikely that any other symbol in the registry would ever collide with that description.

If a user of this utility wants to override the parsing regular expression, they can also use the symbol registry:

```
extractValues[Symbol.for( "extractValues.parse" )] =
  /..some pattern../g;

extractValues( "..some string.." );
```

Aside from the assistance the symbol registry provides in globally storing these values, everything we're seeing here could have been done by just actually using the magic string "extractValues.parse" as the key, rather than the symbol. The improvements exist at the metaprogramming level more than the functional level.

You may have occasion to use a symbol value that has been stored in the registry to look up what description text (key) it's stored under. For example, you may need to signal to another part of your application how to locate a symbol in the registry because you cannot pass the symbol value itself.

You can retrieve a registered symbol's description text (key) using `Symbol.keyFor(..)`:

```
var s = Symbol.for( "something cool" );

var desc = Symbol.keyFor( s );
console.log( desc );           // "something cool"

// get the symbol from the registry again
var s2 = Symbol.for( desc );

s2 === s;                     // true
```

Symbols as Object Properties

If a symbol is used as a property/key of an object, it's stored in a special way so that the property will not show up in a normal enumeration of the object's properties:

```
var o = {
  foo: 42,
  [ Symbol( "bar" ) ]: "hello world",
  baz: true
};

Object.getOwnPropertyNames( o );    // [ "foo", "baz" ]
```

To retrieve an object's symbol properties:

```
Object.getOwnPropertySymbols( o ); // [ Symbol(bar) ]
```

This makes it clear that a property symbol is not actually hidden or inaccessible, as you can always see it in the `Object.getOwnPropertySymbols(..)` list.

Built-In Symbols

ES6 comes with a number of predefined built-in symbols that expose various meta behaviors on JavaScript object values. However, these symbols are *not* registered in the global symbol registry, as one might expect.

Instead, they're stored as properties on the `Symbol` function object. For example, in the "for..of" section earlier in this chapter, we introduced the `Symbol.iterator` value:

```
var a = [1,2,3];  
  
a[Symbol.iterator]; // native function
```

The specification uses the `@@` prefix notation to refer to the built-in symbols, the most common ones being: `@@iterator`, `@@toStringTag`, `@@toPrimitive`. Several others are defined as well, though they probably won't be used as often.

Note: See "Well Known Symbols" in Chapter 7 for detailed information about how these built-in symbols are used for meta programming purposes.

Review

ES6 adds a heap of new syntax forms to JavaScript, so there's plenty to learn!

Most of these are designed to ease the pain points of common programming idioms, such as setting default values to function parameters and gathering the "rest" of the parameters into an array. Destructuring is a powerful tool for more concisely expressing assignments of values from arrays and nested objects.

While features like `=>` arrow functions appear to also be all about shorter and nicer-looking syntax, they actually have very specific behaviors that you should intentionally use only in appropriate situations.

Expanded Unicode support, new tricks for regular expressions, and even a new primitive `symbol` type round out the syntactic evolution of ES6.

Chapter 3: Organization

It's one thing to write JS code, but it's another to properly organize it. Utilizing common patterns for organization and reuse goes a long way to improving the readability and understandability of your code. Remember: code is at least as much about communicating to other developers as it is about feeding the computer instructions.

ES6 has several important features that help significantly improve these patterns, including: iterators, generators, modules, and classes.

Iterators

An *iterator* is a structured pattern for pulling information from a source in one-at-a-time fashion. This pattern has been around programming for a long time. And to be sure, JS developers have been ad hoc designing and implementing iterators in JS programs since before anyone can remember, so it's not at all a new topic.

What ES6 has done is introduce an implicit standardized interface for iterators. Many of the built-in data structures in JavaScript will now expose an iterator implementing this standard. And you can also construct your own iterators adhering to the same standard, for maximal interoperability.

Iterators are a way of organizing ordered, sequential, pull-based consumption of data.

For example, you may implement a utility that produces a new unique identifier each time it's requested. Or you may produce an infinite series of values that rotate through a fixed list, in round-robin fashion. Or you could attach an iterator to a database query result to pull out new rows one at a time.

Although they have not commonly been used in JS in such a manner, iterators can also be thought of as controlling behavior one step at a time. This can be illustrated quite clearly when considering generators (see "Generators" later in this chapter), though you can certainly do the same without generators.

Interfaces

At the time of this writing, ES6 section 25.1.1.2 (<https://people.mozilla.org/~jorendorff/es6-draft.html#sec-iterator-interface>) details the `Iterator` interface as having the following requirement:

```
Iterator [required]
next() {method}: retrieves next IteratorResult
```

There are two optional members that some iterators are extended with:

```
Iterator [optional]
return() {method}: stops iterator and returns IteratorResult
throw() {method}: signals error and returns IteratorResult
```

The `IteratorResult` interface is specified as:

```
IteratorResult
value {property}: current iteration value or final return value
(optional if `undefined`)
done {property}: boolean, indicates completion status
```

Note: I call these interfaces implicit not because they're not explicitly called out in the specification -- they are! -- but because they're not exposed as direct objects accessible to code. JavaScript does not, in ES6, support any notion of "interfaces," so adherence for your own code is purely conventional. However, wherever JS expects an iterator -- a `for..of` loop, for instance -- what you provide must adhere to these interfaces or the code will fail.

There's also an `Iterable` interface, which describes objects that must be able to produce iterators:

```
Iterable
@@iterator() {method}: produces an Iterator
```

If you recall from "Built-In Symbols" in Chapter 2, `@@iterator` is the special built-in symbol representing the method that can produce iterator(s) for the object.

IteratorResult

The `IteratorResult` interface specifies that the return value from any iterator operation will be an object of the form:

```
{ value: ... , done: true / false }
```

Built-in iterators will always return values of this form, but more properties are, of course, allowed to be present on the return value, as necessary.

For example, a custom iterator may add additional metadata to the result object (e.g., where the data came from, how long it took to retrieve, cache expiration length, frequency for the appropriate next request, etc.).

Note: Technically, `value` is optional if it would otherwise be considered absent or unset, such as in the case of the value `undefined`. Because accessing `res.value` will produce `undefined` whether it's present with that value or absent entirely, the presence/absence of the property is more an implementation detail or an optimization (or both), rather than a functional issue.

next() Iteration

Let's look at an array, which is an iterable, and the iterator it can produce to consume its values:

```
var arr = [1, 2, 3];

var it = arr[Symbol.iterator]();

it.next();          // { value: 1, done: false }
it.next();          // { value: 2, done: false }
it.next();          // { value: 3, done: false }

it.next();          // { value: undefined, done: true }
```

Each time the method located at `Symbol.iterator` (see Chapters 2 and 7) is invoked on this `arr` value, it will produce a new fresh iterator. Most structures will do the same, including all the built-in data structures in JS.

However, a structure like an event queue consumer might only ever produce a single iterator (singleton pattern). Or a structure might only allow one unique iterator at a time, requiring the current one to be completed before a new one can be created.

The `it` iterator in the previous snippet doesn't report `done: true` when you receive the `3` value. You have to call `next()` again, in essence going beyond the end of the array's values, to get the complete signal `done: true`. It may not be clear why until later in this section, but that design decision will typically be considered a best practice.

Primitive string values are also iterables by default:

```
var greeting = "hello world";

var it = greeting[Symbol.iterator]();

it.next();           // { value: "h", done: false }
it.next();           // { value: "e", done: false }
..
..
```

Note: Technically, the primitive value itself isn't iterable, but thanks to "boxing", `"hello world"` is coerced/converted to its `String` object wrapper form, which *is* an iterable. See the *Types & Grammar* title of this series for more information.

ES6 also includes several new data structures, called collections (see Chapter 5). These collections are not only iterables themselves, but they also provide API method(s) to generate an iterator, such as:

```
var m = new Map();
m.set( "foo", 42 );
m.set( { cool: true }, "hello world" );

var it1 = m[Symbol.iterator]();
var it2 = m.entries();

it1.next();           // { value: [ "foo", 42 ], done: false }
it2.next();           // { value: [ "foo", 42 ], done: false }
..
..
```

The `next(...)` method of an iterator can optionally take one or more arguments. The built-in iterators mostly do not exercise this capability, though a generator's iterator definitely does (see "Generators" later in this chapter).

By general convention, including all the built-in iterators, calling `next(...)` on an iterator that's already been exhausted is not an error, but will simply continue to return the result `{ value: undefined, done: true }`.

Optional: `return(...)` and `throw(...)`

The optional methods on the iterator interface -- `return(...)` and `throw(...)` -- are not implemented on most of the built-in iterators. However, they definitely do mean something in the context of generators, so see "Generators" for more specific information.

`return(...)` is defined as sending a signal to an iterator that the consuming code is complete and will not be pulling any more values from it. This signal can be used to notify the producer (the iterator responding to `next(...)` calls) to perform any cleanup it may need

to do, such as releasing/closing network, database, or file handle resources.

If an iterator has a `return(...)` present and any condition occurs that can automatically be interpreted as abnormal or early termination of consuming the iterator, `return(...)` will automatically be called. You can call `return(...)` manually as well.

`return(...)` will return an `IteratorResult` object just like `next(...)` does. In general, the optional value you send to `return(...)` would be sent back as `value` in this `IteratorResult`, though there are nuanced cases where that might not be true.

`throw(...)` is used to signal an exception/error to an iterator, which possibly may be used differently by the iterator than the completion signal implied by `return(...)`. It does not necessarily imply a complete stop of the iterator as `return(...)` generally does.

For example, with generator iterators, `throw(...)` actually injects a thrown exception into the generator's paused execution context, which can be caught with a `try..catch`. An uncaught `throw(...)` exception would end up abnormally aborting the generator's iterator.

Note: By general convention, an iterator should not produce any more results after having called `return(...)` or `throw(...)`.

Iterator Loop

As we covered in the " `for..of` " section in Chapter 2, the ES6 `for..of` loop directly consumes a conforming iterable.

If an iterator is also an iterable, it can be used directly with the `for..of` loop. You make an iterator an iterable by giving it a `Symbol.iterator` method that simply returns the iterator itself:

```
var it = {
  // make the `it` iterator an iterable
  [Symbol.iterator]() { return this; },

  next() { .. },
  ..

};

it[Symbol.iterator]() === it;           // true
```

Now we can consume the `it` iterator with a `for..of` loop:

```
for (var v of it) {
  console.log( v );
}
```

To fully understand how such a loop works, recall the `for` equivalent of a `for..of` loop from Chapter 2:

```
for (var v, res; (res = it.next()) && !res.done; ) {  
    v = res.value;  
    console.log( v );  
}
```

If you look closely, you'll see that `it.next()` is called before each iteration, and then `res.done` is consulted. If `res.done` is `true`, the expression evaluates to `false` and the iteration doesn't occur.

Recall earlier that we suggested iterators should in general not return `done: true` along with the final intended value from the iterator. Now you can see why.

If an iterator returned `{ done: true, value: 42 }`, the `for..of` loop would completely discard the `42` value and it'd be lost. For this reason, assuming that your iterator may be consumed by patterns like the `for..of` loop or its manual `for` equivalent, you should probably wait to return `done: true` for signaling completion until after you've already returned all relevant iteration values.

Warning: You can, of course, intentionally design your iterator to return some relevant `value` at the same time as returning `done: true`. But don't do this unless you've documented that as the case, and thus implicitly forced consumers of your iterator to use a different pattern for iteration than is implied by `for..of` or its manual equivalent we depicted.

Custom Iterators

In addition to the standard built-in iterators, you can make your own! All it takes to make them interoperate with ES6's consumption facilities (e.g., the `for..of` loop and the `...` operator) is to adhere to the proper interface(s).

Let's try constructing an iterator that produces the infinite series of numbers in the Fibonacci sequence:

```

var Fib = {
  [Symbol.iterator]() {
    var n1 = 1, n2 = 1;

    return {
      // make the iterator an iterable
      [Symbol.iterator]() { return this; },

      next() {
        var current = n2;
        n2 = n1;
        n1 = n1 + current;
        return { value: current, done: false };
      },
      return(v) {
        console.log(
          "Fibonacci sequence abandoned."
        );
        return { value: v, done: true };
      }
    };
  };
};

for (var v of Fib) {
  console.log( v );

  if (v > 50) break;
}
// 1 1 2 3 5 8 13 21 34 55
// Fibonacci sequence abandoned.

```

Warning: If we hadn't inserted the `break` condition, this `for...of` loop would have run forever, which is probably not the desired result in terms of breaking your program!

The `Fib[Symbol.iterator]()` method when called returns the iterator object with `next()` and `return(...)` methods on it. State is maintained via `n1` and `n2` variables, which are kept by the closure.

Let's *next* consider an iterator that is designed to run through a series (aka a queue) of actions, one item at a time:

```
var tasks = {
  [Symbol.iterator]() {
    var steps = this.actions.slice();

    return {
      // make the iterator an iterable
      [Symbol.iterator]() { return this; },

      next(...args) {
        if (steps.length > 0) {
          let res = steps.shift()( ...args );
          return { value: res, done: false };
        }
        else {
          return { done: true }
        }
      },
      return(v) {
        steps.length = 0;
        return { value: v, done: true };
      }
    };
  },
  actions: []
};
```

The iterator on `tasks` steps through functions found in the `actions` array property, if any, and executes them one at a time, passing in whatever arguments you pass to `next(..)`, and returning any return value to you in the standard `IteratorResult` object.

Here's how we could use this `tasks` queue:

```
tasks.actions.push(
  function step1(x){
    console.log( "step 1:", x );
    return x * 2;
  },
  function step2(x,y){
    console.log( "step 2:", x, y );
    return x + (y * 2);
  },
  function step3(x,y,z){
    console.log( "step 3:", x, y, z );
    return (x * y) + z;
  }
);

var it = tasks[Symbol.iterator]();

it.next( 10 );           // step 1: 10
                     // { value: 20, done: false }

it.next( 20, 50 );        // step 2: 20 50
                     // { value: 120, done: false }

it.next( 20, 50, 120 );   // step 3: 20 50 120
                     // { value: 1120, done: false }

it.next();                // { done: true }
```

This particular usage reinforces that iterators can be a pattern for organizing functionality, not just data. It's also reminiscent of what we'll see with generators in the next section.

You could even get creative and define an iterator that represents meta operations on a single piece of data. For example, we could define an iterator for numbers that by default ranges from `0` up to (or down to, for negative numbers) the number in question.

Consider:

```

if (!Number.prototype[Symbol.iterator]) {
  Object.defineProperty(
    Number.prototype,
    Symbol.iterator,
    {
      writable: true,
      configurable: true,
      enumerable: false,
      value: function iterator(){
        var i, inc, done = false, top = +this;

        // iterate positively or negatively?
        inc = 1 * (top < 0 ? -1 : 1);

        return {
          // make the iterator itself an iterable!
          [Symbol.iterator](){ return this; },

          next() {
            if (!done) {
              // initial iteration always 0
              if (i == null) {
                i = 0;
              }
              // iterating positively
              else if (top >= 0) {
                i = Math.min(top,i + inc);
              }
              // iterating negatively
              else {
                i = Math.max(top,i + inc);
              }
            }

            // done after this iteration?
            if (i == top) done = true;

            return { value: i, done: false };
          }
          else {
            return { done: true };
          }
        };
      };
    }
  );
}

```

Now, what tricks does this creativity afford us?

```

for (var i of 3) {
    console.log( i );
}
// 0 1 2 3

[...-3];           // [0, -1, -2, -3]

```

Those are some fun tricks, though the practical utility is somewhat debatable. But then again, one might wonder why ES6 didn't just ship with such a minor but delightful feature easter egg!?

I'd be remiss if I didn't at least remind you that extending native prototypes as I'm doing in the previous snippet is something you should only do with caution and awareness of potential hazards.

In this case, the chances that you'll have a collision with other code or even a future JS feature is probably exceedingly low. But just beware of the slight possibility. And document what you're doing verbosely for posterity's sake.

Note: I've expounded on this particular technique in this blog post (<http://blog.getify.com/iterating-es6-numbers/>) if you want more details. And this comment (<http://blog.getify.com/iterating-es6-numbers/comment-page-1/#comment-535294>) even suggests a similar trick but for making string character ranges.

Iterator Consumption

We've already shown consuming an iterator item by item with the `for..of` loop. But there are other ES6 structures that can consume iterators.

Let's consider the iterator attached to this array (though any iterator we choose would have the following behaviors):

```
var a = [1, 2, 3, 4, 5];
```

The `...` spread operator fully exhausts an iterator. Consider:

```

function foo(x, y, z, w, p) {
    console.log( x + y + z + w + p );
}

foo( ...a );           // 15

```

`...` can also spread an iterator inside an array:

```
var b = [ 0, ...a, 6 ];
b;                                // [0,1,2,3,4,5,6]
```

Array destructuring (see "Destructuring" in Chapter 2) can partially or completely (if paired with a `... rest/gather operator`) consume an iterator:

```
var it = a[Symbol.iterator]();

var [x,y] = it;                  // take just the first two elements from `it`
var [z, ...w] = it;              // take the third, then the rest all at once

// is `it` fully exhausted? Yep.
it.next();                      // { value: undefined, done: true }

x;                                // 1
y;                                // 2
z;                                // 3
w;                                // [4,5]
```

Generators

All functions run to completion, right? In other words, once a function starts running, it finishes before anything else can interrupt.

At least that's how it's been for the whole history of JavaScript up to this point. As of ES6, a new somewhat exotic form of function is being introduced, called a generator. A generator can pause itself in mid-execution, and can be resumed either right away or at a later time. So it clearly does not hold the run-to-completion guarantee that normal functions do.

Moreover, each pause/resume cycle in mid-execution is an opportunity for two-way message passing, where the generator can return a value, and the controlling code that resumes it can send a value back in.

As with iterators in the previous section, there are multiple ways to think about what a generator is, or rather what it's most useful for. There's no one right answer, but we'll try to consider several angles.

Note: See the *Async & Performance* title of this series for more information about generators, and also see Chapter 4 of this current title.

Syntax

The generator function is declared with this new syntax:

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

The position of the `*` is not functionally relevant. The same declaration could be written as any of the following:

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

...
```

The *only* difference here is stylistic preference. Most other literature seems to prefer `function* foo(..) { ... }`. I prefer `function *foo(..) { ... }`, so that's how I'll present them for the rest of this title.

My reason is purely didactic in nature. In this text, when referring to a generator function, I will use `*foo(..)`, as opposed to `foo(..)` for a normal function. I observe that `*foo(..)` more closely matches the `*` positioning of `function *foo(..) { ... }`.

Moreover, as we saw in Chapter 2 with concise methods, there's a concise generator form in object literals:

```
var a = {
  *foo() { ... }
};
```

I would say that with concise generators, `*foo() { ... }` is rather more natural than `* foo()`. So that further argues for matching the consistency with `*foo()`.

Consistency eases understanding and learning.

Executing a Generator

Though a generator is declared with `*`, you still execute it like a normal function:

```
foo();
```

You can still pass it arguments, as in:

```
function *foo(x,y) {  
    // ..  
}  
  
foo( 5, 10 );
```

The major difference is that executing a generator, like `foo(5,10)` doesn't actually run the code in the generator. Instead, it produces an iterator that will control the generator to execute its code.

We'll come back to this later in "Iterator Control," but briefly:

```
function *foo() {  
    // ..  
}  
  
var it = foo();  
  
// to start/advanced `*foo()`, call  
// `it.next(..)`
```

yield

Generators also have a new keyword you can use inside them, to signal the pause point:

`yield`. Consider:

```
function *foo() {  
    var x = 10;  
    var y = 20;  
  
    yield;  
  
    var z = x + y;  
}
```

In this `*foo()` generator, the operations on the first two lines would run at the beginning, then `yield` would pause the generator. If and when resumed, the last line of `*foo()` would run. `yield` can appear any number of times (or not at all, technically!) in a generator.

You can even put `yield` inside a loop, and it can represent a repeated pause point. In fact, a loop that never completes just means a generator that never completes, which is completely valid, and sometimes entirely what you need.

`yield` is not just a pause point. It's an expression that sends out a value when pausing the generator. Here's a `while..true` loop in a generator that for each iteration `yield` s a new random number:

```
function *foo() {
  while (true) {
    yield Math.random();
  }
}
```

The `yield ..` expression not only sends a value -- `yield` without a value is the same as `yield undefined` -- but also receives (e.g., is replaced by) the eventual resumption value. Consider:

```
function *foo() {
  var x = yield 10;
  console.log( x );
}
```

This generator will first `yield` out the value `10` when pausing itself. When you resume the generator -- using the `it.next(..)` we referred to earlier -- whatever value (if any) you resume with will replace/complete the whole `yield 10` expression, meaning that value will be assigned to the `x` variable.

A `yield ..` expression can appear anywhere a normal expression can. For example:

```
function *foo() {
  var arr = [ yield 1, yield 2, yield 3 ];
  console.log( arr, yield 4 );
}
```

`*foo()` here has four `yield ..` expressions. Each `yield` results in the generator pausing to wait for a resumption value that's then used in the various expression contexts.

`yield` is not technically an operator, though when used like `yield 1` it sure looks like it. Because `yield` can be used all by itself as in `var x = yield;`, thinking of it as an operator can sometimes be confusing.

Technically, `yield ..` is of the same "expression precedence" -- similar conceptually to operator precedence -- as an assignment expression like `a = 3`. That means `yield ..` can basically appear anywhere `a = 3` can validly appear.

Let's illustrate the symmetry:

```

var a, b;

a = 3;           // valid
b = 2 + a = 3;  // invalid
b = 2 + (a = 3); // valid

yield 3;         // valid
a = 2 + yield 3; // invalid
a = 2 + (yield 3); // valid

```

Note: If you think about it, it makes a sort of conceptual sense that a `yield ..` expression would behave similar to an assignment expression. When a paused `yield` expression is resumed, it's completed/replaced by the resumption value in a way that's not terribly dissimilar from being "assigned" that value.

The takeaway: if you need `yield ..` to appear in a position where an assignment like `a = 3` would not itself be allowed, it needs to be wrapped in a `()`.

Because of the low precedence of the `yield` keyword, almost any expression after a `yield ..` will be computed first before being sent with `yield`. Only the `...` spread operator and the `,` comma operator have lower precedence, meaning they'd bind after the `yield` has been evaluated.

So just like with multiple operators in normal statements, another case where `()` might be needed is to override (elevate) the low precedence of `yield`, such as the difference between these expressions:

```

yield 2 + 3;          // same as `yield (2 + 3)`

(yield 2) + 3;        // `yield 2` first, then `+ 3`

```

Just like `=` assignment, `yield` is also "right-associative," which means that multiple `yield` expressions in succession are treated as having been `(..)` grouped from right to left. So, `yield yield yield 3` is treated as `yield (yield (yield 3))`. A "left-associative" interpretation like `((yield) yield) yield 3` would make no sense.

Just like with operators, it's a good idea to use `(..)` grouping, even if not strictly required, to disambiguate your intent if `yield` is combined with other operators or `yield`s.

Note: See the *Types & Grammar* title of this series for more information about operator precedence and associativity.

yield *

In the same way that the `*` makes a `function` declaration into `function *` generator declaration, a `*` makes `yield` into `yield *`, which is a very different mechanism, called *yield delegation*. Grammatically, `yield *...` will behave the same as a `yield ...`, as discussed in the previous section.

`yield * ...` requires an iterable; it then invokes that iterable's iterator, and delegates its own host generator's control to that iterator until it's exhausted. Consider:

```
function *foo() {
    yield *[1, 2, 3];
}
```

Note: As with the `*` position in a generator's declaration (discussed earlier), the `*` positioning in `yield *` expressions is stylistically up to you. Most other literature prefers `yield* ...`, but I prefer `yield *...`, for very symmetrical reasons as already discussed.

The `[1, 2, 3]` value produces an iterator that will step through its values, so the `*foo()` generator will yield those values out as it's consumed. Another way to illustrate the behavior is in `yield` delegating to another generator:

```
function *foo() {
    yield 1;
    yield 2;
    yield 3;
}

function *bar() {
    yield *foo();
}
```

The iterator produced when `*bar()` calls `*foo()` is delegated to via `yield *`, meaning whatever value(s) `*foo()` produces will be produced by `*bar()`.

Whereas with `yield ...` the completion value of the expression comes from resuming the generator with `it.next(...)`, the completion value of the `yield *...` expression comes from the return value (if any) from the delegated-to iterator.

Built-in iterators generally don't have return values, as we covered at the end of the "Iterator Loop" section earlier in this chapter. But if you define your own custom iterator (or generator), you can design it to `return` a value, which `yield *...` would capture:

```

function *foo() {
    yield 1;
    yield 2;
    yield 3;
    return 4;
}

function *bar() {
    var x = yield *foo();
    console.log( "x:", x );
}

for (var v of bar()) {
    console.log( v );
}
// 1 2 3
// x: 4

```

While the `1`, `2`, and `3` values are yielded out of `*foo()` and then out of `*bar()`, the `4` value returned from `*foo()` is the completion value of the `yield *foo()` expression, which then gets assigned to `x`.

Because `yield *` can call another generator (by way of delegating to its iterator), it can also perform a sort of generator recursion by calling itself:

```

function *foo(x) {
    if (x < 3) {
        x = yield *foo( x + 1 );
    }
    return x * 2;
}

foo( 1 );

```

The result from `foo(1)` and then calling the iterator's `next()` to run it through its recursive steps will be `24`. The first `*foo(..)` run has `x` at value `1`, which is `x < 3`. `x + 1` is passed recursively to `*foo(..)`, so `x` is then `2`. One more recursive call results in `x` of `3`.

Now, because `x < 3` fails, the recursion stops, and `return 3 * 2` gives `6` back to the previous call's `yield *...` expression, which is then assigned to `x`. Another `return 6 * 2` returns `12` back to the previous call's `x`. Finally `12 * 2`, or `24`, is returned from the completed run of the `*foo(..)` generator.

Iterator Control

Earlier, we briefly introduced the concept that generators are controlled by iterators. Let's fully dig into that now.

Recall the recursive `*foo(..)` from the previous section. Here's how we'd run it:

```
function *foo(x) {
  if (x < 3) {
    x = yield *foo( x + 1 );
  }
  return x * 2;
}

var it = foo( 1 );
it.next();           // { value: 24, done: true }
```

In this case, the generator doesn't really ever pause, as there's no `yield ..` expression. Instead, `yield *` just keeps the current iteration step going via the recursive call. So, just one call to the iterator's `next()` function fully runs the generator.

Now let's consider a generator that will have multiple steps and thus multiple produced values:

```
function *foo() {
  yield 1;
  yield 2;
  yield 3;
}
```

We already know we can consume an iterator, even one attached to a generator like `*foo()`, with a `for..of` loop:

```
for (var v of foo()) {
  console.log( v );
}
// 1 2 3
```

Note: The `for..of` loop requires an iterable. A generator function reference (like `foo`) by itself is not an iterable; you must execute it with `foo()` to get the iterator (which is also an iterable, as we explained earlier in this chapter). You could theoretically extend the `GeneratorPrototype` (the prototype of all generator functions) with a `Symbol.iterator` function that essentially just does `return this()`. That would make the `foo` reference itself an iterable, which means `for (var v of foo) { .. }` (notice no `()` on `foo`) will work.

Let's instead iterate the generator manually:

```

function *foo() {
    yield 1;
    yield 2;
    yield 3;
}

var it = foo();

it.next();           // { value: 1, done: false }
it.next();           // { value: 2, done: false }
it.next();           // { value: 3, done: false }

it.next();           // { value: undefined, done: true }

```

If you look closely, there are three `yield` statements and four `next()` calls. That may seem like a strange mismatch. In fact, there will always be one more `next()` call than `yield` expression, assuming all are evaluated and the generator is fully run to completion.

But if you look at it from the opposite perspective (inside-out instead of outside-in), the matching between `yield` and `next()` makes more sense.

Recall that the `yield ...` expression will be completed by the value you resume the generator with. That means the argument you pass to `next(..)` completes whatever `yield ...` expression is currently paused waiting for a completion.

Let's illustrate this perspective this way:

```

function *foo() {
    var x = yield 1;
    var y = yield 2;
    var z = yield 3;
    console.log( x, y, z );
}

```

In this snippet, each `yield ...` is sending a value out (`1`, `2`, `3`), but more directly, it's pausing the generator to wait for a value. In other words, it's almost like asking the question, "What value should I use here? I'll wait to hear back."

Now, here's how we control `*foo()` to start it up:

```

var it = foo();

it.next();           // { value: 1, done: false }

```

That first `next()` call is starting up the generator from its initial paused state, and running it to the first `yield`. At the moment you call that first `next()`, there's no `yield ..` expression waiting for a completion. If you passed a value to that first `next()` call, it would currently just be thrown away, because no `yield` is waiting to receive such a value.

Note: An early proposal for the "beyond ES6" timeframe *would* let you access a value passed to an initial `next(..)` call via a separate meta property (see Chapter 7) inside the generator.

Now, let's answer the currently pending question, "What value should I assign to `x`?" We'll answer it by sending a value to the `next next(..)` call:

```
it.next( "foo" );           // { value: 2, done: false }
```

Now, the `x` will have the value `"foo"`, but we've also asked a new question, "What value should I assign to `y`?" And we answer:

```
it.next( "bar" );           // { value: 3, done: false }
```

Answer given, another question asked. Final answer:

```
it.next( "baz" );           // "foo" "bar" "baz"
                           // { value: undefined, done: true }
```

Now it should be clearer how each `yield ..` "question" is answered by the `next next(..)` call, and so the "extra" `next()` call we observed is always just the initial one that starts everything going.

Let's put all those steps together:

```
var it = foo();

// start up the generator
it.next();                 // { value: 1, done: false }

// answer first question
it.next( "foo" );          // { value: 2, done: false }

// answer second question
it.next( "bar" );          // { value: 3, done: false }

// answer third question
it.next( "baz" );          // "foo" "bar" "baz"
                           // { value: undefined, done: true }
```

You can think of a generator as a producer of values, in which case each iteration is simply producing a value to be consumed.

But in a more general sense, perhaps it's appropriate to think of generators as controlled, progressive code execution, much like the `tasks` queue example from the earlier "Custom Iterators" section.

Note: That perspective is exactly the motivation for how we'll revisit generators in Chapter 4. Specifically, there's no reason that `next(..)` has to be called right away after the previous `next(..)` finishes. While the generator's inner execution context is paused, the rest of the program continues unblocked, including the ability for asynchronous actions to control when the generator is resumed.

Early Completion

As we covered earlier in this chapter, the iterator attached to a generator supports the optional `return(..)` and `throw(..)` methods. Both of them have the effect of aborting a paused generator immediately.

Consider:

```
function *foo() {
  yield 1;
  yield 2;
  yield 3;
}

var it = foo();

it.next();           // { value: 1, done: false }

it.return( 42 );    // { value: 42, done: true }

it.next();          // { value: undefined, done: true }
```

`return(x)` is kind of like forcing a `return x` to be processed at exactly that moment, such that you get the specified value right back. Once a generator is completed, either normally or early as shown, it no longer processes any code or returns any values.

In addition to `return(..)` being callable manually, it's also called automatically at the end of iteration by any of the ES6 constructs that consume iterators, such as the `for..of` loop and the `...` spread operator.

The purpose for this capability is so the generator can be notified if the controlling code is no longer going to iterate over it anymore, so that it can perhaps do any cleanup tasks (freeing up resources, resetting status, etc.). Identical to a normal function cleanup pattern, the main way to accomplish this is to use a `finally` clause:

```
function *foo() {
  try {
    yield 1;
    yield 2;
    yield 3;
  }
  finally {
    console.log( "cleanup!" );
  }
}

for (var v of foo()) {
  console.log( v );
}
// 1 2 3
// cleanup!

var it = foo();

it.next();           // { value: 1, done: false }
it.return( 42 );    // cleanup!
                    // { value: 42, done: true }
```

Warning: Do not put a `yield` statement inside the `finally` clause! It's valid and legal, but it's a really terrible idea. It acts in a sense as deferring the completion of the `return(...)` call you made, as any `yield ..` expressions in the `finally` clause are respected to pause and send messages; you don't immediately get a completed generator as expected. There's basically no good reason to opt in to that crazy *bad part*, so avoid doing so!

In addition to the previous snippet showing how `return(...)` aborts the generator while still triggering the `finally` clause, it also demonstrates that a generator produces a whole new iterator each time it's called. In fact, you can use multiple iterators attached to the same generator concurrently:

```
function *foo() {
    yield 1;
    yield 2;
    yield 3;
}

var it1 = foo();
it1.next();           // { value: 1, done: false }
it1.next();           // { value: 2, done: false }

var it2 = foo();
it2.next();           // { value: 1, done: false }

it1.next();           // { value: 3, done: false }

it2.next();           // { value: 2, done: false }
it2.next();           // { value: 3, done: false }

it2.next();           // { value: undefined, done: true }
it1.next();           // { value: undefined, done: true }
```

Early Abort

Instead of calling `return(..)`, you can call `throw(..)`. Just like `return(x)` is essentially injecting a `return x` into the generator at its current pause point, calling `throw(x)` is essentially like injecting a `throw x` at the pause point.

Other than the exception behavior (we cover what that means to `try` clauses in the next section), `throw(..)` produces the same sort of early completion that aborts the generator's run at its current pause point. For example:

```
function *foo() {
    yield 1;
    yield 2;
    yield 3;
}

var it = foo();

it.next();           // { value: 1, done: false }

try {
    it.throw( "Oops!" );
}
catch (err) {
    console.log( err );    // Exception: Oops!
}

it.next();           // { value: undefined, done: true }
```

Because `throw(..)` basically injects a `throw ..` in replacement of the `yield 1` line of the generator, and nothing handles this exception, it immediately propagates back out to the calling code, which handles it with a `try..catch`.

Unlike `return(..)`, the iterator's `throw(..)` method is never called automatically.

Of course, though not shown in the previous snippet, if a `try..finally` clause was waiting inside the generator when you call `throw(..)`, the `finally` clause would be given a chance to complete before the exception is propagated back to the calling code.

Error Handling

As we've already hinted, error handling with generators can be expressed with `try..catch`, which works in both inbound and outbound directions:

```
function *foo() {
  try {
    yield 1;
  }
  catch (err) {
    console.log( err );
  }

  yield 2;

  throw "Hello!";
}

var it = foo();

it.next();           // { value: 1, done: false }

try {
  it.throw( "Hi!" );    // Hi!
                        // { value: 2, done: false }
  it.next();

  console.log( "never gets here" );
}
catch (err) {
  console.log( err );    // Hello!
}
```

Errors can also propagate in both directions through `yield * delegation`:

```

function *foo() {
  try {
    yield 1;
  }
  catch (err) {
    console.log( err );
  }

  yield 2;

  throw "foo: e2";
}

function *bar() {
  try {
    yield *foo();

    console.log( "never gets here" );
  }
  catch (err) {
    console.log( err );
  }
}

var it = bar();

try {
  it.next();           // { value: 1, done: false }

  it.throw( "e1" );   // e1
                      // { value: 2, done: false }

  it.next();           // foo: e2
                      // { value: undefined, done: true }
}
catch (err) {
  console.log( "never gets here" );
}

it.next();           // { value: undefined, done: true }

```

When `*foo()` calls `yield 1`, the `1` value passes through `*bar()` untouched, as we've already seen.

But what's most interesting about this snippet is that when `*foo()` calls `throw "foo: e2"`, this error propagates to `*bar()` and is immediately caught by `*bar()`'s `try..catch` block. The error doesn't pass through `*bar()` like the `1` value did.

`*bar()`'s `catch` then does a normal output of `err` (`"foo: e2"`) and then `*bar()` finishes normally, which is why the `{ value: undefined, done: true }` iterator result comes back from `it.next()`.

If `*bar()` didn't have a `try..catch` around the `yield *...` expression, the error would of course propagate all the way out, and on the way through it still would complete (abort) `*bar()`.

Transpiling a Generator

Is it possible to represent a generator's capabilities prior to ES6? It turns out it is, and there are several great tools that do so, including most notably Facebook's Regenerator tool (<https://facebook.github.io/regenerator/>).

But just to better understand generators, let's try our hand at manually converting. Basically, we're going to create a simple closure-based state machine.

We'll keep our source generator really simple:

```
function *foo() {
  var x = yield 42;
  console.log(x);
}
```

To start, we'll need a function called `foo()` that we can execute, which needs to return an iterator:

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

  return {
    next: function(v) {
      // ...
    }

    // we'll skip `return(..)` and `throw(..)`
  };
}
```

Now, we need some inner variable to keep track of where we are in the steps of our "generator"'s logic. We'll call it `state`. There will be three states: `0` initially, `1` while waiting to fulfill the `yield` expression, and `2` once the generator is complete.

Each time `next(..)` is called, we need to process the next step, and then increment `state`. For convenience, we'll put each step into a `case` clause of a `switch` statement, and we'll hold that in an inner function called `nextState(..)` that `next(..)` can call. Also, because `x` is a variable across the overall scope of the "generator," it needs to live outside the `nextState(..)` function.

Here it is all together (obviously somewhat simplified, to keep the conceptual illustration clearer):

```
function foo() {
  function nextState(v) {
    switch (state) {
      case 0:
        state++;

        // the `yield` expression
        return 42;
      case 1:
        state++;

        // `yield` expression fulfilled
        x = v;
        console.log( x );

        // the implicit `return`
        return undefined;

        // no need to handle state `2`
    }
  }

  var state = 0, x;

  return {
    next: function(v) {
      var ret = nextState( v );

      return { value: ret, done: (state == 2) };
    }

    // we'll skip `return(..)` and `throw(..)`
  };
}
```

And finally, let's test our pre-ES6 "generator":

```
var it = foo();

it.next();           // { value: 42, done: false }

it.next( 10 );      // 10
                    // { value: undefined, done: true }
```

Not bad, huh? Hopefully this exercise solidifies in your mind that generators are actually just simple syntax for state machine logic. That makes them widely applicable.

Generator Uses

So, now that we much more deeply understand how generators work, what are they useful for?

We've seen two major patterns:

- *Producing a series of values*: This usage can be simple (e.g., random strings or incremented numbers), or it can represent more structured data access (e.g., iterating over rows returned from a database query).

Either way, we use the iterator to control a generator so that some logic can be invoked for each call to `next(...)`. Normal iterators on data structures merely pull values without any controlling logic.

- *Queue of tasks to perform serially*: This usage often represents flow control for the steps in an algorithm, where each step requires retrieval of data from some external source. The fulfillment of each piece of data may be immediate, or may be asynchronously delayed.

From the perspective of the code inside the generator, the details of sync or async at a `yield` point are entirely opaque. Moreover, these details are intentionally abstracted away, such as not to obscure the natural sequential expression of steps with such implementation complications. Abstraction also means the implementations can be swapped/refactored often without touching the code in the generator at all.

When generators are viewed in light of these uses, they become a lot more than just a different or nicer syntax for a manual state machine. They are a powerful abstraction tool for organizing and controlling orderly production and consumption of data.

Modules

I don't think it's an exaggeration to suggest that the single most important code organization pattern in all of JavaScript is, and always has been, the module. For myself, and I think for a large cross-section of the community, the module pattern drives the vast majority of code.

The Old Way

The traditional module pattern is based on an outer function with inner variables and functions, and a returned "public API" with methods that have closure over the inner data and capabilities. It's often expressed like this:

```
function Hello(name) {
  function greeting() {
    console.log( "Hello " + name + "!" );
  }

  // public API
  return {
    greeting: greeting
  };
}

var me = Hello( "Kyle" );
me.greeting();           // Hello Kyle!
```

This `Hello(..)` module can produce multiple instances by being called subsequent times. Sometimes, a module is only called for as a singleton (i.e., it just needs one instance), in which case a slight variation on the previous snippet, using an IIFE, is common:

```
var me = (function Hello(name){
  function greeting() {
    console.log( "Hello " + name + "!" );
  }

  // public API
  return {
    greeting: greeting
  };
})( "Kyle" );

me.greeting();           // Hello Kyle!
```

This pattern is tried and tested. It's also flexible enough to have a wide assortment of variations for a number of different scenarios.

One of the most common is the Asynchronous Module Definition (AMD), and another is the Universal Module Definition (UMD). We won't cover the particulars of these patterns and techniques here, but they're explained extensively in many places online.

Moving Forward

As of ES6, we no longer need to rely on the enclosing function and closure to provide us with module support. ES6 modules have first class syntactic and functional support.

Before we get into the specific syntax, it's important to understand some fairly significant conceptual differences with ES6 modules compared to how you may have dealt with modules in the past:

- ES6 uses file-based modules, meaning one module per file. At this time, there is no standardized way of combining multiple modules into a single file.

That means that if you are going to load ES6 modules directly into a browser web application, you will be loading them individually, not as a large bundle in a single file as has been common in performance optimization efforts.

It's expected that the contemporaneous advent of HTTP/2 will significantly mitigate any such performance concerns, as it operates on a persistent socket connection and thus can very efficiently load many smaller files in parallel and interleaved with one another.

- The API of an ES6 module is static. That is, you define statically what all the top-level exports are on your module's public API, and those cannot be amended later.

Some uses are accustomed to being able to provide dynamic API definitions, where methods can be added/removed/replaced in response to runtime conditions. Either these uses will have to change to fit with ES6 static APIs, or they will have to restrain the dynamic changes to properties/methods of a second-level object.

- ES6 modules are singletons. That is, there's only one instance of the module, which maintains its state. Every time you import that module into another module, you get a reference to the one centralized instance. If you want to be able to produce multiple module instances, your module will need to provide some sort of factory to do it.
- The properties and methods you expose on a module's public API are not just normal assignments of values or references. They are actual bindings (almost like pointers) to the identifiers in your inner module definition.

In pre-ES6 modules, if you put a property on your public API that holds a primitive value like a number or string, that property assignment was by value-copy, and any internal update of a corresponding variable would be separate and not affect the public copy on the API object.

With ES6, exporting a local private variable, even if it currently holds a primitive string/number/etc, exports a binding to the variable. If the module changes the variable's value, the external import binding now resolves to that new value.

- Importing a module is the same thing as statically requesting it to load (if it hasn't already). If you're in a browser, that implies a blocking load over the network. If you're on a server (i.e., Node.js), it's a blocking load from the filesystem.

However, don't panic about the performance implications. Because ES6 modules have static definitions, the import requirements can be statically scanned, and loads will happen preemptively, even before you've used the module.

ES6 doesn't actually specify or handle the mechanics of how these load requests work. There's a separate notion of a Module Loader, where each hosting environment (browser, Node.js, etc.) provides a default Loader appropriate to the environment. The importing of a module uses a string value to represent where to get the module (URL, file path, etc.), but this value is opaque in your program and only meaningful to the Loader itself.

You can define your own custom Loader if you want more fine-grained control than the default Loader affords -- which is basically none, as it's totally hidden from your program's code.

As you can see, ES6 modules will serve the overall use case of organizing code with encapsulation, controlling public APIs, and referencing dependency imports. But they have a very particular way of doing so, and that may or may not fit very closely with how you've already been doing modules for years.

CommonJS

There's a similar, but not fully compatible, module syntax called CommonJS, which is familiar to those in the Node.js ecosystem.

For lack of a more tactful way to say this, in the long run, ES6 modules essentially are bound to supersede all previous formats and standards for modules, even CommonJS, as they are built on syntactic support in the language. This will, in time, inevitably win out as the superior approach, if for no other reason than ubiquity.

We face a fairly long road to get to that point, though. There are literally hundreds of thousands of CommonJS style modules in the server-side JavaScript world, and 10 times that many modules of varying format standards (UMD, AMD, ad hoc) in the browser world. It will take many years for the transitions to make any significant progress.

In the interim, module transpilers/converters will be an absolute necessity. You might as well just get used to that new reality. Whether you author in regular modules, AMD, UMD, CommonJS, or ES6, these tools will have to parse and convert to a format that is suitable for whatever environment your code will run in.

For Node.js, that probably means (for now) that the target is CommonJS. For the browser, it's probably UMD or AMD. Expect lots of flux on this over the next few years as these tools mature and best practices emerge.

From here on out, my best advice on modules is this: whatever format you've been religiously attached to with strong affinity, also develop an appreciation for and understanding of ES6 modules, such as they are, and let your other module tendencies fade. They *are* the future of modules in JS, even if that reality is a bit of a ways off.

The New Way

The two main new keywords that enable ES6 modules are `import` and `export`. There's lots of nuance to the syntax, so let's take a deeper look.

Warning: An important detail that's easy to overlook: both `import` and `export` must always appear in the top-level scope of their respective usage. For example, you cannot put either an `import` or `export` inside an `if` conditional; they must appear outside of all blocks and functions.

`export` ing API Members

The `export` keyword is either put in front of a declaration, or used as an operator (of sorts) with a special list of bindings to export. Consider:

```
export function foo() {  
    // ..  
}  
  
export var awesome = 42;  
  
var bar = [1, 2, 3];  
export { bar };
```

Another way of expressing the same exports:

```
function foo() {  
    // ..  
}  
  
var awesome = 42;  
var bar = [1, 2, 3];  
  
export { foo, awesome, bar };
```

These are all called *named exports*, as you are in effect exporting the name bindings of the variables/functions/etc.

Anything you don't *label* with `export` stays private inside the scope of the module. That is, although something like `var bar = ...` looks like it's declaring at the top-level global scope, the top-level scope is actually the module itself; there is no global scope in modules.

Note: Modules *do* still have access to `window` and all the "globals" that hang off it, just not as lexical top-level scope. However, you really should stay away from the globals in your modules if at all possible.

You can also "rename" (aka alias) a module member during named export:

```
function foo() { .. }  
  
export { foo as bar };
```

When this module is imported, only the `bar` member name is available to import; `foo` stays hidden inside the module.

Module exports are not just normal assignments of values or references, as you're accustomed to with the `=` assignment operator. Actually, when you export something, you're exporting a binding (kinda like a pointer) to that thing (variable, etc.).

Within your module, if you change the value of a variable you already exported a binding to, even if it's already been imported (see the next section), the imported binding will resolve to the current (updated) value.

Consider:

```
var awesome = 42;  
export { awesome };  
  
// later  
awesome = 100;
```

When this module is imported, regardless of whether that's before or after the `awesome = 100` setting, once that assignment has happened, the imported binding resolves to the `100` value, not `42`.

That's because the binding is, in essence, a reference to, or a pointer to, the `awesome` variable itself, rather than a copy of its value. This is a mostly unprecedented concept for JS introduced with ES6 module bindings.

Though you can clearly use `export` multiple times inside a module's definition, ES6 definitely prefers the approach that a module has a single export, which is known as a *default export*. In the words of some members of the TC39 committee, you're "rewarded with simpler `import` syntax" if you follow that pattern, and conversely "penalized" with more verbose syntax if you don't.

A default export sets a particular exported binding to be the default when importing the module. The name of the binding is literally `default`. As you'll see later, when importing module bindings you can also rename them, as you commonly will with a default export.

There can only be one `default` per module definition. We'll cover `import` in the next section, and you'll see how the `import` syntax is more concise if the module has a default export.

There's a subtle nuance to default export syntax that you should pay close attention to. Compare these two snippets:

```
function foo(...){  
    // ..  
}  
  
export default foo;
```

And this one:

```
function foo(...){  
    // ..  
}  
  
export { foo as default };
```

In the first snippet, you are exporting a binding to the function expression value at that moment, *not* to the identifier `foo`. In other words, `export default ..` takes an expression. If you later assign `foo` to a different value inside your module, the module import still reveals the function originally exported, not the new value.

By the way, the first snippet could also have been written as:

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

Warning: Although the `function foo...` part here is technically a function expression, for the purposes of the internal scope of the module, it's treated like a function declaration, in that the `foo` name is bound in the module's top-level scope (often called "hoisting"). The same is true for `export default class Foo...`. However, while you *can* do `export var foo = ...`, you currently cannot do `export default var foo = ...` (or `let` or `const`), in a frustrating case of inconsistency. At the time of this writing, there's already discussion of adding that capability in soon, post-ES6, for consistency sake.

Recall the second snippet again:

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

export { foo as default };
```

In this version of the module export, the default export binding is actually to the `foo` identifier rather than its value, so you get the previously described binding behavior (i.e., if you later change `foo`'s value, the value seen on the import side will also be updated).

Be very careful of this subtle gotcha in default export syntax, especially if your logic calls for export values to be updated. If you never plan to update a default export's value, `export default ...` is fine. If you do plan to update the value, you must use `export { ... as default }`. Either way, make sure to comment your code to explain your intent!

Because there can only be one `default` per module, you may be tempted to design your module with one default export of an object with all your API methods on it, such as:

```
export default {
  foo() { ... },
  bar() { ... },
  ...
};
```

That pattern seems to map closely to how a lot of developers have already structured their pre-ES6 modules, so it seems like a natural approach. Unfortunately, it has some downsides and is officially discouraged.

In particular, the JS engine cannot statically analyze the contents of a plain object, which means it cannot do some optimizations for static `import` performance. The advantage of having each member individually and explicitly exported is that the engine *can* do the static analysis and optimization.

If your API has more than one member already, it seems like these principles -- one default export per module, and all API members as named exports -- are in conflict, doesn't it? But you *can* have a single default export as well as other named exports; they are not mutually exclusive.

So, instead of this (discouraged) pattern:

```
export default function foo() { ... }

foo.bar = function() { ... };
foo.baz = function() { ... };
```

You can do:

```
export default function foo() { ... }

export function bar() { ... }
export function baz() { ... }
```

Note: In this previous snippet, I used the name `foo` for the function that `default` labels. That `foo` name, however, is ignored for the purposes of export -- `default` is actually the exported name. When you import this default binding, you can give it whatever name you want, as you'll see in the next section.

Alternatively, some will prefer:

```
function foo() { ... }
function bar() { ... }
function baz() { ... }

export { foo as default, bar, baz, ... };
```

The effects of mixing default and named exports will be more clear when we cover `import` shortly. But essentially it means that the most concise default import form would only retrieve the `foo()` function. The user could additionally manually list `bar` and `baz` as named imports, if they want them.

You can probably imagine how tedious that's going to be for consumers of your module if you have lots of named export bindings. There is a wildcard import form where you import all of a module's exports within a single namespace object, but there's no way to wildcard import to top-level bindings.

Again, the ES6 module mechanism is intentionally designed to discourage modules with lots of exports; relatively speaking, it's desired that such approaches be a little more difficult, as a sort of social engineering to encourage simple module design in favor of large/complex module design.

I would probably recommend you not mix default export with named exports, especially if you have a large API and refactoring to separate modules isn't practical or desired. In that case, just use all named exports, and document that consumers of your module should probably use the `import * as ..` (namespace import, discussed in the next section) approach to bring the whole API in at once on a single namespace.

We mentioned this earlier, but let's come back to it in more detail. Other than the `export default ...` form that exports an expression value binding, all other export forms are exporting bindings to local identifiers. For those bindings, if you change the value of a variable inside a module after exporting, the external imported binding will access the updated value:

```
var foo = 42;
export { foo as default };

export var bar = "hello world";

foo = 10;
bar = "cool";
```

When you import this module, the `default` and `bar` exports will be bound to the local variables `foo` and `bar`, meaning they will reveal the updated `10` and `"cool"` values. The values at time of export are irrelevant. The values at time of import are irrelevant. The bindings are live links, so all that matters is what the current value is when you access the binding.

Warning: Two-way bindings are not allowed. If you import a `foo` from a module, and try to change the value of your imported `foo` variable, an error will be thrown! We'll revisit that in the next section.

You can also re-export another module's exports, such as:

```
export { foo, bar } from "baz";
export { foo as FOO, bar as BAR } from "baz";
export * from "baz";
```

Those forms are similar to just first importing from the `"baz"` module then listing its members explicitly for export from your module. However, in these forms, the members of the `"baz"` module are never imported to your module's local scope; they sort of pass through untouched.

importing API Members

To import a module, unsurprisingly you use the `import` statement. Just as `export` has several nuanced variations, so does `import`, so spend plenty of time considering the following issues and experimenting with your options.

If you want to import certain specific named members of a module's API into your top-level scope, you use this syntax:

```
import { foo, bar, baz } from "foo";
```

Warning: The `{ ... }` syntax here may look like an object literal, or even an object destructuring syntax. However, its form is special just for modules, so be careful not to confuse it with other `{ ... }` patterns elsewhere.

The `"foo"` string is called a *module specifier*. Because the whole goal is statically analyzable syntax, the module specifier must be a string literal; it cannot be a variable holding the string value.

From the perspective of your ES6 code and the JS engine itself, the contents of this string literal are completely opaque and meaningless. The module loader will interpret this string as an instruction of where to find the desired module, either as a URL path or a local filesystem path.

The `foo`, `bar`, and `baz` identifiers listed must match named exports on the module's API (static analysis and error assertion apply). They are bound as top-level identifiers in your current scope:

```
import { foo } from "foo";
foo();
```

You can rename the bound identifiers imported, as:

```
import { foo as theFooFunc } from "foo";  
  
theFooFunc();
```

If the module has just a default export that you want to import and bind to an identifier, you can opt to skip the `{ ... }` surrounding syntax for that binding. The `import` in this preferred case gets the nicest and most concise of the `import` syntax forms:

```
import foo from "foo";  
  
// or:  
import { default as foo } from "foo";
```

Note: As explained in the previous section, the `default` keyword in a module's `export` specifies a named export where the name is actually `default`, as is illustrated by the second more verbose syntax option. The renaming from `default` to, in this case, `foo`, is explicit in the latter syntax and is identical yet implicit in the former syntax.

You can also import a default export along with other named exports, if the module has such a definition. Recall this module definition from earlier:

```
export default function foo() { ... }  
  
export function bar() { ... }  
export function baz() { ... }
```

To import that module's default export and its two named exports:

```
import FOOFN, { bar, baz as BAZ } from "foo";  
  
FOOFN();  
bar();  
BAZ();
```

The strongly suggested approach from ES6's module philosophy is that you only import the specific bindings from a module that you need. If a module provides 10 API methods, but you only need two of them, some believe it wasteful to bring in the entire set of API bindings.

One benefit, besides code being more explicit, is that narrow imports make static analysis and error detection (accidentally using the wrong binding name, for instance) more robust.

Of course, that's just the standard position influenced by ES6 design philosophy; there's nothing that requires adherence to that approach.

Many developers would be quick to point out that such approaches can be more tedious, requiring you to regularly revisit and update your `import` statement(s) each time you realize you need something else from a module. The trade-off is in exchange for convenience.

In that light, the preference might be to import everything from the module into a single namespace, rather than importing individual members, each directly into the scope.

Fortunately, the `import` statement has a syntax variation that can support this style of module consumption, called *namespace import*.

Consider a `"foo"` module exported as:

```
export function bar() { ... }
export var x = 42;
export function baz() { ... }
```

You can import that entire API to a single module namespace binding:

```
import * as foo from "foo";

foo.bar();
foo.x;           // 42
foo.baz();
```

Note: The `* as ..` clause requires the `*` wildcard. In other words, you cannot do something like `import { bar, x } as foo from "foo"` to bring in only part of the API but still bind to the `foo` namespace. I would have liked something like that, but for ES6 it's all or nothing with the namespace import.

If the module you're importing with `* as ..` has a default export, it is named `default` in the namespace specified. You can additionally name the default import outside of the namespace binding, as a top-level identifier. Consider a `"world"` module exported as:

```
export default function foo() { ... }
export function bar() { ... }
export function baz() { ... }
```

And this `import`:

```
import foofn, * as hello from "world";

foofn();
hello.default();
hello.bar();
hello.baz();
```

While this syntax is valid, it can be rather confusing that one method of the module (the default export) is bound at the top-level of your scope, whereas the rest of the named exports (and one called `default`) are bound as properties on a differently named (`hello`) identifier namespace.

As I mentioned earlier, my suggestion would be to avoid designing your module exports in this way, to reduce the chances that your module's users will suffer these strange quirks.

All imported bindings are immutable and/or read-only. Consider the previous import; all of these subsequent assignment attempts will throw `TypeError`s:

```
import foofn, * as hello from "world";

foofn = 42;           // (runtime) TypeError!
hello.default = 42;  // (runtime) TypeError!
hello.bar = 42;      // (runtime) TypeError!
hello.baz = 42;      // (runtime) TypeError!
```

Recall earlier in the "exporting API Members" section that we talked about how the `bar` and `baz` bindings are bound to the actual identifiers inside the `"world"` module. That means if the module changes those values, `hello.bar` and `hello.baz` now reference the updated values.

But the immutable/read-only nature of your local imported bindings enforces that you cannot change them from the imported bindings, hence the `TypeError`s. That's pretty important, because without those protections, your changes would end up affecting all other consumers of the module (remember: singleton), which could create some very surprising side effects!

Moreover, though a module *can* change its API members from the inside, you should be very cautious of intentionally designing your modules in that fashion. ES6 modules are *intended* to be static, so deviations from that principle should be rare and should be carefully and verbosely documented.

Warning: There are module design philosophies where you actually intend to let a consumer change the value of a property on your API, or module APIs are designed to be "extended" by having other "plug-ins" add to the API namespace. As we just asserted, ES6 module APIs should be thought of and designed as static and unchangeable, which strongly restricts and discourages these alternative module design patterns. You can get around these limitations by exporting a plain object, which of course can then be changed at will. But be careful and think twice before going down that road.

Declarations that occur as a result of an `import` are "hoisted" (see the *Scope & Closures* title of this series). Consider:

```
foo();

import { foo } from "foo";
```

`foo()` can run because not only did the static resolution of the `import ..` statement figure out what `foo` is during compilation, but it also "hoisted" the declaration to the top of the module's scope, thus making it available throughout the module.

Finally, the most basic form of the `import` looks like this:

```
import "foo";
```

This form does not actually import any of the module's bindings into your scope. It loads (if not already loaded), compiles (if not already compiled), and evaluates (if not already run) the `"foo"` module.

In general, that sort of import is probably not going to be terribly useful. There may be niche cases where a module's definition has side effects (such as assigning things to the `window` /global object). You could also envision using `import "foo"` as a sort of preload for a module that may be needed later.

Circular Module Dependency

A imports B. B imports A. How does this actually work?

I'll state off the bat that designing systems with intentional circular dependency is generally something I try to avoid. That having been said, I recognize there are reasons people do this and it can solve some sticky design situations.

Let's consider how ES6 handles this. First, module `"A"` :

```
import bar from "B";

export default function foo(x) {
    if (x > 10) return bar( x - 1 );
    return x * 2;
}
```

Now, module `"B"` :

```
import foo from "A";

export default function bar(y) {
  if (y > 5) return foo( y / 2 );
  return y * 3;
}
```

These two functions, `foo(...)` and `bar(...)`, would work as standard function declarations if they were in the same scope, because the declarations are "hoisted" to the whole scope and thus available to each other regardless of authoring order.

With modules, you have declarations in entirely different scopes, so ES6 has to do extra work to help make these circular references work.

In a rough conceptual sense, this is how circular `import` dependencies are validated and resolved:

- If the `"A"` module is loaded first, the first step is to scan the file and analyze all the exports, so it can register all those bindings available for import. Then it processes the `import .. from "B"`, which signals that it needs to go fetch `"B"`.
- Once the engine loads `"B"`, it does the same analysis of its export bindings. When it sees the `import .. from "A"`, it knows the API of `"A"` already, so it can verify the `import` is valid. Now that it knows the `"B"` API, it can also validate the `import .. from "B"` in the waiting `"A"` module.

In essence, the mutual imports, along with the static verification that's done to validate both `import` statements, virtually composes the two separate module scopes (via the bindings), such that `foo(...)` can call `bar(...)` and vice versa. This is symmetric to if they had originally been declared in the same scope.

Now let's try using the two modules together. First, we'll try `foo(..)`:

```
import foo from "foo";
foo( 25 );           // 11
```

Or we can try `bar(..)`:

```
import bar from "bar";
bar( 25 );          // 11.5
```

By the time either the `foo(25)` or `bar(25)` calls are executed, all the analysis/compilation of all modules has completed. That means `foo(..)` internally knows directly about `bar(..)` and `bar(..)` internally knows directly about `foo(..)`.

If all we need is to interact with `foo(..)`, then we only need to import the `"foo"` module. Likewise with `bar(..)` and the `"bar"` module.

Of course, we *can* import and use both of them if we want to:

```
import foo from "foo";
import bar from "bar";

foo( 25 );           // 11
bar( 25 );           // 11.5
```

The static loading semantics of the `import` statement mean that a `"foo"` and `"bar"` that mutually depend on each other via `import` will ensure that both are loaded, parsed, and compiled before either of them runs. So their circular dependency is statically resolved and this works as you'd expect.

Module Loading

We asserted at the beginning of this "Modules" section that the `import` statement uses a separate mechanism, provided by the hosting environment (browser, Node.js, etc.), to actually resolve the module specifier string into some useful instruction for finding and loading the desired module. That mechanism is the system *Module Loader*.

The default module loader provided by the environment will interpret a module specifier as a URL if in the browser, and (generally) as a local filesystem path if on a server such as Node.js. The default behavior is to assume the loaded file is authored in the ES6 standard module format.

Moreover, you will be able to load a module into the browser via an HTML tag, similar to how current script programs are loaded. At the time of this writing, it's not fully clear if this tag will be `<script type="module">` or `<module>`. ES6 doesn't control that decision, but discussions in the appropriate standards bodies are already well along in parallel of ES6.

Whatever the tag looks like, you can be sure that under the covers it will use the default loader (or a customized one you've pre-specified, as we'll discuss in the next section).

Just like the tag you'll use in markup, the module loader itself is not specified by ES6. It is a separate, parallel standard (<http://whatwg.github.io/loader/>) controlled currently by the WHATWG browser standards group.

At the time of this writing, the following discussions reflect an early pass at the API design, and things are likely to change.

Loading Modules Outside of Modules

One use for interacting directly with the module loader is if a non-module needs to load a module. Consider:

```
// normal script loaded in browser via `<script>`,  
// `import` is illegal here  
  
Reflect.Loader.import( "foo" ) // returns a promise for `"foo"``  
.then( function(foo){  
    foo.bar();  
} );
```

The `Reflect.Loader.import(..)` utility imports the entire module onto the named parameter (as a namespace), just like the `import * as foo ..` namespace import we discussed earlier.

Note: The `Reflect.Loader.import(..)` utility returns a promise that is fulfilled once the module is ready. To import multiple modules, you can compose promises from multiple `Reflect.Loader.import(..)` calls using `Promise.all([..])`. For more information about Promises, see "Promises" in Chapter 4.

You can also use `Reflect.Loader.import(..)` in a real module to dynamically/conditionally load a module, where `import` itself would not work. You might, for instance, choose to load a module containing a polyfill for some ES7+ feature if a feature test reveals it's not defined by the current engine.

For performance reasons, you'll want to avoid dynamic loading whenever possible, as it hampers the ability of the JS engine to fire off early fetches from its static analysis.

Customized Loading

Another use for directly interacting with the module loader is if you want to customize its behavior through configuration or even redefinition.

At the time of this writing, there's a polyfill for the module loader API being developed (<https://github.com/ModuleLoader/es6-module-loader>). While details are scarce and highly subject to change, we can explore what possibilities may eventually land.

The `Reflect.Loader.import(..)` call may support a second argument for specifying various options to customize the import/load task. For example:

```
Reflect.Loader.import( "foo", { address: "/path/to/foo.js" } )  
.then( function(foo){  
    // ..  
} )
```

It's also expected that a customization will be provided (through some means) for hooking into the process of loading a module, where a translation/transpilation could occur after load but before the engine compiles the module.

For example, you could load something that's not already an ES6-compliant module format (e.g., CoffeeScript, TypeScript, CommonJS, AMD). Your translation step could then convert it to an ES6-compliant module for the engine to then process.

Classes

From nearly the beginning of JavaScript, syntax and development patterns have all strived (read: struggled) to put on a facade of supporting class-oriented development. With things like `new` and `instanceof` and a `.constructor` property, who couldn't help but be teased that JS had classes hidden somewhere inside its prototype system?

Of course, JS "classes" aren't nearly the same as classical classes. The differences are well documented, so I won't belabor that point any further here.

Note: To learn more about the patterns used in JS to fake "classes," and an alternative view of prototypes called "delegation," see the second half of the *this & Object Prototypes* title of this series.

class

Although JS's prototype mechanism doesn't work like traditional classes, that doesn't stop the strong tide of demand on the language to extend the syntactic sugar so that expressing "classes" looks more like real classes. Enter the ES6 `class` keyword and its associated mechanism.

This feature is the result of a highly contentious and drawn-out debate, and represents a smaller subset compromise from several strongly opposed views on how to approach JS classes. Most developers who want full classes in JS will find parts of the new syntax quite inviting, but will find important bits still missing. Don't worry, though. TC39 is already working on additional features to augment classes in the post-ES6 timeframe.

At the heart of the new ES6 class mechanism is the `class` keyword, which identifies a *block* where the contents define the members of a function's prototype. Consider:

```
class Foo {  
    constructor(a,b) {  
        this.x = a;  
        this.y = b;  
    }  
  
    gimmeXY() {  
        return this.x * this.y;  
    }  
}
```

Some things to note:

- `class Foo` implies creating a (special) function of the name `Foo`, much like you did pre-ES6.
- `constructor(...)` identifies the signature of that `Foo(...)` function, as well as its body contents.
- Class methods use the same "concise method" syntax available to object literals, as discussed in Chapter 2. This also includes the concise generator form as discussed earlier in this chapter, as well as the ES5 getter/setter syntax. However, class methods are non-enumerable whereas object methods are by default enumerable.
- Unlike object literals, there are no commas separating members in a `class` body! In fact, they're not even allowed.

The `class` syntax definition in the previous snippet can be roughly thought of as this pre-ES6 equivalent, which probably will look fairly familiar to those who've done prototype-style coding before:

```
function Foo(a,b) {  
    this.x = a;  
    this.y = b;  
}  
  
Foo.prototype.gimmeXY = function() {  
    return this.x * this.y;  
}
```

In either the pre-ES6 form or the new ES6 `class` form, this "class" can now be instantiated and used just as you'd expect:

```
var f = new Foo( 5, 15 );  
  
f.x;                      // 5  
f.y;                      // 15  
f.gimmeXY();               // 75
```

Caution! Though `class Foo` seems much like `function Foo()`, there are important differences:

- A `Foo(..)` call of `class Foo` *must* be made with `new`, as the pre-ES6 option of `Foo.call(obj)` *will not* work.
- While `function Foo` is "hoisted" (see the *Scope & Closures* title of this series), `class Foo` is not; the `extends ..` clause specifies an expression that cannot be "hoisted." So, you must declare a `class` before you can instantiate it.
- `class Foo` in the top global scope creates a lexical `Foo` identifier in that scope, but unlike `function Foo` does not create a global object property of that name.

The established `instanceof` operator still works with ES6 classes, because `class` just creates a constructor function of the same name. However, ES6 introduces a way to customize how `instanceof` works, using `Symbol.hasInstance` (see "Well-Known Symbols" in Chapter 7).

Another way of thinking about `class`, which I find more convenient, is as a *macro* that is used to automatically populate a `prototype` object. Optionally, it also wires up the `[[Prototype]]` relationship if using `extends` (see the next section).

An ES6 `class` isn't really an entity itself, but a meta concept that wraps around other concrete entities, such as functions and properties, and ties them together.

Tip: In addition to the declaration form, a `class` can also be an expression, as in: `var x = class Y { .. }`. This is primarily useful for passing a class definition (technically, the constructor itself) as a function argument or assigning it to an object property.

extends and super

ES6 classes also have syntactic sugar for establishing the `[[Prototype]]` delegation link between two function prototypes -- commonly mislabeled "inheritance" or confusingly labeled "prototype inheritance" -- using the class-oriented familiar terminology `extends`:

```

class Bar extends Foo {
  constructor(a,b,c) {
    super( a, b );
    this.z = c;
  }

  gimmeXYZ() {
    return super.gimmeXY() * this.z;
  }
}

var b = new Bar( 5, 15, 25 );

b.x;                      // 5
b.y;                      // 15
b.z;                      // 25
b.gimmeXYZ();             // 1875

```

A significant new addition is `super`, which is actually something not directly possible pre-ES6 (without some unfortunate hack trade-offs). In the constructor, `super` automatically refers to the "parent constructor," which in the previous example is `Foo(...)`. In a method, it refers to the "parent object," such that you can then make a property/method access off it, such as `super.gimmeXY()`.

`Bar extends Foo` of course means to link the `[[Prototype]]` of `Bar.prototype` to `Foo.prototype`. So, `super` in a method like `gimmeXYZ()` specifically means `Foo.prototype`, whereas `super` means `Foo` when used in the `Bar` constructor.

Note: `super` is not limited to `class` declarations. It also works in object literals, in much the same way we're discussing here. See "Object `super`" in Chapter 2 for more information.

There Be `super` Dragons

It is not insignificant to note that `super` behaves differently depending on where it appears. In fairness, most of the time, that won't be a problem. But surprises await if you deviate from a narrow norm.

There may be cases where in the constructor you would want to reference the `Foo.prototype`, such as to directly access one of its properties/methods. However, `super` in the constructor cannot be used in that way; `super.prototype` will not work. `super(...)` means roughly to call `new Foo(...)`, but isn't actually a usable reference to `Foo` itself.

Symmetrically, you may want to reference the `Foo(...)` function from inside a non-constructor method. `super.constructor` will point at `Foo(...)` the function, but beware that this function can *only* be invoked with `new`. `new super.constructor(..)` would be valid, but

it wouldn't be terribly useful in most cases, because you can't make that call use or reference the current `this` object context, which is likely what you'd want.

Also, `super` looks like it might be driven by a function's context just like `this` -- that is, that they'd both be dynamically bound. However, `super` is not dynamic like `this` is. When a constructor or method makes a `super` reference inside it at declaration time (in the `class` body), that `super` is statically bound to that specific class hierarchy, and cannot be overridden (at least in ES6).

What does that mean? It means that if you're in the habit of taking a method from one "class" and "borrowing" it for another class by overriding its `this`, say with `call(..)` or `apply(..)`, that may very well create surprises if the method you're borrowing has a `super` in it. Consider this class hierarchy:

```
class ParentA {
  constructor() { this.id = "a"; }
  foo() { console.log( "ParentA:", this.id ); }
}

class ParentB {
  constructor() { this.id = "b"; }
  foo() { console.log( "ParentB:", this.id ); }
}

class ChildA extends ParentA {
  foo() {
    super.foo();
    console.log( "ChildA:", this.id );
  }
}

class ChildB extends ParentB {
  foo() {
    super.foo();
    console.log( "ChildB:", this.id );
  }
}

var a = new ChildA();
a.foo(); // ParentA: a
         // ChildA: a
var b = new ChildB();
b.foo(); // ParentB: b
         // ChildB: b
```

All seems fairly natural and expected in this previous snippet. However, if you try to borrow `b.foo()` and use it in the context of `a` -- by virtue of dynamic `this` binding, such borrowing is quite common and used in many different ways, including mixins most notably -- you may find this result an ugly surprise:

```
// borrow `b.foo()` to use in `a` context
b.foo.call( a );           // ParentB: a
                           // ChildB: a
```

As you can see, the `this.id` reference was dynamically rebound so that `: a` is reported in both cases instead of `: b`. But `b.foo()`'s `super.foo()` reference wasn't dynamically rebound, so it still reported `ParentB` instead of the expected `ParentA`.

Because `b.foo()` references `super`, it is statically bound to the `ChildB / ParentB` hierarchy and cannot be used against the `ChildA / ParentA` hierarchy. There is no ES6 solution to this limitation.

`super` seems to work intuitively if you have a static class hierarchy with no cross-pollination. But in all fairness, one of the main benefits of doing `this`-aware coding is exactly that sort of flexibility. Simply, `class + super` requires you to avoid such techniques.

The choice boils down to narrowing your object design to these static hierarchies -- `class`, `extends`, and `super` will be quite nice -- or dropping all attempts to "fake" classes and instead embrace dynamic and flexible, classless objects and `[[Prototype]]` delegation (see the *this & Object Prototypes* title of this series).

Subclass Constructor

Constructors are not required for classes or subclasses; a default constructor is substituted in both cases if omitted. However, the default substituted constructor is different for a direct class versus an extended class.

Specifically, the default subclass constructor automatically calls the parent constructor, and passes along any arguments. In other words, you could think of the default subclass constructor sort of like this:

```
constructor(...args) {
  super(...args);
}
```

This is an important detail to note. Not all class languages have the subclass constructor automatically call the parent constructor. C++ does, but Java does not. But more importantly, in pre-ES6 classes, such automatic "parent constructor" calling does not happen. Be careful when converting to ES6 `class` if you've been relying on such calls *not* happening.

Another perhaps surprising deviation/limitation of ES6 subclass constructors: in a constructor of a subclass, you cannot access `this` until `super(..)` has been called. The reason is nuanced and complicated, but it boils down to the fact that the parent constructor

is actually the one creating/initializing your instance's `this`. Pre-ES6, it works oppositely; the `this` object is created by the "subclass constructor," and then you call a "parent constructor" with the context of the "subclass" `this`.

Let's illustrate. This works pre-ES6:

```
function Foo() {
    this.a = 1;
}

function Bar() {
    this.b = 2;
    Foo.call( this );
}

// `Bar` "extends" `Foo`
Bar.prototype = Object.create( Foo.prototype );
```

But this ES6 equivalent is not allowed:

```
class Foo {
    constructor() { this.a = 1; }
}

class Bar extends Foo {
    constructor() {
        this.b = 2;           // not allowed before `super()`
        super();             // to fix swap these two statements
    }
}
```

In this case, the fix is simple. Just swap the two statements in the subclass `Bar` constructor. However, if you've been relying pre-ES6 on being able to skip calling the "parent constructor," beware because that won't be allowed anymore.

extend ing Natives

One of the most heralded benefits to the new `class` and `extend` design is the ability to (finally!) subclass the built-in natives, like `Array`. Consider:

```

class MyCoolArray extends Array {
    first() { return this[0]; }
    last() { return this[this.length - 1]; }
}

var a = new MyCoolArray( 1, 2, 3 );

a.length;                      // 3
a;                             // [1,2,3]

a.first();                      // 1
a.last();                       // 3

```

Prior to ES6, a fake "subclass" of `Array` using manual object creation and linking to `Array.prototype` only partially worked. It missed out on the special behaviors of a real array, such as the automatically updating `length` property. ES6 subclasses should fully work with "inherited" and augmented behaviors as expected!

Another common pre-ES6 "subclass" limitation is with the `Error` object, in creating custom error "subclasses." When genuine `Error` objects are created, they automatically capture special `stack` information, including the line number and file where the error is created. Pre-ES6 custom error "subclasses" have no such special behavior, which severely limits their usefulness.

ES6 to the rescue:

```

class Oops extends Error {
    constructor(reason) {
        super(reason);
        this.ouch = reason;
    }
}

// later:
var ouch = new Oops( "I messed up!" );
throw ouch;

```

The `ouch` custom error object in this previous snippet will behave like any other genuine error object, including capturing `stack`. That's a big improvement!

new.target

ES6 introduces a new concept called a *meta property* (see Chapter 7), in the form of `new.target`.

If that looks strange, it is; pairing a keyword with a `.` and a property name is definitely an out-of-the-ordinary pattern for JS.

`new.target` is a new "magical" value available in all functions, though in normal functions it will always be `undefined`. In any constructor, `new.target` always points at the constructor that `new` actually directly invoked, even if the constructor is in a parent class and was delegated to by a `super(..)` call from a child constructor. Consider:

```
class Foo {
  constructor() {
    console.log( "Foo: ", new.target.name );
  }
}

class Bar extends Foo {
  constructor() {
    super();
    console.log( "Bar: ", new.target.name );
  }
  baz() {
    console.log( "baz: ", new.target );
  }
}

var a = new Foo();
// Foo: Foo

var b = new Bar();
// Foo: Bar  <-- respects the `new` call-site
// Bar: Bar

b.baz();
// baz: undefined
```

The `new.target` meta property doesn't have much purpose in class constructors, except accessing a static property/method (see the next section).

If `new.target` is `undefined`, you know the function was not called with `new`. You can then force a `new` invocation if that's necessary.

static

When a subclass `Bar` extends a parent class `Foo`, we already observed that `Bar.prototype` is `[[Prototype]]`-linked to `Foo.prototype`. But additionally, `Bar()` is `[[Prototype]]`-linked to `Foo()`. That part may not have such an obvious reasoning.

However, it's quite useful in the case where you declare `static` methods (not just properties) for a class, as these are added directly to that class's function object, not to the function object's `prototype` object. Consider:

```

class Foo {
    static cool() { console.log( "cool" ); }
    wow() { console.log( "wow" ); }
}

class Bar extends Foo {
    static awesome() {
        super.cool();
        console.log( "awesome" );
    }
    neat() {
        super.wow();
        console.log( "neat" );
    }
}

Foo.cool();           // "cool"
Bar.cool();           // "cool"
Bar.awesome();        // "cool"
                     // "awesome"

var b = new Bar();
b.neat();             // "wow"
                     // "neat"

b.awesome;            // undefined
b.cool;               // undefined

```

Be careful not to get confused that `static` members are on the class's prototype chain. They're actually on the dual/parallel chain between the function constructors.

`Symbol.species` Constructor Getter

One place where `static` can be useful is in setting the `Symbol.species` getter (known internally in the specification as `@@species`) for a derived (child) class. This capability allows a child class to signal to a parent class what constructor should be used -- when not intending the child class's constructor itself -- if any parent class method needs to vend a new instance.

For example, many methods on `Array` create and return a new `Array` instance. If you define a derived class from `Array`, but you want those methods to continue to vend actual `Array` instances instead of from your derived class, this works:

```

class MyCoolArray extends Array {
    // force `species` to be parent constructor
    static get [Symbol.species]() { return Array; }
}

var a = new MyCoolArray( 1, 2, 3 ),
    b = a.map( function(v){ return v * 2; } );

b instanceof MyCoolArray;      // false
b instanceof Array;           // true

```

To illustrate how a parent class method can use a child's species declaration somewhat like `Array#map(..)` is doing, consider:

```

class Foo {
    // defer `species` to derived constructor
    static get [Symbol.species]() { return this; }
    spawn() {
        return new this.constructor[Symbol.species]();
    }
}

class Bar extends Foo {
    // force `species` to be parent constructor
    static get [Symbol.species]() { return Foo; }
}

var a = new Foo();
var b = a.spawn();
b instanceof Foo;                  // true

var x = new Bar();
var y = x.spawn();
y instanceof Bar;                 // false
y instanceof Foo;                 // true

```

The parent class `Symbol.species` does `return this` to defer to any derived class, as you'd normally expect. `Bar` then overrides to manually declare `Foo` to be used for such instance creation. Of course, a derived class can still vend instances of itself using `new this.constructor(..)`.

Review

ES6 introduces several new features that aid in code organization:

- Iterators provide sequential access to data or operations. They can be consumed by

new language features like `for..of` and `...`.

- Generators are locally pause/resume capable functions controlled by an iterator. They can be used to programmatically (and interactively, through `yield` / `next(..)` message passing) *generate* values to be consumed via iteration.
- Modules allow private encapsulation of implementation details with a publicly exported API. Module definitions are file-based, singleton instances, and statically resolved at compile time.
- Classes provide cleaner syntax around prototype-based coding. The addition of `super` also solves tricky issues with relative references in the `[[Prototype]]` chain.

These new tools should be your first stop when trying to improve the architecture of your JS projects by embracing ES6.

Chapter 4: Async Flow Control

It's no secret if you've written any significant amount of JavaScript that asynchronous programming is a required skill. The primary mechanism for managing asynchrony has been the function callback.

However, ES6 adds a new feature that helps address significant shortcomings in the callbacks-only approach to async: *Promises*. In addition, we can revisit generators (from the previous chapter) and see a pattern for combining the two that's a major step forward in async flow control programming in JavaScript.

Promises

Let's clear up some misconceptions: Promises are not about replacing callbacks. Promises provide a trustable intermediary -- that is, between your calling code and the async code that will perform the task -- to manage callbacks.

Another way of thinking about a Promise is as an event listener, on which you can register to listen for an event that lets you know when a task has completed. It's an event that will only ever fire once, but it can be thought of as an event nonetheless.

Promises can be chained together, which can sequence a series of asynchronously completing steps. Together with higher-level abstractions like the `all(...)` method (in classic terms, a "gate") and the `race(...)` method (in classic terms, a "latch"), promise chains provide a mechanism for async flow control.

Yet another way of conceptualizing a Promise is that it's a *future value*, a time-independent container wrapped around a value. This container can be reasoned about identically whether the underlying value is final or not. Observing the resolution of a Promise extracts this value once available. In other words, a Promise is said to be the async version of a sync function's return value.

A Promise can only have one of two possible resolution outcomes: fulfilled or rejected, with an optional single value. If a Promise is fulfilled, the final value is called a fulfillment. If it's rejected, the final value is called a reason (as in, a "reason for rejection"). Promises can only be resolved (fulfillment or rejection) *once*. Any further attempts to fulfill or reject are simply ignored. Thus, once a Promise is resolved, it's an immutable value that cannot be changed.

Clearly, there are several different ways to think about what a Promise is. No single perspective is fully sufficient, but each provides a separate aspect of the whole. The big takeaway is that they offer a significant improvement over callbacks-only async, namely that they provide order, predictability, and trustability.

Making and Using Promises

To construct a promise instance, use the `Promise(..)` constructor:

```
var p = new Promise( function pr(resolve,reject){
    // ...
} );
```

The `Promise(..)` constructor takes a single function (`pr(..)`), which is called immediately and receives two control functions as arguments, usually named `resolve(..)` and `reject(..)`. They are used as:

- If you call `reject(..)`, the promise is rejected, and if any value is passed to `reject(..)`, it is set as the reason for rejection.
- If you call `resolve(..)` with no value, or any non-promise value, the promise is fulfilled.
- If you call `resolve(..)` and pass another promise, this promise simply adopts the state -- whether immediate or eventual -- of the passed promise (either fulfillment or rejection).

Here's how you'd typically use a promise to refactor a callback-reliant function call. If you start out with an `ajax(..)` utility that expects to be able to call an error-first style callback:

```
function ajax(url,cb) {
    // make request, eventually call `cb(..)`
}

// ...

ajax( "http://some.url.1", function handler(err,contents){
    if (err) {
        // handle ajax error
    }
    else {
        // handle `contents` success
    }
});
```

You can convert it to:

```

function ajax(url) {
  return new Promise( function pr(resolve,reject){
    // make request, eventually call
    // either `resolve(..)` or `reject(..)`
  } );
}

// ...

ajax( "http://some.url.1" )
.then(
  function fulfilled(contents){
    // handle `contents` success
  },
  function rejected(reason){
    // handle ajax error reason
  }
);

```

Promises have a `then(...)` method that accepts one or two callback functions. The first function (if present) is treated as the handler to call if the promise is fulfilled successfully. The second function (if present) is treated as the handler to call if the promise is rejected explicitly, or if any error/exception is caught during resolution.

If one of the arguments is omitted or otherwise not a valid function -- typically you'll use `null` instead -- a default placeholder equivalent is used. The default success callback passes its fulfillment value along and the default error callback propagates its rejection reason along.

The shorthand for calling `then(null,handleRejection)` is `catch(handleRejection)`.

Both `then(...)` and `catch(...)` automatically construct and return another promise instance, which is wired to receive the resolution from whatever the return value is from the original promise's fulfillment or rejection handler (whichever is actually called). Consider:

```

ajax( "http://some.url.1" )
.then(
  function fulfilled(contents){
    return contents.toUpperCase();
  },
  function rejected(reason){
    return "DEFAULT VALUE";
  }
)
.then( function fulfilled(data){
  // handle data from original promise's
  // handlers
} );

```

In this snippet, we're returning an immediate value from either `fulfilled(..)` or `rejected(..)`, which then is received on the next event turn in the second `then(..)`'s `fulfilled(..)`. If we instead return a new promise, that new promise is subsumed and adopted as the resolution:

```
ajax( "http://some.url.1" )
  .then(
    function fulfilled(contents){
      return ajax(
        "http://some.url.2?v=" + contents
      );
    },
    function rejected(reason){
      return ajax(
        "http://backup.url.3?err=" + reason
      );
    }
  )
  .then( function fulfilled(contents){
    // `contents` comes from the subsequent
    // `ajax(..)` call, whichever it was
  } );
}
```

It's important to note that an exception (or rejected promise) in the first `fulfilled(..)` will *not* result in the first `rejected(..)` being called, as that handler only responds to the resolution of the first original promise. Instead, the second promise, which the second `then(..)` is called against, receives that rejection.

In this previous snippet, we are not listening for that rejection, which means it will be silently held onto for future observation. If you never observe it by calling a `then(..)` or `catch(..)`, then it will go unhandled. Some browser developer consoles may detect these unhandled rejections and report them, but this is not reliably guaranteed; you should always observe promise rejections.

Note: This was just a brief overview of Promise theory and behavior. For a much more in-depth exploration, see Chapter 3 of the *Async & Performance* title of this series.

Thenables

Promises are genuine instances of the `Promise(..)` constructor. However, there are promise-like objects called *thenables* that generally can interoperate with the Promise mechanisms.

Any object (or function) with a `then(..)` function on it is assumed to be a thenable. Any place where the Promise mechanisms can accept and adopt the state of a genuine promise, they can also handle a thenable.

Thenables are basically a general label for any promise-like value that may have been created by some other system than the actual `Promise(..)` constructor. In that perspective, a thenable is generally less trustable than a genuine Promise. Consider this misbehaving thenable, for example:

```
var th = {
  then: function thener( fulfilled ) {
    // call `fulfilled(..)` once every 100ms forever
    setInterval( fulfilled, 100 );
  }
};
```

If you received that thenable and chained it with `th.then(..)`, you'd likely be surprised that your fulfillment handler is called repeatedly, when normal Promises are supposed to only ever be resolved once.

Generally, if you're receiving what purports to be a promise or thenable back from some other system, you shouldn't just trust it blindly. In the next section, we'll see a utility included with ES6 Promises that helps address this trust concern.

But to further understand the perils of this issue, consider that *any* object in *any* piece of code that's ever been defined to have a method on it called `then(..)` can be potentially confused as a thenable -- if used with Promises, of course -- regardless of if that thing was ever intended to even remotely be related to Promise-style async coding.

Prior to ES6, there was never any special reservation made on methods called `then(..)`, and as you can imagine there's been at least a few cases where that method name has been chosen prior to Promises ever showing up on the radar screen. The most likely case of mistaken thenable will be async libraries that use `then(..)` but which are not strictly Promises-compliant -- there are several out in the wild.

The onus will be on you to guard against directly using values with the Promise mechanism that would be incorrectly assumed to be a thenable.

Promise API

The `Promise` API also provides some static methods for working with Promises.

`Promise.resolve(..)` creates a promise resolved to the value passed in. Let's compare how it works to the more manual approach:

```
var p1 = Promise.resolve( 42 );

var p2 = new Promise( function pr(resolve){
    resolve( 42 );
} );
```

`p1` and `p2` will have essentially identical behavior. The same goes for resolving with a promise:

```
var theP = ajax( .. );

var p1 = Promise.resolve( theP );

var p2 = new Promise( function pr(resolve){
    resolve( theP );
} );
```

Tip: `Promise.resolve(..)` is the solution to the thenable trust issue raised in the previous section. Any value that you are not already certain is a trustable promise -- even if it could be an immediate value -- can be normalized by passing it to `Promise.resolve(..)`. If the value is already a recognizable promise or thenable, its state/resolution will simply be adopted, insulating you from misbehavior. If it's instead an immediate value, it will be "wrapped" in a genuine promise, thereby normalizing its behavior to be async.

`Promise.reject(..)` creates an immediately rejected promise, the same as its `Promise(..)` constructor counterpart:

```
var p1 = Promise.reject( "Oops" );

var p2 = new Promise( function pr(resolve, reject){
    reject( "Oops" );
} );
```

While `resolve(..)` and `Promise.resolve(..)` can accept a promise and adopt its state/resolution, `reject(..)` and `Promise.reject(..)` do not differentiate what value they receive. So, if you reject with a promise or thenable, the promise/thenable itself will be set as the rejection reason, not its underlying value.

`Promise.all([..])` accepts an array of one or more values (e.g., immediate values, promises, thenables). It returns a promise back that will be fulfilled if all the values fulfill, or reject immediately once the first of any of them rejects.

Starting with these values/promises:

```

var p1 = Promise.resolve( 42 );
var p2 = new Promise( function pr(resolve){
    setTimeout( function(){
        resolve( 43 );
    }, 100 );
} );
var v3 = 44;
var p4 = new Promise( function pr(resolve,reject){
    setTimeout( function(){
        reject( "Oops" );
    }, 10 );
} );

```

Let's consider how `Promise.all([..])` works with combinations of those values:

```

Promise.all( [p1,p2,v3] )
.then( function fulfilled(vals){
    console.log( vals );           // [42,43,44]
} );

Promise.all( [p1,p2,v3,p4] )
.then(
    function fulfilled(vals){
        // never gets here
    },
    function rejected(reason){
        console.log( reason );      // Oops
    }
);

```

While `Promise.all([..])` waits for all fulfillments (or the first rejection), `Promise.race([..])` waits only for either the first fulfillment or rejection. Consider:

```
// NOTE: re-setup all test values to
// avoid timing issues misleading you!

Promise.race( [p2,p1,v3] )
.then( function fulfilled(val){
    console.log( val );           // 42
} );

Promise.race( [p2,p4] )
.then(
    function fulfilled(val){
        // never gets here
    },
    function rejected(reason){
        console.log( reason );      // Oops
    }
);

```

Warning: While `Promise.all([])` will fulfill right away (with no values), `Promise.race([])` will hang forever. This is a strange inconsistency, and speaks to the suggestion that you should never use these methods with empty arrays.

Generators + Promises

It *is* possible to express a series of promises in a chain to represent the async flow control of your program. Consider:

```
step1()
.then(
    step2,
    step1Failed
)
.then(
    function step3(msg) {
        return Promise.all( [
            step3a( msg ),
            step3b( msg ),
            step3c( msg )
        ] )
    }
)
.then(step4);
```

However, there's a much better option for expressing async flow control, and it will probably be much more preferable in terms of coding style than long promise chains. We can use what we learned in Chapter 3 about generators to express our async flow control.

The important pattern to recognize: a generator can yield a promise, and that promise can then be wired to resume the generator with its fulfillment value.

Consider the previous snippet's async flow control expressed with a generator:

```
function *main() {  
  
    try {  
        var ret = yield step1();  
    }  
    catch (err) {  
        ret = yield step1Failed( err );  
    }  
  
    ret = yield step2( ret );  
  
    // step 3  
    ret = yield Promise.all( [  
        step3a( ret ),  
        step3b( ret ),  
        step3c( ret )  
    ] );  
  
    yield step4( ret );  
}
```

On the surface, this snippet may seem more verbose than the promise chain equivalent in the earlier snippet. However, it offers a much more attractive -- and more importantly, a more understandable and reason-able -- synchronous-looking coding style (with `=` assignment of "return" values, etc.) That's especially true in that `try..catch` error handling can be used across those hidden async boundaries.

Why are we using Promises with the generator? It's certainly possible to do async generator coding without Promises.

Promises are a trustable system that uninverts the inversion of control of normal callbacks or thunks (see the *Async & Performance* title of this series). So, combining the trustability of Promises and the synchronicity of code in generators effectively addresses all the major deficiencies of callbacks. Also, utilities like `Promise.all([..])` are a nice, clean way to express concurrency at a generator's single `yield` step.

So how does this magic work? We're going to need a *runner* that can run our generator, receive a `yield` ed promise, and wire it up to resume the generator with either the fulfillment success value, or throw an error into the generator with the rejection reason.

Many async-capable utilities/libraries have such a "runner"; for example, `Q.spawn(..)` and my `asynquence`'s `runner(..)` plug-in. But here's a stand-alone runner to illustrate how the process works:

```
function run(gen) {
    var args = [].slice.call( arguments, 1), it;

    it = gen.apply( this, args );

    return Promise.resolve()
        .then( function handleNext(value){
            var next = it.next( value );

            return (function handleResult(next){
                if (next.done) {
                    return next.value;
                }
                else {
                    return Promise.resolve( next.value )
                        .then(
                            handleNext,
                            function handleErr(err) {
                                return Promise.resolve(
                                    it.throw( err )
                                )
                                .then( handleResult );
                            }
                        );
                }
            })( next );
        } );
}
```

Note: For a more prolifically commented version of this utility, see the *Async & Performance* title of this series. Also, the run utilities provided with various async libraries are often more powerful/capable than what we've shown here. For example, `asynquence`'s `runner(..)` can handle `yield`ed promises, sequences, thunks, and immediate (non-promise) values, giving you ultimate flexibility.

So now running `*main()` as listed in the earlier snippet is as easy as:

```
run( main )
.then(
  function fulfilled(){
    // `main()` completed successfully
  },
  function rejected(reason){
    // Oops, something went wrong
  }
);
```

Essentially, anywhere that you have more than two asynchronous steps of flow control logic in your program, you can *and should* use a promise-yielding generator driven by a run utility to express the flow control in a synchronous fashion. This will make for much easier to understand and maintain code.

This yield-a-promise-resume-the-generator pattern is going to be so common and so powerful, the next version of JavaScript after ES6 is almost certainly going to introduce a new function type that will do it automatically without needing the run utility. We'll cover `async function s` (as they're expected to be called) in Chapter 8.

Review

As JavaScript continues to mature and grow in its widespread adoption, asynchronous programming is more and more of a central concern. Callbacks are not fully sufficient for these tasks, and totally fall down the more sophisticated the need.

Thankfully, ES6 adds Promises to address one of the major shortcomings of callbacks: lack of trust in predictable behavior. Promises represent the future completion value from a potentially `async` task, normalizing behavior across sync and `async` boundaries.

But it's the combination of Promises with generators that fully realizes the benefits of rearranging our `async` flow control code to de-emphasize and abstract away that ugly callback soup (aka "hell").

Right now, we can manage these interactions with the aide of various `async` libraries' runners, but JavaScript is eventually going to support this interaction pattern with dedicated syntax alone!

Chapter 5: Collections

Structured collection and access to data is a critical component of just about any JS program. From the beginning of the language up to this point, the array and the object have been our primary mechanism for creating data structures. Of course, many higher-level data structures have been built on top of these, as user-land libraries.

As of ES6, some of the most useful (and performance-optimizing!) data structure abstractions have been added as native components of the language.

We'll start this chapter first by looking at *TypedArrays*, technically contemporary to ES5 efforts several years ago, but only standardized as companions to WebGL and not JavaScript itself. As of ES6, these have been adopted directly by the language specification, which gives them first-class status.

Maps are like objects (key/value pairs), but instead of just a string for the key, you can use any value -- even another object or map! Sets are similar to arrays (lists of values), but the values are unique; if you add a duplicate, it's ignored. There are also weak (in relation to memory/garbage collection) counterparts: WeakMap and WeakSet.

TypedArrays

As we cover in the *Types & Grammar* title of this series, JS does have a set of built-in types, like `number` and `string`. It'd be tempting to look at a feature named "typed array" and assume it means an array of a specific type of values, like an array of only strings.

However, typed arrays are really more about providing structured access to binary data using array-like semantics (indexed access, etc.). The "type" in the name refers to a "view" layered on type of the bucket of bits, which is essentially a mapping of whether the bits should be viewed as an array of 8-bit signed integers, 16-bit signed integers, and so on.

How do you construct such a bit-bucket? It's called a "buffer," and you construct it most directly with the `ArrayBuffer(...)` constructor:

```
var buf = new ArrayBuffer( 32 );
buf.byteLength; // 32
```

`buf` is now a binary buffer that is 32-bytes long (256-bits), that's pre-initialized to all `0`s. A buffer by itself doesn't really allow you any interaction except for checking its `byteLength` property.

Tip: Several web platform features use or return array buffers, such as

`FileReader#readAsArrayBuffer(..)`, `XMLHttpRequest#send(..)`, and `ImageData` (canvas data).

But on top of this array buffer, you can then layer a "view," which comes in the form of a typed array. Consider:

```
var arr = new Uint16Array( buf );
arr.length;                                // 16
```

`arr` is a typed array of 16-bit unsigned integers mapped over the 256-bit `buf` buffer, meaning you get 16 elements.

Endianness

It's very important to understand that the `arr` is mapped using the endian-setting (big-endian or little-endian) of the platform the JS is running on. This can be an issue if the binary data is created with one endianness but interpreted on a platform with the opposite endianness.

Endian means if the low-order byte (collection of 8-bits) of a multi-byte number -- such as the 16-bit unsigned ints we created in the earlier snippet -- is on the right or the left of the number's bytes.

For example, let's imagine the base-10 number `3085`, which takes 16-bits to represent. If you have just one 16-bit number container, it'd be represented in binary as

`0000110000001101` (hexadecimal `0c0d`) regardless of endianness.

But if `3085` was represented with two 8-bit numbers, the endianness would significantly affect its storage in memory:

- `0000110000001101` / `0c0d` (big endian)
- `0000110100001100` / `0d0c` (little endian)

If you received the bits of `3085` as `0000110100001100` from a little-endian system, but you layered a view on top of it in a big-endian system, you'd instead see value `3340` (base-10) and `0d0c` (base-16).

Little endian is the most common representation on the web these days, but there are definitely browsers where that's not true. It's important that you understand the endianness of both the producer and consumer of a chunk of binary data.

From MDN, here's a quick way to test the endianness of your JavaScript:

```
var littleEndian = (function() {
  var buffer = new ArrayBuffer( 2 );
  new DataView( buffer ).setInt16( 0, 256, true );
  return new Int16Array( buffer )[0] === 256;
})();
```

`littleEndian` will be `true` or `false`; for most browsers, it should return `true`. This test uses `DataView(...)`, which allows more low-level, fine-grained control over accessing (setting/getting) the bits from the view you layer over the buffer. The third parameter of the `setInt16(...)` method in the previous snippet is for telling the `DataView` what endianness you're wanting it to use for that operation.

Warning: Do not confuse endianness of underlying binary storage in array buffers with how a given number is represented when exposed in a JS program. For example, `(3085).toString(2)` returns `"110000001101"`, which with an assumed leading four `"0"`s appears to be the big-endian representation. In fact, this representation is based on a single 16-bit view, not a view of two 8-bit bytes. The `DataView` test above is the best way to determine endianness for your JS environment.

Multiple Views

A single buffer can have multiple views attached to it, such as:

```
var buf = new ArrayBuffer( 2 );

var view8 = new Uint8Array( buf );
var view16 = new Uint16Array( buf );

view16[0] = 3085;                                // 13
view8[0];                                         // 12
view8[1];                                         // 12

view8[0].toString( 16 );                         // "d"
view8[1].toString( 16 );                         // "c"

// swap (as if endian!)
var tmp = view8[0];
view8[0] = view8[1];
view8[1] = tmp;

view16[0];                                     // 3340
```

The typed array constructors have multiple signature variations. We've shown so far only passing them an existing buffer. However, that form also takes two extra parameters: `byteOffset` and `length`. In other words, you can start the typed array view at a location

other than `0` and you can make it span less than the full length of the buffer.

If the buffer of binary data includes data in non-uniform size/location, this technique can be quite useful.

For example, consider a binary buffer that has a 2-byte number (aka "word") at the beginning, followed by two 1-byte numbers, followed by a 32-bit floating point number. Here's how you can access that data with multiple views on the same buffer, offsets, and lengths:

```
var first = new Uint16Array( buf, 0, 2 )[0],
    second = new Uint8Array( buf, 2, 1 )[0],
    third = new Uint8Array( buf, 3, 1 )[0],
    fourth = new Float32Array( buf, 4, 4 )[0];
```

TypedArray Constructors

In addition to the `(buffer, [offset, [length]])` form examined in the previous section, typed array constructors also support these forms:

- [constructor] `(length)` : Creates a new view over a new buffer of `length` bytes
- [constructor] `(typedArr)` : Creates a new view and buffer, and copies the contents from the `typedArr` view
- [constructor] `(obj)` : Creates a new view and buffer, and iterates over the array-like or object `obj` to copy its contents

The following typed array constructors are available as of ES6:

- `Int8Array` (8-bit signed integers), `Uint8Array` (8-bit unsigned integers)
 - `Uint8ClampedArray` (8-bit unsigned integers, each value clamped on setting to the `0 - 255` range)
- `Int16Array` (16-bit signed integers), `Uint16Array` (16-bit unsigned integers)
- `Int32Array` (32-bit signed integers), `Uint32Array` (32-bit unsigned integers)
- `Float32Array` (32-bit floating point, IEEE-754)
- `Float64Array` (64-bit floating point, IEEE-754)

Instances of typed array constructors are almost the same as regular native arrays. Some differences include having a fixed length and the values all being of the same "type."

However, they share most of the same `prototype` methods. As such, you likely will be able to use them as regular arrays without needing to convert.

For example:

```

var a = new Int32Array( 3 );
a[0] = 10;
a[1] = 20;
a[2] = 30;

a.map( function(v){
  console.log( v );
} );
// 10 20 30

a.join( "-" );
// "10-20-30"

```

Warning: You can't use certain `Array.prototype` methods with TypedArrays that don't make sense, such as the mutators (`splice(..)`, `push(..)`, etc.) and `concat(..)`.

Be aware that the elements in TypedArrays really are constrained to the declared bit sizes. If you have a `Uint8Array` and try to assign something larger than an 8-bit value into one of its elements, the value wraps around so as to stay within the bit length.

This could cause problems if you were trying to, for instance, square all the values in a TypedArray. Consider:

```

var a = new Uint8Array( 3 );
a[0] = 10;
a[1] = 20;
a[2] = 30;

var b = a.map( function(v){
  return v * v;
} );

b;           // [100, 144, 132]

```

The `20` and `30` values, when squared, resulted in bit overflow. To get around such a limitation, you can use the `TypedArray#from(..)` function:

```

var a = new Uint8Array( 3 );
a[0] = 10;
a[1] = 20;
a[2] = 30;

var b = Uint16Array.from( a, function(v){
  return v * v;
} );

b;           // [100, 400, 900]

```

See the "`Array.from(..)` Static Function" section in Chapter 6 for more information about the `Array.from(..)` that is shared with TypedArrays. Specifically, the "Mapping" section explains the mapping function accepted as its second argument.

One interesting behavior to consider is that TypedArrays have a `sort(..)` method much like regular arrays, but this one defaults to numeric sort comparisons instead of coercing values to strings for lexicographic comparison. For example:

```
var a = [ 10, 1, 2, ];
a.sort();                                // [1,10,2]

var b = new Uint8Array( [ 10, 1, 2 ] );
b.sort();                                // [1,2,10]
```

The `TypedArray#sort(..)` takes an optional compare function argument just like `Array#sort(..)`, which works in exactly the same way.

Maps

If you have a lot of JS experience, you know that objects are the primary mechanism for creating unordered key/value-pair data structures, otherwise known as maps. However, the major drawback with objects-as-maps is the inability to use a non-string value as the key.

For example, consider:

```
var m = {};

var x = { id: 1 },
y = { id: 2 };

m[x] = "foo";
m[y] = "bar";

m[x];                         // "bar"
m[y];                         // "bar"
```

What's going on here? The two objects `x` and `y` both stringify to `"[object Object]"`, so only that one key is being set in `m`.

Some have implemented fake maps by maintaining a parallel array of non-string keys alongside an array of the values, such as:

```

var keys = [], vals = [];

var x = { id: 1 },
    y = { id: 2 };

keys.push( x );
vals.push( "foo" );

keys.push( y );
vals.push( "bar" );

keys[0] === x;                      // true
vals[0];                            // "foo"

keys[1] === y;                      // true
vals[1];                            // "bar"

```

Of course, you wouldn't want to manage those parallel arrays yourself, so you could define a data structure with methods that automatically do the management under the covers. Besides having to do that work yourself, the main drawback is that access is no longer O(1) time-complexity, but instead is O(n).

But as of ES6, there's no longer any need to do this! Just use `Map(..)`:

```

var m = new Map();

var x = { id: 1 },
    y = { id: 2 };

m.set( x, "foo" );
m.set( y, "bar" );

m.get( x );                         // "foo"
m.get( y );                         // "bar"

```

The only drawback is that you can't use the `[]` bracket access syntax for setting and retrieving values. But `get(..)` and `set(..)` work perfectly suitably instead.

To delete an element from a map, don't use the `delete` operator, but instead use the `delete(..)` method:

```

m.set( x, "foo" );
m.set( y, "bar" );

m.delete( y );

```

You can clear the entire map's contents with `clear()`. To get the length of a map (i.e., the number of keys), use the `size` property (not `length`):

```
m.set( x, "foo" );
m.set( y, "bar" );
m.size;                                // 2

m.clear();
m.size;                                // 0
```

The `Map(...)` constructor can also receive an iterable (see "Iterators" in Chapter 3), which must produce a list of arrays, where the first item in each array is the key and the second item is the value. This format for iteration is identical to that produced by the `entries()` method, explained in the next section. That makes it easy to make a copy of a map:

```
var m2 = new Map( m.entries() );

// same as:
var m2 = new Map( m );
```

Because a map instance is an iterable, and its default iterator is the same as `entries()`, the second shorter form is more preferable.

Of course, you can just manually specify an `entries` list (array of key/value arrays) in the `Map(...)` constructor form:

```
var x = { id: 1 },
y = { id: 2 };

var m = new Map( [
  [ x, "foo" ],
  [ y, "bar" ]
] );

m.get( x );                            // "foo"
m.get( y );                            // "bar"
```

Map Values

To get the list of values from a map, use `values(...)`, which returns an iterator. In Chapters 2 and 3, we covered various ways to process an iterator sequentially (like an array), such as the `...` spread operator and the `for..of` loop. Also, "Arrays" in Chapter 6 covers the `Array.from(...)` method in detail. Consider:

```

var m = new Map();

var x = { id: 1 },
    y = { id: 2 };

m.set( x, "foo" );
m.set( y, "bar" );

var vals = [ ...m.values() ];

vals;                                // ["foo","bar"]
Array.from( m.values() );            // ["foo","bar"]

```

As discussed in the previous section, you can iterate over a map's entries using `entries()` (or the default map iterator). Consider:

```

var m = new Map();

var x = { id: 1 },
    y = { id: 2 };

m.set( x, "foo" );
m.set( y, "bar" );

var vals = [ ...m.entries() ];

vals[0][0] === x;                      // true
vals[0][1];                           // "foo"

vals[1][0] === y;                      // true
vals[1][1];                           // "bar"

```

Map Keys

To get the list of keys, use `keys()`, which returns an iterator over the keys in the map:

```

var m = new Map();

var x = { id: 1 },
    y = { id: 2 };

m.set( x, "foo" );
m.set( y, "bar" );

var keys = [ ...m.keys() ];

keys[0] === x;                         // true
keys[1] === y;                         // true

```

To determine if a map has a given key, use `has(...)` :

```
var m = new Map();

var x = { id: 1 },
    y = { id: 2 };

m.set( x, "foo" );

m.has( x );           // true
m.has( y );           // false
```

Maps essentially let you associate some extra piece of information (the value) with an object (the key) without actually putting that information on the object itself.

While you can use any kind of value as a key for a map, you typically will use objects, as strings and other primitives are already eligible as keys of normal objects. In other words, you'll probably want to continue to use normal objects for maps unless some or all of the keys need to be objects, in which case map is more appropriate.

Warning: If you use an object as a map key and that object is later discarded (all references unset) in attempt to have garbage collection (GC) reclaim its memory, the map itself will still retain its entry. You will need to remove the entry from the map for it to be GC-eligible. In the next section, we'll see WeakMaps as a better option for object keys and GC.

WeakMaps

WeakMaps are a variation on maps, which has most of the same external behavior but differs underneath in how the memory allocation (specifically its GC) works.

WeakMaps take (only) objects as keys. Those objects are held *weakly*, which means if the object itself is GC'd, the entry in the WeakMap is also removed. This isn't observable behavior, though, as the only way an object can be GC'd is if there's no more references to it -- once there are no more references to it, you have no object reference to check if it exists in the WeakMap.

Otherwise, the API for WeakMap is similar, though more limited:

```
var m = new WeakMap();

var x = { id: 1 },
    y = { id: 2 };

m.set( x, "foo" );

m.has( x );                      // true
m.has( y );                      // false
```

WeakMaps do not have a `size` property or `clear()` method, nor do they expose any iterators over their keys, values, or entries. So even if you unset the `x` reference, which will remove its entry from `m` upon GC, there is no way to tell. You'll just have to take JavaScript's word for it!

Just like Maps, WeakMaps let you soft-associate information with an object. But they are particularly useful if the object is not one you completely control, such as a DOM element. If the object you're using as a map key can be deleted and should be GC-eligible when it is, then a WeakMap is a more appropriate option.

It's important to note that a WeakMap only holds its *keys* weakly, not its values. Consider:

```
var m = new WeakMap();

var x = { id: 1 },
    y = { id: 2 },
    z = { id: 3 },
    w = { id: 4 };

m.set( x, y );

x = null;                         // { id: 1 } is GC-eligible
y = null;                          // { id: 2 } is GC-eligible
                                    // only because { id: 1 } is

m.set( z, w );

w = null;                          // { id: 4 } is not GC-eligible
```

For this reason, WeakMaps are in my opinion better named "WeakKeyMaps."

Sets

A set is a collection of unique values (duplicates are ignored).

The API for a set is similar to map. The `add(..)` method takes the place of the `set(..)` method (somewhat ironically), and there is no `get(..)` method.

Consider:

```
var s = new Set();

var x = { id: 1 },
y = { id: 2 };

s.add( x );
s.add( y );
s.add( x );

s.size;                      // 2

s.delete( y );
s.size;                      // 1

s.clear();
s.size;                      // 0
```

The `Set(..)` constructor form is similar to `Map(..)`, in that it can receive an iterable, like another set or simply an array of values. However, unlike how `Map(..)` expects *entries* list (array of key/value arrays), `Set(..)` expects a *values* list (array of values):

```
var x = { id: 1 },
y = { id: 2 };

var s = new Set( [x,y] );
```

A set doesn't need a `get(..)` because you don't retrieve a value from a set, but rather test if it is present or not, using `has(..)`:

```
var s = new Set();

var x = { id: 1 },
y = { id: 2 };

s.add( x );

s.has( x );                  // true
s.has( y );                  // false
```

Note: The comparison algorithm in `has(..)` is almost identical to `Object.is(..)` (see Chapter 6), except that `-0` and `0` are treated as the same rather than distinct.

Set Iterators

Sets have the same iterator methods as maps. Their behavior is different for sets, but symmetric with the behavior of map iterators. Consider:

```
var s = new Set();

var x = { id: 1 },
    y = { id: 2 };

s.add( x ).add( y );

var keys = [ ...s.keys() ],
    vals = [ ...s.values() ],
    entries = [ ...s.entries() ];

keys[0] === x;
keys[1] === y;

vals[0] === x;
vals[1] === y;

entries[0][0] === x;
entries[0][1] === x;
entries[1][0] === y;
entries[1][1] === y;
```

The `keys()` and `values()` iterators both yield a list of the unique values in the set. The `entries()` iterator yields a list of entry arrays, where both items of the array are the unique set value. The default iterator for a set is its `values()` iterator.

The inherent uniqueness of a set is its most useful trait. For example:

```
var s = new Set( [1, 2, 3, 4, "1", 2, 4, "5" ] ),
    uniques = [ ...s ];

uniques;                                // [1, 2, 3, 4, "1", "5"]
```

Set uniqueness does not allow coercion, so `1` and `"1"` are considered distinct values.

WeakSets

Whereas a WeakMap holds its keys weakly (but its values strongly), a WeakSet holds its values weakly (there aren't really keys).

```
var s = new WeakSet();

var x = { id: 1 },
    y = { id: 2 };

s.add( x );
s.add( y );

x = null;                      // `x` is GC-eligible
y = null;                      // `y` is GC-eligible
```

Warning: WeakSet values must be objects, not primitive values as is allowed with sets.

Review

ES6 defines a number of useful collections that make working with data in structured ways more efficient and effective.

TypedArrays provide "view"s of binary data buffers that align with various integer types, like 8-bit unsigned integers and 32-bit floats. The array access to binary data makes operations much easier to express and maintain, which enables you to more easily work with complex data like video, audio, canvas data, and so on.

Maps are key-value pairs where the key can be an object instead of just a string/primitive.
Sets are unique lists of values (of any type).

WeakMaps are maps where the key (object) is weakly held, so that GC is free to collect the entry if it's the last reference to an object. WeakSets are sets where the value is weakly held, again so that GC can remove the entry if it's the last reference to that object.

Chapter 6: API Additions

From conversions of values to mathematic calculations, ES6 adds many static properties and methods to various built-in natives and objects to help with common tasks. In addition, instances of some of the natives have new capabilities via various new prototype methods.

Note: Most of these features can be faithfully polyfilled. We will not dive into such details here, but check out "ES6 Shim" (<https://github.com/paulmillr/es6-shim/>) for standards-compliant shims/polyfills.

Array

One of the most commonly extended features in JS by various user libraries is the Array type. It should be no surprise that ES6 adds a number of helpers to Array, both static and prototype (instance).

Array.of(. .) Static Function

There's a well known gotcha with the `Array(...)` constructor, which is that if there's only one argument passed, and that argument is a number, instead of making an array of one element with that number value in it, it constructs an empty array with a `length` property equal to the number. This action produces the unfortunate and quirky "empty slots" behavior that's reviled about JS arrays.

`Array.of(...)` replaces `Array(...)` as the preferred function-form constructor for arrays, because `Array.of(...)` does not have that special single-number-argument case. Consider:

```
var a = Array( 3 );
a.length;                      // 3
a[0];                          // undefined

var b = Array.of( 3 );
b.length;                      // 1
b[0];                          // 3

var c = Array.of( 1, 2, 3 );
c.length;                      // 3
c;                            // [1,2,3]
```

Under what circumstances would you want to use `Array.of(...)` instead of just creating an array with literal syntax, like `c = [1,2,3]`? There's two possible cases.

If you have a callback that's supposed to wrap argument(s) passed to it in an array, `Array.of(...)` fits the bill perfectly. That's probably not terribly common, but it may scratch an itch for you.

The other scenario is if you subclass `Array` (see "Classes" in Chapter 3) and want to be able to create and initialize elements in an instance of your subclass, such as:

```
class MyCoolArray extends Array {
    sum() {
        return this.reduce( function reducer(acc,curr){
            return acc + curr;
        }, 0 );
    }
}

var x = new MyCoolArray( 3 );
x.length;                      // 3 -- oops!
x.sum();                        // 0 -- oops!

var y = [3];                   // Array, not MyCoolArray
y.length;                      // 1
y.sum();                        // `sum` is not a function

var z = MyCoolArray.of( 3 );
z.length;                      // 1
z.sum();                        // 3
```

You can't just (easily) create a constructor for `MyCoolArray` that overrides the behavior of the `Array` parent constructor, because that constructor is necessary to actually create a well-behaving array value (initializing the `this`). The "inherited" static `of(...)` method on the `MyCoolArray` subclass provides a nice solution.

Array.from(..) Static Function

An "array-like object" in JavaScript is an object that has a `length` property on it, specifically with an integer value of zero or higher.

These values have been notoriously frustrating to work with in JS; it's been quite common to need to transform them into an actual array, so that the various `Array.prototype` methods (`map(..)`, `indexOf(..)` etc.) are available to use with it. That process usually looks like:

```
// array-like object
var arrLike = {
  length: 3,
  0: "foo",
  1: "bar"
};

var arr = Array.prototype.slice.call( arrLike );
```

Another common task where `slice(..)` is often used is in duplicating a real array:

```
var arr2 = arr.slice();
```

In both cases, the new ES6 `Array.from(..)` method can be a more understandable and graceful -- if also less verbose -- approach:

```
var arr = Array.from( arrLike );
var arrCopy = Array.from( arr );
```

`Array.from(..)` looks to see if the first argument is an iterable (see "Iterators" in Chapter 3), and if so, it uses the iterator to produce values to "copy" into the returned array. Because real arrays have an iterator for those values, that iterator is automatically used.

But if you pass an array-like object as the first argument to `Array.from(..)`, it behaves basically the same as `slice()` (no arguments!) or `apply(..)` does, which is that it simply loops over the value, accessing numerically named properties from `0` up to whatever the value of `length` is.

Consider:

```
var arrLike = {
  length: 4,
  2: "foo"
};

Array.from( arrLike );
// [ undefined, undefined, "foo", undefined ]
```

Because positions `0`, `1`, and `3` didn't exist on `arrLike`, the result was the `undefined` value for each of those slots.

You could produce a similar outcome like this:

```
var emptySlotsArr = [];
emptySlotsArr.length = 4;
emptySlotsArr[2] = "foo";

Array.from( emptySlotsArr );
// [ undefined, undefined, "foo", undefined ]
```

Avoiding Empty Slots

There's a subtle but important difference in the previous snippet between the `emptySlotsArr` and the result of the `Array.from(...)` call. `Array.from(...)` never produces empty slots.

Prior to ES6, if you wanted to produce an array initialized to a certain length with actual `undefined` values in each slot (no empty slots!), you had to do extra work:

```
var a = Array( 4 );                                // four empty slots!

var b = Array.apply( null, { length: 4 } );        // four `undefined` values
```

But `Array.from(...)` now makes this easier:

```
var c = Array.from( { length: 4 } );                // four `undefined` values
```

Warning: Using an empty slot array like `a` in the previous snippets would work with some array functions, but others ignore empty slots (like `map(...)`, etc.). You should never intentionally work with empty slots, as it will almost certainly lead to strange/unpredictable behavior in your programs.

Mapping

The `Array.from(...)` utility has another helpful trick up its sleeve. The second argument, if provided, is a mapping callback (almost the same as the regular `Array#map(...)` expects) which is called to map/transform each value from the source to the returned target.

Consider:

```

var arrLike = {
  length: 4,
  2: "foo"
};

Array.from( arrLike, function mapper(val, idx){
  if (typeof val == "string") {
    return val.toUpperCase();
  }
  else {
    return idx;
  }
} );
// [ 0, 1, "FOO", 3 ]

```

Note: As with other array methods that take callbacks, `Array.from(..)` takes an optional third argument that if set will specify the `this` binding for the callback passed as the second argument. Otherwise, `this` will be `undefined`.

See "TypedArrays" in Chapter 5 for an example of using `Array.from(..)` in translating values from an array of 8-bit values to an array of 16-bit values.

Creating Arrays and Subtypes

In the last couple of sections, we've discussed `Array.of(..)` and `Array.from(..)`, both of which create a new array in a similar way to a constructor. But what do they do in subclasses? Do they create instances of the base `Array` or the derived subclass?

```

class MyCoolArray extends Array {
  ...
}

MyCoolArray.from( [1, 2] ) instanceof MyCoolArray;      // true

Array.from(
  MyCoolArray.from( [1, 2] )
) instanceof MyCoolArray;                                // false

```

Both `of(..)` and `from(..)` use the constructor that they're accessed from to construct the array. So if you use the base `Array.of(..)` you'll get an `Array` instance, but if you use `MyCoolArray.of(..)`, you'll get a `MycoolArray` instance.

In "Classes" in Chapter 3, we covered the `@@species` setting which all the built-in classes (like `Array`) have defined, which is used by any prototype methods if they create a new instance. `slice(..)` is a great example:

```
var x = new MyCoolArray( 1, 2, 3 );

x.slice( 1 ) instanceof MyCoolArray;           // true
```

Generally, that default behavior will probably be desired, but as we discussed in Chapter 3, you *can* override if you want:

```
class MyCoolArray extends Array {
    // force `species` to be parent constructor
    static get [Symbol.species]() { return Array; }
}

var x = new MyCoolArray( 1, 2, 3 );

x.slice( 1 ) instanceof MyCoolArray;           // false
x.slice( 1 ) instanceof Array;                 // true
```

It's important to note that the `@@species` setting is only used for the prototype methods, like `slice(..)`. It's not used by `of(..)` and `from(..)`; they both just use the `this` binding (whatever constructor is used to make the reference). Consider:

```
class MyCoolArray extends Array {
    // force `species` to be parent constructor
    static get [Symbol.species]() { return Array; }
}

var x = new MyCoolArray( 1, 2, 3 );

MyCoolArray.from( x ) instanceof MyCoolArray;     // true
MyCoolArray.of( [2, 3] ) instanceof MyCoolArray;   // true
```

copyWithin(..) Prototype Method

`Array#copyWithin(..)` is a new mutator method available to all arrays (including Typed Arrays; see Chapter 5). `copyWithin(..)` copies a portion of an array to another location in the same array, overwriting whatever was there before.

The arguments are *target* (the index to copy to), *start* (the inclusive index to start the copying from), and optionally *end* (the exclusive index to stop copying). If any of the arguments are negative, they're taken to be relative from the end of the array.

Consider:

```
[1,2,3,4,5].copyWithin( 3, 0 );           // [1,2,3,1,2]
[1,2,3,4,5].copyWithin( 3, 0, 1 );       // [1,2,3,1,5]
[1,2,3,4,5].copyWithin( 0, -2 );         // [4,5,3,4,5]
[1,2,3,4,5].copyWithin( 0, -2, -1 );     // [4,2,3,4,5]
```

The `copyWithin(..)` method does not extend the array's length, as the first example in the previous snippet shows. Copying simply stops when the end of the array is reached.

Contrary to what you might think, the copying doesn't always go in left-to-right (ascending index) order. It's possible this would result in repeatedly copying an already copied value if the from and target ranges overlap, which is presumably not desired behavior.

So internally, the algorithm avoids this case by copying in reverse order to avoid that gotcha. Consider:

```
[1,2,3,4,5].copyWithin( 2, 1 );           // ???
```

If the algorithm was strictly moving left to right, then the `2` should be copied to overwrite the `3`, then *that* copied `2` should be copied to overwrite `4`, then *that* copied `2` should be copied to overwrite `5`, and you'd end up with `[1,2,2,2,2]`.

Instead, the copying algorithm reverses direction and copies `4` to overwrite `5`, then copies `3` to overwrite `4`, then copies `2` to overwrite `3`, and the final result is `[1,2,2,3,4]`. That's probably more "correct" in terms of expectation, but it can be confusing if you're only thinking about the copying algorithm in a naive left-to-right fashion.

fill(..) Prototype Method

Filling an existing array entirely (or partially) with a specified value is natively supported as of ES6 with the `Array#fill(..)` method:

```
var a = Array( 4 ).fill( undefined );
a;
// [undefined,undefined,undefined,undefined]
```

`fill(..)` optionally takes *start* and *end* parameters, which indicate a subset portion of the array to fill, such as:

```
var a = [ null, null, null, null ].fill( 42, 1, 3 );
a;                                // [null,42,42,null]
```

find(...) Prototype Method

The most common way to search for a value in an array has generally been the `indexof(...)` method, which returns the index the value is found at or `-1` if not found:

```
var a = [1,2,3,4,5];

(a.indexof( 3 ) != -1);           // true
(a.indexof( 7 ) != -1);           // false

(a.indexof( "2" ) != -1);         // false
```

The `indexof(...)` comparison requires a strict `==` match, so a search for `"2"` fails to find a value of `2`, and vice versa. There's no way to override the matching algorithm for `indexof(...)`. It's also unfortunate/ungraceful to have to make the manual comparison to the `-1` value.

Tip: See the *Types & Grammar* title of this series for an interesting (and controversially confusing) technique to work around the `-1` ugliness with the `~` operator.

Since ES5, the most common workaround to have control over the matching logic has been the `some(...)` method. It works by calling a function callback for each element, until one of those calls returns a `true` /truthy value, and then it stops. Because you get to define the callback function, you have full control over how a match is made:

```
var a = [1,2,3,4,5];

a.some( function matcher(v){
    return v == "2";
} );                                // true

a.some( function matcher(v){
    return v == 7;
} );                                // false
```

But the downside to this approach is that you only get the `true` / `false` indicating if a suitably matched value was found, but not what the actual matched value was.

ES6's `find(...)` addresses this. It works basically the same as `some(...)`, except that once the callback returns a `true` /truthy value, the actual array value is returned:

```
var a = [1,2,3,4,5];

a.find( function matcher(v){
    return v == "2";
} );                                // 2

a.find( function matcher(v){
    return v == 7;                    // undefined
});
```

Using a custom `matcher(..)` function also lets you match against complex values like objects:

```
var points = [
    { x: 10, y: 20 },
    { x: 20, y: 30 },
    { x: 30, y: 40 },
    { x: 40, y: 50 },
    { x: 50, y: 60 }
];

points.find( function matcher(point) {
    return (
        point.x % 3 == 0 &&
        point.y % 4 == 0
    );
} );                                // { x: 30, y: 40 }
```

Note: As with other array methods that take callbacks, `find(..)` takes an optional second argument that if set will specify the `this` binding for the callback passed as the first argument. Otherwise, `this` will be `undefined`.

findIndex(..) Prototype Method

While the previous section illustrates how `some(..)` yields a boolean result for a search of an array, and `find(..)` yields the matched value itself from the array search, there's also a need for finding the positional index of the matched value.

`indexof(..)` does that, but there's no control over its matching logic; it always uses `==` strict equality. So ES6's `findIndex(..)` is the answer:

```

var points = [
  { x: 10, y: 20 },
  { x: 20, y: 30 },
  { x: 30, y: 40 },
  { x: 40, y: 50 },
  { x: 50, y: 60 }
];

points.findIndex( function matcher(point) {
  return (
    point.x % 3 == 0 &&
    point.y % 4 == 0
  );
} ); // 2

points.findIndex( function matcher(point) {
  return (
    point.x % 6 == 0 &&
    point.y % 7 == 0
  );
} ); // -1

```

Don't use `findIndex(..) != -1` (the way it's always been done with `indexof(..)`) to get a boolean from the search, because `some(..)` already yields the `true / false` you want. And don't do `a[a.findIndex(..)]` to get the matched value, because that's what `find(..)` accomplishes. And finally, use `indexof(..)` if you need the index of a strict match, or `findIndex(..)` if you need the index of a more customized match.

Note: As with other array methods that take callbacks, `findIndex(..)` takes an optional second argument that if set will specify the `this` binding for the callback passed as the first argument. Otherwise, `this` will be `undefined`.

entries() , values() , keys() Prototype Methods

In Chapter 3, we illustrated how data structures can provide a patterned item-by-item enumeration of their values, via an iterator. We then expounded on this approach in Chapter 5, as we explored how the new ES6 collections (Map, Set, etc.) provide several methods for producing different kinds of iterations.

Because it's not new to ES6, `Array` might not be thought of traditionally as a "collection," but it is one in the sense that it provides these same iterator methods: `entries()` , `values()` , and `keys()` . Consider:

```
var a = [1,2,3];

[...a.values()];           // [1,2,3]
[...a.keys()];            // [0,1,2]
[...a.entries()];          // [ [0,1], [1,2], [2,3] ]

[...a[Symbol.iterator]()]// [1,2,3]
```

Just like with `Set`, the default `Array` iterator is the same as what `values()` returns.

In "Avoiding Empty Slots" earlier in this chapter, we illustrated how `Array.from(...)` treats empty slots in an array as just being present slots with `undefined` in them. That's actually because under the covers, the array iterators behave that way:

```
var a = [];
a.length = 3;
a[1] = 2;

[...a.values()];           // [undefined,2,undefined]
[...a.keys()];            // [0,1,2]
[...a.entries()];          // [ [0,undefined], [1,2], [2,undefined] ]
```

Object

A few additional static helpers have been added to `Object`. Traditionally, functions of this sort have been seen as focused on the behaviors/capabilities of object values.

However, starting with ES6, `Object` static functions will also be for general-purpose global APIs of any sort that don't already belong more naturally in some other location (i.e., `Array.from(...)`).

Object.is(..) Static Function

The `Object.is(..)` static function makes value comparisons in an even more strict fashion than the `==` comparison.

`Object.is(..)` invokes the underlying `SameValue` algorithm (ES6 spec, section 7.2.9). The `SameValue` algorithm is basically the same as the `==` Strict Equality Comparison Algorithm (ES6 spec, section 7.2.13), with two important exceptions.

Consider:

```
var x = NaN, y = 0, z = -0;

x === x;                                // false
y === z;                                // true

Object.is( x, x );                      // true
Object.is( y, z );                      // false
```

You should continue to use `==` for strict equality comparisons; `Object.is(..)` shouldn't be thought of as a replacement for the operator. However, in cases where you're trying to strictly identify a `NaN` or `-0` value, `Object.is(..)` is now the preferred option.

Note: ES6 also adds a `Number.isNaN(..)` utility (discussed later in this chapter) which may be a slightly more convenient test; you may prefer `Number.isNaN(x)` over `Object.is(x,NaN)`. You *can* accurately test for `-0` with a clumsy `x == 0 && 1 / x === -Infinity`, but in this case `Object.is(x,-0)` is much better.

Object.getOwnPropertySymbols(..) Static Function

The "Symbols" section in Chapter 2 discusses the new Symbol primitive value type in ES6.

Symbols are likely going to be mostly used as special (meta) properties on objects. So the `Object.getOwnPropertySymbols(..)` utility was introduced, which retrieves only the symbol properties directly on an object:

```
var o = {
  foo: 42,
  [ Symbol("bar") ]: "hello world",
  baz: true
};

Object.getOwnPropertySymbols( o );    // [ Symbol(bar) ]
```

Object.setPrototypeOf(..) Static Function

Also in Chapter 2, we mentioned the `Object.setPrototypeOf(..)` utility, which (unsurprisingly) sets the `[[Prototype]]` of an object for the purposes of *behavior delegation* (see the *this & Object Prototypes* title of this series). Consider:

```

var o1 = {
  foo() { console.log( "foo" ); }
};

var o2 = {
  // ... o2's definition ...
};

Object.setPrototypeOf( o2, o1 );

// delegates to `o1.foo()`
o2.foo();                                // foo

```

Alternatively:

```

var o1 = {
  foo() { console.log( "foo" ); }
};

var o2 = Object.setPrototypeOf( {
  // ... o2's definition ...
}, o1 );

// delegates to `o1.foo()`
o2.foo();                                // foo

```

In both previous snippets, the relationship between `o2` and `o1` appears at the end of the `o2` definition. More commonly, the relationship between an `o2` and `o1` is specified at the top of the `o2` definition, as it is with classes, and also with `__proto__` in object literals (see "Setting `[[Prototype]]`" in Chapter 2).

Warning: Setting a `[[Prototype]]` right after object creation is reasonable, as shown. But changing it much later is generally not a good idea and will usually lead to more confusion than clarity.

Object.assign(...)

Static Function

Many JavaScript libraries/frameworks provide utilities for copying/mixing one object's properties into another (e.g., jQuery's `extend(...)`). There are various nuanced differences between these different utilities, such as whether a property with value `undefined` is ignored or not.

ES6 adds `Object.assign(...)`, which is a simplified version of these algorithms. The first argument is the *target*, and any other arguments passed are the *sources*, which will be processed in listed order. For each source, its enumerable and own (e.g., not "inherited")

keys, including symbols, are copied as if by plain `=` assignment. `Object.assign(..)` returns the target object.

Consider this object setup:

```

var target = {},
    o1 = { a: 1 }, o2 = { b: 2 },
    o3 = { c: 3 }, o4 = { d: 4 };

// setup read-only property
Object.defineProperty( o3, "e", {
    value: 5,
    enumerable: true,
    writable: false,
    configurable: false
} );

// setup non-enumerable property
Object.defineProperty( o3, "f", {
    value: 6,
    enumerable: false
} );

o3[ Symbol( "g" ) ] = 7;

// setup non-enumerable symbol
Object.defineProperty( o3, Symbol( "h" ), {
    value: 8,
    enumerable: false
} );

Object.setPrototypeOf( o3, o4 );

```

Only the properties `a`, `b`, `c`, `e`, and `Symbol("g")` will be copied to `target`:

```

Object.assign( target, o1, o2, o3 );

target.a;                      // 1
target.b;                      // 2
target.c;                      // 3

Object.getOwnPropertyDescriptor( target, "e" );
// { value: 5, writable: true, enumerable: true,
//   configurable: true }

Object.getOwnPropertySymbols( target );
// [Symbol("g")]

```

The `d`, `f`, and `Symbol("h")` properties are omitted from copying; non-enumerable properties and non-owned properties are all excluded from the assignment. Also, `e` is copied as a normal property assignment, not duplicated as a read-only property.

In an earlier section, we showed using `setPrototypeOf(..)` to set up a `[[Prototype]]` relationship between an `o2` and `o1` object. There's another form that leverages

`Object.assign(..) :`

```
var o1 = {
  foo() { console.log( "foo" ); }
};

var o2 = Object.assign(
  Object.create( o1 ),
  {
    // ... o2's definition ...
  }
);

// delegates to `o1.foo()`
o2.foo();                                // foo
```

Note: `Object.create(..)` is the ES5 standard utility that creates an empty object that is `[[Prototype]]`-linked. See the *this & Object Prototypes* title of this series for more information.

Math

ES6 adds several new mathematic utilities that fill in holes or aid with common operations. All of these can be manually calculated, but most of them are now defined natively so that in some cases the JS engine can either more optimally perform the calculations, or perform them with better decimal precision than their manual counterparts.

It's likely that asm.js/transpiled JS code (see the *Async & Performance* title of this series) is the more likely consumer of many of these utilities rather than direct developers.

Trigonometry:

- `cosh(..)` - Hyperbolic cosine
- `acosh(..)` - Hyperbolic arccosine
- `sinh(..)` - Hyperbolic sine
- `asinh(..)` - Hyperbolic arcsine
- `tanh(..)` - Hyperbolic tangent
- `atanh(..)` - Hyperbolic arctangent

- `hypot(...)` - The squareroot of the sum of the squares (i.e., the generalized Pythagorean theorem)

Arithmetic:

- `cbrt(...)` - Cube root
- `clz32(...)` - Count leading zeros in 32-bit binary representation
- `expm1(...)` - The same as `exp(x) - 1`
- `log2(...)` - Binary logarithm (log base 2)
- `log10(...)` - Log base 10
- `log1p(...)` - The same as `log(x + 1)`
- `imul(...)` - 32-bit integer multiplication of two numbers

Meta:

- `sign(...)` - Returns the sign of the number
- `trunc(...)` - Returns only the integer part of a number
- `fround(...)` - Rounds to nearest 32-bit (single precision) floating-point value

Number

Importantly, for your program to properly work, it must accurately handle numbers. ES6 adds some additional properties and functions to assist with common numeric operations.

Two additions to `Number` are just references to the preexisting globals:

`Number.parseInt(...)` and `Number.parseFloat(...)`.

Static Properties

ES6 adds some helpful numeric constants as static properties:

- `Number.EPSILON` - The minimum value between any two numbers: 2^{-52} (see Chapter 2 of the *Types & Grammar* title of this series regarding using this value as a tolerance for imprecision in floating-point arithmetic)
- `Number.MAX_SAFE_INTEGER` - The highest integer that can "safely" be represented unambiguously in a JS number value: $2^{53} - 1$
- `Number.MIN_SAFE_INTEGER` - The lowest integer that can "safely" be represented unambiguously in a JS number value: $-(2^{53} - 1)$ or $(-2)^{53} + 1$.

Note: See Chapter 2 of the *Types & Grammar* title of this series for more information about "safe" integers.

Number.isNaN(...)

Static Function

The standard global `isNaN(..)` utility has been broken since its inception, in that it returns `true` for things that are not numbers, not just for the actual `NAN` value, because it coerces the argument to a number type (which can falsely result in a NaN). ES6 adds a fixed utility `Number.isNaN(..)` that works as it should:

```
var a = NAN, b = "NaN", c = 42;

isNaN( a );                                // true
isNaN( b );                                // true -- oops!
isNaN( c );                                // false

Number.isNaN( a );                          // true
Number.isNaN( b );                          // false -- fixed!
Number.isNaN( c );                          // false
```

Number.**isFinite**(..) Static Function

There's a temptation to look at a function name like `isFinite(..)` and assume it's simply "not infinite". That's not quite correct, though. There's more nuance to this new ES6 utility. Consider:

```
var a = NAN, b = Infinity, c = 42;

Number.isFinite( a );                      // false
Number.isFinite( b );                      // false

Number.isFinite( c );                      // true
```

The standard global `isFinite(..)` coerces its argument, but `Number.isFinite(..)` omits the coercive behavior:

```
var a = "42";

isFinite( a );                            // true
Number.isFinite( a );                     // false
```

You may still prefer the coercion, in which case using the global `isFinite(..)` is a valid choice. Alternatively, and perhaps more sensibly, you can use `Number.isFinite(+x)`, which explicitly coerces `x` to a number before passing it in (see Chapter 4 of the *Types & Grammar* title of this series).

Integer-Related Static Functions

JavaScript number values are always floating point (IEEE-754). So the notion of determining if a number is an "integer" is not about checking its type, because JS makes no such distinction.

Instead, you need to check if there's any non-zero decimal portion of the value. The easiest way to do that has commonly been:

```
x === Math.floor( x );
```

ES6 adds a `Number.isInteger(..)` helper utility that potentially can determine this quality slightly more efficiently:

```
Number.isInteger( 4 );           // true
Number.isInteger( 4.2 );         // false
```

Note: In JavaScript, there's no difference between `4`, `4.`, `4.0`, or `4.0000`. All of these would be considered an "integer", and would thus yield `true` from `Number.isInteger(..)`.

In addition, `Number.isInteger(..)` filters out some clearly not-integer values that `x === Math.floor(x)` could potentially mix up:

```
Number.isInteger( NaN );        // false
Number.isInteger( Infinity );   // false
```

Working with "integers" is sometimes an important bit of information, as it can simplify certain kinds of algorithms. JS code by itself will not run faster just from filtering for only integers, but there are optimization techniques the engine can take (e.g., asm.js) when only integers are being used.

Because of `Number.isInteger(..)`'s handling of `Nan` and `Infinity` values, defining a `isFloat(..)` utility would not be just as simple as `!Number.isInteger(..)`. You'd need to do something like:

```
function isFloat(x) {
    return Number.isFinite( x ) && !Number.isInteger( x );
}

isFloat( 4.2 );                // true
isFloat( 4 );                  // false

isFloat( NaN );               // false
isFloat( Infinity );          // false
```

Note: It may seem strange, but Infinity should neither be considered an integer nor a float.

ES6 also defines a `Number.isSafeInteger(..)` utility, which checks to make sure the value is both an integer and within the range of `Number.MIN_SAFE_INTEGER - Number.MAX_SAFE_INTEGER` (inclusive).

```
var x = Math.pow( 2, 53 ),
    y = Math.pow( -2, 53 );

Number.isSafeInteger( x - 1 );           // true
Number.isSafeInteger( y + 1 );           // true

Number.isSafeInteger( x );               // false
Number.isSafeInteger( y );               // false
```

String

Strings already have quite a few helpers prior to ES6, but even more have been added to the mix.

Unicode Functions

"Unicode-Aware String Operations" in Chapter 2 discusses `String.fromCodePoint(..)`, `String#codePointAt(..)`, and `String#normalize(..)` in detail. They have been added to improve Unicode support in JS string values.

```
String.fromCodePoint( 0x1d49e );           // ""

"abd" .codePointAt( 2 ).toString( 16 );      // "1d49e"
```

The `normalize(..)` string prototype method is used to perform Unicode normalizations that either combine characters with adjacent "combining marks" or decompose combined characters.

Generally, the normalization won't create a visible effect on the contents of the string, but will change the contents of the string, which can affect how things like the `length` property are reported, as well as how character access by position behave:

```
var s1 = "e\u0301";
s1.length;                                // 2

var s2 = s1.normalize();
s2.length;                                // 1
s2 === "\xE9";                            // true
```

`normalize(..)` takes an optional argument that specifies the normalization form to use. This argument must be one of the following four values: `"NFC"` (default), `"NFD"`, `"NFKC"`, or `"NFKD"`.

Note: Normalization forms and their effects on strings is well beyond the scope of what we'll discuss here. See "Unicode Normalization Forms" (<http://www.unicode.org/reports/tr15/>) for more information.

String.raw(..) Static Function

The `String.raw(..)` utility is provided as a built-in tag function to use with template string literals (see Chapter 2) for obtaining the raw string value without any processing of escape sequences.

This function will almost never be called manually, but will be used with tagged template literals:

```
var str = "bc";

String.raw`\ta${str}d\xE9`;
// "\abcd\xE9", not "abcdé"
```

In the resultant string, `\` and `t` are separate raw characters, not the one escape sequence character `\t`. The same is true with the Unicode escape sequence.

repeat(..) Prototype Function

In languages like Python and Ruby, you can repeat a string as:

```
"foo" * 3;                                // "foofoofoo"
```

That doesn't work in JS, because `*` multiplication is only defined for numbers, and thus `"foo"` coerces to the `Nan` number.

However, ES6 defines a string prototype method `repeat(..)` to accomplish the task:

```
"foo".repeat( 3 );           // "foofoofoo"
```

String Inspection Functions

In addition to `String#indexOf(..)` and `String#lastIndexOf(..)` from prior to ES6, three new methods for searching/inspection have been added: `startsWith(..)`, `endsWith(..)`, and `includes(..)`.

```
var palindrome = "step on no pets";

palindrome.startsWith( "step on" );      // true
palindrome.startsWith( "on", 5 );         // true

palindrome.endsWith( "no pets" );        // true
palindrome.endsWith( "no", 10 );          // true

palindrome.includes( "on" );             // true
palindrome.includes( "on", 6 );           // false
```

For all the string search/inspection methods, if you look for an empty string `""`, it will either be found at the beginning or the end of the string.

Warning: These methods will not by default accept a regular expression for the search string. See "Regular Expression Symbols" in Chapter 7 for information about disabling the `isRegExp` check that is performed on this first argument.

Review

ES6 adds many extra API helpers on the various built-in native objects:

- `Array` adds `of(..)` and `from(..)` static functions, as well as prototype functions like `copyWithin(..)` and `fill(..)`.
- `Object` adds static functions like `is(..)` and `assign(..)`.
- `Math` adds static functions like `acosh(..)` and `clz32(..)`.
- `Number` adds static properties like `Number.EPSILON`, as well as static functions like `Number.isFinite(..)`.
- `String` adds static functions like `String.fromCodePoint(..)` and `String.raw(..)`, as well as prototype functions like `repeat(..)` and `includes(..)`.

Most of these additions can be polyfilled (see ES6 Shim), and were inspired by utilities in common JS libraries/frameworks.

Chapter 7: Meta Programming

Meta programming is programming where the operation targets the behavior of the program itself. In other words, it's programming the programming of your program. Yeah, a mouthful, huh?

For example, if you probe the relationship between one object `a` and another `b` -- are they `[[Prototype]]` linked? -- using `a.isPrototypeOf(b)`, this is commonly referred to as introspection, a form of meta programming. Macros (which don't exist in JS, yet) -- where the code modifies itself at compile time -- are another obvious example of meta programming. Enumerating the keys of an object with a `for..in` loop, or checking if an object is an *instance of* a "class constructor", are other common meta programming tasks.

Meta programming focuses on one or more of the following: code inspecting itself, code modifying itself, or code modifying default language behavior so other code is affected.

The goal of meta programming is to leverage the language's own intrinsic capabilities to make the rest of your code more descriptive, expressive, and/or flexible. Because of the *meta* nature of meta programming, it's somewhat difficult to put a more precise definition on it than that. The best way to understand meta programming is to see it through examples.

ES6 adds several new forms/features for meta programming on top of what JS already had.

Function Names

There are cases where your code may want to introspect on itself and ask what the name of some function is. If you ask what a function's name is, the answer is surprisingly somewhat ambiguous. Consider:

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

var obj = {
    foo: function() {
        // ...
    },
    bar: function baz() {
        // ...
    },
    bam: daz,
    zim() {
        // ...
    }
};
```

In this previous snippet, "what is the name of `obj.foo()`" is slightly nuanced. Is it `"foo"`, `""`, or `undefined`? And what about `obj.bar()` -- is it named `"bar"` or `"baz"`? Is `obj.bam()` named `"bam"` or `"daz"`? What about `obj.zim()`?

Moreover, what about functions which are passed as callbacks, like:

```
function foo(cb) {
    // what is the name of `cb()`` here?
}

foo( function(){
    // I'm anonymous!
} );
```

There are quite a few ways that functions can be expressed in programs, and it's not always clear and unambiguous what the "name" of that function should be.

More importantly, we need to distinguish whether the "name" of a function refers to its `name` property -- yes, functions have a property called `name` -- or whether it refers to the lexical binding name, such as `bar` in `function bar() { ... }`.

The lexical binding name is what you use for things like recursion:

```
function foo(i) {
    if (i < 10) return foo( i * 2 );
    return i;
}
```

The `name` property is what you'd use for meta programming purposes, so that's what we'll focus on in this discussion.

The confusion comes because by default, the lexical name a function has (if any) is also set as its `name` property. Actually there was no official requirement for that behavior by the ES5 (and prior) specifications. The setting of the `name` property was nonstandard but still fairly reliable. As of ES6, it has been standardized.

Tip: If a function has a `name` value assigned, that's typically the name used in stack traces in developer tools.

Inferences

But what happens to the `name` property if a function has no lexical name?

As of ES6, there are now inference rules which can determine a sensible `name` property value to assign a function even if that function doesn't have a lexical name to use.

Consider:

```
var abc = function() {  
    // ...  
};  
  
abc.name;           // "abc"
```

Had we given the function a lexical name like `abc = function def() { .. } , the name property would of course be "def" . But in the absence of the lexical name, intuitively the "abc" name seems appropriate.`

Here are other forms that will infer a name (or not) in ES6:

```

(function(){ ... });           // name:
(function*(){ ... });         // name:
window.foo = function(){ ... }; // name:

class Awesome {
    constructor() { ... }      // name: Awesome
    funny() { ... }            // name: funny
}

var c = class Awesome { ... }; // name: Awesome

var o = {
    foo() { ... },             // name: foo
    *bar() { ... },            // name: bar
    baz: () => { ... },        // name: baz
    bam: function(){ ... },    // name: bam
    get qux() { ... },         // name: get qux
    set fuz() { ... },         // name: set fuz
    ["b" + "iz"]:
        function(){ ... },     // name: biz
    [Symbol( "buz")]:
        function(){ ... }       // name: [buz]
};

var x = o.foo.bind( o );       // name: bound foo
(function(){ ... }).bind( o ); // name: bound

export default function() { ... } // name: default

var y = new Function();        // name: anonymous
var GeneratorFunction =
    function*(){}.__proto__.constructor;
var z = new GeneratorFunction(); // name: anonymous

```

The `name` property is not writable by default, but it is configurable, meaning you can use `Object.defineProperty(..)` to manually change it if so desired.

Meta Properties

In the "`new.target`" section of Chapter 3, we introduced a concept new to JS in ES6: the meta property. As the name suggests, meta properties are intended to provide special meta information in the form of a property access that would otherwise not have been possible.

In the case of `new.target`, the keyword `new` serves as the context for a property access. Clearly `new` is itself not an object, which makes this capability special. However, when `new.target` is used inside a constructor call (a function/method invoked with `new`), `new`

becomes a virtual context, so that `new.target` can refer to the target constructor that `new` invoked.

This is a clear example of a meta programming operation, as the intent is to determine from inside a constructor call what the original `new` target was, generally for the purposes of introspection (examining typing/structure) or static property access.

For example, you may want to have different behavior in a constructor depending on if it's directly invoked or invoked via a child class:

```
class Parent {  
    constructor() {  
        if (new.target === Parent) {  
            console.log( "Parent instantiated" );  
        }  
        else {  
            console.log( "A child instantiated" );  
        }  
    }  
}  
  
class Child extends Parent {}  
  
var a = new Parent();  
// Parent instantiated  
  
var b = new Child();  
// A child instantiated
```

There's a slight nuance here, which is that the `constructor()` inside the `Parent` class definition is actually given the lexical name of the class (`Parent`), even though the syntax implies that the class is a separate entity from the constructor.

Warning: As with all meta programming techniques, be careful of creating code that's too clever for your future self or others maintaining your code to understand. Use these tricks with caution.

Well Known Symbols

In the "Symbols" section of Chapter 2, we covered the new ES6 primitive type `symbol`. In addition to symbols you can define in your own program, JS predefines a number of built-in symbols, referred to as *Well Known Symbols* (WKS).

These symbol values are defined primarily to expose special meta properties that are being exposed to your JS programs to give you more control over JS's behavior.

We'll briefly introduce each and discuss their purpose.

Symbol.iterator

In Chapters 2 and 3, we introduced and used the `@@iterator` symbol, automatically used by `... spreads` and `for..of loops`. We also saw `@@iterator` as defined on the new ES6 collections as defined in Chapter 5.

`Symbol.iterator` represents the special location (property) on any object where the language mechanisms automatically look to find a method that will construct an iterator instance for consuming that object's values. Many objects come with a default one defined.

However, we can define our own iterator logic for any object value by setting the `Symbol.iterator` property, even if that's overriding the default iterator. The meta programming aspect is that we are defining behavior which other parts of JS (namely, operators and looping constructs) use when processing an object value we define.

Consider:

```
var arr = [4,5,6,7,8,9];

for (var v of arr) {
    console.log( v );
}

// 4 5 6 7 8 9

// define iterator that only produces values
// from odd indexes
arr[Symbol.iterator] = function*() {
    var idx = 1;
    do {
        yield this[idx];
    } while ((idx += 2) < this.length);
};

for (var v of arr) {
    console.log( v );
}

// 5 7 9
```

Symbol.toStringTag and Symbol.hasInstance

One of the most common meta programming tasks is to introspect on a value to find out what *kind* it is, usually to decide what operations are appropriate to perform on it. With objects, the two most common inspection techniques are `toString()` and `instanceof`.

Consider:

```
function Foo() {}

var a = new Foo();

a.toString();           // [object Object]
a instanceof Foo;     // true
```

As of ES6, you can control the behavior of these operations:

```
function Foo(greeting) {
    this.greeting = greeting;
}

Foo.prototype[Symbol.toStringTag] = "Foo";

Object.defineProperty( Foo, Symbol.hasInstance, {
    value: function(inst) {
        return inst.greeting == "hello";
    }
} );

var a = new Foo( "hello" ),
    b = new Foo( "world" );

b[Symbol.toStringTag] = "cool";

a.toString();           // [object Foo]
String( b );          // [object cool]

a instanceof Foo;      // true
b instanceof Foo;      // false
```

The `@@toStringTag` symbol on the prototype (or instance itself) specifies a string value to use in the `[object ____]` stringification.

The `@@hasInstance` symbol is a method on the constructor function which receives the instance object value and lets you decide by returning `true` or `false` if the value should be considered an instance or not.

Note: To set `@@hasInstance` on a function, you must use `Object.defineProperty(..)`, as the default one on `Function.prototype` is `writable: false`. See the *this & Object Prototypes* title of this series for more information.

Symbol.species

In "Classes" in Chapter 3, we introduced the `@@species` symbol, which controls which constructor is used by built-in methods of a class that needs to spawn new instances.

The most common example is when subclassing `Array` and wanting to define which constructor (`Array(..)` or your subclass) inherited methods like `slice(..)` should use. By default, `slice(..)` called on an instance of a subclass of `Array` would produce a new instance of that subclass, which is frankly what you'll likely often want.

However, you can meta program by overriding a class's default `@@species` definition:

```
class Cool {
    // defer `@@species` to derived constructor
    static get [Symbol.species]() { return this; }

    again() {
        return new this.constructor[Symbol.species]();
    }
}

class Fun extends Cool {}

class Awesome extends Cool {
    // force `@@species` to be parent constructor
    static get [Symbol.species]() { return Cool; }
}

var a = new Fun(),
    b = new Awesome(),
    c = a.again(),
    d = b.again();

c instanceof Fun;           // true
d instanceof Awesome;       // false
d instanceof Cool;          // true
```

The `Symbol.species` setting defaults on the built-in native constructors to the `return this` behavior as illustrated in the previous snippet in the `cool` definition. It has no default on user classes, but as shown that behavior is easy to emulate.

If you need to define methods that generate new instances, use the meta programming of the `new this.constructor[Symbol.species](..)` pattern instead of the hard-wiring of `new this.constructor(..)` or `new xyz(..)`. Derived classes will then be able to customize `Symbol.species` to control which constructor vends those instances.

Symbol.toPrimitive

In the *Types & Grammar* title of this series, we discussed the `ToPrimitive` abstract coercion operation, which is used when an object must be coerced to a primitive value for some operation (such as `==` comparison or `+` addition). Prior to ES6, there was no way to control this behavior.

As of ES6, the `@@toPrimitive` symbol as a property on any object value can customize that `ToPrimitive` coercion by specifying a method.

Consider:

```
var arr = [1, 2, 3, 4, 5];

arr + 10;           // 1,2,3,4,510

arr[Symbol.toPrimitive] = function(hint) {
  if (hint == "default" || hint == "number") {
    // sum all numbers
    return this.reduce( function(acc, curr){
      return acc + curr;
    }, 0 );
  }
};

arr + 10;           // 25
```

The `Symbol.toPrimitive` method will be provided with a *hint* of `"string"`, `"number"`, or `"default"` (which should be interpreted as `"number"`), depending on what type the operation invoking `ToPrimitive` is expecting. In the previous snippet, the additive `+` operation has no hint (`"default"` is passed). A multiplicative `*` operation would hint `"number"` and a `String(arr)` would hint `"string"`.

Warning: The `==` operator will invoke the `ToPrimitive` operation with no hint -- the `@@toPrimitive` method, if any is called with hint `"default"` -- on an object if the other value being compared is not an object. However, if both comparison values are objects, the behavior of `==` is identical to `==`, which is that the references themselves are directly compared. In this case, `@@toPrimitive` is not invoked at all. See the *Types & Grammar* title of this series for more information about coercion and the abstract operations.

Regular Expression Symbols

There are four well known symbols that can be overridden for regular expression objects, which control how those regular expressions are used by the four corresponding `String.prototype` functions of the same name:

- `@@match` : The `Symbol.match` value of a regular expression is the method used to match all or part of a string value with the given regular expression. It's used by `String.prototype.match(...)` if you pass it a regular expression for the pattern matching.

The default algorithm for matching is laid out in section 21.2.5.6 of the ES6 specification (<http://www.ecma-international.org/ecma-262/6.0/#sec-regexp.prototype-@@match>). You could override this default algorithm and provide extra regex features, such as look-behind assertions.

`Symbol.match` is also used by the `isRegExp` abstract operation (see the note in "String Inspection Functions" in Chapter 6) to determine if an object is intended to be used as a regular expression. To force this check to fail for an object so it's not treated as a regular expression, set the `Symbol.match` value to `false` (or something falsy).

- `@@replace` : The `Symbol.replace` value of a regular expression is the method used by `String.prototype.replace(...)` to replace within a string one or all occurrences of character sequences that match the given regular expression pattern.

The default algorithm for replacing is laid out in section 21.2.5.8 of the ES6 specification (<http://www.ecma-international.org/ecma-262/6.0/#sec-regexp.prototype-@@replace>).

One cool use for overriding the default algorithm is to provide additional `replacer` argument options, such as supporting `"abaca".replace(/a/g, [1,2,3])` producing `"1b2c3"` by consuming the iterable for successive replacement values.

- `@@search` : The `Symbol.search` value of a regular expression is the method used by `String.prototype.search(...)` to search for a sub-string within another string as matched by the given regular expression.

The default algorithm for searching is laid out in section 21.2.5.9 of the ES6 specification (<http://www.ecma-international.org/ecma-262/6.0/#sec-regexp.prototype-@@search>).

- `@@split` : The `Symbol.split` value of a regular expression is the method used by `String.prototype.split(...)` to split a string into sub-strings at the location(s) of the delimiter as matched by the given regular expression.

The default algorithm for splitting is laid out in section 21.2.5.11 of the ES6 specification (<http://www.ecma-international.org/ecma-262/6.0/#sec-regexp.prototype-@@split>).

Overriding the built-in regular expression algorithms is not for the faint of heart! JS ships with a highly optimized regular expression engine, so your own user code will likely be a lot slower. This kind of meta programming is neat and powerful, but it should only be used in cases where it's really necessary or beneficial.

Symbol.isConcatSpreadable

The `@@isConcatSpreadable` symbol can be defined as a boolean property (`Symbol.isConcatSpreadable`) on any object (like an array or other iterable) to indicate if it should be *spread out* if passed to an array `concat(...)`.

Consider:

```
var a = [1,2,3],
  b = [4,5,6];

b[Symbol.isConcatSpreadable] = false;

[].concat( a, b );           // [1,2,3,[4,5,6]]
```

Symbol.unscopables

The `@@unscopables` symbol can be defined as an object property (`Symbol.unscopables`) on any object to indicate which properties can and cannot be exposed as lexical variables in a `with` statement.

Consider:

```
var o = { a:1, b:2, c:3 },
  a = 10, b = 20, c = 30;

o[Symbol.unscopables] = {
  a: false,
  b: true,
  c: false
};

with (o) {
  console.log( a, b, c );           // 1 20 3
}
```

A `true` in the `@@unscopables` object indicates the property should be *unscopable*, and thus filtered out from the lexical scope variables. `false` means it's OK to be included in the lexical scope variables.

Warning: The `with` statement is disallowed entirely in `strict` mode, and as such should be considered deprecated from the language. Don't use it. See the *Scope & Closures* title of this series for more information. Because `with` should be avoided, the `@@unscopables` symbol is also moot.

Proxies

One of the most obviously meta programming features added to ES6 is the `Proxy` feature.

A proxy is a special kind of object you create that "wraps" -- or sits in front of -- another normal object. You can register special handlers (aka *traps*) on the proxy object which are called when various operations are performed against the proxy. These handlers have the opportunity to perform extra logic in addition to *forwarding* the operations on to the original target/wrapped object.

One example of the kind of *trap* handler you can define on a proxy is `get` that intercepts the `[[Get]]` operation -- performed when you try to access a property on an object.

Consider:

```
var obj = { a: 1 },
  handlers = {
    get(target, key, context) {
      // note: target === obj,
      // context === pobj
      console.log( "accessing: ", key );
      return Reflect.get(
        target, key, context
      );
    }
  },
  pobj = new Proxy( obj, handlers );

obj.a;
// 1

pobj.a;
// accessing: a
// 1
```

We declare a `get(...)` handler as a named method on the *handler* object (second argument to `Proxy(...)`), which receives a reference to the *target* object (`obj`), the *key* property name (`"a"`), and the `self /receiver/proxy` (`pobj`).

After the `console.log(...)` tracing statement, we "forward" the operation onto `obj` via `Reflect.get(...)`. We will cover the `Reflect` API in the next section, but note that each available proxy trap has a corresponding `Reflect` function of the same name.

These mappings are symmetric on purpose. The proxy handlers each intercept when a respective meta programming task is performed, and the `Reflect` utilities each perform the respective meta programming task on an object. Each proxy handler has a default definition

that automatically calls the corresponding `Reflect` utility. You will almost certainly use both `Proxy` and `Reflect` in tandem.

Here's a list of handlers you can define on a proxy for a *target* object/function, and how/when they are triggered:

- `get(..)` : via `[[Get]]`, a property is accessed on the proxy (`Reflect.get(..)`, `..` property operator, or `[..]` property operator)
- `set(..)` : via `[[Set]]`, a property value is set on the proxy (`Reflect.set(..)`, the `=` assignment operator, or destructuring assignment if it targets an object property)
- `deleteProperty(..)` : via `[[Delete]]`, a property is deleted from the proxy (`Reflect.deleteProperty(..)` or `delete`)
- `apply(..)` (if *target* is a function) : via `[[Call]]`, the proxy is invoked as a normal function/method (`Reflect.apply(..)`, `call(..)`, `apply(..)`, or the `(..)` call operator)
- `construct(..)` (if *target* is a constructor function) : via `[[Construct]]`, the proxy is invoked as a constructor function (`Reflect.construct(..)` or `new`)
- `getOwnPropertyDescriptor(..)` : via `[[GetOwnProperty]]`, a property descriptor is retrieved from the proxy (`Object.getOwnPropertyDescriptor(..)` or `Reflect.getOwnPropertyDescriptor(..)`)
- `defineProperty(..)` : via `[[DefineOwnProperty]]`, a property descriptor is set on the proxy (`Object.defineProperty(..)` or `Reflect.defineProperty(..)`)
- `getPrototypeOf(..)` : via `[[GetPrototypeOf]]`, the `[[Prototype]]` of the proxy is retrieved (`Object.getPrototypeOf(..)`, `Reflect.getPrototypeOf(..)`, `__proto__`, `Object#isPrototypeOf(..)`, or `instanceof`)
- `setPrototypeOf(..)` : via `[[SetPrototypeOf]]`, the `[[Prototype]]` of the proxy is set (`Object.setPrototypeOf(..)`, `Reflect.setPrototypeOf(..)`, or `__proto__`)
- `preventExtensions(..)` : via `[[PreventExtensions]]`, the proxy is made non-extensible (`Object.preventExtensions(..)` or `Reflect.preventExtensions(..)`)
- `isExtensible(..)` : via `[[IsExtensible]]`, the extensibility of the proxy is probed (`Object.isExtensible(..)` or `Reflect.isExtensible(..)`)
- `ownKeys(..)` : via `[[OwnPropertyKeys]]`, the set of owned properties and/or owned symbol properties of the proxy is retrieved (`Object.keys(..)`, `Object.getOwnPropertyNames(..)`, `Object.getOwnPropertySymbols(..)`, `Reflect.ownKeys(..)`, or `JSON.stringify(..)`)
- `enumerate(..)` : via `[[Enumerate]]`, an iterator is requested for the proxy's enumerable owned and "inherited" properties (`Reflect.enumerate(..)` or `for..in`)
- `has(..)` : via `[[HasProperty]]`, the proxy is probed to see if it has an owned or "inherited" property (`Reflect.has(..)`, `Object#hasOwnProperty(..)`, or `"prop" in obj`)

Tip: For more information about each of these meta programming tasks, see the "`Reflect` API" section later in this chapter.

In addition to the notations in the preceding list about actions that will trigger the various traps, some traps are triggered indirectly by the default actions of another trap. For example:

```
var handlers = {
    getOwnPropertyDescriptor(target,prop) {
        console.log(
            "getOwnPropertyDescriptor"
        );
        return Object.getOwnPropertyDescriptor(
            target, prop
        );
    },
    defineProperty(target,prop,desc){
        console.log( "defineProperty" );
        return Object.defineProperty(
            target, prop, desc
        );
    }
},
proxy = new Proxy( {}, handlers );

proxy.a = 2;
// getOwnPropertyDescriptor
// defineProperty
```

The `getOwnPropertyDescriptor(..)` and `defineProperty(..)` handlers are triggered by the default `set(..)` handler's steps when setting a property value (whether newly adding or updating). If you also define your own `set(..)` handler, you may or may not make the corresponding calls against `context` (not `target` !) which would trigger these proxy traps.

Proxy Limitations

These meta programming handlers trap a wide array of fundamental operations you can perform against an object. However, there are some operations which are not (yet, at least) available to intercept.

For example, none of these operations are trapped and forwarded from `pobj` proxy to `obj` target:

```
var obj = { a:1, b:2 },
  handlers = { ... },
  pobj = new Proxy( obj, handlers );

typeof obj;
String( obj );
obj + "";
obj == pobj;
obj === pobj
```

Perhaps in the future, more of these underlying fundamental operations in the language will be interceptable, giving us even more power to extend JavaScript from within itself.

Warning: There are certain *invariants* -- behaviors which cannot be overridden -- that apply to the use of proxy handlers. For example, the result from the `isExtensible(...)` handler is always coerced to a `boolean`. These invariants restrict some of your ability to customize behaviors with proxies, but they do so only to prevent you from creating strange and unusual (or inconsistent) behavior. The conditions for these invariants are complicated so we won't fully go into them here, but this post (<http://www.2ality.com/2014/12/es6-proxies.html#invariants>) does a great job of covering them.

Revocable Proxies

A regular proxy always traps for the target object, and cannot be modified after creation -- as long as a reference is kept to the proxy, proxying remains possible. However, there may be cases where you want to create a proxy that can be disabled when you want to stop allowing it to proxy. The solution is to create a *revocable proxy*:

```

var obj = { a: 1 },
  handlers = {
    get(target, key, context) {
      // note: target === obj,
      // context === pobj
      console.log( "accessing: ", key );
      return target[key];
    }
  },
  { proxy: pobj, revoke: prevoke } =
  Proxy.revocable( obj, handlers );

pobj.a;
// accessing: a
// 1

// later:
prevoke();

pobj.a;
// TypeError

```

A revocable proxy is created with `Proxy.revocable(..)`, which is a regular function, not a constructor like `Proxy(..)`. Otherwise, it takes the same two arguments: *target* and *handlers*.

The return value of `Proxy.revocable(..)` is not the proxy itself as with `new Proxy(..)`. Instead, it's an object with two properties: *proxy* and *revoke* -- we used object destructuring (see "Destructuring" in Chapter 2) to assign these properties to `pobj` and `prevoke()` variables, respectively.

Once a revocable proxy is revoked, any attempts to access it (trigger any of its traps) will throw a `TypeError`.

An example of using a revocable proxy might be handing out a proxy to another party in your application that manages data in your model, instead of giving them a reference to the real model object itself. If your model object changes or is replaced, you want to invalidate the proxy you handed out so the other party knows (via the errors!) to request an updated reference to the model.

Using Proxies

The meta programming benefits of these Proxy handlers should be obvious. We can almost fully intercept (and thus override) the behavior of objects, meaning we can extend object behavior beyond core JS in some very powerful ways. We'll look at a few example patterns to explore the possibilities.

Proxy First, Proxy Last

As we mentioned earlier, you typically think of a proxy as "wrapping" the target object. In that sense, the proxy becomes the primary object that the code interfaces with, and the actual target object remains hidden/protected.

You might do this because you want to pass the object somewhere that can't be fully "trusted," and so you need to enforce special rules around its access rather than passing the object itself.

Consider:

```

var messages = [],
    handlers = {
        get(target,key) {
            // string value?
            if (typeof target[key] == "string") {
                // filter out punctuation
                return target[key]
                    .replace( /[^\w]/g, "" );
            }

            // pass everything else through
            return target[key];
        },
        set(target,key,value) {
            // only set unique strings, lowercased
            if (typeof value == "string") {
                value = value.toLowerCase();
                if (target.indexOf( value ) == -1) {
                    target.push(value);
                }
            }
            return true;
        },
    },
    messages_proxy =
        new Proxy( messages, handlers );

// elsewhere:
messages_proxy.push(
    "heLLo...", 42, "wOrlD!!", "WoRld!!"
);

messages_proxy.forEach( function(val){
    console.log(val);
} );
// hello world

messages.forEach( function(val){
    console.log(val);
} );
// hello... world!!

```

I call this *proxy first* design, as we interact first (primarily, entirely) with the proxy.

We enforce some special rules on interacting with `messages_proxy` that aren't enforced for `messages` itself. We only add elements if the value is a string and is also unique; we also lowercase the value. When retrieving values from `messages_proxy`, we filter out any punctuation in the strings.

Alternatively, we can completely invert this pattern, where the target interacts with the proxy instead of the proxy interacting with the target. Thus, code really only interacts with the main object. The easiest way to accomplish this fallback is to have the proxy object in the `[[Prototype]]` chain of the main object.

Consider:

```
var handlers = {
    get(target, key, context) {
        return function() {
            context.speak(key + "!");
        };
    }
},
catchall = new Proxy( {}, handlers ),
greeter = {
    speak(who = "someone") {
        console.log( "hello", who );
    }
};

// setup `greeter` to fall back to `catchall`
Object.setPrototypeOf( greeter, catchall );

greeter.speak();           // hello someone
greeter.speak( "world" ); // hello world

greeter.everyone();        // hello everyone!
```

We interact directly with `greeter` instead of `catchall`. When we call `speak(..)`, it's found on `greeter` and used directly. But when we try to access a method like `everyone()`, that function doesn't exist on `greeter`.

The default object property behavior is to check up the `[[Prototype]]` chain (see the *this & Object Prototypes* title of this series), so `catchall` is consulted for an `everyone` property. The proxy `get()` handler then kicks in and returns a function that calls `speak(..)` with the name of the property being accessed (`"everyone"`).

I call this pattern *proxy last*, as the proxy is used only as a last resort.

"No Such Property/Method"

A common complaint about JS is that objects aren't by default very defensive in the situation where you try to access or set a property that doesn't already exist. You may wish to predefine all the properties/methods for an object, and have an error thrown if a nonexistent property name is subsequently used.

We can accomplish this with a proxy, either in *proxy first* or *proxy last* design. Let's consider both.

```

var obj = {
    a: 1,
    foo() {
        console.log( "a:", this.a );
    }
},
handlers = {
    get(target,key,context) {
        if (Reflect.has( target, key )) {
            return Reflect.get(
                target, key, context
            );
        }
        else {
            throw "No such property/method!";
        }
    },
    set(target,key,val,context) {
        if (Reflect.has( target, key )) {
            return Reflect.set(
                target, key, val, context
            );
        }
        else {
            throw "No such property/method!";
        }
    }
},
pobj = new Proxy( obj, handlers );

pobj.a = 3;
pobj.foo();           // a: 3

pobj.b = 4;          // Error: No such property/method!
pobj.bar();          // Error: No such property/method!

```

For both `get(..)` and `set(..)`, we only forward the operation if the target object's property already exists; error thrown otherwise. The proxy object (`pobj`) is the main object code should interact with, as it intercepts these actions to provide the protections.

Now, let's consider inverting with *proxy last* design:

```

var handlers = {
    get() {
        throw "No such property/method!";
    },
    set() {
        throw "No such property/method!";
    }
},
pobj = new Proxy( {}, handlers ),
obj = {
    a: 1,
    foo() {
        console.log( "a:", this.a );
    }
};

// setup `obj` to fall back to `pobj`
Object.setPrototypeOf( obj, pobj );

obj.a = 3;
obj.foo();           // a: 3

obj.b = 4;          // Error: No such property/method!
obj.bar();          // Error: No such property/method!

```

The *proxy last* design here is a fair bit simpler with respect to how the handlers are defined. Instead of needing to intercept the `[[Get]]` and `[[Set]]` operations and only forward them if the target property exists, we instead rely on the fact that if either `[[Get]]` or `[[Set]]` get to our `pobj` fallback, the action has already traversed the whole `[[Prototype]]` chain and not found a matching property. We are free at that point to unconditionally throw the error. Cool, huh?

Proxy Hacking the `[[Prototype]]` Chain

The `[[Get]]` operation is the primary channel by which the `[[Prototype]]` mechanism is invoked. When a property is not found on the immediate object, `[[Get]]` automatically hands off the operation to the `[[Prototype]]` object.

That means you can use the `get(..)` trap of a proxy to emulate or extend the notion of this `[[Prototype]]` mechanism.

The first hack we'll consider is creating two objects which are circularly linked via `[[Prototype]]` (or, at least it appears that way!). You cannot actually create a real circular `[[Prototype]]` chain, as the engine will throw an error. But a proxy can fake it!

Consider:

```

var handlers = {
    get(target,key,context) {
        if (Reflect.has( target, key )) {
            return Reflect.get(
                target, key, context
            );
        }
        // fake circular `[[Prototype]]` link
        else {
            return Reflect.get(
                target[
                    Symbol.for( "[[Prototype]]" )
                ],
                key,
                context
            );
        }
    }
},
obj1 = new Proxy(
{
    name: "obj-1",
    foo() {
        console.log( "foo:", this.name );
    }
},
handlers
),
obj2 = Object.assign(
    Object.create( obj1 ),
{
    name: "obj-2",
    bar() {
        console.log( "bar:", this.name );
        this.foo();
    }
}
);
// fake circular `[[Prototype]]` link
obj1[ Symbol.for( "[[Prototype]]" ) ] = obj2;

obj1.bar();
// bar: obj-1 <-- through proxy faking [[Prototype]]
// foo: obj-1 <-- `this` context still preserved

obj2.foo();
// foo: obj-2 <-- through [[Prototype]]

```

Note: We didn't need to proxy/forward `[[Set]]` in this example, so we kept things simpler. To be fully `[[Prototype]]` emulation compliant, you'd want to implement a `set(..)` handler that searches the `[[Prototype]]` chain for a matching property and respects its descriptor behavior (e.g., `set`, `writable`). See the *this & Object Prototypes* title of this series.

In the previous snippet, `obj2` is `[[Prototype]]` linked to `obj1` by virtue of the `Object.create(..)` statement. But to create the reverse (circular) linkage, we create property on `obj1` at the symbol location `Symbol.for("[[Prototype]]")` (see "Symbols" in Chapter 2). This symbol may look sort of special/magical, but it isn't. It just allows me a conveniently named hook that semantically appears related to the task I'm performing.

Then, the proxy's `get(..)` handler looks first to see if a requested `key` is on the proxy. If not, the operation is manually handed off to the object reference stored in the `Symbol.for("[[Prototype]]")` location of `target`.

One important advantage of this pattern is that the definitions of `obj1` and `obj2` are mostly not intruded by the setting up of this circular relationship between them. Although the previous snippet has all the steps intertwined for brevity's sake, if you look closely, the proxy handler logic is entirely generic (doesn't know about `obj1` or `obj2` specifically). So, that logic could be pulled out into a simple helper that wires them up, like a `setCircularPrototypeOf(..)` for example. We'll leave that as an exercise for the reader.

Now that we've seen how we can use `get(..)` to emulate a `[[Prototype]]` link, let's push the hackery a bit further. Instead of a circular `[[Prototype]]`, what about multiple `[[Prototype]]` linkages (aka "multiple inheritance")? This turns out to be fairly straightforward:

```
var obj1 = {
    name: "obj-1",
    foo() {
        console.log( "obj1.foo:", this.name );
    },
},
obj2 = {
    name: "obj-2",
    foo() {
        console.log( "obj2.foo:", this.name );
    },
    bar() {
        console.log( "obj2.bar:", this.name );
    }
},
handlers = {
    get(target,key,context) {
        if (Reflect.has( target, key )) {
            return Reflect.get(
                target, key, context
            );
        }
    }
};
```

```

        );
    }
    // fake multiple `[[Prototype]]` links
    else {
        for (var P of target[
            Symbol.for( "[[Prototype]]" )
        ]) {
            if (Reflect.has( P, key )) {
                return Reflect.get(
                    P, key, context
                );
            }
        }
    }
},
obj3 = new Proxy(
{
    name: "obj-3",
    baz() {
        this.foo();
        this.bar();
    }
},
handlers
);

// fake multiple `[[Prototype]]` links
obj3[ Symbol.for( "[[Prototype]]" ) ] = [
    obj1, obj2
];

obj3.baz();
// obj1.foo: obj-3
// obj2.bar: obj-3

```

Note: As mentioned in the note after the earlier circular `[[Prototype]]` example, we didn't implement the `set(...)` handler, but it would be necessary for a complete solution that emulates `[[Set]]` actions as normal `[[Prototype]]`s behave.

`obj3` is set up to multiple-delegate to both `obj1` and `obj2`. In `obj3.baz()`, the `this.foo()` call ends up pulling `foo()` from `obj1` (first-come, first-served, even though there's also a `foo()` on `obj2`). If we reordered the linkage as `obj2, obj1`, the `obj2.foo()` would have been found and used.

But as is, the `this.bar()` call doesn't find a `bar()` on `obj1`, so it falls over to check `obj2`, where it finds a match.

`obj1` and `obj2` represent two parallel `[[Prototype]]` chains of `obj3`. `obj1` and/or `obj2` could themselves have normal `[[Prototype]]` delegation to other objects, or either could themselves be a proxy (like `obj3` is) that can multiple-delegate.

Just as with the circular `[[Prototype]]` example earlier, the definitions of `obj1`, `obj2`, and `obj3` are almost entirely separate from the generic proxy logic that handles the multiple-delegation. It would be trivial to define a utility like `setPrototypesOf(..)` (notice the "s"!) that takes a main object and a list of objects to fake the multiple `[[Prototype]]` linkage to. Again, we'll leave that as an exercise for the reader.

Hopefully the power of proxies is now becoming clearer after these various examples. There are many other powerful meta programming tasks that proxies enable.

Reflect API

The `Reflect` object is a plain object (like `Math`), not a function/constructor like the other built-in natives.

It holds static functions which correspond to various meta programming tasks that you can control. These functions correspond one-to-one with the handler methods (*traps*) that Proxies can define.

Some of the functions will look familiar as functions of the same names on `Object`:

- `Reflect.getOwnPropertyDescriptor(..)`
- `Reflect.defineProperty(..)`
- `Reflect.getPrototypeOf(..)`
- `Reflect.setPrototypeOf(..)`
- `Reflect.preventExtensions(..)`
- `Reflect.isExtensible(..)`

These utilities in general behave the same as their `Object.*` counterparts. However, one difference is that the `Object.*` counterparts attempt to coerce their first argument (the target object) to an object if it's not already one. The `Reflect.*` methods simply throw an error in that case.

An object's keys can be accessed/inspected using these utilities:

- `Reflect.ownKeys(..)` : Returns the list of all owned keys (not "inherited"), as returned by both `Object.getOwnPropertyNames(..)` and `Object.getOwnPropertySymbols(..)`. See the "Property Enumeration Order" section for information about the order of keys.
- `Reflect.enumerate(..)` : Returns an iterator that produces the set of all non-symbol keys (owned and "inherited") that are *enumerable* (see the *this & Object Prototypes* title of

this series). Essentially, this set of keys is the same as those processed by a `for..in` loop. See the "Property Enumeration Order" section for information about the order of keys.

- `Reflect.has(..)` : Essentially the same as the `in` operator for checking if a property is on an object or its `[[Prototype]]` chain. For example, `Reflect.has(o, "foo")` essentially performs `"foo" in o`.

Function calls and constructor invocations can be performed manually, separate of the normal syntax (e.g., `(..)` and `new`) using these utilities:

- `Reflect.apply(..)` : For example, `Reflect.apply(foo, thisObj, [42, "bar"])` calls the `foo(..)` function with `thisObj` as its `this`, and passes in the `42` and `"bar"` arguments.
- `Reflect.construct(..)` : For example, `Reflect.construct(foo, [42, "bar"])` essentially calls `new foo(42, "bar")`.

Object property access, setting, and deletion can be performed manually using these utilities:

- `Reflect.get(..)` : For example, `Reflect.get(o, "foo")` retrieves `o.foo`.
- `Reflect.set(..)` : For example, `Reflect.set(o, "foo", 42)` essentially performs `o.foo = 42`.
- `Reflect.deleteProperty(..)` : For example, `Reflect.deleteProperty(o, "foo")` essentially performs `delete o.foo`.

The meta programming capabilities of `Reflect` give you programmatic equivalents to emulate various syntactic features, exposing previously hidden-only abstract operations. For example, you can use these capabilities to extend features and APIs for *domain specific languages* (DSLs).

Property Ordering

Prior to ES6, the order used to list an object's keys/properties was implementation dependent and undefined by the specification. Generally, most engines have enumerated them in creation order, though developers have been strongly encouraged not to ever rely on this ordering.

As of ES6, the order for listing owned properties is now defined (ES6 specification, section 9.1.12) by the `[[OwnPropertyKeys]]` algorithm, which produces all owned properties (strings or symbols), regardless of enumerability. This ordering is only guaranteed for `Reflect.ownKeys(..)` (and by extension, `Object.getOwnPropertyNames(..)` and `Object.getOwnPropertySymbols(..)`).

The ordering is:

1. First, enumerate any owned properties that are integer indexes, in ascending numeric order.
2. Next, enumerate the rest of the owned string property names in creation order.
3. Finally, enumerate owned symbol properties in creation order.

Consider:

```
var o = {};

o[Symbol("c")] = "yay";
o[2] = true;
o[1] = true;
o.b = "awesome";
o.a = "cool";

Reflect.ownKeys( o );           // [1, 2, "b", "a", Symbol(c)]
Object.getOwnPropertyNames( o ); // [1, 2, "b", "a"]
Object.getOwnPropertySymbols( o );// [Symbol(c)]
```

On the other hand, the `[[Enumerate]]` algorithm (ES6 specification, section 9.1.11) produces only enumerable properties, from the target object as well as its `[[Prototype]]` chain. It is used by both `Reflect.enumerate(..)` and `for..in`. The observable ordering is implementation dependent and not controlled by the specification.

By contrast, `Object.keys(..)` invokes the `[[OwnPropertyKeys]]` algorithm to get a list of all owned keys. However, it filters out non-enumerable properties and then reorders the list to match legacy implementation-dependent behavior, specifically with `JSON.stringify(..)` and `for..in`. So, by extension the ordering *also* matches that of `Reflect.enumerate(..)`.

In other words, all four mechanisms (`Reflect.enumerate(..)`, `Object.keys(..)`, `for..in`, and `JSON.stringify(..)`) will match with the same implementation-dependent ordering, though they technically get there in different ways.

Implementations are allowed to match these four to the ordering of `[[OwnPropertyKeys]]`, but are not required to. Nevertheless, you will likely observe the following ordering behavior from them:

```

var o = { a: 1, b: 2 };
var p = Object.create( o );
p.c = 3;
p.d = 4;

for (var prop of Reflect.enumerate( p )) {
    console.log( prop );
}
// c d a b

for (var prop in p) {
    console.log( prop );
}
// c d a b

JSON.stringify( p );
// {"c":3,"d":4}

Object.keys( p );
// ["c", "d"]

```

Boiling this all down: as of ES6, `Reflect.ownKeys(..)`, `Object.getOwnPropertyNames(..)`, and `Object.getOwnPropertySymbols(..)` all have predictable and reliable ordering guaranteed by the specification. So it's safe to build code that relies on this ordering.

`Reflect.enumerate(..)`, `Object.keys(..)`, and `for..in` (as well as `JSON.stringify(..)` by extension) continue to share an observable ordering with each other, as they always have. But that ordering will not necessarily be the same as that of `Reflect.ownKeys(..)`. Care should still be taken in relying on their implementation-dependent ordering.

Feature Testing

What is a feature test? It's a test that you run to determine if a feature is available or not. Sometimes, the test is not just for existence, but for conformance to specified behavior -- features can exist but be buggy.

This is a meta programming technique, to test the environment your program runs in to then determine how your program should behave.

The most common use of feature tests in JS is checking for the existence of an API and if it's not present, defining a polyfill (see Chapter 1). For example:

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

The `if` statement in this snippet is meta programming: we're probing our program and its runtime environment to determine if and how we should proceed.

But what about testing for features that involve new syntax?

You might try something like:

```
try {
    a = () => {};
    ARROW_FUNCS_ENABLED = true;
}
catch (err) {
    ARROW_FUNCS_ENABLED = false;
}
```

Unfortunately, this doesn't work, because our JS programs are compiled. Thus, the engine will choke on the `() => {}` syntax if it is not already supporting ES6 arrow functions. Having a syntax error in your program prevents it from running, which prevents your program from subsequently responding differently if the feature is supported or not.

To meta program with feature tests around syntax-related features, we need a way to insulate the test from the initial compile step our program runs through. For instance, if we could store the code for the test in a string, then the JS engine wouldn't by default try to compile the contents of that string, until we asked it to.

Did your mind just jump to using `eval(...)`?

Not so fast. See the *Scope & Closures* title of this series for why `eval(...)` is a bad idea. But there's another option with less downsides: the `Function(...)` constructor.

Consider:

```
try {
    new Function( "( () => {} )" );
    ARROW_FUNCS_ENABLED = true;
}
catch (err) {
    ARROW_FUNCS_ENABLED = false;
}
```

OK, so now we're meta programming by determining if a feature like arrow functions *can* compile in the current engine or not. You might then wonder, what would we do with this information?

With existence checks for APIs, and defining fallback API polyfills, there's a clear path for what to do with either test success or failure. But what can we do with the information that we get from `ARROW_FUNCS_ENABLED` being `true` or `false`?

Because the syntax can't appear in a file if the engine doesn't support that feature, you can't just have different functions defined in the file with and without the syntax in question.

What you can do is use the test to determine which of a set of JS files you should load. For example, if you had a set of these feature tests in a bootstrapper for your JS application, it could then test the environment to determine if your ES6 code can be loaded and run directly, or if you need to load a transpiled version of your code (see Chapter 1).

This technique is called *split delivery*.

It recognizes the reality that your ES6 authored JS programs will sometimes be able to entirely run "natively" in ES6+ browsers, but other times need transpilation to run in pre-ES6 browsers. If you always load and use the transpiled code, even in the new ES6-compliant environments, you're running suboptimal code at least some of the time. This is not ideal.

Split delivery is more complicated and sophisticated, but it represents a more mature and robust approach to bridging the gap between the code you write and the feature support in browsers your programs must run in.

FeatureTests.io

Defining feature tests for all of the ES6+ syntax, as well as the semantic behaviors, is a daunting task you probably don't want to tackle yourself. Because these tests require dynamic compilation (`new Function(...)`), there's some unfortunate performance cost.

Moreover, running these tests every single time your app runs is probably wasteful, as on average a user's browser only updates once in a several week period at most, and even then, new features aren't necessarily showing up with every update.

Finally, managing the list of feature tests that apply to your specific code base -- rarely will your programs use the entirety of ES6 -- is unruly and error-prone.

The "<https://featuretests.io>" feature-tests-as-a-service offers solutions to these frustrations.

You can load the service's library into your page, and it loads the latest test definitions and runs all the feature tests. It does so using background processing with Web Workers, if possible, to reduce the performance overhead. It also uses LocalStorage persistence to

cache the results in a way that can be shared across all sites you visit which use the service, which drastically reduces how often the tests need to run on each browser instance.

You get runtime feature tests in each of your users' browsers, and you can use those tests results dynamically to serve users the most appropriate code (no more, no less) for their environments.

Moreover, the service provides tools and APIs to scan your files to determine what features you need, so you can fully automate your split delivery build processes.

FeatureTests.io makes it practical to use feature tests for all parts of ES6 and beyond to make sure that only the best code is ever loaded and run for any given environment.

Tail Call Optimization (TCO)

Normally, when a function call is made from inside another function, a second *stack frame* is allocated to separately manage the variables/state of that other function invocation. Not only does this allocation cost some processing time, but it also takes up some extra memory.

A call stack chain typically has at most 10-15 jumps from one function to another and another. In those scenarios, the memory usage is not likely any kind of practical problem.

However, when you consider recursive programming (a function calling itself repeatedly) -- or mutual recursion with two or more functions calling each other -- the call stack could easily be hundreds, thousands, or more levels deep. You can probably see the problems that could cause, if memory usage grows unbounded.

JavaScript engines have to set an arbitrary limit to prevent such programming techniques from crashing by running the browser and device out of memory. That's why we get the frustrating "RangeError: Maximum call stack size exceeded" thrown if the limit is hit.

Warning: The limit of call stack depth is not controlled by the specification. It's implementation dependent, and will vary between browsers and devices. You should never code with strong assumptions of exact observable limits, as they may very well change from release to release.

Certain patterns of function calls, called *tail calls*, can be optimized in a way to avoid the extra allocation of stack frames. If the extra allocation can be avoided, there's no reason to arbitrarily limit the call stack depth, so the engines can let them run unbounded.

A tail call is a `return` statement with a function call, where nothing has to happen after the call except returning its value.

This optimization can only be applied in `strict` mode. Yet another reason to always be writing all your code as `strict`!

Here's a function call that is *not* in tail position:

```
"use strict";

function foo(x) {
    return x * 2;
}

function bar(x) {
    // not a tail call
    return 1 + foo( x );
}

bar( 10 );           // 21
```

`1 + ...` has to be performed after the `foo(x)` call completes, so the state of that `bar(...)` invocation needs to be preserved.

But the following snippet demonstrates calls to `foo(..)` and `bar(..)` where both **are** in tail position, as they're the last thing to happen in their code path (other than the `return`):

```
"use strict";

function foo(x) {
    return x * 2;
}

function bar(x) {
    x = x + 1;
    if (x > 10) {
        return foo( x );
    }
    else {
        return bar( x + 1 );
    }
}

bar( 5 );           // 24
bar( 15 );          // 32
```

In this program, `bar(..)` is clearly recursive, but `foo(..)` is just a regular function call. In both cases, the function calls are in *proper tail position*. The `x + 1` is evaluated before the `bar(..)` call, and whenever that call finishes, all that happens is the `return`.

Proper Tail Calls (PTC) of these forms can be optimized -- called tail call optimization (TCO) -- so that the extra stack frame allocation is unnecessary. Instead of creating a new stack frame for the next function call, the engine just reuses the existing stack frame. That works because a function doesn't need to preserve any of the current state, as nothing happens with that state after the PTC.

TCO means there's practically no limit to how deep the call stack can be. That trick slightly improves regular function calls in normal programs, but more importantly opens the door to using recursion for program expression even if the call stack could be tens of thousands of calls deep.

We're no longer relegated to simply theorizing about recursion for problem solving, but can actually use it in real JavaScript programs!

As of ES6, all PTC should be optimizable in this way, recursion or not.

Tail Call Rewrite

The hitch, however, is that only PTC can be optimized; non-PTC will still work of course, but will cause stack frame allocation as they always did. You'll have to be careful about structuring your functions with PTC if you expect the optimizations to kick in.

If you have a function that's not written with PTC, you may find the need to manually rearrange your code to be eligible for TCO.

Consider:

```
"use strict";

function foo(x) {
    if (x <= 1) return 1;
    return (x / 2) + foo( x - 1 );
}

foo( 123456 );           // RangeError
```

The call to `foo(x-1)` isn't a PTC because its result has to be added to `(x / 2)` before `return`ing.

However, to make this code eligible for TCO in an ES6 engine, we can rewrite it as follows:

```
"use strict";

var foo = (function(){
    function _foo(acc,x) {
        if (x <= 1) return acc;
        return _foo( (x / 2) + acc, x - 1 );
    }

    return function(x) {
        return _foo( 1, x );
    };
})();

foo( 123456 );           // 3810376848.5
```

If you run the previous snippet in an ES6 engine that implements TCO, you'll get the `3810376848.5` answer as shown. However, it'll still fail with a `RangeError` in non-TCO engines.

Non-TCO Optimizations

There are other techniques to rewrite the code so that the call stack isn't growing with each call.

One such technique is called *trampolining*, which amounts to having each partial result represented as a function that either returns another partial result function or the final result. Then you can simply loop until you stop getting a function, and you'll have the result.

Consider:

```

"use strict";

function trampoline( res ) {
    while (typeof res == "function") {
        res = res();
    }
    return res;
}

var foo = (function(){
    function _foo(acc,x) {
        if (x <= 1) return acc;
        return function partial(){
            return _foo( (x / 2) + acc, x - 1 );
        };
    }

    return function(x) {
        return trampoline( _foo( 1, x ) );
    };
})();

foo( 123456 );           // 3810376848.5

```

This reworking required minimal changes to factor out the recursion into the loop in `trampoline(...)`:

1. First, we wrapped the `return _foo ...` line in the `return partial() { .. }` function expression.
2. Then we wrapped the `_foo(1,x)` call in the `trampoline(...)` call.

The reason this technique doesn't suffer the call stack limitation is that each of those inner `partial(...)` functions is just returned back to the `while` loop in `trampoline(...)`, which runs it and then loop iterates again. In other words, `partial(...)` doesn't recursively call itself, it just returns another function. The stack depth remains constant, so it can run as long as it needs to.

Trampolining expressed in this way uses the closure that the inner `partial()` function has over the `x` and `acc` variables to keep the state from iteration to iteration. The advantage is that the looping logic is pulled out into a reusable `trampoline(...)` utility function, which many libraries provide versions of. You can reuse `trampoline(...)` multiple times in your program with different trampolined algorithms.

Of course, if you really wanted to deeply optimize (and the reusability wasn't a concern), you could discard the closure state and inline the state tracking of `acc` into just one function's scope along with a loop. This technique is generally called *recursion unrolling*:

```

"use strict";

function foo(x) {
    var acc = 1;
    while (x > 1) {
        acc = (x / 2) + acc;
        x = x - 1;
    }
    return acc;
}

foo( 123456 );           // 3810376848.5

```

This expression of the algorithm is simpler to read, and will likely perform the best (strictly speaking) of the various forms we've explored. That may seem like a clear winner, and you may wonder why you would ever try the other approaches.

There are some reasons why you might not want to always manually unroll your recursions:

- Instead of factoring out the trampolining (loop) logic for reusability, we've inlined it. This works great when there's only one example to consider, but as soon as you have a half dozen or more of these in your program, there's a good chance you'll want some reusability to keep things shorter and more manageable.
- The example here is deliberately simple enough to illustrate the different forms. In practice, there are many more complications in recursion algorithms, such as mutual recursion (more than just one function calling itself).

The farther you go down this rabbit hole, the more manual and intricate the *unrolling* optimizations are. You'll quickly lose all the perceived value of readability. The primary advantage of recursion, even in the PTC form, is that it preserves the algorithm readability, and offloads the performance optimization to the engine.

If you write your algorithms with PTC, the ES6 engine will apply TCO to let your code run in constant stack depth (by reusing stack frames). You get the readability of recursion with most of the performance benefits and no limitations of run length.

Meta?

What does TCO have to do with meta programming?

As we covered in the "Feature Testing" section earlier, you can determine at runtime what features an engine supports. This includes TCO, though determining it is quite brute force. Consider:

```
"use strict";

try {
  (function foo(x){
    if (x < 5E5) return foo( x + 1 );
  })( 1 );

  TCO_ENABLED = true;
}
catch (err) {
  TCO_ENABLED = false;
}
```

In a non-TCO engine, the recursive loop will fail out eventually, throwing an exception caught by the `try..catch`. Otherwise, the loop completes easily thanks to TCO.

Yuck, right?

But how could meta programming around the TCO feature (or rather, the lack thereof) benefit our code? The simple answer is that you could use such a feature test to decide to load a version of your application's code that uses recursion, or an alternative one that's been converted/transpiled to not need recursion.

Self-Adjusting Code

But here's another way of looking at the problem:

```

"use strict";

function foo(x) {
    function _foo() {
        if (x > 1) {
            acc = acc + (x / 2);
            x = x - 1;
            return _foo();
        }
    }

    var acc = 1;

    while (x > 1) {
        try {
            _foo();
        }
        catch (err) { }
    }

    return acc;
}

foo( 123456 );           // 3810376848.5

```

This algorithm works by attempting to do as much of the work with recursion as possible, but keeping track of the progress via scoped variables `x` and `acc`. If the entire problem can be solved with recursion without an error, great. If the engine kills the recursion at some point, we simply catch that with the `try..catch` and then try again, picking up where we left off.

I consider this a form of meta programming in that you are probing during runtime the ability of the engine to fully (recursively) finish the task, and working around any (non-TCO) engine limitations that may restrict you.

At first (or even second!) glance, my bet is this code seems much uglier to you compared to some of the earlier versions. It also runs a fair bit slower (on larger runs in a non-TCO environment).

The primary advantage, other than it being able to complete any size task even in non-TCO engines, is that this "solution" to the recursion stack limitation is much more flexible than the trampolining or manual unrolling techniques shown previously.

Essentially, `_foo()` in this case is a sort of stand-in for practically any recursive task, even mutual recursion. The rest is the boilerplate that should work for just about any algorithm.

The only "catch" is that to be able to resume in the event of a recursion limit being hit, the state of the recursion must be in scoped variables that exist outside the recursive function(s). We did that by leaving `x` and `acc` outside of the `_foo()` function, instead of passing them as arguments to `_foo()` as earlier.

Almost any recursive algorithm can be adapted to work this way. That means it's the most widely applicable way of leveraging TCO with recursion in your programs, with minimal rewriting.

This approach still uses a PTC, meaning that this code will *progressively enhance* from running using the loop many times (recursion batches) in an older browser to fully leveraging TCO'd recursion in an ES6+ environment. I think that's pretty cool!

Review

Meta programming is when you turn the logic of your program to focus on itself (or its runtime environment), either to inspect its own structure or to modify it. The primary value of meta programming is to extend the normal mechanisms of the language to provide additional capabilities.

Prior to ES6, JavaScript already had quite a bit of meta programming capability, but ES6 significantly ramps that up with several new features.

From function name inferences for anonymous functions to meta properties that give you information about things like how a constructor was invoked, you can inspect the program structure while it runs more than ever before. Well Known Symbols let you override intrinsic behaviors, such as coercion of an object to a primitive value. Proxies can intercept and customize various low-level operations on objects, and `Reflect` provides utilities to emulate them.

Feature testing, even for subtle semantic behaviors like Tail Call Optimization, shifts the meta programming focus from your program to the JS engine capabilities itself. By knowing more about what the environment can do, your programs can adjust themselves to the best fit as they run.

Should you meta program? My advice is: first focus on learning how the core mechanics of the language really work. But once you fully know what JS itself can do, it's time to start leveraging these powerful meta programming capabilities to push the language further!

Chapter 8: Beyond ES6

At the time of this writing, the final draft of ES6 (*ECMAScript 2015*) is shortly headed toward its final official vote of approval by ECMA. But even as ES6 is being finalized, the TC39 committee is already hard at work on features for ES7/2016 and beyond.

As we discussed in Chapter 1, it's expected that the cadence of progress for JS is going to accelerate from updating once every several years to having an official version update once per year (hence the year-based naming). That alone is going to radically change how JS developers learn about and keep up with the language.

But even more importantly, the committee is actually going to work feature by feature. As soon as a feature is spec-complete and has its kinks worked out through implementation experiments in a few browsers, that feature will be considered stable enough to start using. We're all strongly encouraged to adopt features once they're ready instead of waiting for some official standards vote. If you haven't already learned ES6, the time is *past due* to get on board!

As the time of this writing, a list of future proposals and their status can be seen here (<https://github.com/tc39/ecma262#current-proposals>).

Transpilers and polyfills are how we'll bridge to these new features even before all browsers we support have implemented them. Babel, Traceur, and several other major transpilers already have support for some of the post-ES6 features that are most likely to stabilize.

With that in mind, it's already time for us to look at some of them. Let's jump in!

Warning: These features are all in various stages of development. While they're likely to land, and probably will look similar, take the contents of this chapter with more than a few grains of salt. This chapter will evolve in future editions of this title as these (and other!) features finalize.

async function S

In "Generators + Promises" in Chapter 4, we mentioned that there's a proposal for direct syntactic support for the pattern of generators `yield` ing promises to a runner-like utility that will resume it on promise completion. Let's take a brief look at that proposed feature, called `async function`.

Recall this generator example from Chapter 4:

```

run( function *main() {
    var ret = yield step1();

    try {
        ret = yield step2( ret );
    }
    catch (err) {
        ret = yield step2Failed( err );
    }

    ret = yield Promise.all([
        step3a( ret ),
        step3b( ret ),
        step3c( ret )
    ]);

    yield step4( ret );
} )
.then(
    function fulfilled(){
        // `*main()` completed successfully
    },
    function rejected(reason){
        // Oops, something went wrong
    }
);

```

The proposed `async function` syntax can express this same flow control logic without needing the `run(..)` utility, because JS will automatically know how to look for promises to wait and resume. Consider:

```

async function main() {
  var ret = await step1();

  try {
    ret = await step2( ret );
  }
  catch (err) {
    ret = await step2Failed( err );
  }

  ret = await Promise.all( [
    step3a( ret ),
    step3b( ret ),
    step3c( ret )
  ]);

  await step4( ret );
}

main()
.then(
  function fulfilled(){
    // `main()` completed successfully
  },
  function rejected(reason){
    // Oops, something went wrong
  }
);

```

Instead of the `function *main() { .. }` declaration, we declare with the `async function` `main() { .. }` form. And instead of `yield`ing a promise, we `await` the promise. The call to run the function `main()` actually returns a promise that we can directly observe. That's the equivalent to the promise that we get back from a `run(main)` call.

Do you see the symmetry? `async function` is essentially syntactic sugar for the generators + promises + `run(...)` pattern; under the covers, it operates the same!

If you're a C# developer and this `async / await` looks familiar, it's because this feature is directly inspired by C#'s feature. It's nice to see language precedence informing convergence!

Babel, Traceur and other transpilers already have early support for the current status of `async function`s, so you can start using them already. However, in the next section "Caveats", we'll see why you perhaps shouldn't jump on that ship quite yet.

Note: There's also a proposal for `async function*`, which would be called an "async generator." You can both `yield` and `await` in the same code, and even combine those operations in the same statement: `x = await yield y`. The "async generator" proposal

seems to be more in flux -- namely, its return value is not fully worked out yet. Some feel it should be an *observable*, which is kind of like the combination of an iterator and a promise. For now, we won't go further into that topic, but stay tuned as it evolves.

Caveats

One unresolved point of contention with `async function` is that because it only returns a promise, there's no way from the outside to *cancel* an `async function` instance that's currently running. This can be a problem if the `async` operation is resource intensive, and you want to free up the resources as soon as you're sure the result won't be needed.

For example:

```
async function request(url) {
  var resp = await (
    new Promise( function(resolve,reject){
      var xhr = new XMLHttpRequest();
      xhr.open( "GET", url );
      xhr.onreadystatechange = function(){
        if (xhr.readyState == 4) {
          if (xhr.status == 200) {
            resolve( xhr );
          }
          else {
            reject( xhr.statusText );
          }
        }
      };
      xhr.send();
    } )
  );

  return resp.responseText;
}

var pr = request( "http://some.url.1" );

pr.then(
  function fulfilled(responseText){
    // ajax success
  },
  function rejected(reason){
    // Oops, something went wrong
  }
);
```

This `request(...)` that I've conceived is somewhat like the `fetch(...)` utility that's recently been proposed for inclusion into the web platform. So the concern is, what happens if you want to use the `pr` value to somehow indicate that you want to cancel a long-running Ajax request, for example?

Promises are not cancelable (at the time of writing, anyway). In my opinion, as well as many others, they never should be (see the *Async & Performance* title of this series). And even if a promise did have a `cancel()` method on it, does that necessarily mean that calling `pr.cancel()` should actually propagate a cancellation signal all the way back up the promise chain to the `async` function?

Several possible resolutions to this debate have surfaced:

- `async` function `s` won't be cancelable at all (status quo)
- A "cancel token" can be passed to an `async` function at call time
- Return value changes to a cancelable-promise type that's added
- Return value changes to something else non-promise (e.g., observable, or control token with promise and cancel capabilities)

At the time of this writing, `async` function `s` return regular promises, so it's less likely that the return value will entirely change. But it's too early to tell where things will land. Keep an eye on this discussion.

Object.observe(...)

One of the holy grails of front-end web development is data binding -- listening for updates to a data object and syncing the DOM representation of that data. Most JS frameworks provide some mechanism for these sorts of operations.

It appears likely that post ES6, we'll see support added directly to the language, via a utility called `Object.observe(...)`. Essentially, the idea is that you can set up a listener to observe an object's changes, and have a callback called any time a change occurs. You can then update the DOM accordingly, for instance.

There are six types of changes that you can observe:

- add
- update
- delete
- reconfigure
- setPrototypeOf
- preventExtensions

By default, you'll be notified of all these change types, but you can filter down to only the ones you care about.

Consider:

```
var obj = { a: 1, b: 2 };

Object.observe(
  obj,
  function(changes){
    for (var change of changes) {
      console.log( change );
    }
  },
  [ "add", "update", "delete" ]
);

obj.c = 3;
// { name: "c", object: obj, type: "add" }

obj.a = 42;
// { name: "a", object: obj, type: "update", oldValue: 1 }

delete obj.b;
// { name: "b", object: obj, type: "delete", oldValue: 2 }
```

In addition to the main "add", "update", and "delete" change types:

- The "reconfigure" change event is fired if one of the object's properties is reconfigured with `Object.defineProperty(..)`, such as changing its `writable` attribute. See the *this & Object Prototypes* title of this series for more information.
- The "preventExtensions" change event is fired if the object is made non-extensible via `Object.preventExtensions(..)`.

Because both `Object.seal(..)` and `Object.freeze(..)` also imply `Object.preventExtensions(..)`, they'll also fire its corresponding change event. In addition, "reconfigure" change events will also be fired for each property on the object.

- The "setPrototype" change event is fired if the `[[Prototype]]` of an object is changed, either by setting it with the `__proto__` setter, or using `Object.setPrototypeOf(..)`.

Notice that these change events are notified immediately after said change. Don't confuse this with proxies (see Chapter 7) where you can intercept the actions before they occur. Object observation lets you respond after a change (or set of changes) occurs.

Custom Change Events

In addition to the six built-in change event types, you can also listen for and fire custom change events.

Consider:

```
function observer(changes){
  for (var change of changes) {
    if (change.type == "recalc") {
      change.object.c =
        change.object.oldValue +
        change.object.a +
        change.object.b;
    }
  }
}

function changeObj(a,b) {
  var notifier = Object.getNotifier( obj );

  obj.a = a * 2;
  obj.b = b * 3;

  // queue up change events into a set
  notifier.notify( {
    type: "recalc",
    name: "c",
    oldValue: obj.c
  } );
}

var obj = { a: 1, b: 2, c: 3 };

Object.observe(
  obj,
  observer,
  ["recalc"]
);

changeObj( 3, 11 );

obj.a;          // 12
obj.b;          // 30
obj.c;          // 3
```

The `change` set (`"recalc"` custom event) has been queued for delivery to the observer, but not delivered yet, which is why `obj.c` is still `3`.

The changes are by default delivered at the end of the current event loop (see the *Async & Performance* title of this series). If you want to deliver them immediately, use `Object.deliverChangeRecords(observer)`. Once the change events are delivered, you can observe `obj.c` updated as expected:

```
obj.c;           // 42
```

In the previous example, we called `notifier.notify(..)` with the complete change event record. An alternative form for queuing change records is to use `performChange(..)`, which separates specifying the type of the event from the rest of event record's properties (via a function callback). Consider:

```
notifier.performChange( "recalc", function(){
  return {
    name: "c",
    // `this` is the object under observation
    oldValue: this.c
  };
} );
```

In certain circumstances, this separation of concerns may map more cleanly to your usage pattern.

Ending Observation

Just like with normal event listeners, you may wish to stop observing an object's change events. For that, you use `Object.unobserve(..)`.

For example:

```
var obj = { a: 1, b: 2 };

Object.observe( obj, function observer(changes) {
  for (var change of changes) {
    if (change.type == "setPrototypeOf") {
      Object.unobserve(
        change.object, observer
      );
      break;
    }
  }
} );
```

In this trivial example, we listen for change events until we see the `"setPrototypeOf"` event come through, at which time we stop observing any more change events.

Exponentiation Operator

An operator has been proposed for JavaScript to perform exponentiation in the same way that `Math.pow(...)` does. Consider:

```
var a = 2;

a ** 4;           // Math.pow( a, 4 ) == 16

a **= 3;         // a = Math.pow( a, 3 )
a;               // 8
```

Note: `**` is essentially the same as it appears in Python, Ruby, Perl, and others.

Objects Properties and ...

As we saw in the "Too Many, Too Few, Just Enough" section of Chapter 2, the `...` operator is pretty obvious in how it relates to spreading or gathering arrays. But what about objects?

Such a feature was considered for ES6, but was deferred to be considered after ES6 (aka "ES7" or "ES2016" or ...). Here's how it might work in that "beyond ES6" timeframe:

```
var o1 = { a: 1, b: 2 },
o2 = { c: 3 },
o3 = { ...o1, ...o2, d: 4 };

console.log( o3.a, o3.b, o3.c, o3.d );
// 1 2 3 4
```

The `...` operator might also be used to gather an object's destructured properties back into an object:

```
var o1 = { b: 2, c: 3, d: 4 };
var { b, ...o2 } = o1;

console.log( b, o2.c, o2.d );           // 2 3 4
```

Here, the `...o2` re-gathers the destructured `c` and `d` properties back into an `o2` object (`o2` does not have a `b` property like `o1` does).

Again, these are just proposals under consideration beyond ES6. But it'll be cool if they do land.

Array#includes(. .)

One extremely common task JS developers need to perform is searching for a value inside an array of values. The way this has always been done is:

```
var vals = [ "foo", "bar", 42, "baz" ];

if (vals.indexOf( 42 ) >= 0) {
    // found it!
}
```

The reason for the `>= 0` check is because `indexOf(..)` returns a numeric value of `0` or greater if found, or `-1` if not found. In other words, we're using an index-returning function in a boolean context. But because `-1` is truthy instead of falsy, we have to be more manual with our checks.

In the *Types & Grammar* title of this series, I explored another pattern that I slightly prefer:

```
var vals = [ "foo", "bar", 42, "baz" ];

if (~vals.indexOf( 42 )) {
    // found it!
}
```

The `~` operator here conforms the return value of `indexOf(..)` to a value range that is suitably boolean coercible. That is, `-1` produces `0` (falsy), and anything else produces a non-zero (truthy) value, which is what we for deciding if we found the value or not.

While I think that's an improvement, others strongly disagree. However, no one can argue that `indexOf(..)`'s searching logic is perfect. It fails to find `NaN` values in the array, for example.

So a proposal has surfaced and gained a lot of support for adding a real boolean-returning array search method, called `includes(..)`:

```
var vals = [ "foo", "bar", 42, "baz" ];

if (vals.includes( 42 )) {
    // found it!
}
```

Note: `Array#includes(...)` uses matching logic that will find `NaN` values, but will not distinguish between `-0` and `0` (see the *Types & Grammar* title of this series). If you don't care about `-0` values in your programs, this will likely be exactly what you're hoping for. If you *do* care about `-0`, you'll need to do your own searching logic, likely using the `Object.is(...)` utility (see Chapter 6).

SIMD

We cover Single Instruction, Multiple Data (SIMD) in more detail in the *Async & Performance* title of this series, but it bears a brief mention here, as it's one of the next likely features to land in a future JS.

The SIMD API exposes various low-level (CPU) instructions that can operate on more than a single number value at a time. For example, you'll be able to specify two *vectors* of 4 or 8 numbers each, and multiply the respective elements all at once (data parallelism!).

Consider:

```
var v1 = SIMD.float32x4( 3.14159, 21.0, 32.3, 55.55 );
var v2 = SIMD.float32x4( 2.1, 3.2, 4.3, 5.4 );

SIMD.float32x4.mul( v1, v2 );
// [ 6.597339, 67.2, 138.89, 299.97 ]
```

SIMD will include several other operations besides `mul(..)` (multiplication), such as `sub()`, `div()`, `abs()`, `neg()`, `sqrt()`, and many more.

Parallel math operations are critical for the next generations of high performance JS applications.

WebAssembly (WASM)

Brendan Eich made a late breaking announcement near the completion of the first edition of this title that has the potential to significantly impact the future path of JavaScript: WebAssembly (WASM). We will not be able to cover WASM in detail here, as it's extremely early at the time of this writing. But this title would be incomplete without at least a brief mention of it.

One of the strongest pressures on the recent (and near future) design changes of the JS language has been the desire that it become a more suitable target for transpilation/cross-compilation from other languages (like C/C++, ClojureScript, etc.). Obviously, performance

of code running as JavaScript has been a primary concern.

As discussed in the *Async & Performance* title of this series, a few years ago a group of developers at Mozilla introduced an idea to JavaScript called ASM.js. ASM.js is a subset of valid JS that most significantly restricts certain actions that make code hard for the JS engine to optimize. The result is that ASM.js compatible code running in an ASM-aware engine can run remarkably faster, nearly on par with native optimized C equivalents. Many viewed ASM.js as the most likely backbone on which performance-hungry applications would ride in JavaScript.

In other words, all roads to running code in the browser *lead through JavaScript*.

That is, until the WASM announcement. WASM provides an alternate path for other languages to target the browser's runtime environment without having to first pass through JavaScript. Essentially, if WASM takes off, JS engines will grow an extra capability to execute a binary format of code that can be seen as somewhat similar to a bytecode (like that which runs on the JVM).

WASM proposes a format for a binary representation of a highly compressed AST (syntax tree) of code, which can then give instructions directly to the JS engine and its underpinnings, without having to be parsed by JS, or even behave by the rules of JS. Languages like C or C++ can be compiled directly to the WASM format instead of ASM.js, and gain an extra speed advantage by skipping the JS parsing.

The near term for WASM is to have parity with ASM.js and indeed JS. But eventually, it's expected that WASM would grow new capabilities that surpass anything JS could do. For example, the pressure for JS to evolve radical features like threads -- a change that would certainly send major shockwaves through the JS ecosystem -- has a more hopeful future as a future WASM extension, relieving the pressure to change JS.

In fact, this new roadmap opens up many new roads for many languages to target the web runtime. That's an exciting new future path for the web platform!

What does it mean for JS? Will JS become irrelevant or "die"? Absolutely not. ASM.js will likely not see much of a future beyond the next couple of years, but the majority of JS is quite safely anchored in the web platform story.

Proponents of WASM suggest its success will mean that the design of JS will be protected from pressures that would have eventually stretched it beyond assumed breaking points of reasonability. It is projected that WASM will become the preferred target for high-performance parts of applications, as authored in any of a myriad of different languages.

Interestingly, JavaScript is one of the lesser likely languages to target WASM in the future. There may be future changes that carve out subsets of JS that might be tenable for such targeting, but that path doesn't seem high on the priority list.

While JS likely won't be much of a WASM funnel, JS code and WASM code will be able to interoperate in the most significant ways, just as naturally as current module interactions. You can imagine calling a JS function like `foo()` and having that actually invoke a WASM function of that name with the power to run well outside the constraints of the rest of your JS.

Things which are currently written in JS will probably continue to always be written in JS, at least for the foreseeable future. Things which are transpiled to JS will probably eventually at least consider targeting WASM instead. For things which need the utmost in performance with minimal tolerance for layers of abstraction, the likely choice will be to find a suitable non-JS language to author in, then targeting WASM.

There's a good chance this shift will be slow, and will be years in the making. WASM landing in all the major browser platforms is probably a few years out at best. In the meantime, the WASM project (<https://github.com/WebAssembly>) has an early polyfill to demonstrate proof-of-concept for its basic tenets.

But as time goes on, and as WASM learns new non-JS tricks, it's not too much a stretch of imagination to see some currently-JS things being refactored to a WASM-targetable language. For example, the performance sensitive parts of frameworks, game engines, and other heavily used tools might very well benefit from such a shift. Developers using these tools in their web applications likely won't notice much difference in usage or integration, but will just automatically take advantage of the performance and capabilities.

What's certain is that the more real WASM becomes over time, the more it means to the trajectory and design of JavaScript. It's perhaps one of the most important "beyond ES6" topics developers should keep an eye on.

Review

If all the other books in this series essentially propose this challenge, "you (may) not know JS (as much as you thought)," this book has instead suggested, "you don't know JS anymore." The book has covered a ton of new stuff added to the language in ES6. It's an exciting collection of new language features and paradigms that will forever improve our JS programs.

But JS is not done with ES6! Not even close. There's already quite a few features in various stages of development for the "beyond ES6" timeframe. In this chapter, we briefly looked at some of the most likely candidates to land in JS very soon.

`async function` s are powerful syntactic sugar on top of the generators + promises pattern (see Chapter 4). `Object.observe(...)` adds direct native support for observing object change events, which is critical for implementing data binding. The `**` exponentiation operator, `...` for object properties, and `Array#includes(...)` are all simple but helpful improvements to existing mechanisms. Finally, SIMD ushers in a new era in the evolution of high performance JS.

Cliché as it sounds, the future of JS is really bright! The challenge of this series, and indeed of this book, is incumbent on every reader now. What are you waiting for? It's time to get learning and exploring!

Acknowledgments

I have many people to thank for making this book title and the overall series happen.

First, I must thank my wife Christen Simpson, and my two kids Ethan and Emily, for putting up with Dad always pecking away at the computer. Even when not writing books, my obsession with JavaScript glues my eyes to the screen far more than it should. That time I borrow from my family is the reason these books can so deeply and completely explain JavaScript to you, the reader. I owe my family everything.

I'd like to thank my editors at O'Reilly, namely Simon St.Laurent and Brian MacDonald, as well as the rest of the editorial and marketing staff. They are fantastic to work with, and have been especially accommodating during this experiment into "open source" book writing, editing, and production.

Thank you to the many folks who have participated in making this book series better by providing editorial suggestions and corrections, including Shelley Powers, Tim Ferro, Evan Borden, Forrest L. Norvell, Jennifer Davis, Jesse Harlin, and many others. A big thank you to Rick Waldron for writing the Foreword for this title.

Thank you to the countless folks in the community, including members of the TC39 committee, who have shared so much knowledge with the rest of us, and especially tolerated my incessant questions and explorations with patience and detail. John-David Dalton, Jurij "kangax" Zaytsev, Mathias Bynens, Axel Rauschmayer, Nicholas Zakas, Angus Croll, Reginald Braithwaite, Dave Herman, Brendan Eich, Allen Wirfs-Brock, Bradley Meck, Domenic Denicola, David Walsh, Tim Disney, Peter van der Zee, Andrea Giammarchi, Kit Cambridge, Eric Elliott, André Bargull, Caitlin Potter, Brian Terlson, Ingvar Stepanyan, Chris Dickinson, Luke Hoban, and so many others, I can't even scratch the surface.

The *You Don't Know JS* book series was born on Kickstarter, so I also wish to thank all my (nearly) 500 generous backers, without whom this book series could not have happened:

Jan Szpila, nokiko, Murali Krishnamoorthy, Ryan Joy, Craig Patchett, pdqtrader, Dale Fukami, ray hatfield, R0drigo Perez [Mx], Dan Petitt, Jack Franklin, Andrew Berry, Brian Grinstead, Rob Sutherland, Sergi Meseguer, Phillip Gourley, Mark Watson, Jeff Carouth, Alfredo Sumaran, Martin Sachse, Marcio Barrios, Dan, AimelyneM, Matt Sullivan, Delnatte Pierre-Antoine, Jake Smith, Eugen Tudorancea, Iris, David Trinh, simonstl, Ray Daly, Uros Gruber, Justin Myers, Shai Zonis, Mom & Dad, Devin Clark, Dennis Palmer, Brian Panahi Johnson, Josh Marshall, Marshall, Dennis Kerr, Matt Steele, Erik Slagter, Sacah, Justin Rainbow, Christian Nilsson, Delapouite, D.Pereira, Nicolas Hoizey, George V. Reilly, Dan Reeves, Bruno Laturner, Chad Jennings, Shane

King, Jeremiah Lee Cohick, od3n, Stan Yamane, Marko Vucinic, Jim B, Stephen Collins, Ægir Þorsteinsson, Eric Pederson, Owain, Nathan Smith, Jeanetteurphy, Alexandre ELISÉ, Chris Peterson, Rik Watson, Luke Matthews, Justin Lowery, Morten Nielsen, Vernon Kesner, Chetan Shenoy, Paul Tregoin, Marc Grabanski, Dion Almaer, Andrew Sullivan, Keith Elsass, Tom Burke, Brian Ashenfelter, David Stuart, Karl Swedberg, Graeme, Brandon Hays, John Christopher, Gior, manoj reddy, Chad Smith, Jared Harbour, Minoru TODA, Chris Wigley, Daniel Mee, Mike, Handyface, Alex Jahraus, Carl Furrow, Rob Foulkrod, Max Shishkin, Leigh Penny Jr., Robert Ferguson, Mike van Hoenselaar, Hasse Schougaard, rajan venkataguru, Jeff Adams, Trae Robbins, Rolf Langenhuijzen, Jorge Antunes, Alex Koloskov, Hugh Greenish, Tim Jones, Jose Ochoa, Michael Brennan-White, Naga Harish Muvva, Barkóczi Dávid, Kitt Hodsdon, Paul McGraw, Sascha Goldhofer, Andrew Metcalf, Markus Krogh, Michael Mathews, Matt Jared, Juanfran, Georgie Kirschner, Kenny Lee, Ted Zhang, Amit Pahwa, Inbal Sinai, Dan Raine, Schabse Laks, Michael Tervoort, Alexandre Abreu, Alan Joseph Williams, NicolasD, Cindy Wong, Reg Braithwaite, LocalPCGuy, Jon Friskics, Chris Merriman, John Pena, Jacob Katz, Sue Lockwood, Magnus Johansson, Jeremy Crapsey, Grzegorz Pawłowski, nico nuzzaci, Christine Wilks, Hans Bergren, charles montgomery, Ariel Fogel, Ivan Kolev, Daniel Campos, Hugh Wood, Christian Bradford, Frédéric Harper, Ionuț Dan Popa, Jeff Trimble, Rupert Wood, Trey Carrico, Pancho Lopez, Joël kuijten, Tom A Marra, Jeff Jewiss, Jacob Rios, Paolo Di Stefano, Soledad Penades, Chris Gerber, Andrey Dolganov, Wil Moore III, Thomas Martineau, Kareem, Ben Thouret, Udi Nir, Morgan Laupies, jory carson-burson, Nathan L Smith, Eric Damon Walters, Derry Lozano-Hoyland, Geoffrey Wiseman, mkeehner, KatieK, Scott MacFarlane, Brian LaShomb, Adrien Mas, christopher ross, Ian Littman, Dan Atkinson, Elliot Jobe, Nick Dozier, Peter Wooley, John Hoover, dan, Martin A. Jackson, Héctor Fernando Hurtado, andy enamorato, Paul Seltmann, Melissa Gore, Dave Pollard, Jack Smith, Philip Da Silva, Guy Israeli, @megalithic, Damian Crawford, Felix Gliesche, April Carter Grant, Heidi, jim tierney, Andrea Giammarchi, Nico Vignola, Don Jones, Chris Hartjes, Alex Howes, john gibbon, David J. Groom, BBox, Yu 'Dilys' Sun, Nate Steiner, Brandon Satrom, Brian Wyant, Wesley Hales, Ian Pouncey, Timothy Kevin Oxley, George Terezakis, sanjay raj, Jordan Harband, Marko McLion, Wolfgang Kaufmann, Pascal Peuckert, Dave Nugent, Markus Liebelt, Welling Guzman, Nick Cooley, Daniel Mesquita, Robert Syvarth, Chris Coyier, Rémy Bach, Adam Dougal, Alistair Duggin, David Loidolt, Ed Richer, Brian Chenault, GoldFire Studios, Carles Andrés, Carlos Cabo, Yuya Saito, roberto ricardo, Barnett Klane, Mike Moore, Kevin Marx, Justin Love, Joe Taylor, Paul Dijou, Michael Kohler, Rob Cassie, Mike Tierney, Cody Leroy Lindley, tofiji, Shimon Schwartz, Raymond, Luc De Brouwer, David Hayes, Rhys Brett-Bowen, Dmitry, Aziz Khoury, Dean, Scott Tolinski - Level Up, Clement Boirie, Djordje Lukic, Anton Kotenko, Rafael Corral, Philip Hurwitz, Jonathan Pidgeon, Jason Campbell, Joseph C., SwiftOne, Jan Hohner, Derick Bailey, getify, Daniel Cousineau,

Chris Charlton, Eric Turner, David Turner, Joël Galeran, Dharma Vagabond, adam, Dirk van Bergen, dave ❤️♪★ furf, Vedran Zakanj, Ryan McAllen, Natalie Patrice Tucker, Eric J. Bivona, Adam Spooner, Aaron Cavano, Kelly Packer, Eric J, Martin Drenovac, Emilis, Michael Pelikan, Scott F. Walter, Josh Freeman, Brandon Hudgeons, vijay chennupati, Bill Glennon, Robin R., Troy Forster, otaku_coder, Brad, Scott, Frederick Ostrander, Adam Brill, Seb Flippence, Michael Anderson, Jacob, Adam Randlett, Standard, Joshua Clanton, Sebastian Kouba, Chris Deck, SwordFire, Hannes Papenberg, Richard Woeber, hnzz, Rob Crowther, Jedidiah Broadbent, Sergey Chernyshev, Jay-Ar Jamon, Ben Combee, luciano bonachela, Mark Tomlinson, Kit Cambridge, Michael Melgares, Jacob Adams, Adrian Bruinhout, Bev Wieber, Scott Puleo, Thomas Herzog, April Leone, Daniel Mizieliński, Kees van Ginkel, Jon Abrams, Erwin Heiser, Avi Laviad, David newell, Jean-Francois Turcot, Niko Roberts, Erik Dana, Charles Neill, Aaron Holmes, Grzegorz Ziolkowski, Nathan Youngman, Timothy, Jacob Mather, Michael Allan, Mohit Seth, Ryan Ewing, Benjamin Van Treese, Marcelo Santos, Denis Wolf, Phil Keys, Chris Yung, Timo Tijhof, Martin Lekvall, Agendine, Greg Whitworth, Helen Humphrey, Dougal Campbell, Johannes Harth, Bruno Girin, Brian Hough, Darren Newton, Craig McPheat, Olivier Tille, Dennis Roethig, Mathias Bynens, Brendan Stromberger, sundeep, John Meyer, Ron Male, John F Croston III, gigante, Carl Bergenhem, B.J. May, Rebekah Tyler, Ted Foxberry, Jordan Reese, Terry Suitor, afeliz, Tom Kiefer, Darragh Duffy, Kevin Vanderbeken, Andy Pearson, Simon Mac Donald, Abid Din, Chris Joel, Tomas Theunissen, David Dick, Paul Grock, Brandon Wood, John Weis, dgrebb, Nick Jenkins, Chuck Lane, Johnny Megahan, marzsman, Tatu Tamminen, Geoffrey Knauth, Alexander Tarmolov, Jeremy Tymes, Chad Auld, Sean Parmelee, Rob Staenke, Dan Bender, Yannick derwa, Joshua Jones, Geert Plaisier, Tom LeZotte, Christen Simpson, Stefan Bruvik, Justin Falcone, Carlos Santana, Michael Weiss, Pablo Viloslada, Peter deHaan, Dimitris Iliopoulos, seyDoggy, Adam Jordens, Noah Kantrowitz, Amol M, Matthew Winnard, Dirk Ginader, Phinam Bui, David Rapson, Andrew Baxter, Florian Bougel, Michael George, Alban Escalier, Daniel Sellers, Sasha Rudan, John Green, Robert Kowalski, David I. Teixeira (@ditma, Charles Carpenter, Justin Yost, Sam S, Denis Ciccale, Kevin Sheurs, Yannick Croissant, Pau Fracés, Stephen McGowan, Shawn Searcy, Chris Ruppel, Kevin Lamping, Jessica Campbell, Christopher Schmitt, Sablons, Jonathan Reisdorf, Bunni Gek, Teddy Huff, Michael Mullany, Michael Fürstenberg, Carl Henderson, Rick Yoesting, Scott Nichols, Hernán Ciudad, Andrew Maier, Mike Stapp, Jesse Shawl, Sérgio Lopes, jsulak, Shawn Price, Joel Clermont, Chris Ridmann, Sean Timm, Jason Finch, Aiden Montgomery, Elijah Manor, Derek Gathright, Jesse Harlin, Dillon Curry, Courtney Myers, Diego Cadenas, Arne de Bree, João Paulo Dubas, James Taylor, Philipp Kraeutli, Mihai Păun, Sam Gharegozlou, joshjs, Matt Murchison, Eric Windham, Timo Behrmann, Andrew Hall, joshua price, Théophile Villard

This book series is being produced in an open source fashion, including editing and production. We owe GitHub a debt of gratitude for making that sort of thing possible for the community!

Thank you again to all the countless folks I didn't name but who I nonetheless owe thanks. May this book series be "owned" by all of us and serve to contribute to increasing awareness and understanding of the JavaScript language, to the benefit of all current and future community contributors.