You Don't Know JS Yet: Get Started - 2nd Edition

Table of Contents

You Don't Know JS Yet: Get Started - 2nd Edition

Appendix A: Exploring Further

In this appendix, we're going to explore some topics from the main chapter text in a bit more detail. Think of this content as an optional preview of some of the more nuanced details covered throughout the rest of the book series.

Values vs References

In Chapter 2, we introduced the two main types of values: primitives and objects. But we didn't discuss yet one key difference between the two: how these values are assigned and passed around.

In many languages, the developer can choose between assigning/passing a value as the value itself, or as a reference to the value. In JS, however, this decision is entirely determined by the kind of value. That surprises a lot of developers from other languages when they start using JS.

If you assign/pass a value itself, the value is copied. For example:

```
var myName = "Kyle";

var yourName = myName;
```

Here, the `yourName` variable has a separate copy of the `"Kyle"` string from the value that's stored in `myName`. That's because the value is a primitive, and primitive values are always assigned/passed as **value copies**.

Here's how you can prove there's two separate values involved:

```
var myName = "Kyle";

var yourName = myName;

myName = "Frank";

console.log(myName);
// Frank

console.log(yourName);
// Kyle
```

See how `yourName` wasn't affected by the re-assignment of `myName` to `"Frank"`? That's because each variable holds its own copy of the value.

By contrast, references are the idea that two or more variables are pointing at the same value, such that modifying this shared value would be reflected by an access via any of those references. In JS, only object values (arrays, objects, functions, etc) are treated as references.

Consider:

```
var myAddress = {
    street: "123 JS Blvd",
    city: "Austin",
    state: "TX"
};

var yourAddress = myAddress;

// I've got to move to a new house!
myAddress.street = "456 TS Ave";

console.log(yourAddress.street);
// 456 TS Ave
```

Because the value assigned to `myAddress` is an object, it's held/assigned by reference, and thus the assignment to the `yourAddress` variable is a copy of the reference, not the object value itself. That's why the updated value assigned to the `myAddress.street` is reflected the when we access `yourAddress.street`. `myAddress` and `yourAddress` have copies of the reference to the single shared object, so an update to one is an update to both.

Again, JS chooses the value-copy vs reference-copy behavior based on the value type. Primitives are held by value, objects are held by reference. There's no way to override this in JS, in either direction.

So Many Function Forms

Recall this snippet from the "Functions" section in Chapter 2:

```
var awesomeFunction = function(coolThings) {
    // ..
    return amazingStuff;
};
```

The function expression above is referred to as an *anonymous function expression*, since it has no name identifier between the `function` keyword and the `(..)`

parameter list. This point confuses many JS developers because as of ES6, JS performs a "name inference" on an anonymous function:

```
awesomeFunction.name;
// "awesomeFunction"
```

The `name` property of a function will reveal either its directly given name (in the case of a declaration) or its inferred name in the case of an anonymous function expression. That value is generally used by developer tools when inspecting a function value or when reporting an error stack trace.

So even an anonymous function expression *might* get a name. However, name inference only happens in limited cases such as when the function expression is assigned (with `=`). If you pass a function expression as an argument to a function call, for example, no name inference occurs, the `name` property will be an empty string, and the developer console will usually report "(anonymous function)".

Even if a name is inferred, **it's still an anonymous function.** Why? Because the inferred name is a metadata string value, not an available identifier to refer to the function. An anonymous function doesn't have an identifier to use to refer to itself from inside itself – for recursion, event unbinding, etc.

Compare the anonymous function expression form to:

```
// let awesomeFunction = ..
// const awesomeFunction = ..
var awesomeFunction = function someName(coolThings) {
    // ..
    return amazingStuff;
};

awesomeFunction.name;
// "someName"
```

This function expression is a *named function expression*, since the identifier `someName` is directly associated with the function expression at compile time; the association with the identifier `awesomeFunction` still doesn't happen until runtime at the time of that statement. Those two identifiers don't have to match; sometimes it makes sense to have them be different, othertimes it's better to have them be the same.

Notice also that the explicit function name, the identifier `someName`, takes precedence when assigning a *name* for the `name` property.

Should function expressions be named or anonymous? Opinions vary widely on this. Most developers tend to be unconcerned with using anonymous functions. They're shorter, and unquestionably more common in the broad sphere of JS code out there.

In my opinion, if a function exists in your program, it has a purpose; otherwise, take it out! And if it has a purpose, it has a natural name that describes that purpose.

If a function has a name, you the code author should include that name in the code, so that the reader does not have to infer that name from reading and mentally executing that function's source code. Even a trivial function body like `x * 2` has to be read to infer a name like "double" or "multBy2"; that brief extra mental work is unnecessary when you could just take a second to name the function "double" or "multBy2" *once*, saving the reader that repeated mental work every time it's read in the future.

There are, regrettably in some respects, many other function definition forms in JS as of late 2019 (maybe more in the future!).

Here are some more declaration forms:

```
// generator function declaration
function *two() { .. }

// async function declaration
async function three() { .. }

// async generator function declaration
async function *four() { .. }

// named function export declaration (ES6 modules)
export function five() { .. }
```

And here are some more of the (many!) function expression forms:

```
// IIFE
(function(){ .. })();
(function namedIIFE(){ .. })();

// asynchronous IIFE
(async function(){ .. })();
(async function namedAIIFE(){ .. })();

// arrow function expressions
var f;
f = () => 42;
f = x => x * 2;
f = (x) => x * 2;
f = (x,y) => x * y;
f = x => ({ x: x * 2 });
f = x => { return x * 2; };
```

```
f = async x => {
    var y = await doSomethingAsync(x);
    return y * 2;
};
someOperation( x => x * 2 );
// ..
```

Keep in mind that arrow function expressions are **syntactically anonymous**, meaning the syntax doesn't provide a way to provide a direct name identifier for the function. The function expression may get an inferred name, but only if it's one of the assignment forms, not in the (more common!) form of being passed as a function call argument (as in the last line of the snippet).

Since I don't think anonymous functions are a good idea to use frequently in your programs, I'm not a fan of using the `=>` arrow function form. This kind of function actually has a specific purpose – handling the `this` keyword lexically – but that doesn't mean we should use it for every function we write. Use the most appropriate tool for each job.

Functions can also be specified in class definitions and object literal definitions. They're typically referred to as "methods" when in these forms, though in JS this term doesn't have much observable difference over "function".

```
class SomethingKindaGreat {
    // class methods
    coolMethod() { .. }   // no commas!
    boringMethod() { .. }
}

var EntirelyDifferent = {
    // object methods
    coolMethod() { .. },   // commas!
    boringMethod() { .. },

    // (anonymous) function expression property
    oldSchool: function() { .. }
};
```

Phew! That's a lot of different ways to define functions.

There's no simple shortcut path here; you just have to build familiarity with all the function forms so you can recognize them in existing code and use them appropriately in the code you write. Study them closely and practice!

Coercive Conditional Comparison

Yes, that section name is quite a mouthful. But what are we talking about? We're talking about conditional expressions needing to perform coercion-oriented comparisons to make their decisions.

`if` and `? :`-ternary statements, as well as the test clauses in `while` and `for` loops, all perform an implicit value comparison. But what sort? Is it "strict" or "coercive"? Both, actually.

Consider:

```
var x = 1;

if (x) {
    // will run!
}

while (x) {
    // will run, once!
    x = false;
}
```

You might think of these `(x)` conditional expressions like this:

```
var x = 1;

if (x == true) {
    // will run!
}

while (x == true) {
    // will run, once!
    x = false;
}
```

In this specific case – the value of `x` being `1` – that mental model works, but it's not accurate more broadly. Consider:

```
var x = "hello";

if (x) {
    // will run!
}
```

```
if (x == true) {
    // won't run :(
}
```

Oops. So what is the `if` statement actually doing? This is the more accurate mental model.

```
var x = "hello";

if (Boolean(x) == true) {
    // will run
}

// which is the same as:

if (Boolean(x) === true) {
    // will run
}
```

Since the `Boolean(..)` function always returns a value of type boolean, the `==` vs `===` in that above snippet is irrelevant; they'll both do the same thing. But the important part is to see that before the comparison, a coercion occurs, from whatever type `x` currently is, to boolean.

You just can't get away from coercions in JS comparisons. Buckle down and learn them.

Prototypal "Classes"

In Chapter 3, we introduced prototypes and showed how we can link objects through a prototype chain.

Another way of wiring up such prototype linkages served as the (honestly, ugly) predecessor to the elegance of the ES6 `class` system (see Chapter 2), and is referred to as prototypal classes.

TIP: |
:— |
While this style of code is quite uncommon in JS these days, it's still perplexingly rather common to be asked about it in job interviews! |

Let's first recall the `Object.create(..)` style of coding:

```
var Classroom = {
    welcome() {
        console.log("Welcome, students!");
```

```
    }
};

var mathClass = Object.create(Classroom);

mathClass.welcome();
// Welcome, students!
```

Here, a `mathClass` object is linked via its prototype to a `Classroom` object. Through this link, the `mathClass.welcome()` function call is delegated to the method defined on `Classroom`.

The prototypal class pattern would have labeled this delegation behavior "inheritance", and alternately have defined it (with the same behavior) as:

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

Classroom.prototype.welcome = function hello() {
    console.log("Welcome, students!");
};

var mathClass = new Classroom();

mathClass.welcome();
// Welcome, students!
```

All functions by default reference an empty object at a property named `prototype`. Despite the confusing naming, this is **not** the function's *prototype* – where the function is prototype linked to – but rather the prototype object to *link to* when other objects are created by calling the function with `new`.

We add a `welcome` property to that empty `Classroom.prototype` object, pointing at a `hello()` function.

Then `new Classroom()` creates a new object (assigned to `mathClass`), and prototype links it to the existing `Classroom.prototype` object.

Though `mathClass` does not have a `welcome()` property/function, it successfully delegates to `Classroom.prototype.welcome()`.

This "prototypal class" pattern is now strongly discouraged, in favor of using ES6's `class` mechanism:

```
class Classroom {
    constructor() {
```

```
        // ..
    }

    welcome() {
        console.log("Welcome, students!");
    }
}

var mathClass = new Classroom();

mathClass.welcome();
// Welcome, students!
```

Under the covers, the same prototype linkage is wired up, but this `class` syntax fits the class-oriented design pattern much more cleanly than "prototypal classes".

You Don't Know JS Yet: Get Started - 2nd Edition

Appendix B: Practice, Practice, Practice!

In this appendix, we'll explore some exercises and their suggested solutions. These are just to *get you started* with practice over the concepts from the book.

Practicing Comparisons

Let's practice working with value types and comparisons (Chapter 4, Pillar 3) where coercion will need to be involved.

`scheduleMeeting(..)` should take a start time (in 24hr format as a string "hh:mm") and a meeting duration (number of minutes). It should return `true` if the meeting falls entirely within the work day (according to the times specified in `dayStart` and `dayEnd`); return `false` if the meeting violates the work day bounds.

```
const dayStart = "07:30";
const dayEnd = "17:45";

function scheduleMeeting(startTime,durationMinutes) {
    // ..TODO..
}

scheduleMeeting("7:00",15);     // false
scheduleMeeting("07:15",30);    // false
scheduleMeeting("7:30",30);     // true
scheduleMeeting("11:30",60);    // true
scheduleMeeting("17:00",45);    // true
scheduleMeeting("17:30",30);    // false
scheduleMeeting("18:00",15);    // false
```

Try to solve this yourself first. Consider the usage of equality and relational comparison operators, and how coercion impacts this code. Once you have code that works, *compare* your solution(s) to the code in "Suggested Solutions" at the end of this appendix.

Practicing Closure

Now let's practice with closure (Chapter 4, Pillar 1).

The `range(..)` function takes a number as its first argument, representing the first number in a desired range of numbers. The second argument is also a number representing the end of the desired range (inclusive). If the second argument is omitted, then another function should be returned that expects that argument.

```
function range(start,end) {
    // ..TODO..
}

range(3,3);    // [3]
range(3,8);    // [3,4,5,6,7,8]
range(3,0);    // []

var start3 = range(3);
var start4 = range(4);

start3(3);     // [3]
start3(8);     // [3,4,5,6,7,8]
start3(0);     // []

start4(6);     // [4,5,6]
```

Try to solve this yourself first.

Once you have code that works, *compare* your solution(s) to the code in "Suggested Solutions" at the end of this appendix.

Practicing Prototypes

Finally, let's work on `this` and objects linked via prototype (Chapter 4, Pillar 2).

Define a slot machine with 3 reels that can individually `spin()`, and then `display()` the current contents of all the reels.

The basic behavior of a single reel is defined in the `reel` object below. But the slot machine needs individual reels -- objects that delegate to `reel`, and which each have a `position` property.

A reel only *knows how* to `display()` its current slot symbol, but a slot machine typically shows 3 symbols per reel: the current slot (`position`), one slot above (`position - 1`), and one slot below (`position + 1`). So displaying the slot machine should end up displaying a 3 x 3 grid of slot symbols.

```
function randMax(max) {
    return Math.trunc(1E9 * Math.random()) % max;
}

var reel = {
    symbols: [ "♠", "♥", "♦", "♣", "☺", "★", "☾", "☀" ],
    spin() {
        if (this.position == null) {
            this.position = randMax(this.symbols.length - 1);
        }
        this.position = (
            this.position + 100 + randMax(100)
        ) % this.symbols.length;
    },
    display() {
        if (this.position == null) {
```

```
            this.position = randMax(this.symbols.length - 1);
        }
        return this.symbols[this.position];
    }
};

var slotMachine = {
    reels: [
        // this slot machine needs 3 separate reels
        // hint: Object.create(..)
    ],
    spin() {
        this.reels.forEach(function spinReel(reel){
            reel.spin();
        });
    },
    display() {
        // TODO
    }
};

slotMachine.spin();
slotMachine.display();
// ☾ | ☀ | ★
// ☀ | ♠ | ☾
// ♠ | ♥ | ☀

slotMachine.spin();
slotMachine.display();
// ♦ | ♠ | ♣
// ♣ | ♥ | ☺
// ☺ | ♦ | ★
```

Try to solve this yourself first.

Hints:

1. use the `%` modulo operator for wrapping `position` as you access symbols circularly around a reel.
2. use `Object.create(..)` to create an object and prototype-link it to another object. Once linked, delegation allows the objects to share `this` context during method invocation.
3. instead of modifying the reel object directly to show each of the three positions, you can use another temporary object (`Object.create(..)` again) with its own `position` , to delegate from.

Once you have code that works, *compare* your solution(s) to the code in "Suggested Solutions" at the end of this appendix.

Suggested Solutions

Keep in mind that these suggested solutions are just that: suggestions. There's many different ways to solve these practice exercises. Compare your approach to what you see here, and consider the pros and cons of each.

Suggested solution for "Comparisons" (Pillar 3) practice:

```
const dayStart = "07:30";
const dayEnd = "17:45";

function scheduleMeeting(startTime,durationMinutes) {
    var [ , meetingStartHour, meetingStartMinutes ] =
        startTime.match(/^(\d{1,2}):(\d{2})$/) || [];

    durationMinutes = Number(durationMinutes);

    if (
        typeof meetingStartHour == "string" &&
        typeof meetingStartMinutes == "string"
    ) {
        let durationHours = Math.floor(durationMinutes / 60);
        durationMinutes = durationMinutes - (durationHours * 60);
        let meetingEndHour = Number(meetingStartHour) + durationHours;
        let meetingEndMinutes = Number(meetingStartMinutes) + durationMinutes;

        if (meetingEndMinutes > 60) {
            meetingEndHour = meetingEndHour + 1;
            meetingEndMinutes = meetingEndMinutes - 60;
        }

        // re-compose fully-qualified time strings
        // (to make comparison easier)
        let meetingStart = `${
            meetingStartHour.padStart(2,"0")
        }:${
            meetingStartMinutes.padStart(2,"0")
        }`;
        let meetingEnd = `${
            String(meetingEndHour).padStart(2,"0")
        }:${
            String(meetingEndMinutes).padStart(2,"0")
        }`;

        // NOTE: since expressions are all strings,
        // comparisons here are alphabetic, but that's
        // safe here since they're fully qualified
        // time strings (ie, "07:15" < "07:30")
        return (
            meetingStart >= dayStart &&
            meetingEnd <= dayEnd
        );
    }

    return false;
}

scheduleMeeting("7:00",15);     // false
scheduleMeeting("07:15",30);    // false
scheduleMeeting("7:30",30);     // true
scheduleMeeting("11:30",60);    // true
scheduleMeeting("17:00",45);    // true
```

```
scheduleMeeting("17:30",30);    // false
scheduleMeeting("18:00",15);    // false
```

Suggested solution for "Closure" (Pillar 1) practice:

```
function range(start,end) {
    start = Number(start) || 0;

    if (end === undefined) {
        return function getEnd(end) {
            return getRange(start,end);
        };
    }
    else {
        end = Number(end) || 0;
        return getRange(start,end);
    }


    // **********************

    function getRange(start,end) {
        var ret = [];
        for (let i = start; i <= end; i++) {
            ret.push(i);
        }
        return ret;
    }
}

range(3,3);    // [3]
range(3,8);    // [3,4,5,6,7,8]
range(3,0);    // []

var start3 = range(3);
var start4 = range(4);

start3(3);     // [3]
start3(8);     // [3,4,5,6,7,8]
start3(0);     // []

start4(6);     // [4,5,6]
```

Suggested solution for "Prototypes" (Pillar 2) practice:

```
function randMax(max) {
    return Math.trunc(1E9 * Math.random()) % max;
}

var reel = {
    symbols: [ "♠", "♥", "♦", "♣", "☺", "★", "☾", "☀" ],
    spin() {
        if (this.position == null) {
            this.position = randMax(this.symbols.length - 1);
```

```
        }
        this.position = (
            this.position + 100 + randMax(100)
        ) % this.symbols.length;
    },
    display() {
        if (this.position == null) {
            this.position = randMax(this.symbols.length - 1);
        }
        return this.symbols[this.position];
    }
};

var slotMachine = {
    reels: [
        Object.create(reel),
        Object.create(reel),
        Object.create(reel)
    ],
    spin() {
        this.reels.forEach(function spinReel(reel){
            reel.spin();
        });
    },
    display() {
        var lines = [];

        // display all 3 lines on the slot machine
        for (let linePos = -1; linePos <= 1; linePos++) {
            let line = this.reels.map(function getSlot(reel){
                var slot = Object.create(reel);
                slot.position = (
                    reel.symbols.length + reel.position + linePos
                ) % reel.symbols.length;
                return reel.display.call(slot);
            });
            lines.push(line.join(" | "));
        }

        return lines.join("\n");
    }
};

slotMachine.spin();
slotMachine.display();
// ☾ | ☀ | ★
// ☀ | ♠ | ☾
// ♠ | ♥ | ☀

slotMachine.spin();
slotMachine.display();
// ♦ | ♠ | ♣
// ♣ | ♥ | ☺
// ☺ | ♦ | ★
```

That's it for this book. But now it's time to look for real projects to practice these ideas on. Just keep coding, because that's the best way to learn!

You Don't Know JS Yet: Get Started - 2nd Edition

Chapter 1: What *Is* JavaScript?

You don't know JS, yet. Neither do I, not fully anyway. None of us do. But we can all start getting to know JS better.

In this first chapter of the first book of the *You Don't Know JS Yet* (YDKJSY) series, we will take some time to build a foundation to move forward on. We need to start by covering a variety of important background housekeeping details, clearing up some myths and misconceptions about what the language really is (and isn't!).

This is valuable insight into the identity and process of how JS is organized and maintained; all JS developers should understand it. If you want to get to know JS, this is how to *get started* taking the first steps in that journey.

About This Book

I emphasize the word journey because *knowing JS* is not a destination, it's a direction. No matter how much time you spend with the language, you will always be able to find something else to learn and understand a little better. So don't look at this book as something to rush through for a quick achievement. Instead, patience and persistence are best as you take these first few steps.

Following this background chapter, the rest of the book lays out a high-level map of what you will find as you dig into and study JS with the YDKJSY books.

In particular, Chapter 4 identifies three main pillars around which the JS language is organized: scope/closures, prototypes/objects, and types/coercion. JS is a broad and sophisticated language, with many features and capabilities. But all of JS is founded on these three foundational pillars.

Keep in mind that even though this book is titled "Get Started", it's **not intended as a beginner/intro book**. This book's main job is to get you ready for studying JS deeply throughout the rest of the series; it's written assuming you already have familiarity with JS over at least several months experience before moving on in YDKJSY. So to get the most out *Get Started*, make sure you spend plenty of time writing JS code to build up your experience.

Even if you've already written a lot of JS before, this book should not be skimmed over or skipped; take your time to fully process the material here. **A good start always depends on a solid first step.**

What's With That Name?

The name JavaScript is probably the most mistaken and misunderstood programming language name.

Is this language related to Java? Is it only the script form for Java? Is it only for writing scripts and not real programs?

The truth is, the name JavaScript is an artifact of marketing shenanigans. When Brendan Eich first conceived of the language, he code named it Mocha. Internally at Netscape, the brand LiveScript was used. But when it came time to publicly name the language, "JavaScript" won the vote.

Why? Because this language was originally designed to appeal to an audience of mostly Java programmers, and because the word "script" was popular at the time to refer to lightweight programs. These lightweight "scripts" would be the first ones to embed inside of pages on this new thing called the web!

In other words, JavaScript was a marketing ploy to try to position this language as a palatable alternative to writing the heavier and more well-known Java of the day. It could just as easily have been called "WebJava", for that matter.

There are some superficial resemblances between JavaScript's code and Java code. Those similarities don't particularly come from shared development, but from both languages targeting developers with assumed syntax expectations from C (and to an extent, C++).

For example, we use the `{` to begin a block of code and the `}` to end that block of code, just like C/C++ and Java. We also use the `;` to punctuate the end of a statement.

In some ways, legal relationships run even deeper than the syntax. Oracle (via Sun), the company that still owns and runs Java, also owns the official trademark for the name "JavaScript" (via Netscape). This trademark is almost never enforced, and likely couldn't be at this point.

For these reasons, some have suggested we use JS instead of JavaScript. That is a very common shorthand, if not a good candidate for an official language branding itself. Indeed, these books use JS almost exclusively to refer to the language.

Further distancing the language from the Oracle-owned trademark, the official name of the language specified by TC39 and formalized by the ECMA standards body is **ECMAScript**. And indeed, since 2016, the official language name has also been suffixed by the revision year; as of this writing, that's ECMAScript 2019, or otherwise abbreviated ES2019.

In other words, the JavaScript / JS that runs in your browser or in Node.js, is *an* implementation of the ES2019 standard.

NOTE: |
:— |

Don't use terms like "JS6" or "ES8" to refer to the language. Some do, but those terms only serve to perpetuate confusion. "ES20xx" or just "JS" are what you should stick to. |

Whether you call it JavaScript, JS, ECMAScript, or ES2019, it's most definitely not a variant of the Java language!

> "Java is to JavaScript as ham is to hamster." –Jeremy Keith, 2009

Language Specification

I mentioned TC39, the technical steering committee that manages JS. Their primary task is managing the official specification for the language. They meet regularly to vote on any agreed changes, which they then submit to ECMA, the standards organization.

JS's syntax and behavior are defined in the ES specification.

ES2019 happens to be the 10th major numbered specification/revision since JS's inception in 1995, so in the specification's official URL as hosted by ECMA, you'll find "10.0":

https://www.ecma-international.org/ecma-262/10.0/

The TC39 committee is comprised of between 50 and about 100 different people from a broad section of web-invested companies, such as browser makers (Mozilla, Google, Apple) and device makers (Samsung, etc). All members of the committee are volunteers, though many of them are employees of these companies and so may receive compensation in part for their duties on the committee.

TC39 meets generally about every other month, usually for about 3 days, to review work done by members since the last meeting, discuss issues, and vote on proposals. Meeting locations rotate among member companies willing to host.

All TC39 proposals progress through a five stage process – of course, since we're programmers, it's 0-based! – Stage 0 through Stage 4. You can read more about the Stage process here: https://tc39.es/process-document/

Stage 0 means roughly, someone on TC39 thinks it's a worthy idea and plans to champion and work on it. That means lots of ideas that non-TC39 members "propose", through informal means such as social media or blog posts, are really "pre-stage 0". You have to get a TC39 member to champion a proposal for it to be considered "Stage 0" officially.

Once a proposal reaches "Stage 4" status, it is eligible to be included in the next yearly revision of the language. It can take anywhere from several months to a few years for a proposal to work its way through these stages.

All proposals are managed in the open, on TC39's Github repository: https://github.com/tc39/proposals

Anyone, whether on TC39 or not, is welcome to participate in these public discussions and the processes for working on the proposals. However, only TC39 members can attend meetings and vote on the proposals and changes. So in effect, the voice of a TC39 member carries a lot of weight in where JS will go.

Contrary to some established and frustratingly perpetuated myth, there are *not* multiple versions of JavaScript in the wild. There's just **one JS**, the official standard as maintained by TC39 and ECMA.

Back in the early 2000's, when Microsoft maintained a forked and reverse-engineered (and not entirely compatible) version of JS called "JScript", there were legitimately "multiple versions" of JS. But those days are long gone. It's outdated and inaccurate to make such claims about JS today.

All major browsers and device makers have committed to keeping their JS implementations compliant with this one central specification. Of course, engines implement features at different times. But it should never be the case that the v8 engine (Chrome's JS engine) implements a specified feature differently or incompatibly as compared to the SpiderMonkey engine (Mozilla's JS engine).

That means you can learn **one JS**, and rely on that same JS everywhere.

The Web Rules Everything About (JS)

While the array of environments that run JS is constantly expanding – from browsers, to servers (Node.js), to robots, to lightbulbs, to... – the one environment that rules JS is the web. In other words, how JS is implemented for web browsers is, in all practicality, the only reality that matters.

For the most part, the JS defined in the specification and the JS that runs in browser-based JS engines is the same. But there are some differences that must be considered.

Sometimes the JS specification will dictate some new or refined behavior, and yet that won't exactly match with how it works in browser-based JS engines. Such a mismatch is historical: JS engines have had 20+ years of observable behaviors around corner cases of features that have come to be relied on by web content. As such, sometimes the JS engines will refuse to conform to a specification-dictated change because it would break that web content.

In these cases, often TC39 will backtrack and simply choose to conform the specification to the reality of the web. For example, TC39 planned to add a `contains(..)` method for Arrays, but it was found that this name conflicted with old JS frameworks still in use on some sites, so they changed the name to a non-conflicting `includes(..)`. The same happened with a comedic/tragic JS *community crisis* dubbed "smooshgate", where the planned `flatten(..)` method was eventually renamed `flat(..)`.

But occasionally, TC39 will decide the specification should stick firm on some point even though it is unlikely that browser-based JS engines will ever conform.

The solution? Appendix B, "Additional ECMAScript Features for Web Browsers". As of the time of writing, here's the ES2019 Appendix B: https://www.ecma-international.org/ecma-262/10.0/#sec-additional-ecmascript-features-for-web-browsers The JS specification includes this appendix to detail out any known mismatches between the official JS specification and the reality of JS on the web. In other words, these are exceptions that are allowed *only* for web JS; other JS environments must stick to the letter of the law.

Section B.1 and B.2 cover *additions* to JS (syntax and APIs) that web JS includes, again for historical reasons, but which TC39 does not plan to formally specify in the core of JS. Examples include `0`-prefixed octal literals, the global `escape(..)` / `unescape(..)` utilities, String "helpers" like `anchor(..)` and `blink()`, and the RegExp `compile(..)` method.

Section B.3 includes some conflicts where code may run in both web and non-web JS engines, but where the behavior *could* be observably different, resulting in different outcomes. Most of the listed changes involve situations which are labeled as early errors when code is running in strict mode.

Appendix B *gotchas* aren't encountered very often, but it's still a good idea to avoid these constructs to be future safe. Wherever possible, adhere to the JS specification and don't rely on behavior that's only applicable in certain JS engine enviroments.

Not All (Web) JS...

Is this code a JS program?

```
alert("Hello, JS!");
```

Depends on how you look at things. The `alert(..)` function shown here is not included in the JS specification, but is *is* in all web JS environments. Yet, you won't find it in Appendix B, so what gives?

Various JS environments (like browser JS engines, Node.js, etc) add APIs into the global scope of your JS programs that give you environment-specific capabilities, like being able to pop an alert-style box in the user's browser.

In fact, a wide range of JS-looking APIs, like `fetch(..)`, `getCurrentLocation(..)`, and `getUserMedia(..)`, are all web APIs that look like JS. In Node.js, we can access hundreds of API methods from various built-in modules, like `fs.write(..)`.

Another common example is `console.log(..)` (and all the other `console.*` methods!). These are not specified in JS, but because of their universal utility are defined by pretty much every JS environment, according to a roughly-agreed consensus.

So `alert(..)` and `console.log(..)` are not defined by JS. But they *look* like JS. They are functions and object methods and they obey JS syntax rules. The behaviors behind them are controlled by the environment running the JS engine, but on the surface they definitely have to abide by JS to be able to play in the JS playground.

Most of the cross-browser differences people complain about with "JS is so inconsistent!" claims are actually due to differences in how those environment behaviors work, not in how the JS itself works.

So an `alert(..)` call *is* JS, but `alert` itself is really just a guest, not part of the official JS specification.

It's Not Always JS

Using the console/REPL (Read-Evaluate-Print-Loop) in your browser's Developer Tools (or Node) feels like a pretty straightforward JS environment at first glance. But it's not, really.

Developer Tools are... tools for developers. Their primary purpose is to make life easier for developers. They prioritize DX (Developer Experience). It is *not* a goal of such tools to accurately and purely reflect all nuances of strict-spec JS behavior. As such, there's many quirks that may act as "gotchas" if you're treating the console as a *pure* JS environment.

This convenience is a good thing, by the way! I'm glad developer tools make developers' lives easier! I'm glad we have nice UX charms like auto-complete of variables/properties, etc. I'm just pointing out that we can't and shouldn't expect such tools to *always* adhere strictly to the way JS programs are handled, because that's not the purpose of these tools.

Since such tools vary in behavior from browser to browser, and since they change (sometimes rather frequently), I'm not going to "hardcode" any of the specific details into this text, thereby ensuring this book text is outdated quickly.

But I'll just hint at some examples of quirks that have been true at various points in different JS console environments, to reinforce my point about not assuming native JS behavior while using them:

- Whether a `var` or `function` declaration in the top-level "global scope" of the console actually creates a real global variable (and mirrored `window` property, and vice versa!).
- What happens with multiple `let` and `const` declarations in the top-level "global scope".
- Whether `"use strict";` on one line-entry (pressing `<enter>` after) enables strict mode for the rest of that console session, the way it would on the first

line of a .js file, as well as whether you can use `"use strict";` beyond the "first line" and still get strict mode turned on for that session.

- How non-strict mode `this` default-binding works for function calls, and whether the "global object" used will contain expected global variables.
- How hoisting (see "Scope & Closures", Chapter 3) works across multiple line entries.
- ...several others

The developer console is not trying to pretend to be a JS compiler that handles your entered code exactly the same way the JS engine handles a .js file. It's trying to make it easy for you to quickly enter a few lines of code and see the results immediately. These are entirely different use-cases, and as such, it's unreasonable to expect one tool to handle both equally.

Don't trust what behavior you see in a developer console as representing *exact* to-the-letter JS semantics; for that, read the specification. Instead, think of the console as a "JS friendly" environment. That's useful in its own right.

Many Faces

The term "paradigm" in programming language context refers to a broad (almost universal) mindset and approach to structuring code. Within a paradigm, there are myriad variations of style and form that distinguish programs, including countless different libraries and frameworks which leave their unique signature on any given code.

But no matter what a program's individual style may be, the big picture divisions around paradigms are almost always evident at first glance of any program.

Typical paradigm-level code categories include: procedural, object-oriented (OO/classes), and functional (FP).

1. Procedural style organizes code in a top-down, linear progression through a pre-determined set of operations, usually collected together in related units called procedures.
2. OO style organizes code by collecting logic and data together into units called classes.
3. FP style organizes code into functions (pure computations as opposed to procedures), and the adaptations of those functions as values.

Paradigms are neither right nor wrong. They're orientations that guide and mold how programmers approach problems and solutions, how they structure and maintain their code.

Some languages are heavily slanted toward one paradigm – C is procedural, Java/C++ are almost entirely class oriented, and Haskell is FP through and through.

But many languages also support code patterns that can come from, and even mix-n-match from, different paradigms. So called "multi-paradigm languages" offer ultimate flexibility. In some cases, a single program can even have two or more expressions of these paradigms sitting side-by-side.

JavaScript is most definitely a multi-paradigm language. You can write procedural, class-oriented, or FP-style code, and you can make those decisions on a line-by-line basis instead of being forced into an all-or-nothing choice.

Backwards & Forwards

One of the most foundational principles that guides JavaScript is preservation of *backwards compatibility*. Many are confused by the implications of this term, and often confuse it with a related but different term: *forwards compatibility*.

Let's set the record straight.

Backwards compatibility means that once something is accepted as valid JS, there will not be a future change to the language that causes that code to become invalid JS. Code written in 1995 – however primitive or limited it may have been! – should still work today. As TC39 members often proclaim, "we don't break the web!"

The idea is that JS developers can write code with confidence that their code won't stop working unpredictably because a browser update is released. This makes the decision to choose JS for a program a more wise and safe investment, for years into the future.

That "guarantee" is no small thing. Maintaining backwards compatibility, stretched out across almost 25 years of the language's history, creates an enormous burden and a whole slew of unique challenges. You'd be hard pressed to find many other examples in computing of such a commitment to backwards compatibility.

The costs of sticking to this principle should not be casually dismissed. It necessarily creates a very high bar to including changing or extending the language; any decision becomes effectively permanent, mistakes and all. Once it's in JS, it can't be taken out because it might break programs, even if we'd really, really like to remove it!

There are some small exceptions to this rule. JS has had some backwards-incompatible changes, but TC39 is extremely cautious in doing so. They study existing code on the web (via browser data gathering) to estimate the impact of such breakage, and browsers ultimately decide and vote on whether they're willing to take the heat from users for a very small scale breakage weighed

against the benefits of fixing or improving some aspect of the language for many more sites (and users).

These kinds of changes are rare, and are almost always in corner cases of usage that are unlikely to be observably breaking in many sites.

Compare *backwards compatibility* to its counterpart, *forwards compatibility*. Being forwards-compatible means that including a new addition to the language in a program would not cause that program to break if it were run in an older JS engine. **JS is not forwards-compatible**, despite many wishing such, and even incorrectly believing the myth that it is.

HTML and CSS, by contrast, are forwards-compatible but not backwards-compatible. If you dug up some HTML or CSS written back in 1995, it's entirely possible it would not work (or work the same) today. But, if you use a new feature from 2019 in a browser from 2010, the page isn't "broken" – the unrecognized CSS / HTML is skipped over, while the rest of the CSS / HTML would be processed accordingly.

It may seem desirable for forwards-compatibility to be included in programming language design, but it's generally impractical to do so. Markup (HTML) or styling (CSS) are declarative in nature, so it's much easier to "skip over" unrecognized declarations with minimal impact to other recognized declarations.

But chaos and non-determinism would ensue if a programming language engine selectively skipped statements (or even expressions!) that it didn't understand, as it's impossible to ensure that a subsequent part of the program wasn't expecting the skipped over part to have been processed.

Though JS isn't, and can't be, forwards-compatible, it's critical to recognize JS's backwards compatibility, including the enduring benefits to the web and the constraints and difficulties it places on JS as a result.

Jumping The Gaps

Since JS is not forwards-compatible, it means that there is always the potential for a gap between code that you can write that's valid JS, and the oldest engine that your site or application needs to support. If you run a program that uses an ES2019 feature in an engine from 2016, you're very likely to see the program break and crash.

If the feature is a new syntax, the program will in general completely fail to compile and run, usually throwing a syntax error. If the feature is an API (such as ES6's `Object.is(..)`), the program may run up to a point but then throw a runtime exception and stop once it encounters the reference to the unknown API.

Does this mean JS developers should always lag behind the pace of progress, using only code which is on the trailing edge of the oldest JS engine environments they need to support? No!

But it does mean that JS developers need to take special care to address this gap.

For new and incompatible syntax, the solution is transpiling. Transpiling is a contrived and community-invented term to describe using a tool to convert the source code of a program from one form to another (but still as textual source code). Typically, forwards-compatibility problems related to syntax are solved by using a transpiler – the most common one being Babel (https://babeljs.io) – to convert from that newer JS syntax version to an equivalent of code that uses an older syntax.

For example, a developer may write a snippet of code like:

```
if (something) {
    let x = 3;
    console.log(x);
}
else {
    let x = 4;
    console.log(x);
}
```

This is how the code would look in the source code tree for that application. But when producing the file(s) to deploy to the public website, the Babel transpiler might convert that code to look like this:

```
var x$0;
var x$1;
if (something) {
    x$0 = 3;
    console.log(x$0);
}
else {
    x$1 = 4;
    console.log(x$1);
}
```

The original snippet relied on `let` to create block-scoped `x` variables in both the `if` and `else` clauses which did not interfere with each other. An equivalent program (with minimal re-working) that Babel can produce just chooses to name two different variables with unique names, producing the same non-interference outcome.

NOTE: |
:— |
The `let` keyword was added in ES6 (in 2015). The above example of transpiling

would only need to apply if an application needed to run in an pre-ES6 supporting JS environment. The example here is just for simplicity of illustration. When ES6 was new, the need for such a transpilation was quite prevalent, but in 2019 it's much less common to need to support pre-ES6 environments. The "target" used for transpiliation is thus a sliding window that shifts upward only as decisions are made for a site/application to stop supporting some old browser/engine. |

You may wonder: why go to the trouble of using a tool to convert from a newer syntax version to an older one? Couldn't we just write the two variables and skip using the `let` keyword? The reason is, it's strongly recommended that developers use the latest version of JS so that their code is clean and communicates its ideas most effectively.

Developers should focus on writing the clean, new syntax forms, and let the tools take care of producing a forwards-compatible version of that code that is suitable to deploy and run on the oldest supported JS engine environments.

Filling The Gaps

If the forwards-compatibility issue is not related to new syntax, but rather to a missing API method that was only recently added, the most common solution is to provide a definition for that missing API method that stands in and acts as if the older environment had already had it natively defined. This pattern is called a polyfill (aka "shim").

Consider this code:

```
// getSomeRecords() returns us a promise for some
// data it will fetch
var pr = getSomeRecords();

// show the UI spinner while we get the data
startSpinner();

pr
.then(renderRecords)   // render if successful
.catch(showError)      // show an error if not
.finally(hideSpinner)  // always hide the spinner
```

This code uses an ES2019 feature, the `finally(..)` method on the promise prototype. If this code were used in a pre-ES2019 environment, the `finally(..)` method would not exist, and an error would occur.

A basic polyfill for `finally(..)` in pre-ES2019 environments could look like this:

```
if (!Promise.prototype.finally) {
    Promise.prototype.finally = function f(fn){
        return this.then(fn,fn);
    };
}
```

NOTE: |
:— |
This is only a simple illustration of a naive polyfill for `finally(..)`. Don't use this approach in your code; always use a robust, official polyfill wherever possible, such as the collection of polyfills/shims in ES-Shim. |

The `if` statement protects the polyfill definition by preventing it from running in any environment where the JS engine has already defined that method. In older environments, the polyfill is defined, but in newer environments the `if` statement is quietly skipped.

Transpilers like Babel typically detect which polyfills your code needs and provide them automatically for you. But occassionally you may need to include/define them explicitly, which works similar to the above snippet.

Always write code using the most appropriate features to communicate its ideas and intent effectively. In general, this means using the most recent stable JS version. Avoid negatively impacting the code's readability by trying to manually adjust for the syntax/API gaps. That's what tools are for!

Transpilation and polyfilling are two highly effective techniques for addressing that gap between code that uses the latest stable features in the language and the old environments a site or application needs to still support. Since JS isn't going to stop improving, the gap will never go away. Both techniques should be embraced as a standard part of every JS project's production chain going forward.

What's In an Interpretation?

A long-debated question for code written in JS: is it an interpreted script or a compiled program? The majority opinion seems to be that JS is an interpreted (scripting) language. But the truth is more complicated than that.

For much of the history of programming languages, "interpreted" languages and "scripting" languages have been looked down on as inferior compared to their compiled counterparts. The reasons for this acrimony are numerous, including a perception of lack of performance optimization, as well as dislike of certain language characteristics, such as scripting languages generally using dynamic typing instead of the "more mature" statically typed languages.

Languages regarded as "compiled" usually produce a portable (binary) representation of the program that is distributed for execution later. Since we don't

really observe that kind of model with JS – we distribute the source code, not the binary form – many claim that disqualifies JS from the category. In reality, the distribution model for a program's "executable" form has become drastically more varied and also less relevant over the last few decades; to the question at hand, it doesn't really matter so much anymore what form of a program gets passed around.

These misinformed claims and criticisms should be set aside. The real reason it matters to have a clear picture on whether JS is interpreted or compiled relates to the nature of how errors are handled.

Historically, scripted or interpreted languages were executed in generally a top-down and line-by-line fashion; there's typically not an initial pass through the program to process it before execution begins (see Figure 1).

Fig. 1: Interpreted/Scripted Execution

In scripted or interpreted languages, an error on line 5 of a program won't be discovered until lines 1 through 4 have already executed. Notably, the error on line 5 might be due to a runtime condition, such as some variable or value having an unsuitable value for an operation, or it may be due to a malformed statement/command on that line. Depending on context, deferring error handling to the line the error occurs on may be a desirable or undesirable effect.

Compare that to languages which do go through a processing step (typically, called parsing) before any execution occurs, as illustrated in Figure 2:

Fig. 2: Parsing + Compilation + Execution

In this processing model, an invalid command (such as broken syntax) on line 5 would be caught during the parsing phase, before any execution has begun, and none of the program would run. For catching syntax (or otherwise "static") errors, generally it's preferred to know about them ahead of any doomed partial execution.

So what do "parsed" languages have in common with "compiled" languages? First, all compiled languages are parsed. So a parsed language is quite a ways down the road toward being compiled already. In classic compilation theory, the last remaining step after parsing is code generation: producing an executable form.

Once any source program has been fully parsed, it's very common that its subsequent execution will, in some form or fashion, include a translation from the parsed form of the program – usually called an Abstract Syntax Tree (AST) – to that executable form.

In other words, parsed languages usually also perform code generation before execution, so it's not that much of a stretch to say that, in spirit, they're compiled languages.

JS source code is parsed before it is executed. The specification requires as much, because it calls for "early errors" – statically determined errors in code, such as

a duplicate parameter name – to be reported before the code starts executing. Those errors cannot be recognized without the code having been parsed.

So **JS is a parsed language**, but is it *compiled*?

The answer is closer to yes than no. The parsed JS is converted to an optimized (binary) form, and that "code" is subsequently executed (Figure 2); the engine does not commonly switch back into line-by-line execution (like Figure 2) mode after it has finished all the hard work of parsing – most languages/engines wouldn't, because that would be highly inefficient.

To be specific, this "compilation" produces a binary byte code (of sorts), which is then handed to the "JS virtual machine" to execute. Some like to say this VM is "interpreting" the byte code. But then that means Java, and a dozen other JVM-driven languages, for that matter, are interpreted rather than compiled. Of course, that contradicts the typical assertion that Java/etc are compiled languages.

Interestingly, while Java and JavaScript are very different languages, the question of interpreted/compiled is pretty closely related between them!

Another wrinkle is that JS engines can employ multiple passes of JIT (Just-In-Time) processing/optimization on the generated code (post parsing), which again could reasonably be labeled either "compilation" or "interpretation" depending on perspective. It's actually a fantastically complex situation under the hood of a JS engine.

So what do these nitty gritty details boil down to? Step back and consider the entire flow of a JS source program:

1. After a program leaves a developer's editor, it gets transpiled by Babel, then packed by Webpack (and perhaps half a dozen other build processes), then it gets delivered in that very different form to a JS engine.
2. The JS engine parses the code to an AST.
3. Then the engine converts that AST to a kind-of byte code, a binary intermediate representation (IR), which is then refined/converted even further by the optimizing JIT compiler.
4. Finally, the JS VM executes the program.

To visualize thoses steps, again:

Fig. 3: Parsing, Compiling, and Executing JS

Is JS handled more like an interpreted, line-by-line script, as in Figure 1, or is it handled more like a compiled language that's processed in one-to-several passes first, before execution (as in Figures 2 and 3)?

I think it's clear that in spirit, if not in practice, **JS is a compiled language**.

And again, the reason that matters is, since JS is compiled, we are informed of static errors (such as malformed syntax) before our code is executed. That is a substantively different interaction model than we get with traditional "scripting" programs, and arguably more helpful!

Web Assembly (WASM)

One dominating concern that has driven a significant amount of JS's evolution is performance, both how quickly JS can be parsed/compiled and how quickly that compiled code can be executed.

In 2013, engineers from Mozilla Firefox demonstrated a port of the Unreal 3 game engine from C to JS. The ability for this code to run in a browser JS engine at full 60fps performance was predicated on a set of optimizations that the JS engine could perform specifically because the JS version of the Unreal engine's code used a style of code that favored a subset of the JS language, named "ASM.js".

This subset is valid JS written in ways that are somewhat uncommon in normal coding, but which signal certain important typing information to the engine that allow it to make key optimizations. ASM.js was introduced as one way of addressing the pressures on the run-time performance of JS.

But it's important to note that ASM.js was never intended to be code that was authored by developers, but rather a representation of a program having been transpiled from another language (such as C), where these typing "annotations" were inserted automatically by the tooling.

Several years after ASM.js demonstrated the validity of tooling-created versions of programs that can be processed more efficiently by the JS engine, another group of engineers (also, initially, from Mozilla) released Web Assembly (WASM).

WASM is similar to ASM.js in that it's original intent was to provide a path for non-JS programs (C, etc) to be converted to a form that could run in the JS engine. Unlike ASM.js, WASM chose to additionally get around some of the inherent delays in JS parsing/compilation before a program can execute, by representing the program in a form that is entirely unlike JS.

WASM is a representation format more akin to Assembly (hence, its name) that can be processed by a JS engine by skipping the parsing/compilation that the JS engine normally does. The parsing/compilation of a WASM-targeted program happen ahead of time (AOT); what's distributed is a binary-packed program ready for the JS engine to execute with very minimal processing.

An initial motivation for WASM was clearly the potential performance improvements. While that continues to be a focus, WASM is additionally motivated by the desire to bring more parity for non-JS languages to the web platform. For example, if a language like Go supports threaded programming, but JS (the language) does not, WASM offers the potential for such a Go program to be

converted to a form the JS engine can understand, without needing a threads feature in the JS language itself.

In other words, WASM relieves the pressure to add features to JS that are mostly/exclusively intended to be used by transpiled programs from other languages. That means JS feature development can judged (by TC39) without being skewed by interests/demands in other language ecosystems, while still letting those languages have a viable path onto the web.

Another perspective on WASM that's emerging is, interestingly, not even directly related to the web (W). WASM is evolving to become a cross-platform virtual machine (VM) of sorts, where programs can be compiled once and run in a variety of different system environments.

So, WASM isn't only for the web, and WASM also isn't JS. Ironically, even though WASM runs in the JS engine, the JS language is one of the least suitable languages to source WASM programs with, because WASM relies heavily on static typing information. Even TypeScript (TS) – ostensibly, JS + static types – is not quite suitable (as it stands) to transpile to WASM , though language variants like AssemblyScript are attempting to bridge the gap between JS/TS and WASM.

This book isn't about WASM, so I won't spend much more time discussing it, except to make one final point. *Some* folks have suggested WASM points to a future where JS is excised from, or minimized in, the web. These folks often harbor ill feelings about JS, and want some other language – any other language! – to replace it. Since WASM lets other languages run in the JS engine, on its face this isn't an entirely fanciful fairytale.

But let me just state simply: WASM will not replace JS. WASM significantly augments what the web (including JS) can accomplish. That's a great thing, entirely orthogonal to whether some people will use it as an escape hatch from having to write JS.

***Strict*ly Speaking**

Back in 2009 with the release of ES5, JS added *strict mode* as an opt-in mechanism for encouraging better JS programs.

The benefits of strict mode far outweigh the costs, but old habits die hard and the inertia of existing (aka "legacy") code bases is really hard to shift. So sadly, more than 10 years later, strict mode's *optionality* means that it's still not necessarily the default for JS programmers.

Why strict mode? Strict mode shouldn't be thought of as a restriction on what you can't do, but rather as a guide to the best way to do things so that the JS engine has the best chance of optimizing and efficiently running the code. Most JS code is worked on by teams of developers, so the *strict*-ness of strict mode

(along with tooling like linters!) often helps collaboration on code by avoiding some of the more problematic mistakes that slip by in non-strict mode.

Most strict mode controls are in the form of *early errors*, meaning errors that aren't strictly syntax errors but are still thrown at compile time (before the code is run). For example, strict mode disallows naming two function parameters the same, and results in an early error. Some other strict mode controls are only observable during runtime, such as how the `this` keyword defaults to `undefined` instead of the global object.

Rather than fighting and arguing with strict mode, like a kid who just wants to defy whatever their parents tell them not to do, the best mindset is that strict mode is like a linter reminding you how JS *should* be written to have the highest quality and best chance at performance. If you find yourself feeling handcuffed, trying to work around strict mode, that should be a blaring red warning flag that you need to back up and rethink the whole approach.

Strict mode is switched on per program (per file!) like this:

```
// only whitespace and comments are allowed
// before the use-strict pragma

"use strict";

// everything in this file runs in strict
// mode
```

The strict mode pragma must appear at the top of a file, with only whitespace or comments being allowed before it.

WARNING: |
:— |
Something to be aware of is that even a stray `;` all by itself appearing before the strict mode pragma will render the pragma useless; no errors are thrown because it's valid JS to have a string literal expression in a statement position, but it also will silently *not* turn on strict mode! |

Strict mode can alternatively be turned on per-function scope, with exactly the same rules/admonitions about its positioning):

```
function someOperations() {
    // whitespace and comments are fine here
    "use strict";

    // all this code will run in strict mode
}
```

Interestingly, if a file has strict mode turned on, the function-level strict mode pragmas are disallowed. So you have to pick one or the other.

The **only** valid reason to use a per-function approach to strict mode is when you are converting an existing non-strict mode program file and need to make the changes little by little over time. Otherwise, it's vastly better to simply turn strict mode on for the entire file/program.

Many have wondered if there would ever be a time when JS made strict mode the default? The answer is, almost certainly not. As we discussed earlier around backwards compatibility, if a JS engine update started assuming code was strict mode even if it's not marked as such, it's possible that this code would break as a result of strict mode's controls.

However, there are a few factors that reduce the future impact of this non-default "obscurity" of strict mode.

For one, virtually all transpiled code ends up in strict mode even if the original source code isn't written as such. Most JS code in production has been transpiled, so that means most JS is already adhering to strict mode It's possible to undo that assumption, but you really have to go out of your way to do so, so it's highly unlikely.

Moreover, a wide shift is happening towards more/most new JS code being written using the ES6 module format. ES6 modules assume strict mode, so all code in such files is automatically defaulted to strict mode.

Taken together, strict mode is largely the de facto default even though technically it's not actually the default.

Defined

JS is an implementation of the ECMAScript standard (version ES2019 as of this writing), which is guided by the TC39 committee and hosted by ECMA. It runs in browsers and other JS environments such as Node.js.

JS is a multi-paradigm language, meaning the syntax and capabilities allow a developer to mix-and-match (and bend and reshape!) concepts from various major paradigms, such as procedural, object-oriented (OO/classes), and functional (FP).

JS is a compiled language, meaning the tools (including the JS engine) process and verify a program (reporting any errors!) before it executes.

With our language now *defined*, let's start getting to know its ins and outs.

You Don't Know JS Yet: Get Started - 2nd Edition

Chapter 2: Surveying JS

The best way to learn JS is to start writing JS.

To do that, you need to know how the language works, and that's what we'll focus on here. Even if you've programmed in other languages before, take your time getting comfortable with JS, and make sure to practice each piece.

This chapter is not an exhaustive reference on every bit of syntax of the JS language. It's also not intended to be a complete "intro to JS" primer.

Instead, we're just going to survey some of the major topic-areas of the language. Our goal is to get a better *feel* for it, so that we can move forward writing our own programs with more confidence. We'll revisit many of these topics in successively more detail as you go through the rest of this book, and the rest of the series.

Please don't expect this chapter to be a quick read. It's long and there's plenty of detail to chew on. Take your time.

TIP: |
:— |
If you're still getting familiar with JS, I suggest you reserve plenty of extra time to work through this chapter. Take each section and ponder and explore the topic for awhile. Look through existing JS programs and compare what you see in them to the code and explanations (and opinions!) presented here. You will get a lot more out of the rest of the book and series with a solid foundation of JS's *nature*. |

Each File Is A Program

Almost every website (web application) you use is comprised of many different JS files (typically with the .js file extension). It's tempting to think of the whole thing (the application) as one program. But JS sees it differently.

In JS, each standalone file is its own separate program.

The reason this matters is primarily around error handling. Since JS treats files as programs, one file may fail (during parse/compile or execution) and that will not necessarily prevent the next file from being processed. Obviously, if your application depends on five .js files, and one of them fails, the overall application will probably only partially operate, at best. It's important to ensure that each file works properly, and that to whatever extent possible, they handle failure in other files as gracefully as possible.

It may surprise you to consider separate .js files as separate JS programs. From the perspective of your usage of an application, it sure seems like one big program. That's because the execution of the application allows these individual *programs* to cooperate and act as one program.

The only way multiple standalone .js files act as a single program is by sharing their state (and access to their public functionality) via the "global scope". They mix together in this global scope namespace, so at runtime they act as as whole.

NOTE: |
:— |
Many projects use build process tools that end up combining separate files from the project into a single file to be delivered to a web page. When this happens, JS treats this single combined file as the entire program. |

Since ES6, JS has also supported a module format in addition to the typical standalone JS program format. Modules are also file-based. If a file is loaded via module-loading mechanism such as an `import` statement or a `<script type=module>` tag, all its code is treated as a single module.

Though you wouldn't typically think about a module – basically, a collection of state and publicly-exposed methods to operate on that state – as a standalone program, JS does in fact still treat each module separately. Similar to how "global scope" allows standalone files to mix together at runtime, importing a module into another allows runtime interoperation between them.

Regardless of which code organization pattern (and loading mechanism) is used for a file – standalone or module – you should still think of each file as its own (mini) program, which may then cooperate with other (mini) programs to perform the functions of your overall application.

Values

The most fundamental unit of information in a program is a value. Values are data. They're how the program maintains state. Values come in two forms in JS: **primitive** and **object**.

Values are embedded in our programs using *literals*. For example:

```
console.log("My name is Kyle.");
// My name is Kyle.
```

In this program, the value `"My name is Kyle."` is a primitive string literal; strings are ordered collections of characters, usually used to represent words and sentences.

I used the double-quote `"` character to *delimit* (surround, separate, define) the string value. But I could have used the single-quote `'` character as well. The

choice of which quote character is entirely stylistic. The important thing, for code readability and maintainability sake, is to pick one and to use it consistently throughout the program.

Another option to delimit a string literal is to use the back-tick ` character. However, this choice is not merely stylistic; there's a behavioral difference as well. Consider:

```
console.log("My name is ${ firstName }.");
// My name is ${ firstName }.

console.log('My name is ${ firstName }.');
// My name is ${ firstName }.

console.log(`My name is ${ firstName }.`);
// My name is Kyle.
```

Assuming this program has already defined a variable `firstName` with the string value `"Kyle"`, the `-delimited string then resolves the variable expression (indicated with `${ .. }`) to its current value. This is called **interpolation**.

The back-tick `-delimited string can be used without including interpolated expressions, but that defeats the whole purpose of that alternate string literal syntax:

```
console.log(`Am I confusing you by omitting interpolation?`);
// Am I confusing you by omitting interpolation?
```

The better approach is to use `"` or `'` (again, pick one and stick to it!) for strings *unless you need* interpolation; reserve ` only for strings that will include interpolated expressions.

Other than strings, JS programs often contain other primitive literal values such as booleans and numbers.

```
while (false) {
    console.log(3.141592);
}
```

`while` represents a loop type, a way to repeat operations *while* its condition is true.

In this case, the loop will never run (and nothing will be printed), because we used the `false` boolean value as the loop conditional. `true` would have resulted in a loop that keeps going forever, so be careful!

The number `3.141592` is, as you may know, an approximation of mathematical PI to the first six digits. Rather than embed such a value, however, you would typically use the predefined `Math.PI` value for that purpose. Another variation on numbers is the `bigint` (big-integer) primitive type, which is used for storing arbitrarily large numbers.

Numbers are most often used in programs for counting steps, such as loop iterations, and accessing information in numeric positions (ie, an array index). We'll cover arrays/objects in a little bit, but as an example, if there was an array called `names`, we could access the element in its second position like this:

```
console.log(`My name is ${ names[1] }.`);
// My name is Kyle.
```

We used `1` for the element in the second position, instead of `2`, because like in most programming languages, JS array indices are 0-based (`0` is the first position).

In addition to strings, numbers, and booleans, two other *primitive* values in JS programs are `null` and `undefined`. While there are differences between them (some historic and some contemporary), for the most part both values serve the purpose of indicating *emptiness* (or absence) of a value.

Many developers prefer to treat them both consistently in this fashion, which is to say that the values are assumed to be indistinguishable. If care is taken, this is often possible. However, it's safest and best to use only `undefined` as the single empty value, even though `null` seems attractive in that it's shorter to type!

```
while (value != undefined) {
    console.log("Still got something!");
}
```

The final primitive value to be aware of is a symbol, which is a special-purpose value that behaves as a hidden unguessable value. Symbols are almost exclusively used as special keys on objects.

```
hitchhikersGuide[ Symbol("meaning of life") ];
// 42
```

You won't encounter direct usage of symbols very often in typical JS programs. They're mostly used in low-level code such as in libraries and frameworks.

Arrays And Objects

Besides primitives, the other value type in JS is an object value.

As mentioned earlier, arrays are a special type of object that's comprised of an ordered and numerically indexed list of data:

```
names = [ "Frank", "Kyle", "Peter", "Susan" ];

names.length;
// 4

names[0];
// Frank

names[1];
// Kyle
```

JS arrays can hold any value type, either primitive or object (including other arrays). As we'll see towards the end of Chapter 3, even functions are values that can be held in arrays or objects.

NOTE: |
:— |
Functions, like arrays, are a special kind (aka, sub-type) of object. We'll cover functions in more detail in a bit. |

Objects are more general: an unordered, keyed collection of any various values. In other words, you access the element by a string location name (aka "key" or "property") rather than by its numeric position (as with arrays). For example:

```
name = {
    first: "Kyle",
    last: "Simpson",
    age: 39,
    specialties: [ "JS", "Table Tennis" ]
};

console.log(`My name is ${ name.first }.`);
```

Here, `name` represents an object, and `first` represents the name of a location of information in that object (value collection). Another syntax option that accesses information in an object by its property/key uses the square-brackets `[ ]`, such as `name["first"]`.

Value Type Determination

For distinguishing values, the `typeof` operator tells you its built-in type, if primitive, or `"object"` otherwise:

```
typeof 42;                  // "number"
typeof "abc";               // "string"
typeof true;                // "boolean"
typeof undefined;           // "undefined"
typeof null;                // "object" -- oops, JS bug!
typeof { "a": 1 };          // "object"
typeof [1,2,3];             // "object"
typeof function Hello(){};  // "function"
```

WARNING: |
:— |
`typeof null` unfortunately returns `"object"` instead of the expected `"null"`. Also, `typeof` returns the specific `"function"` for functions, but not the expected `"array"` for arrays. |

Converting from one value type to another, such as from string to number, is referred to in JS as "coercion". We'll cover this in more detail later in this chapter.

Primitive values and object values behave differently when they're assigned or passed around. We'll cover these details in Appendix A, "Values vs References".

Declaring And Using Variables

To be explicit about something that may not have been obvious in the previous section: in JS programs, values can either appear as literal values (as many of the above examples illustrate), or they can be held in variables; think of variables as just containers for values.

Variables have to be declared (created) to be used. There are various syntax forms that declare variables (aka, "identifiers"), and each form has different implied behaviors.

For example, consider the `var` statement:

```
var name = "Kyle";
var age;
```

The `var` keyword declares a variable to be used in that part of the program, and optionally allows an initial assignment of a value.

Another similar keyword is `let`:

```
let name = "Kyle";
let age;
```

The `let` keyword has some differences to `var`, with the most obvious being that `let` allows a more limited access to the variable than `var`. This is called "block scoping" as opposed to regular or function scoping.

Consider:

```
var adult = true;

if (adult) {
    var name = "Kyle";
    let age = 39;
    console.log("Shhh, this is a secret!");
}

console.log(name);
// Kyle

console.log(age);
// Error!
```

The attempt to access `age` outside of the `if` statement results in an error, because `age` was block-scoped to the `if`, whereas `name` was not.

Block-scoping is very useful for limiting how widespread variable declarations are in our programs, which helps prevent accidental overlap of their names.

But `var` is still useful in that it communicates "this variable will be seen by a wider scope". Both declaration forms can be appropriate in any given part of a program, depending on the circumstances.

NOTE: |
:— |
It's very common to suggest that `var` should be avoided in favor of `let` (or `const`!), generally because of perceived confusion over how the scoping behavior of `var` has worked since the beginning of JS. I believe this to be overly restrictive advice and ultimately unhelpful. It's assuming you are unable to learn and use a feature properly in combination with other features. I believe you *can* and *should* learn any features available, and use them where appropriate! |

A third declaration form is `const`. It's like `let` but has an additional limitation that it must be given a value at the moment it's declared, and cannot be re-assigned a different value later.

Consider:

```
const myBirthday = true;
let age = 39;

if (myBirthday) {
    age = age + 1;    // OK!
    myBirthday = false;  // Error!
}
```

The `myBirthday` constant is not allowed to be re-assigned.

`const` declared variables are not "unchangeable", they just cannot be re-assigned. It's ill-advised to use `const` with object values, because those values can still be changed even though the variable can't be re-assigned. This leads to potential confusion down the line, so I think it's wise to avoid situations like:

```
const actors = [ "Morgan Freeman", "Jennifer Aniston" ];

actors[2] = "Tom Cruise";   // OK :(

actors = [];                // Error!
```

The best semantic use of a `const` is when you have a simple primitive value that you want to give a useful name to, such as using `myBirthday` instead of `true`. This makes programs easier to read.

TIP: |
:— |
If you stick to using `const` only with primitive values, you never have the confusion of the difference between re-assignment (not allowed) and mutation (allowed)! That's the safest and best way to use `const`. |

Besides `var` / `let` / `const`, there are other syntactic forms that declare identifiers (variables) in various scopes. For example:

```
function hello(name) {
    console.log(`Hello, ${ name }.`);
}

hello("Kyle");
// Hello, Kyle.
```

Here, the identifier `hello` is created in the outer scope, and it's also automatically associated so that it references the function. But the named parameter `name` is created only inside the function, and thus is only accessible inside that function's scope.

Both `hello` and `name` act generally as if they were declared with `var`.

Another syntax that declares a variable is the `catch` clause of a `try..catch` statement:

```
try {
    someError();
}
catch (err) {
    console.log(err);
}
```

The `err` is a block-scoped variable that exists only inside the `catch` clause, as if it had been declared with `let`.

Functions

The word "function" has a variety of meanings in programming. For example, in the world of Functional Programming, "function" has a precise mathematical definition and implies a strict set of rules to abide by.

In JS, we should consider "function" to take the broader meaning of another related term: "procedure". A procedure is a collection of statements that can be invoked one or more times, may be provided some inputs, and may give back one or more outputs.

In earlier days of JS, the way to define a function looked like this:

```
function awesomeFunction(coolThings) {
    // ..
    return amazingStuff;
}
```

This is called a function declaration because it appears as a statement by itself, not as an expression that's part of another statement. The association between the identifier `awesomeFunction` and the function value happens immediately during the compile phase of the code, before that code is executed.

In contrast to a function declaration statement, a function expression can be defined and assigned like this:

```
// let awesomeFunction = ..
// const awesomeFunction = ..
var awesomeFunction = function(coolThings) {
    // ..
    return amazingStuff;
};
```

This function is an expression that is assigned to the variable `awesomeFunction`. Different from the function declaration form, a function expression is not associated with its identifier until that statement during runtime.

It's extremely important to note that in JS, functions are values that can be assigned (as shown in this snippet) and passed around. In fact, JS functions are a special type of the object value type. Not all languages treat functions as values, but it's essential for a language to support the Functional Programming pattern, as JS does.

JS functions can receive parameter input:

```
function greeting(myName) {
    console.log(`Hello, ${ myName }!`);
}

greeting("Kyle");
// Hello, Kyle!
```

In this snippet, `myName` is called a parameter, which acts as a local variable inside the function. Functions can be defined to receive any number of parameters, from none upward, as you see fit. Each parameter is assigned the argument value that you pass in that position (`"Kyle"`, here) of the function call.

Functions also can return values using the `return` keyword:

```
function greeting(myName) {
    return `Hello, ${ myName }!`;
}

var msg = greeting("Kyle");

console.log(msg);
// Hello, Kyle!
```

You can only `return` a single value, but if you have more values to return, you can wrap them up into a single object/array.

Since functions are values, they can be assigned as properties on objects:

```
var whatToSay = {
    greeting() {
        console.log("Hello!");
    },
    question() {
        console.log("What's your name?");
```

```
    },
    answer() {
        console.log("My name is Kyle.");
    }
};

whatToSay.greeting();
// Hello!
```

In this snippet, references to three functions (`greeting()`, `question()`, and `answer()`) are included in the object held by `whatToSay`. Each function can be called by accessing the property to retrieve the function reference value. Compare this straightforward style of defining functions on an object to the more sophisticated `class` syntax discussed later in this chapter.

There are many varied forms that `function`s take in JS. We dig into these variations in Appendix A, "So Many Function Forms".

Comparisons

Making decisions in programs requires comparing values to determine their identity and relationship to each other. JS has several mechanisms to enable value comparison, so let's take a closer look at them.

Equal. . . ish

The most common comparison in JS programs asks the question, "is this X value *the same as* that Y value?" What exactly does "the same as" really mean to JS, though?

For ergonomic and historical reasons, the meaning is more complicated than the obvious *exact identity* sort of matching. Sometimes an equality comparison intends *exact* matching, but other times the desired comparison is a bit broader, allowing *closely similar* or *interchangable* matching. In other words, we must be aware of the nuanced differences between an **equality** comparison and an **equivalence** comparison.

If you've spent any time working with and reading about JS, you've certainly seen the so called "triple-equals" `===` operator, also described as the "strict equality" operator. That seems rather straight forward, right? Surely, "strict" means strict, as in narrow and *exact*.

Not *exact*ly.

Yes, most values participating in an `===` equality comparison will fit with that *exact same* intuition. Consider some examples:

```
3 === 3.0;              // true
"yes" === "yes";        // true
null === null;          // true
false === false;        // true

42 === "42";            // false
"hello" === "Hello";    // false
true === 1;             // false
0 === null;             // false
"" === null;            // false
null === undefined;     // false
```

NOTE: |
:— |
Another way `===`'s equality comparison is often described is, "checking both the value and the type". In several of the above examples, like `42 === "42"`, the *type* of both values – number, string, etc – does seem to be the distinguishing factor. There's more to it than that, though. **All** value comparisons in JS consider the type of the values being compared, not *just* the `===` operator. Specifically, `===` disallows any sort of type conversion (aka, "coercion") in its comparison, where other JS comparisons *do* allow coercion. |

But the `===` operator does have some nuance to it, a fact many JS developers gloss over, to their detriment. The `===` operator is designed to *lie* in two cases of special values: `NaN` and `-0`. Consider:

```
NaN === NaN;            // false
0 === -0;               // true
```

In the case of `NaN`, the `===` operator *lies* and says that an occurrence of `NaN` is not equal to another `NaN`. In the case of `-0` – yes, this is a real, distinct value you can use intentionally in your programs! – the `===` operator *lies* and says it's equal to the regular `0` value.

TIP: |
:— |
When making such comparisons, since the *lying* is likely bothersome, don't use `===`. For `NaN` comparisons, use the `Number.isNaN(..)` utility, which does not *lie*. For `-0` comparison, use the `Object.is(..)` utility, which also does not *lie*. `Object.is(..)` can also be used for non-*lying* `NaN` checks, if you prefer. Humorously, you could think of `Object.is(..)` as the "quadruple-equals" `====`, the really-really-strict comparison! |

There are deeper historical and technical reasons for these *lies*, but that doesn't change the fact that `===` is not actually *strictly exactly equal* comparison, in the *strictest* sense.

The story gets even more complicated when we consider comparisons of object values (non-primitives). Consider:

```
[ 1, 2, 3 ] === [ 1, 2, 3 ];    // false
{ a: 42 } === { a: 42 }         // false
(x => x * 2) === (x => x * 2)   // false
```

What's going on here?

It may seem reasonable to assume that an equality check considers the *nature* or *contents* of the value; after all, `42 === 42` considers the actual `42` value and compares it. But when it comes to objects, a content-aware comparison is generally referred to as "structural equality".

JS does not define `===` as *structural equality* for object values. Instead, `===` uses *identity equality* for object values.

In JS, all object values are held by reference (see "Values vs References" in Appendix A), are assigned and passed by reference-copy, **and** to our current discussion, are compared by reference (identity) equality. Consider:

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

// assignment is by reference-copy, so
// y references the *same* array as x,
// not another copy of it.
var y = x;

y === x;              // true
y === [ 1, 2, 3 ];    // false
x === [ 1, 2, 3 ];    // false
```

In this snippet, `y === x` is true because both variables hold a reference to the same initial array. But the `=== [1,2,3]` comparisons both fail because `y` and `x`, respectively, are being compared to new *different* arrays `[1,2,3]`. The array structure and contents don't matter in this comparison, only the **reference identity**.

JS does not provide a mechanism for structural equality comparison of object values, only reference identity comparison. To do structural equality comparison, you'll need to implement the checks yourself.

But beware, it's more complicated than you'll assume. For example, how might you determine if two function references are "structurally equivalent"? Even stringifying to compare their source code text wouldn't take into account things like closure. JS doesn't provide structural equality comparison because it's almost intractable to handle all the corner cases!

Coercive Comparisons

As mentioned earlier, coercion means a value of one type being converted to its respective representation in another type (to whatever extent possible). As we'll discuss in Chapter 4, coercion is a core pillar of the JS language, not some optional feature that can reasonably be avoided.

But where coercion meets comparison operators (like equality), confusion and frustration unfortunately crop up more often than not.

Few JS features draw more ire in the broader JS community than the `==` operator, generally referred to as the "loose equality" operator. The majority of all writing and public discourse on JS condemns this operator as poorly designed and dangerous/bug-ridden when used in JS programs. Even the creator of the language himself, Brendan Eich, has lamented how it was designed as a big mistake.

From what I can tell, most of this frustration comes from a pretty short list of confusing corner cases, but a deeper problem is the extremely widespread misconception that it performs its comparisons without considering the types of its compared values.

The `==` operator performs an equality comparison similarly to how the `===` performs it. In fact, both operators consider the type of the values being compared. And if the comparison is between the same value type, both `==` and `===` **do exactly the same thing, no difference whatsoever.**

If the value types being compared are different, the `==` differs from `===` in that it allows coercion before the comparison. In other words, they both want to compare values of like types, but `==` allows type conversions *first*, and once the types have been converted to be the same on both sides, then `==` does the same thing as `===`. Instead of "loose equality", the `==` operator should be described as "coercive equality".

Consider:

```
42 == "42";             // true
1 == true;              // true
```

In both comparisons, the value types are different, so the `==` causes the non-number values (`"42"` and `true`) to be converted to numbers (`42` and `1`, respectively) before the comparisons are made.

Just being aware of this nature of `==` – that it prefers primitive and numeric comparisons – helps you avoid most of the troublesome corner cases, such as staying away from a gotchas like `"" == 0` or `0 == false`.

You may be thinking, "Oh, well, I will always just avoid any coercive equality comparison (using `===` instead) to avoid those corner cases"! Eh, sorry, that's not quite as likely as you would hope.

There's a pretty good chance that you'll use relational comparison operators like `<`, `>` (and even `<=` and `>=`).

Just like `==`, these operators will perform as if they're "strict" if the types being relationally compared already match, but they'll allow coercion first (generally, to numbers) if the types differ.

Consider:

```
var arr = [ "1", "10", "100", "1000" ];
for (let i = 0; i < arr.length && arr[i] < 500; i++) {
    // will run 3 times
}
```

The `i < arr.length` comparison is "safe" from coercion because `i` and `arr.length` are always numbers. The `arr[i] < 500` invokes coercion, though, because the `arr[i]` values are all strings. Those comparisons thus become `1 < 500`, `10 < 500`, `100 < 500`, and `1000 < 500`. Since that last one is false, the loop stops after its third iteration.

These relational operators typically use numeric comparisons, except in the case where **both** values being compared are already strings; in this case, they use alphabetical (dictionary-like) comparison of the strings:

```
var x = "10";
var y = "9";

x < y;      // true, watch out!
```

There's no way to get these relational operators to avoid coercion, other than to just never use mismatched types in the comparisons. That's perhaps admirable as a goal, but it's still pretty likely you're going to run into a case where the types *may* differ.

The wiser approach is not to avoid coercive comparisons, but to embrace and learn their ins and outs.

Coercive comparisons crop up in other places in JS, such as conditionals (`if`, etc), which we'll revisit in Appendix A, "Coercive Conditional Comparison".

How We Organize In JS

Two major patterns for organizing code (data and behavior) are used broadly across the JS ecosystem: classes and modules. These patterns are not mutually exclusive; many programs can and do use both. Other programs will stick with just one pattern, or even neither!

In some respects, these patterns are very different. But interestingly, in other ways, they're just different sides of the same coin. Being proficient in JS requires understanding both patterns and where they are appropriate (and not!).

Classes

The terms "object oriented", "class oriented" and "classes" are all very loaded full of detail and nuance; they're not universal in definition.

We will use a common and somewhat traditional definition here, the one most likely familiar to those with backgrounds in "object oriented" languages like C++ and Java.

A class in a program is a definition of a "type" of custom data structure that includes both data and behaviors that operate on that data. Classes define how such a data structure works, but classes are not themselves concrete values. To get a concrete value that you can use in the program, a class must be *instantiated* (with the `new` keyword) one or more times.

Consider:

```
class Page {
    constructor(text) {
        this.text = text;
    }

    print() {
        console.log(this.text);
    }
}

class Notebook {
    constructor() {
        this.pages = [];
    }

    addPage(text) {
        var page = new Page(text);
        this.pages.push(page);
    }

    print() {
        for (let page of this.pages) {
            page.print();
        }
    }
```

```
}

var mathNotes = new Notebook();
mathNotes.addPage("Arithmetic: + - * / ...");
mathNotes.addPage("Trigonometry: sin cos tan ...");

mathNotes.print();
// ..
```

In the `Page` class, the data is a string of text stored in a `this.text` member property. The behavior is `print()`, a method that dumps the text to the console.

For the `Notebook` class, the data is an array of `Page` instances. The behavior is `addPage(..)`, a method that instantiates new `Page` pages and adds them to the list, as well as `print()` which prints out all the pages in the notebook.

The statement `mathNotes = new Notebook()` creates an instance of the `Notebook` class, and `page = new Page(text)` is where instances of the `Page` class are created.

Behavior (methods) can only be called on instances (not the classes themselves), such as `mathNotes.addPage(..)` and `page.print()`.

The `class` mechanism allows packaging data (`text` and `pages`) to be organized together with their behaviors (`addPage(..)`, `print()`). The same program could have been built without any `class` definitions, but it would likely have been much less organized, harder to read and reason about, and more susceptible to bugs and subpar maintenance.

Class Inheritance Another aspect inherent to traditional "class oriented" design, though a bit less commonly used in JS, is "inheritance" (and "polymorphism"). Consider:

```
class Publication {
    constructor(title,author,pubDate) {
        this.title = title;
        this.author = author;
        this.pubDate = pubDate;
    }

    print() {
        console.log(`
            Title: ${ this.title }
            By: ${ this.author }
            ${ this.pubDate }
        `);
    }
```

```
}
```

This `Publication` class defines a set of common behavior that any publication might need.

Now let's consider more specific types of publication, like `Book` and `BlogPost`:

```
class Book extends Publication {
    constructor(bookDetails) {
        super(
            bookDetails.title,
            bookDetails.author,
            bookDetails.publishedOn
        );
        this.publisher = bookDetails.publisher;
        this.ISBN = bookDetails.ISBN;
    }

    print() {
        super.print();
        console.log(`
            Published By: ${ this.publisher }
            ISBN: ${ this.ISBN }
        `);
    }
}

class BlogPost extends Publication {
    constructor(title,author,pubDate,URL) {
        super(title,author,pubDate);
        this.URL = URL;
    }

    print() {
        super.print();
        console.log(this.URL);
    }
}
```

Both `Book` and `BlogPost` use the `extends` clause to *extend* the general definition of `Publication` to include additional behavior. The `super(..)` call in each constructor delegates to the parent `Publication` class's constructor for its initialization work, and then they do more specific things according to their respective publication type (aka, "sub-class" or "child class").

Now consider using these child classes:

```
var YDKJS = new Book({
    title: "You Don't Know JS",
    author: "Kyle Simpson",
    publishedOn: "June 2014",
    publisher: "O'reilly",
    ISBN: "123456-789"
});

YDKJS.print();
// Title: You Don't Know JS
// By: Kyle Simpson
// June 2014
// Published By: O'reilly
// ISBN: 123456-789

var forAgainstLet = new BlogPost(
    "For and against let",
    "Kyle Simpson",
    "October 27, 2014",
    "https://davidwalsh.name/for-and-against-let"
);

forAgainstLet.print();
// Title: For and against let
// By: Kyle Simpson
// October 27, 2014
// https://davidwalsh.name/for-and-against-let
```

Notice that both child class instances have a `print()` method, which was an override of the *inherited* `print()` method from the parent `Publication` class. Each of those overridden child class `print()` methods call `super.print()` to invoke the inherited version of the `print()` method.

The fact that both the inherited and overriden methods can have the same name and co-exist is called *polymorphism*.

Inheritance is a powerful tool for organizing data/behavior in separate logical units (classes), but allowing the child class to cooperate with the parent by accessing/using its behavior and data.

Modules

The module pattern has essentially the same goal as the class pattern, which is to group data and behavior together into logical units. Also like classes, modules can "include" or "access" the data and behaviors of other modules, for cooperation sake.

But modules have some important differences from classes. Most notably, the syntax is entirely different.

Classic Modules ES6 added a module syntax form to native JS syntax, which we'll look at in a moment. But from the early days of JS, modules was an important and common pattern that was leveraged in countless JS programs, even without a dedicated syntax.

The key hallmarks of a *classic module* are an outer function (that runs at least once), which returns an "instance" of the module with one or more functions exposed that can operate on the module instance's internal (hidden) data.

Because a module of this form is *just a function*, and calling it produces an "instance" of the module, another description for these functions is "module factories".

Consider the classic module form of the earlier `Publication`, `Book`, and `BlogPost` classes:

```
function Publication(title,author,pubDate) {
    var publicAPI = {
        print() {
            console.log(`
                Title: ${ title }
                By: ${ author }
                ${ pubDate }
            `);
        }
    };

    return publicAPI;
}

function Book(bookDetails) {
    var pub = Publication(
        bookDetails.title,
        bookDetails.author,
        bookDetails.publishedOn
    );

    var publicAPI = {
        print() {
            pub.print();
            console.log(`
                Published By: ${ bookDetails.publisher }
                ISBN: ${ bookDetails.ISBN }
```

```
            `);
        }
    };

    return publicAPI;
}

function BlogPost(title,author,pubDate,URL) {
    var pub = Publication(title,author,pubDate);

    var publicAPI = {
        print() {
            pub.print();
            console.log(URL);
        }
    };

    return publicAPI;
}
```

Comparing these forms to the `class` forms, there are more similarities than differences.

The `class` form stores methods and data on an object instance, which must be accessed with the `this.` prefix. With modules, the methods and data are accessed as identifier variables in scope, without any `this.` prefix.

With `class`, the "API" of an instance is implicit in the class definition – also, all data and methods are public. With the module factory function, you explicitly create and return an object with any publicly exposed methods, and any data or other unreferenced methods remain private inside the factory function.

There are other variations to this factory function form that are quite common across JS, even in 2019; you may run across these forms in different JS programs: AMD ("Asynchronous Module Definition"), UMD ("Universal Module Definition"), and CommonJS (classic Node.js style modules). The variations, however, are minor (yet not quite compatible). Still, all of these forms rely on the same basic principles.

Consider also the usage (aka, "instantiation") of these module factory functions:

```
var YDKJS = Book({
    title: "You Don't Know JS",
    author: "Kyle Simpson",
    publishedOn: "June 2014",
    publisher: "O'reilly",
    ISBN: "123456-789"
```

```
});

YDKJS.print();
// Title: You Don't Know JS
// By: Kyle Simpson
// June 2014
// Published By: O'reilly
// ISBN: 123456-789

var forAgainstLet = BlogPost(
    "For and against let",
    "Kyle Simpson",
    "October 27, 2014",
    "https://davidwalsh.name/for-and-against-let"
);

forAgainstLet.print();
// Title: For and against let
// By: Kyle Simpson
// October 27, 2014
// https://davidwalsh.name/for-and-against-let
```

The only observable difference here is the lack of using `new`, calling the module factories as normal functions.

ES Modules ES modules (ESM), introduced to the JS language in ES6, are meant to serve much the same spirit and purpose as the existing *classic modules* just described, especially taking into account important variations and use-cases from AMD, UMD, and CommonJS.

The implementation approach does however differ significantly.

First, there's no wrapping function to *define* a module. The wrapping context is a file. ESMs are always file-based; one file, one module.

Second, you don't interact with a module's "API" explicitly, but rather use the `export` keyword to add a variable or method to its public API definition. If something is defined in a module but not `export`ed, then it stays hidden (just as with *classic modules*).

Third, and maybe most noticeably different from previously discussed patterns, you don't "instantiate" an ES module, you just `import` it to use its single instance. ESMs are, in effect, "singletons", in that there's only one instance ever created, at first `import` in your program, and all other `import`s just receive a reference to that same single instance. If your module needs to support multiple instantiations, you have to provide a *classic module* style factory function on your ESM definition for that purpose.

In our running example, we do assume multiple-instantiation, so these following snippets will mix both ESM and *classic modules*:.

Consider the file `publication.js`:

```
function printDetails(title,author,pubDate) {
    console.log(‘
        Title: ${ title }
        By: ${ author }
        ${ pubDate }
    ‘);
}

export function create(title,author,pubDate) {
    var publicAPI = {
        print() {
            printDetails(title,author,pubDate);
        }
    };

    return publicAPI;
}
```

To import and use this module, from another ES module like `blogpost.js`:

```
import { create as createPub } from "publication.js";

function printDetails(pub,URL) {
    pub.print();
    console.log(URL);
}

export function create(title,author,pubDate,URL) {
    var pub = createPub(title,author,pubDate);

    var publicAPI = {
        print() {
            printDetails(pub,URL);
        }
    };

    return publicAPI;
}
```

And finally, to use this module, we import into another ES module like `main.js`:

```
import { create as createBlogPost } from "blogpost.js";

var forAgainstLet = createBlogPost(
    "For and against let",
    "Kyle Simpson",
    "October 27, 2014",
    "https://davidwalsh.name/for-and-against-let"
);

forAgainstLet.print();
// Title: For and against let
// By: Kyle Simpson
// October 27, 2014
// https://davidwalsh.name/for-and-against-let
```

NOTE: |
:— |
The `as createBlogPost` clause in the `import` statement above is optional; if omitted, a top level function just named `create(..)` would be imported. In this case, I'm renaming it for readability sake; its more generic factory name of `create(..)` becomes more semantically descriptive of its purpose as `createBlogPost(..)`. |

As shown, ES modules can use *classic modules* internally if they need to support multiple-instantiation. Alternatively, we could have exposed a `class` from our module instead of a `create(..)` factory function, with generally the same outcome. However, since you're already using ESM at that point, I'd recommend sticking with *classic modules* instead of `class`.

If your module only needs a single instance, you can skip the extra layers of complexity: `export` its public methods directly.

The Rabbit Hole Deepens

As promised at the top of this chapter, we just glanced over a wide surface area of the main parts of the JS language. Your head may still be spinning, but that's entirely natural after such a firehose of information!

Even with just this "brief" survey of JS, we covered or hinted at a ton of details you should carefully consider and ensure you are comfortable with. I'm serious when I suggest: re-read this chapter, maybe several times.

In the next chapter, we're going to dig much deeper into some important aspects of how JS works at its core. But before you follow that rabbit hole deeper, make sure you've taken adequate time to fully digest what we've just covered here.

You Don't Know JS Yet: Get Started - 2nd Edition

Chapter 3: Digging To The Roots Of JS

If you've read Chapters 1 and 2, and taken the time to digest and percolate, you're hopefully starting to *get* JS a little more. If you skipped/skimmed them (especially Chapter 2), I recommend you consider going back to spend some more time with that material.

In Chapter 2, we surveyed syntax, patterns, and behaviors at a high level. In this chapter, our attention shifts to some of the lower-level root characteristics of JS that underpin virtually every line of code we write.

Be aware: this chapter digs much deeper than you're likely used to thinking about a programming language. My goal is to help you appreciate the core of how JS works, what makes it tick. This chapter should begin to answer some of the "Why?" questions that are may be cropping up as you explore JS. However, this material is still not an exhaustive exposition of the language; that's what the rest of the book series is for! Our goal here is still just to *get started*, and become more comfortable with, the *feel* of JS, how it ebbs and flows.

Don't run so quickly through this material that you get lost in the weeds. As I've said a dozen times already, **take your time**. Even still, you'll probably finish this chapter with remaining questions. That's OK, because there's a whole book series ahead of you keep exploring!

Iteration

Since programs are essentially built to process data (and make decisions on that data), the patterns used to step through the data have a big impact on the program's readability.

The iterator pattern has been around for decades, and suggests a "standardized" approach to consuming data from a source one *chunk* at a time. The idea is that it's more common and helpful iterate the data source – to progressively handle the collection of data by processing the first part, then the next, and so on, rather than handling the entire set all at once.

Imagine a data structure that represents a relational database `SELECT` query, which typically organizes the results as rows. If this query had only one or a couple of rows, you could handle the entire result set at once, and assign each row to a local variable, and perform whatever operations on that data that were appropriate.

But if the query has 100 or 1000 (or more!) rows, you'll need iterative processing to deal with this data (typically, a loop).

The iterator pattern defines a data structure called an "iterator" that has a reference to an underlying data source (like the query result rows), which exposes a method like `next()`. Calling `next()` returns the next piece of data (ie, a "record" or "row" from a database query).

You don't always know how many pieces of data that you will need to iterate through, so the pattern typically indicates completion by some special value or exception once you iterate through the entire set and *go past the end*.

The importance of the iterator pattern is in adhering to a *standard* way of processing data iteratively, which creates cleaner and easier to understand code, as opposed to having every data structure/source define its own custom way of handling its data.

After many years of various JS community efforts around mutually-agreed-upon iteration techniques, ES6 standardized a specific protocol for the iterator pattern directly in the language. The protocol defines a `next()` method whose return is an object called an *iterator result*; the object has `value` and `done` properties, where `done` is a boolean that is `false` until the iteration over the underlying data source is complete.

Consuming Iterators

With the ES6 iteration protocol in place, it's workable to consume a data source one value at a time, checking after each `next()` call for `done` to be `true` to stop the iteration. But this approach is rather manual, so ES6 also included several mechanisms (syntax and APIs) for standardized consumption of these iterators.

One such mechanism is the `for..of` loop:

```
// given an iterator of some data source:
var it = /* .. */;

// loop over its results one at a time
for (let val of it) {
    console.log(`Iterator value: ${ val }`);
}
// Iterator value: ..
// Iterator value: ..
// ..
```

NOTE: |
:— |
We'll omit the manual loop equivalent here, but it's definitely less readable than the `for..of` loop! |

Another mechanism that's often used for consuming iterators is the `...` operator. This operator actually has two symmetrical forms: *spread* and *rest* (or *gather*, as I prefer). The *spread* form is an iterator-consumer.

To *spread* an iterator, you have to have *something* to spread it into. There are two possibilities in JS: an array or an argument list for a function call.

An array spread:

```
// spread an iterator into an array,
// with each iterated value occupying
// an array element position.
var vals = [ ...it ];
```

A function call spread:

```
// spread an iterator into a function,
// call with each iterated value
// occupying an argument position.
doSomethingUseful( ...it );
```

In both cases, the iterator-spread form of `...` follows the iterator-consumption protocol (the same as the `for..of` loop) to retrieve all available values from an iterator and place (aka, spread) them into the receiving context (array, argument list).

Iterables

The iterator-consumption protocol is technically defined for consuming *iterables*; an iterable is a value that can be iterated over.

The protocol automatically creates an iterator instance from an iterable, and consumes *just that iterator instance* to its completion. This means a single iterable could be consumed more than once; each time, a new iterator instance would be created and used.

So where do we find iterables?

ES6 defined the basic data structure/collection types in JS as iterables. This includes strings, arrays, maps, sets, and others.

Consider:

```
// an array is an iterable
var arr = [ 10, 20, 30 ];

for (let val of arr) {
```

```
    console.log(`Array value: ${ val }`);
}
// Array value: 10
// Array value: 20
// Array value: 30
```

Since arrays are iterables, we can shallow-copy an array using iterator consumption via the `...` spread operator:

```
var arrCopy = [ ...arr ];
```

We can also iterate the characters in a string one at a time:

```
var greeting = "Hello world!";
var chars = [ ...greeting ];

chars;
// [ "H", "e", "l", "l", "o", " ",
//   "w", "o", "r", "l", "d", "!" ]
```

A `Map` data structure uses objects as keys, associating a value (of any type) with that object. Maps have a different default iteration than seen above, in that the iteration is not just over the map's values but instead its *entries* – an *entry* is a tuple (2-element array) including both a key and a value.

Consider:

```
// given two DOM elements, `btn1` and `btn2`

var buttonNames = new Map();
buttonNames.set(btn1,"Button 1");
buttonNames.set(btn2,"Button 2");

for (let [btn,btnName] of buttonNames) {
    btn.addEventListener("click",function onClick(){
        console.log(`Clicked ${ btnName }`);
    });
}
```

In the `for..of` loop over the default map iteration, we use the `[btn,btnName]` syntax (called "array destructuring") to break down each consumed tuple into the respective key/value pairs (`btn1` / `"Button 1"` and `btn2` / `"Button 2"`).

Each of the built-in iterables in JS expose a default iteration, one which likely matches your intuition. But you can also choose a more specific iteration if necessary. For example, if we want to consume only the values of the above `buttonNames` map, we can call `values()` to get a values-only iterator:

```
for (let btnName of buttonNames.values()) {
    console.log(btnName);
}
// Button 1
// Button 2
```

Or if we want the index *and* value in an array iteration, we can make an entries iterator with the `entries()` method:

```
var arr = [ 10, 20, 30 ];

for (let [idx,val] of arr.entries()) {
    console.log(`[${ idx }]: ${ val }`);
}
// [0]: 10
// [1]: 20
// [2]: 30
```

For the most part, all built-in iterables in JS have three iterator forms available: keys-only (`keys()`), values-only (`values()`), and entries (`entries()`).

NOTE: |
:— |
You may have noticed a nuanced shift that occurred in this discussion. We started by talking about consuming **iterators**, but then switched to talking about iterating over **iterables**. The iteration-consumption protocol expects an *iterable*, but the reason we can provide a direct *iterator* is, an iterator is just an iterable of itself! In other words, when JS tries to create an iterator instance **from something that's already an iterator**, it just returns the iterator. |

Beyond just using built-in iterables, you can also ensure your own data structures adhere to the iteration protocol; doing so means you opt into the ability to consume your data with `for..of` loops and the `...` operator. "Standardizing" on this protocol means code that is overall more readily recognizable and readable.

Closure

Perhaps without realizing it, almost every JS developer has made use of closure. In fact, closure is one of the most pervasive programming functionalities across a majority of languages. It might even be as important to understand as variables or loops, that's how fundamental it is.

Yet it feels kind of hidden, almost magical. And it's often talked about in either very abstract or very informal terms, which does little to help us nail down exactly what it is.

It's critical we be able to recognize where closure is used in our programs, as the presence or lack of closure is sometimes the cause of bugs (or even the source of performance impairments).

So can we define closure in a pragmatic way that tries to bring some concrete clarity to the topic?

> Closure is the ability of a function to remember and continue to access variables defined outside its scope, even when that function is executed in a different scope.

We see two definitional characteristics here. First, closure is part of the nature of a function. Objects don't get closures, functions do. Second, to observe a closure, you must execute a function in a different scope than where that function was originally defined.

Consider:

```
function greeting(msg) {
    return function who(name) {
        console.log(`${ msg }, ${ name }!`);
    };
}

var hello = greeting("Hello");
var howdy = greeting("Howdy");

hello("Kyle");
// Hello, Kyle!

hello("Sarah");
// Hello, Sarah!

howdy("Grant");
// Howdy, Grant!
```

First, the `greeting(..)` outer function is executed, creating an instance of the inner function `who(..)`; that function closes over the variable `msg`, which is the parameter from the outer scope of `greeting(..)`. When that inner function is returned, its reference is assigned to the `hello` variable in the outer scope. Then we call `greeting(..)` a second time, creating a new inner function instance, with a new closure over a new `msg`, and return that reference to be assigned to `howdy`.

When the `greeting(..)` function finishes running, normally we would expect all of its variables to be garbage collected (removed from memory). We'd expect

each `msg` to go away, but they don't. The reason is closure. Since the inner function instances are still alive (assigned to `hello` and `howdy`, respectively), their closures are still preserving the `msg` variables.

These closures are not a snapshot of the `msg` variable's value; they are a direct link and preservation of the variable itself. That means closure can actually observe (or make!) updates to these variables over time.

```
function counter(step = 1) {
    var count = 0;
    return function increaseCount(){
        count = count + step;
        return count;
    };
}

var incBy1 = counter(1);
var incBy3 = counter(3);

incBy1();       // 1
incBy1();       // 2

incBy3();       // 3
incBy3();       // 6
incBy3();       // 9
```

Each instance of the inner `increaseCount()` function is closed over both the `count` and `step` variables from its outer `counter(..)` function's scope. `step` remains the same over time, but `count` is updated on each invocation of that inner function. Since closure is over the variables and not just snapshots of the values, these updates are preserved.

Closure is most common when working with asynchronous code, such as with callbacks. Consider:

```
function getSomeData(url) {
    ajax(url,function onResponse(resp){
        console.log(`Response (from ${ url }): ${ resp }`);
    });
}

getSomeData("https://some.url/wherever");
// Response (from https://some.url/wherever): ..whatever..
```

The inner function `onResponse(..)` is closed over `url`, and thus preserves and remembers it until the Ajax call returns and executes `onResponse(..)`. Even

though `getSomeData(..)` finishes right away, the `url` parameter variable is kept alive in the closure for as long as needed.

It's not necessary that the outer scope be a function – it usually is, but not always – just that there be at least one variable in an outer scope than an inner function accesses, and thus closes over.

```
for (let [idx,btn] of buttons.entries()) {
    btn.addEventListener("click",function onClick(evt){
       console.log(`Clicked on button (${ idx })!`);
    });
}
```

Because this loop is using `let` declarations, each iteration gets new block-scoped (aka, local) `idx` and `btn` variables; the loop also creates a new inner `onClick(..)` function each time. That inner function closes over `idx`, preserving it for as long as the click handler is set on the `btn`. So when each button is clicked, its handler can print its associated index value, because the handler remembers its respective `idx` variable.

Remember: this closure is not over the value (like `1` or `3`), but over the variable `idx` itself.

Closure is one of the most prevalent and important programming patterns in any language. But that's especially true of JS; it's hard to imagine doing anything useful without leveraging closure in one way or another.

If you're still feeling unclear or shaky about closure, the majority of "Scope & Closures" (Book 2 of this series) is focused on the topic.

`this` Keyword

One of JS's most powerful mechanisms is also one of its most misunderstood: the `this` keyword. One common misconception is that a function's `this` refers to the function itself. Because of how `this` works in other languages, another misconception is that `this` points the instance that a method belongs to. Both are incorrect.

As discussed previously, when a function is defined, it is *attached* to its enclosing scope via closure. Scope is the set of rules that controls how references to identifiers (variables) are determined.

But functions also have another characteristic besides their scope that influences what they can access. This characteristic is best described as an *execution context*, and it's exposed to the function via its `this` keyword.

Scope is static and contains a fixed set of variables available at the moment and location you define a function, but a function's execution *context* is dynamic,

entirely dependent on **how it is called** (regardless of where it is defined or even called from).

It's important to realize: `this` is not a fixed characteristic of a function based on the function's definition, but rather a dynamic characteristic that's determined each time the function is called.

One way to think about the *execution context* is that it's a tangible object whose properties are made available to a function while it executes. Compare that to scope, which can also be thought of as an *object*; except, the *scope object* is hidden inside the JS engine, it's always the same for that function, and its *properties* take the form of identifier variables available inside the function.

Consider:

```
function classroom(teacher) {
    return function study() {
        console.log(
            `${ teacher } wants you to study ${ this.topic }`
        );
    };
}

var assignment = classroom("Kyle");
```

The outer `classroom(..)` function makes no reference to a `this` keyword, so it's just like any other function we've seen so far. But the inner `study()` function does reference `this`, which makes it a `this`-aware function. In other words, it's a function that is dependent on its *execution context*.

NOTE: |
:— |
`study()` is also closed over the `teacher` variable from its outer scope. |

The inner `study()` function is returned from `classroom("Kyle")` and assigned to a variable called `assignment`. So how can `assignment()` (aka `study()`) be called?

```
assignment();
// Kyle wants you to study undefined  -- Oops :(
```

In this snippet, we call `assignment()` as a plain, normal function, without providing it any *execution context*.

Since this program is not in strict mode (See Chapter 1, "Strictly Speaking"), context-aware functions that are called **without any context specified** default the context to the global object (`window` in the browser). As there is no

global variable named `topic` (and thus no such property on the global object), `this.topic` resolves to `undefined`.

Now consider:

```
var homework = {
    topic: "JS",
    assignment: assignment
};

homework.assignment();
// Kyle wants you to study JS
```

A copy of the `assignment` function reference is set as a property on the `homework` object, and then it's called as `homework.assignment()`. That means the `this` for that function call will be the `homework` object. Hence, `this.topic` resolves to `"JS"`.

Lastly:

```
var otherHomework = {
    topic: "Math"
};

assignment.call(otherHomework);
// Kyle wants you to study Math
```

A third way to invoke a function is with the `call(..)` method, which takes an object (`otherHomework` here) to use for setting the `this` reference for the function call. `this.topic` thus resolves to `"Math"`.

The same context-aware function invoked three different ways, gives different answers each time for what object `this` will reference.

The benefit of `this`-aware functions – and their dynamic context – is the ability to more flexibly re-use a single function with data from different objects. A function that closes over a scope can never reference a different scope or set of variables. But a function that has dynamic `this` context awareness can be quite helpful for certain tasks.

Prototypes

Where `this` is a characteristic of function execution, a prototype is a characteristic of an object, and specifically resolution of a property access.

Think about a prototype as a linkage between two objects; the linkage is hidden behind the scenes, though there are ways to expose and observe it. This prototype

linkage occurs when an object is created; it's linked to another object that already exists.

A series of objects linked together via prototypes is called the "prototype chain".

The purpose of this prototype linkage (ie, from an object B to another object A) is so that accesses against B for properties/methods that B does not have, are *delegated* to A to handle. Delegation of property/method access allows two (or more!) objects to cooperate with each other to perform a task.

Consider defining an object as a normal literal:

```
var homework = {
    topic: "JS"
};
```

The `homework` object only has a single property on it: `topic`. However, its default prototype linkage connects to the `Object.prototype` object, which has common built-in methods on it like `toString()` and `valueOf()`, among others.

We can observe this prototype linkage *delegation* from `homework` to `Object.prototype`:

```
homework.toString();    // [object Object]
```

`homework.toString()` works even though `homework` doesn't have a `toString()` method defined; the delegation invokes `Object.prototype.toString()` instead.

Object Linkage

To define an object prototype linkage, you can create the object using the `Object.create(..)` utility:

```
var homework = {
    topic: "JS"
};

var otherHomework = Object.create(homework);

otherHomework.topic;
// "JS"
```

The first argument to `Object.create(..)` specifies an object to link the newly created object to, and then returns the newly created (and linked!) object.

NOTE: |
:— |

`Object.create(null)` creates an object that is not prototype linked anywhere, so it's purely just a standalone object; in some circumstances, that may be preferable. |

Figure 4 shows how the three objects (`otherHomework`, `homework`, and `Object.prototype`) are linked in a prototype chain:

Fig. 4: Objects in a prototype chain

Delegation through the prototype chain only applies for accesses to lookup the value in a property. If you assign to a property of an object, that will apply directly to the object regardless of where that object is prototype linked to.

Consider:

```
homework.topic;
// "JS"

otherHomework.topic;
// "JS"

otherHomework.topic = "Math";
otherHomework.topic;
// "Math"

homework.topic;
// "JS" -- not "Math"
```

The assignment to `topic` creates a property of that name directly on `otherHomework`; there's no effect on the `topic` property on `homework`. The next statement then accesses `otherHomework.topic`, and we see the non-delegated answer from that new property: `"Math"`.

Figure 5 shows the objects/properties after the assignment that creates the `otherHomework.topic` property:

Fig. 5: Shadowed property 'topic'

The `topic` on `otherHomework` is "shadowing" the property of the same name on the `homework` object in the chain.

NOTE: |
:— |
Another frankly more convoluted but perhaps still more common way of creating an object with a prototype linkage is using the "prototypal class" pattern, from before `class` (see Chapter 2, "Classes") was added in ES6. We'll cover this topic in more detail in Appendix A, "Prototypal 'Classes' ". |

`this` Revisited

We covered the `this` keyword earlier, but its true importance shines when considering how it powers prototype-delegated function calls. Indeed, one of the main reasons `this` supports dynamic context based on how the function is called is so that method calls on objects which delegate through the prototype chain still maintain the expected `this`.

Consider:

```
var homework = {
    study() {
        console.log(`Please study ${ this.topic }`);
    }
};

var jsHomework = Object.create(homework);
jsHomework.topic = "JS";
jsHomework.study();
// Please study JS

var mathHomework = Object.create(homework);
mathHomework.topic = "Math";
mathHomework.study();
// Please study Math
```

The two objects `jsHomework` and `mathHomework` each prototype link to the single `homework` object, which has the `study()` function. `jsHomework` and `mathHomework` are each given their own `topic` property (see Figure 6).

Fig. 6: Two objects linked to a common parent

`jsHomework.study()` delegates to `homework.study()`, but its `this` (in `this.topic`) for that execution resolves to `jsHomework` because of how the function is called, so `this.topic` is `"JS"`. Similarly for `mathHomework.study()` delegating to `homework.study()` but still resolving `this` to `mathHomework`, and thus `this.topic` as `"Math"`.

The above code snippet would be far less useful if `this` was resolved to `homework`. Yet, in many other languages, it would seem `this` would be `homework` because the `study()` method is indeed defined on `homework`.

Unlike many other languages, JS's `this` being dynamic is a critical component of allowing prototype delegation, and indeed `class`, to work as expected!

Asking Why

The intended take-away from this chapter is that there's a lot more to JS under the hood than is obvious from glancing at the surface.

As you are *getting started* learning and knowing JS more closely, one of the most important skills you can practice and bolster is curiosity, and the art of asking "why?" when you encounter something in the language.

Even though this chapter has gone quite deep on some of the topics, many details have still been entirely skimmed over. There's much more to learn here, and the path to that starts with you asking the *right* questions of your code. Asking the right questions is a critical skill of becoming a better developer.

In the final chapter of this book, we're going to briefly look at how JS is divided, as covered across the rest of the *You Don't Know JS Yet* book series. Also, don't skip Appendix B of this book, which has some practice code to review some of the main topics covered in this book.

You Don't Know JS Yet: Get Started - 2nd Edition

Chapter 4: The Bigger Picture

This book surveys what you need to be aware of as you *get started* with JS. The goal is to fill in gaps that readers newer to JS might have tripped over in their early encounters with the language. I also hope that we've hinted at enough deeper detail throughout to pique your curiosity to want to dig more into the language.

The rest of the books in this series are where we will unpack all of the rest of the language, in far greater detail than we could have done in a few brief chapters here.

Remember to take your time, though. Rather than rushing onto the next book in an attempt to churn through all the books expediently, spend some time going back over the material in this book. Spend some more time looking through code in your current projects, and comparing what you see to what's been discussed so far.

When you're ready, this final chapter divides the organization of the JS language into three main pillars, then offers a brief roadmap of what to expect from the rest of the book series, and how I suggest you proceed. Also, don't skip the appendices, especially Appendix B, "Practice, Practice, Practice!".

Pillar 1: Scope and Closure

The organization of variables into units of scope (functions, blocks) is one of the most foundational characteristics of any language; perhaps no other characteristic has a greater impact on how programs behave.

Scopes are like buckets, and variables are like marbles you put into those buckets. The scope model of a language is like the rules that help you determine which color marbles go in which matching-color buckets.

Scopes nest inside each other, and for any given expression or statement, only variables at that level of scope nesting, or in higher/outer scopes, are accessible; variables from lower/inner scopes are hidden and inaccessible.

This is how scopes behave in most languages, which is called lexical scope. The scope unit boundaries, and how variables are organized in them, is determined at the time the program is parsed (compiled). In other words, it's an author-time decision: where you locate a function/scope in the program determines what the scope structure of that part of the program will be.

JS is lexically scoped, though many claim it isn't, because of two particular characteristics of its model that are not present in other lexically scoped languages.

The first is commonly called *hoisting*: when all variables declared anywhere in a scope are treated as if they're declared at the beginning of the scope. The other is that `var` declared variables are function scoped, even if they appear inside a block.

Neither hoisting nor function-scoped `var` are sufficient to back the claim that JS is not lexically scoped. `let` / `const` declarations have a peculiar error behavior called the "Temporal Dead Zone" (TDZ) which results in observable but unusable variables. Though TDZ can be strange to encounter, it's *also* not an invalidation of lexical scoping. All of these are just unique parts of the language that should be learned and understood by all JS developers.

Closure is a natural result of lexical scope when the language has functions as first-class values, as JS does. When a function makes reference to variables from an outer scope, and that function is passed around as a value and executed in other scopes, it maintains access to its original scope variables; this is closure.

Across all of programming, but especially in JS, closure drives many of the most important programming patterns, including modules. As I see it, modules are as *with the grain* as you can get, when it comes to code organization in JS.

To dig further into scope, closures, and how modules work, read Book 2, *Scope & Closures*.

Pillar 2: Prototypes

The second pillar of the language is the prototypes system. We covered this topic in-depth in Chapter 3 ("Prototypes"), but I just want to make a few more comments about its importance.

JS is one of very few languages where you have the option to create objects directly and explicitly, without first defining their structure in a class.

For many years, people implemented the class design pattern on top of prototypes – so called, "prototypal inheritance" (see Appendix A) – and then with the advent of ES6's `class` keyword, the language doubled-down on its inclination towards OO/class style programming.

But I think that focus has obscured the beauty and power of the prototype system: the ability for two objects to simply connect with each other and cooperate dynamically (during function/method execution) through sharing a `this` context.

Classes are just one pattern you can build on top of such power. But another approach, in a very different direction, is to simply embrace objects as objects, forget classes altogether, and let objects cooperate through the prototype chain.

This is called *behavior delegation*. I think delegation is more powerful than class inheritance, as a means for organizing behavior and data in our programs.

But class inheritance gets almost all the attention. And the rest goes to functional programming (FP), as the sort of "anti-class" way of designing programs. This saddens me, because it snuffs out any chance for exploration of delegation as a viable alternative.

I encourage you to spend plenty of time deep in Book 3, *Objects & Classes*, to see how object delegation holds far more potential than we've perhaps realized. This isn't an anti-`class` message, but it is intentionally a "classes aren't the only way to use objects" message that I want more JS developers to consider.

Object delegation is, I would argue, far more *with the grain* of JS, than classes (more on *grains* in a bit).

Pillar 3: Types and Coercion

The third pillar of JS is by far the most overlooked part of JS's nature.

The vast majority of developers have strong misconceptions about how *types* work in programming languages, and especially how they work in JS. A tidal wave of interest in the broader JS community has begun to shift to "static typing" approaches, using type-aware tooling like TypeScript or Flow.

I agree that JS developers should learn more about types, and should learn more about how JS manages type conversions. I also agree that type-aware tooling can help developers, assuming they have gained and used this knowledge in the first place!

But I don't agree at all that the inevitable conclusion of this is to decide JS's type mechanism is bad and that we need to cover up JS's types with solutions outside the language. We don't have to follow the "static typing" way to be smart and solid with types in our programs. There are other options, if you're just willing to go *against the grain* of the crowd, and *with the grain* of JS (again, more on that below).

Arguably, this pillar is more important than the other two, in the sense that no JS program will do anything useful if it doesn't properly leverage JS's value types, as well as the conversion (coercion) of values between types.

Even if you love TypeScript / Flow, you are not going to get the most out of those tools or coding approaches if you aren't deeply familiar with how the language itself manages value types.

To learn more about JS types and coercion, check out Book 4, *Types & Grammar*. But please don't skip over this topic just because you've always heard that we should use `===` and forget about the rest.

Without learning this pillar, your foundation in JS is shaky and incomplete at best.

With The Grain

I have some advice to share on continuing your learning journey with JS, and your path through the rest of this book series: be aware of the *grain* – recall various references to *grain* earlier in this chapter.

First, consider the *grain* (as in, wood) of how most people approach and use JS. You've probably already noticed that these books cut against that *grain* in many respects. In YDKJSY, I respect you the reader enough to explain all the parts of JS, not only some select popular parts. I believe you're both capable and deserving of that knowledge.

But that is not what you'll find from a lot of other material out there. It also means that the more you follow and adhere to the guidance from these books – that you think carefully and analyze for yourself what's best in your code – the more you will stand out. That can be a good and bad thing. If you ever want to break out from the crowd, you're going to have to break from how the crowd does it!

But I've also had many people tell me that they quoted some topic/explanation from these books during a job interview, and the interviewer told the candidate they were wrong; indeed, people have reportedly lost out on job offers as a result.

As much as possible, I endeavor in these books to provide completely accurate information about JS, informed generally from the specification itself. But I also dose out quite a bit of my opinions on how you can interpret and use JS to the best benefit in your programs. I don't present opinion as fact, or vice versa. You'll always know which is which in these books.

Facts about JS are not really up for debate. Either the specification says something, or it doesn't. If you don't like what the specification says, or my relaying of it, take that up with TC39! If you're in an interview and they claim you're wrong on the facts, ask them right then and there if you can look it up in the specification. If the interviewer won't re-consider, then you shouldn't want to work there anyway.

But if you choose to align with my opinions, you have to be prepared to back up those choices with *why* you feel that way. Don't just parrot what I say. Own your opinions. Defend them. And if someone you were hoping to work with disagrees, walk away with your head still held high. It's a big JS, and there's plenty of room for lots of different ways.

In other words, don't be afraid to go against the *grain*, as I have done with these books and all my teachings. Nobody can tell you how you will best make use of JS; that's for you to decide. I'm merely trying to empower you in coming to your own conclusions, no matter what they are.

On the other hand, there's a *grain* you really should pay attention to and follow: the *grain* of how JS works, at the language level. There are things that work

well and naturally in JS, given the right practice and approach, and there are things you really shouldn't try to do in the language.

Can you make your JS program look like a Java, C#, or Perl program? What about Python or Ruby, or even PHP? To varying degrees, sure you can. But should you?

No, I don't think you should. I think you should learn and embrace the JS way, and make your JS programs as JS'y as is practical. Some will think that means sloppy and informal programming, but I don't mean that at all. I just mean that JS has a lot of patterns and idioms that are recognizably "JS", and going with that *grain* is the general path to best success.

Finally, maybe the most important *grain* to recognize is how the existing program(s) you're working on, and developers you're working with, do stuff. Don't read these books and then try to change *all that grain* in your existing projects over night. That approach will always fail.

You'll have to shift these things little by little, over time. Work on building consensus with your fellow developers on why it's important to re-visit and re-consider an approach. But do so with just one small topic at a time, and let before-and-after code comparisons do most of the talking. Bring everyone on the team together to discuss, and push for decisions that are based on analysis and evidence from the code rather than the inertia of, "our senior devs have always done it this way".

That's the most important advice I can impart to help you learn JS. Always keep looking for better ways to use what JS gives us to author more readable code. Everyone who works on your code, including your future self, will thank you!

In Order

So now you've got a broader perspective on what's left to explore in JS, and the right attitude to approach the rest of your journey.

But one of the most common practical questions I get at this point is, "What order should I read the books?" There is a straightforward answer... but it also depends.

My suggestion for most readers is to proceed through this series like in this order:

1. Get started with a solid foundation of JS from *Get Started* (Book 1) – good news, you've already almost finished this book!
2. In *Scope & Closures* (Book 2), dig into the first pillar of JS: lexical scope, how that supports closure, and how the module pattern organizes code.

3. In *Objects & Classes* (Book 3), focus on the second pillar of JS: how JS's `this` works, how object prototypes support delegation, and how prototypes enable the `class` mechanism for OO-style code organization.

4. In *Types & Grammar* (Book 4), tackle the third and final pillar of JS: types and type coercion, as well as how JS's syntax and grammar define how we write our code.

5. With the **three pillars** solidly in place, *Sync & Async* (Book 5) then explores how we use flow control to model state change in our programs, both synchronously (right away) and asynchronously (over time).

6. The series concludes with *ES.Next & Beyond* (Book 6), a forward look at the near- and mid-term future of JS, including a variety of features likely coming to your JS programs before too long.

That's the intended order to read this book series.

However, books 2, 3, and 4 can generally be read in any order, depending on which topic you feel most curious about and comfortable exploring first. But I don't recommend you skip any of these three books – not even *Types & Grammar*, as some of you will be tempted to do! – even if you think you already have that topic down.

Book 5 (*Sync & Async*) is crucial for deeply understanding JS, but if you start digging in and find it's too intimidating, this book can be deferred until you're more experienced with the language. The more JS you've written (and struggled with!), the more you'll come to appreciate this book. So don't be afraid to come back to it at a later time.

The final book in the series, *ES.Next & Beyond*, in some respects stands alone. It can be read at the end, as I suggest, or right after *Getting Started* if you're looking for a shortcut to broaden your radar of what JS is all about. This book will also be more likely to receive updates in the future, so you'll probably want to re-visit it occasionally.

However you choose to proceed with YDKJSY, check out the appendices of this book first, especially practicing the snippets in Appendix B, "Practice, Practice, Practice!" Did I mention you should go practice!? There's no better way to learn code than to write it.