You Don't Know JS Yet: Objects & Classes - 2nd Edition

Foreword

Kyle Simpson has an unmatched brilliance in the art of explanation.

In April of 2015 I was honored to author the foreword for Kyle's book You Don't Know JS: ES6 & Beyond, an exciting and detailed deep-dive into new JavaScript language features that hadn't even yet been ratified by Ecma (that wouldn't happen until June 2015), but had already made their way out into the world. At the time, there was gap in meaningful documentation and educational resources, and I declared that no better person had stepped up to fill it. I stand by everything I wrote in 2015, and I'm here in 2022 to double down for You Don't Know JS Yet: Objects & Classes.

We are all better off for the time, effort and resources that Kyle pours into forming a better understanding of the JavaScript programming language for himself, and for the betterment of his peers: learning, honing and distilling complex semantics into easily digestable concepts that can be built upon in infinite, myriad ways. And that's exactly what we witness here: Kyle becomes an expert in programming subject matter by consuming it at every level. By probing the semantics of new language features, luring them out from the shadows, releasing them from arcane origins and freeing them for our consumption and growth.

Even as a successful professional software engineer, I keep Kyle's works close at hand. At times these tomes are helpful in explaining deeply complex concepts to teammates and peers, while other times they serve as refreshers for myself, because there's always some interesting take-away or new perspective to gain.

Rick Waldron (@rwaldron) Lead Software Engineer Lightning Web Security Salesforce

You Don't Know JS Yet - 2nd Edition

Preface

Welcome to the 2nd edition of the widely acclaimed *You Don't Know JS* (**YDKJS**) book series: *You Don't Know JS* ***Yet*** (**YDKJSY**).

If you've read any of the 1st edition books, you can expect a refreshed approach in these new ones, with plenty of updated coverage of what's changed in JS over the last five years. But what I hope and believe you'll still *get* is the same commitment to respecting JS and digging into what really makes it tick.

If this is your first time reading these books, I'm glad you're here. Prepare for a deep and extensive journey into all the corners of JavaScript.

If you are new to programming or JS, be aware that these books are not intended as a gentle "intro to JavaScript." This material is, at times, complex and challenging, and goes much deeper than is typical for a first-time learner. You're welcome here no matter what your background is, but these books are written assuming you're already comfortable with JS and have at least 6–9 months experience with it.

The Parts

These books approach JavaScript intentionally opposite of how *The Good Parts* treats the language. No, that doesn't mean we're looking at *the bad parts*, but rather, exploring **all the parts**.

You may have been told, or felt yourself, that JS is a deeply flawed language that was poorly designed and inconsistently implemented. Many have asserted that it's the worst most popular language in the world; that nobody writes JS because they want to, only because they have to given its place at the center of the web. That's a ridiculous, unhealthy, and wholly condescending claim.

Millions of developers write JavaScript every day, and many of them appreciate and respect the language.

Like any great language, it has its brilliant parts as well as its scars. Even the creator of JavaScript himself, Brendan Eich, laments some of those parts as mistakes. But he's wrong: they weren't mistakes at all. JS is what it is today—the world's most ubiquitous and thus most influential programming language—precisely because of *all those parts*.

Don't buy the lie that you should only learn and use a small collection of *good parts* while avoiding all the bad stuff. Don't buy the "X is the new Y" snake oil, that some new feature of the language instantly relegates all usage of a previous feature as obsolete and ignorant. Don't listen when someone says your code isn't "modern" because it isn't yet using a stage-0 feature that was only proposed a few weeks ago!

Every part of JS is useful. Some parts are more useful than others. Some parts require you to be more careful and intentional.

I find it absurd to try to be a truly effective JavaScript developer while only using a small sliver of what the language has to offer. Can you imagine a construction worker with a toolbox full of tools, who only uses their hammer and scoffs at the screwdriver or tape measure as inferior? That's just silly.

My unreserved claim is that you should go about learning all parts of JavaScript, and where appropriate, use them! And if I may be so bold as to suggest: it's time to discard any JS books that tell you otherwise.

The Title?

So what's the title of the series all about?

I'm not trying to insult you with criticism about your current lack of knowledge or understanding of JavaScript. I'm not suggesting you can't or won't be able to learn JavaScript. I'm not boasting about secret advanced insider wisdom that I and only a select few possess.

Seriously, all those were real reactions to the original series title before folks even read the books. And they're baseless.

The primary point of the title "You Don't Know JS Yet" is to point out that most JS developers don't take the time to really understand how the code that they write works. They know *that* it works—that it produces a desired outcome. But they either don't understand exactly *how*, or worse, they have an inaccurate mental model for the *how* that falters on closer scrutiny.

I'm presenting a gentle but earnest challenge to you the reader, to set aside the assumptions you have about JS, and approach it with fresh eyes and an invigorated curiosity that leads you to ask *why* for every line of code you write. Why does it do what it does? Why is one way better or more appropriate than the other half-dozen ways you could have accomplished it? Why do all the "popular kids" say to do X with your code, but it turns out that Y might be a better choice?

I added "Yet" to the title, not only because it's the second edition, but because ultimately I want these books to challenge you in a hopeful rather than discouraging way.

But let me be clear: I don't think it's possible to ever fully *know* JS. That's not an achievement to be obtained, but a goal to strive after. You don't finish knowing everything about JS, you just keep learning more and more as you spend more time with the language. And the deeper you go, the more you revisit what you *knew* before, and you re-learn it from that more experienced perspective.

I encourage you to adopt a mindset around JavaScript, and indeed all of software

development, that you will never fully have mastered it, but that you can and should keep working to get closer to that end, a journey that will stretch for the entirety of your software development career, and beyond.

You can always know JS better than you currently do. That's what I hope these YDKJSY books represent.

The Mission

The case doesn't really need to be made for why developers should take JS seriously—I think it's already more than proven worthy of first-class status among the world's programming languages.

But a different, more important case still needs to be made, and these books rise to that challenge.

I've taught more than 5,000 developers from teams and companies all over the world, in more than 25 countries on six continents. And what I've seen is that far too often, what *counts* is generally just the result of the program, not how the program is written or how/why it works.

My experience not only as a developer but in teaching many other developers tells me: you will always be more effective in your development work if you more completely understand how your code works than you are solely *just* getting it to produce a desired outcome.

In other words, *good enough to work* is not, and should not be, *good enough*.

All developers regularly struggle with some piece of code not working correctly, and they can't figure out why. But far too often, JS developers will blame this on the language rather than admitting it's their own understanding that is falling short. These books serve as both the question and answer: why did it do *this*, and here's how to get it to do *that* instead.

My mission with YDKJSY is to empower every single JS developer to fully own the code they write, to understand it and to write with intention and clarity.

The Path

Some of you have started reading this book with the goal of completing all six books, back to back.

I would like to caution you to consider changing that plan.

It is not my intention that YDKJSY be read straight through. The material in these books is dense, because JavaScript is powerful, sophisticated, and in parts rather complex. Nobody can really hope to *download* all this information to their brains in a single pass and retain any significant amount of it. That's unreasonable, and it's foolish to try.

My suggestion is you take your time going through YDKJSY. Take one chapter, read it completely through start to finish, and then go back and re-read it section by section. Stop in between each section, and practice the code or ideas from that section. For larger concepts, it probably is a good idea to expect to spend several days digesting, re-reading, practicing, then digesting some more.

You could spend a week or two on each chapter, and a month or two on each book, and a year or more on the whole series, and you would still not be squeezing every ounce of YDKJSY out.

Don't binge these books; be patient and spread out your reading. Interleave reading with lots of practice on real code in your job or on projects you participate in. Wrestle with the opinions I've presented along the way, debate with others, and most of all, disagree with me! Run a study group or book club. Teach mini-workshops at your office. Write blog posts on what you've learned. Speak about these topics at local JS meetups.

It's never my goal to convince you to agree with my opinion, but to encourage you to own and be able to defend your opinions. You can't get *there* with an expedient read-through of these books. That's something that takes a long while to emerge, little by little, as you study and ponder and re-visit.

These books are meant to be a field-guide on your wanderings through JavaScript, from wherever you currently are with the language, to a place of deeper understanding. And the deeper you understand JS, the more questions you will ask and the more you will have to explore! That's what I find so exciting!

I'm so glad you're embarking on this journey, and I am so honored you would consider and consult these books along the way. It's time to start *getting to know JS*.

You Don't Know JS Yet: Objects & Classes - 2nd Edition

Chapter 1: Object Foundations

NOTE:
Work in progress

Everything in JS is an object.

This is one of the most pervasive, but most incorrect, "facts" that perpetually circulates about JS. Let the myth busting commence.

JS definitely has objects, but that doesn't mean that all values are objects. Nevertheless, objects are arguably the most important (and varied!) value type in the language, so mastering them is critical to your JS journey.

The object mechanism is certainly the most flexible and powerful container type – something you put other values into; every JS program you write will use them in one way or another. But that's not why objects deserve top billing for this book. Objects are the foundation for the second of JS's three pillars: the prototype.

Why are prototypes (along with the `this` keyword, covered later in the book) so core to JS as to be one of its three pillars? Among other things, prototypes are how JS's object system can express the class design pattern, one of the most widely relied on design patterns in all of programming.

So our journey here will start with objects, build up a compelete understanding of prototypes, de-mystify the `this` keyword, and explore the `class` system.

About This Book

Welcome to book 3 in the *You Don't Know JS Yet* series! If you already finished *Get Started* (the first book) and *Scope & Closures* (the second book), you're in the right spot! If not, before you proceed I encourage you to read those two as foundations before diving into this book.

The first edition of this book is titled, "this & Object Prototypes". In that book, our focus started with the `this` keyword, as it's arguably one of the most confused topics in all of JS. The book then spent the majority of its time focused on expositing the prototype system and advocating for embrace of the lesser-known "delegation" pattern instead of class designs. At the time of that book's writing (2014), ES6 would still be almost 2 years to its completion, so I felt the early sketches of the `class` keyword only merited a brief addendum of coverage.

It's quite an understatement to say a lot has changed in the JS landscape in the almost 8 years since that book. ES6 is old news now; at the time of *this* book's writing, JS has seen 7 yearly updates **after ES6** (ES2016 through ES2022).

Now, we still need to talk about how `this` works, and how that relates to methods invoked against various objects. And `class` actually operates (mostly!) via the prototype chain deep under the covers. But JS developers in 2022 are almost never writing code to explicitly wire up prototypal inheritance anymore. And as much as I personally wish differently, class design patterns – not "behavior delegation" – are how the majority of data and behavior organization (data structures) in JS are expressed.

This book reflects JS's current reality: thus the new sub-title, new organization and focus of topics, and complete re-write of the previous edition's text.

Objects As Containers

One common way of gathering up multiple values in a single container is with an object. Objects are collections of key/value pairs. There are also sub-types of object in JS with specialized behaviors, such as arrays (numerically indexed) and even functions (callable); more on these sub-types later.

NOTE:

Keys are often referred to as "property names", with the pairing of a property name and a value often called a "property". This book will use those terms distinctly in that manner.

Regular JS objects are typically declared with literal syntax, like this:

```
myObj = {
    // ..
};
```

Note: There's an alternate way to create an object (using `myObj = new Object()`), but this is not common or preferred, and is almost never the appropriate way to go about it. Stick with object literal syntax.

It's easy to get confused what pairs of `{ .. }` mean, since JS overloads the curly brackets to mean any of the following, depending on the context used:

- delimit values, like object literals
- define object destructuring patterns (more on this later)
- delimit interpolated string expressions, like `` `some ${ getNumber() } thing` ``
- define blocks, like on `if` and `for` loops
- define function bodies

Though it can sometimes be challenging as you read code, look for whether a `{ .. }` curly brace pair is used in the program where a value/expression is valid to appear; if so, it's an object literal, otherwise it's one of the other overloaded uses.

Defining Properties

Inside the object literal curly braces, you define properties (name and value) with `propertyName: propertyValue` pairs, like this:

```
myObj = {
    favoriteNumber: 42,
    isDeveloper: true,
    firstName: "Kyle"
};
```

The values you assign to the properties can be literals, as shown, or can be computed by expression:

```
function twenty() { return 20; }

myObj = {
    favoriteNumber: (twenty() + 1) * 2,
};
```

The expression `(twenty() + 1) * 2` is evaluated immediately, with the result (`42`) assigned as the property value.

Developers sometimes wonder if there's a way to define an expression for a property value where the expression is "lazy", meaning it's not computed at the time of assignment, but defined later. JS does not have lazy expressions, so the only way to do so is for the expression to be wrapped in a function:

```
function twenty() { return 20; }
function myNumber() { return (twenty() + 1) * 2; }

myObj = {
    favoriteNumber: myNumber    // notice, NOT `myNumber()` as a function call
};
```

In this case, `favoriteNumber` is not holding a numeric value, but rather a function reference. To compute the result, that function reference must be explicitly executed.

Looks Like JSON?

You may notice that this object-literal syntax we've seen thus far resembles a related syntax, "JSON" (JavaScript Object Notation):

```
{
    "favoriteNumber": 42,
    "isDeveloper": true,
    "firstName": "Kyle"
}
```

The biggest differences between JS's object literals and JSON are, for objects defined as JSON:

1. property names must be quoted with `"` double-quote characters
2. property values must be literals (either primitives, objects, or arrays), not arbitrary JS expressions

In JS programs, an object literal does not require quoted property names – you *can* quote them (`'` or `"` allowed), but it's usually optional. There are however characters that are valid in a property name, but which cannot be included without surrounding quotes; for example, leading numbers or whitespace:

```
myObj = {
    favoriteNumber: 42,
    isDeveloper: true,
    firstName: "Kyle",
    "2 nicknames": [ "getify", "ydkjs" ]
};
```

One other minor difference is, JSON syntax – that is, text that will be *parsed* as JSON, such as from a `.json` file – is stricter than general JS. For example, JS allows comments (`// ..` and `/* .. */`), and trailing `,` commas in object and array expressions; JSON does not allow any of these. Thankfully, JSON does still allow arbitrary whitespace.

Property Names

Property names in object literals are almost always treated/coeced as string values. One exception to this is for integer (or "integer looking") property "names":

```
anotherObj = {
    42:       "<-- this property name will be treated as an integer",
    "41":     "<-- ...and so will this one",

    true:     "<-- this property name will be treated as a string",
    [myObj]:  "<-- ...and so will this one"
};
```

The `42` property name will be treated as an integer property name (aka, index); the `"41"` string value will also be treated as such since it *looks like* an integer. By contrast, the `true` value will become the string property name `"true"`, and

the `myObj` identifier reference, *computed* via the surrounding `[ .. ]`, will coerce the object's value to a string (generally the default `"[object Object]"`).

WARNING:

If you need to actually use an object as a key/property name, never rely on this computed string coercion; its behavior is surprising and almost certainly not what's expected, so program bugs are likely to occur. Instead, use a more specialized data structure, called a `Map` (added in ES6), where objects used as property "names" are left as-is instead of being coerced to a string value.

As with `[myObj]` above, you can *compute* any **property name** (distinct from computing the property value) at the time of object literal definition:

```
anotherObj = {
    ["x" + (21 * 2)]: true
};
```

The expression `"x" + (21 * 2)`, which must appear inside of `[ .. ]` brackets, is computed immediately, and the result (`"x42"`) is used as the property name.

Symbols As Property Names

ES6 added a new primitive value type of `Symbol`, which is often used as a special property name for storing and retrieving property values. They're created via the `Symbol(..)` function call (**without** the `new` keyword), which accepts an optional description string used only for friendlier debugging purposes; if specified, the description is inaccessible to the JS program and thus not used for any other purpose than debug output.

```
myPropSymbol = Symbol("optional, developer-friendly description");
```

NOTE:

Symbols are sort of like numbers or strings, except that their value is *opaque* to, and globally unique within, the JS program. In other words, you can create and use symbols, but JS doesn't let you know anything about, or do anything with, the underlying value; that's kept as a hidden implementation detail by the JS engine.

Computed property names, as previously described, are how to define a symbol property name on an object literal:

```
myPropSymbol = Symbol("optional, developer-friendly description");

anotherObj = {
    [myPropSymbol]: "Hello, symbol!"
};
```

The computed property name used to define the property on `anotherObj` will be the actual primitive symbol value (whatever it is), not the optional description string (`"optional, developer-friendly description"`).

Because symbols are globally unique in your program, there's **no** chance of accidental collision where one part of the program might accidentally define a property name the same as another part of the program tried defined/assigned.

Symbols are also useful to hook into special default behaviors of objects, and we'll cover that in more detail in "Extending the MOP" in the next chapter.

Concise Properties

When defining an object literal, it's common to use a property name that's the same as an existing in-scope identifier that holds the value you want to assign.

```
coolFact = "the first person convicted of speeding was going 8 mph";

anotherObj = {
    coolFact: coolFact
};
```

NOTE:

That would have been the same thing as the quoted property name definition `"coolFact": coolFact`, but JS developers rarely quote property names unless strictly necessary. Indeed, it's idiomatic to avoid the quotes unless required, so it's discouraged to include them unnecessarily.

In this situation, where the property name and value expression identifier are identical, you can omit the property-name portion of the property definition, as a so-called "concise property" definition:

```
coolFact = "the first person convicted of speeding was going 8 mph";

anotherObj = {
    coolFact   // <-- concise property short-hand
};
```

The property name is `"coolFact"` (string), and the value assigned to the property is what's in the `coolFact` variable at that moment: `"the first person convicted of speeding was going 8 mph"`.

At first, this shorthand convenience may seem confusing. But as you get more familiar with seeing this very common and popular feature being used, you'll likely favor it for typing (and reading!) less.

Concise Methods

Another similar shorthand is defining functions/methods in an object literal using a more concise form:

```
anotherObj = {
    // standard function property
    greet: function() { console.log("Hello!"); },

    // concise function/method property
    greet2() { console.log("Hello, friend!"); }
};
```

While we're on the topic of concise method properties, we can also define generator functions (another ES6 feature):

```
anotherObj = {
    // instead of:
    //   greet3: function*() { yield "Hello, everyone!"; }

    // concise generator method
    *greet3() { yield "Hello, everyone!"; }
};
```

And though it's not particularly common, concise methods/generators can even have quoted or computed names:

```
anotherObj = {
    "greet-4"() { console.log("Hello, audience!"); },

    // concise computed name
    [ "gr" + "eet 5" ]() { console.log("Hello, audience!"); },

    // concise computed generator name
    *[ "ok, greet 6".toUpperCase() ]() { yield "Hello, audience!"; }
};
```

Object Spread

Another way to define properties at object literal creation time is with a form of the `...` syntax – it's not technically an operator, but it certainly seems like one – often referred to as "object spread".

The `...` when used inside an object literal will "spread" out the contents (properties, aka key/value pairs) of another object value into the object being defined:

```
anotherObj = {
    favoriteNumber: 12,
```

```
    ...myObj,   // object spread, shallow copies `myObj`

    greeting: "Hello!"
}
```

The spreading of `myObj`'s properties is shallow, in that it only copies the top-level properties from `myObj`; any values those properties hold are simply assigned over. If any of those values are references to other objects, the references themselves are assigned (by copy), but the underlying object values are *not* duplicated – so you end up with multiple shared references to the same object(s).

You can think of object spreading like a `for` loop that runs through the properties one at a time and does an `=` style assignment from the source object (`myObj`) to the target object (`anotherObj`).

Also, consider these property definition operations to happen "in order", from top to bottom of the object literal. In the above snippet, since `myObj` has a `favoriteNumber` property, the object spread will end up overwriting the `favoriteNumber: 12` property assignment from the previous line. Moreover, if `myObj` had contained a `greeting` property that was copied over, the next line (`greeting: "Hello!"`) would override that property definition.

NOTE:

Object spread also only copies *owned* properties (those directly on the object) that are *enumerable* (allowed to be enumerated/listed). It does not duplicate the property – as in, actually mimic the property's exact characteristics – but rather do a simple assignment style copy. We'll cover more such details in the "Property Descriptors" section of the next chapter.

A common way `...` object spread is used is for performing *shallow* object duplication:

```
myObjShallowCopy = { ...myObj };
```

Keep in mind you cannot `...` spread into an existing object value; the `...` object spread syntax can only appear inside the `{ .. }` object literal, which is creating a new object value. To perform a similar shallow object copy but with APIs instead of syntax, see the "Object Entries" section later in this chapter (with coverage of `Object.entries(..)` and `Object.fromEntries(..)`).

But if you instead want to copy object properties (shallowly) into an *existing* object, see the "Assigning Properties" section later in this chapter (with coverage of `Object.assign(..)`).

Deep Object Copy

Also, since `...` doesn't do full, deep object duplication, the object spread is generally only suitable for duplicating objects that hold simple, primitive values

only, not references to other objects.

Deep object duplication is an incredibly complex and nuanced operation. Duplicating a value like `42` is obvious and straightforward, but what does it mean to copy a function (which is a special kind of object, also held by reference), or to copy an external (not entirely in JS) object reference, such as a DOM element? And what happens if an object has circular references (like where a nested descendant object holds a reference back up to an outer ancestor object)? There's a variety of opinions in the wild about how all these corner cases should be handled, and thus no single standard exists for deep object duplication.

For deep object duplication, the standard approaches have been:

1. Use a library utility that declares a specific opinion on how the duplication behaviors/nuances should be handled.
2. Use the `JSON.parse(JSON.stringify(..))` round-trip trick – this only "works" correctly if there are no circular references, and if there are no values in the object that cannot be properly serialized with JSON (such as functions).

Recently, though, a third option has landed. This is not a JS feature, but rather a companion API provided to JS by environments like the web platform. Objects can be deep copied now using `structuredClone(..)`[1].

```
myObjCopy = structuredClone(myObj);
```

The underlying algorithm behind this built-in utility supports duplicating circular references, as well as **many more** types of values than the `JSON` round-trip trick. However, this algorithm still has its limits, including no support for cloning functions or DOM elements.

Accessing Properties

Property access of an existing object is preferably done with the `.` operator:

```
myObj.favoriteNumber;     // 42
myObj.isDeveloper;        // true
```

If it's possible to access a property this way, it's strongly suggested to do so.

If the property name contains characters that cannot appear in identifiers, such as leading numbers or whitespace, `[ .. ]` brackets can be used instead of the `.`:

```
myObj["2 nicknames"];     // [ "getify", "ydkjs" ]

anotherObj[42];           // "<-- this property name will..."
anotherObj["41"];         // "<-- this property name will..."
```

[1] "Structured Clone Algorithm", HTML Specification; https://html.spec.whatwg.org/multipage/structured-data.html#structured-cloning ; Accessed July 2022

Even though numeric property "names" remain as numbers, property access via the `[ .. ]` brackets will coerce a string representation to a number (e.g., `"42"` as the `42` numeric equivalent), and then access the associated numeric property accordingly.

Similar to the object literal, the property name to access can be computed via the `[ .. ]` brackets. The expression can be a simple identifier:

```
propName = "41";
anotherObj[propName];
```

Actually, what you put between the `[ .. ]` brackets can be any arbitrary JS expression, not just identifiers or literal values like `42` or `"isDeveloper"`. JS will first evaluate the expression, and the resulting value will then be used as the property name to look up on the object:

```
function howMany(x) {
    return x + 1;
}

myObj[`${ howMany(1) } nicknames`];   // [ "getify", "ydkjs" ]
```

In this snippet, the expression is a back-tick delimited `` `template string literal` `` with an interpolated expression of the function call `howMany(1)`. The overall result of that expression is the string value `"2 nicknames"`, which is then used as the property name to access.

Object Entries

You can get a listing of the properties in an object, as an array of tuples (two-element sub-arrays) holding the property name and value:

```
myObj = {
    favoriteNumber: 42,
    isDeveloper: true,
    firstName: "Kyle"
};

Object.entries(myObj);
// [ ["favoriteNumber",42], ["isDeveloper",true], ["firstName","Kyle"] ]
```

Added in ES6, `Object.entries(..)` retrieves this list of entries – containing only owned an enumerable properties; see the "Property Descriptors" section in the next chapter – from a source object.

Such a list can be looped/iterated over, potentially assigning properties to another existing object. However, it's also possible to create a new object from a list of entries, using `Object.fromEntries(..)` (added in ES2019):

```
myObjShallowCopy = Object.fromEntries( Object.entries(myObj) );
```

```
// alternate approach to the earlier discussed:
// myObjShallowCopy = { ...myObj };
```

Destructuring

Another approach to accessing properties is through object destructuring (added in ES6). Think of destructuring as defining a "pattern" that describes what an object value is supposed to "look like" (structurally), and then asking JS to follow that "pattern" to systematically access the contents of an object value.

The end result of object destructuring is not another object, but rather one or more assignments to other targets (variables, etc) of the values from the source object.

Imagine this sort of pre-ES6 code:

```
myObj = {
    favoriteNumber: 42,
    isDeveloper: true,
    firstName: "Kyle"
};

const favoriteNumber = (
    myObj.favoriteNumber !== undefined ? myObj.favoriteNumber : 42
);
const isDev = myObj.isDeveloper;
const firstName = myObj.firstName;
const lname = (
    myObj.lastName !== undefined ? myObj.lastName : "--missing--"
);
```

Those accesses of the property values, and assignments to other identifiers, is generally called "manual destructuring". To use the declarative object destructuring syntax, it might look like this:

```
myObj = {
    favoriteNumber: 42,
    isDeveloper: true,
    firstName: "Kyle"
};

const { favoriteNumber = 12 } = myObj;
const {
    isDeveloper: isDev,
    firstName: firstName,
    lastName: lname = "--missing--"
} = myObj;
```

```
favoriteNumber;    // 42
isDev;             // true
firstName;         // "Kyle"
lname;             // "--missing--"
```

As shown, the `{ .. }` object destructuring resembles an object literal value definition, but it appears on the left-hand side of the `=` operator rather than on the right-hand side where an object value expression would appear. That makes the `{ .. }` on the left-hand side a destructuring pattern rather than another object definition.

The `{ favoriteNumber } = myObj` destructuring tells JS to find a property named `favoriteNumber` on the object, and to assign its value to an identifier of the same name. The single instance of the `favoriteNumber` identifier in the pattern is similar to "concise properties" as discussed earlier in this chapter: if the source (property name) and target (identifier) are the same, you can omit one of them and only list it once.

The `= 12` part tells JS to provide `12` as a default value for the assignment to `favoriteNumber`, if the source object either doesn't have a `favoriteNumber` property, or if the property holds an `undefined` value.

In the second destructuring pattern, the `isDeveloper: isDev` pattern is instructing JS to find a property named `isDeveloper` on the source object, and assign its value to an identifier named `isDev`. It's sort of a "renaming" of the source to the target. By contrast, `firstName: firstName` is providing the source and target for an assignment, but is redundant since they're identical; a single `firstName` would have sufficed here, and is generally more preferred.

The `lastName: lname = "--missing--"` combines both source-target renaming and a default value (if the `lastName` source property is missing or `undefined`).

The above snippet combines object destructuring with variable declarations – in this example, `const` is used, but `var` and `let` work as well – but it's not inherently a declaration mechanism. Destructuring is about access and assignment (source to target), so it can operate against existing targets rather than declaring new ones:

```
let fave;

// surrounding ( ) are required syntax here,
// when a declarator is not used
({ favoriteNumber: fave } = myObj);

fave;  // 42
```

Object destructuring syntax is generally preferred for its declarative and more readable style, over the heavily imperative pre-ES6 equivalents. But don't go

overboard with destructuring. Sometimes just doing `x = someObj.x` is perfectly fine!

Conditional Property Access

Recently (in ES2020), a feature known as "optional chaining" was added to JS, which augments property access capabilities (especially nested property access). The primary form is the two-character compound operator `?.`, like `A?.B`.

This operator will check the left-hand side reference (`A`) to see if it's null'ish (`null` or `undefined`). If so, the rest of the property access expression is short-circuited (skipped), and `undefined` is returned as the result (even if it was `null` that was actually encountered!). Otherwise, `?.` will access the property just as a normal `.` operator would.

For example:

```
myObj?.favoriteNumber
```

Here, the null'ish check is performed against the `myObj`, meaning that the `favoriteNumber` property access is only performed if the value in `myObj` is non-null'ish. Note that it doesn't verify that `myObj` is actually holding a real object, only that it's non-nullish. However, all non-nullish values can "safely" (no JS exception) be "accessed" via the `.` operator, even if there's no matching property to retrieve.

It's easy to get confused into thinking that the null'ish check is against the `favoriteNumber` property. But one way to keep it straight is to remember that the `?` is on the side where the safety check is performed, while the `.` is on the side that is only conditionally evaluated if the non-null'ish check passes.

Typically, the `?.` operator is used in nested property accesses that may be 3 or more levels deep, such as:

```
myObj?.address?.city
```

The equivalent operation with the `?.` operator would look like this:

```
(myObj != null && myObj.address != null) ? myObj.address.city : undefined
```

Again, remember that no check has been performed against the right-most property (`city`) here.

Also, the `?.` should not universally be used in place of every single `.` operator in your programs. You should endeavor to know if a `.` property access will succeed or not before making the access, whenever possible. Use `?.` only when the nature of the values being accessed is subject to conditions that cannot be predicted/controlled.

For example, in the previous snippet, the `myObj?.` usage is probably mis-guided, because it really shouldn't be the case that you start a chain of property access

against a variable that might not even hold a top-level object (aside from its contents potentially missing certain properties in certain conditions).

Instead, I would recommend usage more like this:

```
myObj.address?.city
```

And that expression should only be used in part of your program where you're sure that `myObj` is at least holding a valid object (whether or not it has an `address` property with a sub-object in it).

Another form of the "optional chaining" operator is `?.[`, which is used when the property access you want to make conditional/safe requires a `[ .. ]` bracket.

```
myObj["2 nicknames"]?.[0];    // "getify"
```

Everything asserted about how `?.` behaves goes the same for `?.[`.

WARNING:

There's a third form of this feature, named "optional call", which uses `?.(` as the operator. It's used for performing a non-null'ish check on a property before executing the function value in the property. For example, instead of `myObj.someFunc(42)`, you can do `myObj.someFunc?.(42)`. The `?.(` checks to make sure `myObj.someFunc` is non-null'ish before invoking it (with the `(42)` part). While that may sound like a useful feature, I think this is dangerous enough to warrant complete avoidance of this form/construct.My concern is that `?.(` makes it seem as if we're ensuring that the function is "callable" before calling it, when in fact we're only checking if it's non-null'ish. Unlike `?.` which can allow a "safe" `.` access against a non-null'ish value that's also not an object, the `?.(` non-null'ish check isn't similarly "safe". If the property in question has any non-null'ish, non-function value in it, like `true` or `"Hello"`, the `(42)` call part will be invoked and yet throw a JS exception. So in other words, this form is unfortunately masquerading as more "safe" than it actually is, and should thus be avoided in essentially all circumstances. If a property value can ever *not be* a function, do a more fullsome check for its function'ness before trying to invoke it. Don't pretend that `?.(` is doing that for you, or future readers/maintainers of your code (including your future self!) will likely regret it.

Accessing Properties On Non-Objects

This may sound counter-intuitive, but you can generally access properties/methods from values that aren't themselves objects:

```
fave = 42;

fave;              // 42
fave.toString();   // "42"
```

Here, `fave` holds a primitive `42` number value. So how can we do `.toString` to access a property from it, and then `()` to invoke the function held in that property?

This is a tremendously more indepth topic than we'll get into in this book; see book 4, "Types & Grammar", of this series for more. However, as a quick glimpse: if you perform a property access (`.` or `[ .. ]`) against a non-object, non-null'ish value, JS will by default (temporarily!) coerce the value into an object-wrapped representation, allowing the property access against that implicitly instantiated object.

This process is typically called "boxing", as in putting a value inside a "box" (object container).

So in the above snippet, just for the moment that `.toString` is being accessed on the `42` value, JS will box this value into a `Number` object, and then perform the property access.

Note that `null` and `undefined` can be object-ified, by calling `Object(null)` / `Object(undefined)`. However, JS does not automatically box these null'ish values, so property access against them will fail (as discussed earlier in the "Conditional Property Access" section).

NOTE:

Boxing has a counterpart: unboxing. For example, the JS engine will take an object wrapper – like a `Number` object wrapped around `42` – created with `Number(42)` or `Object(42)` – and unwrap it to retrieve the underlying primitive `42`, whenever a mathematical operation (like `*` or `-`) encounters such an object. Unboxing behavior is way out of scope for our discussion, but is covered fully in the aforementioned "Types & Grammar" title.

Assigning Properties

Whether a property is defined at the time of object literal definition, or added later, the assignment of a property value is done with the `=` operator, as any other normal assignment would be:

```
myObj.favoriteNumber = 123;
```

If the `favoriteNumber` property doesn't already exist, that statement will create a new property of that name and assign its value. But if it already exists, that statement will re-assign its value.

WARNING:

An = assignment to a property may fail (silently or throwing an exception), or it may not directly assign the value but instead invoke a *setter* function that performs some operation(s). More details on these behaviors in the next chapter.

It's also possible to assign one or more properties at once – assuming the source properties (name and value pairs) are in another object – using the `Object.assign(..)` (added in ES6) method:

```
// shallow copy all (owned and enumerable) properties
// from `myObj` into `anotherObj`
Object.assign(anotherObj,myObj);

Object.assign(
    /*target=*/anotherObj,
    /*source1=*/{
        someProp: "some value",
        anotherProp: 1001,
    },
    /*source2=*/{
        yetAnotherProp: false
    }
);
```

`Object.assign(..)` takes the first object as target, and the second (and optionally subsequent) object(s) as source(s). Copying is done in the same manner as described earlier in the "Object Spread" section.

Deleting Properties

Once a property is defined on an object, the only way to remove it is with the `delete` operator:

```
anotherObj = {
    counter: 123
};

anotherObj.counter;   // 123

delete anotherObj.counter;

anotherObj.counter;   // undefined
```

Contrary to common misconception, the JS `delete` operator does **not** directly do any deallocation/freeing up of memory, through garbage collection (GC).

The only thing it does is remove a property from an object. If the value in the property was a reference (to another object/etc), and there are no other surviving references to that value once the property is removed, that value would likely then be eligible for removal in a future sweep of the GC.

Calling `delete` on anything other than an object property is a misuse of the `delete` operator, and will either fail silently (in non-strict mode) or throw an exception (in strict mode).

Deleting a property from an object is distinct from assigning it a value like `undefined` or `null`. A property assigned `undefined`, either initially or later, is still present on the object, and might still be revealed when enumerating the contents

Determining Container Contents

You can determine an object's contents in a variety of ways. To ask an object if it has a specific property:

```
myObj = {
    favoriteNumber: 42,
    coolFact: "the first person convicted of speeding was going 8 mph",
    beardLength: undefined,
    nicknames: [ "getify", "ydkjs" ]
};

"favoriteNumber" in myObj;            // true

myObj.hasOwnProperty("coolFact");     // true
myObj.hasOwnProperty("beardLength");  // true

myObj.nicknames = undefined;
myObj.hasOwnProperty("nicknames");    // true

delete myObj.nicknames;
myObj.hasOwnProperty("nicknames");    // false
```

There *is* an important difference between how the `in` operator and the `hasOwnProperty(..)` method behave. The `in` operator will check not only the target object specified, but if not found there, it will also consult the object's `[[Prototype]]` chain (covered in the next chapter). By contrast, `hasOwnProperty(..)` only consults the target object.

If you're paying close attention, you may have noticed that `myObj` appears to have a method property called `hasOwnProperty(..)` on it, even though we didn't define such. That's because `hasOwnProperty(..)` is defined as a built-in on `Object.prototype`, which by default is "inherited by" all normal objects. There is risk inherent to accessing such an "inherited" method, though. Again,

more on prototypes in the next chapter.

Better Existence Check

ES2022 (almost official at time of writing) has already settled on a new feature, `Object.hasOwn(..)`. It does essentially the same thing as `hasOwnProperty(..)`, but it's invoked as a static helper external to the object value instead of via the object's `[[Prototype]]`, making it safer and more consistent in usage:

```
// instead of:
myObj.hasOwnProperty("favoriteNumber")

// we should now prefer:
Object.hasOwn(myObj,"favoriteNumber")
```

Even though (at time of writing) this feature is just now emerging in JS, there are polyfills that make this API available in your programs even when running in a previous JS environment that doesn't yet have the feature defined. For example, a quick stand-in polyfill sketch:

```
// simple polyfill sketch for `Object.hasOwn(..)`
if (!Object.hasOwn) {
    Object.hasOwn = function hasOwn(obj,propName) {
        return Object.prototype.hasOwnProperty.call(obj,propName);
    };
}
```

Including a polyfill patch such as that in your program means you can safely start using `Object.hasOwn(..)` for property existence checks no matter whether a JS environment has `Object.hasOwn(..)` built in yet or not.

Listing All Container Contents

We already discussed the `Object.entries(..)` API earlier, which tells us what properties an object has (as long as they're enumerable – more in the next chapter).

There's a variety of other mechanisms available, as well. `Object.keys(..)` gives us list of the enumerable property names (aka, keys) in an object – names only, no values; `Object.values(..)` instead gives us list of all values held in enumerable properties.

But what if we wanted to get *all* the keys in an object (enumerable or not)? `Object.getOwnPropertyNames(..)` seems to do what we want, in that it's like `Object.keys(..)` but also returns non-enumerable property names. However, this list **will not** include any Symbol property names, as those are treated as special locations on the object. `Object.getOwnPropertySymbols(..)` returns

all of an object's Symbol properties. So if you concatenate both of those lists together, you'd have all the direct (*owned*) contents of an object.

Yet as we've implied several times already, and will cover in full detail in the next chapter, an object can also "inherit" contents from its `[[Prototype]]` chain. These are not considered *owned* contents, so they won't show up in any of these lists.

Recall that the `in` operator will potentially traverse the entire chain looking for the existence of a property. Similarly, a `for..in` loop will traverse the chain and list any enumerable (owned or inherited) properties. But there's no built-in API that will traverse the whole chain and return a list of the combined set of both *owned* and *inherited* contents.

Temporary Containers

Using a container to hold multiple values is sometimes just a temporary transport mechanism, such as when you want to pass multiple values to a function via a single argument, or when you want a function to return multiple values:

```
function formatValues({ one, two, three }) {
    // the actual object passed in as an
    // argument is not accessible, since
    // we destructured it into three
    // separate variables

    one = one.toUpperCase();
    two = `--${two}--`;
    three = three.substring(0,5);

    // this object is only to transport
    // all three values in a single
    // return statement
    return { one, two, three };
}

// destructuring the return value from
// the function, because that returned
// object is just a temporary container
// to transport us multiple values
const { one, two, three } =

    // this object argument is a temporary
    // transport for multiple input values
    formatValues({
       one: "Kyle",
       two: "Simpson",
```

```
        three: "getify"
    });

one;      // "KYLE"
two;      // "--Simpson--"
three;    // "getif"
```

The object literal passed into `formatValues(..)` is immediately parameter destructured, so inside the function we only deal with three separate variables (`one`, `two`, and `three`). The object literal `return`ed from the function is also immediately destructured, so again we only deal with three separate variables (`one`, `two`, `three`).

This snippet illustrates the idiom/pattern that an object is sometimes just a temporary transport container rather than a meaningful value in and of itself.

Containers Are Collections Of Properties

The most common usage of objects is as containers for multiple values. We create and manage property container objects by:

- defining properties (named locations), either at object creation time or later
- assigning values, either at object creation time or later
- accessing values later, using the location names (property names)
- deleting properties via `delete`
- determining container contents with `in`, `hasOwnProperty(..)` / `hasOwn(..)`, `Object.entries(..)` / `Object.keys(..)`, etc

But there's a lot more to objects than just static collections of property names and values. In the next chapter, we'll dive under the hood to look at how they actually work.

You Don't Know JS Yet: Objects & Classes - 2nd Edition

Chapter 2: How Objects Work

NOTE:
Work in progress

Objects are not just containers for multiple values, though clearly that's the context for most interactions with objects.

To fully understand the object mechanism in JS, and get the most out of using objects in our programs, we need to look more closely at a number of characteristics of objects (and their properties) which can affect their behavior when interacting with them.

These characteristics that define the underlying behavior of objects are collectively referred to in formal terms as the "metaobject protocol" (MOP)[1]. The MOP is useful not only for understanding how objects will behave, but also for overriding the default behaviors of objects to bend the language to fit our program's needs more fully.

Property Descriptors

Each property on an object is internally described by what's known as a "property descriptor". This is, itself, an object (aka, "metaobject") with several properties (aka "attributes") on it, dictating how the target property behaves.

We can retrieve a property descriptor for any existing property using `Object.getOwnPropertyDescriptor(..)` (ES5):

```
myObj = {
    favoriteNumber: 42,
    isDeveloper: true,
    firstName: "Kyle"
};

Object.getOwnPropertyDescriptor(myObj,"favoriteNumber");
// {
//     value: 42,
//     enumerable: true,
//     writable: true,
//     configurable: true
// }
```

[1] "Metaobject", Wikipedia; https://en.wikipedia.org/wiki/Metaobject ; Accessed July 2022.

We can even use such a descriptor to define a new property on an object, using `Object.defineProperty(..)` (ES5):

```
anotherObj = {};

Object.defineProperty(anotherObj,"fave",{
    value: 42,
    enumerable: true,      // default if omitted
    writable: true,        // default if omitted
    configurable: true     // default if omitted
});

anotherObj.fave;           // 42
```

If an existing property has not already been marked as non-configurable (with `configurable: false` in its descriptor), it can always be re-defined/overwritten using `Object.defineProperty(..)`.

WARNING:

A number of earlier sections in this chapter refer to "copying" or "duplicating" properties. One might assume such copying/duplication would be at the property descriptor level. However, none of those operations actually work that way; they all do simple `=` style access and assignment, which has the effect of ignoring any nuances in how the underlying descriptor for a property is defined.

Though it seems far less common out in the wild, we can even define multiple properties at once, each with their own descriptor:

```
anotherObj = {};

Object.defineProperties(anotherObj,{
    "fave": {
        // a property descriptor
    },
    "superFave": {
        // another property descriptor
    }
});
```

It's not very common to see this usage, because it's rarer that you need to specifically control the definition of multiple properties. But it may be useful in some cases.

Accessor Properties

A property descriptor usually defines a `value` property, as shown above. However, a special kind of property, known as an "accessor property" (aka, a getter/setter), can be defined. For these a property like this, its descriptor does not define a fixed `value` property, but would instead look something like this:

```
{
    get() { .. },    // function to invoke when retrieving the value
    set(v) { .. },   // function to invoke when assigning the value
    // .. enumerable, etc
}
```

A getter looks like a property access (`obj.prop`), but under the covers it invokes the `get()` method as defined; it's sort of like if you had called `obj.prop()`. A setter looks like a property assignment (`obj.prop = value`), but it invokes the `set(..)` method as defined; it's sort of like if you had called `obj.prop(value)`.

Let's illustrate a getter/setter accessor property:

```
anotherObj = {};

Object.defineProperty(anotherObj,"fave",{
    get() { console.log("Getting 'fave' value!"); return 123; },
    set(v) { console.log(`Ignoring ${v} assignment.`); }
});

anotherObj.fave;
// Getting 'fave' value!
// 123

anotherObj.fave = 42;
// Ignoring 42 assignment.

anotherObj.fave;
// Getting 'fave' value!
// 123
```

Enumerable, Writable, Configurable

Besides `value` or `get()` / `set(..)`, the other 3 attributes of a property descriptor are (as shown above):

- `enumerable`
- `writable`
- `configurable`

The `enumerable` attribute controls whether the property will appear in various enumerations of object properties, such as `Object.keys(..)`,

`Object.entries(..)`, `for..in` loops, and the copying that occurs with the `...` object spread and `Object.assign(..)`. Most properties should be left enumerable, but you can mark certain special properties on an object as non-enumerable if they shouldn't be iterated/copied.

The `writable` attribute controls whether a `value` assignment (via `=`) is allowed. To make a property "read only", define it with `writable: false`. However, as long as the property is still configurable, `Object.defineProperty(..)` can still change the value by setting `value` differently.

The `configurable` attribute controls whether a property's **descriptor** can be re-defined/overwritten. A property that's `configurable: false` is locked to its definition, and any further attempts to change it with `Object.defineProperty(..)` will fail. A non-configurable property can still be assigned new values (via `=`), as long as `writable: true` is still set on the property's descriptor.

Object Sub-Types

There are a variety of specialized sub-types of objects in JS. But by far, the two most common ones you'll interact with are arrays and `function`s.

NOTE:

By "sub-type", we mean the notion of a derived type that has inherited the behaviors from a parent type but then specialized or extended those behaviors. In other words, values of these sub-types are fully objects, but are also *more than just* objects.

Arrays

Arrays are objects that are specifically intended to be **numerically indexed**, rather than using string named property locations. They are still objects, so a named property like `favoriteNumber` is legal. But it's greatly frowned upon to mix named properties into numerically indexed arrays.

Arrays are preferably defined with literal syntax (similar to objects), but with the `[ .. ]` square brackets rather than `{ .. }` curly brackets:

```
myList = [ 23, 42, 109 ];
```

JS allows any mixture of value types in arrays, including objects, other arrays, functions, etc. As you're likely already aware, arrays are "zero-indexed", meaning the first element in the array is at the index `0`, not `1`:

```
myList = [ 23, 42, 109 ];

myList[0];      // 23
myList[1];      // 42
```

Recall that any string property name on an object that "looks like" an integer – is able to be validly coerced to a numeric integer – will actually be treated like an integer property (aka, integer index). The same goes for arrays. You should always use `42` as an integer index (aka, property name), but if you use the string `"42"`, JS will assume you meant that as an integer and do that for you.

```
// "2" works as an integer index here, but it's not advised
myList["2"];    // 109
```

One exception to the "no named properties on arrays" *rule* is that all arrays automatically expose a `length` property, which is automatically kept updated with the "length" of the array.

```
myList = [ 23, 42, 109 ];

myList.length;   // 3

// "push" another value onto the end of the list
myList.push("Hello");

myList.length;   // 4
```

WARNING:

Many JS developers incorrectly believe that array `length` is basically a *getter* (see "Accessor Properties" earlier in this chapter), but it's not. The offshoot is that these developers feel like it's "expensive" to access this property – as if JS has to on-the-fly recompute the length – and will thus do things like capture/store the length of an array before doing a non-mutating loop over it. This used to be "best practice" from a performance perspective. But for at least 10 years now, that's actually been an anti-pattern, because the JS engine is more efficient at managing the `length` property than our JS code is at trying to "outsmart" the engine to avoid invoking something we think is a *getter*. It's more efficient to let the JS engine do its job, and just access the property whenever and however often it's needed.

Empty Slots JS arrays also have a really unfortunate "flaw" in their design, referred to as "empty slots". If you assign an index of an array more than one position beyond the current end of the array, JS will leave the in between slots "empty" rather than auto-assigning them to `undefined` as you might expect:

```
myList = [ 23, 42, 109 ];
myList.length;              // 3

myList[14] = "Hello";
myList.length;              // 15
```

```
myList;                     // [ 23, 42, 109, empty x 11, "Hello" ]

// looks like a real slot with a
// real `undefined` value in it,
// but beware, it's a trick!
myList[9];                  // undefined
```

You might wonder why empty slots are so bad? One reason: there are APIs in JS, like array's `map(..)`, where empty slots are surprisingly skipped over! Never, ever intentionally create empty slots in your arrays. This in undebateably one of JS's "bad parts".

Functions

I don't have much specifically to say about functions here, other than to point out that they are also sub-object-types. This means that in addition to being executable, they can also have named properties added to or accessed from them.

Functions have two pre-defined properties you may find yourself interacting with, specifially for meta-programming purposes:

```
function help(opt1,opt2,...remainingOpts) {
    // ..
}

help.name;          // "help"
help.length;        // 2
```

The `length` of a function is the count of its explicitly defined parameters, up to but not including a parameter that either has a default value defined (e.g., `param = 42`) or a "rest parameter" (e.g., `...remainingOpts`).

Avoid Setting Function-Object Properties You should avoid assigning properties on function objects. If you're looking to store extra information associated with a function, use a separate `Map(..)` (or `WeakMap(..)`) with the function object as the key, and the extra information as the value.

```
extraInfo = new Map();

extraInfo.set(help,"this is some important information");

// later:
extraInfo.get(help);    // "this is some important information"
```

Object Characteristics

In addition to defining behaviors for specific properties, certain behaviors are configurable across the whole object:

- extensible
- sealed
- frozen

Extensible

Extensibility refers to whether an object can have new properties defined/added to it. By default, all objects are extensible, but you can change shut off extensibility for an object:

```
myObj = {
    favoriteNumber: 42
};

myObj.firstName = "Kyle";                   // works fine

Object.preventExtensions(myObj);

myObj.nicknames = [ "getify", "ydkjs" ];    // fails
myObj.favoriteNumber = 123;                 // works fine
```

In non-strict-mode, an assignment that creates a new property will silently fail, whereas in strict mode an exception will be thrown.

Sealed

// TODO

Frozen

// TODO

Extending The MOP

As mentioned at the start of this chapter, objects in JS behave according to a set of rules referred to as the Metaobject Protocol (MOP)[2]. Now that we understand more fully how objects work by default, we want to turn our attention to how we can hook into some of these default behaviors and override/customize them.

// TODO

[2]"Metaobject", Wikipedia; https://en.wikipedia.org/wiki/Metaobject ; Accessed July 2022.

`[[Prototype]]` Chain

One of the most important, but least obvious, characteristics of an object (part of the MOP) is referred to as its "prototype chain"; the official JS specification notation is `[[Prototype]]`. Make sure not to confuse this `[[Prototype]]` with a public property named `prototype`. Despite the naming, these are distinct concepts.

The `[[Prototype]]` is an internal linkage that an object gets by default when its created, pointing to another object. This linkage is a hidden, often subtle characteristic of an object, but it has profound impacts on how interactions with the object will play out. It's referred to as a "chain" because one object links to another, which in turn links to another, … and so on. There is an *end* or *top* to this chain, where the linkage stops and there's no further to go. More on that shortly.

We already saw several implications of `[[Prototype]]` linkage in Chapter 1. For example, by default, all objects are `[[Prototype]]`-linked to the built-in object named `Object.prototype`.

WARNING:

That `Object.prototype` name itself can be confusing, since it uses a property called `prototype`. How are `[[Prototype]]` and `prototype` related!? Put such questions/confusion on pause for a bit, as we'll come back an explain the differences between `[[Prototype]]` and `prototype` later in this chapter. For the moment, just assume the presence of this important but weirdly named built-in object, `Object.prototype`.

Let's consider some code:

```
myObj = {
    favoriteNumber: 42
};
```

That should look familiar from Chapter 1. But what you *don't see* in this code is that the object there was automatically linked (via its internal `[[Prototype]]`) to that automatically built-in, but weirdly named, `Object.prototype` object.

When we do things like:

```
myObj.toString();                           // "[object Object]"

myObj.hasOwnProperty("favoriteNumber");    // true
```

We're taking advantage of this internal `[[Prototype]]` linkage, without really realizing it. Since `myObj` does not have `toString` or `hasOwnProperty` properties defined on it, those property accesses actually end up **DELEGATING** the access to continue its lookup along the `[[Prototype]]` chain.

Since `myObj` is `[[Prototype]]`-linked to the object named `Object.prototype`, the lookup for `toString` and `hasOwnProperty` properties continues on that object; and indeed, these methods are found there!

The ability for `myObj.toString` to access the `toString` property even though it doesn't actually have it, is commonly referred to as "inheritance", or more specifically, "prototypal inheritance". The `toString` and `hasOwnProperty` properties, along with many others, are said to be "inherited properties" on `myObj`.

NOTE:

I have a lot of frustrations with the usage of the word "inheritance" here – it should be called "delegation"! – but that's what most people refer to it as, so we'll begrudgingly comply and use that same terminology for now (albeit under protest, with " quotes). I'll save my objections for an appendix of this book.

`Object.prototype` has several built-in properties and methods, all of which are "inherited" by any object that is `[[Prototype]]`-linked, either directly or indirectly through another object's linkage, to `Object.prototype`.

Some common "inherited" properties from `Object.prototype` include:

- `constructor`
- `__proto__`
- `toString()`
- `valueOf()`
- `hasOwnProperty(..)`
- `isPrototypeOf(..)`

Recall `hasOwnProperty(..)`, which we saw earlier gives us a boolean check for whether a certain property (by string name) is owned by an object:

```
myObj = {
    favoriteNumber: 42
};

myObj.hasOwnProperty("favoriteNumber");    // true
```

It's always been considered somewhat unfortunate (semantic organization, naming conflicts, etc) that such an important utility as `hasOwnProperty(..)` was included on the Object `[[Prototype]]` chain as an instance method, instead of being defined as a static utility.

As of ES2022, JS has finally added the static version of this utility: `Object.hasOwn(..)`.

```
myObj = {
    favoriteNumber: 42
};
```

```
Object.hasOwn(myObj,"favoriteNumber");   // true
```

This form is now considered the more preferable and robust option, and the instance method (`hasOwnProperty(..)`) form should now generally be avoided.

Somewhat unfortunately and inconsistently, there's not (yet, as of time of writing) corresponding static utilities, like `Object.isPrototype(..)` (instead of the instance method `isPrototypeOf(..)`). But at least `Object.hasOwn(..)` exists, so that's progress.

Creating An Object With A Different `[[Prototype]]`

By default, any object you create in your programs will be `[[Prototype]]`-linked to that `Object.prototype` object. However, you can create an object with a different linkage like this:

```
myObj = Object.create(differentObj);
```

The `Object.create(..)` method takes its first argument as the value to set for the newly created object's `[[Prototype]]`.

One downside to this approach is that you aren't using the `{ .. }` literal syntax, so you don't initially define any contents for `myObj`. You typically then have to define properties one-by-one, using `=`.

NOTE:

The second, optional argument to `Object.create(..)` is – like the second argument to `Object.defineProperties(..)` as discussed earlier – an object with properties that hold descriptors to initially define the new object with. In practice out in the wild, this form is rarely used, likely because it's more awkward to specify full descriptors instead of just name/value pairs. But it may come in handy in some limited cases.

Alternately, but less preferably, you can use the `{ .. }` literal syntax along with a special (and strange looking!) property:

```
myObj = {
    __proto__: differentObj,

    // .. the rest of the object definition
};
```

WARNING:

The strange looking `__proto__` property has been in some JS engines for more than 20 years, but was only standardized in JS as of ES6 (in 2015). Even still, it was added in Appendix B of the specification[3], which lists features that TC39 begrudgingly includes because they exist popularly in various browser-based JS engines and therefore are a de-facto reality even if they didn't originate with TC39. This feature is thus "guaranteed" by the spec to exist in all conforming browser-based JS engines, but is not necessarily guaranteed to work in other independent JS engines. Node.js uses the JS engine (v8) from the Chrome browser, so Node.js gets `__proto__` by default/accident. Be careful when using `__proto__` to be aware of all the JS engine environments your code will run in.

Whether you use `Object.create(..)` or `__proto__`, the created object in question will usually be `[[Prototype]]`-linked to a different object than the default `Object.prototype`.

Empty `[[Prototype]]` Linkage We mentioned above that the `[[Prototype]]` chain has to stop somewhere, so as to have lookups not continue forever. `Object.prototype` is typically the top/end of every `[[Prototype]]` chain, as its own `[[Prototype]]` is `null`, and therefore there's nowhere else to continue looking.

However, you can also define objects with their own `null` value for `[[Prototype]]`, such as:

```
emptyObj = Object.create(null);
// or: emptyObj = { __proto__: null }

emptyObj.toString;    // undefined
```

It can be quite useful to create an object with no `[[Prototype]]` linkage to `Object.prototype`. For example, as mentioned in Chapter 1, the `in` and `for..in` constructs will consult the `[[Prototype]]` chain for inherited properties. But this may be undesirable, as you may not want something like `"toString" in myObj` to resolve successfully.

Moreover, an object with an empty `[[Prototype]]` is safe from any accidental "inheritance" collision between its own property names and the ones it "inherits" from elsewhere. These types of (useful!) objects are sometimes referred to in popular parlance as "dictionary objects".

[3] "Appendix B: Additional ECMAScript Features for Web Browsers", ECMAScript 2022 Language Specification; https://262.ecma-international.org/13.0/#sec-additional-ecmascript-features-for-web-browsers ; Accessed July 2022

`[[Prototype]]` vs `prototype`

Notice that public property name `prototype` in the name/location of this special object, `Object.prototype`? What's that all about?

`Object` is the `Object(..)` function; by default, all functions (which are themselves objects!) have such a `prototype` property on them, pointing at an object.

Any here's where the name conflict between `[[Prototype]]` and `prototype` really bites us. The `prototype` property on a function doesn't define any linkage that the function itself experiences. Indeed, functions (as objects) have their own internal `[[Prototype]]` linkage somewhere else – more on that in a second.

Rather, the `prototype` property on a function refers to an object that should be *linked TO* by any other object that is created when calling that function with the `new` keyword:

```
myObj = {};

// is basically the same as:
myObj = new Object();
```

Since the `{ .. }` object literal syntax is essentially the same as a `new Object()` call, the built-in object named/located at `Object.prototype` is used as the internal `[[Prototype]]` value for the new object we create and name `myObj`.

Phew! Talk about a topic made significantly more confusing just because of the name overlap between `[[Prototype]]` and `prototype`!

But where do functions themselves (as objects!) link to, `[[Prototype]]` wise? They link to `Function.prototype`, yet another built-in object, located at the `prototype` property on the `Function(..)` function.

In other words, you can think of functions themselves as having been "created" by a `new Function(..)` call, and then `[[Prototype]]`-linked to the `Function.prototype` object. This object contains properties/methods all functions "inherit" by default, such as `toString()` (to string serialize the source code of a function) and `call(..)` / `apply(..)` / `bind(..)` (we'll explain these later in this book).

Objects Behavior

Properties on objects are internally defined and controlled by a "descriptor" metaobject, which includes attributes such as `value` (the property's present value) and `enumerable` (a boolean controlling whether the property is included in enumerable-only listings of properties/property names).

The way object and their properties work in JS is referred to as the "metaob-

ject protocol" (MOP)[4]. We can control the precise behavior of properties via `Object.defineProperty(..)`, as well as object-wide behaviors with `Object.freeze(..)`. But even more powerfully, we can hook into and override certain default behaviors on objects using special pre-defined Symbols.

Prototypes are internal linkages between objects that allow property or method access against one object – if the property/method requested is absent – to be handled by "delegating" that access lookup to another object. When the delegation involves a method, the context for the method to run in is shared from the initial object to the target object via the `this` keyword.

[4]"Metaobject", Wikipedia; https://en.wikipedia.org/wiki/Metaobject ; Accessed July 2022.

You Don't Know JS Yet: Objects & Classes - 2nd Edition

Chapter 3: Classy Objects

NOTE:
Work in progress

The class-design pattern generally entails defining a *type of thing* (class), including data (members) and behaviors (methods), and then creating one or more concrete *instances* of this class definition as actual objects that can interact and perform tasks. Moreover, class-orientation allows declaring a relationship between two or more classes, through what's called "inheritance", to derive new and augmented "subclasses" that mix-n-match and even re-define behaviors.

Prior to ES6 (2015), JS developers mimicked aspects of class-oriented (aka "object-oriented") design using plain functions and objects, along with the `[[Prototype]]` mechanism (as explained in the previous chapter) – so called "prototypal classes".

But to many developers joy and relief, ES6 introduced dedicated syntax, including the `class` and `extends` keywords, to express class-oriented design more declaratively.

At the time of ES6's `class` being introduced, this new dedicated syntax was almost entirely *just syntactic sugar* to make class definitions more convenient and readable. However, in the many years since ES6, `class` has matured and grown into its own first class feature mechanism, accruing a significant amount of dedicated syntax and complex behaviors that far surpass the pre-ES6 "prototypal class" capabilities.

Even though `class` now bears almost no resemblance to older "prototypal class" code style, the JS engine is still *just* wiring up objects to each other through the existing `[[Prototype]]` mechanism. In other words, `class` is not its own separate pillar of the language (as `[[Prototype]]` is), but more like the fancy, decorative *Capital* that tops the pillar/column.

That said, since `class` style code has now replaced virtually all previous "prototypal class" coding, the main text here focuses only on `class` and its various particulars. For historical purposes, we'll briefly cover the old "prototypal class" style in an appendix.

When Should I Class-Orient My Code?

Class-orientation is a design pattern, which means it's a choice for how you organize the information and behavior in your program. It has pros and cons.

It's not a universal solution for all tasks.

So how do you know when you should use classes?

In a theoretical sense, class-orientation is a way of dividing up the business domain of a program into one or more pieces that can each be defined by an "is-a" classification: grouping a thing into the set (or sets) of characteristics that thing shares with other similar things. You would say "X is a Y", meaning X has (at least) all the characteristics of a thing of kind Y.

For example, consider computers. We could say a computer is electrical, since it uses electrical current (voltage, amps, etc) as power. It's furthermore electronic, because it manipulates the electrical current beyond simply routing electrons around (electrical/magnetic fields), creating a meaningful circuit to manipulate the current into performing more complex tasks. By contrast, a basic desk lamp is electrical, but not really electronic.

We could thus define a class `Electrical` to describe what electrical devices need and can do. We could then define a further class `Electronic`, and define that in addition to being electrical, `Electronic` things manipulate electricity to create more specialized outcomes.

Here's where class-orientation starts to shine. Rather than re-define all the `Electrical` characteristics in the `Electronic` class, we can define `Electronic` in such a way that it "shares" or "inherits" those characteristics from `Electrical`, and then augments/redefines the unique behaviors that make a device electronic. This relationship between the two classes – called "inheritance" – is a key aspect of class-orientation.

So class-orientation is a way of thinking about the entities our program needs, and classifying them into groupings based on their characteristics (what information they hold, what operations can be performed on that data), and defining the relationships between the different grouping of characteristics.

But moving from the theoretical into in a bit more pragmatic perspective: if your program needs to hold and use multiple collections (instances) of alike data/behavior at once, you *may* benefit from class-orientation.

Time For An Example

Here's a short illustration.

A couple of decades ago, right after I had gone through nearly all of a Computer Science degree in college, I found myself sitting in my first professional software developer job. I was tasked with building, all by myself, a timesheet and payroll tracking system. I built the backend in PHP (using MySQL for the DB) and used JS for the interface (early as it was in its maturity way back around the turn of the century).

Since my CS degree had emphasized class-orientation heavily throughout my

courses, I was eager to put all that theory to work. For my program's design, I defined the concept of a "timesheet" entity as a collection of 2-3 "week" entities, and each "week" as a collection of 5-7 "day" entities, and each "day" as a collection of "task" entities.

If I wanted to know how many hours were logged into a timesheet instance, I could call a `totalTime()` operation on that instance. The timesheet defined this operation by looping over its collection of weeks, calling `totalTime()` on each of them and summing the values. Each week did the same for all its days, and each day did the same for all its tasks.

The notion being illustrated here, one of the fundamentals of design patterns like class-orientation, is called *encapsulation*. Each entity level encapsulated (e.g., controlled, hid, abstracted) internal details (data and behavior) while presenting a useful external interface.

But encapsulation alone isn't a sufficient justification for class-orientation. Other design patterns offer sufficient encapsulation.

How did my class design take advantage of inheritance? I had a base class that defined a set of operations like `totalTime()`, and each of my entity class types extended/subclassed this base class. That meant that each of them inherited this summation-of-total-time capability, but where each of them applied their own extensions and definitions for the internal details of *how* to do that work.

There's yet another aspect of the design pattern at play, which is *composition*: each entity was defined as a collection of other entities.

Single vs Multiple

I mentioned above that a pragmatic way of deciding if you need class-orientation is if your program is going to have multiple instances of a single kind/type of behavior (aka, "class"). In the timesheet example, we had 4 classes: Timesheet, Week, Day, and Task. But for each class, we had multiple instances of each at once.

Had we instead only needed a single instance of a class, like just one `Computer` thing that was an instance of the `Electronic` class, which was a subclass of the `Electrical` class, then class-orientation may not offer quite as much benefit. In particular, if the program doesn't need to create an instance of the `Electrical` class, then there's no particular benefit to separating `Electrical` from `Electronic`, so we aren't really getting any help from the inheritance aspect of class-orientation.

So, if you find yourself designing a program by dividing up a business problem domain into different "classes" of entities, but in the actual code of the program you are only ever need one concrete *thing* of one kind/definition of behavior (aka, "class"), you might very well not actually need class-orientation. There are other design patterns which may be a more efficient match to your effort.

But if you find yourself wanting to define classes, and subclasses which inherit from them, and if you're going to be instantiating one or more of those classes multiple times, then class-orientation is a good candidate. And to do class-orientation in JS, you're going to need the `class` keyword.

Keep It `classy`

`class` defines either a declaration or expression for a class. As a declaration, a class definition appears in a statement position and looks like this:

```
class Point2d {
    // ..
}
```

As an expression, a class definition appears in a value position and can either have a name or be anonymous:

```
// named class expression
const pointClass = class Point2d {
    // ..
};

// anonymous class expression
const anotherClass = class {
    // ..
};
```

The contents of a `class` body typically include one or more method definitions:

```
class Point2d {
    setX(x) {
        // ..
    }
    setY(y) {
        // ..
    }
}
```

Inside a `class` body, methods are defined without the `function` keyword, and there's no `,` or `;` separators between the method definitions.

NOTE:

Inside a `class` block, all code runs in strict-mode even without the `"use strict"` pragma present in the file or its functions. In particular, this impacts the `this` behavior for function calls, as explained in Chapter 4.

The Constructor

One special method that all classes have is called a "constructor". If omitted, there's a default empty constructor assumed in the definition.

The constructor is invoked any time a `new` instance of the class is created:

```
class Point2d {
    constructor() {
        console.log("Here's your new instance!");
    }
}

var point = new Point2d();
// Here's your new instance!
```

Even though the syntax implies a function actually named `constructor` exists, JS defines a function as specified, but with the name of the class (`Point2d` above):

```
typeof Point2d;       // "function"
```

It's not *just* a regular function, though; this special kind of function behaves a bit differently:

```
Point2d.toString();
// class Point2d {
//   ..
// }

Point2d();
// TypeError: Class constructor Point2d cannot
// be invoked without 'new'

Point2d.call({});
// TypeError: Class constructor Point2d cannot
// be invoked without 'new'
```

You can construct as many different instances of a class as you need:

```
var one = new Point2d();
var two = new Point2d();
var three = new Point2d();
```

Each of `one`, `two`, and `three` here are objects that are independent instances of the `Point2d` class.

NOTE:

Each of the `one`, `two`, and `three` objects have a `[[Prototype]]` linkage to the `Point2d.prototype` object (see Chapter 2). In this code, `Point2d` is both a `class` definition and the constructor function of the same name.

If you add a property to the object `one`:

```
one.value = 42;
```

That property now exists only on `one`, and does not exist in any way that the independent `two` or `three` objects can access:

```
two.value;      // undefined
three.value;    // undefined
```

Class Methods

As shown above, a class definition can include one or more method definitions:

```
class Point2d {
    constructor() {
        console.log("Here's your new instance!");
    }
    setX(x) {
        console.log(`Setting x to: ${x}`);
        // ..
    }
}

var point = new Point2d();

point.setX(3);
// Setting x to: 3
```

The `setX` property (method) *looks like* it exists on (is owned by) the `point` object here. But that's a mirage. Each class method is added to the `prototype`object, a property of the constructor function.

So, `setX(..)` only exists as `Point2d.prototype.setX`. Since `point` is `[[Prototype]]` linked to `Point2d.prototype` (see Chapter 2) via the `new` keyword instantiation, the `point.setX(..)` reference traverses the `[[Prototype]]` chain and finds the method to execute.

Class methods should only be invoked via an instance; `Point2d.setX(..)` doesn't work because there *is no* such property. You *could* invoke `Point2d.prototype.setX(..)`, but that's not generally proper/advised in standard class-oriented coding. Always access class methods via the instances.

Class Instance `this`

We will cover the `this` keyword in much more detail in a subsequent chapter. But as it relates to class-oriented code, the `this` keyword generally refers to the current instance that is the context of any method invocation.

In the constructor, as well as any methods, you can use `this.` to either add or access properties on the current instance:

```
class Point2d {
    constructor(x,y) {
        // add properties to the current instance
        this.x = x;
        this.y = y;
    }
    toString() {
        // access the properties from the current instance
        console.log(`(${this.x},${this.y})`);
    }
}

var point = new Point2d(3,4);

point.x;                // 3
point.y;                // 4

point.toString();       // (3,4)
```

Any properties not holding function values, which are added to a class instance (usually via the constructor), are referred to as *members*, as opposed to the term *methods* for executable functions.

While the `point.toString()` method is running, its `this` reference is pointing at the same object that `point` references. That's why both `point.x` and `this.x` reveal the same `3` value that the constructor set with its `this.x = x` operation.

Public Fields

Instead of defining a class instance member imperatively via `this.` in the constructor or a method, classes can declaratively define *fields* in the `class` body, which correspond directly to members that will be created on each instance:

```
class Point2d {
    // these are public fields
    x = 0
    y = 0

    constructor(x,y) {
        // set properties (fields) on the current instance
```

```
        this.x = x;
        this.y = y;
    }
    toString() {
        // access the properties from the current instance
        console.log(`(${this.x},${this.y})`);
    }
}
```

Public fields can have a value initialization, as shown above, but that's not required. If you don't initialize a field in the class definition, you almost always should initialize it in the constructor.

Fields can also reference each other, via natural `this.` access syntax:

```
class Point3d {
    // these are public fields
    x
    y = 4
    z = this.y * 5

    // ..
}
```

TIP:

You can mostly think of public field declarations as if they appear at the top of the `constructor(..)`, each prefixed with an implied `this.` that you get to omit in the declarative `class` body form. But, there's a catch! See "That's Super!" later for more information about it.

Just like computed property names (see Chapter 1), field names can be computed:

```
var coordName = "x";

class Point2d {
    // computed public field
    [coordName.toUpperCase()] = 42

    // ..
}

var point = new Point2d(3,4);

point.x;        // 3
point.y;        // 4
```

```
point.X;        // 42
```

Avoid This One pattern that has emerged and grown quite popular, but which I firmly believe is an anti-pattern for `class`, looks like the following:

```
class Point2d {
    x = null
    y = null
    getDoubleX = () => this.x * 2

    constructor(x,y) {
        this.x = x;
        this.y = y;
    }
    toString() { /* .. */ }
}

var point = new Point2d(3,4);

point.getDoubleX();     // 6
```

See the field holding an `=>` arrow function? I say this is a no-no. But why? Let's unwind what's going on.

First, why do this? Because JS developers seem to be perpetually frustrated by the dynamic `this` binding rules (see Chapter 4), so they force a `this` binding via the `=>` arrow function. That way, no matter how `getDoubleX()` is invoked, it's always `this`-bound to the particular instance. That's an understandable convenience to desire, but… it betrays the very nature of the `this` / `[[Prototype]]` pillar of the language. How?

Let's consider the equivalent code to the previous snippet:

```
class Point2d {
    constructor(x,y) {
        this.x = null;
        this.y = null;
        this.getDoubleX = () => this.x * 2;

        this.x = x;
        this.y = y;
    }
    toString() { /* .. */ }
}

var point = new Point2d(3,4);
```

```
point.getDoubleX();     // 6
```

Can you spot the problem? Look closely. I'll wait.

…

We've made it clear repeatedly so far that `class` definitions put their methods on the class constructor's `prototype` object – that's where they belong! – such that there's just one of each function and it's inherited (shared) by all instances. That's what will happen with `toString()` in the above snippet.

But what about `getDoubleX()`? That's essentially a class method, but it won't be handled by JS quite the same as `toString()` will. Consider:

```
Object.hasOwn(point,"x");                // true -- good
Object.hasOwn(point,"toString");         // false -- good
Object.hasOwn(point,"getDoubleX");       // true -- oops :(
```

You see now? By defining a function value and attaching it as a field/member property, we're losing the shared prototypal method'ness of the function, and it becomes just like any per-instance property. That means we're creating a new function property **for each instance**, rather than it being created just once on the class constructor's `prototype`.

That's wasteful in performance and memory, even if by a tiny bit. That alone should be enough to avoid it.

But I would argue that way more importantly, what you've done with this pattern is invalidate the very reason why using `class` and `this`-aware methods is even remotely useful/powerful!

If you go to all the trouble to define class methods with `this.` references throughout them, but then you lock/bind most or all of those methods to a specific object instance, you've basically travelled all the way around the world just to go next door.

If all you want are function(s) that are statically fixed to a particular "context", and don't need any dynamicism or sharing, what you want is… **closure**. And you're in luck: I wrote a whole book in this series ("Scope & Closures") on how to use closure so functions remember/access their statically defined scope (aka "context"). That's a way more appropriate, and simpler to code, approach to get what you're after.

Don't abuse/misuse `class` and turn it into a over-hyped, glorified collection of closure.

To be clear, I'm *not* saying: never use `=>` arrow functions inside classes.

I *am* saying: never attach an `=>` arrow function as an instance property in place of a dynamic prototypal class method, either out of mindless habit, or laziness in typing fewer characters, or misguided `this`-binding convenience.

In a subsequent chapter, we'll dive deep into how to understand and properly leverage the full power of the dynamic `this` mechanism.

Class Extension

The way to unlock the power of class inheritance is through the `extends` keyword, which defines a relationship between two classes:

```
class Point2d {
    x = 3
    y = 4

    getX() {
        return this.x;
    }
}

class Point3d extends Point2d {
    x = 21
    y = 10
    z = 5

    printDoubleX() {
        console.log(`double x: ${this.getX() * 2}`);
    }
}

var point = new Point2d();

point.getX();                   // 3

var anotherPoint = new Point3d();

anotherPoint.getX();            // 21
anotherPoint.printDoubleX();    // double x: 42
```

Take a few moments to re-read that code snippet and make sure you fully understand what's happening.

The base class `Point2d` defines fields (members) called `x` and `y`, and gives them the initial values `3` and `4`, respectively. It also defines a `getX()` method that accesses this `x` instance member and returns it. We see that behavior illustrated in the `point.getX()` method call.

But the `Point3d` class extends `Point2d`, making `Point3d` a derived-class, child-class, or (most commonly) subclass. In `Point3d`, the same `x` property that's inherited from `Point2d` is re-initialized with a different `21` value, as is the `y` overridden to value from `4`, to `10`.

It also adds a new `z` field/member method, as well as a `printDoubleX()` method, which itself calls `this.getX()`.

When `anotherPoint.printDoubleX()` is invoked, the inherited `this.getX()` is thus invoked, and that method makes reference to `this.x`. Since `this` is pointing at the class instance (aka, `anotherPoint`), the value it finds is now `21` (instead of `3` from the `point` object's `x` member).

Extending Expressions

// TODO: cover `class Foo extends ..` where `..` is an expression, not a class-name

Overriding Methods

In addition to overriding a field/member in a subclass, you can also override (redefine) a method:

```
class Point2d {
    x = 3
    y = 4

    getX() {
        return this.x;
    }
}

class Point3d extends Point2d {
    x = 21
    y = 10
    z = 5

    getX() {
        return this.x * 2;
    }
    printX() {
        console.log(`double x: ${this.getX()}`);
    }
}

var point = new Point3d();

point.printX();        // double x: 42
```

The `Point3d` subclass overrides the inherited `getX()` method to give it different behavior. However, you can still instantiate the base `Point2d` class, which would then give an object that uses the original (`return this.x;`) definition for `getX()`.

If you want to access an inherited method from a subclass even if it's been overridden, you can use `super` instead of `this`:

```
class Point2d {
    x = 3
    y = 4

    getX() {
        return this.x;
    }
}

class Point3d extends Point2d {
    x = 21
    y = 10
    z = 5

    getX() {
        return this.x * 2;
    }
    printX() {
        console.log(`x: ${super.getX()}`);
    }
}

var point = new Point3d();

point.printX();        // x: 21
```

The ability for methods of the same name, at different levels of the inheritance hierarchy, to exhibit different behavior when either accessed directly, or relatively with `super`, is called *method polymorphism*. It's a very powerful part of class-orientation, when used appropriately.

That's Super!

In addition to a subclass method accessing an inherited method definition (even if overriden on the subclass) via `super.` reference, a subclass constructor must manually invoke the inherited base class constructor via `super(..)` function invocation:

```
class Point2d {
    x
    y
    constructor(x,y) {
        this.x = x;
        this.y = y;
```

```
    }
}

class Point3d extends Point2d {
    z
    constructor(x,y,z) {
        super(x,y);
        this.z = z;
    }
    toString() {
        console.log(`(${this.x},${this.y},${this.z})`);
    }
}

var point = new Point3d(3,4,5);

point.toString();       // (3,4,5)
```

WARNING:

An explicitly defined subclass constructor *must* call `super(..)` to run the inherited class's initialization, and that must occur before the subclass constructor makes any references to `this` or finishes/returns. Otherwise, a runtime exception will be thrown when that subclass constructor is invoked (via `new`). If you omit the subclass constructor, the default constructor automatically – thankfully! – invokes `super()` for you.

One nuance to be aware of: if you define a field (public or private) inside a subclass, and explicitly define a `constructor(..)` for this subclass, the field initializations will be processed not at the top of the constructor, but *between* the `super(..)` call and any subsequent code in the constructor.

Pay close attention to the order of console messages here:

```
class Point2d {
    x
    y
    constructor(x,y) {
        console.log("Running Point2d(..) constructor");
        this.x = x;
        this.y = y;
    }
}

class Point3d extends Point2d {
    z = console.log("Initializing field 'z'")
```

```
    constructor(x,y,z) {
        console.log("Running Point3d(..) constructor");
        super(x,y);

        console.log(`Setting instance property 'z' to ${z}`);
        this.z = z;
    }
    toString() {
        console.log(`(${this.x},${this.y},${this.z})`);
    }
}

var point = new Point3d(3,4,5);
// Running Point3d(..) constructor
// Running Point2d(..) constructor
// Initializing field 'z'
// Setting instance property 'z' to 5
```

As the console messages illustrate, the `z = ..` field initialization happens *immediately after* the `super(x,y)` call, *before* the `console.log(`Setting instance...`)` is executed. Perhaps think of it like the field initializations attached to the end of the `super(..)` call, so they run before anything else in the constructor does.

Which Class? You may need to determine in a constructor if that class is being instantiated directly, or being instantiated from a subclass with a `super()` call. We can use a special "pseudo property" `new.target`:

```
class Point2d {
    // ..

    constructor(x,y) {
        if (new.target === Point2) {
            console.log("Constructing 'Point2d' instance");
        }
    }

    // ..
}

class Point3d extends Point2d {
    // ..

    constructor(x,y,z) {
        super(x,y);
```

```
        if (new.target === Point3d) {
            console.log("Constructing 'Point3d' instance");
        }
    }

    // ..
}

var point = new Point2d(3,4);
// Constructing 'Point2d' instance

var anotherPoint = new Point3d(3,4,5);
// Constructing 'Point3d' instance
```

But Which Kind Of Instance?

You may want to introspect a certain object instance to see if it's an instance of a specific class. We do this with the `instanceof` operator:

```
class Point2d { /* .. */ }
class Point3d extends Point2d { /* .. */ }

var point = new Point2d(3,4);

point instanceof Point2d;           // true
point instanceof Point3d;           // false

var anotherPoint = new Point3d(3,4,5);

anotherPoint instanceof Point2d;    // true
anotherPoint instanceof Point3d;    // true
```

It may seem strange to see `anotherPoint instanceof Point2d` result in `true`. To understand why better, perhaps it's useful to visualize both `[[Prototype]]` chains:

```
Point2d.prototype
        /       \
       /         \
      /           \
  point   Point3d.prototype
                   \
                    \
                     \
                   anotherPoint
```

The `instanceof` operator doesn't just look at the current object, but rather

traverses the entire class inheritance hierarchy (the `[[Prototype]]` chain) until it finds a match. Thus, `anotherPoint` is an instance of both `Point3d` and `Point2d`.

To illustrate this fact a little more obviously, another (less ergonomic) way of going about the same kind of check as `instanceof` is with the (inherited from `Object.prototype`) utility, `isPrototypeOf(..)`:

```
Point2d.prototype.isPrototypeOf(point);             // true
Point3d.prototype.isPrototypeOf(point);             // false

Point2d.prototype.isPrototypeOf(anotherPoint);      // true
Point3d.prototype.isPrototypeOf(anotherPoint);      // true
```

This utility makes it a little clearer why both `Point2d.prototype.isPrototypeOf(anotherPoint)` and `anotherPoint instanceof Point2d` result in `true`: the object `Point2d.prototype` *is* in the `[[Prototype]]` chain of `anotherPoint`.

If you instead wanted to check if the object instance was *only and directly* created by a certain class, check the instance's `constructor` property.

```
point.constructor === Point2d;          // true
point.constructor === Point3d;          // false

anotherPoint.constructor === Point2d;   // false
anotherPoint.constructor === Point3d;   // true
```

NOTE:

The `constructor` property shown here is *not* actually present on (owned) the `point` or `anotherPoint` instance objects. So where does it come from!? It's on each object's `[[Prototype]]` linked prototype object:
`Point2d.prototype.constructor === Point2d` and
`Point3d.prototype.constructor === Point3d`.

"Inheritance" Is Sharing, Not Copying

It may seem as if `Point3d`, when it `extends` the `Point2d` class, is in essence getting a *copy* of all the behavior defined in `Point2d`. Moreover, it may seem as if the concrete object instance `anotherPoint` receives, *copied down* to it, all the methods from `Point3d` (and by extension, also from `Point2d`).

However, that's not the correct mental model to use for JS's implementation of class-orientation. Recall this base class and subclass definition, as well as instantiation of `anotherPoint`:

```
class Point2d {
    x
    y
```

```
    constructor(x,y) {
        this.x = x;
        this.y = y;
    }
}

class Point3d extends Point2d {
    z
    constructor(x,y,z) {
        super(x,y);
        this.z = z;
    }
    toString() {
        console.log(`(${this.x},${this.y},${this.z})`);
    }
}

var anotherPoint = new Point3d(3,4,5);
```

If you inspect the `anotherPoint` object, you'll see it only has the `x`, `y`, and `z` properties (instance members) on it, but not the `toString()` method:

```
Object.hasOwn(anotherPoint,"x");                    // true
Object.hasOwn(anotherPoint,"y");                    // true
Object.hasOwn(anotherPoint,"z");                    // true

Object.hasOwn(anotherPoint,"toString");             // false
```

Where is that `toString()` method located? On the prototype object:

```
Object.hasOwn(Point3d.prototype,"toString");    // true
```

And `anotherPoint` has access to that method via its `[[Prototype]]` linkage (see Chapter 2). In other words, the prototype objects **share access** to their method(s) with the subclass(es) and instance(s). The method(s) stay in place, and are not copied down the inheritance chain.

As nice as the `class` syntax is, don't forget what's really happening under the syntax: JS is *just* wiring up objects to each other along a `[[Prototype]]` chain.

Static Class Behavior

We've so far emphasized two different locations for data or behavior (methods) to reside: on the constructor's prototype, or on the instance. But there's a third option: on the constructor (function object) itself.

In a traditional class-oriented system, methods defined on a class are not concrete things you could ever invoke or interact with. You have to instantiate a class to have a concrete object to invoke those methods with. Prototypal

languages like JS blur this line a bit: all class-defined methods are "real" functions residing on the constructor's prototype, and you could therefore invoke them. But as I asserted earlier, you really *should not* do so, as this is not how JS assumes you will write your `class`es, and there are some weird corner-case behaviors you may run into. Best to stay on the narrow path that `class` lays out for you.

Not all behavior that we define and want to associate/organize with a class *needs* to be aware of an instance. Moreover, sometimes a class needs to publicly define data (like constants) that developers using that class need to access, independent of any instance they may or may not have created.

So, how does a class system enable defining such data and behavior that should be available with a class but independent of (unaware of) instantiated objects? **Static properties and functions**.

NOTE:

I'll use "static property" / "static function", rather than "member" / "method", just so it's clearer that there's a distinction between instance-bound members / instance-aware methods, and non-instance properties and instance-unaware functions.

We use the `static` keyword in our `class` bodies to distinguish these definitions:

```
class Point2d {
    // class statics
    static origin = new Point2d(0,0)
    static distance(point1,point2) {
        return Math.sqrt(
            ((point2.x - point1.x) ** 2) +
            ((point2.y - point1.y) ** 2)
        );
    }

    // instance members and methods
    x
    y
    constructor(x,y) {
        this.x = x;
        this.y = y;
    }
    toString() {
        return `(${this.x},${this.y})`;
    }
}
```

```
console.log(`Starting point: ${Point2d.origin}`);
// Starting point: (0,0)

var next = new Point2d(3,4);
console.log(`Next point: ${next}`);
// Next point: (3,4)

console.log(`Distance: ${
    Point2d.distance( Point2d.origin, next )
}`);
// Distance: 5
```

The `Point2d.origin` is a static property, which just so happens to hold a constructed instance of our class. And `Point2d.distance(..)` is a static function that computes the 2-dimensional cartesian distance between two points.

Of course, we could have put these two somewhere other than as `static`s on the class definition. But since they're directly related to the `Point2d` class, it makes *most sense* to organize them there.

NOTE:

Don't forget that when you use the `class` syntax, the name `Point2d` is actually the name of a constructor function that JS defines. So `Point2d.origin` is just a regular property access on that function object. That's what I meant at the top of this section when I referred to a third location for storing *things* related to classes; in JS, `static`s are stored as properties on the constructor function. Take care not to confuse those with properties stored on the constructor's `prototype` (methods) and properties stored on the instance (members).

Static Property Initializations

The value in a static initialization (`static whatever = ..`) can include `this` references, which refers to the class itself (actually, the constructor) rather than to an instance:

```
class Point2d {
    // class statics
    static originX = 0
    static originY = 0
    static origin = new this(this.originX,this.originY)

    // ..
}
```

WARNING:

I don't recommend actually doing the `new this(..)` trick I've illustrated here. That's just for illustration purposes. The code would read more cleanly with `new Point2d(this.originX,this.originY)`, so prefer that approach.

An important detail not to gloss over: unlike public field initializations, which only happen once an instantiation (with `new`) occurs, class static initializations always run *immediately* after the `class` has been defined. Moreover, the order of static initializations matters; you can think of the statements as if they're being evaluated one at a time.

Also like class members, static properties do not have to be initialized (default: `undefined`), but it's much more common to do so. There's not much utility in declaring a static property with no initialized value (`static whatever`); Accessing either `Point2d.whatever` or `Point2d.nonExistent` would both result in `undefined`.

Recently (in ES2022), the `static` keyword was extended so it can now define a block inside the `class` body for more sophisticated initialization of `static`s:

```
class Point2d {
    // class statics
    static origin = new Point2d(0,0)
    static distance(point1,point2) {
        return Math.sqrt(
            ((point2.x - point1.x) ** 2) +
            ((point2.y - point1.y) ** 2)
        );
    }

    // static initialization block (as of ES2022)
    static {
        let outerPoint = new Point2d(6,8);
        this.maxDistance = this.distance(
            this.origin,
            outerPoint
        );
    }

    // ..
}

Point2d.maxDistance;        // 10
```

The `let outerPoint = ..` here is not a special `class` feature; it's exactly like a normal `let` declaration in any normal block of scope (see the "Scope &

Closures" book of this series). We're merely declaring a localized instance of `Point2d` assigned to `outerPoint`, then using that value to derive the assignment to the `maxDistance` static property.

Static initialization blocks are also useful for things like `try..catch` statements around expression computations.

Static Inheritance

Class statics are inherited by subclasses (obviously, as statics!), can be overriden, and `super` can be used for base class references (and static function polymorphism), all in much the same way as inheritance works with instance members/methods:

```
class Point2d {
    static origin = /* .. */
    static distance(x,y) { /* .. */ }

    static {
        // ..
        this.maxDistance = /* .. */;
    }

    // ..
}

class Point3d extends Point2d {
    // class statics
    static origin = new Point3d(
        // here, `this.origin` references wouldn't
        // work (self-referential), so we use
        // `super.origin` references instead
        super.origin.x, super.origin.y, 0
    )
    static distance(point1,point2) {
        // here, super.distance(..) is Point2d.distance(..),
        // if we needed to invoke it

        return Math.sqrt(
            ((point2.x - point1.x) ** 2) +
            ((point2.y - point1.y) ** 2) +
            ((point2.z - point1.z) ** 2)
        );
    }

    // instance members/methods
    z
```

```
    constructor(x,y,z) {
        super(x,y);     // <-- don't forget this line!
        this.z = z;
    }
    toString() {
        return `(${this.x},${this.y},${this.z})`;
    }
}

Point2d.maxDistance;        // 10
Point3d.maxDistance;        // 10
```

As you can see, the static property `maxDistance` we defined on `Point2d` was inherited as a static property on `Point3d`.

TIP:

Remember: any time you define a subclass constructor, you'll need to call `super(..)` in it, usually as the first statement. I find that all too easy to forget.

Don't skip over the underlying JS behavior here. Just like method inheritance discussed earlier, the static "inheritance" is *not* a copying of these static properties/functions from base class to subclass; it's sharing via the `[[Prototype]]` chain. Specifically, the constructor function `Point3d()` has its `[[Prototype]]` linkage changed by JS (from the default of `Function.prototype`) to `Point2d`, which is what allows `Point3d.maxDistance` to delegate to `Point2d.maxDistance`.

It's also interesting, perhaps only historically now, to note that static inheritance – which was part of the original ES6 `class` mechanism feature set! – was one specific feature that went beyond "just syntax sugar". Static inheritance, as we see it illustrated here, was *not* possible to achieve/emulate in JS prior to ES6, in the old prototypal-class style of code. It's a special new behavior introduced only as of ES6.

Private Class Behavior

Everything we've discussed so far as part of a `class` definition is publicly visible/accessible, either as static properties/functions on the class, methods on the constructor's `prototype`, or member properties on the instance.

But how do you store information that cannot be seen from outside the class? This was one of the most asked for features, and biggest complaints with JS's `class`, up until it was finally addressed in ES2022.

`class` now supports new syntax for declaring private fields (instance members) and private methods. In addition, private static properties/functions are possible.

Motivation?

Before we illustrate how to do `class` privates, it bears contemplating why this is a helpful feature?

With closure-oriented design patterns (again, see the "Scope & Closures" book of this series), we automatically get "privacy" built-in. When you declare a variable inside a scope, it cannot be seen outside that scope. Period. Reducing the scope visibility of a declaration is helpful in preventing namespace collisions (identical variable names).

But it's even more important to ensure proper "defensive" design of software, the so called "Principle of Least Privilege" [1]. POLP states that we should only expose a piece of information or capability in our software to the smallest surface area necessary.

Over-exposure opens our software up to several issues that complicate software security/maintenance, including another piece of code acting maliciously to do something our code didn't expect or intend. Moreover, there's the less critical but still as problematic concern of other parts of our software relying on (using) parts of our code that we should have reserved as hidden implementation detail. Once other code relies on our code's implementation details, we stop being able to refactor our code without potentially breaking other parts of the program.

So, in short, we *should* hide implementation details if they're not necessary to be exposed. In this sense, JS's `class` system feels a bit too permissive in that everything defaults to being public. Class-private features are a welcomed addition to more proper software design.

Too Private? All that said, I have to throw a bit of a damper on the class-private party.

I've suggested strongly that you should only use `class` if you're going to really take advantage of most or all of what class-orientation gives you. Otherwise, you'd be better suited using other core pillar features of JS for organizing code, such as with the closure pattern.

One of the most important aspects of class-orientation is subclass inheritance, as we've seen illustrated numerous times so far in this chapter. Guess what happens to a private member/method in a base class, when it's extended by a subclass?

[1] "Principle of Least Privilege", Wikipedia; https://en.wikipedia.org/wiki/Principle_of_least_privilege ; Accessed July 2022

Private members/methods are private **only to the class they're defined in**, and are **not** inherited in any way by a subclass. Uh oh.

That might not seem like too big of a concern, until you start working with `class` and private members/methods in real software. You might quickly run up against a situation where you need to access a private method, or more often even, just a private member, from the subclass, so that the subclass can extend/augment the behavior of the base class as desired. And you might scream in frustration pretty quickly once you realize this is not possible.

What comes next is inevitably an awkward decision: do you just go back to making it public, so the subclass can access it? Ugh. Or, worse, do you try to re-design the base class to contort the design of its members/methods, such that the lack of access is partially worked around. That often involves exhausting over-parameterization (with privates as default parameter values) of methods, and other such tricks. Double ugh.

There's not a particularly great answer here, to be honest. If you have experience with class-orientation in more traditional class languages like Java or C++, you're probably dubious as to why we don't have *protected* visibility in between *public* and *private*. That's exactly what *protected* is for: keeping something private to a class AND any of its subclasses. Those languages also have *friend* features, but that's beyond the scope of our discussion here.

Sadly, not only does JS not have *protected* visibility, it seems (even as useful as it is!) to be unlikely as a JS feature. It's been discussed in great detail for over a decade (before ES6 was even a thing), and there've been multiple proposals for it.

I shouldn't say it will *never* happen, because that's not solid ground to stake on in any software. But it's very unlikely, because it actually betrays the very pillar that `class` is built on. If you are curious, or (more likely) certain that there's just *got to be a way*, I'll cover the incompatibility of *protected* visibility within JS's mechanisms in an appendix.

The point here is, as of now, JS has no *protected* visibility, and it won't any time soon. And *protected* visibility is actually, in practice, way more useful than *private* visibility.

So we return to the question: **Why should you care to make any `class` contents private?**

If I'm being honest: maybe you shouldn't. Or maybe you should. That's up to you. Just go into it aware of the stumbling blocks.

Private Members/Methods

You're excited to finally see the syntax for magical *private* visibility, right? Please don't shoot the messenger if you feel angered or sad at what you're about to see.

```
class Point2d {
    // statics
    static samePoint(point1,point2) {
        return point1.#ID === point2.#ID;
    }

    // privates
    #ID = null
    #assignID() {
        this.#ID = Math.round(Math.random() * 1e9);
    }

    // publics
    x
    y
    constructor(x,y) {
        this.#assignID();
        this.x = x;
        this.y = y;
    }
}

var one = new Point2d(3,4);
var two = new Point2d(3,4);

Point2d.samePoint(one,two);          // false
Point2d.samePoint(one,one);          // true
```

No, JS didn't do the sensible thing and introduce a `private` keyword like they did with `static`. Instead, they introduced the `#`. (insert lame joke about social-media millienials loving hashtags, or something)

TIP:

And yes, there's a million and one discussions about why not. I could spend chapters recounting the whole history, but honestly I just don't care to. I think this syntax is ugly, and many others do, too. And some love it! If you're in the latter camp, though I rarely do something like this, I'm just going to say: **just accept it**. It's too late for any more debate or pleading.

The `#whatever` syntax (including `this.#whatever` form) is only valid inside `class` bodies. It will throw syntax errors if used outside of a `class`.

Unlike public fields/instance members, private fields/instance members *must* be declared in the `class` body. You cannot add a private member to a class declaration dynamically while in the constructor method; `this.#whatever =`

`..` type assignments only work if the `#whatever` private field is declared in the class body. Moreover, though private fields can be re-assigned, they cannot be `deleted` from an instance, the way a public field/class member can.

Subclassing + Privates I warned earlier that subclassing with classes that have private members/methods can be a limiting trap. But that doesn't mean they cannot be used together.

Because "inheritance" in JS is sharing (through the `[[Prototype]]` chain), if you invoke an inherited method in a subclass, and that inherited method in turn accesses/invokes privates in its host (base) class, this works fine:

```
class Point2d { /* .. */ }

class Point3d extends Point2d {
    z
    constructor(x,y,z) {
        super(x,y);
        this.z = z;
    }
}

var one = new Point3d(3,4,5);
```

The `super(x,y)` call in this constructor invokes the inherited base class constructor (`Point2d(..)`), which itself accesses `Point2d`'s private method `#assignID()` (see the earlier snippet). No exception is thrown, even though `Point3d` cannot directly see or access the `#ID` / `#assignID()` privates that are indeed stored on the instance (named `one` here).

In fact, even the inherited `static samePoint(..)` function will work from either `Point3d` or `Point2d`:

```
Point2d.samePoint(one,one);          // true
Point3d.samePoint(one,one);          // true
```

Actually, that shouldn't be that surprising, since:

```
Point2d.samePoint === Point3d.samePoint;
```

The inherited function reference is *the exact same function* as the base function reference; it's not some copy of the function. Because the function in question has no `this` reference in it, no matter from where it's invoked, it should produce the same outcome.

It's still a shame though that `Point3d` has no way to access/influence, or indeed even knowledge of, the `#ID` / `#assignID()` privates from `Point2d`:

```
class Point2d { /* .. */ }
```

```
class Point3d extends Point2d {
    z
    constructor(x,y,z) {
        super(x,y);
        this.z = z;

        console.log(this.#ID);      // will throw!
    }
}
```

WARNING:

Notice that this snippet throws an early static syntax error at the time of defining the `Point3d` class, before ever even getting a chance to create an instance of the class. The same exception would be thrown if the reference was `super.#ID` instead of `this.#ID`.

Existence Check Keep in mind that only the `class` itself knows about, and can therefore check for, such a private field/method.

You may want to check to see if a private field/method exists on an object instance. For example (as shown below), you may have a static function or method in a class, which receives an external object reference passed in. To check to see if the passed-in object reference is of this same class (and therefore has the same private members/methods in it), you basically need to do a "brand check" against the object.

Such a check could be rather convoluted, because if you access a private field that doesn't already exist on the object, you get a JS exception thrown, requiring ugly `try..catch` logic.

But there's a cleaner approach, so called an "ergonomic brand check", using the `in` keyword:

```
class Point2d {
    // statics
    static samePoint(point1,point2) {
        // "ergonomic brand checks"
        if (#ID in point1 && #ID in point2) {
            return point1.#ID === point2.#ID;
        }
        return false;
    }

    // privates
    #ID = null
    #assignID() {
```

```
        this.#ID = Math.round(Math.random() * 1e9);
    }

    // publics
    x
    y
    constructor(x,y) {
        this.#assignID();
        this.x = x;
        this.y = y;
    }
}

var one = new Point2d(3,4);
var two = new Point2d(3,4);

Point2d.samePoint(one,two);          // false
Point2d.samePoint(one,one);          // true
```

The `#privateField in someObject` check will not throw an exception if the field isn't found, so it's safe to use without `try..catch` and use its simple boolean result.

Exfiltration Even though a member/method may be declared with *private* visibility, it can still be exfiltrated (extracted) from a class instance:

```
var id, func;

class Point2d {
    // privates
    #ID = null
    #assignID() {
        this.#ID = Math.round(Math.random() * 1e9);
    }

    // publics
    x
    y
    constructor(x,y) {
        this.#assignID();
        this.x = x;
        this.y = y;

        // exfiltration
        id = this.#ID;
        func = this.#assignID;
```

```
    }
}

var point = new Point2d(3,4);

id;                     // 7392851012 (...for example)

func;                   // function #assignID() { .. }
func.call(point,42);

func.call({},100);
// TypeError: Cannot write private member #ID to an
// object whose class did not declare it
```

The main concern here is to be careful when passing private methods as callbacks (or in any way exposing privates to other parts of the program). There's nothing stopping you from doing so, which can create a bit of an unintended privacy disclosure.

Private Statics

Static properties and functions can also use `#` to be marked as private:

```
class Point2d {
    static #errorMsg = "Out of bounds."
    static #printError() {
        console.log(`Error: ${this.#errorMsg}`);
    }

    // publics
    x
    y
    constructor(x,y) {
        if (x > 100 || y > 100) {
            Point2d.#printError();
        }
        this.x = x;
        this.y = y;
    }
}

var one = new Point2d(30,400);
// Error: Out of bounds.
```

The `#printError()` static private function here has a `this`, but that's referencing the `Point2d` class, not an instance. As such, the `#errorMsg` and `#printError()` are independent of instances and thus are best as statics. More-

over, there's no reason for them to be accessible outside the class, so they're marked private.

Remember: private statics are similarly not-inherited by subclasses just as private members/methods are not.

Gotcha: Subclassing With Static Privates and `this` Recall that inherited methods, invoked from a subclass, have no trouble accessing (via `this.#whatever` style references) any privates from their own base class:

```
class Point2d {
    // ..

    getID() {
        return this.#ID;
    }

    // ..
}

class Point3d extends Point2d {
    // ..

    printID() {
        console.log(`ID: ${this.getID()}`);
    }
}

var point = new Point3d(3,4,5);
point.printID();
// ID: ..
```

That works just fine.

Unfortunately, and (to me) quite unexpectedly/inconsistently, the same is not true of private statics accessed from inherited public static functions:

```
class Point2d {
    static #errorMsg = "Out of bounds."
    static printError() {
        console.log(`Error: ${this.#errorMsg}`);
    }

    // ..
}

class Point3d extends Point2d {
    // ..
```

```
}

Point2d.printError();
// Error: Out of bounds.

Point3d.printError === Point2d.printError;
// true

Point3d.printError();
// TypeError: Cannot read private member #errorMsg
// from an object whose class did not declare it
```

The `printError()` static is inherited (shared via `[[Prototype]]`) from `Point2d` to `Point3d` just fine, which is why the function references are identical. Like the non-static snippet just above, you might have expected the `Point3d.printError()` static invocation to resolve via the `[[Prototype]]` chain to its original base class (`Point2d`) location, thereby letting it access the base class's `#errorMsg` static private.

But it fails, as shown by the last statement in that snippet. The reason it fails here, but not with the previous snippet, is a convoluted brain twister. I'm not going to dig into the *why* explanation here, frankly because it boils my blood to do so.

There's a *fix*, though. In the static function, instead of `this.#errorMsg`, swap that for `Point2d.#errorMsg`, and now it works:

```
class Point2d {
    static #errorMsg = "Out of bounds."
    static printError() {
        // the fixed reference vvvvvv
        console.log(`Error: ${Point2d.#errorMsg}`);
    }

    // ..
}

class Point3d extends Point2d {
    // ..
}

Point2d.printError();
// Error: Out of bounds.

Point3d.printError();
// Error: Out of bounds.  <-- phew, it works now!
```

If public static functions are being inherited, use the class name to access any

private statics instead of using `this.` references. Beware that gotcha!

Class Example

OK, we've laid out a bunch of disparate class features. I want to wrap up this chapter by trying to illustrate a sampling of these capabilities in a single example that's a little less basic/contrived.

```
class CalendarItem {
    static #UNSET = Symbol("unset")
    static #isUnset(v) {
        return v === this.#UNSET;
    }
    static #error(num) {
        return this[`ERROR_${num}`];
    }
    static {
        for (let [idx,msg] of [
            "ID is already set.",
            "ID is unset.",
            "Don't instantiate 'CalendarItem' directly.",
        ].entries()) {
            this[`ERROR_${(idx+1)*100}`] = msg;
        }
    }
    static isSameItem(item1,item2) {
        if (#ID in item1 && #ID in item2) {
            return item1.#ID === item2.#ID;
        }
        else {
            return false;
        }
    }

    #ID = CalendarItem.#UNSET
    #setID(id) {
        if (CalendarItem.#isUnset(this.#ID)) {
            this.#ID = id;
        }
        else {
            throw new Error(CalendarItem.#error(100));
        }
    }

    description = null
    startDateTime = null
```

```
    constructor() {
        if (new.target !== CalendarItem) {
            let id = Math.round(Math.random() * 1e9);
            this.#setID(id);
        }
        else {
            throw new Error(CalendarItem.#error(300));
        }
    }
    getID() {
        if (!CalendarItem.#isUnset(this.#ID)) {
            return this.#ID;
        }
        else {
            throw new Error(CalendarItem.#error(200));
        }
    }
    getDateTimeStr() {
        if (this.startDateTime instanceof Date) {
            return this.startDateTime.toUTCString();
        }
    }
    summary() {
        console.log(`(${
            this.getID()
        }) ${
            this.description
        } at ${
            this.getDateTimeStr()
        }`);
    }
}

class Reminder extends CalendarItem {
    #complete = false;  // <-- no ASI, semicolon needed

    [Symbol.toStringTag] = "Reminder"
    constructor(description,startDateTime) {
        super();

        this.description = description;
        this.startDateTime = startDateTime;
    }
    isComplete() {
        return !!this.#complete;
```

```
    }
    markComplete() {
        this.#complete = true;
    }
    summary() {
        if (this.isComplete()) {
            console.log(`(${this.getID()}) Complete.`);
        }
        else {
            super.summary();
        }
    }
}

class Meeting extends CalendarItem {
    #getEndDateTimeStr() {
        if (this.endDateTime instanceof Date) {
            return this.endDateTime.toUTCString();
        }
    }

    endDateTime = null;  // <-- no ASI, semicolon needed

    [Symbol.toStringTag] = "Meeting"
    constructor(description,startDateTime,endDateTime) {
        super();

        this.description = description;
        this.startDateTime = startDateTime;
        this.endDateTime = endDateTime;
    }
    getDateTimeStr() {
        return `${
            super.getDateTimeStr()
        } - ${
            this.#getEndDateTimeStr()
        }`;
    }
}
```

Take some time to read and digest those `class` definitions. Did you spot most of the `class` features we talked about in this chapter?

NOTE:

One question you may have: why didn't I move the repeated logic of `description` and `startDateTime` setting from both subclass constructors into the single base constructor? This is a nuanced point, but it's not my intention that `CalendarItem` ever be directly instantiated; it's what in class-oriented terms we refer to as an "abstract class". That's why I'm using `new.target` to throw an error if the `CalendarItem` class is ever directly instantiated! So I don't want to imply by signature that the `CalendarItem(..)` constructor should ever be directly used.

Let's now see these three classes in use:

```
var callMyParents = new Reminder(
    "Call my parents to say hi",
    new Date("July 7, 2022 11:00:00 UTC")
);
callMyParents.toString();
// [object Reminder]
callMyParents.summary();
// (586380912) Call my parents to say hi at
// Thu, 07 Jul 2022 11:00:00 GMT
callMyParents.markComplete();
callMyParents.summary();
// (586380912) Complete.
callMyParents instanceof Reminder;
// true
callMyParents instanceof CalendarItem;
// true
callMyParents instanceof Meeting;
// false


var interview = new Meeting(
    "Job Interview: ABC Tech",
    new Date("June 23, 2022 08:30:00 UTC"),
    new Date("June 23, 2022 09:15:00 UTC")
);
interview.toString();
// [object Meeting]
interview.summary();
// (994337604) Job Interview: ABC Tech at Thu,
// 23 Jun 2022 08:30:00 GMT - Thu, 23 Jun 2022
// 09:15:00 GMT
interview instanceof Meeting;
// true
```

```
interview instanceof CalendarItem;
// true
interview instanceof Reminder;
// false


Reminder.isSameItem(callMyParents,callMyParents);
// true
Meeting.isSameItem(callMyParents,interview);
// false
```

Admittedly, some bits of this example are a little contrived. But honestly, I think pretty much all of this is plausible and reasonable usages of the various `class` features.

By the way, there's probably a million different ways to structure the above code logic. I'm by no means claiming this is the *right* or *best* way to do so. As an exercise for the reader, try your hand and writing it yourself, and take note of things you did differently than my approach.

You Don't Know JS Yet: Objects & Classes - 2nd Edition

Chapter 4: This Works

NOTE:
Work in progress

We've seen the `this` keyword used quite a bit so far, but haven't really dug in to understand exactly how it works in JS. It's time we do so.

But to properly understand `this` in JS, you need to set aside any preconceptions you may have, especially assumptions from how `this` works in other programming languages you may have experience in.

Here's the most important thing to understand about `this`: the determination of what value (usually, object) `this` points at is not made at author time, but rather determined at runtime. That means you cannot simply look at a `this`-aware function (even a method in a `class` definition) and know for sure what `this` will hold while that function runs.

Instead, you have to find each place the function is invoked, and look at *how* it's invoked (not even *where* matters). That's the only way to fully answer what `this` will point to.

In fact, a single `this`-aware function can be invoked at least four different ways, and any of those approaches will end up assigning a different `this` for that particular function invocation.

So the typical question we might ask when reading code – "What does `this` point to the function?" – is not actually a valid question. The question you really have to ask is, "When the function is invoked a certain way, what `this` will be assigned for that invocation?"

If your brain is already twisting around just reading this chapter intro… good! Settle in for a rewiring of how you think about `this` in JS.

This Aware

I used the phrase `this`-aware just a moment ago. But what exactly do I mean by that?

Any function that has a `this` keyword in it.

If a function does not have `this` in it anywhere, then the rules of how `this` behaves don't affect that function in any way. But if it *does* have even a single `this` in it, then you absolutely cannot determine how the function will behave

without figuring out, for each invocation of the function, what `this` will point to.

It's sort of like the `this` keyword is a placeholder in a template. That placeholder's value-replacement doesn't get determined when we author the code; it gets determined while the code is running.

You might think I'm just playing word games here. Of course, when you write the program, you write out all the calls to each function, so you've already determined what the `this` is going to be when you authored the code, right? Right!?

Not so fast!

First of all, you don't always write all the code that invokes your function(s). Your `this`-aware function(s) might be passed as a callback(s) to some other code, either in your code base, or in a third-party framework/utility, or even inside a native built-in mechanism of the language or environment that's hosting your program.

But even aside from passing functions as callbacks, several mechanisms in JS allow for conditional runtime behaviors to determine which value (again, usually object) will be set for the `this` of a particular function invocation. So even though you may have written all that code, you *at best* will have to mentally execute the different conditions/paths that end up affecting the function invocation.

And why does all this matter?

Because it's not just you, the author of the code, that needs to figure this stuff out. It's *every single reader* of your code, forever. If anyone (even your future self) wants to read a piece of code that defines a `this`-aware function, that inevitably means that, to fully understand and predict its behavior, that person will have to find, read, and understand every single invocation of that function.

This Confuses Me

Now, in fairness, that's already partially true if we consider a function's parameters. To understand how a function is going to work, we need to know what is being passed into it. So any function with at least one parameter is, in a similar sense, *argument*-aware – meaning, what argument(s) is/are passed in and assigned to the parameter(s) of the function.

But with parameters, we often have a bit more of a hint from the function itself what the parameters will do and hold.

We often see the names of the parameters declared right in the function header, which goes a long way to explaining their nature/purpose. And if there are defaults for the parameters, we often see them declared inline with `= whatever` clauses. Moreover, depending on the code style of the author, we may see

in the first several lines of the function a set of logic that applies to these parameters; this could be assertions about the values (disallowed values, etc), or even modifications (type conversion, formatting, etc).

Actually, `this` is very much like a parameter to a function, but it's an implicit parameter rather than an explicit one. You don't see any signal that `this` is going to be used, in the function header anywhere. You have to read the entire function body to see if `this` appears anywhere.

The "parameter" name is always `this`, so we don't get much of a hint as to its nature/purpose from such a general name. In fact, there's historically a lot of confusion of what "this" even is supposed to mean. And we rarely see much if anything done to validate/convert/etc the `this` value applied to a function invocation. In fact, virtually all `this`-aware code I've seen just neatly assumes the `this` "parameter" is holding exactly what value is expected. Talk about **a trap for unexpected bugs!**

So What Is This?

If `this` is an implicit parameter, what's its purpose? What's being passed in?

Hopefully you have already read the "Scope & Closures" book of this series. If not, I strongly encourage you to circle back and read that one once you've finished this one. In that book, I explained at length how scopes (and closures!) work, an especially important characteristic of functions.

Lexical scope (including all the variables closed over) represents a *static* context for the function's lexical identifier references to be evaluated against. It's fixed/static because at author time, when you place functions and variable declarations in various (nested) scopes, those decisions are fixed, and unaffected by any runtime conditions.

By contrast, a different programming language might offer *dynamic* scope, where the context for a function's variable references is not determined by author-time decisions but by runtime conditions. Such a system would be undoubtedly more flexible than static context – though with flexibility often comes complexity.

To be clear: JS scope is always and only lexical and *static* (if we ignore non-strict mode cheats like `eval(..)` and `with`). However, one of the truly powerful things about JS is that it offers another mechanism with similar flexibility and capabilities to *dynamic* scope.

The `this` mechanism is, effectively, *dynamic* context (not scope); it's how a `this`-aware function can be dynamically invoked against different contexts – something that's impossible with closure and lexical scope identifiers!

Why Is This So Implicit?

You might wonder why something as important as a *dynamic* context is handled as an implicit input to a function, rather than being an explicit argument passed

in.

That's a very important question, but it's not one we can quite answer, yet. Hold onto that question though.

Can We Get On With This?

So why have I belabored *this* subject for a couple of pages now? You get it, right!? You're ready to move on.

My point is, you the author of code, and all other readers of the code even years or decades in the future, need to be `this`-aware. That's the choice, the burden, you place on the reading of such code. And yes, that goes for the choice to use `class` (see Chapter 3), as most class methods will be `this`-aware out of necessity.

Be aware of *this* `this` choice in code you write. Do it intentionally, and do it in such a way as to produce more outcome benefit than burden. Make sure `this` usage in your code *carries its own weight*.

Let me put it *this* way: don't use `this`-aware code unless you really can justify it, and you've carefully weighed the costs. Just because you've seen a lot of code examples slinging around `this` in others' code, doesn't mean that `this` belongs in *this* code you're writing.

The `this` mechanism in JS, paired with `[[Prototype]]` delegation, is an extremely powerful pillar of the language. But as the cliche goes: "with great power comes great responsibility". Anecdotally, even though I really like and appreciate *this* pillar of JS, probably less than 5% of the JS code I ever write uses it. And when I do, it's with restraint. It's not my default, go-to JS capability.

This Is It!

OK, enough of the wordy lecture. You're ready to dive into `this` code, right?

Let's revisit (and extend) `Point2d` from Chapter 3, but just as an object with data properties and functions on it, instead of using `class`:

```
var point = {
    x: null,
    y: null,

    init(x,y) {
        this.x = x;
        this.y = y;
    },
    rotate(angleRadians) {
        var rotatedX = this.x * Math.cos(angleRadians) -
            this.y * Math.sin(angleRadians);
```

```
        var rotatedY = this.x * Math.sin(angleRadians) +
            this.y * Math.cos(angleRadians);
        this.x = rotatedX;
        this.y = rotatedY;
    },
    toString() {
        return `(${this.x},${this.y})`;
    },
};
```

As you can see, the `init(..)`, `rotate(..)`, and `toString()` functions are `this`-aware. You might be in the habit of assuming that the `this` reference will obviously always hold the `point` object. But that's not guaranteed in any way.

Keep reminding yourself as you go through the rest of this chapter: the `this` value for a function is determined by *how* the function is invoked. That means you can't look at the function's definition, nor where the function is defined (not even the enclosing `class`!). In fact, it doesn't even matter where the function is called from.

We only need to look at *how* the functions are called; that's the only factor that matters.

Implicit Context Invocation

Consider this call:

```
point.init(3,4);
```

We're invoking the `init(..)` function, but notice the `point.` in front of it? This is an *implicit context* binding. It says to JS: invoke the `init(..)` function with `this` referencing `point`.

That is the *normal* way we'd expect a `this` to work, and that's also one of the most common ways we invoke functions. So the typical invocation gives us the intuitive outcome. That's a good thing!

Default Context Invocation

But what happens if we do this?

```
const init = point.init;
init(3,4);
```

You might assume that we'd get the same outcome as the previous snippet. But that's not how JS `this` assignment works.

The *call-site* for the function is `init(3,4)`, which is different than `point.init(3,4)`. When there's no *implicit context* (`point.`), nor any other kind of `this` assignment mechanism, the *default context* assignment occurs.

What will `this` reference when `init(3,4)` is invoked like that?

It depends.

Uh oh. Depends? That sounds confusing.

Don't worry, it's not as bad as it sounds. The *default context* assignment depends on whether the code is in strict-mode or not. But thankfully, virtually all JS code these days is running in strict-mode; for example, ESM (ES Modules) always run in strict-mode, as does code inside a `class` block. And virtually all transpiled JS code (via Babel, TypeScript, etc) is written to declare strict-mode.

So almost all of the time, modern JS code will be running in strict-mode, and thus the *default assignment* context won't "depend" on anything; it's pretty straightforward: `undefined`. That's it!

NOTE:

Keep in mind: `undefined` does not mean "not defined"; it means, "defined with the special empty `undefined` value". I know, I know… the name and meaning are mismatched. That's language legacy baggage, for you. (shrugging shoulders)

That means `init(3,4)`, if run in strict-mode, would throw an exception. Why? Because the `this.x` reference in `init(..)` is a `.x` property access on `undefined` (i.e., `undefined.x`), which is not allowed:

```
"use strict";

var point = { /* .. */ };

const init = point.init;
init(3,4);
// TypeError: Cannot set properties of
// undefined (setting 'x')
```

Stop for a moment and consider: why would JS choose to default the context to `undefined`, so that any *default context* invocation of a `this`-aware function will fail with such an exception?

Because a `this`-aware function **always needs a `this`**. The invocation `init(3,4)` isn't providing a `this`, so that *is* a mistake, and *should* raise an exception so the mistake can be corrected. The lesson: never invoke a `this`-aware function without providing it a `this`!

Just for completeness sake: in the less common non-strict mode, the *default context* is the global object – JS defines it as `globalThis`, which in browser JS is essentially an alias to `window`, and in Node it's `global`. So, when `init(3,4)` runs in non-strict mode, the `this.x` expression is `globalThis.x` – also known

as `window.x` in the browser, or `global.x` in Node. Thus, `globalThis.x` gets set as `3` and `globalThis.y` gets set as `4`.

```
// no strict-mode here, beware!

var point = { /* .. */ };

const init = point.init;
init(3,4);

globalThis.x;   // 3
globalThis.y;   // 4
point.x;        // null
point.y;        // null
```

That's unfortunate, because it's almost certainly *not* the intended outcome. Not only is it bad if it's a global variable, but it's also *not* changing the property on our `point` object, so program bugs are guaranteed.

WARNING:

Ouch! Nobody wants accidental global variables implicitly created from all over the code. The lesson: always make sure your code is running in strict-mode!

Explicit Context Invocation

Functions can alternately be invoked with *explicit context*, using the built-in `call(..)` or `apply(..)` utilities:

```
var point = { /* .. */ };

const init = point.init;

init.call( point, 3, 4 );
// or: init.apply( point, [ 3, 4 ] )

point.x;        // 3
point.y;        // 4
```

`init.call(point,3,4)` is effectively the same as `point.init(3,4)`, in that both of them assign `point` as the `this` context for the `init(..)` invocation.

NOTE:

Both `call(..)` and `apply(..)` utilities take as their first argument a `this` context value; that's almost always an object, but can technically can be any value (number, string, etc). The `call(..)` utility takes subsequent arguments and passes them through to the invoked function, whereas `apply(..)` expects its second argument to be an array of values to pass as arguments.

It might seem awkward to contemplate invoking a function with the *explicit context* assignment (`call(..)` / `apply(..)`) style in your program. But it's more useful than might be obvious at first glance.

Let's recall the original snippet:

```
var point = {
    x: null,
    y: null,

    init(x,y) {
        this.x = x;
        this.y = y;
    },
    rotate(angleRadians) { /* .. */ },
    toString() {
        return `(${this.x},${this.y})`;
    },
};

point.init(3,4);

var anotherPoint = {};
point.init.call( anotherPoint, 5, 6 );

point.x;                // 3
point.y;                // 4
anotherPoint.x;         // 5
anotherPoint.y;         // 6
```

Are you seeing what I did there?

I wanted to define `anotherPoint`, but I didn't want to repeat the definitions of those `init(..)` / `rotate(..)` / `toString()` functions from `point`. So I "borrowed" a function reference, `point.init`, and explicitly set the empty object `anotherPoint` as the `this` context, via `call(..)`.

When `init(..)` is running at that moment, `this` inside it will reference `anotherPoint`, and that's why the `x` / `y` properties (values `5` / `6`, respectively) get set there.

Any `this`-aware functions can be borrowed like this: `point.rotate.call(anotherPoint, ..)`, `point.toString.call(anotherPoint)`.

Revisiting Implicit Context Invocation Another approach to share behavior between `point` and `anotherPoint` would have been:

```
var point = { /* .. */ };

var anotherPoint = {
    init: point.init,
    rotate: point.rotate,
    toString: point.toString,
};

anotherPoint.init(5,6);

anotherPoint.x;          // 5
anotherPoint.y;          // 6
```

This is another way of "borrowing" the functions, by adding shared references to the functions on any target object (e.g., `anotherPoint`). The call-site invocation `anotherPoint.init(5,6)` is the more natural/ergonomic style that relies on *implicit context* assignment.

It may seem this approach is a little cleaner, comparing `anotherPoint.init(5,6)` to `point.init.call(anotherPoint,5,6)`.

But the main downside is having to modify any target object with such shared function references, which can be verbose, manual, and error-prone. Sometimes such an approach is acceptable, but many other times, *explicit context* assignment with `call(..)` / `apply(..)` is more preferable.

New Context Invocation

We've so far seen three different ways of context assignment at the function call-site: *default*, *implicit*, and *explicit*.

A fourth way to call a function, and assign the `this` for that invocation, is with the `new` keyword:

```
var point = {
    // ..

    init: function() { /* .. */ }

    // ..
};

var anotherPoint = new point.init(3,4);
```

```
anotherPoint.x;     // 3
anotherPoint.y;     // 4
```

TIP:

This example has a bit of nuance to be explained. The `init: function() { .. }` form shown here – specifically, a function expression assigned to a property – is required for the function to be validly called with the `new` keyword. From previous snippets, the concise method form of `init() { .. }` defines a function that *cannot* be called with `new`.

You've typically seen `new` used with `class` for creating instances. But as an underlying mechanism of the JS language, `new` is not inherently a `class` operation.

In a sense, the `new` keyword hijacks a function and forces its behavior into a different mode than a normal invocation. Here are the 4 special steps that JS performs when a function is invoked with `new`:

1. create a brand new empty object, out of thin air.
2. link the `[[Prototype]]` of that new empty object to the function's `.prototype` object (see Chapter 2).
3. invoke the function with the `this` context set to that new empty object.
4. if the function doesn't return its own object value explicitly (with a `return .. ` statement), assume the function call should instead return the new object (from steps 1-3).

WARNING:

Step 4 implies that if you `new` invoke a function that *does* return its own object – like `return { .. }`, etc – then the new object from steps 1-3 is *not* returned. That's a tricky gotcha to be aware of, in that it effectively discards that new object before the program has a chance to receive and store a reference to it. Essentially, `new` should never be used to invoke a function that has explicit `return .. ` statement(s) in it.

To understand these 4 `new` steps more concretely, I'm going to illustrate them in code, as an alternate to using the `new` keyword:

```
// alternative to:
//   var anotherPoint = new point.init(3,4)

var anotherPoint;
// this is a bare block to hide local
```

```
// `let` declarations
{
    // (Step 1)
    let tmpObj = {};

    // (Step 2)
    Object.setPrototypeOf(
        tmpObj, point.init.prototype
    );
    // or: tmpObj.__proto__ = point.init.prototype

    // (Step 3)
    let res = point.init.call(tmpObj,3,4);

    // (Step 4)
    anotherPoint = (
        typeof res !== "object" ? tmpObj : res
    );
}
```

Clearly, the `new` invocation streamlines that set of manual steps!

TIP:

The `Object.setPrototypeOf(..)` in step 2 could also have been done via the `__proto__` property, such as `tmpObj.__proto__ = point.init.prototype`, or even as part of the object literal (step 1) with `tmpObj = { __proto__: point.init.prototype }`.

Skipping some of the formality of these steps, let's recall an earlier snippet and see how `new` approximates a similar outcome:

```
var point = { /* .. */ };

// this approach:
var anotherPoint = {};
point.init.call(anotherPoint,5,6);

// can instead be approximated as:
var yetAnotherPoint = new point.init(5,6);
```

That's a bit nicer! But there's a caveat here.

Using the other functions that `point` holds against `anotherPoint` / `yetAnotherPoint`, we won't want to do with `new`. Why? Because `new` is creating a *new* object, but that's not what we want if we intend to invoke a function against an existing object.

Instead, we'll likely use *explicit context* assignment:

```
point.rotate.call( anotherPoint, /*angleRadians=*/Math.PI );

point.toString.call( yetAnotherPoint );
// (5,6)
```

Review This

We've seen four rules for `this` context assignment in function calls. Let's put them in order of precedence:

1. Is the function invoked with `new`, creating and setting a *new* `this`?
2. Is the function invoked with `call(..)` or `apply(..)`, *explicitly* setting `this`?
3. Is the function invoked with an object reference at the call-site (e.g., `point.init(..)`), *implicitly* setting `this`?
4. If none of the above… are we in non-strict mode? If so, *default* the `this` to `globalThis`. But if in strict-mode, *default* the `this` to `undefined`.

These rules, *in this order*, are how JS determines the `this` for a function invocation. If multiple rules match a call-site (e.g., `new point.init.call(..)`), the first rule from the list to match wins.

That's it, you're now master over the `this` keyword. Well, not quite. There's a bunch more nuance to cover. But you're well on your way!

An Arrow Points Somewhere

Everything I've asserted so far about `this` in functions, and how its determined based on the call-site, makes one giant assumption: that you're dealing with a *regular* function (or method).

So what's an *irregular* function?!? It looks like this:

```
const x = x => x <= x;
```

NOTE:

Yes, I'm being a tad sarcastic and unfair to call an arrow function "irregular" and to use such a contrived example. It's a joke, ok?

Here's a real example of an `=>` arrow function:

```
const clickHandler = evt =>
    evt.target.matches("button") ?
        this.theFormElem.submit() :
        evt.stopPropagation();
```

For comparison sake, let me also show the non-arrow equivalent:

```
const clickHandler = function(evt) {
    evt.target.matches("button") ?
        this.theFormElem.submit() :
        evt.stopPropagation();
};
```

Or if we went a bit old-school about it – this is my jam! – we could try the standalone function declaration form:

```
function clickHandler(evt) {
    evt.target.matches("button") ?
        this.theFormElem.submit() :
        evt.stopPropagation();
}
```

Or if the function appeared as a method in a `class` definition, or as a concise method in an object literal, it would look like this:

```
// ..
clickHandler(evt) {
    evt.target.matches("button") ?
        this.theFormElem.submit() :
        evt.stopPropagation();
}
```

What I really want to focus on is how each of these forms of the function will behave with respect to their `this` reference, and whether the first `=>` form differs from the others (hint: it does!). But let's start with a little quiz to see if you've been paying attention.

For each of those function forms just shown, how do we know what each `this` will reference?

Where's The Call-site?

Hopefully, you responded with something like: "first, we need to see how the functions are called."

Fair enough.

Let's say our program looks like this:

```
var infoForm = {
    theFormElem: null,
    theSubmitBtn: null,

    init() {
        this.theFormElem =
            document.getElementById("the-info-form");
```

```
        this.theSubmitBtn =
            theFormElem.querySelector("button[type=submit]");

        // is *this* the call-site?
        this.theSubmitBtn.addEventListener(
            "click",
            this.clickHandler,
            false
        );
    },

    // ..
}
```

Ah, interesting. Half of you readers have never seen actual DOM API code like `getElementById(..)`, `querySelector(..)`, and `addEventListener(..)` before. I heard the confusion bells whistle just now!

NOTE:

Sorry, I'm dating myself, here. I've been doing this stuff long enough that I remember when we did that kind of code long before we had utilities like jQuery cluttering up the code with `$` everywhere. And after many years of front-end evolution, we seem to have landed somewhere quite a bit more "modern" – at least, that's the prevailing presumption.

I'm guessing many of you these days are used to seeing component-framework code (React, etc) somewhat like this:

```
// ..

infoForm(props) {
    return (
        <form ref={this.theFormElem}>
            <button type=submit onClick=this.clickHandler>
                Click Me
            </button>
        </form>
    );
}

// ..
```

Of course, there's a bunch of other ways that code might be shaped, depending on if you're using one framework or another, etc.

Or maybe you're not even using `class` / `this` style components anymore, because you've moved everything to hooks and closures. In any case, for our

discussion purposes, *this* chapter is all about `this`, so we need to stick to a coding style like the above, to have code related to the discussion.

And neither of those two previous code snippets show the `clickHandler` function being defined. But I've said repeatedly so far, that doesn't matter; all that matters is … what? say it with me… all that matters is *how* the function is invoked.

So how is `clickHandler` being invoked? What's the call-site, and which context assignment rule does it match?

Hidden From Sight

If you're stuck, don't worry. I'm deliberately making this difficult, to point something very important out.

When the `"click"` or `onClick=` event handler bindings happen, in both cases, we specified `this.clickHandler`, which implies that there is a `this` context object with a property on it called `clickHandler`, which is holding our function definition.

So, is `this.clickHandler` the call-site? If it was, what assignment rule applies? The *implicit context* rule (#3)?

Unfortunately, no.

The problem is, **we cannot actually see the call-site** in this program. Uh oh.

If we can't see the call-site, how do we know *how* the function is going to actually get called?

That's the exact point I'm making.

It doesn't matter that we passed in `this.clickHandler`. That is merely a reference to a function object value. It's not a call-site.

Under the covers, somewhere inside a framework, library, or even the JS environment itself, when a user clicks the button, a reference to the `clickHandler(..)` function is going to be invoked. And as we've implied, that call-site is even going to pass in the DOM event object as the `evt` argument.

Since we can't see the call-site, we have to *imagine* it. Might it look like…?

```
// ..
eventCallback( domEventObj );
// ..
```

If it did, which `this` rule would apply? The *default context* rule (#4)?

Or, what if the call-site looked like this…?

```
// ..
eventCallback.call( domElement, domEventObj );
```

Now which `this` rule would apply? The *explicit context* rule (#2)?

Unless you open and view the source code for the framework/library, or read the documentation/specification, you won't *know* what to expect of that call-site. Which means that predicting, ultimately, what `this` points to in the `clickHandler` function you write, is… to put it mildly… a bit convoluted.

This Is Wrong

To spare you any more pain here, I'll cut to the chase.

Pretty much all implementations of a click-handler mechanism are going to do something like the `.call(..)`, and they're going to set the DOM element (e.g., button) the event listener is bound to, as the *explicit context* for the invocation.

Hmmm… is that ok, or is that going to be a problem?

Recall that our `clickHandler(..)` function is `this`-aware, and that its `this.theFormElem` reference implies referencing an object with a `theFormElem` property, which in turn is pointing at the parent `<form>` element. DOM buttons do not, by default, have a `theFormElem` property on them.

In other words, the `this` reference that our event handler will have set for it is almost certainly wrong. Oops.

Unless we want to rewrite the `clickHandler` function, we're going to need to fix that.

Fixing `this`

Let's consider some options to address the mis-assignment. To keep things focused, I'll stick to this style of event binding for the discussion:

```
this.submitBtnaddEventListener(
    "click",
    this.clickHandler,
    false
);
```

Here's one way to address it:

```
// store a fixed reference to the current
// `this` context
var context = this;

this.submitBtn.addEventListener(
    "click",
    function handler(evt){
```

```
        return context.clickHandler(evt);
    },
    false
);
```

TIP:

Most older JS code that uses this approach will say something like `var self = this` instead of the `context` name I'm giving it here. "Self" is a shorter word, and sounds cooler. But it's also entirely the wrong semantic meaning. The `this` keyword is not a "self" reference to the function, but rather the context for that current function invocation. Those may seem like the same thing at a glance, but they're completely different concepts, as different as apples and a Beatles song. So… to paraphrase them, "Hey developer, don't make it bad. Take a sad `self` and make it better `context`."

What's going on here? I recognized that the enclosing code, where the `addEventListener` call is going to run, has a current `this` context that is correct, and we need to ensure that same `this` context is applied when `clickHandler(..)` gets invoked.

I defined a surrounding function (`handler(..)`) and then forced the call-site to look like:

```
context.clickHandler(evt);
```

TIP:

Which `this` context assignment rule is applied here? That's right, the *implicit context* rule (#3).

Now, it doesn't matter what the internal call-site of the library/framework/environment looks like. But, why?

Because we're now *actually* in control of the call-site. It doesn't matter how `handler(..)` gets invoked, or what its `this` is assigned. It only matters than when `clickHandler(..)` is invoked, the `this` context is set to what we wanted.

I pulled off that trick not only by defining a surrounding function (`handler(..)`) so I can control the call-site, but… and this is important, so don't miss it… I defined `handler(..)` as a NON-`this`-aware function! There's no `this` keyword inside of `handler(..)`, so whatever `this` gets set (or not) by the library/framework/environment, is completely irrelevant.

The `var context = this` line is critical to the trick. It defines a lexical variable `context`, which is not some special keyword, holding a snapshot of the value in the outer `this`. Then inside `clickHandler`, we merely reference a lexical variable (`context`), no relative/magic `this` keyword.

Lexical This

The name for this pattern, by the way, is "lexical this", meaning a `this` that behaves like a lexical scope variable instead of like a dynamic context binding.

But it turns out JS has an easier way of performing the "lexical this" magic trick. Are you ready for the trick reveal!?

…

The `=>` arrow function! Tada!

That's right, the `=>` function is, unlike all other function forms, special, in that it's not special at all. Or, rather, that it doesn't define anything special for `this` behavior whatsoever.

In an `=>` function, the `this` keyword… **is not a keyword**. It's absolutely no different from any other variable, like `context` or `happyFace` or `foobarbaz`.

Let me illustrate *this* point more directly:

```
function outer() {
    console.log(this.value);

    // define a return an "inner"
    // function
    var inner = () => {
        console.log(this.value);
    };

    return inner;
}

var one = {
    value: 42,
};
var two = {
    value: "sad face",
};

var innerFn = outer.call(one);
// 42

innerFn.call(two);
// 42   <-- not "sad face"
```

The `innerFn.call(two)` would, for any *regular* function definition, have resulted in `"sad face"` here. But since the `inner` function we defined and returned (and assigned to `innerFn`) was an *irregular* `=>` arrow function, it has no special `this` behavior, but instead has "lexical this" behavior.

When the `innerFn(..)` (aka `inner(..)`) function is invoked, even with an *explicit context* assignment via `.call(..)`, that assignment is ignored.

NOTE:

I'm not sure why `=>` arrow functions even have a `call(..)` / `apply(..)` on them, since they are silent no-op functions. I guess it's for consistency with normal functions. But as we'll see later, there are other inconsistencies between *regular* functions and *irregular* `=>` arrow functions.

When a `this` is encountered (`this.value`) inside an `=>` arrow function, `this` is treated like a normal lexical variable, not a special keyword. And since there is no `this` variable in that function itself, JS does what it always does with lexical variables: it goes up one level of lexical scope – in this case, to the surrounding `outer(..)` function, and it checks to see if there's any registered `this` in that scope.

Luckily, `outer(..)` is a *regular* function, which means it has a normal `this` keyword. And the `outer.call(one)` invocation assigned `one` to its `this`.

So, `innerFn.call(two)` is invoking `inner()`, but when `inner()` looks up a value for `this`, it gets… `one`, not `two`.

Back To The... Button You thought I was going to make a pun joke and say "future" there, didn't you!?

A more direct and appropriate way of solving our earlier issue, where we had done `var context = this` to get a sort of faked "lexical this" behavior, is to use the `=>` arrow function, since its primary design feature is… "lexical this".

```
this.submitBtn.addEventListener(
    "click",
    evt => this.clickHandler(evt),
    false
);
```

Boom! Problem solved! Mic drop!

Hear me on *this*: the `=>` arrow function is *not* – I repeat, *not* – about typing fewer characters. The primary point of the `=>` function being added to JS was to give us "lexical this" behavior without having to resort to `var context = this` (or worse, `var self = this`) style hacks.

TIP:

If you need "lexical this", always prefer an `=>` arrow function. If you don't need "lexical this", well… the `=>` arrow function might not be the best tool for the job.

Confession Time I've said all along in this chapter, that how you write a function, and where you write the function, has *nothing* to do with how its `this` will be assigned.

For regular functions, that's true. But when we consider an irregular `=>` arrow function, it's not entirely accurate anymore.

Recall the original `=>` form of `clickHandler` from earlier in the chapter?

```
const clickHandler = evt =>
    evt.target.matches("button") ?
        this.theFormElem.submit() :
        evt.stopPropagation();
```

If we use that form, in the same context as our event binding, it could look like this:

```
const clickHandler = evt =>
    evt.target.matches("button") ?
        this.theFormElem.submit() :
        evt.stopPropagation();

this.submitBtn.addEventListener("click",clickHandler,false);
```

A lot of developers prefer to even further reduce it, to an inline `=>` arrow function:

```
this.submitBtn.addEventListener(
    "click",
    evt => evt.target.matches("button") ?
        this.theFormElem.submit() :
        evt.stopPropagation(),
    false
);
```

When we write an `=>` arrow function, we know for sure that its `this` binding will exactly be the current `this` binding of whatever surrounding function is running, regardless of what the call-site of the `=>` arrow function looks like. So in other words, *how* we wrote the `=>` arrow function, and *where* we wrote it, does matter.

That doesn't fully answer the `this` question, though. It just shifts the question to *how the enclosing function was invoked*. Actually, the focus on the call-site is still the only thing that matters.

But the nuance I'm confessing to having omitted until *this* moment is: it matters *which* call-site we consider, not just *any* call-site in the current call stack. The call-site that matters is, the nearest function-invocation in the current call stack ***that actually assigns a `this` context***.

Since an `=>` arrow function never has a `this`-assigning call-site (no matter what), that call-site isn't relevant to the question. We have to keep stepping up the call stack until we find a function invocation that *is* `this`-assigning – even if such invoked function is not itself `this`-aware.

THAT is the only call-site that matters.

Find The Right Call-Site Let me illustrate, with a convoluted mess of a bunch of nested functions/calls:

```
globalThis.value = { result: "Sad face" };

function one() {
    function two() {
        var three = {
            value: { result: "Hmmm" },

            fn: () => {
                const four = () => this.value;
                return four.call({
                    value: { result: "OK", },
                });
            },
        };
        return three.fn();
    };
    return two();
}

new one();          // ???
```

Can you run through that (nightmare) in your head and determine what will be returned from the `new one()` invocation?

It could be any of these:

```
// from `four.call(..)`:
{ result: "OK" }

// or, from `three` object:
{ result: "Hmmm" }

// or, from the `globalThis.value`:
```

```
{ result: "Sad face" }

// or, empty object from the `new` call:
{}
```

The call-stack for that `new one()` invocation is:

```
four          |
three.fn      |
two           | (this = globalThis)
one           | (this = {})
[ global ]    | (this = globalThis)
```

Since `four()` and `fn()` are both `=>` arrow functions, the `three.fn()` and `four.call(..)` call-sites are not `this`-assigning; thus, they're irrelevant for our query. What's the next invocation to consider in the call-stack? `two()`. That's a regular function (it can accept `this`-assignment), and the call-site matches the *default context* assignment rule (#4). Since we're not in strict-mode, `this` is assigned `globalThis`.

When `four()` is running, `this` is just a normal variable. It looks then to its containing function (`three.fn()`), but it again finds a function with no `this`. So it goes up another level, and finds a `two()` *regular* function that has a `this` defined. And that `this` is `globalThis`. So the `this.value` expression resolves to `globalThis.value`, which returns us… `{ result: "Sad face" }`.

…

Take a deep breath. I know that's a lot to mentally process. And in fairness, that's a super contrived example. You'll almost never see all that complexity mixed in one call-stack.

But you absolutely will find mixed call-stacks in real programs. You need to get comfortable with the analysis I just illustrated, to be able to unwind the call-stack until you find the most recent `this`-assigning call-site.

Remember the addage I quoted earlier: "with great power comes great responsibility". Choosing `this`-oriented code (even `class`es) means choosing both the flexibility it affords us, as well as needing to be comfortable navigating the call-stack to understand how it will behave.

That's the only way to effectively write (and later read!) `this`-aware code.

This Is Bound To Come Up

Backing up a bit, there's another option if you don't want to use an `=>` arrow function's "lexical this" behavior to address the button event handler functionality.

In addition to `call(..)` / `apply(..)` – these invoke functions, remember! – JS functions also have a third utility built in, called `bind(..)` – which does *not*

invoke the function, just to be clear.

The `bind(..)` utility defines a *new* wrapped/bound version of a function, where its `this` is preset and fixed, and cannot be overridden with a `call(..)` or `apply(..)`, or even an *implicit context* object at the call-site:

```
this.submitBtn.addEventListener(
    "click",
    this.clickHandler.bind(this),
    false
);
```

Since I'm passing in a `this`-bound function as the event handler, it similarly doesn't matter how that utility tries to set a `this`, because I've already forced the `this` to be what I wanted: the value of `this` from the surrounding function invocation context.

Hardly New This pattern is often referred to as "hard binding", since we're creating a function reference that is strongly bound to a particular `this`. A lot of JS writings have claimed that the `=>` arrow function is essentially just syntax for the `bind(this)` hard-binding. It's not. Let's dig in.

If you were going to create a `bind(..)` utility, it might look kinda like *this*:

```
function bind(fn,context) {
    return function bound(...args){
        return fn.apply(context,args);
    };
}
```

NOTE:

This is not actually how `bind(..)` is implemented. The behavior is more sophisticated and nuanced. I'm only illustrating one portion of its behavior in this snippet.

Does that look familiar? It's using the good ol' fake "lexical this" hack. And under the covers, it's an *explicit context* assignment, in this case via `apply(..)`.

So wait… doesn't that mean we could just do it with an `=>` arrow function?

```
function bind(fn,context) {
    return (...args) => fn.apply(context,args);
}
```

Eh… not quite. As with most things in JS, there's a bit of nuance. Let me illustrate:

```
// candidate implementation, for comparison
function fakeBind(fn,context) {
```

```
    return (...args) => fn.apply(context,args);
}

// test subject
function thisAwareFn() {
    console.log(`Value: ${this.value}`);
}

// control data
var obj = {
    value: 42,
};

// experiment
var f = thisAwareFn.bind(obj);
var g = fakeBind(thisAwareFn,obj);

f();            // Value: 42
g();            // Value: 42

new f();        // Value: undefined
new g();        // <--- ???
```

First, look at the `new f()` call. That's admittedly a strange usage, to call `new` on a hard-bound function. It's probably quite rare that you'd ever do so. But it shows something kind of interesting. Even though `f()` was hard-bound to a `this` context of `obj`, the `new` operator was able to hijack the hard-bound function's `this` and re-bind it to the newly created and empty object. That object has no `value` property, which is why we see `"Value: undefined"` printed out.

If that feels strange, I agree. It's a weird corner nuance. It's not something you'd likely ever exploit. But I point it out not just for trivia. Refer back to the four rules presented earlier in this chapter. Remember how I asserted their order-of-precedence, and `new` was at the top (#1), ahead of *explicit* `call(..)` / `apply(..)` assignment rule (#2)?

Since we can sort of think of `bind(..)` as a variation of that rule, we now see that order-of-precedence proven. `new` is more precedent than, and can override, even a hard-bound function. Sort of makes you think the hard-bound function is maybe not so "hard"-bound, huh?!

But… what's going to happen with the `new g()` call, which is invoking `new` on the returned `=>` arrow function? Do you predict the same outcome as `new f()`?

Sorry to disappoint.

That line will actually throw an exception, because an `=>` function cannot be used with the `new` keyword.

But why? My best answer, not being authoritative on TC39 myself, is that conceptually and actually, an `=>` arrow function is not a function with a hard-bound `this`, it's a function that has no `this` at all. As such, `new` makes no sense against such a function, so JS just disallows it.

NOTE:

Recall earlier, when I pointed out that `=>` arrow functions have `call(..)`, `apply(..)`, and indeed even a `bind(..)`. But we've see that such functions basically ignore these utilities as no-ops. It's a bit strange, in my opinion, that `=>` arrow functions have all those utilities as pass-through no-ops, but for the `new` keyword, that's not just, again, a no-op pass-through, but rather disallowed with an exception.

But the main point is: an `=>` arrow function is *not* a syntactic form of `bind(this)`.

Losing This Battle

Returning once again to our button event handler example:

```
this.submitBtnaddEventListener(
    "click",
    this.clickHandler,
    false
);
```

There's a deeper concern we haven't yet addressed.

We've seen several different approaches to construct a different callback function reference to pass in there, in place of `this.clickHandler`.

But whichever of those ways we choose, they are producing literally a different function, not just an in-place modification to our existing `clickHandler` function.

Why does that matter?

Well, first of all, the more functions we create (and re-create), the more processing time (very slight) and the more memory (pretty small, usually) we're chewing up. And when we're re-creating a function reference, and throwing an old one away, that's also leaving un-reclaimed memory sitting around, which puts pressure on the garbage collector (GC) to more often, pause the universe of our program momentarily while it cleans up and reclaims that memory.

If hooking up this event listening is a one-time operation, no big deal. But if it's happening over and over again, the system-level performance effects *can* start to add up. Ever had an otherwise smooth animation jitter? That was probably the GC kicking in, cleaning up a bunch of reclaimable memory.

But another concern is, for things like event handlers, if we're going to remove an event listener at some later time, we need to keep a reference to the exact same function we attached originally. If we're using a library/framework, often (but not always!) they take care of that little dirty-work detail for you. But otherwise, it's on us to make sure that whatever function we plan to attach, we hold onto a reference just in case we need it later.

So the point I'm making is: presetting a `this` assignment, no matter how you do it, so that it's predictable, comes with a cost. A system level cost and a program maintenance/complexity cost. It is *never* free.

One way of reacting to that fact is to decide, OK, we're just going to manufacture all those `this`-assigned function references once, ahead of time, up-front. That way, we're sure to reduce both the system pressure, and the code pressure, to a minimum.

Sounds reasonable, right? Not so fast.

Pre-Binding Function Contexts If you have a one-off function reference that needs to be `this`-bound, and you use an `=>` arrow or a `bind(this)` call, I don't see any problems with that.

But if most or all of the `this`-aware functions in a segment of your code invoked in ways where the `this` isn't the predictable context you expect, and so you decide you need to hard-bind them all… I think that's a big warning signal that you're going about things the wrong way.

Please recall the discussion in the "Avoid This" section from Chapter 3, which started with this snippet of code:

```
class Point2d {
    x = null
    getDoubleX = () => this.x * 2

    constructor(x,y) {
        this.x = x;
        this.y = y;
    }
    toString() { /* .. */ }
}

var point = new Point2d(3,4);
```

Now imagine we did this with that code:

```
const getX = point.getDoubleX;

// later, elsewhere
```

```
getX();          // 6
```

As you can see, the problem we were trying to solve is the same as we've been dealing with here in this chapter. It's that we wanted to be able to invoke a function reference like `getX()`, and have that *mean* and *behave like* `point.getDoubleX()`. But `this` rules on *regular* functions don't work that way.

So we used an `=>` arrow function. No big deal, right!?

Wrong.

The real root problem is that we *want* two conflicting things out of our code, and we're trying to use the same *hammer* for both *nails*.

We want to have a `this`-aware method stored on the `class` prototype, so that there's only one definition for the function, and all our subclasses and instances nicely share that same function. And the way they all share is through the power of the dynamic `this` binding.

But at the same time, we *also* want those function references to magically stay `this`-assinged to our instance when we pass those function references around and other code is in charge of the call-site.

In other words, sometimes we want something like `point.getDoubleX` to mean, "give me a reference that's `this`-assigned to `point`", and other times we want the same expression `point.getDoubleX` to mean, give me a dynamic `this`-assignable function reference so it can properly get the context I need it to at this moment.

Perhaps JS could offer a different operator besides `.`, like `::` or `->` or something like that, which would let you distinguish what kind of function reference you're after. In fact, there's a long-standing proposal for a `this`-binding operator (`::`), that picks up attention from time to time, and then seems to stall out. Who knows, maybe someday such an operator will finally land, and we'll have better options.

But I strongly suspect that even if it does land someday, it's going to vend a whole new function reference, exactly as the `=>` or `bind(this)` approaches we've already talked about. It won't come as a free and perfect solution. There will always be a tension between wanting the same function to sometimes be `this`-flexible and sometimes be `this`-predictable.

What JS authors of `class`-oriented code often run up against, sooner or later, is this exact tension. And you know what they do?

They don't consider the *costs* of simply pre-binding all the class's `this`-aware methods as instead `=>` arrow functions in member properties. They don't realize that it's completely defeated the entire purpose of the `[[Prototype]]` chain.

And they don't realize that if fixed-context is what they *really need*, there's an entirely different mechanism in JS that is better suited for that purpose.

Take A More Critical Look So when you do this sort of thing:

```
class Point2d {
    x = null
    y = null
    getDoubleX = () => this.x * 2
    toString = () => `(${this.x},${this.y})`

    constructor(x,y) {
        this.x = x;
        this.y = y;
    }
}

var point = new Point2d(3,4);
var anotherPoint = new Point2d(5,6);

var f = point.getDoubleX;
var g = anotherPoint.toString;

f();            // 6
g();            // (5,6)
```

I say, "ick!", to the hard-bound `this`-aware methods `getDoubleX()` and `toString()` there. To me, that's a code smell. But here's an even *worse* approach that has been favored by many developers in the past:

```
class Point2d {
    x = null
    y = null

    constructor(x,y) {
        this.x = x;
        this.y = y;
        this.getDoubleX = this.getDoubleX.bind(this);
        this.toString = this.toString.bind(this);
    }
    getDoubleX() { return this.x * 2; }
    toString() { return `(${this.x},${this.y})`; }
}

var point = new Point2d(3,4);
var anotherPoint = new Point2d(5,6);
```

```
var f = point.getDoubleX;
var g = anotherPoint.toString;

f();            // 6
g();            // (5,6)
```

Double ick.

In both cases, you're using a `this` mechanism but completely betraying/neutering it, by taking away all the powerful dynamicism of `this`.

You really should at least be contemplating this alternate approach, which skips the whole `this` mechanism altogether:

```
function Point2d(px,py) {
    var x = px;
    var y = py;

    return {
        getDoubleX() { return x * 2; },
        toString() { return `(${x},${y})`; }
    };
}

var point = Point2d(3,4);
var anotherPoint = Point2d(5,6);

var f = point.getDoubleX;
var g = anotherPoint.toString;

f();            // 6
g();            // (5,6)
```

You see? No ugly or complex `this` to clutter up that code or worry about corner cases for. Lexical scope is super straightforward and intuitive.

When all we want is for most/all of our function behaviors to have a fixed and predictable context, the most appropriate solution, the most straightforward and even performant solution, is lexical variables and scope closure.

When you go to all to the trouble of sprinkling `this` references all over a piece of code, and then you cut off the whole mechanism at the knees with `=>` "lexical this" or `bind(this)`, you chose to make the code more verbose, more complex, more overwrought. And you got nothing out of it that was more beneficial, except to follow the `this` (and `class`) bandwagon.

…

Deep breath. Collect yourself.

I'm talking to myself, not you. But if what I just said bothers you, I'm talking to you, too!

OK, listen. That's just my opinion. If you don't agree, that's fine. But apply the same level of rigor to thinking about how these mechanisms work, as I have, when you decide what conclusions you want to arrive at.

Variations

Before we close out our lengthy discussion of `this`, there's a few irregular variations on function calls that we should discuss.

Indirect Function Calls

Recall this example from earlier in the chapter?

```
var point = {
    x: null,
    y: null,

    init(x,y) {
        this.x = x;
        this.y = y;
    },
    rotate(angleRadians) { /* .. */ },
    toString() { /* .. */ },
};

var init = point.init;
init(3,4);                  // broken!
```

This is broken because the `init(3,4)` call-site doesn't provide the necessary `this`-assignment signal. But there's other ways to observe a similar breakage. For example:

```
(1,point.init)(3,4);        // broken!
```

This strange looking syntax is first evaluating an expression `(1,point.init)`, which is a comma series expression. The result of such an expression is the final evaluated value, which in this case is the function reference (held by `point.init`).

So the outcome puts that function reference onto the expression stack, and then invokes that value with `(3,4)`. That's an indirect invocation of the function. And what's the result? It actually matches the *default context* assignment rule (#4) we looked at earlier in the chapter.

Thus, in non-strict mode, the `this` for the `point.init(..)` call will be `globalThis`. Had we been in strict-mode, it would have been `undefined`, and

the `this.x = x` operation would then have thrown an exception for invalidly accessing the `x` property on the `undefined` value.

There's several different ways to get an indirect function invocation. For example:

```
(()=>point.init)()(3,4);     // broken!
```

And another example of indirect function invocation is the Immediately Invoked Function Expression (IIFE) pattern:

```
(function(){
    // `this` assigned via "default" rule
})();
```

As you can see, the function expression value is put onto the expression stack, and then it's invoked with the `()` on the end.

But what about this code:

```
(point.init)(3,4);
```

What will be the outcome of that code?

By the same reasoning we've seen in the previous examples, it stands to reason that the `point.init` expression puts the function value onto the expression stack, and then invoked indirectly with `(3,4)`.

Not quite, though! JS grammar has a special rule to handle the invocation form `(someIdentifier)(..)` as if it had been `someIdentifier(..)` (without the `(..)` around the identifier name).

Wondering why you might want to ever force the *default context* for `this` assignment via an indirect function invocation?

Accessing `globalThis`

Before we answer that, let's introduce another way of performing indirect function `this` assignment. Thus far, the indirect function invocation patterns shown are sensitive to strict-mode. But what if we wanted an indirect function `this` assignment that doesn't respect strict-mode.

The `Function(..)` constructor takes a string of code and dynamically defines the equivalent function. However, it always does so as if that function had been declared in the global scope. And furthermore, it ensures such function *does not* run in strict-mode, no matter the strict-mode status of the program. That's the same outcome as running an indirect

One niche usage of such strict-mode agnostic indirect function `this` assignment is for getting a reliable reference to the true global object prior to when the JS specification actually defined the `globalThis` identifier (for example, in a polyfill for it):

```
"use strict";

var gt = new Function("return this")();
gt === globalThis;                          // true
```

In fact, a similar outcome, using the comma operator trick (see previous section) and `eval(..)`:

```
"use strict";

function getGlobalThis() {
    return (1,eval)("this");
}

getGlobalThis() === globalThis;       // true
```

NOTE:

`eval("this")` would be sensitive to strict-mode, but `(1,eval)("this")` is not, and therefor reliably gives us the `globalThis` in any program.

Unfortunately, the `new Function(..)` and `(1,eval)(..)` approaches both have an important limitation: that code will be blocked in browser-based JS code if the app is served with certain Content-Security-Policy (CSP) restrictions, disallowing dynamic code evaluation (for security reasons).

Can we get around this? Yes, mostly. [1]

The JS specification says that a getter function defined on the global object, or on any object that inherits from it (like `Object.prototype`), runs the getter function with `this` context assigned to `globalThis`, regardless of the program's strict-mode.

```
// Adapted from: https://mathiasbynens.be/notes/globalthis#robust-polyfill
function getGlobalThis() {
    Object.defineProperty(Object.prototype,"__get_globalthis__",{
        get() { return this; },
        configurable: true
    });
    var gt = __get_globalthis__;
    delete Object.prototype.__get_globalthis__;
    return gt;
}

getGlobalThis() === globalThis;       // true
```

[1] "A horrifying globalThis polyfill in universal JavaScript"; Mathias Bynens; April 18 2019; https://mathiasbynens.be/notes/globalthis#robust-polyfill ; Accessed July 2022

Yeah, that's super gnarly. But that's JS `this` for you!

Template Tag Functions

There's one more unusual variation of function invocation we should cover: tagged template functions.

Template strings – what I prefer to call interpolated literals – can be "tagged" with a prefix function, which is invoked with the parsed contents of the template literal:

```
function tagFn(/* .. */) {
    // ..
}

tagFn`actually a function invocation!`;
```

As you can see, there's no `(..)` invocation syntax, just the tag function (`tagFn`) appearing before the `` `template literal` ``; whitespace is allowed between them, but is very uncommon.

Despite the strange appearance, the function `tagFn(..)` will be invoked. It's passed the list of one or more string literals that were parsed from the template literal, along with any interpolated expression values that were encountered.

We're not going to cover all the ins and outs of tagged template functions – they're seriously one of the most powerful and interesting features ever added to JS – but since we're talking about `this` assignment in function invocations, for completeness sake we need to talk about how `this` will be assigned.

The other form for tag functions you may encounter is:

```
var someObj = {
    tagFn() { /* .. */ }
};

someObj.tagFn`also a function invocation!`;
```

Here's the easy explanation: `tagFn`..`` and `someObj.tagFn`..`` will each have `this`-assignment behavior corresponding to call-sites as `tagFn(..)` and `someObj.tagFn(..)`, respectively. In other words, `tagFn`..`` behaves by the *default context* assignment rule (#4), and `someObj.tagFn`..`` behaves by the *implicit context* assignment rule (#3).

Luckily for us, we don't need to worry about the `new` or `call(..)` / `apply(..)` assignment rules, as those forms aren't possible with tag functions.

It should be pointed out that it's pretty rare for a tagged template literal function to be defined as `this`-aware, so it's fairly unlikely you'll need to apply these rules. But just in case, now you're in the *know*.

Stay Aware

So, that's `this`. I'm willing to bet for many of you, it was a bit more… shall we say, involved… than you might have been expecting.

The good news, perhaps, is that in practice you don't often trip over all these different complexities. But the more you use `this`, the more it requires you, and the readers of your code, to understand how it actually works.

The lesson here is that you should be intentional and aware of all aspects of `this` before you go sprinkling it about your code. Make sure you're using it most effectively and taking full advantage of this important pillar of JS.

You Don't Know JS Yet: Objects & Classes - 2nd Edition

Chapter 5: Delegation

NOTE:
Work in progress

We've thoroughly explored objects, prototypes, classes, and now the `this` keyword. But we're now going to revisit what we've learned so far from a bit of a different perspective.

What if you could leverage all the power of the objects, prototypes, and dynamic `this` mechanisms together, without ever using `class` or any of its descendants?

In fact, I would argue JS is inherently less class-oriented than the `class` keyword might appear. Because JS is a dynamic, prototypal language, its strong suit is actually… *delegation*.

Preamble

Before we begin looking at delegation, I want to offer a word of caution. This perspective on JS's object `[[Prototype]]` and `this` function context mechanisms is *not* mainstream. It's *not* how framework authors and libraries utilize JS. You won't, to my knowledge, find any big apps out there using this pattern.

So why on earth would I devote a chapter to such a pattern, if it's so unpopular?

Good question. The cheeky answer is: because it's my book and I can do what I feel like!

But the deeper answer is, because I think developing *this* understanding of one of the language's core pillars helps you *even if* all you ever do is use `class`-style JS patterns.

To be clear, delegation is not my invention. It's been around as a design pattern for decades. And for a long time, developers argued that prototypal delegation was *just* the dynamic form of inheritance.[1] But I think that was a mistake to conflate the two.[2]

[1] "Treaty of Orlando"; Henry Lieberman, Lynn Andrea Stein, David Ungar; Oct 6, 1987; https://web.media.mit.edu/~lieber/Publications/Treaty-of-Orlando-Treaty-Text.pdf ; PDF; Accessed July 2022

[2] "Classes vs. Prototypes, Some Philosophical and Historical Observations"; Antero Taivalsaari; Apr 22, 1996; https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.56.4713&rep=rep1&type=pdf ; PDF; Accessed July 2022

For the purposes of this chapter, I'm going to present delegation, as implemented via JS mechanics, as an alternative design pattern, positioned somewhere between class-orientation and object-closure/module patterns.

The first step is to *de-construct* the `class` mechanism down to its individual parts. Then we'll cherry-pick and mix the parts a bit differently.

What's A Constructor, Anyway?

In Chapter 3, we saw `constructor(..)` as the main entry point for construction of a `class` instance. But the `constructor(..)` doesn't actually do any *creation* work, it's only *initialization* work. In other words, the instance is already created by the time the `constructor(..)` runs and initializes it – e.g., `this.whatever` types of assignments.

So where does the *creation* work actually happen? In the `new` operator. As the section "New Context Invocation" in Chapter 4 explains, there are four steps the `new` keyword performs; the first of those is the creation of a new empty object (the instance). The `constructor(..)` isn't even invoked until step 3 of `new`'s efforts.

But `new` is not the only – or perhaps even, best – way to *create* an object "instance". Consider:

```
// a non-class "constructor"
function Point2d(x,y) {
    // create an object (1)
    var instance = {};

    // initialize the instance (3)
    instance.x = x;
    instance.y = y;

    // return the instance (4)
    return instance;
}

var point = Point2d(3,4);

point.x;                        // 3
point.y;                        // 4
```

There's no `class`, just a regular function definition (`Point2d(..)`). There's no `new` invocation, just a regular function call (`Point2d(3,4)`). And there's no `this` references, just regular object property assignments (`instance.x = ..`).

The term that's most often used to refer to this pattern of code is that `Point2d(..)` here is a *factory function*. Invoking it causes the construction

(creation and initialization) of an object, and returns that back to us. That's an extremely common pattern, at least as common as class-oriented code.

I comment-annotated `(1)`, `(3)`, and `(4)` in that snippet, which roughly correspond to steps 1, 3, and 4 of the `new` operation. But where's step 2?

If you recall, step 2 of `new` is about linking the object (created in step 1) to another object, via its `[[Prototype]]` slot (see Chapter 2). So what object might we want to link our `instance` object to? We could link it to an object that holds functions we'd like to associate/use with our instance.

Let's amend the previous snippet:

```
var prototypeObj = {
    toString() {
        return `(${this.x},${this.y})`;
    },
}

// a non-class "constructor"
function Point2d(x,y) {
    // create an object (1)
    var instance = {
        // link the instance's [[Prototype]] (2)
        __proto__: prototypeObj,
    };

    // initialize the instance (3)
    instance.x = x;
    instance.y = y;

    // return the instance (4)
    return instance;
}

var point = Point2d(3,4);

point.toString();           // (3,4)
```

Now you see the `__proto__` assignment that's setting up the internal `[[Prototype]]` linkage, which was the missing step 2. I used the `__proto__` here merely for illustration purposes; using `setPrototypeOf(..)` as shown in Chapter 4 would have accomplished the same task.

New Factory Instance

What do you think would happen if we used `new` to invoke the `Point2d(..)` function as shown here?

```
var anotherPoint = new Point2d(5,6);

anotherPoint.toString(5,6);          // (5,6)
```

Wait! What's going on here? A regular, non-`class` factory function in invoked with the `new` keyword, as if it was a `class`. Does that change anything about the outcome of the code?

No… and yes. `anotherPoint` here is exactly the same object as it would have been had I not used `new`. But! The object that `new` creates, links, and assigns as `this` context? *That* object was completely ignored and thrown away, ultimately to be garbage collected by JS. Unfortunately, the JS engine cannot predict that you're not going to use the object that you asked `new` to create, so it always still gets cteated even if it goes unused.

That's right! Using a `new` keyword against a factory function might *feel* more ergonomic or familiar, but it's quite wasteful, in that it creates **two** objects, and wastefully throws one of them away.

Factory Initialization

In the current code example, the `Point2d(..)` function still looks an awful lot like a normal `constructor(..)` of a `class` definition. But what if we moved the initialization code to a separate function, say named `init(..)`:

```
var prototypeObj = {
    init(x,y) {
        // initialize the instance (3)
        this.x = x;
        this.y = y;
    },
    toString() {
        return `(${this.x},${this.y})`;
    },
}

// a non-class "constructor"
function Point2d(x,y) {
    // create an object (1)
    var instance = {
        // link the instance's [[Prototype]] (2)
        __proto__: prototypeObj,
    };

    // initialize the instance (3)
    instance.init(x,y);

    // return the instance (4)
```

```
    return instance;
}

var point = Point2d(3,4);

point.toString();           // (3,4)
```

The `instance.init(..)` call makes use of the `[[Prototype]]` linkage set up via `__proto__` assignment. Thus, it *delegates* up the prototype chain to `prototypeObj.init(..)`, and invokes it with a `this` context of `instance` – via *implicit context* assignment (see Chapter 4).

Let's continue the deconstruction. Get ready for a switcheroo!

```
var Point2d = {
    init(x,y) {
        // initialize the instance (3)
        this.x = x;
        this.y = y;
    },
    toString() {
        return `(${this.x},${this.y})`;
    },
};
```

Whoa, what!? I discarded the `Point2d(..)` function, and instead renamed the `prototypeObj` as `Point2d`. Weird.

But let's look at the rest of the code now:

```
// steps 1, 2, and 4
var point = { __proto__: Point2d, };

// step 3
point.init(3,4);

point.toString();           // (3,4)
```

And one last refinement: let's use a built-in utility JS provides us, called `Object.create(..)`:

```
// steps 1, 2, and 4
var point = Object.create(Point2d);

// step 3
point.init(3,4);

point.toString();           // (3,4)
```

What operations does `Object.create(..)` perform?

1. create a brand new empty object, out of thin air.
2. link the `[[Prototype]]` of that new empty object to the function's `.prototype` object.

If those look familiar, it's because those are exactly the same first two steps of the `new` keyword (see Chapter 4).

Let's put this back together now:

```
var Point2d = {
    init(x,y) {
        this.x = x;
        this.y = y;
    },
    toString() {
        return `(${this.x},${this.y})`;
    },
};

var point = Object.create(Point2d);

point.init(3,4);

point.toString();           // (3,4)
```

Hmmm. Take a few moments to ponder what's been derived here. How does it compare to the `class` approach?

This pattern ditches the `class` and `new` keywords, but accomplishes the exact same outcome. The *cost*? The single `new` operation was broken up into two statements: `Object.create(Point2d)` and `point.init(3,4)`.

Help Me Reconstruct! If having those two operations separate bothers you – is it *too deconstructed*!? – they can always be recombined in a little factory helper:

```
function make(objType,...args) {
    var instance = Object.create(objType);
    instance.init(...args);
    return instance;
}

var point = make(Point2d,3,4);

point.toString();           // (3,4)
```

TIP:

Such a `make(..)` factory function helper works generally for any object-type, as long as you follow the implied convention that each `objType` you link to has a function named `init(..)` on it.

And of course, you can still create as many instances as you'd like:

```
var point = make(Point2d,3,4);

var anotherPoint = make(Point2d,5,6);
```

Ditching Class Thinking

Quite frankly, the *deconstruction* we just went through only ends up in slightly different, and maybe slightly better or slightly worse, code as compared to the `class` style. If that's all delegation was about, it probably wouldn't even be useful enough for more than a footnote, much less a whole chapter.

But here's where we're going to really start pushing the class-oriented thinking itself, not just the syntax, aside.

Class-oriented design inherently creates a hierarchy of *classification*, meaning how we divide up and group characteristics, and then stack them vertically in an inheritance chain. Moreover, defining a subclass is a specialization of the generalized base class. Instantiating is a specialization of the generalized class.

Behavior in a traditional class hierarchy is a vertical composition through the layers of the inheritance chain. Attempts have been made over the decades, and even become rather popular at times, to flatten out deep hierarchies of inheritance, and favor a more horizontal composition through *mixins* and related ideas.

I'm not asserting there's anything wrong with those ways of approaching code. But I am saying that they aren't *naturally* how JS works, so adopting them in JS has been a long, winding, complicated road, and has variously accreted lots of nuanced syntax to retrofit on top of JS's core `[[Prototype]]` and `this` pillar.

For the rest of this chapter, I intend to discard both the syntax of `class` *and* the thinking of *class*.

Delegation Illustrated

So what is delegation about? At its core, it's about two or more *things* sharing the effort of completing a task.

Instead of defining a `Point2d` general parent *thing* that represents shared behavior that a set of one or more child `point` / `anotherPoint` *things* inherit

from, delegation moves us to building our program with discrete peer *things* that cooperate with each other.

I'll sketch that out in some code:

```
var Coordinates = {
    setX(x) {
        this.x = x;
    },
    setY(y) {
        this.y = y;
    },
    setXY(x,y) {
        this.setX(x);
        this.setY(y);
    },
};

var Inspect = {
    toString() {
        return `(${this.x},${this.y})`;
    },
};

var point = {};

Coordinates.setXY.call(point,3,4);
Inspect.toString.call(point);          // (3,4)

var anotherPoint = Object.create(Coordinates);

anotherPoint.setXY(5,6);
Inspect.toString.call(anotherPoint);   // (5,6)
```

Let's break down what's happening here.

I've defined `Coordinates` as a concrete object that holds some behaviors I associate with setting point coordinates (`x` and `y`). I've also defined `Inspect` as a concrete object that holds some debug inspection logic, such as `toString()`.

I then create two more concrete objects, `point` and `anotherPoint`.

`point` has no specific `[[Prototype]]` (default: `Object.prototype`). Using *explicit context* assignment (see Chapter 4), I invoke the `Coordinates.setXY(..)` and `Inspect.toString()` utilities in the context of `point`. That is what I call *explicit delegation*.

`anotherPoint` is `[[Prototype]]` linked to `Coordinates`, mostly for a bit of convenience. That lets me use *implicit context* assignment with

`anotherPoint.setXY(..)`. But I can still *explicitly* share `anotherPoint` as context for the `Inspect.toString()` call. That's what I call *implicit delegation*.

Don't miss *this*: We still accomplished composition: we composed the behaviors from `Coordinates` and `Inspect`, during runtime function invocations with `this` context sharing. We didn't have to author-combine those behaviors into a single `class` (or base-subclass `class` hierarchy) for `point` / `anotherPoint` to inherit from. I like to call this runtime composition, **virtual composition**.

The *point* here is: none of these four objects is a parent or child. They're all peers of each other, and they all have different purposes. We can organize our behavior in logical chunks (on each respective object), and share the context via `this` (and, optionally `[[Prototype]]` linkage), which ends up with the same composition outcomes as the other patterns we've examined thus far in the book.

That is the heart of the **delegation** pattern, as JS embodies it.

TIP:

In the first edition of this book series, this book ("this & Object Prototypes") coined a term, "OLOO", which stands for "Objects Linked to Other Objects" – to stand in contrast to "OO" ("Object Oriented"). In this preceding example, you can see the essence of OLOO: all we have are objects, linked to and cooperating with, other objects. I find this beautiful in its simplicity.

Composing Peer Objects

Let's take *this delegation* even further.

In the preceding snippet, `point` and `anotherPoint` merely held data, and the behaviors they delegated to were on other objects (`Coordinates` and `Inspect`). But we can add behaviors directly to any of the objects in a delegation chain, and those behaviors can even interact with each other, all through the magic of *virtual composition* (`this` context sharing).

To illustrate, we'll evolve our current *point* example a fair bit. And as a bonus we'll actually draw our points on a `<canvas>` element in the DOM. Let's take a look:

```
var Canvas = {
    setOrigin(x,y) {
        this.ctx.translate(x,y);

        // flip the canvas context vertically,
        // so coordinates work like on a normal
        // 2d (x,y) graph
```

```
        this.ctx.scale(1,-1);
    },
    pixel(x,y) {
        this.ctx.fillRect(x,y,1,1);
    },
    renderScene() {
        // clear the canvas
        var matrix = this.ctx.getTransform();
        this.ctx.resetTransform();
        this.ctx.clearRect(
            0, 0,
            this.ctx.canvas.width,
            this.ctx.canvas.height
        );
        this.ctx.setTransform(matrix);

        this.draw();  // <-- where is draw()?
    },
};

var Coordinates = {
    setX(x) {
        this.x = Math.round(x);
    },
    setY(y) {
        this.y = Math.round(y);
    },
    setXY(x,y) {
        this.setX(x);
        this.setY(y);
        this.render();   // <-- where is render()?
    },
};

var ControlPoint = {
    // delegate to Coordinates
    __proto__: Coordinates,

    // NOTE: must have a <canvas id="my-canvas">
    // element in the DOM
    ctx: document.getElementById("my-canvas")
        .getContext("2d"),

    rotate(angleRadians) {
        var rotatedX = this.x * Math.cos(angleRadians) -
            this.y * Math.sin(angleRadians);
```

```
        var rotatedY = this.x * Math.sin(angleRadians) +
            this.y * Math.cos(angleRadians);
        this.setXY(rotatedX,rotatedY);
    },
    draw() {
        // plot the point
        Canvas.pixel.call(this,this.x,this.y);
    },
    render() {
        // clear the canvas, and re-render
        // our control-point
        Canvas.renderScene.call(this);
    },
};

// set the logical (0,0) origin at this
// physical location on the canvas
Canvas.setOrigin.call(ControlPoint,100,100);

ControlPoint.setXY(30,40);
// [renders point (30,40) on the canvas]

// ..
// later:

// rotate the point about the (0,0) origin
// 90 degrees counter-clockwise
ControlPoint.rotate(Math.PI / 2);
// [renders point (-40,30) on the canvas]
```

OK, that's a lot of code to digest. Take your time and re-read the snippet several times. I added a couple of new concrete objects (`Canvas` and `ControlPoint`) alongside the previous `Coordinates` object.

Make sure you see and understand the interactions between these three concrete objects.

`ControlPoint` is linked (via `__proto__`) to *implicitly delegate* (`[[Prototype]]` chain) to `Coordinates`.

Here's an *explicit delegation*: `Canvas.setOrigin.call(ControlPoint,100,100);`; I'm invoking the `Canvas.setOrigin(..)` call in the context of `ControlPoint`. That has the effect of sharing `ctx` with `setOrigin(..)`, via `this`.

`ControlPoint.setXY(..)` delegates *implicitly* to `Coordinates.setXY(..)`, but still in the context of `ControlPoint`. Here's a key detail that's easy to miss: see the `this.render()` inside of `Coordinates.setXY(..)`? Where does that come from? Since the `this` context is `ControlPoint` (not `Coordinates`),

it's invoking `ControlPoint.render()`.

`ControlPoint.render()` *explicitly delegates* to `Canvas.renderScene()`, again still in the `ControlPoint` context. `renderScene()` calls `this.draw()`, but where does that come from? Yep, still from `ControlPoint` (via `this` context).

And `ControlPoint.draw()`? It *explicitly delegates* to `Canvas.pixel(..)`, yet again still in the `ControlPoint` context.

All three objects have methods that end up invoking each other. But these calls aren't particularly hard-wired. `Canvas.renderScene()` doesn't call `ControlPoint.draw()`, it calls `this.draw()`. That's important, because it means that `Canvas.renderScene()` is more flexible to use in a different `this` context – e.g., against another kind of *point* object besides `ControlPoint`.

It's through the `this` context, and the `[[Prototype]]` chain, that these three objects basically are mixed (composed) virtually together, as needed at each step, so that they work together **as if they're one object rather than three seperate objects**.

That's the *beauty* of virtual composition as realized by the delegation pattern in JS.

Flexible Context

I mentioned above that we can pretty easily add other concrete objects into the mix. Here's an example:

```
var Coordinates = { /* .. */ };

var Canvas = {
    /* .. */
    line(start,end) {
        this.ctx.beginPath();
        this.ctx.moveTo(start.x,start.y);
        this.ctx.lineTo(end.x,end.y);
        this.ctx.stroke();
    },
};

function lineAnchor(x,y) {
    var anchor = {
        __proto__: Coordinates,
        render() {},
    };
    anchor.setXY(x,y);
    return anchor;
}
```

```
var GuideLine = {
    // NOTE: must have a <canvas id="my-canvas">
    // element in the DOM
    ctx: document.getElementById("my-canvas")
        .getContext("2d"),

    setAnchors(sx,sy,ex,ey) {
        this.start = lineAnchor(sx,sy);
        this.end = lineAnchor(ex,ey);
        this.render();
    },
    draw() {
        // plot the point
        Canvas.line.call(this,this.start,this.end);
    },
    render() {
        // clear the canvas, and re-render
        // our line
        Canvas.renderScene.call(this);
    },
};

// set the logical (0,0) origin at this
// physical location on the canvas
Canvas.setOrigin.call(GuideLine,100,100);

GuideLine.setAnchors(-30,65,45,-17);
// [renders line from (-30,65) to (45,-17)
//   on the canvas]
```

That's pretty nice, I think!

But I think another less-obvious benefit is that having objects linked dynamically via `this` context tends to make testing different parts of the program independently, somewhat easier.

For example, `Object.setPrototypeOf(..)` can be used to dynamically change the `[[Prototype]]` linkage of an object, delegating it to a different object such as a mock object. Or you could dynamically redefine `GuideLine.draw()` and `GuideLine.render()` to *explicitly delegate* to a `MockCanvas` instead of `Canvas`.

The `this` keyword, and the `[[Prototype]]` link, are a tremendously flexible mechanism when you understand and leverage them fully.

Why *This*?

OK, so it's hopefully clear that the delegation pattern leans heavily on implicit input, sharing context via `this` rather than through an explicit parameter.

You might rightly ask, why not just always pass around that context explicitly? We can certainly do so, but… to manually pass along the necessary context, we'll have to change pretty much every single function signature, and any corresponding call-sites.

Let's revisit the earlier `ControlPoint` delegation example, and implement it without any delegation-oriented `this` context sharing. Pay careful attention to the differences:

```
var Canvas = {
    setOrigin(ctx,x,y) {
        ctx.translate(x,y);
        ctx.scale(1,-1);
    },
    pixel(ctx,x,y) {
        ctx.fillRect(x,y,1,1);
    },
    renderScene(ctx,entity) {
        // clear the canvas
        var matrix = ctx.getTransform();
        ctx.resetTransform();
        ctx.clearRect(
            0, 0,
            ctx.canvas.width,
            ctx.canvas.height
        );
        ctx.setTransform(matrix);

        entity.draw();
    },
};

var Coordinates = {
    setX(entity,x) {
        entity.x = Math.round(x);
    },
    setY(entity,y) {
        entity.y = Math.round(y);
    },
    setXY(entity,x,y) {
        this.setX(entity,x);
        this.setY(entity,y);
        entity.render();
    },
};

var ControlPoint = {
```

```
    // NOTE: must have a <canvas id="my-canvas">
    // element in the DOM
    ctx: document.getElementById("my-canvas")
        .getContext("2d"),

    setXY(x,y) {
        Coordinates.setXY(this,x,y);
    },
    rotate(angleRadians) {
        var rotatedX = this.x * Math.cos(angleRadians) -
            this.y * Math.sin(angleRadians);
        var rotatedY = this.x * Math.sin(angleRadians) +
            this.y * Math.cos(angleRadians);
        this.setXY(rotatedX,rotatedY);
    },
    draw() {
        // plot the point
        Canvas.pixel(this.ctx,this.x,this.y);
    },
    render() {
        // clear the canvas, and re-render
        // our control-point
        Canvas.renderScene(this.ctx,this);
    },
};

// set the logical (0,0) origin at this
// physical location on the canvas
Canvas.setOrigin(ControlPoint.ctx,100,100);

// ..
```

To be honest, some of you may prefer that style of code. And that's OK if you're in that camp. This snippet avoids `[[Prototype]]` entirely, and only relies on far fewer basic `this.`-style references to properties and methods.

By contrast, the delegation style I'm advocating for in this chapter is unfamiliar and uses `[[Prototype]]` and `this` sharing in ways you're not likely familiar with. To use such a style effectively, you'll have to invest the time and practice to build a deeper familiarity.

But in my opinion, the "cost" of avoiding virtual composition through delegation can be felt across all the function signatures and call-sites; I find them way more cluttered. That explicit context passing is quite a tax.

In fact, I'd never advocate that style of code at all. If you want to avoid delegation, it's probably best to just stick to `class` style code, as seen in Chapter 3. As an exercise left to the reader, try to convert the earlier `ControlPoint` /

`GuideLine` code snippets to use `class`.