YDKJS: THE UNBOOKS

You Don’t Know JS Yet: The Unbooks

Previously Unpublished 2nd-Ed Book Content (Raw & Uncut)

Kyle Simpson

This book is available at
https://leanpub.com/ydkjsy-unbooks

This version was published on 2025-04-24

Tweet This Book!

Please help Kyle Simpson by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I'm reading "YDKJS Yet: The Unbooks", previously unpublished (raw and uncut!) content from the YDKJS 2nd edition series! https://leanpub.com/ydkjsy-unbooks #YDKJSYetUnbooks

The suggested hashtag for this book is #YDKJSYetUnbooks.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#YDKJSYetUnbooks

Published by GetiPub (http://getipub.com), a division of Getify Solutions, Inc., and produced by Leanpub (https://leanpub.com/ydkjsy-unbooks).

April 2025: Second Edition

Revision History for the Second Edition

2020-04-15: First Release

I must first thank my wife and kids, whose constant support is what allows me to keep going. I also want to thank the 500 original backers of the Kickstarter for "You Don't Know JS" (1st ed), as well as the hundreds of thousands of folks who bought and read those books since. Without your financial support, this second edition wouldn't be happening. Thanks also to the interviewer at a certain avian social media company who said I didn't "know enough about JS"... you helped me name the series.

Next, I owe much of my current career path to Marc Grabanski and Frontend Masters. Marc took a chance on me and gave me my first shot at teaching years ago, and I wouldn't have then become a writer had it not been for that! Frontend Masters is the Premier Sponsor of YDKJSY 2nd Edition. Thank you, Frontend Masters (and Marc).

Lastly, my editor, Simon St.Laurent, who helped me conceive the original YDKJS and was my first book editor. Simon's support and guidance have profoundly impacted me and been an integral part of shaping me into the writer I am today. From those drinks we enjoyed at the Driskill all those years back, where YDKJS was born, through today, thank you so much Simon for shepherding and improving these books!

Contents

Preface

This is the unfinished finish line.

You Don’t Know JS Yet: The Unbooks isn’t just the last release in the 2nd edition of the YDKJS series—it’s the odd one, the raw one, the one that was never supposed to exist. But here it is, because even unfinished things can still carry meaning, and sometimes clarity comes not just from what gets published, but from what doesn’t.

The first part of this book contains two fully drafted but previously unpublished titles: Book 3: Objects & Classes and Book 4: Types & Coercion. These were written years ago but never formally edited or finalized for print. For a long time, I wasn’t sure they were worth releasing at all—not because the content isn’t valuable (it is), but because I hold a high bar for what I put in front of you. These chapters are complete in structure and intention, but they’re raw. Unpolished. Unvarnished. You’ll see the uneven edges. That’s by design.

The second part of this book—what I call “The Lost Books”—covers what would have been the final two titles in the series. Book 5: Sync & Async and Book 6: ES.Next & Beyond were never written, and never will be. But that doesn’t mean there’s nothing to say. Instead of stretching them into books they never earned the right to be, I distilled their essence into two reflective chapters. They’re a glimpse into the ideas I wrestled with, and the reasons I ultimately chose not to press on.

If you've made it this far in the series, you're likely already familiar with the tone and mission of YDKJS: to treat JavaScript seriously, to explore it thoroughly, and to challenge you—gently but firmly—to question what you think you know about how it works. That mission hasn't changed here. If anything, this volume sharpens it. It asks not only "What do we teach?" but also "What do we leave out, and why?"

I won't pretend this book follows the same carefully plotted arc as the others. It's less of a textbook and more of a time capsule—part artifact, part epilogue. But in a way, it's the most honest thing I could publish. These are the chapters that didn't fit the mold, the ones I debated over, struggled with, and ultimately chose to release just as they are.

There's something freeing about that. This book is for the curious, the completist, the student who still cares about understanding JavaScript—not just using it. If that's you, I hope this odd little volume feels like a final walk around the edges of the map. Some parts you'll recognize. Some parts will feel unfinished. But you'll see the whole picture a little more clearly for it.

Thanks for walking all this way with me. These Unbooks may not have the final polish—but they do, I hope, still shine.

Objects & Classes (Unbook 3)

Chapter 1: Object Foundations (Objects & Classes)

"Everything in JS is an object."

— most JS developers

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 = {
    // ..
};
```

TIP

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 descr\
iption");
```

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 descr\
iption");

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 go\
ing 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 go\
ing 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, au\
dience!"; }
};
```

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].

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

```
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.\
.."
```

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", "ydk\
js" ]
```

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], ["first\
Name","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(myO\
bj) );

// 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.favoriteNu\
mber : 42
);
const isDev = myObj.isDeveloper;
const firstName = myObj.firstName;
const lname = (
    myObj.lastName !== undefined ? myObj.lastName : "--mi\
ssing--"
);
```

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 destucturing 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.<br><br>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,p\
ropName);
    };
}
```

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.

Chapter 2: How Objects Work (Objects & Classes)

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)[2]. 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.

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

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
// }
```

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 retrievin\
g 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 `functions`.

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 advis\
ed
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, specifically 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 informa\
tion"
```

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)[3]. 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

`[[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

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

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[4], 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.

[4]"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

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".

`[[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 "metaobject protocol" (MOP)[5]. 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.

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

Types & Grammar (Unbook 4)

Chapter 1: Primitive Values (Types & Grammar)

In Chapter 1 of the "Objects & Classes" book of this series, we confronted the common misconception that "everything in JS is an object". We now circle back to that topic, and again dispel that myth.

Here, we'll look at the core value types of JS, specifically the non-object types called *primitives*.

Value Types

JS doesn't apply types to variables or properties – what I call, "container types" – but rather, values themselves have types – what I call, "value types".

The language provides seven built-in, primitive (non-object) value types: [6]

- `undefined`
- `null`
- `boolean`
- `number`

[6] "4.4.5 primitive value", ECMAScript 2022 Language Specification; https://tc39.es/ecma262/#sec-primitive-value ; Accessed August 2022

- `bigint`
- `symbol`
- `string`

These value-types define collections of one or more concrete values, each with a set of shared behaviors for all values of each type.

Type-Of

Any value's value-type can be inspected via the `typeof` operator, which always returns a `string` value representing the underlying JS value-type:

```js
typeof true;            // "boolean"

typeof 42;              // "number"

typeof 42n;             // "bigint"

typeof Symbol("42");    // "symbol"
```

The `typeof` operator, when used against a variable instead of a value, is reporting the value-type of *the value in the variable*:

```js
greeting = "Hello";
typeof greeting;        // "string"
```

JS variables themselves don't have types. They hold any arbitrary value, which itself has a value-type.

Non-objects?

What specifically makes the 7 primitive value types distinct from the object value types (and sub-types)? Why shouldn't we just consider them all as essentially *objects* under the covers?

Consider:

```
myName = "Kyle";

myName.nickname = "getify";

console.log(myName.nickname);           // undefined
```

This snippet appears to silently fail to add a `nickname` property to a primitive string. Taken at face value, that might imply that primitives are really just objects under the covers, as many have (wrongly) asserted over the years.

WARNING

One might explain that silent failure as an example of *auto-boxing* (see "Automatic Objects" in Chapter 3), where the primitive is implicitly converted to a `String` instance wrapper object while attempting to assign the property, and then this internal object is thrown away after the statement completes. In fact, I said exactly that in the first edition of this book. But I was wrong; oops!

Something deeper is at play, as we see in this version of the previous snippet:

```
"use strict";

myName = "Kyle";

myName.nickname = "getify";
// TypeError: Cannot create property 'nickname'
// on string 'Kyle'
```

Interesting! In strict-mode, JS enforces a restriction that disallows setting a new property on a primitive value, as if implicitly promoting it to a new object.

By contrast, in non-strict mode, JS allows the violation to go unmentioned. So why? Because strict-mode was added to the language in ES5.1 (2011), more than 15 years in, and such a change would have broken existing programs had it not been defined as sensitive to the new strict-mode declaration.

So what can we conclude about the distinction between primitives and objects? Primitives are values that *are not allowed to have properties*; only objects are allowed such.

TIP

This particular distinction seems to be contradicted by expressions like `"hello".length`; even in strict-mode, it returns the expected value `5`. So it certainly *seems* like the string has a `length` property! But, as just previously mentioned, the correct explanation is *auto-boxing*; we'll cover the topic in "Automatic Objects" in Chapter 3.

Empty Values

The `null` and `undefined` types both typically represent an emptiness or absence of value.

Unfortunately, the `null` value-type has an unexpected `typeof` result. Instead of `"null"`, we see:

```
typeof null;            // "object"
```

No, that doesn't mean that `null` is somehow a special kind of object. It's just a legacy of early days of JS, which cannot be changed because of how much code out in the wild it would break.

The `undefined` type is reported both for explicit `undefined` values and any place where a seemingly missing value is encountered:

```
typeof undefined;               // "undefined"

var whatever;

typeof whatever;                // "undefined"
typeof nonExistent;             // "undefined"

whatever = {};
typeof whatever.missingProp;    // "undefined"

whatever = [];
typeof whatever[10];            // "undefined"
```

NOTE

The `typeof nonExistent` expression is referring to an undeclared variable `nonExistent`. Normally, accessing an undeclared variable reference would cause an exception, but the `typeof` operator is afforded the special ability to safely access even non-existent identifiers and calmly return `"undefined"` instead of throwing an exception.

However, each respective “empty” type has exactly one value, of the same name. So `null` is the only value in the `null` value-type, and `undefined` is the only value in the `undefined` value-type.

Null’ish

Semantically, `null` and `undefined` types both represent general emptiness, or absence of another affirmative, meaningful value.

NOTE

JS operations which behave the same whether `null` or `undefined` is encountered, are referred to as “null’ish” (or “nullish”). I guess “undefined’ish” would look/sound too weird!

For a lot of JS, especially the code developers write, these two *nullish* values are interchangeable; the decision to intentionally use/assign `null` or `undefined` in any given scenario is situation dependent and left up to the developer.

JS provides a number of capabilities for helping treat the two nullish values as indistinguishable.

For example, the `==` (coercive-equality comparison) operator specifically treats `null` and `undefined` as coercively equal to each other, but to no other values in the language. As such, a `.. == null` check is safe to perform if you want to check if a value is specifically either `null` or `undefined`:

```
if (greeting == null) {
    // greeting is nullish/empty
}
```

Another (recent) addition to JS is the `??` (nullish-coalescing) operator:

```
who = myName ?? "User";

// equivalent to:
who = (myName != null) ? myName : "User";
```

As the ternary equivalent illustrates, `??` checks to see if `myName` is non-nullish, and if so, returns its value. Otherwise, it returns the other operand (here, `"User"`).

Along with `??`, JS also added the `?.` (nullish conditional-chaining) operator:

```js
record = {
    shippingAddress: {
        street: "123 JS Lane",
        city: "Browserville",
        state: "XY"
    }
};

console.log( record?.shippingAddress?.street );
// 123 JS Lane

console.log( record?.billingAddress?.street );
// undefined
```

The `?.` operator checks the value immediately preceding (to the left) value, and if it's nullish, the operator stops and returns an `undefined` value. Otherwise, it performs the `.` property access against that value and continues with the expression.

Just to be clear: `record?.` is saying, "check `record` for nullish before `.` property access". Additionally, `billingAddress?.` is saying, "check `billingAddress` for nullish before `.` property access".

WARNING

Some JS developers believe that the newer `?.` is superior to `.`, and should thus almost always be used instead of `.`. I believe that's an unwise perspective. First of all, it's adding extra visual clutter, which should only be done if you're getting benefit from it. Secondly, you should be aware of, and planning for, the emptiness of some value, to justify using `?.`. If you always expect a non-nullish value to be present in some expression, using `?.` to access a property on it is not only unnecessary/wasteful, but also could potentially hide future bugs where your assumption of value-presence had failed but `?.` covered it up. As with most features in JS, use `.` where it's most appropriate, and use `?.` where it's most appropriate. Never substitute one when the other is more appropriate.

There's also a somewhat strange `?.[` form of the operator, not `?[`, for when you need to use `[ .. ]` style access instead of `.` access:

```
record?.["shipping" + "Address"]?.state;    // XY
```

Yet another variation, referred to as "optional-call", is `?.(`, and is used when conditionally calling a function if the value is non-nullish:

```
// instead of:
//   if (someFunc) someFunc(42);
//
// or:
//   someFunc && someFunc(42);

someFunc?.(42);
```

The `?.(` operator seems like it is checking to see if `someFunc(..)` is a valid function that can be called. But it's not! It's only checking to make sure the value is non-nullish before trying to invoke it. If it's some other non-nullish but also non-function value type, the execution attempt will still fail with a `TypeError` exception.

WARNING

Because of that gotcha, I *strongly dislike* this operator form, and caution anyone against ever using it. I think it's a poorly conceived feature that does more harm (to JS itself, and to programs) than good. There's very few JS features I would go so far as to say, "never use it." But this is one of the truly *bad parts* of the language, in my opinion.

Distinct'ish

It's important to keep in mind that `null` and `undefined` *are* actually distinct types, and thus `null` can be noticeably different from `undefined`. You can, carefully, construct programs that mostly treat them as indistinguishable. But that requires care and discipline by the developer. From JS's perspective, they're more often distinct.

There are cases where `null` and `undefined` will trigger different behavior by the language, which is important to keep in mind. We won't cover all the cases exhaustively here, but here's on example:

```
function greet(msg = "Hello") {
    console.log(msg);
}

greet();                // Hello
greet(undefined);       // Hello
greet("Hi");            // Hi

greet(null);            // null
```

The `= ..` clause on a parameter is referred to as the "parameter default". It only kicks in and assigns its default value to the parameter if the argument in that position is missing, or is exactly the `undefined` value. If you pass `null`, that clause doesn't trigger, and `null` is thus assigned to the parameter.

There's no *right* or *wrong* way to use `null` or `undefined` in a program. So the takeaway is: be careful when choosing one value or the other. And if you're using them interchangeably, be extra careful.

Boolean Values

The `boolean` type contains two values: `false` and `true`.

In the "old days", programming languages would, by convention, use `0` to mean `false` and `1` to mean `true`. So you can think of the `boolean` type, and the keywords `false` and `true`, as a semantic convenience sugar on top of the `0` and `1` values:

```
// isLoggedIn = 1;
isLoggedIn = true;

isComplete = 0;
// isComplete = false;
```

Boolean values are how all decision making happens in a JS program:

```
if (isLoggedIn) {
    // do something
}

while (!isComplete) {
    // keep going
}
```

The `!` operator negates/flips a boolean value to the other one: `false` becomes `true`, and `true` becomes `false`.

String Values

The `string` type contains any value which is a collection of one or more characters, delimited (surrounding on either side) by quote characters:

```
myName = "Kyle";
```

JS does not distinguish a single character as a different type as some languages do; `"a"` is a string just like `"abc"` is.

Strings can be delimited by double-quotes (`"`), single-quotes (`'`), or back-ticks (`` ` ``). The ending delimiter must always match the starting delimiter.

Strings have an intrinsic length which corresponds to how many code-points – actually, code-units, more on that in a bit – they contain.

```
myName = "Kyle";

myName.length;      // 4
```

This does not necessarily correspond to the number of visible characters present between the start and end delimiters (aka, the string literal). It can sometimes be a little confusing to keep straight the difference between a string literal and the underlying string value, so pay close attention.

NOTE

We'll cover length computation of strings in detail, in Chapter 2.

JS Character Encodings

What type of character encoding does JS use for string characters?

You've probably heard of "Unicode" and perhaps even "UTF-8" (8-bit) or "UTF-16" (16-bit). If you're like me (before doing the research it took to write this text), you might have just hand-waved and decided that's all you need to know about character encodings in JS strings.

But… it's not. Not even close.

It turns out, you need to understand how a variety of aspects of Unicode work, and even to consider concepts from UCS-2

(2-byte Universal Character Set), which is similar to UTF-16, but not quite the same. [7]

Unicode defines all the "characters" we can represent universally in computer programs, by assigning a specific number to each, called code-points. These numbers range from `0` all the way up to a maximum of `1114111` (`10FFFF` in hexadecimal).

The standard notation for Unicode characters is `U+` followed by `4`-`6` hexadecimal characters. For example, the `⊠` (heart symbol) is code-point `10084` (`2764` in hexadecimal), and is thus notated with `U+2764`.

The first group of 65,535 code points in Unicode is called the BMP (Basic Multilingual Plane). These can all be represented with 16 bits (2 bytes). When representing Unicode characters from the BMP, it's fairly straightforward, as they can *fit* neatly into single UTF-16 JS characters.

All the rest of the code points are grouped into 16 so called "supplemental planes" or "astral planes". These code-points require more than 16 bits to represent – 21 bits to be exact – so when representing extended/supplemental characters above the BMP, JS actually stores these code-points as a pairing of two adjacent 16-bit code units, called *surrogate halves* (or *surrogate pairs*).

For example, the Unicode code point `127878` (hexadecimal `1F386`) is `⊠` (fireworks symbol). JS stores this in a string value as two surrogate-halve code units: `U+D83C` and `U+DF86`. Keep in mind that these two parts of the whole character do *not* standalone; they're only valid/meaningful when paired immediately adjacent to each other.

[7] "JavaScript's internal character encoding: UCS-2 or UTF-16?"; Mathias Bynens; January 20 2012; https://mathiasbynens.be/notes/javascript-encoding ; Accessed July 2022

This has implications on the length of strings, because a single visible character like the ⊠ fireworks symbol, when in a JS string, is a counted as `2` characters for the purposes of the string length!

We'll revisit Unicode characters in a bit, and then cover the challenges of computing string length in Chapter `2`.

Escape Sequences

If `"` or `'` are used to delimit a string literal, the contents are only parsed for *character-escape sequences*: \ followed by one or more characters that JS recognizes and parses with special meaning. Any other characters in a string that don't parse as escape-sequences (single-character or multi-character), are inserted as-is into the string value.

For single-character escape sequences, the following characters are recognized after a `\`: `b`, `f`, `n`, `r`, `t`, `v`, `0`, `'`, `"`, and `\`. For example, `\n` means new-line, `\t` means tab, etc.

If a `\` is followed by any other character (except `x` and `u` – explained below), like for example `\k`, that sequence is interpreted as the `\` being an unnecessary escape, which is thus dropped, leaving just the literal character itself (`k`).

To include a `"` in the middle of a `"`-delimited string literal, use the `\"` escape sequence. Similarly, if you're including a `'` character in the middle of a `'`-delimited string literal, use the `\'` escape sequence. By contrast, a `'` does *not* need to be escaped inside a `"`-delimited string, nor vice versa.

```
myTitle = "Kyle Simpson (aka, \"getify\"), former O'Reill\
y author";

console.log(myTitle);
// Kyle Simpson (aka, "getify"), former O'Reilly author
```

In text, forward slash `/` is most common. But occasionally, you need a backward slash `\`. To include a literal `\` backslash character without it performing as the start of a character-escape sequence, use the `\\` (double backslashes).

So, then... what would `\\\` (three backslashes) in a string parse as? The first two `\`'s would be a `\\` escape sequence, thereby inserting just a single `\` character in the string value, and the remaining `\` would just escape whatever character comes immediately after it.

One place backslashes show up commonly is in Windows file paths, which use the `\` separator instead of the `/` separator used in linux/unix style paths:

```
windowsFontsPath =
    "C:\\Windows\\Fonts\\";

console.log(windowsFontsPath);
// C:\Windows\Fonts\"
```

TIP

What about four backslashes \\\\ in a string literal? Well, that's just two \\ escape sequences next to each other, so it results in two adjacent backslashes (\\) in the underlying string value. You might recognize there's an odd/even rule pattern at play. You should thus be able to deciper any odd (\\\\\, \\\\\\\\\, etc) or even (\\\\\\, \\\\\\\\\\, etc) number of backslashes in a string literal.

Line Continuation

The \ character followed by an actual new-line character (not just literal `n`) is a special case, and it creates what's called a line-continuation:

```
greeting = "Hello \
Friends!";

console.log(greeting);
// Hello Friends!
```

As you can see, the new-line at the end of the `greeting =` line is immediately preceded by a \, which allows this string literal to continue onto the subsequent line. Without the escaping \ before it, a new-line – the actual new-line, not the `\n` character escape sequence – appearing in a `"` or `'` delimited string literal would actually produce a JS syntax parsing error.

Because the end-of-line \ turns the new-line character into a line continuation, the new-line character is omitted from the string, as shown by the `console.log(..)` output.

NOTE

This line-continuation feature is often referred to as "multi-line strings", but I think that's a confusing label. As you can see, the string value itself doesn't have multiple lines, it only was defined across multiple lines via the line continuations. A multi-line string would actually have multiple lines in the underlying value. We'll revisit this topic later in this chapter when we cover Template Literals.

Multi-Character Escapes

Multi-character escape sequences may be hexadecimal or Unicode sequences.

Hexadecimal escape sequences are used to encode any of the base ASCII characters (codes `0`-`255`), and look like `\x` followed by exactly two hexadecimal characters (`0-9` and `a-f` / `A-F` – case insensitive). For example, `A9` or `a9` are decimal value `169`, which corresponds to:

```
copyright = "\xA9";  // or "\xa9"

console.log(copyright);     // ©
```

For any normal character that can be typed on a keyboard, such as `"a"`, it's usually most readable to just specify the literal character, as opposed to a more obfuscated hexadecimal representation:

```
"a" === "\x61";                // true
```

Unicode In Strings

Unicode escape sequences alone can encode any of the characters from the Unicode BMP. They look like `\u` followed by exactly four hexadecimal characters.

For example, the escape-sequence `\u00A9` (or `\u00a9`) corresponds to that same © symbol, while `\u263A` (or `\u263a`) corresponds to the Unicode character with code-point `9786`: ⊠ (smiley face symbol).

When any character-escape sequence (regardless of length) is recognized, the single character it represents is inserted into the string, rather than the original separate characters. So, in the string `"\u263A"`, there's only one (smiley) character, not six individual characters.

But as explained earlier, many Unicode code-points are well above `65535`. For example, `1F4A9` (or `1f4a9`) is decimal code-point `128169`, which corresponds to the funny ⊠ (pile-of-poo) symbol.

But `\u1F4A9` wouldn't work to include this character in a string, since it would be parsed as the Unicode escape sequence `\u1F4A`, followed by a literal `9` character. To address this limitation, a variation of Unicode escape sequences was introduced to allow an arbitrary number of hexadecimal characters after the `\u`, by surrounding them with `{ .. }` curly braces:

```
myReaction = "\u{1F4A9}";

console.log(myReaction);
// ⊠
```

Recall the earlier discussion of extended (non-BMP) Unicode characters and *surrogate halves*? The same ⊠ could also be defined with two explicit code-units, that form a surrogate pair:

```
myReaction = "\uD83D\uDCA9";

console.log(myReaction);
// ⊠
```

All three representations of this same character are stored internally by JS identically, and are indistinguishable:

```
"⊠" === "\u{1F4A9}";                // true
"\u{1F4A9}" === "\uD83D\uDCA9";     // true
```

Even though JS doesn't care which way such a character is represented in your program, consider the readability differences carefully when authoring your code.

NOTE

Even though ⊠ looks like a single character, its internal representation affects things like the length computation of a string with that character in it. We'll cover length computation of strings in Chapter 2.

Unicode Normalization

Another wrinkle in Unicode string handling is that even certain single BMP characters can be represented in different ways.

For example, the `"é"` character can either be represented as itself (code-point `233`, aka `\xe9` or `\u00e9` or `\u{e9}`), or as the combination of two code-points: the `"e"` character (code-point `101`, aka `\x65`, `\u0065`, `\u{65}`) and the *combining tilde* (code-point `769`, aka `\u0301`, `\u{301}`).

Consider:

```
eTilde1 = "é";
eTilde2 = "\u00e9";
eTilde3 = "\u0065\u0301";

console.log(eTilde1);       // é
console.log(eTilde2);       // é
console.log(eTilde3);       // é
```

The string literal assigned to `eTilde3` in this snippet stores the accent mark as a separate *combining mark* symbol. Like surrogate pairs, a combining mark only makes sense in connection with the symbol it's adjacent to (usually after).

The rendering of the Unicode symbol should be the same regardless, but how the `"é"` character is internally stored affects things like `length` computation of the containing string, as well as equality and relational comparison (more on these in Chapter 2):

```
eTilde1.length;             // 2
eTilde2.length;             // 1
eTilde3.length;             // 2

eTilde1 === eTilde2;        // false
eTilde1 === eTilde3;        // true
```

One particular challenge is that you may copy-paste a string with an `"é"` character visible in it, and that character you copied may have been in the *composed* or *decomposed* form. But there's no visual way to tell, and yet the underlying string value in the literal will be different:

```
"é" === "é";            // false!!
```

This internal representation difference can be quite challenging if not carefully planned for. Fortunately, JS provides a `normalize(..)` utility method on strings to help:

```
eTilde1 = "é";
eTilde2 = "\u{e9}";
eTilde3 = "\u{65}\u{301}";

eTilde1.normalize("NFC") === eTilde2;
eTilde2.normalize("NFD") === eTilde3;
```

The `"NFC"` normalization mode combines adjacent code-points into the *composed* code-point (if possible), whereas the `"NFD"` normalization mode splits a single code-point into its *decomposed* code-points (if possible).

And there can actually be more than two individual *decomposed* code-points that make up a single *composed* code-point – for example, a single character could have several diacritical marks applied to it.

When dealing with Unicode strings that will be compared, sorted, or length analyzed, it's very important to keep Unicode normalization in mind, and use it where necessary.

Unicode Grapheme Clusters

A final complication of Unicode string handling is the support for clustering of multiple adjacent code-points into a single visually distinct symbol, referred to as a *grapheme* (or a *grapheme cluster*).

An example would be a family emoji such as `"⊠⊠⊠⊠"`, which is actually made up of 7 code-points that all cluster/group together into a single visual symbol.

Consider:

```
familyEmoji = "\u{1f469}\u{200d}\u{1f469}\u{200d}\u{1f466\
}\u{200d}\u{1f466}";

familyEmoji;            // ⊠⊠⊠⊠
```

This emoji is *not* a single registered Unicode code-point, and as such, there's no *normalization* that can be performed to compose these 7 separate code-points into a single entity. The visual rendering logic for such composite symbols is quite complex, well beyond what most of JS developers want to embed into our programs. Libraries do exist for handling some of this logic, but they're often large and still don't necessarily cover all of the nuances/variations.

Unlike surrogate pairs and combining marks, the symbols in grapheme clusters can in fact act as standalone characters, but have the special combining behavior when placed adjacent to each other.

This kind of complexity significantly affects length computations, comparison, sorting, and many other common string-oriented operations.

Template Literals

I mentioned earlier that strings can alternately be delimited with `` `..` `` back-ticks:

```
myName = `Kyle`;
```

All the same rules for character encodings, character escape sequences, and lengths apply to these types of strings.

However, the contents of these template (string) literals are additionally parsed for a special delimiter sequence `${ .. }`, which marks an expression to evaluate and interpolate into the string value at that location:

```
myName = `Kyle`;

greeting = `Hello, ${myName}!`;

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

Everything between the `{ .. }` in such a template literal is an arbitrary JS expression. It can be simple variables like `myName`, or complex JS programs, or anything in between (even another template literal expression!).

TIP

This feature is commonly called "template literals" or "template strings", but I think that's confusing. "Template" usually means, in programming contexts, a reusable set of text that can be re-evaluated with different data. For example, *template engines* for pages, email templates for newsletter campaigns, etc. This JS feature is not re-usable. It's a literal, and it produces a single, immediate value (usually a string). You can put such a value in a function, and call the function multiple times. But then the function is acting as the template, not the the literal itself. I prefer instead to refer to this feature as *interpolated literals*, or the funny, short-hand: *interpoliterals*. I just think that name is more accurately descriptive.

Template literals also have an interesting different behavior with respect to new-lines, compared to classic `"` or `'` delimited strings. Recall that for those strings, a line-continuation required a `\` at the end of each line, right before a new-line. Not so, with template literals!

```
myPoem = `
Roses are red
Violets are blue
C3PO's a funny robot
and so R2.`;

console.log(myPoem);
//
// Roses are red
// Violets are blue
// C3PO's a funny robot
// and so R2.
```

Line-continuations with template literals do *not require* escaping. However, that means the new-line is part of the string, even the first new-line above. In other words, `myPoem` above holds a truly *multi-line string*, as shown. However, if you `\` escape the end of any line in a template literal, the new-line will be omitted, just like with non-template literal strings.

Template literals usually result in a string value, but not always. A form of template literal that may look kind of strange is called a *tagged template literal*:

```
price = formatCurrency`The cost is: ${totalCost}`;
```

Here, `formatCurrency` is a tag applied to the template literal value, which actually invokes `formatCurrency(..)` as a function, passing it the string literals and interpolated expressions parsed from the value. This function can then assemble those in any way it sees fit – such as formatting a `number` value as currency in the current locale – and return whatever value, string or otherwise, that it wants.

So tagged template literals are not always strings; they can be any value. But untagged template literals *will always be* strings.

Some JS developers believe that untagged template literal strings are best to use for *all* strings, even if not using any expression interpolation or multiple lines. I disagree. I think they should only be used when interpolating (or multi-line'ing).

TIP

The principle I always apply in making such determinations: use the closest-matched, and least capable, feature/tool, for any task.

Moreover, there are a few places where `` `..` `` style strings are disallowed. For example, the `"use strict"` pragma cannot use back-ticks, or the pragma will be silently ignored (and thus the program accidentally runs in non-strict mode). Also, this style of strings cannot be used in quoted property names of object literals, destruturing patterns, or in the ES Module `import .. from ..` module-specifier clause.

My take: use `` `..` `` delimited strings where allowed, but only when interpolation/multi-line is needed; and keep using `".."` or `'..'` delimited strings for everything else.

Number Values

The `number` type contains any numeric value (whole number or decimal), such as `-42` or `3.1415926`. These values are represented by the JS engine as 64-bit, IEEE-754 double-precision binary floating-point values. [8]

JS `numbers` are always decimals; whole numbers (aka "integers") are not stored in a different/special way. An "integer" stored as a `number` value merely has nothing non-zero as its fraction portion; `42` is thus indistinguishable in JS from `42.0` and `42.000000`.

We can use `Number.isInteger(..)` to determine if a `number` value has any non-zero fraction or not:

[8] "IEEE-754"; https://en.wikipedia.org/wiki/IEEE_754 ; Accessed July 2022

```
Number.isInteger(42);           // true
Number.isInteger(42.0);         // true
Number.isInteger(42.000000);    // true

Number.isInteger(42.0000001);   // false
```

Parsing vs Coercion

If a string value holds numeric-looking contents, you may need to convert from that string value to a `number`, for mathematical operation purposes.

However, it's very important to distinguish between parsing-conversion and coercive-conversion.

We can parse-convert with JS's built-in `parseInt(..)` or `parseFloat(..)` utilities:

```
someNumericText = "123.456";

parseInt(someNumericText,10);               // 123
parseFloat(someNumericText);                // 123.456

parseInt("42",10) === parseFloat("42");     // true

parseInt("512px");                          // 512
```

NOTE

Parsing is only relevant for string values, as it's a character-by-character (left-to-right) operation. It doesn't make sense to parse the contents of a `boolean`, nor to parse the contents of a `number` or a `null`; there's nothing to parse. If you pass anything other than a string value to `parseInt(..)` / `parseFloat(..)`, those utilities first convert that value to a string and then try to parse it. That's almost certainly problematic (leading to bugs) or wasteful – `parseInt(42)` is silly, and `parseInt(42.3)` is an abuse of `parseInt(..)` to do the job of `Math.floor(..)`.

Parsing pulls out numeric-looking characters from the string value, and puts them into a `number` value, stopping once it encounters a character that's non-numeric (e.g., not `-`, `.` or `0-9`). If parsing fails on the first character, both utilities return the special `NaN` value (see "Invalid Number" below), indicating the operation was invalid and failed.

When `parseInt(..)` encounters the `.` in `"123.456"`, it stops, using just the `123` in the resulting `number` value. `parseFloat(..)` by contrast accepts this `.` character, and keeps right on parsing a float with any decimal digits after the `.`.

The `parseInt(..)` utility specifically, takes as an optional – but *actually*, rather necessary – second argument, `radix`: the numeric base to assume for interpreting the string characters for the `number` (range `2 - 36`). `10` is for standard base-10 numbers, `2` is for binary, `8` is for octal, and `16` is for hexadecimal. Any other unusual `radix`, like `23`, assumes digits in order, `0 - 9` followed by the `a - z` (case insensitive) character ordination. If the specified radix is outside the `2 - 36` range, `parseInt(..)` fails as invalid and returns the `NaN` value.

If `radix` is omitted, the behavior of `parseInt(..)` is rather nuanced and confusing, in that it attempts to make a best-guess for a radix, based on what it sees in the first character. This historically has lead to lots of subtle bugs, so never rely on the default auto-guessing; always specify an explicit radix (like `10` in the calls above).

`parseFloat(..)` always parses with a radix of `10`, so no second argument is accepted.

WARNING

One surprising difference between `parseInt(..)` and `parseFloat(..)` is that `parseInt(..)` will not fully parse scientific notation (e.g., `"1.23e+5"`), instead stopping at the `.` as it's not valid for integers; in fact, even `"1e+5"` stops at the `"e"`. `parseFloat(..)` on the other hand fully parses scientific notation as expected.

In contrast to parsing-conversion, coercive-conversion is an all-or-nothing sort of operation. Either the entire contents of the string are recognized as numeric (integer or floating-point), or the whole conversion fails (resulting in `NaN` – again, see "Invalid Number" later in this chapter).

Coercive-conversion can be done explicitly with the `Number(..)` function (no `new` keyword) or with the unary `+` operator in front of the value:

```
someNumericText = "123.456";

Number(someNumericText);        // 123.456
+someNumericText;               // 123.456

Number("512px");                // NaN
+"512px";                       // NaN
```

Other Numeric Representations

In addition to defining numbers using traditional base-10 numerals (`0-9`), JS supports defining whole-number-only number literals in three other bases: binary (base-2), octal (base-8), and hexadecimal (base-16).

```
// binary
myAge = 0b101010;
myAge;                  // 42

// octal
myAge = 0o52;
myAge;                  // 42

// hexadecimal
myAge = 0x2a;
myAge;                  // 42
```

As you can see, the prefixes `0b` (binary), `0o` (octal), and `0x` (hexadecimal) signal defining numbers in the different bases, but decimals are not allowed on these numeric literals.

NOTE

JS syntax allows `0B`, `0O`, and `0X` prefixes as well. However, please don't ever use those uppercase prefix forms. I think any sensible person would agree: `0O` is much easier to confuse at a glance than `0o` (which is, itself, a bit visually ambiguous at a glance). Always stick to the lowercase prefix forms!

It's important to realize that you're not defining a *different number*, just using a different form to produce the same underlying numeric value.

By default, JS represents the underlying numeric value in output/string fashion with standard base-`10` form. However, `number` values have a built-in `toString(..)` method that produces a string representation in any specified base/radix (as with `parseInt(..)`, in the range `2` - `36`):

```
myAge = 42;

myAge.toString(2);          // "101010"
myAge.toString(8);          // "52"
myAge.toString(16);         // "2a"
myAge.toString(23);         // "1j"
myAge.toString(36);         // "16"
```

You can round-trip any arbitrary-radix string representation back into a `number` using `parseInt(..)`, with the appropriate radix:

```
myAge = 42;

parseInt(myAge.toString("23"),23);      // 42
```

Another allowed form for specifying number literals is using scientific notation:

```
myAge = 4.2E1;      // or 4.2e1 or 4.2e+1

myAge;              // 42
```

`4.2E1` (or `4.2e1`) means, `4.2 * (10 ** 1)` (`10` to the `1` power). The exponent can optionally have a sign `+` or `-`. If the sign is omitted, it's assumed to be `+`. A negative exponent makes the number smaller (moves the decimal leftward) rather than larger (moving the decimal rightward):

```
4.2E-3;             // 0.0042
```

This scientific notation form is especially useful for readability when specifying larger powers of `10`:

```
someBigPowerOf10 = 1000000000;

// vs:

someBigPowerOf10 = 1e9;
```

By default, JS will represent (e.g., as string values, etc) either very large or very small numbers – specifically, if the values require more than `21` digits of precision – using this same scientific notation:

```
ratherBigNumber = 123 ** 11;
ratherBigNumber.toString();     // "9.748913698143826e+22"

prettySmallNumber = 123 ** -11;
prettySmallNumber.toString();   // "1.0257553107587752e-2\
3"
```

Numbers with smaller absolute values (closer to `0`) than these thresholds can still be forced into scientific notation form (as strings):

```
plainBoringNumber = 42;

plainBoringNumber.toExponential();      // "4.2e+1"
plainBoringNumber.toExponential(0);     // "4e+1"
plainBoringNumber.toExponential(4);     // "4.2000e+1"
```

The optional argument to `toExponential(..)` specifies the number of decimal digits to include in the string representation.

Another readability affordance for specifying numeric literals in code is the ability to insert `_` as a digit separator wherever its convenient/meaningful to do so. For example:

```
someBigPowerOf10 = 1_000_000_000;

totalCostInPennies = 123_45;  // vs 12_345
```

The decision to use `12345` (no separator), `12_345` (like "12,345"), or `123_45` (like "123.45") is entirely up to the author of the code; JS ignores the separators. But depending on the context, `123_45` could be more semantically meaningful (readability wise) than the more traditional three-digit-grouping-from-the-right-separated-with-commas style mimicked with `12_345`.

IEEE-754 Bitwise Binary Representations

IEEE-754[9] is a technical standard for binary representation of decimal numbers. It's widely used by most computer programming languages, including JS, Python, Ruby, etc.

I'm not going to cover it exhaustively, but I think a brief primer on how numbers work in languages like JS is more than warranted, given how few programmers have *any* familiarity with it.

In 64-bit IEEE-754 – so called "double-precision", because originally IEEE-754 used to be 32-bit, and now it's double that! – the 64 bits are divided into three sections: 52 bits for the number's base value (aka, "fraction", "mantissa", or "significand"), 11 bits for the exponent to raise `2` to before multiplying, and 1 bit for the sign of the ultimate value.

NOTE

Since only 52 of the 64 bits are actually used to represent the base value, `number` doesn't actually have `2^64` values in it. According to the specification for the `number` type[10], the number of values is precisely `2^64 - 2^53 + 3`, or about 18 quintillion, split about evenly between positive and negative numbers.

These bits are arranged left-to-right, as so (S = Sign Bit, E = Exponent Bit, M = Mantissa Bit):

[9] "IEEE-754"; https://en.wikipedia.org/wiki/IEEE_754 ; Accessed July 2022

[10] "6.1.6.1 The Number Type", ECMAScript 2022 Language Specification; https://262.ecma-international.org/13.0/#sec-ecmascript-language-types-number-type ; Accessed August 2022

```
SEEEEEEEEEEEMMMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM
```

So, the number `42` (or `42.000000`) would be represented by these bits:

```
// 42:
01000000010001010000000000000000
00000000000000000000000000000000
```

The sign bit is `0`, meaning the number is positive (`1` means negative).

The `11`-bit exponent is binary `10000000100`, which in base-10 is `1028`. But in IEEE-`754`, this value is interpreted as being stored unsigned with an "exponent bias" of `1023`, meaning that we're shifting up the exponent range from `-1022:1023` to `1:2046` (where `0` and `2047` are reserved for special representations). So, take `1028` and subtract the bias `1023`, which gives an effective exponent of `5`. We raise `2` to that value (`2^5`), giving `32`.

NOTE

If the subtracting `1023` from the exponent value gives a negative (e.g., `-3`), that's still interpreted as `2`'s exponent; raising `2` to negative numbers just produces smaller and smaller values.

The remaining `52` bits give us the base value `01010000...`, interpreted as binary decimal `1.0101000...` (with all trailing zeros). Converting *that* to base-10, we get `1.3125000...`. Finally, then multiply that by `32` already computed from the exponent. The result: `42`.

As you might be able to tell now, this IEEE-754 number representation standard is called "floating point" because the decimal point "floats" back-and-forth along the bits, depending on the specified exponent value.

The number `42.0000001`, which is only different from `42.000000` by just `0.0000001`, would be represented by these bits:

```
// 42.0000001:
01000000010001010000000000000000
00000000110101101011111110010101
```

Notice how the previous bit pattern and this one differ by quite a few bits in the trailing positions! The binary decimal fraction containing all those extra `1` bits (`1.010100000000...01011111110010101`) converts to base-10 as `1.3125000031250000365`2, which multiplied by `32` gives us exactly `42.0000001`.

We'll revisit more details about floating-point (im)precision in Chapter 2. But now you understand a *bit more* about how IEEE-754 works!

Number Limits

As might be evident now that you've seen how IEEE-754 works, the 52 bits of the number's base must be shared, representing both the whole number portion (if any) as well as the decimal portion (if any), of the intended `number` value. Essentially, the larger the whole number portion to be represented, the less bits are available for the decimal portion, and vice versa.

The largest value that can accurately be stored in the `number` type is exposed as `Number.MAX_VALUE`:

```
Number.MAX_VALUE;           // 1.7976931348623157e+308
```

You might expect that value to be a decimal value, given the representation. But on closer inspection, `1.79E308` is (approximately) `2^1024 - 1`. That seems much more like it should be an integer, right? We can verify:

```
Number.isInteger(Number.MAX_VALUE);         // true
```

But what happens if you go above the max value?

```
Number.MAX_VALUE === (Number.MAX_VALUE + 1);
// true -- oops!

Number.MAX_VALUE === (Number.MAX_VALUE + 10000000);
// true
```

So, is `Number.MAX_VALUE` actually the largest value representable in JS? It's certainly the largest *finite* `number` value.

IEEE-754 defines a special infinite value, which JS exposes as `Infinity`; there's also a `-Infinity` at the far other end of the number line. Values can be tested to see if they are finite or infinite:

```
Number.isFinite(Number.MAX_VALUE);  // true

Number.isFinite(Infinity);          // false
Number.isFinite(-Infinity);         // false
```

You can't ever count upwards (with `+ 1`) from `Number.MAX_VALUE` to `Infinity`, no matter how long you let the program run, because the `+ 1` operation isn't actually incrementing beyond the top `Number.MAX_VALUE` value.

However, JS arithmetic operations (`+`, `*`, and even `/`) can definitely overflow the `number` type on the top-end, in which case `Infinity` is the result:

```
Number.MAX_VALUE + 1E291;           // 1.7976931348623157\
e+308
Number.MAX_VALUE + 1E292;           // Infinity

Number.MAX_VALUE * 1.0000000001;    // Infinity

1 / 1E-308;                         // 1e+308
1 / 1E-309;                         // Infinity
```

TIP

The reverse is not true: an arithmetic operation on an infinite value *will never* produce a finite value.

Going from the very large to the very, very small – actually, closest to zero, which is not the same thing as going very, very negative! – the smallest absolute decimal value you could theoretically store in the `number` type would be `2^-1022` (remember the IEEE-754 exponent range?), or around `2E-308`. However, JS engines are allowed by the specification to vary in their internal representations for this lower limit. Whatever the engine's effective lower limit is, it'll be exposed as `Number.MIN_VALUE`:

```
Number.MIN_VALUE;                 // 5e-324 <-- usually!
```

Most JS engines seem to have a minimum representable value around `5E-324` (about `2^-1074`). Depending on the engine and/or platform, a different value may be exposed. Be careful about any program logic that relies on such implementation-dependent values.

Safe Integer Limits

Since `Number.MAX_VALUE` is an integer, you might assume that it's the largest integer in the language. But that's not really accurate.

The largest integer you can accurately store in the `number` type is `2^53 - 1`, or `9007199254740991`, which is *way smaller* than `Number.MAX_VALUE` (about `2^1024 - 1`). This special safer value is exposed as `Number.MAX_SAFE_INTEGER`:

```
maxInt = Number.MAX_SAFE_INTEGER;

maxInt;             // 9007199254740991

maxInt + 1;         // 9007199254740992

maxInt + 2;         // 9007199254740992
```

We've seen that integers larger than `9007199254740991` can show up. However, those larger integers are not "safe", in that the precision/accuracy start to break down when you do operations with them. As shown above, the `maxInt + 1` and `maxInt + 2` expressions both errantly give the same result, illustrating the hazard when exceeding the `Number.MAX_SAFE_INTEGER` limit.

But what's the smallest safe integer?

Depending on how you interpret "smallest", you could either answer `0` or... `Number.MIN_SAFE_INTEGER`:

```
Number.MIN_SAFE_INTEGER;    // -9007199254740991
```

And JS provides a utility to determine if a value is an integer in this safe range (`-2^53 + 1` - `2^53 - 1`):

```
Number.isSafeInteger(2 ** 53);      // false
Number.isSafeInteger(2 ** 53 - 1);  // true
```

Double Zeros

It may surprise you to learn that JS has two zeros: `0`, and `-0` (negative zero). But what on earth is a "negative zero"? [11] A mathematician would surely balk at such a notion.

This isn't just a funny JS quirk; it's mandated by the IEEE-754[12] specification. All floating point numbers are signed, including zero. And though JS does kind of hide the existence of `-0`, it's entirely possible to produce it and to detect it:

[11] "Signed Zero", Wikipedia; https://en.wikipedia.org/wiki/Signed_zero ; Accessed August 2022

[12] "IEEE-754"; https://en.wikipedia.org/wiki/IEEE_754 ; Accessed July 2022

```
function isNegZero(v) {
    return v == 0 && (1 / v) == -Infinity;
}

regZero = 0 / 1;
negZero = 0 / -1;

regZero === negZero;        // true -- oops!
Object.is(-0,regZero);      // false -- phew!
Object.is(-0,negZero);      // true

isNegZero(regZero);         // false
isNegZero(negZero);         // true
```

You may wonder why we'd ever need such a thing as `-0`. It can be useful when using numbers to represent both the magnitude of movement (speed) of some item (like a game character or an animation) and also its direction (e.g., negative = left, positive = right).

Without having a signed zero value, you couldn't tell which direction such an item was pointing at the moment it came to rest.

NOTE

While JS defines a signed zero in the `number` type, there is no corresponding signed zero in the `bigint` number type. As such, `-0n` is just interpreted as `0n`, and the two are indistinguishable.

Invalid Number

Mathematical operations can sometimes produce an invalid result. For example:

```
42 / "Kyle";            // NaN
```

It's probably obvious, but if you try to divide a number by a string, that's an invalid mathematical operation.

Another type of invalid numeric operation is trying to coercively-convert a non-numeric resembling value to a `number`. As discussed earlier, we can do so with either the `Number(..)` function or the unary `+` operator:

```
myAge = Number("just a number");

myAge;                  // NaN

+undefined;             // NaN
```

All such invalid operations (mathematical or coercive/numeric) produce the special `number` value called `NaN`.

The historical root of "NaN" (from the IEEE-754[13] specification) is as an acronym for "Not a Number". Technically, there are about 9 quadrillion values in the 64-bit IEEE-754 number space designated as "NaN", but JS treats all of them indistinguishably as the single `NaN` value.

Unfortunately, that *not a number* meaning produces confusion, since `NaN` is *absolutely* a `number`.

[13] "IEEE-754"; https://en.wikipedia.org/wiki/IEEE_754 ; Accessed July 2022

TIP

Why is `NaN` a `number`?!? Think of the opposite: what if a mathematical/numeric operation, like `+` or `/`, produced a non-`number` value (like `null`, `undefined`, etc)? Wouldn't that be really strange and unexpected? What if they threw exceptions, so that you had to `try..catch` all your math? The only sensible behavior is, numeric/-mathematical operations should *always* produce a `number`, even if that value is invalid because it came from an invalid operation.

To avoid such confusion, I strongly prefer to define "NaN" as any of the following instead:

- "iNvalid Number"
- "Not actual Number"
- "Not available Number"
- "Not applicable Number"

`NaN` is a special value in JS, in that it's the only value in the language that lacks the *identity property* – it's never equal to itself.

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

So unfortunately, the `===` operator cannot check a value to see if it's `NaN`. But there are some ways to do so:

```
politicianIQ = "nothing" / Infinity;

Number.isNaN(politicianIQ);         // true

Object.is(NaN,politicianIQ);        // true
[ NaN ].includes(politicianIQ);     // true
```

Here's a fact of virtually all JS programs, whether you realize it or not: `NaN` happens. Seriously, almost all programs that do any math or numeric conversions are subject to `NaN` showing up.

If you're not properly checking for `NaN` in your programs where you do math or numeric conversions, I can say with some degree of certainty: you probably have a number bug in your program somewhere, and it just hasn't bitten you yet (that you know of!).

WARNING

JS originally provided a global function called `isNaN(..)` for `NaN` checking, but it unfortunately has a long-standing coercion bug. `isNaN("Kyle")` returns `true`, even though the string value `"Kyle"` is most definitely *not* the `NaN` value. This is because the global `isNaN(..)` function forces any non-`number` argument to coerce to a `number` first, before checking for `NaN`. Coercing `"Kyle"` to a `number` produces `NaN`, so now the function sees a `NaN` and returns `true`! This buggy global `isNaN(..)` still exists in JS, but should never be used. When `NaN` checking, always use `Number.isNaN(..)`, `Object.is(..)`, etc.

BigInteger Values

As the maximum safe integer in JS `numbers` is `9007199254740991` (see above), such a relatively low limit can present a problem if a JS program needs to perform larger integer math, or even just hold values like 64-bit integer IDs (e.g., Twitter Tweet IDs).

For that reason, JS provides the alternate `bigint` type (Big-Integer), which can store arbitrarily large (theoretically not limited, except by finite machine memory and/or JS implementation) integers.

To distinguish a `bigint` from a whole (integer) `number` value, which would otherwise both look the same (`42`), JS requires an `n` suffix on `bigint` values:

```
myAge = 42n;        // this is a bigint, not a number

myKidsAge = 11;     // this is a number, not a bigint
```

Let's illustrate the upper un-boundedness of `bigint`:

```
Number.MAX_SAFE_INTEGER;        // 9007199254740991

Number.MAX_SAFE_INTEGER + 2;    // 9007199254740992 -- oo\
ps!

myBigInt = 9007199254740991n;

myBigInt + 2n;                  // 9007199254740993n -- p\
hew!

myBigInt ** 2n;                 // 81129638414606663681390\
0495662081n
```

As you can see, the `bigint` value-type is able to do precise arithmetic above the integer limit of the `number` value-type.

WARNING

Notice that the `+` operator required `.. + 2n` instead of just `.. + 2`? You cannot mix `number` and `bigint` value-types in the same expression. This restriction is annoying, but it protects your program from invalid mathematical operations that would give non-obvious unexpected results.

A `bigint` value can also be created with the `BigInt(..)` function; for example, to convert a whole (integer) `number` value to a `bigint`:

```
myAge = 42n;

inc = 1;

myAge += BigInt(inc);

myAge;              // 43n
```

WARNING

Though it may seem counter-intuitive to some readers, `BigInt(..)` is *always* called without the `new` keyword. If `new` is used, an exception will be thrown.

That's definitely one of the most common usages of the `BigInt(..)` function: to convert `number`s to `bigint`s, for mathematical operation purposes.

But it's not that uncommon to represent large integer values as strings, especially if those values are coming to the JS environment from other language environments, or via certain exchange formats, which themselves do not support `bigint`-style values.

As such, `BigInt(..)` is useful to coerce those string values to `bigint`s:

```
myBigInt = BigInt("12345678901234567890");

myBigInt;                       // 12345678901234567890n
```

Unlike `parseInt(..)`, if any character in the string is non-numeric (`0-9` digits or `-`), including `.` or even a trailing `n` suffix character, an exception will be thrown. In other words, `BigInt(..)` is an all-or-nothing coercion-conversion, not a parsing-conversion.

NOTE

I think it's absurd that `BigInt(..)` won't accept the trailing `n` character while string coercing (and thus effectively ignore it). I lobbied vehemently for that behavior, in the TC39 process, but was ultimately denied. In my opinion, it's now a tiny little gotcha wart on JS, but a wart nonetheless.

Symbol Values

The `symbol` type contains special opaque values called "symbols". These values can only be created by the `Symbol(..)` function:

```js
secret = Symbol("my secret");
```

WARNING

Just as with `BigInt(..)`, the `Symbol(..)` function must be called without the `new` keyword.

The `"my secret"` string passed into the `Symbol(..)` function call is *not* the symbol value itself, even though it seems that way. It's merely an optional descriptive label, used only for debugging purposes for the benefit of the developer.

The underlying value returned from `Symbol(..)` is a special kind of value that resists the program/developer inspecting anything about its underlying representation. That's what I mean by "opaque".

NOTE

You could think of symbols as if they are monotonically incrementing integer numbers – indeed, that's similar to how at least some JS engines implement them. But the JS engine will never expose any representation of a symbol's underlying value in any way that you or the program can see.

Symbols are guaranteed by the JS engine to be unique (only within the program itself), and are unguessable. In other words, a duplicate symbol value can never be created in a program.

You might be wondering at this point what symbols are used for?

One typical usage is as "special" values that the developer distinguishes from any other values that could accidentally collide. For example:

```
EMPTY = Symbol("not set yet");
myNickname = EMPTY;

// later:

if (myNickname == EMPTY) {
    // ..
}
```

Here, I've defined a special `EMPTY` value and initialized `myNickname` to it. Later, I check to see if it's still that special value, and then perform some action if so. I might not want to have used `null` or `undefined` for such purposes, as another developer might be able to pass in one of those common built-in values. `EMPTY` by contrast here is a unique, unguessable value that only I've defined and have control over and access to.

Perhaps even more commonly, symbols are often used as special (meta-) properties on objects:

```
myInfo = {
    name: "Kyle Simpson",
    nickname: "getify",
    age: 42
};

// later:
PRIVATE_ID = Symbol("private unique ID, don't touch!");

myInfo[PRIVATE_ID] = generateID();
```

It's important to note that symbol properties are still publicly visible on any object; they're not *actually* private. But they're treated as special and set-apart from the normal collection of object properties. It's similar to if I had done instead:

```
Object.defineProperty(myInfo,"__private_id_dont_touch",{
    value: generateID(),
    enumerable: false,
});
```

By convention only, most developers know that if a property name is prefixed with _ (or even more so, __!), that means it's "pseudo-private" and to leave it alone unless they're really supposed to access it.

Symbols basically serve the same use-case, but a bit more ergonomically than the prefixing approach.

Well-Known Symbols (WKS)

JS pre-defines a set of symbols, referred to as *well-known symbols* (WKS), that represent certain special meta-programming hooks on objects. These symbols are stored as static properties on the `Symbol` function object. For example:

```
myInfo = {
    // ..
};

String(myInfo);         // [object Object]

myInfo[Symbol.toStringTag] = "my-info";
String(myInfo);         // [object my-info]
```

`Symbol.toStringTag` is a well-known symbol for accessing and overriding the default string representation of a plain object (`"[object Object]"`), replacing the `"Object"` part with a different value (e.g., `"my-info"`).

See the "Objects & Classes" book of this series for more information about Well-Known Symbols and metaprogramming.

Global Symbol Registry

Often, you want to keep symbol values private, such as inside a module scope. But occasionally, you want to expose them so they're accessible globally throughout all the files in a JS program.

Instead of just attaching them as global variables (i.e., properties on the `globalThis` object), JS provides an alternate *global namespace* to register symbols in:

```
// retrieve if already registered,
// otherwise register
PRIVATE_ID = Symbol.for("private-id");

// elsewhere:

privateIDKey = Symbol.keyFor(PRIVATE_ID);
privateIDKey;           // "private-id"

// elsewhere:

// retrieve symbol from registry undeer
// specified key
privateIDSymbol = Symbol.for(privateIDKey);
```

The value passed to `Symbol.for(..)` is *not* the same as passed to `Symbol(..)`. `Symbol.for(..)` expects a unique *key* for the symbol to be registered under in the global registry, whereas `Symbol(..)` optionally accepts a descriptive label (not necessarily unique).

If the registry doesn't have a symbol under that specified *key*, a new symbol (with no descriptive label) is created and automatically registered there. Otherwise, `Symbol.for(..)` returns whatever previously registered symbol is under that *key*.

Going in the opposite direction, if you have the symbol value itself, and want to retrieve the *key* it's registered under, `Symbol.keyFor(..)` takes the symbol itself as input, and returns the *key* (if any). That's useful in case it's more convenient to pass around the *key* string value than the symbol itself.

Object or Primitive?

Unlike other primitives like `42`, where you can create multiple copies of the same value, symbols *do* act more like specific object references in that they're always completely unique (for purposes of value assignment and equality comparison). The specification also categorizes the `Symbol()` function under the "Fundamental Objects" section, calling the function a "constructor", and even defining its `prototype` property.

However, as mentioned earlier, `new` cannot be used with `Symbol(..)`; this is similar to the `BigInt()` "constructor". We clearly know `bigint` values are primitives, so `symbol` values seem to be of the same *kind*.

And in the specification's "Terms and Definitions", it lists symbol as a primitive value. [14] Moreover, the values themselves are used in JS programs as primitives rather than objects. For example, symbols are primarily used as keys in objects – we know objects cannot use other object values as keys! – along with strings, which are also primitives.

As mentioned earlier, some JS engines even internally implement symbols as unique, monotonically incrementing integers (primitives!).

Finally, as explained at the top of this chapter, we know primitive values are *not allowed* to have properties set on them, but are *auto-boxed* (see "Automatic Objects" in Chapter 3) internally to the corresponding object-wrapper type to facilitate property/method access. Symbols follow all these exact behaviors, the same as all the other primitives.

All this considered, I think symbols are *much more* like

[14] "4.4.5 primitive value", ECMAScript 2022 Language Specification; https://tc39.es/ecma262/#sec-primitive-value ; Accessed August 2022

primitives than objects, so that's how I present them in this book.

Primitives Are Built-In Types

We've now dug deeply into the seven primitive (non-object) value types that JS provides automatically built-in.

Before we move on to discussing JS's built-in object value type, we want to take a closer look at the kinds of behaviors we can expect from JS values. We'll do so in-depth, in the next chapter.

Chapter 2: Primitive Behaviors (Types & Grammar)

So far, we've explored seven built-in primitive value types in JS: `null`, `undefined`, `boolean`, `string`, `number`, `bigint`, and `symbol`.

Chapter 1 was quite a lot to take in, much more involved than I bet most readers expected. If you're still catching your breath after reading all that, don't worry about taking a bit of a break before continuing on here!

Once you're clear headed and ready to move on, let's dig into certain behaviors implied by value types for all their respective values. We'll take a careful and closer look at all of these various behaviors.

Primitive Immutability

All primitive values are immutable, meaning nothing in a JS program can reach into the contents of the value and modify it in any way.

```
myAge = 42;

// later:

myAge = 43;
```

The `myAge = 43` statement doesn't change the value. It reassigns a different value `43` to `myAge`, completely replacing the previous value of `42`.

New values are also created through various operations, but again these do not modify the original value:

```
42 + 1;             // 43

"Hello" + "!";      // "Hello!"
```

The values `43` and `"Hello!"` are new, distinct values from the previous `42` and `"Hello"` values, respectively.

Even a string value, which looks like merely an array of characters -- and array contents are typically mutable -- is immutable:

```
greeting = "Hello.";

greeting[5] = "!";

console.log(greeting);      // Hello.
```

WARNING

In non-strict mode, assigning to a read-only property (like `greeting[5] = ..`) silently fails. In strict-mode, the disallowed assignment will throw an exception.

The nature of primitive values being immutable is not affected *in any way* by how the variable or object property holding the value is declared. For example, whether `const`, `let`, or `var` are used to declare the `greeting` variable above, the string value it holds is immutable.

`const` doesn't create immutable values, it declares variables that cannot be reassigned (aka, immutable assignments) – see the "Scope & Closures" title of this series for more information.

A property on an object may be marked as read-only – with the `writable: false` descriptor attribute, as discussed in the "Objects & Classes" title of this series. But that still has no affect on the nature of the value, only on preventing the reassignment of the property.

Primitives With Properties?

Additionally, properties *cannot* be added to any primitive values:

```
greeting = "Hello.";

greeting.isRendered = true;

greeting.isRendered;        // undefined
```

This snippet looks like it's adding a property `isRendered` to the value in `greeting`, but this assignment silently fails (even in strict-mode).

Property access is not allowed in any way on nullish primitive values `null` and `undefined`. But properties *can* be accessed on all other primitive values – yes, that sounds counter-intuitive.

For example, all string values have a read-only `length` property:

```
greeting = "Hello.";

greeting.length;            // 6
```

`length` can not be set, but it can be accesses, and it exposes the number of code-units stored in the value (see "JS Character Encodings" in Chapter 1), which often means the number of characters in the string.

NOTE

Sort of. For most standard characters, that's true; one character is one code-point, which is one code-unit. However, as explained in Chapter 1, extended Unicode characters above code-point `65535` will be stored as two code-units (surrogate halves). Thus, for each such character, `length` will include `2` in its count, even though the character visually prints as one symbol.

Non-nullish primitive values also have a couple of standard built-in methods that can be accessed:

```js
greeting = "Hello.";

greeting.toString();    // "Hello." <-- redundant
greeting.valueOf();     // "Hello."
```

Additionally, most of the primitive value-types define their own methods with specific behaviors inherent to that type. We'll cover these later in this chapter.

NOTE

As already briefly mentioned in Chapter 1, technically, these sorts of property/method accesses on primitive values are facilitated by an implicit coercive behavior called *auto-boxing*. We'll cover this in detail in "Automatic Objects" in Chapter 3.

Primitive Assignments

Any assignment of a primitive value from one variable/container to another is a *value-copy*:

```js
myAge = 42;

yourAge = myAge;        // assigned by value-copy

myAge;                  // 42
yourAge;                // 42
```

Here, the `myAge` and `yourAge` variables each have their own copy of the number value `42`.

NOTE

Inside the JS engine, it *may* be the case that only one `42` value exists in memory, and the engine points both `myAge` and `yourAge` variables at the shared value. Since primitive values are immutable, there's no danger in a JS engine doing so. But what's important to us as JS developers is, in our programs, `myAge` and `yourAge` act as if they have their own copy of that value, rather than sharing it.

If we later reassign `myAge` to `43` (when I have a birthday), it doesn't affect the `42` that's still assigned to `yourAge`:

```
myAge++;            // sort of like: myAge = myAge + 1

myAge;              // 43
yourAge;            // 42 <-- unchanged
```

String Behaviors

String values have a number of specific behaviors that every JS developer should be aware of.

String Character Access

Though strings are not actually arrays, JS allows `[ .. ]` array-style access of a character at a numeric (`0`-based) index:

```
greeting = "Hello!";

greeting[4];            // "o"
```

If the value/expression between the `[ .. ]` doesn't resolve to a number, the value will be implicitly coerced to its whole/integer numeric representation (if possible).

```
greeting["4"];          // "o"
```

If the value/expression resolves to a number outside the integer range of `0 - length - 1` (or `NaN`), or if it's not a `number` value-type, the access will instead be treated as a property access with the string equivalent property name. If the property access thus fails, the result is `undefined`.

NOTE

We'll cover coercion in-depth later in the book.

Character Iteration

Strings are not arrays, but they certainly mimic arrays closely in many ways. One such behavior is that, like arrays, strings are iterables. This means that the characters (code-units) of a string can be iterated individually:

```
myName = "Kyle";

for (let char of myName) {
    console.log(char);
}
// K
// y
// l
// e

chars = [ ...myName ];
chars;
// [ "K", "y", "l", "e" ]
```

Values, such as strings and arrays, are iterables (via `...`, `for..of`, and `Array.from(..)`), if they expose an iterator-producing method at the special symbol property location `Symbol.iterator` (see "Well-Known Symbols" in Chapter 1):

```
myName = "Kyle";
it = myName[Symbol.iterator]();

it.next();      // { value: "K", done: false }
it.next();      // { value: "y", done: false }
it.next();      // { value: "l", done: false }
it.next();      // { value: "e", done: false }
it.next();      // { value: undefined, done: true }
```

NOTE

The specifics of the iterator protocol, including the fact that the `{ value: "e" .. }` result still shows `done: false`, are covered in detail in the "Sync & Async" title of this series.

Length Computation

As mentioned in Chapter 1, string values have a `length` property that automatically exposes the length of the string; this property can only be accessed; attempts to set it are silently ignored.

The reported `length` value somewhat corresponds to the number of characters in the string (actually, code-units), but as we saw in Chapter 1, it's more complex when Unicode characters are involved.

Most people visually distinguish symbols as separate characters; this notion of an independent visual symbol is referred to as a *grapheme*, or a *grapheme cluster*. So when counting the "length" of a string, we typically mean that we're counting the number of graphemes.

But that's not how the computer deals with characters.

In JS, each *character* is a code-unit (16 bits), with a code-point value at or below `65535`. The `length` property of a string always counts the number of code-units in the string value, not code-points. A code-unit might represent a single character by itself, or it may be part of a surrogate pair, or it may be combined with an adjacent *combining* symbol, or part of a grapheme cluster. As such, `length` doesn't match the typical notion of counting visual characters/graphemes.

To get closer to an expected/intuitive *grapheme length* for a string, the string value first needs to be normalized with `normalize("NFC")` (see "Normalizing Unicode" in Chapter 1) to produce any *composed* code-units (where possible), in case any characters were originally stored *decomposed* as separate code-units.

For example:

```
favoriteItem = "teléfono";
favoriteItem.length;            // 9 -- uh oh!

favoriteItem = favoriteItem.normalize("NFC");
favoriteItem.length;            // 8 -- phew!
```

Unfortunately, as we saw in Chapter 1, we'll still have the possibility of characters of code-point greater the `65535`, and thus needing a surrogate pair to be represented. Such characters will count double in the `length`:

```
// "⊠" === "\u260E"
oldTelephone = "⊠";
oldTelephone.length;            // 1

// "⊠" === "\u{1F4F1}" === "\uD83D\uDCF1"
cellphone = "⊠";
cellphone.length;               // 2 -- oops!
```

So what do we do?

One fix is to use character iteration (via `...` operator) as we saw in the previous section, since it automatically returns each combined character from a surrogate pair:

```
cellphone = "⊠";
cellphone.length;               // 2 -- oops!
[ ...cellphone ].length;        // 1 -- phew!
```

But, unfortunately, grapheme clusters (as explained in Chapter 1) throw yet another wrench into a string's length computation. For example, if we take the thumbs down emoji (`"\u{1F44E}"` and add to it the skin-tone modifier for medium-dark skin (`"\u{1F3FE}"`), we get:

```
// "⊠⊠" = "\u{1F44E}\u{1F3FE}"
thumbsDown = "⊠⊠";

thumbsDown.length;              // 4 -- oops!
[ ...thumbsDown ].length;       // 2 -- oops!
```

As you can see, these are two distinct code-points (not a surrogate pair) that, by virtue of their ordering and adjacency, cause the computer's Unicode rendering to draw the thumbs-down symbol but with a darker skin tone than its default. The computed string length is thus `2`.

It would take replicating most of a platform's complex Unicode rendering logic to be able to recognize such clusters of code-points as a single "character" for length-counting sake. There are libraries that purport to do so, but they're not necessarily perfect, and they come at a hefty cost in terms of extra code.

NOTE

As a Twitter user, you might expect to be able to put 280 thumbs-down emojis into a single tweet, since it looks like a single character. Twitter counts the `"⊠"` (default thumbs-down), the `"⊠⊠"` (medium-dark-skintone thumbs-down), and even the `"⊠⊠⊠⊠"` (family emoji grapheme cluster) all as 2 characters each, even though their respective string lengths (from JS's perspective) are `2`, `4`, and `7`; thus, you can only fit half the number of emojis (140 instead of 280) in a tweet. In fact, Twitter implemented this change in 2018 to specifically level the counting of all Unicode characters, at 2 characters per symbol. [15] That was a welcomed change for Twitter users, especially those who want to use emoji characters that are most representative of intended gender, skintone, etc. Still, it *is* curious that Twitter chose to count all Unicode/emoji symbols as 2 characters each, instead of the more intuitive 1 character (grapheme) each.

Counting the *length* of a string to match our human intuitions is a remarkably challenging task, perhaps more of an art than a science. We can get acceptable approximations in many cases, but there's plenty of other cases that may confound our programs.

[15]"New update to the Twitter-Text library: Emoji character count"; Andy Piper; Oct 2018; https://twittercommunity.com/t/new-update-to-the-twitter-text-library-emoji-character-count/114607 ; Accessed July 2022

Internationalization (i18n) and Localization (l10n)

To serve the growing need for JS programs to operate as expected in any international language/culture context, the ECMAScript committee also publishes the ECMAScript Internationalization API. [16]

A JS program defaults to a locale/language according to the environment running the program (web browser page, Node instance, etc). The in-effect locale affects sorting (and value comparisons), formatting, and several other assumed behaviors. Such altered behaviors are perhaps a bit more obvious with strings, but they can also be seen with numbers (and dates!).

But string characters also can have language/locale information embedded in them, which takes precedence over the environment default. If the string character is ambiguous/shared in terms of its language/locale (such as `"a"`), the default environment setting is used.

Depending on the contents of the string, it may be interpreted as being ordered from left-to-right (LTR) or right-to-left (RTL). As such, many of the string methods we'll cover later use logical descriptors in their names, like "start", "end", "begin", "end", and "last", rather than directional terms like "left" and "right".

For example, Hebrew and Arabic are both common RTL languages:

[16] ECMAScript 2022 Internationalization API Specification; https://402.ecma-international.org/9.0/ ; Accessed August 2022

```
hebrewHello = "\u{5e9}\u{5dc}\u{5d5}\u{5dd}";

console.log(hebrewHello);                     // ⊠⊠⊠⊠
```

Notice that the first listed character in the string literal (`"\u{5e9}"`) is actually the right-most character when the string is rendered?

Even though Hebrew is an RTL language, you don't actually type the characters in the string literal in reversed (RTL) order the way they should be rendered. You enter the characters in logical order, where position `0` is the first character, position `1` is the second character, etc. The rendering layer is where RTL characters are reversed to be shown in their correct order.

That also means that if you access `hebrewHello[0]` (or `hebrewHello.charAt(0)`) – to get the character as position `0` – you get `"⊠"` because that's logically the first character of the string, not `"⊠"` (logically the last character of the string). Index-positional access follows the logical position, not the rendered position.

Here's the same example in another RTL language, Arabic:

```
arabicHello = "\u{631}\u{62d}\u{628}\u{627}";

console.log(arabicHello);                     // ⊠⊠⊠⊠

console.log(arabicHello[0]);                  // ⊠
```

JS programs can force a specific language/locale, using various `Intl` APIs such as `Intl.Collator`: [17]

[17] "Intl.Collator", MDN; https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Intl/Collator ; Accessed August 2022

```js
germanStringSorter = new Intl.Collator("de");

listOfGermanWords = [ /* .. */ ];

germanStringSorter.compare("Hallo","Welt");
// -1 (or negative number)

// examples adapted from MDN:
//
germanStringSorter.compare("Z","z");
// 1 (or positive number)

caseFirstSorter = new Intl.Collator("de",{ caseFirst: "up\
per", });
caseFirstSorter.compare("Z","z");
// -1 (or negative number)
```

Multiple-word strings can be segmented using `Intl.Segmenter`:[18]

```js
arabicHelloWorld = "\u{645}\u{631}\u{62d}\u{628}\u{627} \
\u{628}\u{627}\u{644}\u{639}\u{627}\u{644}\u{645}";

console.log(arabicHelloWorld);      // ⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠

arabicSegmenter = new Intl.Segmenter("ar",{ granularity: \
"word" });

for (
    let { segment: word, isWordLike } of
    arabicSegmenter.segment(arabicHelloWorld)
) {
    if (isWordLike) {
```

[18]"Intl.Segmenter", MDN; https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Intl/Segmenter ; Accessed August 2022

```
        console.log(word);
    }
}
// ⊠⊠⊠⊠⊠
⊠⊠⊠⊠⊠//
```

NOTE

The `segment(..)` method (from instances of`Intl.Segmenter`) returns a standard JS iterator, which the `for..of` loop here consumes. More on iteration protocols in the “Sync & Async” title of this series.

String Comparison

String values can be compared (for both equality and relational ordering) to other string values, using various built-in operators. It’s important to keep in mind that such comparisons are sensitive to the actual string contents, including especially the underlying code-points from non-BPM Unicode characters.

Both equality and relational comparison are case-sensitive, for any characters where uppercase and lowercase are well-defined. To make case-insensitive comparisons, normalize the casing of both values first (with `toUpperCase()` or `toLowerCase()`).

String Equality

The `===` and `==` operators (along with their negated counterparts `!==` and `!=`, respectively) are the most common way equality comparisons are made for primitive values, including string values:

```
"my name" === "my n\x61me";                // true

"my name" !== String.raw`my n\x61me`;      // true
```

The `===` operator[19] – often referred to as "strict equality" – first checks to see if the types match, and if not, returns `false` right away. If the types match, then it checks to see if the values are the same; for strings, this is a per-code-unit comparison, from start to end.

Despite the "strict" naming, there are nuances to `===` (such as `-0` and `NaN` handling), but we'll cover those later.

Coercive Equality

By contrast, the `==` operator[20] – often referred to as "loose equality" – performs *coercive equality*: if the value-types of the two operands do not match, `==` first coerces one or both operands until the value-types *do* match, and then it hands off the comparison internally to `===`.

Coercion is an extremely important topic – it's an inherent part of the JS types system, one of the language's 3 pillars – but we're only going to briefly introduce it here in this chapter, and revisit it in detail later.

[19]"7.2.16 IsStrictlyEqual(x,y)", ECMAScript 2022 Language Specification; https://262.ecma-international.org/13.0/#sec-isstrictlyequal ; Accessed August 2022

[20]"7.2.15 IsLooselyEqual(x,y)", ECMAScript 2022 Language Specification; https://262.ecma-international.org/13.0/#sec-islooselyequal ; Accessed August 2022

NOTE

You may have heard the oft-quoted, but nevertheless inaccurate, explanation that the difference between == and === is that == compares the values while == compares both the values and the types. Not true, and you can read the spec yourself to verify – both `isStrictlyEqual(..)` and `isLooselyEqual(..)` specification algorithms are linked as footnotes in the preceding paragraphs. To summarize, though: both == and === are aware of and sensitive to the types of the operands. If the operand types are the same, both operators do literally the exact same thing; if the types differ, == forces coercion until the types match, whereas === returns `false` immediately.

It's extremely common for developers to assert that the == operator is confusing and too hard to use without surprises (thus the near universal preference for ===). I think that's totally bogus, and in fact, JS developers should be defaulting to == (and avoiding === if possible). But we need a lot more discussion to back such a controversial statement; hold onto your objections until we revisit it later.

For now, to gain some intuition about the coercive nature of ==, the most illuminating observation is that if the types don't match, == *prefers* numeric comparison. That means it will attempt to convert both operands to numbers, and then perform the equality check (the same as ===).

So, as it relates to our present discussion, actual string equality can *only be* checked if both operands are already strings:

```
// actual string equality check (via === internally):
"42" == "42";           // true
```

== does not really perform string equality checks itself. If the operand value-types are both strings, == just hands off the comparison to ===. If they're not both strings, the coercive steps in == will reduce the comparison matching to numeric instead of string:

```
// numeric (not string!) equality check:
42 == "42";             // true
```

We'll cover numeric equality later in this chapter.

Really Strict Equality

In addition to == and ===, JS provides the `Object.is(..)` utility, which returns `true` if both arguments are *exactly identical*, and `false` otherwise (no exceptions or nuances):

```
Object.is("42",42);             // false

Object.is("42","\x34\x32");     // true
```

Since === adds a = onto the end of == to make it more strict in behavior, I kind of half-joke that the `Object.is(..)` utility is like a ==== (a fourth = added) operator, for the really-truly-strict-no-exceptions kind of equality checking!

That said, === (and == by virtue of its internal delegation to ===) are *extremely predictable*, with no weird exceptions, when it comes to comparing two actually-already-string values. I strongly recommend using == for such checks (or ===), and reserve `Object.is(..)` for the corner cases (which are numeric).

String Relational Comparisons

In addition to equality checks between strings, JS supports relational comparisons between primitive values, like strings: <, <=, >, and >=.

The < (less-than) and > (greater-than) operations compare two string values lexicographically – like you would sort words in a dictionary – and should thus be fairly self explanatory:

```
"hello" < "world";          // true
```

NOTE

As mentioned earlier, the running JS program has a default locale, and these operators compare according to that locale.

Like ==, the < and > operators are numerically coercive. Any non-number values are coerced to numbers. So the only way to do a relational comparison with strings is to ensure both operands are already string values.

Perhaps somewhat surprisingly, the < and > have no strict-comparison equivalent, the way === avoids the coercion of ==. These operators are always coercive (when the types don't match), and there's no way in JS to avoid that.

So what happens when both values are *numeric-looking* strings?

```
"100" < "11";               // true
```

Numerically, of course, `100` should *not be* less than `11`.

But relational comparisons between two strings use the lexicographic ordering. So the second `"0"` character (in `"100"`) is less than the second `"1"` (in `"11"`), and thus `"100"` would be sorted in a *dictionary* before `"11"`. The relational operators only coerce to numbers if the operand types are not already strings.

The `<=` (less-than-or-equal) and `>=` (greater-than-or-equal) operators are effectively a shorthand for a compound check.

```
"hello" <= "hello";                             // true
("hello" < "hello") || ("hello" == "hello");    // true

"hello" >= "hello";                             // true
("hello" > "hello") || ("hello" == "hello");    // true
```

NOTE

Here's an interesting bit of specification nuance: JS doesn't actually define the underlying greater-than (for >) or greater-than-or-equal (for >=) operations. Instead, it defines them by reversing the arguments to their *less-than* complement counterparts. So `x > y` is treated by JS essentially as `y <= x`, and `x >= y` is treated by JS essentially as `y < x`. So JS only needs to specify how < and == work, and thus gets > and >= for free!

Locale-Aware Relational Comparisons

As I mentioned a moment ago, the relational operators assume and use the current in-effect locale. However, it can sometimes be useful to force a specific locale for comparisons (such as when sorting a list of strings).

JS provides the method `localCompare(..)` on JS strings for this purpose:

```
"hello".localeCompare("world");
// -1 (or negative number)

"world".localeCompare("hello","en");
// 1 (or positive number)

"hello".localeCompare("hello","en",{ ignorePunctuation: t\
rue });
// 0

// examples from MDN:
//
// in German, ä sorts before z
"ä".localeCompare("z","de");
// -1 (or negative number) // a negative value

// in Swedish, ä sorts after z
"ä".localeCompare("z","sv");
// 1 (or positive number)
```

The optional second and third arguments to `localeCompare(..)` control which locale to use, via the `Intl.Collator` API[^INTLCollatorApi], as covered earlier.

You might use `localeCompare(..)` when sorting an array of strings:

```
studentNames = [
    "Lisa",
    "Kyle",
    "Jason"
];

// Array::sort() mutates the array in place
studentNames.sort(function alphabetizeNames(name1,name2){
    return name1.localeCompare(name2);
});

studentNames;
// [ "Jason", "Kyle", "Lisa" ]
```

But as discussed earlier, a more straightforward way (and slightly more performant when sorting many strings) is using `Intl.Collator` directly:

```
studentNames = [
    "Lisa",
    "Kyle",
    "Jason"
];

nameSorter = new Intl.Collator("en");

// Array::sort() mutates the array in place
studentNames.sort(nameSorter.compare);

studentNames;
// [ "Jason", "Kyle", "Lisa" ]
```

String Concatenation

Two or more string values can be concatenated (combined) into a new string value, using the + operator:

```
greeting = "Hello, " + "Kyle!";

greeting;                 // Hello, Kyle!
```

The + operator will act as a string concatenation if either of the two operands (values on left or right sides of the operator) are already a string (even an empty string `""`).

If one operand is a string and the other is not, the one that's not a string will be coerced to its string representation for the purposes of the concatenation:

```
userCount = 7;

status = "There are " + userCount + " users online";

status;         // There are 7 users online
```

String concatenation of this sort is essentially interpolation of data into the string, which is the main purpose of template literals (see Chapter 1). So the following code will have the same outcome but is generally considered to be the more preferred approach:

```
userCount = 7;

status = `There are ${userCount} users online`;

status;         // There are 7 users online
```

Other options for string concatenation include `"one".concat("two","three")` and `[ "one", "two", "three" ].join("")`, but these kinds of approaches are only preferable when the number of strings to concatenate is dependent on runtime conditions/computation. If the string has a fixed/known set of content, as above, template literals are the better option.

String Value Methods

String values provide a whole slew of additional string-specific methods (as properties):

- `charAt(..)`: produces a new string value at the numeric index, similar to `[ .. ]`; unlike `[ .. ]`, the result is always a string, either the character at position `0` (if a valid number outside the indices range), or the empty string `""` (if missing/invalid index)
- `at(..)` is similar to `charAt(..)`, but negative indices count backwards from the end of the string
- `charCodeAt(..)`: returns the numeric code-unit (see “JS Character Encodings” in Chapter 1) at the specified index
- `codePointAt(..)`: returns the whole code-point starting at the specified index; if a surrogate pair is found there, the whole character (code-point) s returned
- `substr(..)` / `substring(..)` / `slice(..)`: produces a new string value that represents a range of characters from the original string; these differ in how the range's start/end indices are specified or determined
- `toUpperCase()`: produces a new string value that's all uppercase characters
- `toLowerCase()`: produces a new string value that's all lowercase characters
- `toLocaleUpperCase()` / `toLocaleLowerCase()`: uses locale mappings for uppercase or lowercase operations

- `concat(..)`: produces a new string value that's the concatenation of the original string and all of the string value arguments passed in
- `indexOf(..)`: searches for a string value argument in the original string, optionally starting from the position specified in the second argument; returns the `0`-based index position if found, or `-1` if not found
- `lastIndexOf(..)`: like `indexOf(..)` but, from the end of the string (right in LTR locales, left in RTL locales)
- `includes(..)`: similar to `indexOf(..)` but returns a boolean result
- `search(..)`: similar to `indexOf(..)` but with a regular-expression matching as specified
- `trimStart()` / `trimEnd()` / `trim()`: produces a new string value with whitespace trimmed from the start of the string (left in LTR locales, right in RTL locales), or the end of the string (right in LTR locales, left in RTL locales), or both
- `repeat(..)`: produces a new string with the original string value repeated the specified number of times
- `split(..)`: produces an array of string values as split at the specified string or regular-expression boundaries
- `padStart(..)` / `padEnd(..)`: produces a new string value with padding (default " " whitespace, but can be overridden) applied to either the start (left in LTR locales, right in RTL locales) or the end (right in LTR locales), left in RTL locales), so that the final string result is at least of a specified length
- `startsWith(..)` / `endsWith(..)`: checks either the start (left in LTR locales, right in RTL locales) or the end (right in LTR locales) of the original string for the string value argument; returns a boolean result

- `match(..)` / `matchAll(..)`: returns an array-like regular-expression matching result against the original string
- `replace(..)`: returns a new string with a replacement from the original string, of one or more matching occurrences of the specified regular-expression match
- `normalize(..)`: produces a new string with Unicode normalization (see "Unicode Normalization" in Chapter 1) having been performed on the contents
- `localCompare(..)`: function that compares two strings according to the current locale (useful for sorting); returns a negative number (usually `-1` but not guaranteed) if the original string value is comes before the argument string value lexicographically, a positive number (usually `1` but not guaranteed) if the original string value comes after the argument string value lexicographically, and `0` if the two strings are identical
- `anchor()`, `big()`, `blink()`, `bold()`, `fixed()`, `fontcolor()`, `fontsize()`, `italics()`, `link()`, `small()`, `strike()`, `sub()`, and `sup()`: historically, these were useful in generating HTML string snippets; they're now deprecated and should be avoided

WARNING

Many of the methods described above rely on position indices. As mentioned earlier in the "Length Computation" section, these positions are dependent on the internal contents of the string value, which means that if an extended Unicode character is present and takes up two code-unit slots, that will count as two index positions instead of one. Failing to account for *decomposed* code-units, surrogate pairs, and grapheme cluseters is a common source of bugs in JS string handling.

These string methods can all be called directly on a literal value, or on a variable/property that's holding a string value. When applicable, they produce a new string value rather than modifying the existing string value (since strings are immutable):

```
"all these letters".toUpperCase();      // ALL THESE LETT\
ERS

greeting = "Hello!";
greeting.repeat(2);                     // Hello!Hello!
greeting;                               // Hello!
```

Static `String` Helpers

The following string utility functions are provided directly on the `String` object, rather than as methods on individual string values:

* `String.fromCharCode(..)` / `String.fromCodePoint(..)`: produce a string from one or more arguments representing the code-units (`fromCharCode(..)`) or whole code-points (`fromCodePoint(..)`)
* `String.raw(..)`: a default template-tag function that allows interpolation on a template literal but prevents character escape sequences from being parsed, so they remain in their *raw* individual input characters from the literal

Moreover, most values (especially primitives) can be explicitly coerced to their string equivalent by passing them to the `String(..)` function (no `new` keyword). For example:

```
String(true);           // "true"
String(42);             // "42"
String(Infinity);       // "Infinity"
String(undefined);      // "undefined"
```

We'll cover much more detail about such type coercions in a later chapter.

Number Behaviors

Numbers are used for a variety of tasks in our programs, but mostly for mathematical computations. Pay close attention to how JS numbers behave, to ensure the outcomes are as expected.

Floating Point Imprecision

We need to revisit our discussion of IEEE-754 from Chapter 1.

One of the classic gotchas of any IEEE-754 number system in any programming language – NOT UNIQUELY JS! – is that not all operations and values can fit neatly into the IEEE-754 representations.

The most common illustration is:

```
point3a = 0.1 + 0.2;
point3b = 0.3;

point3a;                        // 0.30000000000000004
point3b;                        // 0.3

point3a === point3b;            // false <-- oops!
```

The operation `0.1 + 0.2` ends up creating floating-point error (drift), where the value stored is actually `0.30000000000000004`.

The respective bit representations are:

```
// 0.30000000000000004
00111111110100110011001100110011
00110011001100110011001100110100

// 0.3
00111111110100110011001100110011
00110011001100110011001100110011
```

If you look closely at those bit patterns, only the last 2 bits differ, from `00` to `11`. But that's enough for those two numbers to be unequal!

Again, just to reinforce: this behavior is **NOT IN ANY WAY** unique to JS. This is exactly how any IEEE-754 conforming programming language will work in the same scenario. As

I asserted above, the majority of all programming languages use IEEE-754, and thus they will all suffer this same fate.

The temptation to make fun of JS for `0.1 + 0.2 !== 0.3` is strong, I know. But here it's completely bogus.

NOTE

Pretty much all programmers need to be aware of IEEE-754 and make sure they are careful about these kinds of gotchas. It's somewhat amazing, in a disappointing way, how few of them have any idea how IEEE-754 works. If you've taken your time reading and understanding these concepts so far, you're now in that rare tiny percentage who actually put in the effort to understand the numbers in their programs!

Epsilon Threshold

A common piece of advice to work around such floating-point imprecision uses this *very small* `number` value defined by JS:

```
Number.EPSILON;                     // 2.220446049250313e-16
```

Epsilon is the smallest difference JS can represent between `1` and the next value greater than `1`. While this value is technically implementation/platform dependent, it's generally about `2.2E-16`, or `2^-52`.

To those not paying close enough attention to the details here – including my past self! – it's generally assumed that any skew in floating point precision from a single operation should never be greater than `Number.EPSILON`. Thus, in theory, we can use `Number.EPSILON` as a *very small* tolerance value to ensure number equality comparisons are *safe*:

```
function safeNumberEquals(a,b) {
    return Math.abs(a - b) < Number.EPSILON;
}

point3a = 0.1 + 0.2;
point3b = 0.3;

// are these safely "equal"?
safeNumberEquals(point3a,point3b);      // true
```

WARNING:

In the first edition "Types & Grammar" book, I indeed recommended exactly this approach. I was wrong. I should have researched the topic more closely.

But, it turns out, this approach isn't safe at all:

```
point3a = 10.1 + 0.2;
point3b = 10.3;

safeNumberEquals(point3a,point3b);      // false :(
```

Well… that's a bummer!

Unfortunately, `Number.EPSILON` only works as a "safely equal" error threshold for certain small numbers/operations, and in other cases, it's far too small, and yields false negatives.

You could scale `Number.EPSILON` by some factor to produce a larger threshold that avoids false negatives but still filters out all the floating point skew in your program. But what factor to use is entirely a manual judgement call based on what magnitude of values, and operations on them, your program will entail. There's no automatic way to compute a reliable, universal threshold.

Unless you really know what you're doing, you should just *not* use this `Number.EPSILON` threshold approach at all.

TIP

If you'd like to read more details and solid advice on this topic, I highly recommend reading this post. [21] But if we can't use `Number.EPSILON` to avoid the perils of floating-point skew, what do we do? If you can avoid floating-point altogether by scaling all your numbers up so they're all whole number integers (or bigints) while performing math, do so. Only deal with decimal values when you have to output/represent a final value after all the math is done. If that's not possible/practical, use an arbitrary precision decimal emulation library and avoid `number` values entirely. Or do your math in another external programming environment that's not based on IEEE-754.

Numeric Comparison

Like strings, number values can be compared (for both equality and relational ordering) using the same operators.

[21] "PLEASE don't follow the code recipe in the accepted answer", Stack Overflow; Daniel Scott; July 2019; https://stackoverflow.com/a/56967003/228852 ; Accessed August 2022

Remember that no matter what form the number value takes when being specified as a literal (base-10, octal, hexadecimal, exponential, etc), the underlying value stored is what will be compared. Also keep in mind the floating point imprecision issues discussed in the previous section, as the comparisons will be sensitive to the exact binary contents.

Numeric Equality

Just like strings, equality comparisons for numbers use either the `==` / `===` operators or `Object.is(..)`. Also recall that if the types of both operands are the same, `==` performs identically to `===`.

```
42 == 42;                   // true
42 === 42;                  // true

42 == 43;                   // false
42 === 43;                  // false

Object.is(42,42);           // true
Object.is(42,43);           // false
```

For `==` coercive equality (when the operand types don't match), if either operand is not a string value, `==` prefers a numeric equality check (meaning both operands are coerced to numbers).

```
// numeric (not string!) comparison
42 == "42";                 // true
```

In this snippet, the coercive equality coerces `"42"` to `42`, not vice versa (`42` to `"42"`). Once both types are `number`, then their values are compared for exact equality, the same as `===` would.

Recall that JS doesn't distinguish between values like `42`, `42.0`, and `42.000000`; under the covers, they're all the same. Unsurpisingly, the `==` and `===` equality checks verify that:

```
42 == 42.0;                     // true
42.0 == 42.00000;               // true
42.00 === 42.000;               // true
```

The intuition you likely have is, if two numbers are literally the same, they're equal. And that's how JS interprets it. But `0.3` is not literally the same as the result of `0.1 + 0.2`, because (as we saw earlier), the latter produces an underlying value that's *very close* to `0.3`, but is not exactly identical.

What's interesting is, the two values are *so close* that their difference is less than the `Number.EPSILON` threshold, so JS can't actually represent that difference *accurately*.

You might then think, at least informally, that such JS numbers should be "equal", since the difference between them is too small to represent. But notice: JS *can* represent that there *is* a difference, which is why you see that `4` at the very end of the decimal when JS evaluates `0.1 + 0.2`. And you *could* type out the number literal `0.00000000000000004` (aka, `4e-17`), being that difference between `0.3` and `0.1 + 0.2`.

What JS cannot do, with its IEEE-754 floating point numbers, is represent a number that small in an *accurate* enough way that operations on it produce expected results. It's too small to be fully and properly represented in the `number` type JS provides.

So `0.1 + 0.2 == 0.3` resolves to `false`, because there's a difference between the two values, even though JS can't accurately represent or do anything with a value as small as that difference.

Also like we saw with strings, the `!=` (coercive not-equal) and `!==` (strict-not-equal) operators work with numbers. `x != y` is basically `!(x == y)`, and `x !== y` is basically `!(x === y)`.

There are two frustrating exceptions in numeric equality (whether you use `==` or `===`):

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

`NaN` is never equal to itself (even with `===`), and `-0` is always equal to `0` (even with `===`). It sometimes surprises folks that even `===` has these two exceptions in it.

However, the `Object.is(..)` equality check has neither of these exceptions, so for equality comparisons with `NaN` and `-0`, avoid the `==` / `===` operators and use `Object.is(..)` – or for `NaN` specifically, `Number.isNaN(..)`.

Numeric Relational Comparisons

Just like with string values, the JS relational operators (`<`, `<=`, `>`, and `>=`) operate with numbers. The `<` (less-than) and `>` (greater-than) operations should be fairly self explanatory:

```
41 < 42;                        // true

0.1 + 0.2 > 0.3;                // true (ugh, IEEE-754)
```

Remember: just like `==`, the `<` and `>` operators are also coercive, meaning that any non-number values are coerced to numbers

– unless both operands are already strings, as we saw earlier. There are no strict relational comparison operators.

If you're doing relational comparisons between numbers, the only way to avoid coercion is to ensure that the comparisons always have two numbers. Otherwise, these operators will do *coercive relational* comparisons similar to how `==` performs *coercive equality* comparisons.

Mathematical Operators

As I asserted earlier, the main reason to have numbers in a programming language is to perform mathematical operations with them. So let's talk about how we do so.

The basic arithmetic operators are `+` (addition), `-` (subtraction), `*` (multiplication), and `/` (division). Also available are the operators `**` (exponentiation) and `%` (modulo, aka *division remainder*). There are also `+=`, `-=`, `*=`, `/=`, `**=`, and `%=` forms of the operators, which additionally assign the result back to the left operand – must be a valid assignment target like a variable or property.

NOTE

As we've already seen, the `+` operator is overloaded to work with both numbers and strings. When one or both operands is a string, the result is a string concatenation (including coercing either operand to a string if necessary). But if neither operand is a string, the result is a numeric addition, as expected.

All these mathematical operators are *binary*, meaning they expect two value operands, one on either side of the operator;

they all expect the operands to be number values. If either or both operands are non-numbers, the non-number operand(s) is/are coerced to numbers to perform the operation. We'll cover coercion in detail in a later chapter.

Consider:

```
40 + 2;                 // 42
44 - 2;                 // 42
21 * 2;                 // 42
84 / 2;                 // 42
7 ** 2;                 // 49
49 % 2;                 // 1

40 + "2";               // "402" (string concatenation)
44 - "2";               // 42 (because "2" is coerced to \
2)
21 * "2";               // 42 (..ditto..)
84 / "2";               // 42 (..ditto..)
"7" ** "2";             // 49 (both operands are coerced \
to numbers)
"49" % "2";             // 1 (..ditto..)
```

The `+` and `-` operators also come in a *unary* form, meaning they only have one operand; again, the operand is expected to be a number, and coerced to a number if not:

```
+42;                    // 42
-42;                    // -42

+"42";                  // 42
-"42";                  // -42
```

You might have noticed that `-42` looks like it's just a "negative forty-two" numeric literal. That's not quite right. A nuance of

JS syntax is that it doesn't recognize negative numeric literals. Instead, JS treats this as a positive numeric literal `42` that's preceded, and negated, by the unary `-` operator in front of it.

Somewhat surprisingly, then:

```
-42;                        // -42
- 42;                       // -42
-
    42;                     // -42
```

As you can see, whitespace (and even new lines) are allowed between the `-` unary operator and its operand; actually, this is true of all operators and operands.

Increment and Decrement

There are two other unary numeric operators: `++` (increment) and `--` decrement. They both perform their respective operation and then reassign the result to the operand – must be a valid assignment target like a variable or property.

You may sort of think of `++` as equivalent to `+= 1`, and `--` as equivalent to `-= 1`:

```
myAge = 42;

myAge++;
myAge;                      // 43

numberOfHeadHairs--;
```

However, these are special operators in that they can appear in a postfix (after the operand) position, as above, or in a prefix (before the operand) position:

```
myAge = 42;

++myAge;
myAge;                  // 43

--numberofHeadHairs;
```

It may seem peculiar that prefix and postfix positions seem to give the same result (incrementing or decrementing) in such examples. The difference is subtle, and isn't related to the final reassigned result. We'll revisit these particular operators in a later chapter to dig into the positional differences.

Bitwise Operators

JS provides several bitwise operators to perform bit-level operations on number values.

However, these bit operations are not performed against the packed bit-pattern of IEEE-754 numbers (see Chapter 1). Instead, the operand number is first converted to a 32-bit signed *integer*, the bit operation is performed, and then the result is converted back into an IEEE-754 number.

Keep in mind, just like any other primitive operators, these just compute new values, not actually modifying a value in place.

- `&` (bitwise AND): Performs an AND operation with each corresponding bit from the two operands; `42 & 36 === 32` (i.e., `0b00...101010 & 0b00...100100 === 0b00..100000`)
- `|` (bitwise OR): Performs an OR operation with each corresponding bit from the two operands; `42 | 36`

`=== 46` (i.e., `0b00...101010 | 0b00...100100 === 0b00...101110`)
- `^` (bitwise XOR): Performs an XOR (eXclusive-OR) operation with each corresponding bit from the two operands; `42 ^ 36 === 14` (i.e., `0b00...101010 ^ 0b00...100100 === 0b00...001110`)
- `~` (bitwise NOT): Performs a NOT operation against the bits of a single operand; `~42 === -43` (i.e., `~0b00...101010 === 0b11...010101`); using 2's complement, the signed integer has the first bit set to `1` meaning negative, and the rest of the bits (when flipped back, according to 2's complement, which is 1's complement bit flipping and then adding `1`) would be `43` (`0b10...101011`); the equivalent of `~` in decimal number arithmetic is `~x === -(x + 1)`, so `~42 === -43`
- `<<` (left shift): Performs a left-shift of the bits of the left operand by the count of bits specified by the right operand; `42 << 3 == 336` (i.e., `0b00...101010 << 3 === 0b00...101010000`)
- `>>` (right shift): Performs a sign-propagating right-shift of the bits of the left operand by the count of bits specified by the right operand, discarding the bits that fall off the right side; whatever the leftmost bit is (`0`, or `1` is negative) is copied in as bits on the left (thereby preserving the sign of the original value in the result); `42 >> 3 === 5` (i.e., `0b00..101010 >> 3 === 0b00...000101`)
- `>>>` (zero-fill right shift, aka unsigned right shift): Performs the same right-shift as `>>`, but `0` fills on the bits shifted in from the left side instead of copying the leftmost bit (thereby ignoring the sign of the original value in the result); `42 >>> 3 === 5` but `-43 >>> 3 === 536870906` (i.e., `0b11...010101 >>> 3 ===`

`0b0001...111010`)
- `&=`, `|=`, `<<=`, `>>=`, and `>>>=` (bitwise operators with assignment): Performs the corresponding bitwise operation, but then assigns the result to the left operand (which must be a valid assignment target, like a variable or property, not just a literal value); note that `~=` is missing from the list, because there is no such "binary negate with assignment" operator

In all honesty, bitwise operations are not very common in JS. But you may sometimes see a statement like:

```
myGPA = 3.54;

myGPA | 0;              // 3
```

Since the bitwise operators act only on 32-bit integers, the `| 0` operation truncates (i.e., `Math.trunc(..)`) any decimal value, leaving only the integer.

WARNING

A common misconception is that `| 0` is like *floor* (i.e., `Math.floor(..)`). The result of `| 0` agrees with `Math.floor(..)` on positive numbers, but differs on negative numbers, because by standard definition, *floor* is an operation that rounds-down towards `-Infinity`. `| 0` merely discards the decimal bits, which is in fact truncation.

Number Value Methods

Number values provide the following methods (as properties) for number-specific operations:

- `toExponential(..)`: produces a string representation of the number using scientific notation (e.g., `"4.2e+1"`)
- `toFixed(..)`: produces a non-scientific-notation string representation of the number with the specified number of decimal places (rounding or zero-padding as necessary)
- `toPrecision(..)`: like `toFixed(..)`, except it applies the numeric argument as the number of significant digits (i.e., precision) including both the whole number and decimal places if any
- `toLocaleString(..)`: produces a string representation of the number according to the current locale

```
myAge = 42;

myAge.toExponential(3);         // "4.200e+1"
```

One particular nuance of JS syntax is that `.` can be ambiguous when dealing with number literals and property/method access.

If a `.` comes immediately (no whitespace) after a numeric literal digit, and there's not already a `.` decimal in the number value, the `.` is assumed to be a starting the decimal portion of the number. But if the position of the `.` is unambiguously *not* part of the numeric literal, then it's always treated as a property access.

```
42 .toExponential(3);           // "4.200e+1"
```

Here, the whitespace disambiguates the `.`, designating it as a property/method access. It's perhaps more common/preferred to use `(..)` instead of whitespace for such disambiguation:

```
(42).toExponential(3);          // "4.200e+1"
```

An unusual-looking effect of this JS parsing grammar rule:

```
42..toExponential(3);           // "4.200e+1"
```

So called the "double-dot" idiom, the first `.` in this expression is a decimal, and thus the second `.` is unambiguously *not* a decimal, but rather a property/method access.

Also, notice there's no digits after the first `.`; it's perfectly legal syntax to leave a trailing `.` on a numeric literal:

```
myAge = 41. + 1.;

myAge;                          // 42
```

Values of `bigint` type cannot have decimals, so the parsing is unambiguous that a `.` after a literal (with the trailing `n`) is always a property access:

```
42n.toString();                 // 42
```

Static `Number` Properties

- `Number.EPSILON`: The smallest value possible between `1` and the next highest number
- `Number.NaN`: The same as the global `NaN` symbol, the special invalid number
- `Number.MIN_SAFE_INTEGER` / `Number.MAX_SAFE_INTEGER`: The positive and negative integers with the largest absolute value (furthest from `0`)
- `Number.MIN_VALUE` / `Number.MAX_VALUE`: The minimum (positive value closest to `0`) and the maximum (positive value furthest from `0`) representable by the `number` type
- `Number.NEGATIVE_INFINITY` / `Number.POSITIVE_INFINITY`: Same as global `-Infinity` and `Infinity`, the values that represent the largest (non-finite) values furthest from `0`

Static `Number` Helpers

- `Number.isFinite(..)`: returns a boolean indicating if the value is finite – a `number` that's not `NaN`, nor one of the two infinities
- `Number.isInteger(..)` / `Number.isSafeInteger(..)`: both return booleans indicating if the value is a whole `number` with no decimal places, and if it's within the *safe* range for integers (`-2^53 + 1` - `2^53 - 1`)
- `Number.isNaN(..)`: The bug-fixed version of the global `isNaN(..)` utility, which identifies if the argument provided is the special `NaN` value
- `Number.parseFloat(..)` / `Number.parseInt(..)`: utilities to parse string values for numeric digits, left-to-

right, until the end of the string or the first non-float (or non-integer) character is encountered

Static `Math` Namespace

Since the main usage of `number` values is for performing mathematical operations, JS includes many standard mathematical constants and operation utilities on the `Math` namespace.

There's a bunch of these, so I'll omit listing every single one. But here's a few for illustration purposes:

```
Math.PI;                        // 3.141592653589793

// absolute value
Math.abs(-32.6);                // 32.6

// rounding
Math.round(-32.6);              // -33

// min/max selection
Math.min(100,Math.max(0,42));   // 42
```

Unlike `Number`, which is also the `Number(..)` function (for number coercion), `Math` is just an object that holds these properties and static function utilities; it cannot be called as a function.

WARNING

One peculiar member of the `Math` namespace is `Math.random()`, for producing a random floating point value between `0` and `1.0`. It's unusual to consider random number generation – a task that's inherently stateful/side-effect'ing – as a mathematical operation. It's also long been a footgun security-wise, as the pseudo-random number generator (PRNG) that JS uses is *not* secure (can be predicted) from a cryptography perspective. The web platform stepped in several years ago with the safer `crypto.getRandomValues(..)` API (based on a better PRNG), which fills a typed-array with random bits that can be interpreted as one or more integers (of type-specified maximum magnitude). Using `Math.random()` is universally discouraged now.

BigInts and Numbers Don't Mix

As we covered in Chapter 1, values of `number` type and `bigint` type cannot mix in the same operations. That can trip you up even if you're doing a simple increment of the value (like in a loop):

```
myAge = 42n;

myAge + 1;                  // TypeError thrown!
myAge += 1;                 // TypeError thrown!

myAge + 1n;                 // 43n
myAge += 1n;                // 43n

myAge++;
myAge;                      // 44n
```

As such, if you're using both `number` and `bigint` values in your programs, you'll need to manually coerce one value-type to the other somewhat regularly. The `BigInt(..)` function (no `new` keyword) can coerce a `number` value to `bigint`. Vice versa, to go the other direction from `bigint` to `number`, use the `Number(..)` function (again, no `new` keyword):

```
BigInt(42);                 // 42n

Number(42n);                // 42
```

Keep in mind though: coercing between these types has some risk:

```
BigInt(4.2);                // RangeError thrown!
BigInt(NaN);                // RangeError thrown!
BigInt(Infinity);           // RangeError thrown!

Number(2n ** 1024n);        // Infinity
```

Primitives Are Foundational

Over the last two chapters, we've dug deep into how primitive values behave in JS. I bet more than a few readers were, like me, ready to skip over these topics. But now, hopefully, you see the importance of understanding these concepts.

The story doesn't end here, though. Far from it! In the next chapter, we'll turn our attention to understanding JS's object types (objects, arrays, etc).

Sync & Async (Unbook 5)

Chapter 1: Lost Book (Sync & Async)

The biggest question a reader may be wondering is: what happened to this book? Why did it get canceled or never fully written?

I'll just address that head on:

I decided to cancel writing this book (in full, at least) due to the significant shifts in the frontend development industry, and developers becoming heavily reliant on frameworks for nearly all applications.

The Plan

The goal of this book series has been to uncover and distill the core parts of the JS language, and any patterns built directly on top of those mechanisms, for managing different tasks in the application and data lifecycle. In the first edition of this book ("Async & Performance"), we talked extensively about how the (new, at the time) generators feature provided a capable meta-programming layer to approximate the (proposed, at the time) "async await" pattern for synchronous-looking asynchronous flow control.

In the ensuing years since that book was written, `async..await` landed as a feature in JS, and has now become the de facto standard for how developers manage flow control with promises over their asynchronous tasks (as opposed to promise chains).

Certainly, this second edition book would have re-visited generators. But we no longer need them to approximate `async..await`, so we could have focused on different reasons for generators to be useful – and indeed, there are many powerful uses of them.

Moreover, we now have "async generators" in JS, which are like mashing up both the generator-iterator protocol and the async-await protocol into a single function type, which let's us produce, essentially, streams of asynchronous data.

This book was planning to cover all the (exciting, to me!) synchronous and asynchronous iteration and data management patterns that flow from such mechanisms. I also planned to cover observables (which have recently landed in the browser, not as part of the JS specification but part of the web platform), and even signals (which are progressing, at time of writing, towards inclusion in JS).

So… with all those good and meaty details, you may even more be wondering, why didn't this book warrant being written?

The Reality

Unfortunately, the majority of you will never actually do much with any of these mechanisms, at least not at the level of interacting with them as JS features. That's because the frameworks you use for your applications all have their own flavor and style of abstractions they present you, for your code to utilize.

One of the greatest frustrations I used to encounter when teaching about core JS topics was essentially, "this pattern is great, but how can I use it in React?" And the horrible answer I usually had to give was: "You don't."

The only people who really are going to use these low level direct JS features? Framework (and library) authors. Anyone else using their tools? They will use whatever abstractions those tools provide. To some extent, that was always true of JS. But it's so much more true today in 2025 (time of writing) than it was when I first started these JS books (1st edition) *way back* in 2014.

And as if that isn't enough of a barrier, the advent of AI has even more accelerated the trend. More and more developers are using code that is partially or mostly generated by AI, and those LLMs are almost always going to spit out code that's centered around the popular libraries and frameworks.

So bottom line: I don't want to write a whole book where the only real audience of the material are the relatively few folks who make the frameworks and libraries, but where the vast majority of folks will be frustrated that they can't really make practical use of what they read about.

I think such a book would really be disingenuous and wasteful of readers' time. I care far too much about you, dear reader, than to subject you to that.

That said...

I do however feel like there's a little more juice we could squeeze out of this. So... in the next chapter, I'm going to briefly cover some of these topics, mostly just as quick glances, perhaps piquing curiosity and sparking ideas for readers.

Just so you're aware, *this book*, this content you're presently reading, is *not* found anywhere else publicly, online. It's exclusive, right here!

Second Edition Thank Yous

The following 371 Kickstarter backers generously backed the campaign to write/publish the four remaining books of the second edition. Without their faithful and patient support, these books would not have happened.

I am deeply grateful to each of you!

Marc at Frontend Masters, Bassim, Tasos Tsournos, Lauren Clark, Simon, Appgrader, Marcos S., Noah Rodenbeek, Lichtjaeger, Jon Miller, Anil, Greg Gerard, Frank Deberle, Davide Bonifacio, Brandon Leichty, Rowdy Rabouw, Gaspar Radu, Sukumar Vaddi, Gordon, jakecodes, Leo Furze-Waddock, Nick de Jong, D. Vinci, Brian Vogel, Gareth loot, Simon Taylor, Chris O'Brien, Nayana Davis, Mark Kramer, Sunil Samuel, @nestor.j.g, Ryan McDaniel, Mert Even, Haim Yulzari, Josh Marks, Chiril Sarajiu, Barnabas Jovanovics, LynchyC, Yahya Jideh, Chris Weber, Dan Cortes, Johnny Tordgeman, Ky Lee, Brian Wisti, Steven Marrocco, Thomas Randolph, Petri Lindholm, John Cole,

github, @denysmarkov, Jacob Scherber, Pierre-Yves Lebrun, mcekiera, Matthew Wasbrough, Génicot Jean-Baptiste, Adam Zając, Lenny Erlesand, Samuel Gustafsson, Hunter Jansen, Theo Armour, Nate Hargitt, Anon, Github repo, cawel, mpelikan, @farisaziz12, Ojars, Camilo Segura, Sean Seagren, Michael Vendivel, Evan, Eric Schwertfeger, Gene Garbutt, Elena Rogleva, Fiona Cheung, Anton Levholm, Lorenzo Bersano, Ando NARY, Ruben Krbashyan, Anonymous please, @jcubic, Bhavin Dave, A. Hitchcock, H0rn0chse, Yaniv Wainer, Zach, Raúl Pineda, Rohan Gupta, Karthik, Kapil, Ricardo Trejos, InvisibleLuis, BruceRobertson, Neil Lupton, Chris Schweda, Luca Mezzalira, antonio molinari, David Pinezich, Jon Barson, Nick Kaufmann, Just Andrew, Rock Kayode Winner, @omar12, Page Han, Aurélien Bottazini, Michael, Petr Siegl, Ilya Sarantsev, Alfredo Delgado, aharvard, Jannaee, Aaron McBride, Toma, epmatsw, Igor "kibertoad" Savin, Christian Rackerseder, NC Patro, Kevin, Brian Holt, Brian Ashenfelter, Selina Chang, cwavedave, Alex Grant, Craig Robertson, Eduardo Sanz Martin, oieduardorabelo, Esteban Massuh, tedhexaflow, Gershon Gerchikov, Harika Yedidi, Brad Dougherty, Nitin, Leo Balter, Syed Ahmad, Kaz de Groot, Pinnemouche Studio, Jerome Amos, Dan Poynor, John Liu, @thedavefulton, Madeline Bernard, Ikigai42, Antonio Chillaron, Sachin, Prakasam Venkatachalam, jmarti705, Mihailo23, Mihailo Pantovic, Magloire, samrudh, Mykenzie Rogers, Len, Lyza Danger Gardner, Ryan, Roman, Radojica

Radivojevic, Gabrien Symons, Ryan Parker, Andrés, Merlin, rushabh_badani, notacouch, Anna Borja, Steve Albers, Marc at Frontend Masters, Bala Vemula, @chrismcdonald84, stern9, Janne Hellsten, Alexandre Madurell, Tanner Hodges, Joe Chellman, Joachim Kliemann, Stefano Frasson Pianizzola, Sergey Kochergan, Spiridonov Dmitriy, IonutBihari, Alexandru Olteanu, Javi, Marlee Peters, @vadocondes1, Gerardo Leal, Albert Sebastian, Atish Raina, Andreas Gebhardt, David Deren, Maksym Gerashchenko, Alexandru, Matt Peck, William Lacroix, Pavlo, Jon, Brett Walker, Iosif Psychas, Ferran Buireu, crs1138, Emiliano anichini, Max Koretskyi, Sander Elias, Michael Romanov, Barkóczi Dávid, Daw-Chih Liou, Dale Caffull, Amanda Dillon, Mike, Justin Hefko, Muhammad Ali Shah, Ketan Srivastav, redeemefy, Stefan Trivunčić, Manuel Juan Fosela Águila, Dragan Majstorović, Harsha C G, Himanshu, Luke, Sai Ponnada, Mark Franco, David Whittaker, Dr. Teresa Vasquez, Ian Wright, Lora Rusinouskaya, Petar, Harish, Mairead, shimon simo moyal, Sunny Puri, Максим Кочанов, Alex Georoceanu, Nicolas Carreras, damijanc, zach.dev, Coati, Brian Whitton, Denis Ciccale, Piotr Seefeld, Chase Hagwood, Amritha K, Κώστας Μηναΐδης, Trey Aughenbaugh, J David Eisenberg, Paul Thaden, Corina S, Chris Dhanaraj, Nahid Hossain, Justin McCullough, Arseny, Mark Trostler, Lucy Barker, Maaz Syed Adeeb, mcginkel, Derick Rodriguez, Helen Tsui, Rus Ustyugov, Vassilis Mastorostergios, Ryan Ewing, Rob Huelga, jinujj, ultimateoverload, Chaos, Andy Howell (spacebeers), Archana,

AG Grid, theblang, Coyotiv School of Software Engineering, Ayush Rajniwal, Manish Bhatt, Shea Leslein, Jordan Chipman, jg0x42, Arvind Kumar, Eduardo Grigolo, Peter Svegrup, Jakub Kotula, William Richardson, Jonah and Ali, nicciwill, Lauren Hodges, Travis Sauer, Alexandros, Abhas, Kirankumar Ambati, Gopalakrishnan, Mika Rehman, Sreeram Sama, Shubhamsatyam Verma, Heejae Chang, Andrico karoulla, Niek Heezemans, Stanislav Horáček, Luis Ibanhi, Jasmine Wang, Yunier, Brian Barrow, Matteo Hertel, Aleksandar Milicevic, achung89, kushavi, Ahmed Fouad, Venkat Kaluva, Ian Wotkun, Andreas Näsman, ivan-siachoque, Paul Gain, Santhosh R, Gustavo Morales, ScottAwseome, Fredrik Thorkildsen, Manvel, holleB, James Sullivan, Adam Kaźmierczak, carlottosson, Alvee, Alex Reardon, Olie Chan, Fredrik S, Brett.Buskirk, Rui Sereno, Nathan Strong, lostdesign, ppseprus, James, anthonybsd, Alena Charnova, Kevin K, @codingthirty, Tim Davis, Jonathan Yee, Christa, Fabian Merchan, Nathanael McDaniel, Dave N, Brian Chirgwin, Abdulrahman (Abdu) Assabri, rmeja, Jan Václavek, Phillip Hogan, Adhithya Rajagopalan (xadhix), Jason Humphrey, Antoinette Smith, Elliot Redhead, zokocx, John Sims, Michalis Garganourakis, Adarsh Konchady, Anton Oleg Dobrovolskyy, George Tharakan, syd, Ryan D., Iris Nathan, Srishti Gupta, Miguel Rivero, @saileshraghavan, Yojan, @bgollum, Junyts, Like Ezugworie, Vsh13, LocalPCGuy, DMGabriel, Juan Tincho, William Greenlaw, atisbacsi, cris ryan tan, Jonathan Clifron, Daniel Dolich, Praj, Caisman, Michał, Mark C, 3xpedia

A special thanks to:

- A. Hitchcock
- Alexandru
- Appgrader
- Coyotiv School of Software Engineering
- Gaspar Radu
- IonutBihari
- jmarti705
- John Liu
- Syed Ahmad
- Travis Sauer
- William Greenlaw

All of you are fantastic!