01 Mental Models

Read this code:

```
let a = 10;
let b = a;
a = 0;
```

What are the values of `a` and `b` after it runs? Work it out in your head before reading further.

If you've been writing JavaScript for a while, you might object: "This snippet is much simpler than the code I'm writing every day. What's the point?"

The goal of this exercise isn't to introduce you to variables. We assume you're already familiar with them. Instead, it is to make you notice and reflect on

your *mental model*.

What's a Mental Model?

Read the code above again with an intention to *really be sure* what the result is. (We'll see why this intention is important a bit later.)

While you're reading it for the second time, pay close attention to what's happening in your head, step by step. You might notice a monologue like this:

- `let a = 10;`
  - Declare a variable called `a`. Set it to `10`.
- `let b = a;`
  - Declare a variable called `b`. Set it to `a`.
  - Wait, what's `a` again? Ah, it was `10`. So `b` is `10` too.
- `a = 0;`
  - Set the `a` variable to `0`.
- So `a` is `0` now, and `b` is `10`. That's our answer.

Maybe your monologue is a bit different. Maybe you say "assign" instead of "set," or maybe you read it in a slightly different order. Maybe you arrived at a different result. Pay attention to how exactly it was different. Note how even this monologue doesn't capture what's really happening in your head. You might say "set `b` to `a`," but what does it even mean to *set* a variable?

You might find that for every familiar fundamental programming concept (like a variable) and operations on it (like setting its value), there is a set of deep-rooted analogies that you associated with it. Some of them may come from the real world. Others may be repurposed from other fields you learned first, like numbers

from math. These analogies might overlap and even contradict each other, but they still help you make sense of what's happening in the code.

For example, many people first learned about variables as "boxes"—containers that hold your stuff. Even if you don't vividly imagine boxes anymore when you see a variable, they might still behave "boxily" in your imagination.

These approximations of how something works in your head are known as "mental models." It may be hard if you've been programming for a long time, but try to notice and introspect your mental models. They're probably a combination of visual, spatial, and mechanical mental shortcuts. These intuitions (like "boxiness" of variables) influence how we read code our whole lives.

Unfortunately, sometimes our mental models are wrong. Maybe a tutorial we read early on sacrificed accuracy in order to make a concept easier to explain. Maybe, when we started learning JavaScript, we incorrectly "brought over" an expected behavior from a language we learned earlier. Maybe we inferred a mental model from some piece of code but never really verified it was accurate.

Identifying and fixing these problems is what *Just JavaScript* is all about. We will gradually build (or possibly rebuild) your mental model of JavaScript to be accurate and useful. A good mental model will give you confidence in your own code, and it will let you understand (and fix) code that someone else wrote.

(By the way, `a` being `0` and `b` being `10` *is* the correct answer.)

Coding, Fast and Slow

"Thinking, Fast and Slow" is a book by Daniel Kahneman that explores the two different "systems" humans use when thinking.

Whenever we can, we rely on the "fast" system, which is good at pattern matching and "gut reactions." We share this system (which is necessary for survival!) with

many animals, and it gives us amazing powers like the ability to walk without falling over. But it's not good at planning.

Uniquely, thanks to the development of the frontal lobe, humans also possess a "slow" thinking system. This "slow" system is responsible for complex step-by-step reasoning. It lets us plan future events, engage in arguments, or follow mathematical proofs.

Because using the "slow" system is so mentally draining, we tend to default to the "fast" one—even when dealing with intellectual tasks like coding.

Imagine that you're in the middle of a lot of work, and you want to quickly identify what this function does. Take a glance at it:

```
function duplicateSpreadsheet(original) {
  if (original.hasPendingChanges) {
    throw new Error('You need to save the file before you can duplicate it.'
  }
  let copy = {
    created: Date.now(),
    author: original.author,
    cells: original.cells,
    metadata: original.metadata,
  };
  copy.metadata.title = 'Copy of ' + original.metadata.title;
  return copy;
}
```

You've probably noticed that:

- This function duplicates a spreadsheet.
- It throws an error if the original spreadsheet isn't saved.
- It prepends "Copy of" to the new spreadsheet's title.

What you might *not* have noticed (great job if you did!) is that this function *also* accidentally changes the title of the original spreadsheet. Missing bugs like this is something that happens to every programmer, every day.

Now that you know a bug exists, will you read the code differently? If you used the "fast" thinking system at first, you might switch to the more laborious "slow" system when you realize there's a bug in the code.

When we use the "fast" system, we guess what the code does based on its overall structure, naming conventions and comments. Using the "slow" system, we retrace what the code does step by step—a tiring and time-consuming process.

This is why having an accurate mental model is so important. Simulating a computer in your head is hard, and when you have to fall back to the "slow" thinking system, your mental model is all you can rely on. With the wrong mental model, you'll fundamentally misunderstand what to expect from your code, and all your effort will be wasted.

Don't worry if you can't find the bug at all—it just means you'll get the most out of this course! Over the next modules, we'll rebuild our mental model of JavaScript together, so that you can catch bugs like this immediately.

In the next module, we'll start building mental models for some of the most fundamental JavaScript concepts—values and expressions.

Finished reading?

Mark this episode as learned to track your progress.

UP NEXT

The JavaScript Universe →

02 The JavaScript Universe

In the beginning was the Value.

What *is* a value? It's hard to say.

What is a point in geometry? What is a word in human language? A value is a fundamental concept in JavaScript—so we can't define it through other terms.

Instead, we'll define it through examples. Numbers and strings are values. Objects and functions are values, too.

There are also a lot of things that are *not* values, like the pieces of our code—our `if` statements, loops, and variable declarations, for example.

Values and Code

As we start building our mental model, one of the first common misconceptions we need to clear up is that values *are* our code. Instead, we need to think of them separately—our code interacts with values, but values exist in a completely separate space.

To distinguish between values and code in my JavaScript program, I like to imagine this drawing of the Little Prince by Antoine de Saint-Exupéry:

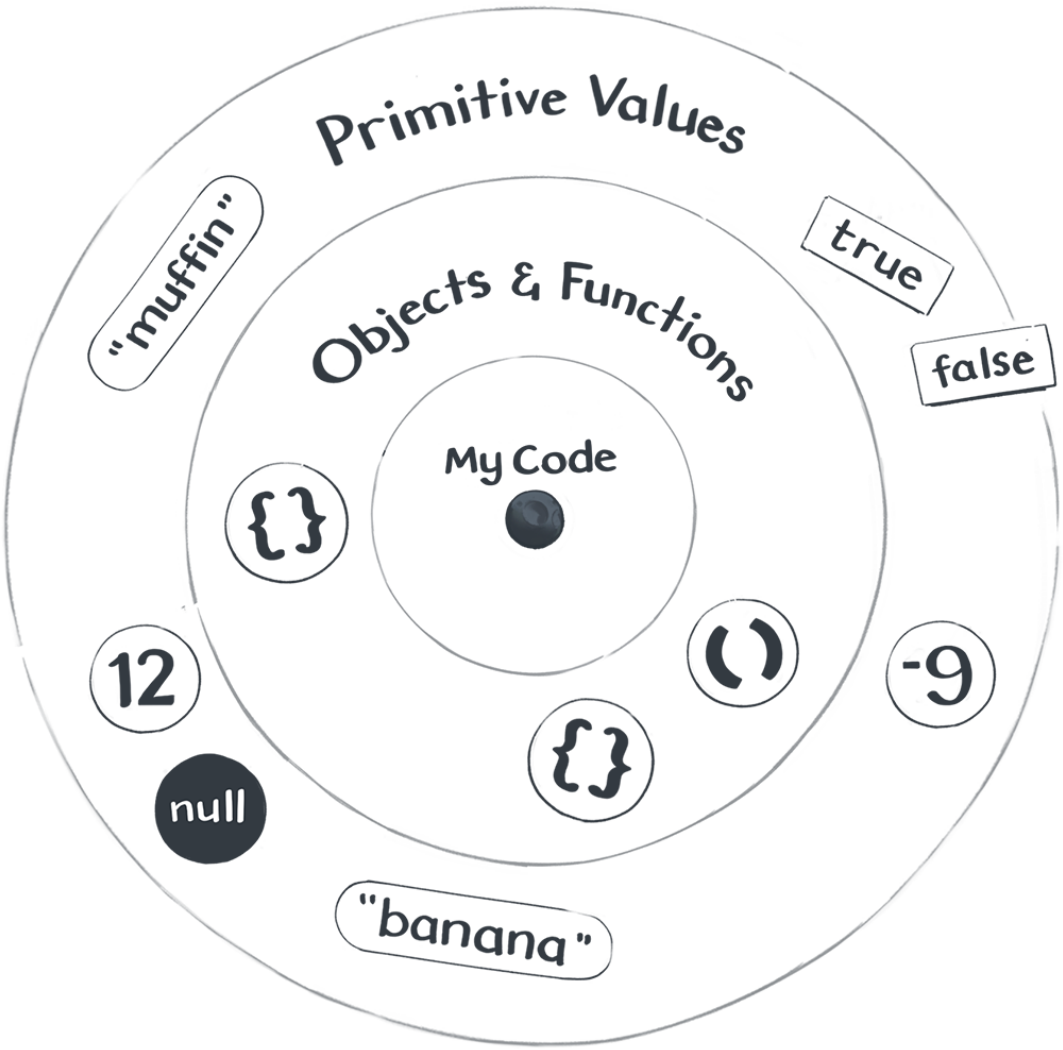
I'm standing on a small planet, holding a list of instructions. This list is my program—my code. As I read through my list of instructions, I can see a lot going on—there are `if` statements, variable declarations, commas and curly braces.

My code contains instructions like "make a function call," "do this thing many times," or even "throw an error." I read through these instructions step by step from the surface of my little world.

But every once in a while, I look up.

On a clear night, I see the different values in the JavaScript sky: booleans, numbers, strings, symbols, functions and objects, `null` and `undefined` —oh my! I might refer to them in my code, but they don't exist *inside* my code.

In our JavaScript universe, values float in space.



"Hold on," you might say, "I always thought of values as being inside of my code!" Here, I'm asking you to take a leap of faith. It will take a few more modules for this mental model to pay off. Give it five minutes. I know what I'm doing.

Values

Broadly, there are two kinds of values.

Primitive Values

Primitive values are like stars—cold and distant, but always there when I need them. Even from the surface of my small planet, I can find them and point them out. They can be numbers and strings, among other things. All primitive values have something in common: **They are a permanent part of our JavaScript universe. I can point to them, but I can't create, destroy, or change them.**

To see primitive values in practice, open your browser's console and log them:

```
console.log(2);
console.log("hello");
console.log(undefined);
```

Objects and Functions

Objects and functions are also values but, unlike primitive values, **I can manipulate them from my code.** If primitive values are like distant stars, then objects and functions are more like asteroids floating around my planet. They're not part of my code, but they're close enough to manipulate.

Fun Fact

Functions are objects, but because they include a few unique additional features, we're going to refer to them separately to avoid confusion.

Go ahead and log a few of them to the browser console:

```
console.log({});
console.log([]);
console.log(x => x * 2);
```

Notice how the browser console displays them differently from primitive values. Some browsers might display an arrow before them, or do something special when you click them. If you have a few different browsers installed, compare how they visualize objects and functions.

Types of Values

At first, all values in the JavaScript cosmos might look the same—just bright dots in the sky. But we are here to study all of the different things floating above us in our JavaScript universe, so we'll need a way to categorize them.

We can break values down into types—values of the same type behave in similar ways. As an aspiring astronomer, you might want to know about *every* type of value that can be observed in the JavaScript sky.

After almost twenty-five years of studying JavaScript, the scientists have only discovered nine such types:

Primitive Values

- **Undefined** (`undefined`), used for unintentionally missing values.
- **Null** (`null`), used for intentionally missing values.
- **Booleans** (`true` and `false`), used for logical operations.
- **Numbers** (`-100` , `3.14` , and others), used for math calculations.
- **BigInts** (uncommon and new), used for math on big numbers.
- **Strings** (`"hello"` , `"abracadabra"` , and others), used for text.

- **Symbols** (uncommon), used to perform rituals and hide secrets.

Objects and Functions

- **Objects** (`{}` and others), used to group related data and code.
- **Functions** (`x => x * 2` and others), used to refer to code.

No Other Types

You might ask: "But what about other types I have used, like arrays?"

In JavaScript, there are no other fundamental value types other than the ones we have just enumerated. The rest are all objects! For example, even arrays, dates, and regular expressions fundamentally *are* objects in JavaScript:

```
console.log(typeof([])); // "object"
console.log(typeof(new Date())); // "object"
console.log(typeof(/(hello|goodbye)/)); // "object"
```

Fun Fact

"I see," you might reply, "this is because *everything* is an object!" Alas, this is a popular urban legend, but it's not true.

Although code like `"hi".toUpperCase()` makes `"hi"` seem like an object, this is nothing but an illusion. JavaScript creates a temporary object when you do this, and then immediately discards it. It's fine if this mechanism doesn't click for you yet. It is indeed rather confusing!

For now, you only need to remember that primitive values, such as numbers and strings, are *not* objects.

Checking a Type

There are only nine types of values, but how do we know a particular value's type?

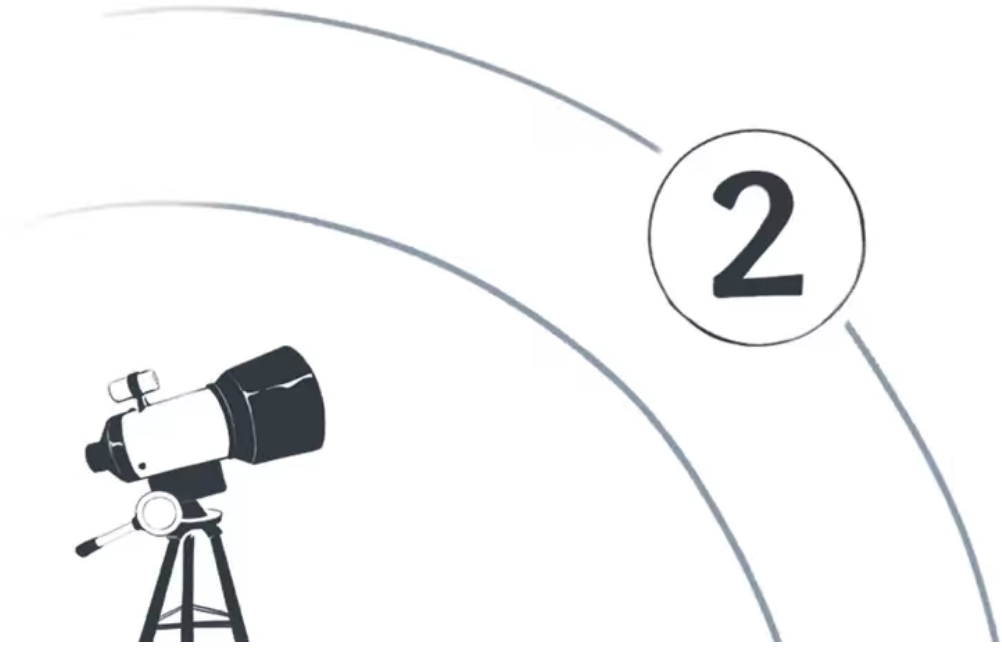
If we want to check a value's type, we can ask with the `typeof` operator. Below are a few examples you can try in the browser console:

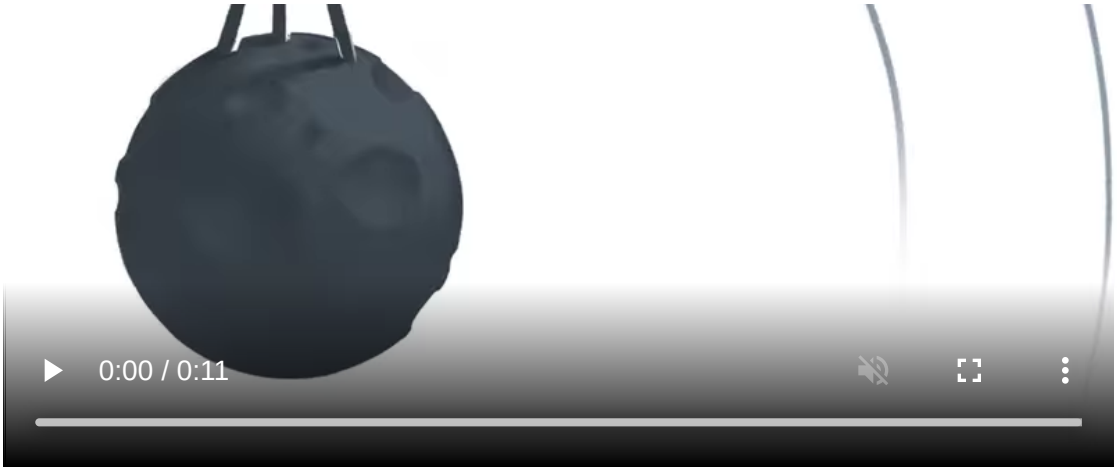
```
console.log(typeof(2)); // "number"
console.log(typeof("hello")); // "string"
console.log(typeof(undefined)); // "undefined"
```

Strictly speaking, using parens isn't required with `typeof`. For example, `typeof 2` would work just as fine as `typeof(2)`. However, sometimes parens are required to avoid an ambiguity. One of the cases below would break if we omitted the parens after `typeof`. Try to guess which one it is:

```
console.log(typeof({})); // "object"
console.log(typeof([])); // "object"
console.log(typeof(x => x * 2)); // "function"
```

You can verify your guess in the browser console.





You might have questions. Good. If you ask a question, our JavaScript universe might answer it! Provided, of course, that you know how to ask.

Expressions

There are many questions JavaScript can't answer. If you want to know whether it's better to confess your true feelings to your best friend or to keep waiting until you both turn into skeletons, JavaScript won't be of much help.

But there are some questions that JavaScript would be *delighted* to answer. These questions have a special name—they are called *expressions*.

If we "ask" the expression `2 + 2`, JavaScript will "answer" with the value `4`.

```
console.log(2 + 2); // 4
```

For another example, remember how we determined the type of a value with `typeof`. In fact, that was also an expression! Our "question" was `typeof(2)` and the JavaScript universe answered it with the string value `"number"`.

```
console.log(typeof(2)); // "number"
```

Expressions are questions that JavaScript can answer. JavaScript answers expressions in the only way it knows how—with values.

If the word "expression" confuses you, think of it as a piece of code that *expresses* a value. You might hear people say that `2 + 2` "results in" or "evaluates to" `4`. These are all different ways to say the same thing.

We ask JavaScript `2 + 2`, and it answers with `4`. **Expressions always result in a single value.** Now we know enough about expressions to be dangerous!

Recap

Let's recap what we know so far:

1. **There are values, and then there's code.** We can think of values as different things "floating" in our JavaScript universe. They don't exist *inside* our code, but we can refer to them from our code.
2. **There are two categories of values: there are *Primitive Values*, and then there are *Objects and Functions*.** In total, there are nine separate types. Each type serves a specific purpose, but some are rarely used.

3. **Some values are lonely.** For example, `null` is the only value of the Null type, and `undefined` is the only value of the Undefined type. As we will learn later, these two lonely values are quite the troublemakers!
4. **We can ask questions with expressions.** Expressions exist in our code, so they are not values. Rather, JavaScript will *answer* our expressions with values. For example, the `2 + 2` *expression* is answered with the *value* `4`.
5. **We can inspect the type of something by wrapping it in a `typeof` expression.** For example, `typeof(4)` results in the string value `"number"`.

Quiz

Now it's time to put what we learned to action.

Even if you already have a decent amount of experience with JavaScript don't skip the exercises! I personally learned some of these things only a few years ago.

After the exercises, we will continue to refine our mental model. This module presents a crude sketch—an approximation. We will focus on different parts of the picture and fill them in with more details, like a progressive JPEG image.

These might seem like small steps, but we're laying the foundation for everything else to come. We're building our JavaScript universe, together.

Finished reading?

Mark this episode as learned to track your progress.

Mark as learned

UP NEXT

Take a Quiz →

03 Values and Variables

We'll kick off this module with a little code snippet.

```
let reaction = 'yikes';
reaction[0] = 'l';
console.log(reaction);
```

What do you expect this code to do? It's okay if you're not sure. **Try to find the answer using your current knowledge of JavaScript.**

I want you to take a few moments and write down your exact thought process for each line of this code, step by step. Pay attention to any gaps or uncertainties in your existing mental model and write them down, too. If you have any doubts, try to articulate them as clearly as you can.

Answer

Don't reveal until you have finished writing.

Reveal

Primitive Values Are Immutable

Did you get the answer right? This might seem like the kind of trivia question that only comes up in JavaScript interviews. Even so, it illustrates an important point about primitive values.

We can't change primitive values.

I will explain this with a small example. Strings (which are primitive) and arrays (which are not) have some superficial similarities. An array is a sequence of items, and a string is a sequence of characters:

```
let arr = [212, 8, 506];
let str = 'hello';
```

We can access the array's first item and the string's first character similarly. It almost feels like strings are arrays:

```
console.log(arr[0]); // 212
console.log(str[0]); // "h"
```

But they're not. Let's take a closer look. We can change an array's first item:

```
arr[0] = 420;
console.log(arr); // [420, 8, 506]
```

Intuitively, it's easy to assume that we can do the same to a string:

```
str[0] = 'j'; // ???
```

But we can't.

It's an important detail we need to add to our mental model. A string is a primitive value, and **all primitive values are immutable.** "Immutable" is a fancy Latin way to say "unchangeable." Read-only. We can't mess with primitive values. At all.

JavaScript won't let us set a property on any primitive value, be it a number, string or something else. Whether it will silently refuse our request or throw an error depends on which mode our code is in. But rest assured that this will never work:

```
let fifty = 50;
fifty.shades = 'gray'; // No!
```

Like any number, `50` is a primitive value. We can't set properties on it.

Remember that in our JavaScript universe, all primitive values are distant stars, floating farthest from our code. We can point to them, but they will always stay where they are, unchanged.

I find it strangely comforting.

Variables and Values—A Contradiction?

You've seen that primitive values are read-only—or, in the parlance of our times, immutable. Now, use the following code snippet to test your mental model.

```
let pet = 'Narwhal';
pet = 'The Kraken';
console.log(pet); // ?
```

Like before, write down your thought process in a few sentences. Don't rush ahead. Pay close attention to how you're thinking about each line, step by step. Does the immutability of strings play a role here? If it does, what role does it play?

Answer

Don't reveal until you have finished writing.

Reveal

If you thought [illegible] right! The answer is `"The Kraken"`. Immutability doesn't play a role here.

Don't despair if you got it wrong. This example may seem like it's contradicting string immutability, **but it's not.**

When you're new to a language, sometimes it's necessary to put aside contradictions so that you can avoid rabbit holes and continue learning. But now that you are committed to building a mental model, you need to question contradictions.

Contradictions reveal gaps in mental models.

Variables Are Wires

Let's look at this example again.

```
let pet = 'Narwhal';
pet = 'The Kraken';
console.log(pet); // "The Kraken"
```

We know that string values can't change because they are primitive. But the `pet` variable *does* change to `"The Kraken"`. What's up with that?

This might seem like it's a contradiction, but it's not. We said primitive *values* can't change, but we didn't say anything about *variables*! As we refine our mental model, we need to untangle a couple of related concepts:

Variables are not values.

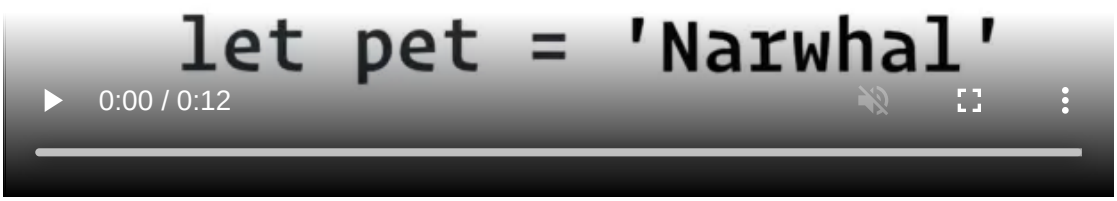
Variables point to values.

Assigning a Value to a Variable

In our JavaScript universe, a variable is a wire that *points to* a value.

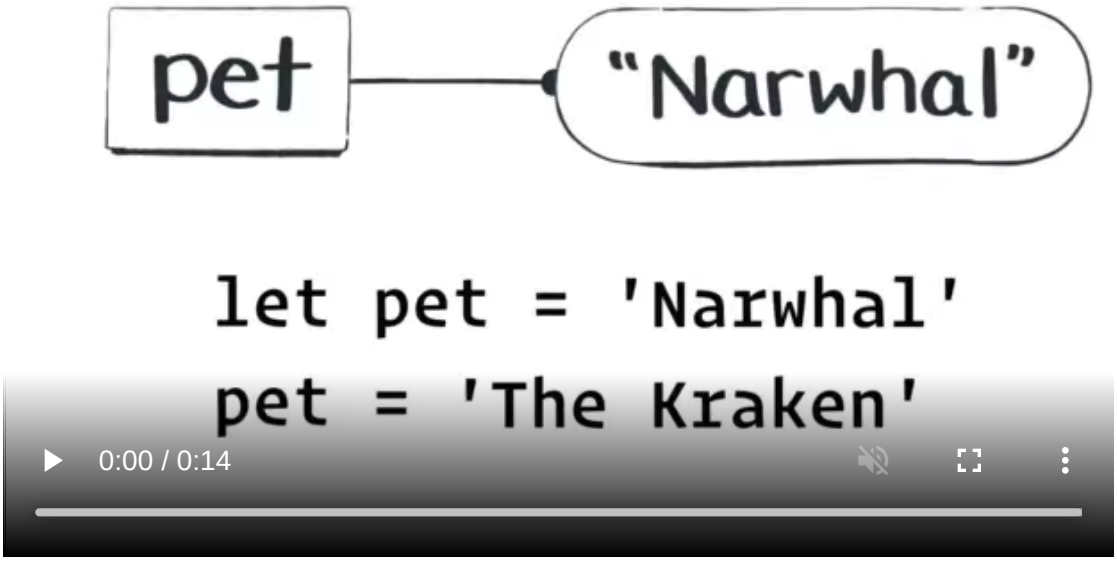
For example, I can point the `pet` variable to the `"Narwhal"` value. (I can also say that I'm *assigning* the value `"Narwhal"` to the variable called `pet`):

```
let pet = 'Narwhal';
```



But what if I want a different pet? No problem—I can point `pet` to another value:

```
pet = 'The Kraken';
```



All I am doing here is instructing JavaScript to point the variable, or a "wire", on the left side (`pet`) to the value on the right side (`'The Kraken'`). It will keep pointing at that value unless I reassign it again later.

Rules of Assignment

There are two rules when we want to use the `=` assignment operator:

1. **The left side of an assignment must be a "wire"—such as the `pet` variable.** Note that the left side can't be a value. (Try these examples in the console):

```
20000 = 'leagues under the sea'; // Nope.
'war' = 'peace'; // Nope.
```

2. **The right side of an assignment must be an expression, so it always results in a value.** Our expression can be something simple, like `2` or `'hello'`. It can also be a more complicated expression—for example:

```
pet = count + ' Dalmatians';
```

Here, `count + ' Dalmatians'` is an expression—a question to JavaScript. JavaScript will answer it with a value (for example, `"101 Dalmatians"`). From now on, the `pet` variable "wire" will point to that particular value.

Fun Fact

If the right side must be an expression, does this mean that simple things—numbers like `2` or strings like `'The Kraken'`—written in code are also expressions? Yes! Such expressions are called *literals*—because we *literally* write down the values that they result in.

Reading a Value of a Variable

I can also *read* the value of a variable—for example, to log it:

```
console.log(pet);
```

That's hardly surprising.

But note that it is not the `pet` *variable* that we pass to `console.log`. We might say that colloquially, but we can't really pass *variables* to functions. We pass the current *value* of the `pet` variable. How does this work?

It turns out that a variable name like `pet` can serve as an expression too! When we write `pet`, we're asking JavaScript a question: "What is the current value of `pet`?" To answer our question, JavaScript follows `pet` "wire," and gives us back the value at the end of this "wire."

So the same expression can give us different values at different times!

Nitpicking

Who cares if you say "pass a variable" or "pass a value"? Isn't the difference hopelessly pedantic? I certainly don't encourage interrupting your colleagues to correct them. That would be a waste of everyone's time.

But you need to have clarity on *what you can do* with each JavaScript concept in your head. You can't skate a bike. You can't fly an avocado. You can't sing a mosquito. And you can't pass a variable—at least not in JavaScript.

Here's a small example of why these details matter.

```
function double(x) {
  x = x * 2;
}

let money = 10;
double(money);
console.log(money); // ?
```

If we thought `double(money)` was *passing a variable*, we could expect that `x = x * 2` would double the `money` variable.

But that's not right: `double(money)` means "figure out the *value* of `money`, and then *pass that value* to `double`." So `money` still points to `10`. What a scam!

What are the different JavaScript concepts in your head? How do they relate to each other and how can we interact with them from code?

Write down a short list of the ones you use most often.

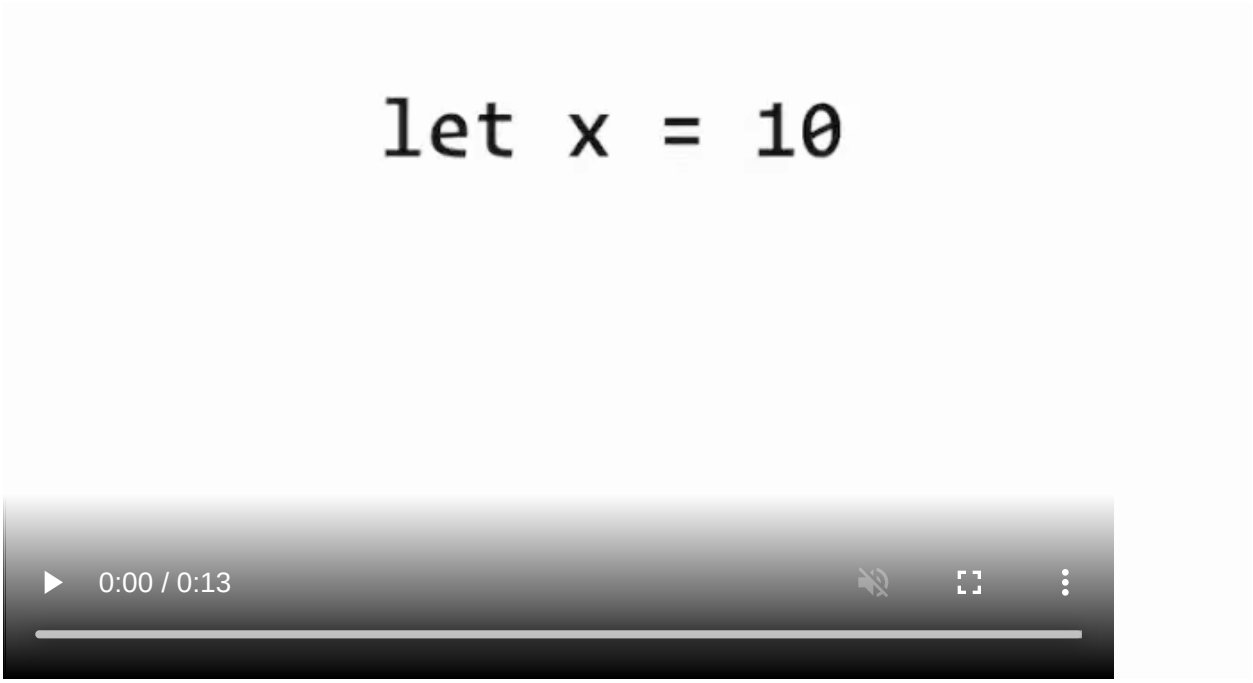
Putting it Together

Now let's revisit the first example from *Mental Models*:

```
let x = 10;
let y = x;
x = 0;
```

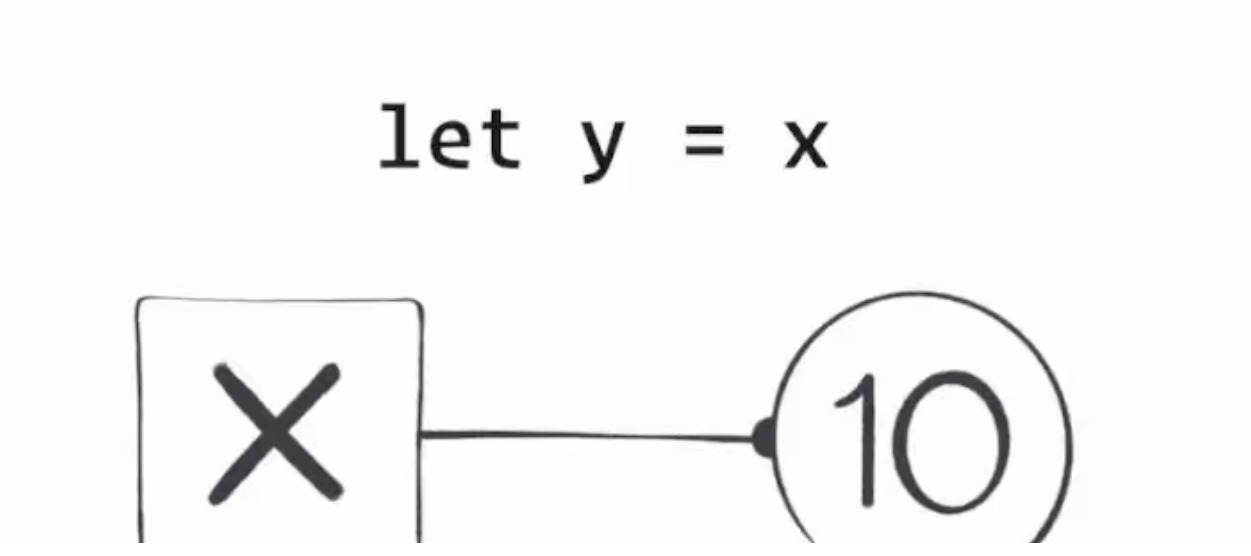
I suggest that you take a piece of paper or a drawing app and sketch out a diagram of what happens to the "wires" of the `x` and `y` variables step by step.

You can do the sketch right here:



The first line doesn't do much:

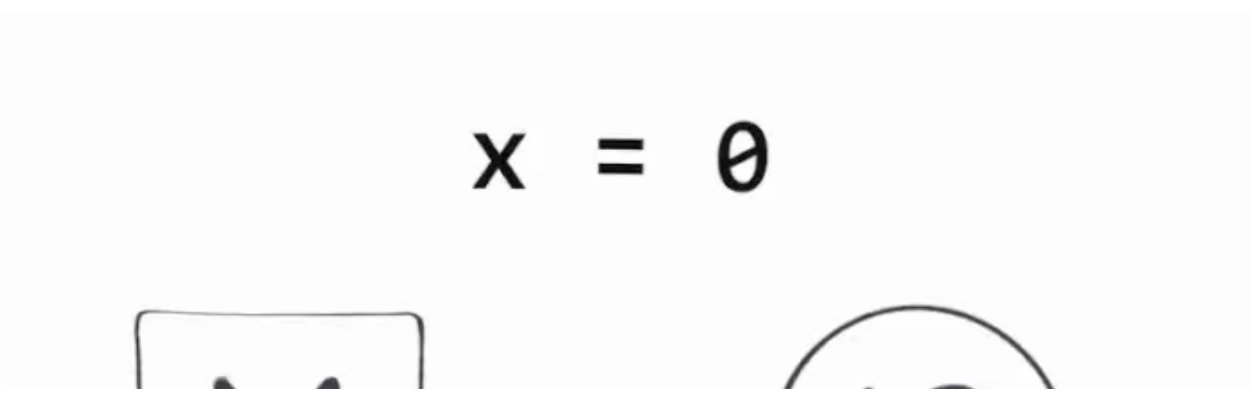
- Declare a variable called `x`.
  - *Draw the `x` variable wire.*
- Assign to `x` the value of `10`.
  - ***Point the `x` wire to the value `10`.***

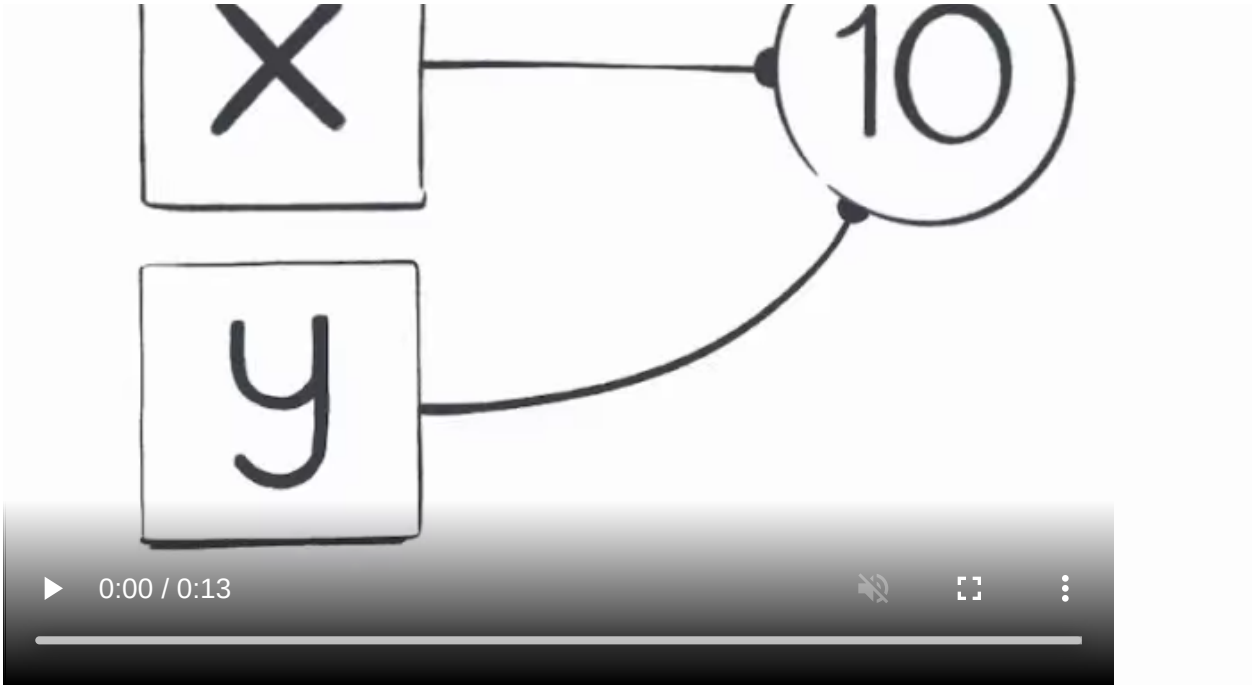


The second line is short, but it does quite a few things:

- Declare a variable called `y`.
  - *Draw the `y` variable wire.*
- Assign to `y` the value of `x`.
  - Evaluate the expression: `x`.
    - *The "question" we want to answer is `x`.*
    - ***Follow the `x` wire—the answer is the value `10`.***
  - The `x` expression resulted in the value `10`.
  - Therefore, assign to `y` the value of `10`.
  - ***Point the `y` wire to the value `10`.***

Finally, we get to the third line:





- Assign to `x` the value of `0`.
  - ***Point the `x` wire to the value `0`.***

After these three lines of code have run, the `x` variable points to the value `0`, and the `y` variable points to the value `10`.

Note that `y = x` did not mean point `y` to `x`. We can't point variables to each other! Variables always point to *values*. When we see an assignment, we "ask" the right side's value, and point the "wire" on the left side to *that value*.

I mentioned in *Mental Models* that it is common to think of variables as boxes. The universe we're building is not going to have any boxes at all. **It only has wires!**

This might seem a bit annoying, but using wires makes it much easier to explain numerous other concepts, like strict equality, object identity, and mutation. We're going to stick with wires, so you might as well start getting used to them now!

Our universe is full of wires.

Recap

- **Primitive values are immutable.** They're a permanent part of our JavaScript universe—we can't create, destroy, or change them. For example, we can't set a property on a string value because it is a primitive value. Arrays are *not* primitive, so we *can* set their properties.
- **Variables are not values.** Each variable *points to* a particular value. We can change *which* value it points to by using the `=` assignment operator.
- **Variables are like wires.** A "wire" is not a JavaScript concept—but it helps us imagine how variables point to values. When we do an assignment, there's always a wire on the left, and an expression (resulting in a value) on the right.
- **Look out for contradictions.** If two things that you learned seem to contradict each other, don't get discouraged. Usually it's a sign that there's a deeper truth lurking underneath.
- **Language matters.** We're building a mental model so that we can be confident in what *can* or *cannot* happen in our universe. We might speak about these ideas in a casual way (and nitpicking is often counterproductive) but our understanding of the meaning behind the terms needs to be precise.

Exercises

This module also has exercises for you to practice!

Don't skip them!

Even though you're likely familiar with the concept of variables, these exercises will help you cement the mental model we're building. We need this foundation

before we can get to more complex topics.

Finished reading?

Mark this episode as learned to track your progress.

Mark as learned

UP NEXT

Take a Quiz →

04 Studying from the Inside

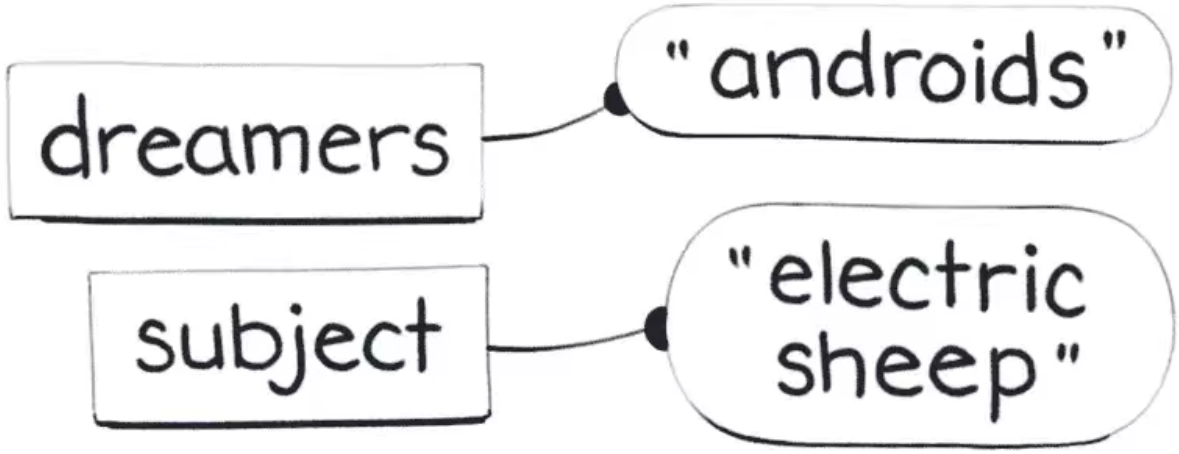
In the next module, we'll take a closer look at our JavaScript universe and the values in it. But before we can get to that, we need to address the elephant in the room. *Is our JavaScript universe even real?*

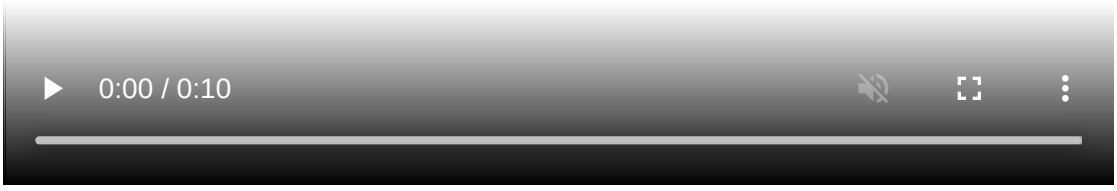
The JavaScript Simulation

I live on my small planet inside our JavaScript universe.

When I ask our universe a question, it answers with a value. I certainly didn't come up with all these values myself. The variables, the expressions, the values—they populate our world. The JavaScript world around me is absolutely real to me.

But sometimes, there is a moment of silence before the next line of code; a brief pause before the next function call. During those moments, I see visions of a world that's much bigger than our JavaScript universe.





In these visions, there are no variables and values; no expressions and no literals. Instead, there are quarks. There are atoms and electrons. There's water and life. Perhaps you're familiar with this universe, too?

There, sentient beings called "humans" use special machines called "computers" to simulate *our* JavaScript universe. Some of them do it for amusement. Some of them do it for profit. Some of them do it for no reason at all. At their whim, our whole universe is created and destroyed a trillion times a day.

Maybe our JavaScript universe isn't so real, after all.

Knowing this, we can study our world in two ways.

Studying From the Outside

One way to study our JavaScript universe would be to *study it from the outside*.

We might focus on how a simulation of our world—a JavaScript engine—"really" works. For example, we might learn that this string of text—*a value in our world*—is a sequence of bytes stored inside a silicon chip.

This approach puts our mental focus on the physical world of people and computers. Our approach is different.

Studying From the Inside

We will be studying our world *from the inside*. Transport yourself mentally into the JavaScript universe and stand next to me. We will observe our universe's laws and perform experiments like physicists in the physical universe.

We will learn about our JavaScript world without thinking about how it's implemented. This is similar to a physicist discussing the properties of a star without questioning whether the *physical* world is real. It doesn't matter! Whether we're studying the physical or the JavaScript universe, we can describe them on their own terms.

Our mental model will not attempt to explain how a value is represented in the computer's memory. The answer changes all the time! The answer even changes while your program is running. If you've heard a simple explanation about how JavaScript "really" represents numbers, strings, or objects in memory, it is most likely wrong.

To us, each string is a value—not a "pointer" or a "memory address"—a *value*. **In our universe, a value is good enough.** Don't allow "memory cells" and other low-level metaphors to distract you from building an accurate high-level mental model of JavaScript. It's turtles all the way down, anyway!

If you're coming from a lower-level language, set aside your intuitions about "passing by reference," "allocating on stack," "copying on write," and so on. These models of how a computer works often make it *harder* to be confident in what can or cannot happen in JavaScript. We'll look at some of the lower-level details, but only where it really matters. They can serve as an *addition* to our mental model, rather than its foundation.

The foundation of our mental model is values. Each value belongs to a type. Primitive values are immutable. We can point to values using "wires" we call variables. This foundation—this understanding of values—will help us continue building our mental model.

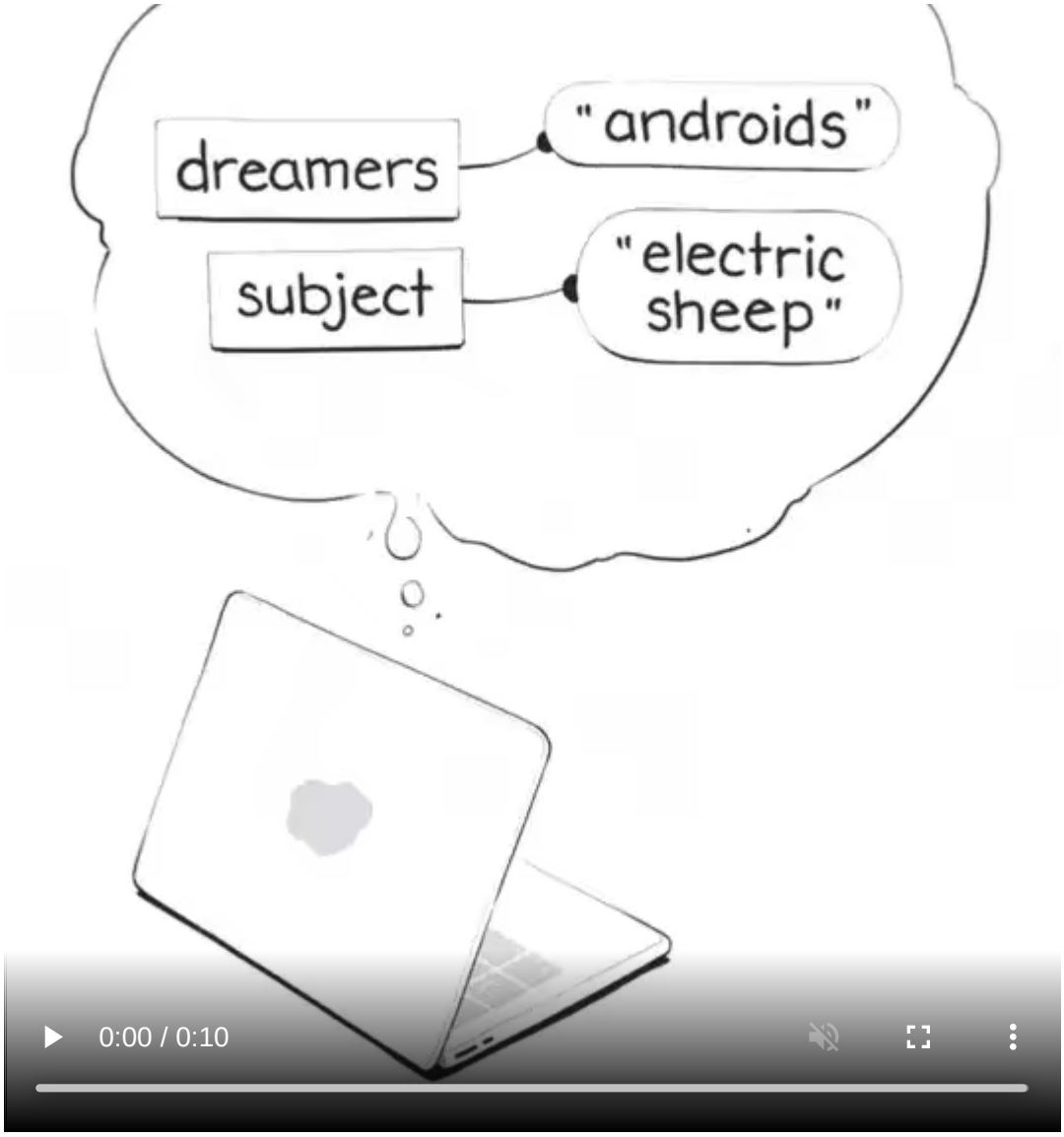
As for these strange visions, I don't give them much thought anymore. I have wires to point and questions to ask. I better get to it!

The stars are bright when I look at them.

Are they still there when I blink?

I shrug.

"Implementation details."



Finished reading?

Mark this episode as learned to track your progress.

Mark as learned

UP NEXT

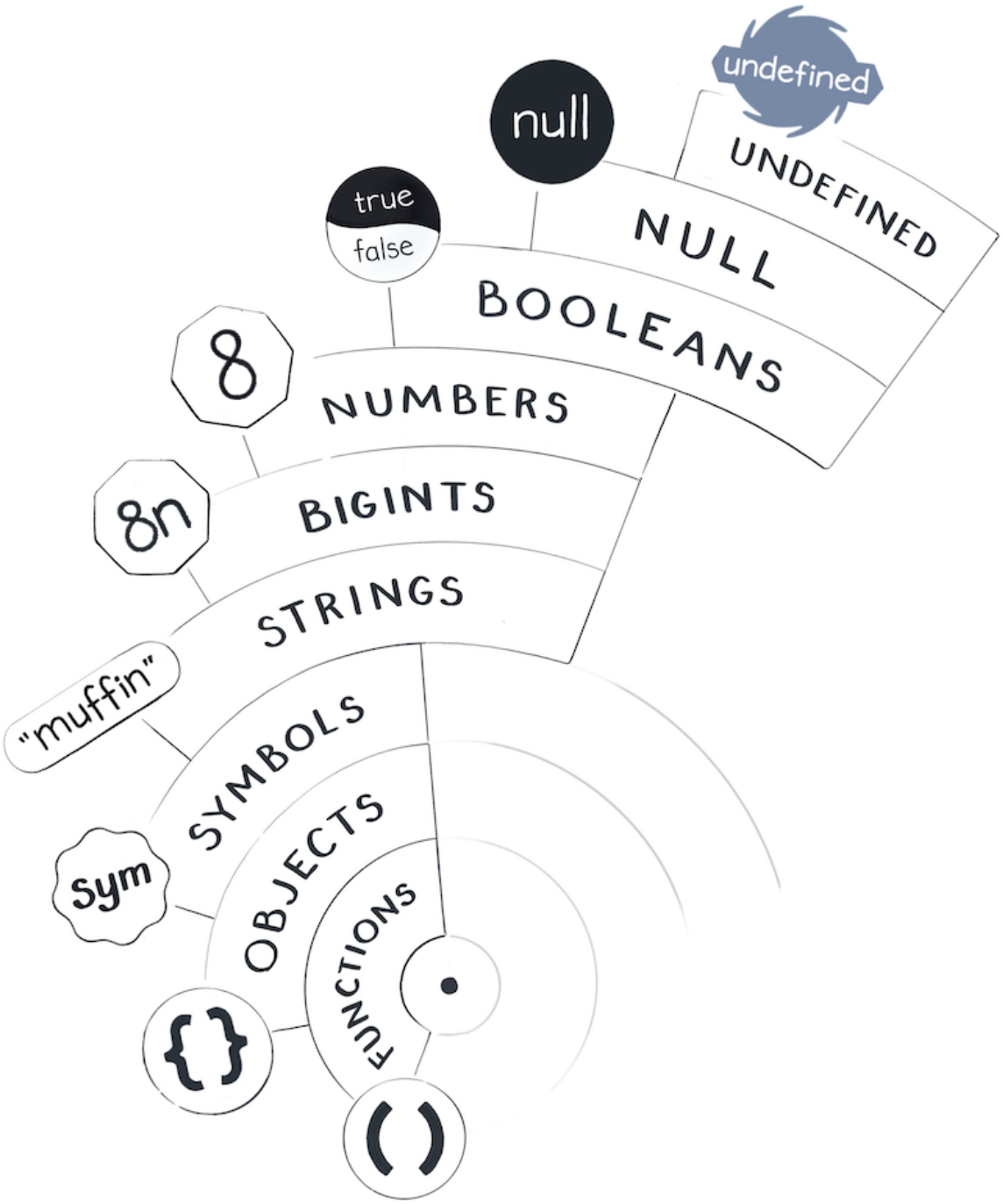
Meeting the Primitive Values →

05 Meeting the Primitive Values

Until now, we have been observing our JavaScript universe from the surface of our planet. We have familiarized ourselves with the values that populate our universe from a distance, but in this module, we're changing that. We're going to hop in a spaceship and go exploring, introducing ourselves to every value in our JavaScript universe.

Spending the time to look at each value in detail might feel unnecessary, but you can only say there are "two apples" when you clearly see that they're two *distinct* apples. Distinguishing values from one another is key to understanding *equality* in JavaScript—which will be our next topic.

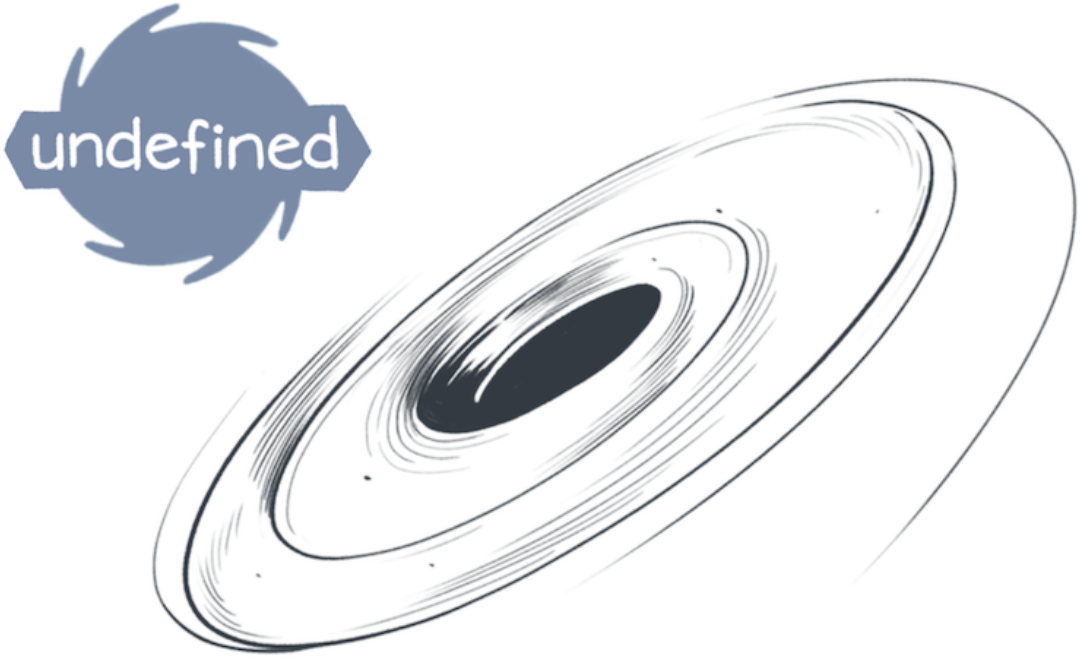
Our spaceship will guide us through the "celestial spheres" of JavaScript to meet different values. We'll meet the primitive values first: Booleans, Numbers, Strings, and so on. Later, we'll meet Objects and Functions. Consider it a sightseeing tour.



Undefined

We'll start our tour with the Undefined type. This is a very straightforward place to start, because **there is only one value of this type—** `undefined` .

```
console.log(typeof(undefined)); // "undefined"
```



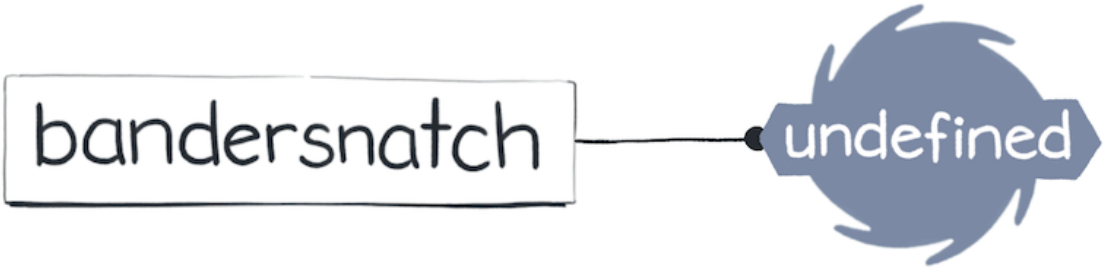
It's called undefined, so you might think it's not there—but it *is* a value, and a very real one! Like a black hole, `undefined` is grumpy and can often spell trouble. For example, reading a property from it will break your program:

```
let person = undefined;
console.log(person.mood); // TypeError!
```

Oh, well. Luckily, there is only one `undefined` in our entire JavaScript universe. You might wonder: why does it exist at all? In JavaScript, it represents the concept of an *unintentionally* missing value.

You could use it in your own code by writing `undefined`—like you write `2` or `"hello"`. However, `undefined` *also* commonly "occurs naturally." It shows up in some situations where JavaScript doesn't know what value you wanted. For example, if you forget to assign a variable, it will point to `undefined`:

```
let bandersnatch;
console.log(bandersnatch); // undefined
```



Then you can point it to another value, or to `undefined` again if you want.

Don't get too hung up on its name. It's tempting to think of `undefined` as some kind of variable status, e.g. "this variable is not yet defined." But that's a completely misleading way to think about it! In fact, if you read a variable that was *actually* not defined (or before the `let` declaration), you will get an error:

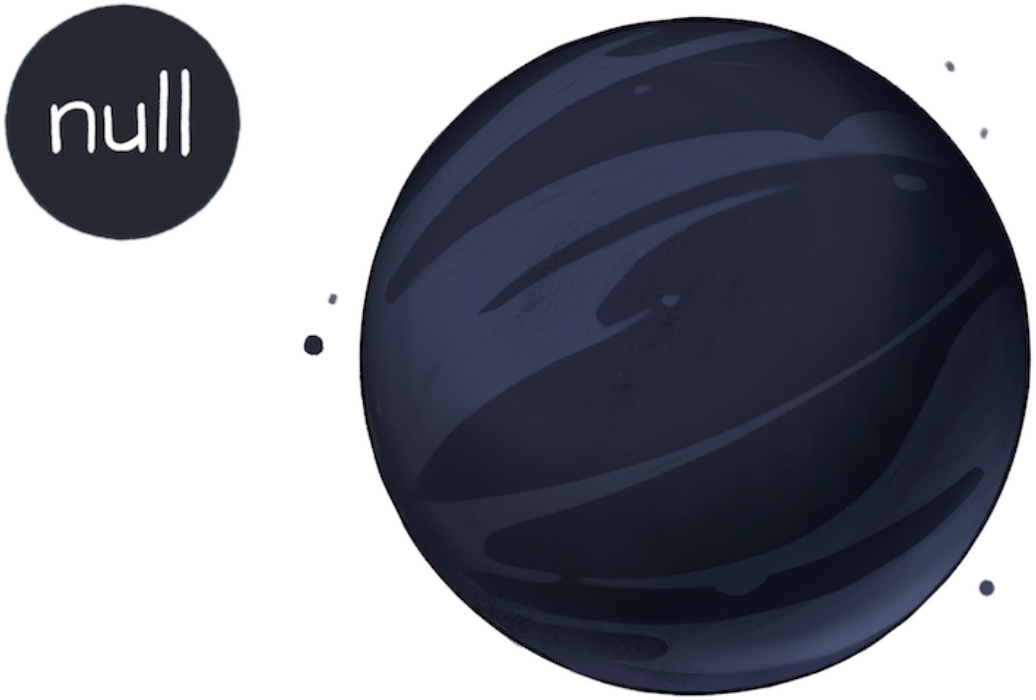
```
console.log(jabberwocky); // ReferenceError!
let jabberwocky;
```

That has nothing to do with undefined.

Really, `undefined` is a regular primitive value, like `2` or `"hello"`.

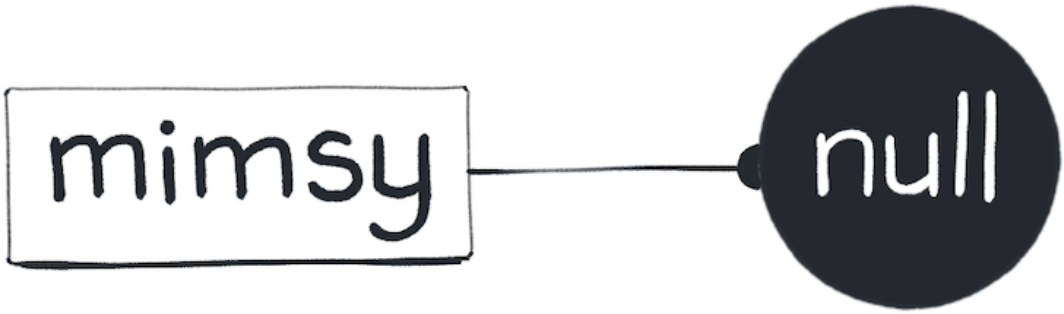
Handle it with care.

Null



The next stop on our tour is Null. You can think of `null` as `undefined`'s sister; **there is only one value of this type—** `null` . It behaves very similarly to undefined. For example, it will also throw a fuss when you try to access its properties:

```
let mimsy = null;
console.log(mimsy.mood); // TypeError!
```



Fun Fact

`null` is the only value of its own type. **However, `null` is also a liar.** Due to a bug in JavaScript, it pretends to be an object:

```
console.log(typeof(null)); // "object" (a lie!)
```

You might think this means `null` is an object. Don't fall into this trap! It is a primitive value, and it doesn't behave in any way like an object.

Unfortunately, `typeof(null)` is a historical accident that we'll have to live with forever.

In practice, `null` is used for *intentionally* missing values. Why have both `null` and `undefined`? This could help you distinguish a coding mistake (which might result in `undefined`) from valid missing data (which you might express as `null`). However, this is only a convention, and JavaScript doesn't enforce this usage. Some people avoid both of them as much as possible!

I don't blame them.

Booleans

Next on our tour, we'll meet booleans! Like day and night or on and off, there are only two boolean values: `true` and `false`.

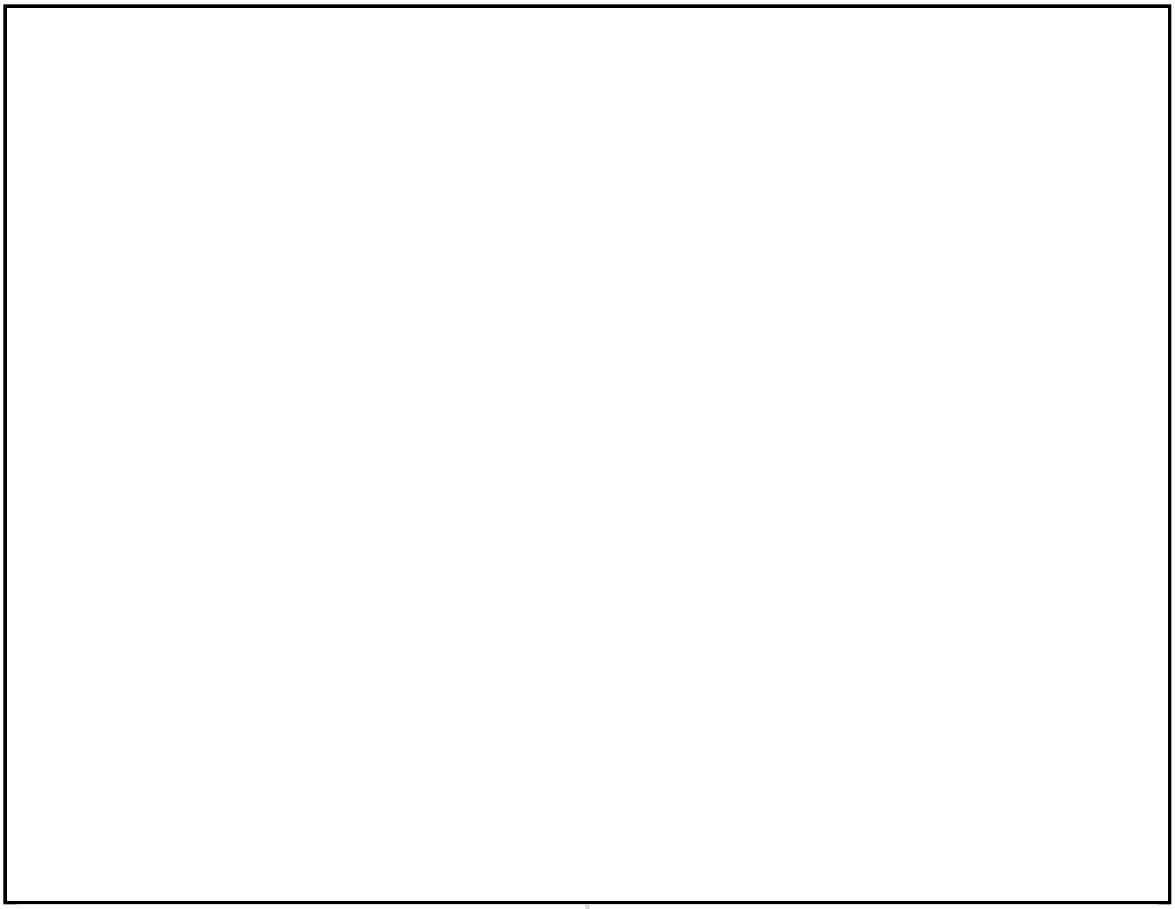
```
console.log(typeof(true)); // "boolean"
console.log(typeof(false)); // "boolean"
```

We can perform logical operations with them:

```
let isSad = true;
let isHappy = !isSad; // The opposite
let isFeeling = isSad || isHappy; // Is at least one of them true?
```

```
let isConfusing = isSad && isHappy; // Are both true?
```

Before continuing our tour of the JavaScript universe, let's check our mental model. Use the sketchpad below or a piece of paper and draw the variables (remember to think of them as wires) and the values for the above lines of code.



Answer

Don't reveal until you have finished sketching.

Reveal

Check your answer. If you got this right, `isFeeling` points to `true`, and `isConfusing` points to `false`. (If you got different answers, there is a mistake somewhere along the way—walk through each line step by step.)

Next, verify that **there is only one `true` and one `false` value on your sketch**. This is important! Regardless of how booleans are stored in

the memory, *in our mental model* there are only two of them.

Numbers

So far, we have introduced ourselves to four values: `null`, `undefined`, `true`, and `false`.

Hold on tight, as we add eighteen quintillion, four hundred and thirty-seven quadrillion, seven hundred and thirty-six trillion, eight hundred and seventy-four billion, four hundred and fifty-four million, eight hundred and twelve thousand, six hundred and twenty-four more values to our mental model!

I am, of course, talking about numbers:

```
console.log(typeof(28)); // "number"
console.log(typeof(3.14)); // "number"
console.log(typeof(-140)); // "number"
```

At first, numbers might seem unremarkable, but let's get to know them a little better!

Math for Computers

JavaScript numbers don't behave exactly the same way as regular mathematical numbers do. Here is a snippet that demonstrates it:

```
console.log(0.1 + 0.2 === 0.3); // false
console.log(0.1 + 0.2 === 0.30000000000000004); // true
```

This might look very surprising! Contrary to popular belief, this doesn't mean that JavaScript numbers are broken. This behavior is common in different programming languages. It even has a name: *floating-point math*.

You see, JavaScript doesn't implement the kind of math we use in real life. Floating-point math is "math for computers." Don't worry too much about this name or how it works exactly. Very few people know about all its subtleties, and that's the point! It works well enough in practice that most of the time you won't think about it. Still, let's take a quick look at what makes it different.

Colors and Numbers

Have you ever used a scanner to turn a physical photo or a document into a digital one? This analogy can help us understand JavaScript numbers.

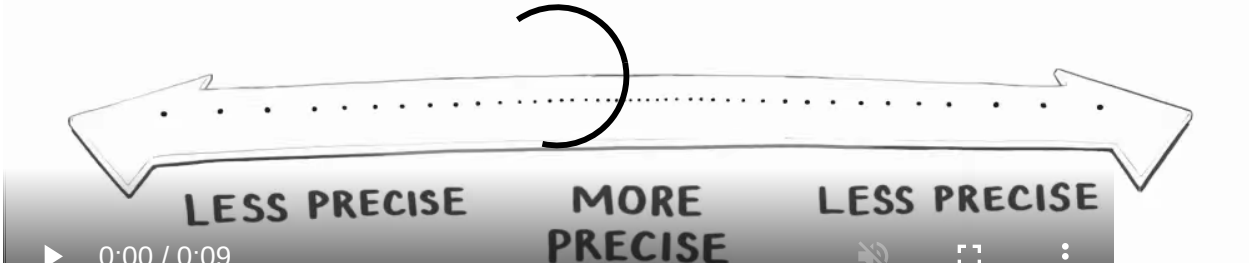
Scanners usually distinguish at most 16 million colors. If you draw a picture with a red crayon and scan it, the scanned image should come out red too—but it will have the *closest* red color our scanner picked from those 16 million colors. So if you have two red crayons with ever so slightly different colors, the scanner might be fooled into thinking their color is exactly the same!

We can say that a scanner uses colors with limited precision.

Floating-point math is similar. In real math, there is an infinite set of numbers. But in floating-point math, **there are only 18 quintillion of them.** So when we write numbers in our code or do calculations with them, JavaScript picks the *closest* numbers that it knows about—just like our scanner does with colors.

In other words, JavaScript uses numbers with limited precision.

We can imagine all of the JavaScript numbers on an axis. The closer we are to `0`, the more precision numbers have, and the closer they "sit" to each other:



This is because relatively small numbers occur more often in our programs, and we usually want them to be precise.

As we move from `0` in either direction, we start losing precision. At some point, even two closest JavaScript numbers stay further apart than by `1`:

```
console.log(Number.MAX_SAFE_INTEGER);     // 9007199254740991
console.log(Number.MAX_SAFE_INTEGER + 1); // 9007199254740992
console.log(Number.MAX_SAFE_INTEGER + 2); // 9007199254740992 (again!)
console.log(Number.MAX_SAFE_INTEGER + 3); // 9007199254740994
console.log(Number.MAX_SAFE_INTEGER + 4); // 9007199254740996
console.log(Number.MAX_SAFE_INTEGER + 5); // 9007199254740996 (again!)
```

Luckily, any **whole** numbers between `Number.MIN_SAFE_INTEGER` and `Number.MAX_SAFE_INTEGER` are exact. This is why `10 + 20 === 30`.

But when we write `0.1` or `0.2`, we don't get *exactly* `0.1` and `0.2`. We get the closest available numbers in JavaScript. They are almost exactly the same, but there might be a tiny difference. These tiny differences add up, which is why `0.1 + 0.2` doesn't give us *exactly* the same number as writing `0.3`.

If this is still confusing, don't worry. You can learn more about floating-point math, but you already know more than I did when I started writing this guide! Unless you work on finance apps, you likely won't need to worry about this.

Special Numbers

It is worth noting that floating-point math includes a few *special numbers*. You might occasionally run into `NaN`, `Infinity`, `-Infinity`, and `-0`. They exist because sometimes you might execute operations like `1 / 0`, and JavaScript needs to represent their result somehow. The floating-point math standard specifies how they work, and what happens when you use them.

Here's how special numbers may come up in your code:

```
let scale = 0;
let a = 1 / scale; // Infinity
let b = 0 / scale; // NaN
let c = -a; // -Infinity
let d = 1 / c; // -0
```

Out of these special numbers, `NaN` is particularly interesting. `NaN`, which is the result of `0 / 0` and some other invalid math, stands for "not a number."

You might be confused by why it claims to be a number:

```
console.log(typeof(NaN)); // "number"
```

However, there is no trick here. From a JavaScript perspective, `NaN` *is* a numeric value. It is not null, undefined, a string, or some other type. But in the floating-point math, the *name* for that term is "not a number". So it *is* a numeric value. It happens to be called "not a number" because it represents an invalid result.

It's uncommon to write code using these special numbers. However, they might come up due to a coding mistake. So it's good to know they exist.

BigInts

If numbers expanded our JavaScript universe, the next stop on our tour, BigInts, will really keep us busy exploring. In fact, we could explore them forever!

Regular numbers can't represent large integers with precision, so BigInts fill that gap (literally). How many BigInts are there in our universe? The specification says they have *arbitrary precision*. This means that **in our JavaScript universe, there is an infinite number of BigInts—one for each integer in math**.

If this sounds strange, consider that you're already comfortable with the idea of there being infinite integers in math. (If you're not, give it a few moments!) It's not much of a leap then from a "math universe" to a "JavaScript universe."

(And from there, we can go straight to the Pepsi Universe.)

You can see the idea of arbitrary precision illustrated here:

```
let alot = 9007199254740991n; // n at the end makes it a BigInt!
console.log(alot + 1n); // 9007199254740992n
console.log(alot + 2n); // 9007199254740993n
console.log(alot + 3n); // 9007199254740994n
console.log(alot + 4n); // 9007199254740995n
console.log(alot + 5n); // 9007199254740996n
```

No funny business with the rounding! BigInts are great for financial calculations where precision is especially important.

But keep in mind that nothing is free. Operations with *truly* huge numbers may take time and resources—we can't fit all the possible BigInts inside the computer memory. If we tried, at some point it would crash or freeze. But conceptually, we could tour our JavaScript universe for eternity and never reach every single BigInt.

BigInts were only recently added to JavaScript, so you won't see them used widely yet and if you use an older browser, they won't work.

Strings

Our next tour stop is Strings, which represent text in JavaScript. There are three ways to write strings (single quotes, double quotes, and backticks), but they refer to the same concept. These three string literals result in the same string value:

```
console.log(typeof("こんにちは")); // "string"
console.log(typeof('こんにちは')); // "string"
console.log(typeof(`こんにちは`)); // "string"
```

An empty string is a string, too:

```
console.log(typeof('')); // "string"
```

Strings Aren't Objects

All strings have a few built-in properties.

```
let cat = 'Cheshire';
console.log(cat.length); // 8
console.log(cat[0]); // "C"
console.log(cat[1]); // "h"
```

This doesn't mean that strings are objects! String properties are special and don't behave the way object properties do. For example, you can't assign anything to `cat[0]`. Strings are primitives, and all primitives are immutable.

A Value for Every Conceivable String

In our universe, there is a distinct value for every conceivable string. Yes, this includes your grandmother's maiden name, the fanfic you published ten years ago under an alias, and the script of Matrix 5 which hasn't been written yet.

Of course, all possible strings can't literally fit inside a computer memory chip. But *the idea* of every possible string can fit inside your head. Our JavaScript universe is a model for humans, not for computers!

This might prompt a question. Does this code *create* a string?

```
// Try it in your console
let answer = prompt('Enter your name');
console.log(answer); // ?
```

Or does it merely *summon* a string that already exists in our universe?

The answer to this question depends on whether we're studying JavaScript "from the outside" or "from the inside."

Outside our mental model, the answer depends on a specific implementation. Whether a string is represented as a single block of memory, multiple blocks, or a special data structure like a rope, is up to the JavaScript engine.

But as we discussed in Studying from the Inside, we have agreed to talk about the JavaScript universe as if we lived inside of it. We won't make statements about it that we can't verify from inside the universe—by running some code.

The question of whether a string already "exists" or is "created" is not something we can test from the code. *Inside* our mental model, this question doesn't mean anything. We can't set up an experiment to say whether strings "get created" or "get summoned" within our JavaScript universe.

To keep our mental model simple, we will say that **all conceivable string values already exist from the beginning—one value for every distinct string.**

Symbols

We've already explored quite a bit of our JavaScript universe, but there is just one more (quick) stop on the first part of our tour: Symbols.

```
let alohomora = Symbol();
console.log(typeof(alohomora)); // "symbol"
```

It's important to know that Symbols exist, but it's hard to explain their role and behavior without diving deeper into objects and properties. Symbols serve a similar purpose to door keys: they let you hide away some information inside an object and control which parts of the code can access it. They are also relatively rare, so in this tour of our universe, we're going to skip them. Sorry, symbols!

To Be Continued

Now that we've met all of the primitive values, we'll take a small break from our tour. Let's recap the primitive values we've encountered so far!

- **Undefined**
- **Null**
- **Booleans**
- **Numbers**
- **BigInts**
- **Strings**
- **Symbols**

We've also learned a few interesting facts about JavaScript numbers:

- **Not all numbers can be perfectly represented in JavaScript.** Their decimal part offers more precision closer to `0`, and less precision further away from it.
- **Numbers from invalid math operations like `1 / 0` or `0 / 0` are special.** `NaN` is one of such numbers. They may appear due to coding mistakes.
- **`typeof(NaN)` is a number because `NaN` is a numeric value.** It's called "Not a Number" because it represents the *idea* of an "invalid" number.

We will continue our sightseeing tour by meeting the non-primitive values, specifically, objects and functions.

Exercises

This module also has exercises for you to practice!

Don't skip them!

Even though you're likely familiar with the concept of primitive values, these exercises will help you cement the mental model we're building. We need this foundation before we can get to more complex topics.

Finished reading?

Mark this episode as learned to track your progress.

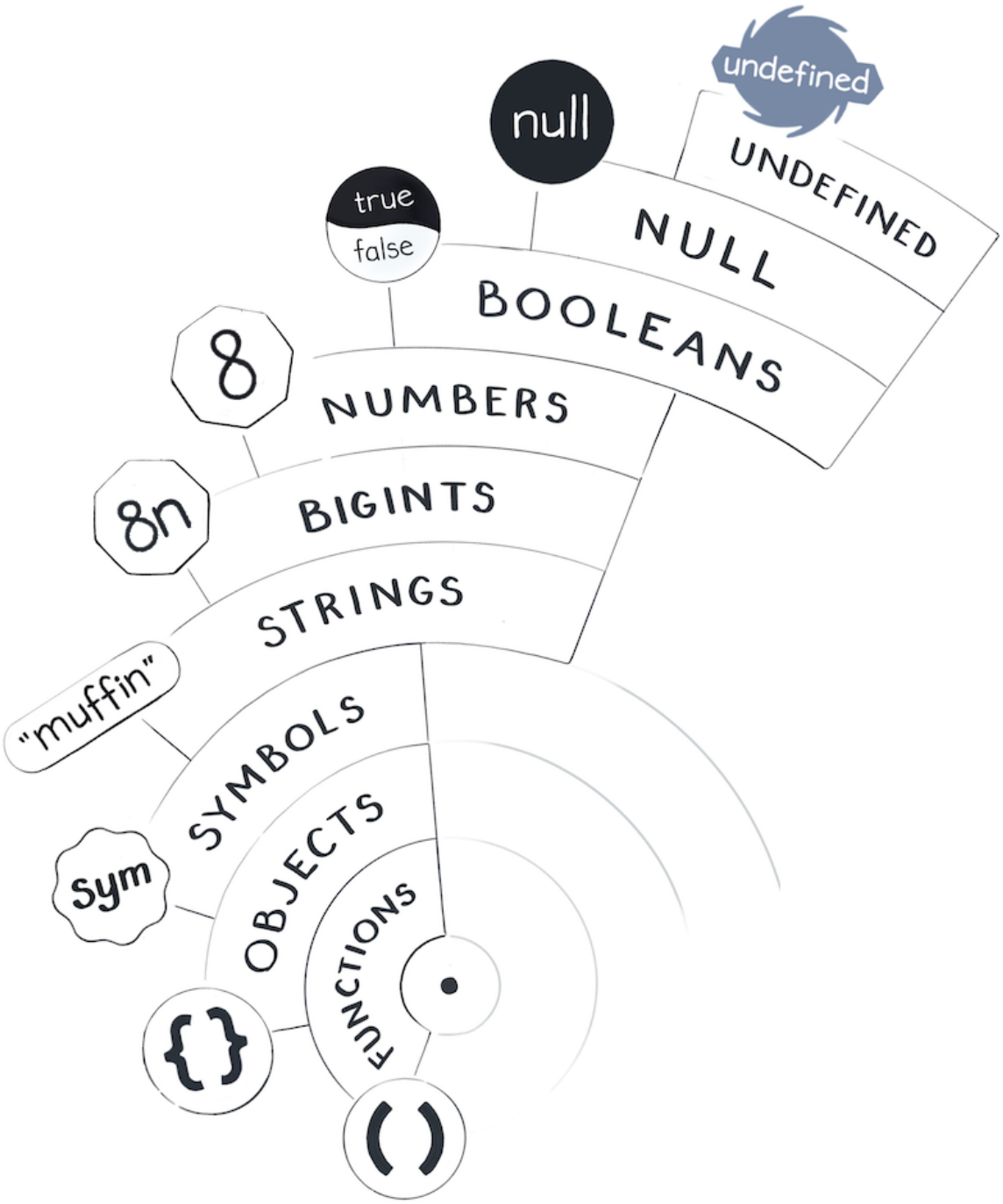
Mark as learned

UP NEXT

Take a Quiz →

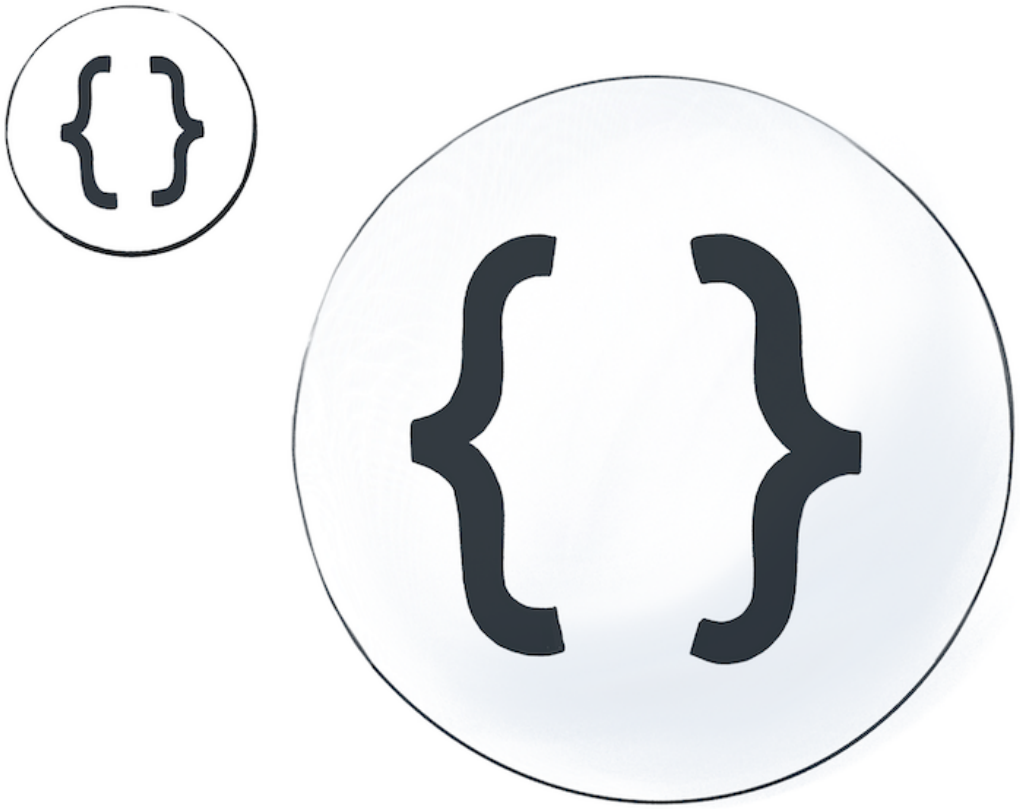
06 Meeting Objects and Functions

Without further ado, let's resume the tour of our JavaScript universe!



In the previous module, we've looked at primitive values: Undefined, Null, Booleans, Numbers, BigInts, and Strings. We will now introduce ourselves to non-primitive values—these are types that let us *make our own values*.

Objects



At last, we got to objects!

This includes arrays, dates, RegExps, and other non-primitive values:

```
console.log(typeof({})); // "object"
console.log(typeof([])); // "object"
console.log(typeof(new Date())); // "object"
console.log(typeof(/\d+/)); // "object"
console.log(typeof(Math)); // "object"
```

Unlike everything before, objects are *not* primitive values. This also means that by default, they're mutable (we can change them). We can access their properties with `.` or `[]`:

```
let rapper = { name: 'Malicious' };
rapper.name = 'Malice'; // Dot notation
rapper['name'] = 'No Malice'; // Bracket notation
```

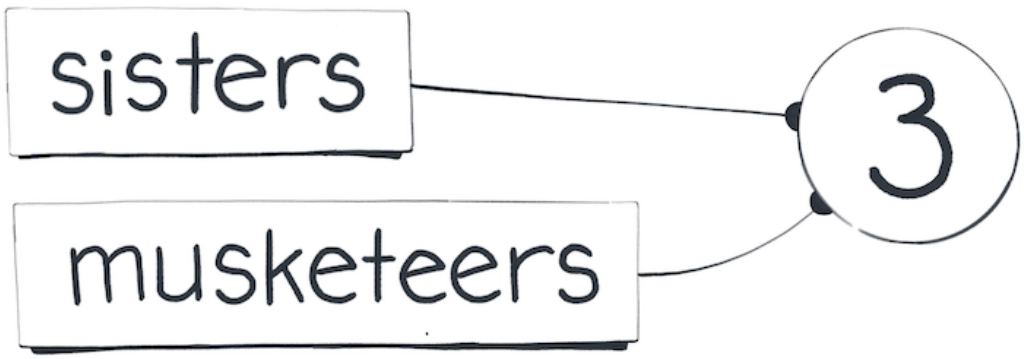
We haven't talked about properties in detail yet, so your mental model about them might be fuzzy. We will return to properties in a future module.

Making Our Own Objects

There is one thing in particular that makes objects exciting and unique. **We can make more of them! We can populate our JavaScript universe with our own objects.**

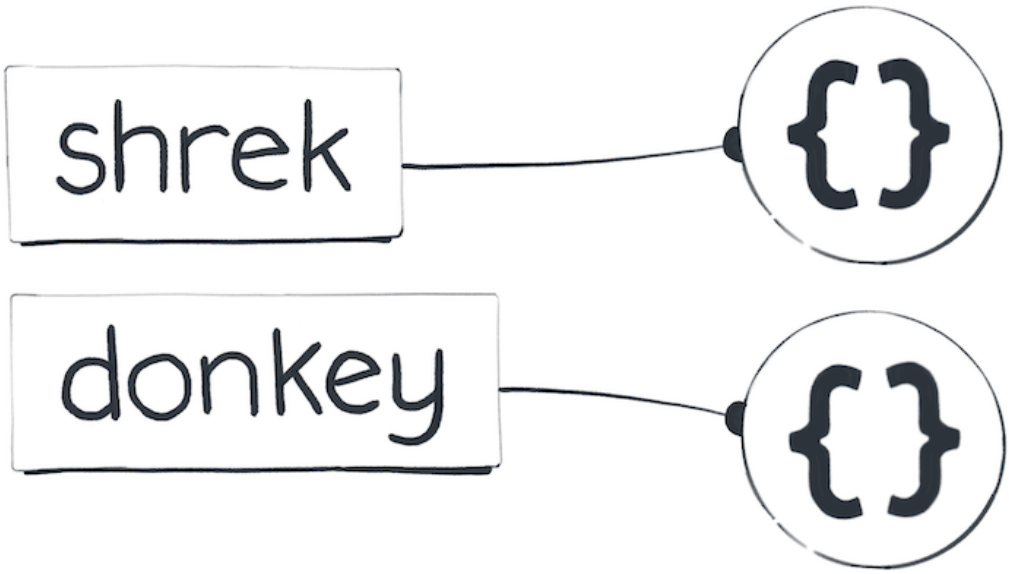
In our mental model, all of the primitive values we've discussed— `null`, `undefined`, booleans, numbers, and strings—have "always existed." We can't create a new string or a new number, we can only "summon" that value:

```
let sisters = 3;
let musketeers = 3;
```



What makes objects different is that we *can* create more of them. **Every time we use the `{}` object literal, we *create* a brand new object value:**

```
let shrek = {};
let donkey = {};
```



The same goes for arrays, dates, and any other objects. For example, the `[]` array literal *creates* a new array value—a value that never existed before.

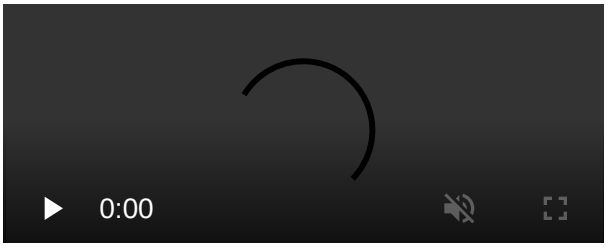
Do Objects Disappear?

You might wonder: do objects ever disappear, or do they hang around forever? JavaScript is designed in a way that we can't tell one way or the other from inside our code. For example, we can't *destroy* an object:

```
let junk = {};
junk = null; // Doesn't necessarily destroy an object
```

Instead, JavaScript is a garbage-collected language.

This means that although we can't destroy an object, it *might* eventually "disappear" if there is no way to reach it by following the wires from our code.



JavaScript doesn't offer guarantees about *when* garbage collection happens.

Unless you're trying to figure out why an app is using too much memory, you don't need to think about garbage collection too often. I only mention it here so that you know that we can create objects—but we cannot destroy them.

In our universe, objects and functions float closest to our code. This reminds us that we can manipulate them and even make more of them.

Functions

We've reached the last stop on our tour!

It is particularly strange to think about functions as values that are separate from my code. After all, they *are* my code. Or are they not?

Functions are Values

We define functions so that we can call them later and run the code inside them. However, to really understand functions in JavaScript, we need to forget about *why* they're useful for a second. Instead, we will think about functions as yet another kind of value: a number, an object, *a function*.

To understand functions, we will compare them to numbers and objects.

First, consider this `for` loop that runs `console.log(2)` seven times:

```
for (let i = 0; i < 7; i++) {
  console.log(2);
}
```

How many different values does it pass to `console.log`? To answer this, let's recall what `2` means when we write it down. It is a number literal. A literal is an expression—a question to our universe. There is only one value for every number in our universe, so it "answers" our question by "summoning" the same value for the number `2` every time. **So the answer is one value.** We will see the log seven times—but we are passing the same value in each call.

Now let's briefly revisit objects.

Here is another `for` loop that runs `console.log({})` seven times:

```
for (let i = 0; i < 7; i++) {
  console.log({});
}
```

How many different values does it pass to `console.log` now? Here, too, `{}` is a literal—except it's an *object literal*. As we just learned, our JavaScript universe doesn't "answer" an object literal by summoning anything. Instead, it *creates* a new object value—which will be the result of the `{}` object literal. **So the code above creates and logs seven completely distinct object values.**

Let that sink in.

Now have a look at functions.

```
for (let i = 0; i < 7; i++) {
  console.log(function() {});
}
```

How many different values does this code pass to `console.log`?

Answer

The answer Don't reveal until you have decided on an answer.

Every time we execute a line of code that contains a function expression, a brand new fun ears in our universe.

Reveal

Here, too, `function() {}` is an expression. Like any expression, a function expression is a "question" to our JavaScript universe—**which answers us by *creating* a new function value every time we ask**. This is very similar to how `{}` creates a new object value when it executes. Functions are like objects!

Technically, functions *are* objects in JavaScript. We'll keep treating them as a separate fundamental type because they have unique capabilities compared to regular objects. But, generally speaking, if you can do something to an object, you can also do that to a function too. They are very special objects.

Calling a Function

What does this code print?

```
let countDwarves = function() { return 7; };
let dwarves = countDwarves;
console.log(dwarves);
```

You might think that it prints `7`, especially if you're not looking very closely.

Now check this snippet in the console! The exact thing it prints depends on the browser, but you will see *the function itself* instead of the number `7` there.

If you follow our mental model, this behavior should make sense:

1. First, we created a new function value with a `function() { }` expression, and pointed the `countDwarves` variable at this value.
2. Next, we pointed the `dwarves` variable at the value that `countDwarves` is pointing to—which is the same function value.
3. Finally, we logged the value that `dwarves` is currently pointing to.

At no point did we call our function!

As a result, both `countDwarves` and `dwarves` point to the same value, which happens to be a function. See, *functions are values*. We can point variables to them, just like we can do with numbers or objects.

Of course, if we want to *call* a function, we can do that too:

```
let countDwarves = function() { return 7; };
let dwarves = countDwarves(); // () is a function call
console.log(dwarves);
```

Note that neither the `let` declaration nor the `=` assignment have anything to do with our function call. It's `()` that performs the function call—and it alone!

Adding `()` changes the meaning of our code:

- `let dwarves = countDwarves` means "Point `dwarves` to the value that `countDwarves` is currently pointing to."
- `let dwarves = countDwarves()` means "Point `dwarves` to the value **returned by** the function that `countDwarves` is currently pointing to."

In fact, `countDwarves()` is also an expression. It's known as a *call expression*. To "answer" a call expression, JavaScript runs the code inside our function, and hands us the returned value as the result (in this example, it's `7`).

Recap

That was quite a journey! Over the last two modules, we have looked at every value type in JavaScript. Let's recap each type of value that we encountered, starting with the different primitive types:

- **Undefined**
- **Null**
- **Booleans**
- **Numbers**
- **BigInts**
- **Strings**
- **Symbols**

Then, there are the special types below, which let us *make our own values*:

- **Objects**
- **Functions**

It was fun to visit the different "celestial spheres" of JavaScript. Now that we've introduced ourselves to all the values, we've also learned what makes them *distinct* from one another.

Primitive values (strings, numbers, etc...), which we encountered in the first part of our tour, have always existed in our universe. **For example, writing `2` or `"hello"` always "summons" the same number or a string value. Objects and functions behave differently and allow us to generate our own values. Writing `{}` or `function() {}` always *creates* a brand new, different value.** This idea is crucial to understanding equality in JavaScript, which will be our next topic.

Exercises

This module also has exercises for you to practice!

Don't skip them!

Even though you're likely familiar with the concept of objects and functions, these exercises will help you cement the mental model we're building. We need this foundation before we can get to more complex topics.

Finished reading?

Mark this episode as learned to track your progress.

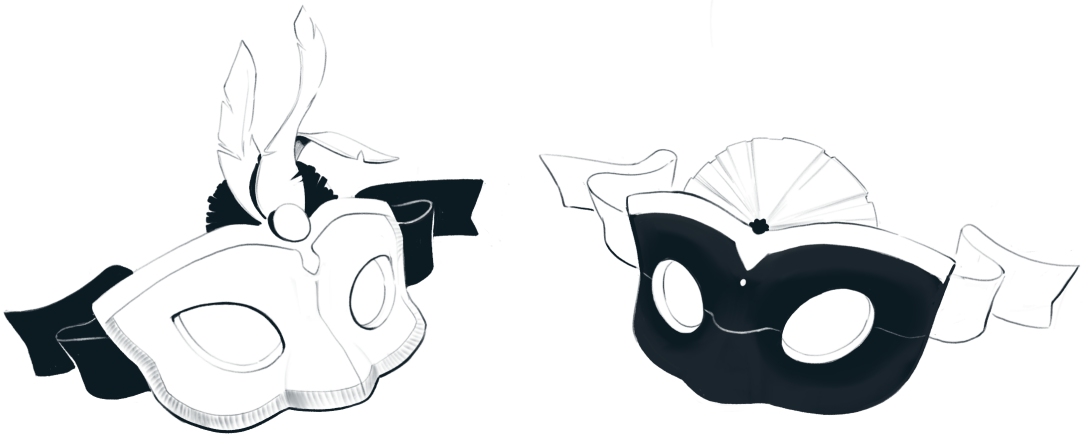
UP NEXT

Take a Quiz →

07 Equality of Values

It's time to talk about equality in JavaScript! Here's why it matters.

Imagine attending a masked carnival with only a few types of masks to choose from. Groups of people with the exact same outfit would be joking, dancing, and moving around the room. It would be confusing! You might talk to two people, and not realize that you really talked to the same person twice. Or you might *think* you talked to one person when, in reality, you talked to two different people!



If you don't have a clear mental model of equality in JavaScript, every day is like a carnival—and not in a good way. You're never quite sure if you're dealing with the same value, or with two different values. As a result, you'll often make mistakes—like changing a value you didn't intend to change.

Luckily, we've already done most of the work to establish the concept of equality in JavaScript. It fits into our mental model in a very natural way.

Kinds of Equality

In JavaScript, there are several kinds of equality. If you've been writing JavaScript for a while, you're probably familiar with at least two of them:

- **Strict Equality:** `a === b` (triple equals).
- **Loose Equality:** `a == b` (double equals).
- **Same Value Equality:** `Object.is(a, b)`.

Most tutorials don't mention *same value equality* at all. We'll take a road less traveled, and explain it first. We can then use it to explain the other kinds of equality.

Same Value Equality: `Object.is(a, b)`

In JavaScript, `Object.is(a, b)` tells us if `a` and `b` are the same value:

```
console.log(Object.is(2, 2)); // true
console.log(Object.is({}, {})); // false
```

This is called **same value equality**.

> ## Fun Fact
>
> Despite `Object` in the method name, `Object.is` is not specific to objects. It can compare any two values, whether they are objects or not!

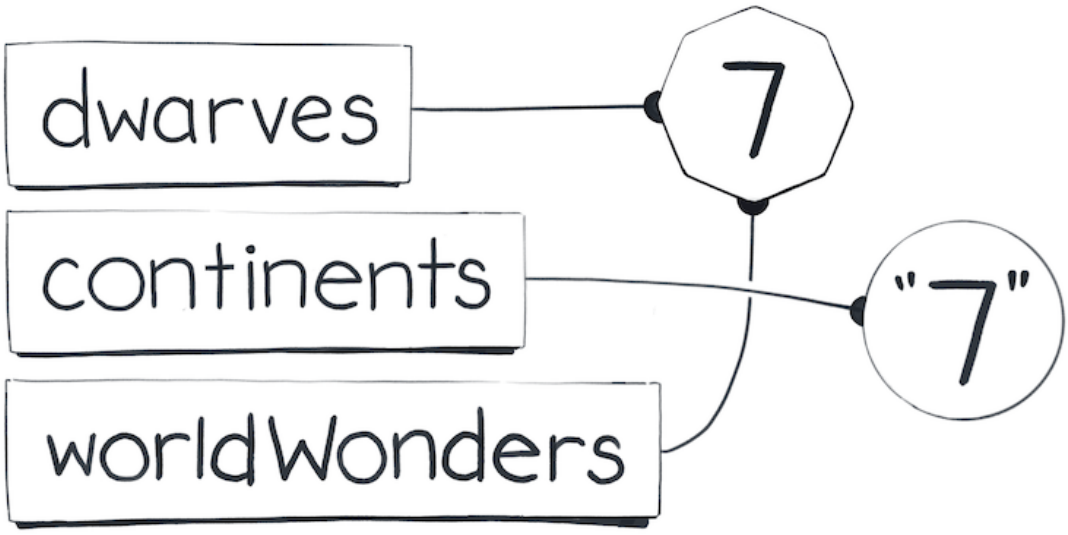
What does "same value" mean, exactly, in our mental model? You might already know this intuitively, but let's verify your understanding.

Check Your Intuition

Consider this example from our earlier exercises:

```
let dwarves = 7;
let continents = '7';
let worldWonders = 3 + 4;
```

As a reminder, our sketch for this snippet looked like this:



Now try to answer these questions **using the diagram above**:

```
console.log(Object.is(dwarves, continents)); // ?
console.log(Object.is(continents, worldWonders)); // ?
console.log(Object.is(worldWonders, dwarves)); // ?
```

Write down your answers and think about *how* you would explain them.

Answer

Don't reveal until you have finished writing.

Reveal

This was not a [illegible]

1. `Object.is(dwarves, c[illegible]` `false` because `dwarves` and `continents` **point to different values**.
2. `Object.is(continents, worldWonders)` is `false` because `continents` and `worldWonders` **point to different values**.
3. `Object.is(worldWonders, dwarves)` is `true` because `worldWonders` and `dwarves` **point to the same value**.

If two values are represented by a single shape on our diagram, it means that they aren't really *two* different values. They are the same value! In those instances, `Object.is(a, b)` returns `true`.

In the previous module, we explored the types of values in our JavaScript universe. As we got to know these values, we were learning what makes them distinct from one another. As a result, we also learned the opposite—what it means for values to be the same.

If you struggle with this idea, you might want to revisit our celestial tour of values and work through the exercises again. It *will* make sense, I promise!

But What About Objects?

By this point, you might be worried about objects. You might have heard that equality doesn't work with objects, or that it compares "references." **If you have existing intuitions like these, set them aside completely for a moment.**

Instead, look at this code snippet:

```
let banana = {};
let cherry = banana;
let chocolate = cherry;
cherry = {};
```

Open a notepad or use the sketchpad below and draw a diagram of variables and values. You'll want to draw it step by step, as it's hard to do in your head.

Remember that `{}` always means "create a new object value." Also, remember that `=` means "point the wire on the left side to the value on the right side."

After you finish drawing, write down your answers to these questions:

```
console.log(Object.is(banana, cherry)); // ?
console.log(Object.is(cherry, chocolate)); // ?
console.log(Object.is(chocolate, banana)); // ?
```

Make sure to **use your diagram** to answer them.

Answer

Don't reveal until you have finished sketching and writing.

Reveal

1. `let banana = {};`
   - Declare a `banana` variable.
   - Create a new object value `{}`.
   - Point the `banana` wire to it.
2. `let cherry = banana;`
   - Declare a `cherry` variable.
   - Point the `cherry` wire to where `banana` is pointing.
3. `let chocolate = cherry;`
   - Then, we declare a `chocolate` variable.
   - Point the `chocolate` wire to where `cherry` is pointing.
4. `cherry = {};`
   - Create a new object value `{}`.
   - Point the `cherry` wire to it.

After the last step, your diagram should look like this:

Now let's check your answers:

1. `Object.is(banana, cherry)` is `false` because `banana` and `cherry` point to different values.
2. `Object.is(cherry, chocolate)` is `false` because `cherry` and `chocolate` point to different values.
3. `Object.is(chocolate, banana)` is `true` because `chocolate` and `banana` point to the same value.

As you can see, we didn't need any additional concepts to explain how **same value equality** works for objects. It comes naturally using our mental model. And that's all there is to know about it!

Strict Equality: `a === b`

You have probably used the **strict equality** operator before:

```
console.log(2 === 2); // true
console.log({} === {}); // false
```

There is also a corresponding opposite `!==` operator.

Same Value Equality vs. Strict Equality

So what's the difference between `Object.is` and `===`?

Same value equality— `Object.is(a, b)` —has a direct meaning in our mental model. It corresponds to the idea of "the same value" in our universe.

In almost all cases, the same intuition works for **strict value equality**. For example, `2 === 2` is `true` because `2` always "summons" the same value:

Conversely, `{} === {}` is `false` because each `{}` *creates* a different value:

In the above examples, `a === b` behaves the same way as `Object.is(a, b)`. However, there are **two rare cases** where the behavior of `===` is different.

Remember memorizing irregular verbs when you were learning English? The cases below are similar—**consider them as exceptions to the rule.**

Both of these unusual cases involve "special numbers" we discussed in the past:

1. `NaN === NaN` **is** `false`, although they are the same value.
2. `-0 === 0` **and** `0 === -0` **are** `true`, although they are different values.

These cases are uncommon, but we'll still take a closer look at them.

First Special Case: `NaN`

As we saw in our celestial tour of values, `NaN` is a special number that shows up when we do invalid math like `0 / 0`:

```
let width = 0 / 0; // NaN
```

Further calculations with `NaN` will give you `NaN` again:

```
let height = width * 2; // NaN
```

You probably won't do this intentionally, but it can happen if there is a flaw in your data or calculations.

Remember that `NaN === NaN` **is always** `false`**:**

```
console.log(width === height); // false
```

However, `NaN` is the *same value* as `NaN`:

```
console.log(Object.is(width, height)); // true
```

That's confusing.

The reason for `NaN === NaN` being `false` is largely historical, so I suggest accepting it as a fact of life. You might run into this if you try to write some code that checks a value for being `NaN` (for example, to print a warning).

```
function resizeImage(size) {
  if (size === NaN) {
    // This will never get logged: the check is always false!
    console.log('Something is wrong.');
  }
  // ...
}
```

Instead, here are a few ways (they all work!) to check if `size` is `NaN`:

- `Number.isNaN(size)`
- `Object.is(size, NaN)`
- `size !== size`

The last one might be particularly surprising. Give it a few moments. If you don't see how it detects `NaN`, try re-reading this section and thinking about it again.

Answer

Don't reveal until you figured out the answer or gave it a few minutes.

Reveal

Fun Fact

A quick historical anecdote: ensuring developers could detect `NaN` this way was one of the original reasons for making `NaN === NaN` return `false`! This was decided before JavaScript even existed.

Second Special Case: `-0`

In regular math, there is no such concept as "minus zero," but it exists in floating-point math for practical reasons. Here's an interesting fact about it.

Both `0 === -0` and `-0 === 0` are always `true`:

```
let width = 0; // 0
let height = -width; // -0
console.log(width === height); // true
```

However, `0` is a *different value* from `-0`:

```
console.log(Object.is(width, height)); // false
```

That's confusing too.

In practice, I haven't run into a case where this matters in my entire career.

Coding Exercise

Now that you know how `Object.is` and `===` work, I have a small coding exercise for you. You don't have to complete it, but it's a fun brainteaser.

Write a function called `strictEquals(a, b)` that returns the same value as `a === b`. Your implementation must not use the `===` or `!==` operators.

Here is my answer if you want to check yourself. This function is utterly useless, but writing it helps make sense of `===`.

Don't Panic

Hearing about these special numbers and how they behave can be overwhelming. Don't stress too much about these special cases!

They're not very common. Now that you know they exist, you will recognize them in practice. In most cases, we can trust our "equality intuition" for both `Object.is(a, b)` and `a === b`.

Loose Equality

Finally, we get to the last kind of equality.

Loose equality (double equals) is the bogeyman of JavaScript. Here are a couple of examples to make your skin crawl:

```
console.log([[]] == ''); // true
console.log(true == [1]); // true
console.log(false == [0]); // true
```

Wait, what?!

The rules of **loose equality** (also called "abstract equality") are arcane and confusing. Many coding standards prohibit the use of `==` and `!=` in code altogether.

Although Just JavaScript doesn't take strong opinions on what features you should or shouldn't use, we're not going to cover **loose equality** in much detail. It's uncommon in modern codebases, and its rules don't play a larger role in the language—or in our mental model.

Fun Fact

The rules of loose equality are widely acknowledged as an early bad design decision of JavaScript, but if you are still curious, you can check out how it works here. Don't feel pressured to memorize it—you'll need that memory for other topics!

The one relatively common use case worth knowing is:

```
if (x == null) {
  // ...
}
```

This code is equivalent to writing:

```
if (x === null || x === undefined) {
  // ...
}
```

However, even that usage of `==` might be controversial on some teams. It's best to discuss how much `==` is tolerated in your codebase as a team before using it.

Recap

- JavaScript has several kinds of equality. They include **same value equality**, **strict equality**, and **loose equality**.
- **Same value equality**, or `Object.is(a, b)`, matches the concept of the *sameness of values* that we introduced in the previous module.

    - Understanding this kind of equality helps prevent bugs! You will often need to know when you're dealing with the same value, and when you're dealing with two different values.
    - When we draw a diagram of values and variables, the *same value* cannot appear twice. `Object.is(a, b)` is `true` when variables `a` and `b` point to the same value on our diagram.
    - **Same value equality** is verbose and a bit annoying to write, but it's also the easiest to explain, which is why we started with it.

- In practice, you will use **strict equality**, or `a === b`, most often. It is equivalent to the **same value equality** except for two rare special cases:
    - `NaN === NaN` is `false`, even though they are the same value.
    - `0 === -0` and `-0 === 0` is `true`, but they are different values.
- You can check whether `x` is `NaN` using `Number.isNaN(x)`.
- **Loose equality** (`==`) uses a set of arcane rules and is often avoided.

Exercises

This module also has exercises for you to practice!

Don't skip them!

Even though you're likely familiar with the concept of equality, these exercises will help you cement the mental model we're building. We need this foundation before we can get to more complex topics.

Finished reading?

Mark this episode as learned to track your progress.

Mark as learned

UP NEXT

Take a Quiz →

08 Properties

Meet Sherlock Holmes, a world-renowned detective from London:

```
let sherlock = {
  surname: 'Holmes',
  address: { city: 'London' }
};
```

His friend, John Watson, recently moved in:

```
let john = {
  surname: 'Watson',
  address: sherlock.address
};
```

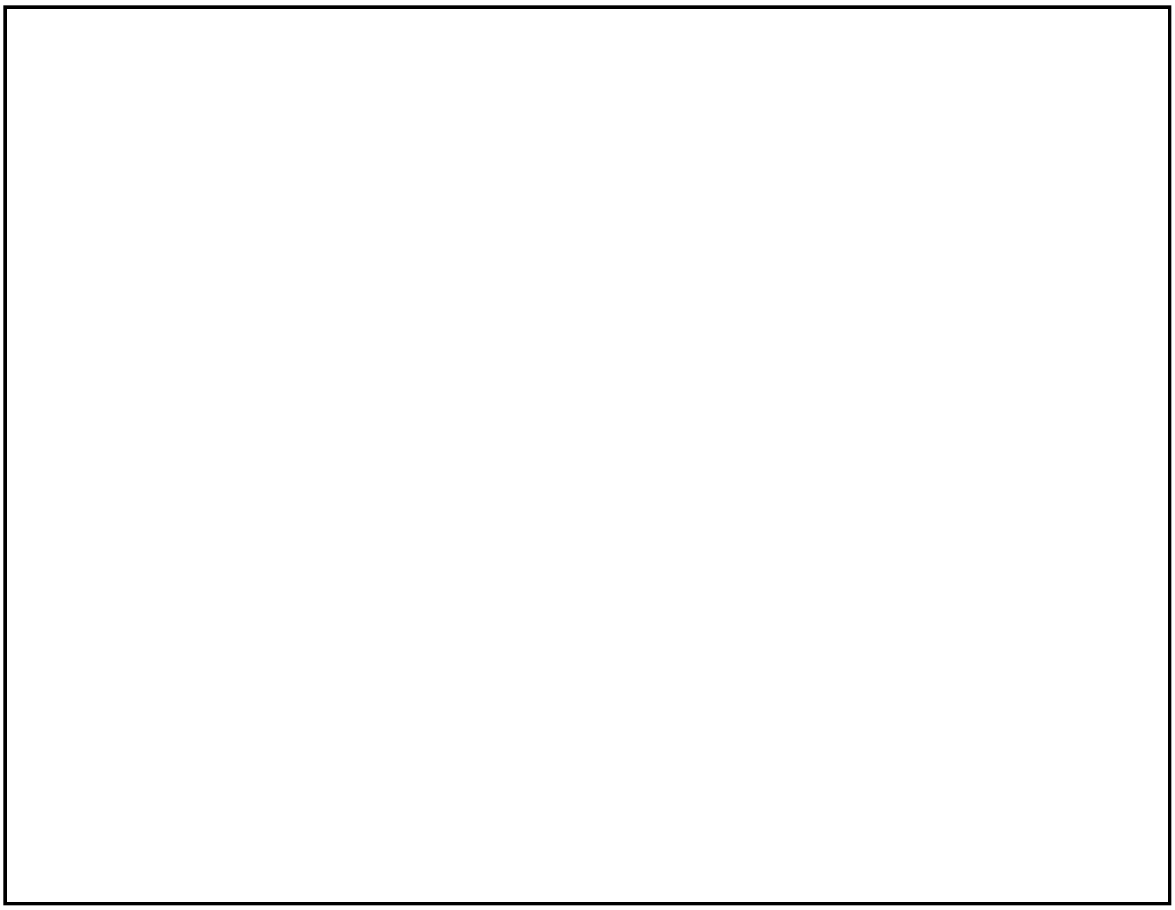
Sherlock is a brilliant detective, but a difficult flatmate. One day, John decides he's had enough—he changes his surname and moves to Malibu:

```
john.surname = 'Lennon';
john.address.city = 'Malibu';
```

Time for a small exercise. **Write down your answers to these questions:**

```
console.log(sherlock.surname); // ?
console.log(sherlock.address.city); // ?
console.log(john.surname); // ?
console.log(john.address.city); // ?
```

Before re-reading the code, I want you to approach these questions in a particular way. Use a sketchpad or get paper and a pen, and **sketch out your mental model of what happens on every line**. It's okay if you're not sure *how* to represent it. We haven't yet discussed these topics, so use your best guess.



Then, with the help of your final sketch, answer the four questions above.

Answer

Don't reveal until you have written the answers to four questions.

Reveal

This is not a typo—they are in Malibu. It's not so easy to get away from Sherlock! With an inaccurate mental model, one might conclude that `sherlock.address.city` is `"London"` —but it's not.

To see why, we need to learn how properties work in our JavaScript universe.

Properties

We've talked about objects before. For example, here is a `sherlock` variable pointing to an object value. We create a new object value by writing `{}`:

```
let sherlock = {};
```

In our universe, it might look like this:

However, objects are primarily useful to *group* related data together. For example, we might want to group different facts about Sherlock:

```
let sherlock = {
  surname: 'Holmes',
  age: 64,
};
```

Here, `sherlock` is still a variable, but `surname` and `age` are not. They are *properties*. Unlike variables, properties *belong* to a particular object.

In our JavaScript universe, both variables and properties act like "wires." However, the wires of properties *start from objects* rather than from our code:

Here, we can see that the `sherlock` **variable** points to an object we have created. That object has two **properties**. Its `surname` property points to the `"Holmes"` string value, and its `age` property points to the `64` number value.

Importantly, properties don't *contain* values—they *point* to them! Remember that our universe is full of wires. Some of them start from our code (variables), and others start from objects (properties). All wires always point to values.

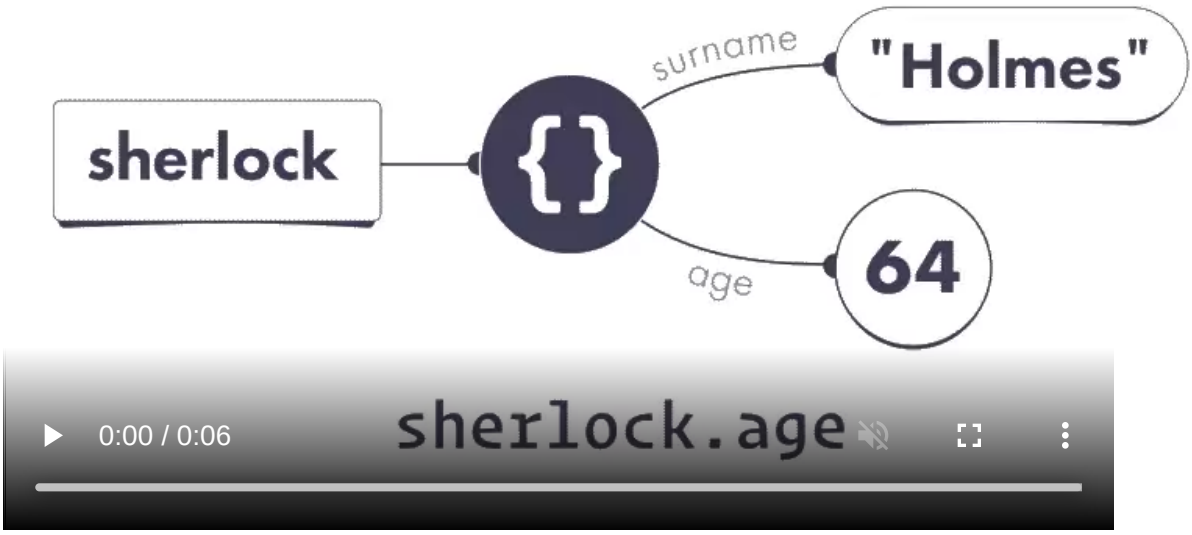
Before reading this, you might have imagined that values live "inside" objects because they appear "inside" in code. This intuition often leads to mistakes, so we will be "thinking in wires" instead. Take one more look at the code and the diagram. Make sure you're comfortable with them before you continue.

Reading a Property

We can read a property's current value by using the "dot notation":

```
console.log(sherlock.age); // 64
```

Here, `sherlock.age` is our old friend, an *expression*—a question to our JavaScript universe. To answer it, JavaScript first follows the `sherlock` wire:



It leads to an object. From that object, JavaScript follows the `age` property.

Our object's `age` property points to `64`, so `sherlock.age` results in `64`.

Property Names

One important thing to keep in mind when naming a property is that a single object can't have two properties with the same name. For example, our object can't have two properties called `age`.

Property names are also always case-sensitive! For example, `age` and `Age` would be two completely different properties from JavaScript's point of view.

If we don't know a property name ahead of time, but we have it in code as a string value, we can use the `[]` "bracket notation" to read it from an object:

```
let sherlock = { surname: 'Holmes', age: 64 };
let propertyName = prompt('What do you want to know?');
alert(sherlock[propertyName]); // Read property by its name
```

Try this code in your browser console and enter `age` when prompted.

Assigning to a Property

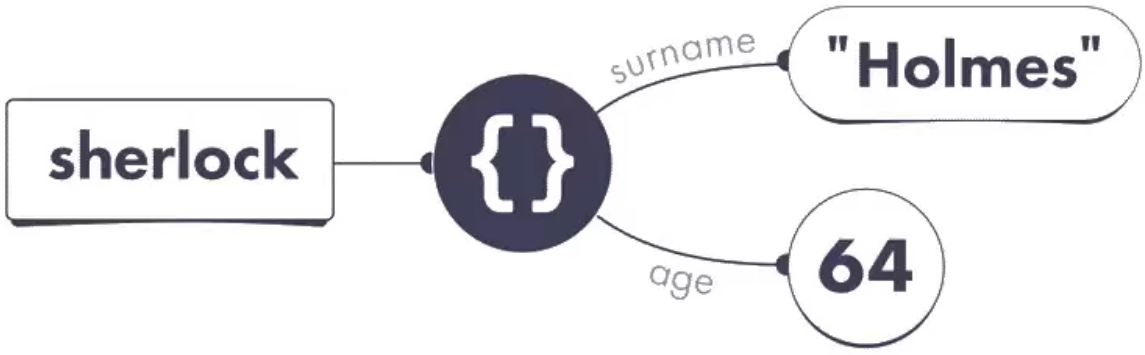
What happens when we *assign* a value to a property?

```
sherlock.age = 65;
```

Let's split this code into the left and the right side, separated by `=`.

First, we figure out which wire is on the left side: `sherlock.age`.

We follow the `sherlock` wire, and then pick the `age` property wire:

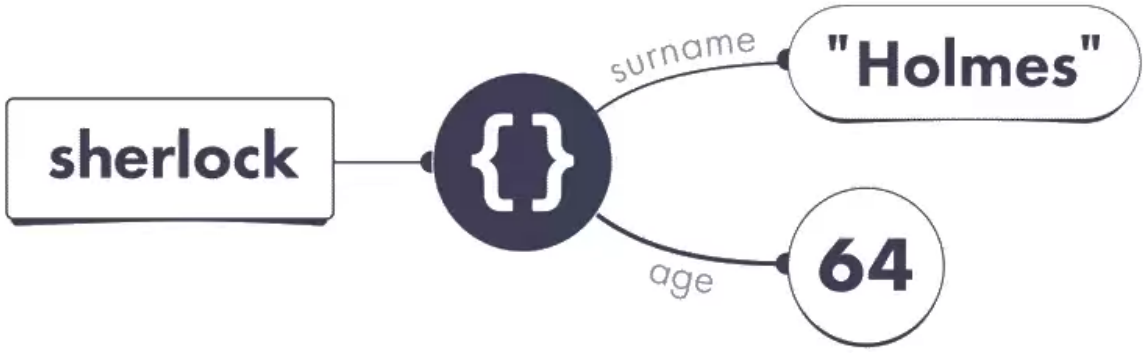


Note that we don't *follow* the `age` wire to `64`. We don't care what its current value is. On the left side of the assignment, we are looking for **the wire itself**.

Remember which wire we picked? Carry on.

Next, we figure out which value is on the right side: `65`.

Unlike the left side, the right side of an assignment always expresses a *value*. In this example, the right side's value is the number value `65`. Let's summon it:



Now we are ready to perform the assignment.

At last, we point the wire on the left side to the value on the right side:

And we're done! From now on, reading `sherlock.age` would give us `65`.

Missing Properties

You might wonder what happens if we read a property that doesn't exist:

```
let sherlock = { surname: 'Holmes', age: 64 };
console.log(sherlock.boat); // ?
```

We know that `sherlock.boat` is a property expression, but how does our JavaScript universe decide which value to "answer" us with?

JavaScript uses a set of rules that looks something like this:

1. Figure out the value of the part before the dot (`.`).
2. If that value is `null` or `undefined`, throw an error immediately.
3. Check whether a property with that name exists on our object:
   a. If **it exists**, answer with the value this property points to.
   b. If **it doesn't exist**, answer with the `undefined` value.

These rules are a bit simplified, but they already tell us a lot about how JavaScript works! For example, `sherlock` points to an object that **doesn't** have a `boat` property. So `sherlock.boat` gives us `undefined` as an answer:

```
let sherlock = { surname: 'Holmes', age: 64 };
```

```
console.log(sherlock.boat); // undefined
```

Note **this does not mean** that our object has a `boat` property pointing to `undefined`! It only has two properties, and neither of them is called `boat`:

It is tempting to think `sherlock.boat` directly corresponds to the concept of a property in our mental model, but **that's not quite correct**. It is a *question*—and so JavaScript *follows the rules* to answer it.

It looks at the object that `sherlock` points to, sees that it **doesn't** have a `boat` property, and gives us back the `undefined` value because **that's what the rules say**. There is no deeper reason for this: computers follow the rules.

Fun Fact

Fundamentally, it's because every expression needs to result in *some* value, or throw an error. Some other languages throw an error if you access a property that doesn't exist—but JavaScript is not one of them!

Scroll up and re-read the rules again. Can you apply them in practice?

```
let sherlock = { surname: 'Holmes', age: 64 };
console.log(sherlock.boat.name); // ?
```

What happens if we run this code? **Don't guess—follow the rules.**

Hint: there are two dots, so you need to follow the rules two times.

Answer

Don't reveal until you have finished writing.

Reveal

The answer is t… …ows an error:

- We need to first figure ou… `sherlock.boat`.
  - To do that, we need to figure out the value of `sherlock`.
    - The `sherlock` variable points to an object.
    - Therefore, the value of `sherlock` is that object.
  - An object is not `null` or `undefined`, so we keep going.
  - That object **does not** have a `boat` property.
  - Therefore, the value of `sherlock.boat` is `undefined`.
- We've got `undefined` on the left side of the dot (`.`).
- The rules say that `null` or `undefined` on the left side is an error.

If this still seems confusing, scroll up and mechanically follow the rules.

Recap

- Properties are wires—a bit like variables. They both point to values. Unlike variables, properties **start from objects** in our universe.
- Properties belong to particular objects. You can't have more than one property with the same name on an object.
- Generally, you can perform an assignment in three steps:

  1. Figure out which wire is on the left.
  2. Figure out which value is on the right.
  3. Point that wire to that value.

- An expression like `obj.property` is calculated in three steps:

1. Figure out which value is on the left.
2. If it's `null` or `undefined`, throw an error.
3. If that property exists, the result is the value it points to. If that property doesn't exist, the result is `undefined`.

Note that this mental model of properties is still a bit simplified. It is good enough for now, but it will need to expand as you learn more about the JavaScript universe.

If you got confused by the Sherlock Holmes example in the beginning, try walking through it again using our new mental model and focusing on properties as wires. The next module will include a detailed walkthrough in case you're still not quite sure why it works this way.

Exercises

This module also has exercises for you to practice!

Don't skip them!

Even though you're likely familiar with the concept of properties and assignment, these exercises will help you cement the mental model we're building. We need this foundation before we can get to more complex topics.

Finished reading?

Mark this episode as learned to track your progress.

Mark as learned

UP NEXT

Take a Quiz →

09 Mutation

In the previous module about *properties*, we introduced the mystery of Sherlock Holmes moving to Malibu, but we haven't explained it yet.

This time, we will walk through the code step-by-step and draw our diagrams together so you can check your mental model.

Step 1: Declaring the `sherlock` Variable

We start with this variable declaration:

```
let sherlock = {
  surname: 'Holmes',
  address: { city: 'London' }
};
```

Draw the diagram for this step now:

Answer

Don't reveal until you have drawn the diagram.

Reveal

Your diagram should look something like this:

There is a `sherlock` variable that points to an object. That object has two properties. Its `surname` property points to the `"Holmes"` string value. Its `address` property points to another object. That other object only has one property called `city`. That property points to the `"London"` string value.

Take a close look at my process for drawing this diagram:

Was your process similar?

No Nested Objects

Notice that we have not one, but *two* completely separate objects here. Two pairs of curly braces mean two objects.

Objects might appear "nested" in code, but in our universe, each object is completely separate. An object cannot be "inside" of another object! If you have been visualizing nested objects, now is the time to change your mental model.

Step 2: Declaring the `john` Variable

In this step, we declare another variable:

```
let john = {
  surname: 'Watson',
  address: sherlock.address
};
```

Edit the diagram you drew earlier to reflect these changes.

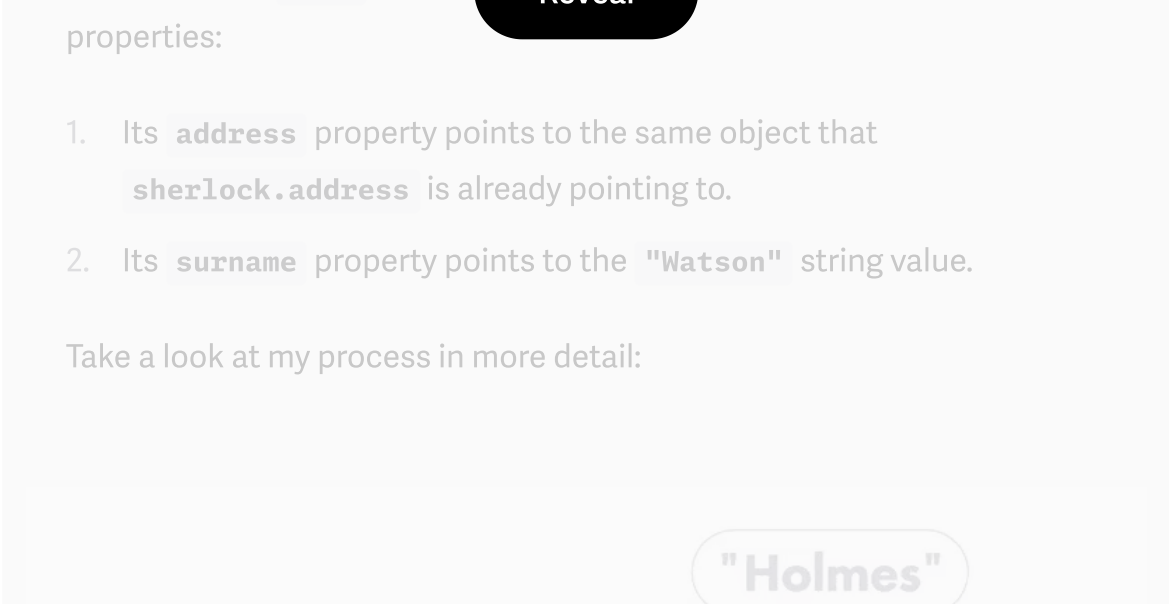
Load previous sketch

Answer

Don’t reveal until you have drawn the diagram.

Reveal

Your additional drawing should look like this:



There is a new `john` variable pointing to an object with two properties:

1. Its `address` property points to the same object that `sherlock.address` is already pointing to.
2. Its `surname` property points to the `"Watson"` string value.

Take a look at my process in more detail:

Did you do anything differently?

Properties Always Point to Values

When you see `address: sherlock.address`, it is tempting to think that John's `address` property points to Sherlock's `address` property.

This is misleading.

Remember: a property always *points to a value!* It can't point to another property or a variable. In general, all wires in our universe *point to values*.

When we see `address: sherlock.address`, we must figure out the value of `sherlock.address`, and point John's `address` property wire *to that value*.

It's the value itself (the object previously created with `{ city: 'London' }`) that matters during the assignment, not how we found it

(`sherlock.address`).

As a result, there are now two different `address` properties pointing to the same object. Can you spot them both on the diagram?

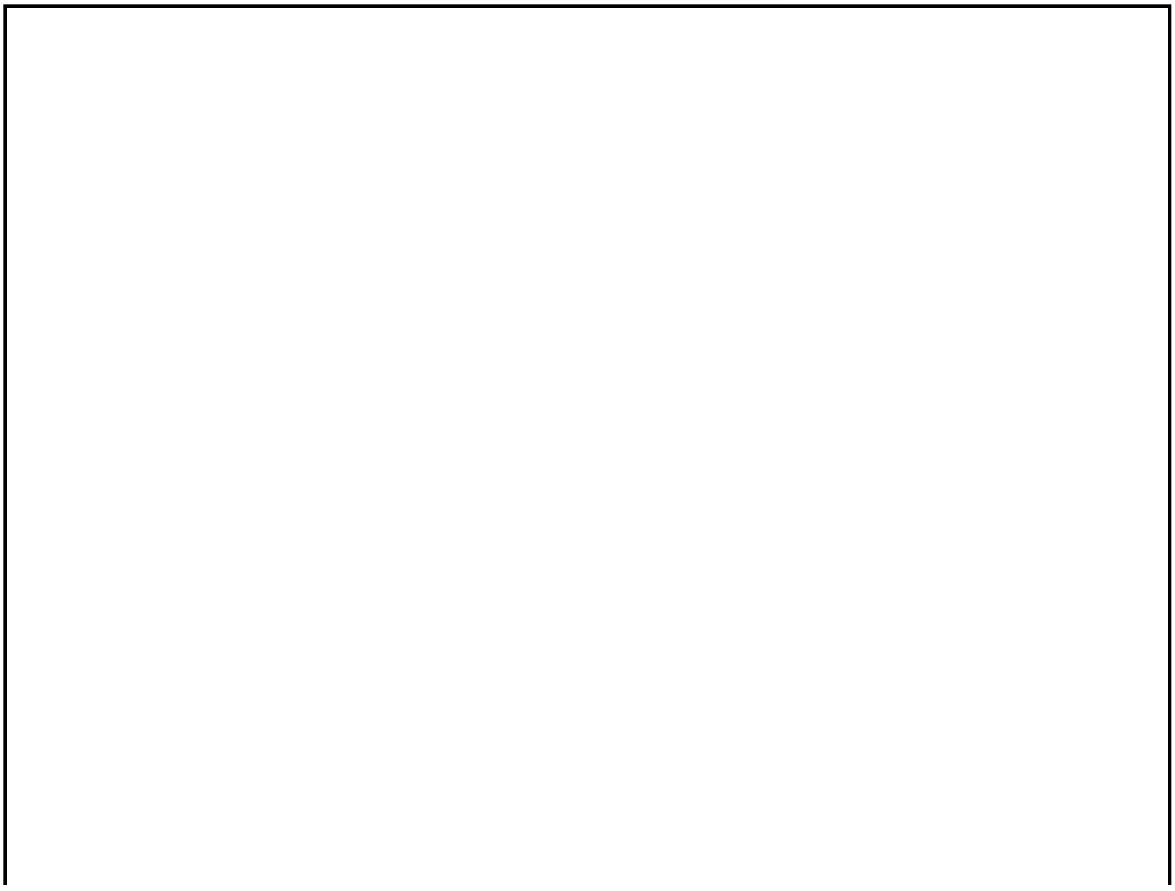
Step 3: Changing the Properties

Now—if you recall the last step of our example—John has an identity crisis, and gets sick of the London drizzle. He decides to change his name and move to Malibu. We do this by setting a few properties:

```
john.surname = 'Lennon';
john.address.city = 'Malibu';
```

How do we change the diagram to reflect it?

Load previous sketch



Answer

Don't reveal until you have drawn the diagram.

Reveal

Your diagram should look like this:

John's `surname` property now points to the `"Lennon"` string value. More interestingly, the object that both `address` properties are pointing to has changed. Its `city` property now points to the `"Malibu"` string value.

In a strange case of location hijacking, both Sherlock and John have ended up in Malibu. Follow the wires using the diagram and verify this:

Here is my process for the last series of changes:

We figure out the wire, then the value, and point the wire to that value.

The result should make sense now, but this example is confusing on a deeper level. Where is the *mistake* in it? How do we actually fix the code so that John moves to Malibu alone? To make sense of it, we need to talk about mutation.

Mutation

Mutation is a fancy way of saying "change."

For example, we could say that we *changed* an object's property, or we could say that we *mutated* that object (and its property). This is the same thing.

People like to say "mutate" because this word has a sinister undertone. It reminds you to exercise extra caution. This doesn't mean mutation is "bad"—it's just programming!—but you need to be very intentional about it.

Let's recall our original task. We wanted to give John a different surname, and move him to Malibu. Now let's look at our two mutations:

```
// Step 3: Changing the Properties
john.surname = 'Lennon';
john.address.city = 'Malibu';
```

Which objects are being mutated here?

The first line mutates the object `john` points to. This makes sense: we *mean* to change John's surname. That object represents John's data, so we mutate its `surname` property.

However, the second line does something very different. It doesn't mutate the object that `john` points to. Rather, it mutates a completely different object—

the one we can reach via `john.address`. And if we look at the diagram, we know we'll reach the same object via `sherlock.address`!

By mutating an object used elsewhere in the program, we've made a mess.

> ### Fun Fact
>
> This is why the intuition of objects being "nested" is so wrong! It makes us forget that there may be many objects pointing to the object we changed.

Possible Solution: Mutating Another Object

One way to fix this would be to avoid mutating shared data:

```
// Replace Step 3 with this code:
john.surname = 'Lennon';
john.address = { city: 'Malibu' };
```

The difference in the second line is subtle, but very important.

With `john.address.city = "Malibu"`, we are mutating the `city` property of the object that `john.address` points to. Because `john.address` and `sherlock.address` point to the same object, we unintentionally mutated shared data.

With `john.address = { city: 'Malibu' }`, we are mutating the `address` property of the object that `john` points to. In other words, *we are only mutating the object representing John's data*. This is why `sherlock.address.city` remains unchanged:

As you can see, visually similar code may produce very different results. *Always pay attention to the wires!*

Alternative Solution: No Object Mutation

There is another way we can make `john.address.city` give us `"Malibu"` while `sherlock.address.city` continues to say `"London"`:

```
// Replace Step 3 with this code:
john = {
  surname: 'Lennon',
  address: { city: 'Malibu' }
};
```

Here, we don't mutate John's object at all. Instead, we *reassign* the `john` variable to point to a "new version" of John's data. From now on, `john` points to a different object, whose `address` also points to a completely new object:

You might notice there's now an "abandoned" old version of the John object on our diagram. We don't need to worry about it. JavaScript will eventually automatically remove it from memory if there are no wires pointing to it.

Note that both of these approaches satisfy all of our requirements:

- `console.log(sherlock.surname); // "Sherlock"`
- `console.log(sherlock.address.city); // "London"`
- `console.log(john.surname); // "Lennon"`
- `console.log(john.address.city); // "Malibu"`

Compare their diagrams. Do you have a personal preference for either of these fixes? What are, in your opinion, their advantages and disadvantages?

Learn From Sherlock

Sherlock Holmes once said, "When you have eliminated the impossible, whatever remains, however improbable, must be the truth."

As your mental model becomes more complete, you will find it easier to debug problems because you will know what possible causes to look for.

For example, if you know that `sherlock.address.city` has changed after running some code, the wires from our diagram suggest three explanations:

1. Maybe the `sherlock` variable was reassigned.
2. Maybe the object we could reach via `sherlock` was mutated, and its `address` property was set to something different.
3. Maybe the object we could reach via `sherlock.address` was mutated, and its `city` property was set to something different.

Your mental model gives you a starting point from which you can investigate bugs. **This works the other way around too.** Sometimes, you can tell a piece of code is *not* the source of a problem—because the mental model proves it!

Say, if we point the `john` variable to a different object, we can be fairly sure that `sherlock.address.city` won't change. Our diagram shows that changing the `john` wire doesn't affect any wires coming from `sherlock`:

Still, keep in mind that unless you're Sherlock Holmes, you can rarely be *fully* confident in something. This approach is only as good as your mental model! Mental models will help you come up with theories, but you need to confirm them with `console.log` or a debugger.

Let vs. Const

It is worth noting you can use the `const` keyword as an alternative to `let`:

```
const shrek = { species: 'ogre' };
```

The `const` keyword lets you create read-only variables—also known as *constants*. Once we declare a constant, we can't point it to a different value:

```
shrek = fiona; // TypeError
```

But there's a crucial nuance. **We can still mutate the *object* our wire points to:**

```
shrek.species = 'human';
console.log(shrek.species); // 'human'
```

In this example, it is only the `shrek` *variable wire itself* that is read-only (`const`). It points to an object—and *that* object's properties can be mutated!

The usefulness of `const` is a hotly debated topic. Some prefer to ban `let` altogether and always use `const`. Others say that programmers should be trusted to reassign their own variables. Whatever your preference may be, remember that `const` prevents variable reassignment—not object mutation.

Is Mutation Bad?

Don't walk away thinking that mutation is "bad." That's a lazy oversimplification that obscures real understanding. If data changes over time, a mutation happens *somewhere*. The question is, *what* should be mutated, *where*, and *when*?

Mutation is "spooky action at a distance." Changing `john.address.city` led to `console.log(sherlock.address.city)` printing something else.

By the time you mutate an object, variables and properties may already be pointing to it. Your mutation affects any code "following" those wires later.

This is both a blessing and a curse. Mutation makes it easy to change some data and immediately "see" the change across the whole program. However, undisciplined mutation makes it harder to predict what the program will do.

There is a school of thought that mutation is best contained to a very narrow layer of your application. The benefit, according to this philosophy, is that your program's behavior is more predictable. The downside is that you write more code to "pass things around" and avoid mutation.

It's worth noting that mutating *new* objects that you've just created is always okay because there are no other wires pointing to them yet. In other cases, I advise you to be very intentional about what you're mutating and when. The extent to which you'll rely on mutation depends on your app's architecture.

Recap

- Objects are never "nested" in our universe—pay close attention to the wires.
- Changing an object's property is also called *mutating* that object.
- If you mutate an object, your code will "see" that change via any wires pointing to that object. Sometimes, this may be what you want. However, mutating accidentally shared data may cause bugs.
- You can declare a variable with `const` instead of `let`. That allows you to enforce that this variable always points to the same value. But remember that `const` does *not* prevent object mutation!
- Mutating the objects you've just created in code is safe. Broadly, how much you'll use mutation depends on your app's architecture.

Exercises

This module also has exercises for you to practice!

Don't skip them!

Even though you're likely familiar with the concept of mutation, these exercises will help you cement the mental model we're building. We need this foundation before we can get to more complex topics.

Finished reading?

Mark this episode as learned to track your progress.

Mark as learned

UP NEXT

Take a Quiz →

10 Prototypes

In previous modules, we've covered object properties and mutation, but we're not quite done—we still need to talk about prototypes!

Here is a small riddle to check our mental model:

```
let pizza = {};
console.log(pizza.taste); // "pineapple"
```

Ask yourself: is this possible?

We have just created an empty object with `{}`. We definitely didn't set any properties on it before logging, so it seems like `pizza.taste` can't point to `"pineapple"`. We would expect `pizza.taste` to give us `undefined` instead —we usually get `undefined` when a property doesn't exist, right?

And yet, it *is* possible that `pizza.taste` is `"pineapple"`! This may be a contrived example, but it shows that our mental model is incomplete.

In this module, we'll introduce *prototypes*. Prototypes explain what happens in this puzzle and—more importantly—are at the heart of several other fundamental JavaScript features. Occasionally people neglect to learn about prototypes because they seem too unusual, but the core idea is remarkably simple.

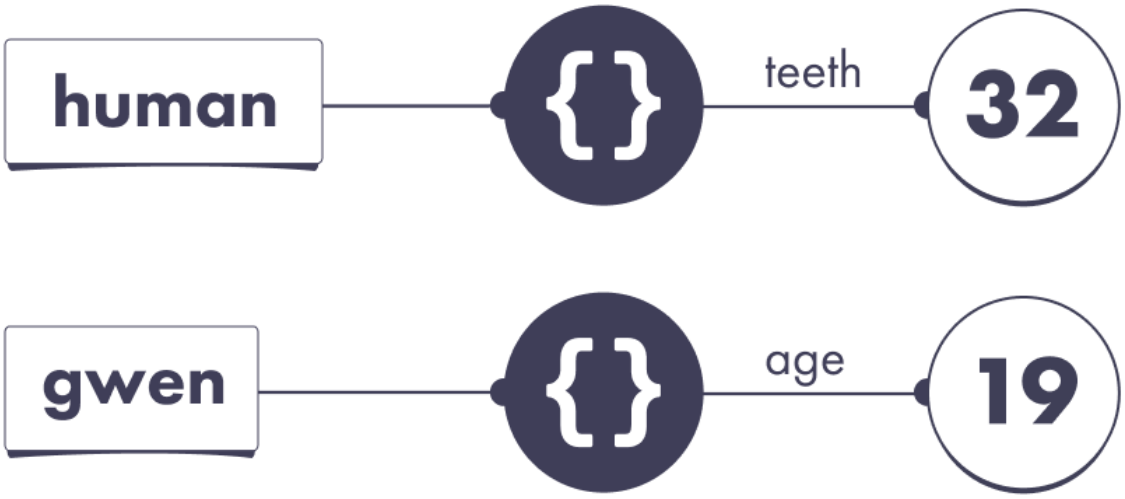
Prototypes

Here's a couple of variables pointing to a couple of objects:

```
let human = {
  teeth: 32
};

let gwen = {
  age: 19
};
```

We can represent them visually in a familiar way:



In this example, `gwen` points to an object without a `teeth` property. According to the rules we've learned, logging this property would give us `undefined`:

```
console.log(gwen.teeth); // undefined
```

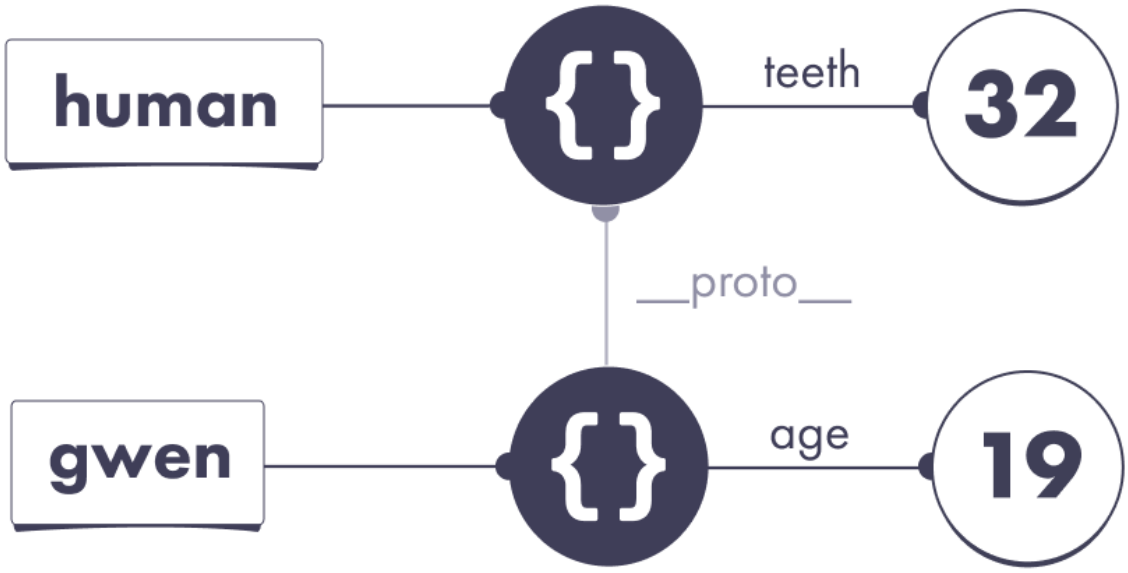
But the story doesn't have to end here. JavaScript's default behavior returns `undefined`, but we can instruct it to *continue searching for our missing property on another object*. We can do it with one line of code:

```
let human = {
  teeth: 32
};

let gwen = {
  // We added this line:
  __proto__: human,
  age: 19
};
```

What is that mysterious `__proto__` property?

It represents the JavaScript concept of a *prototype*. Any JavaScript object may choose another object as a prototype. We will discuss what that means in practice but for now, let's think of it as a special `__proto__` wire:



Take a moment to verify the diagram matches the code. We drew it just like we did in the past. The only new thing is the mysterious `__proto__` wire.

By specifying `__proto__` (also known as our object's *prototype*), we instruct JavaScript to continue looking for missing properties on that object instead.

Prototypes in Action

When we went looking for `gwen.teeth`, we got `undefined` because the `teeth` property doesn't exist on the object that `gwen` points to.

But thanks to `__proto__: human`, the answer is different:

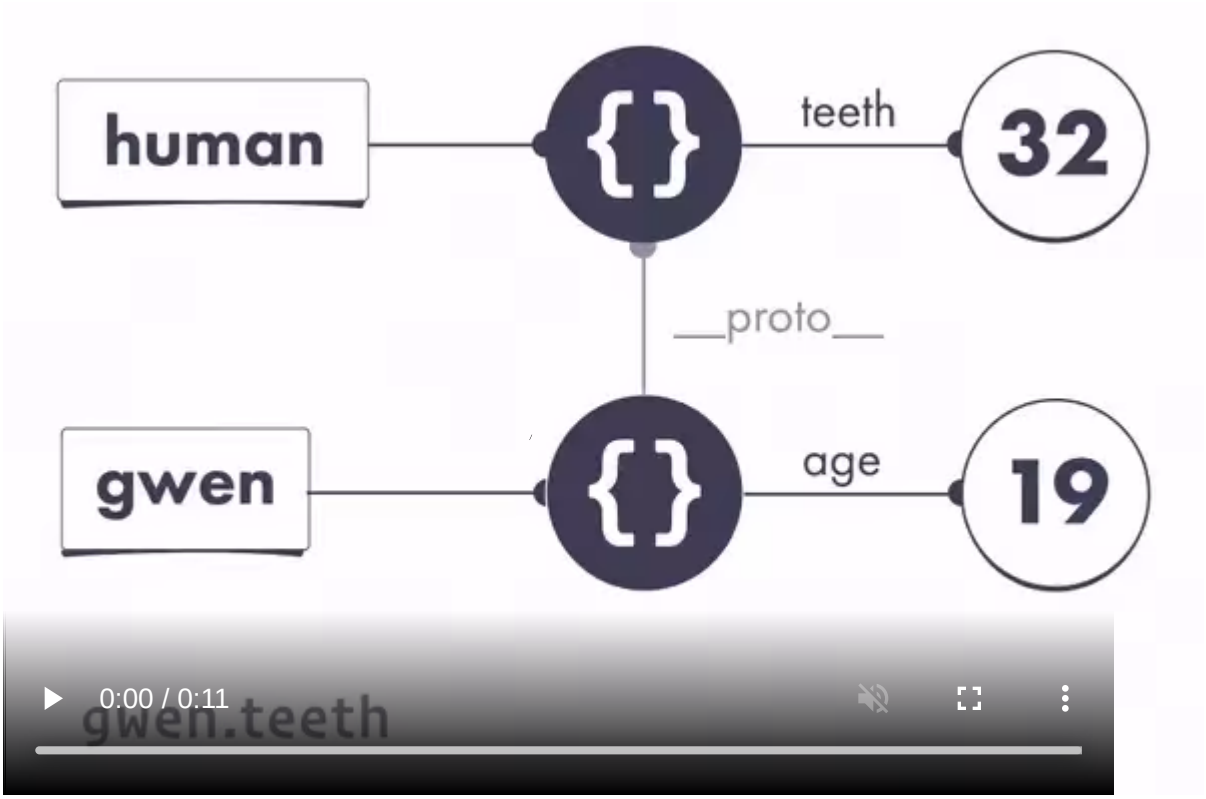
```
let human = {
  teeth: 32
};

let gwen = {
```

```
  // "Look for other properties here"
  __proto__: human,
  age: 19
};

console.log(gwen.teeth); // 32
```

Now the sequence of steps looks like this:



1. Follow the `gwen` wire. It leads to an object.
2. Does this object have a `teeth` property?
    - No.
    - **But it has a prototype.** Let's check it out.
3. Does *that* object have a `teeth` property?
    - Yes, it points to `32`.
    - Therefore, the result of `gwen.teeth` is `32`.

This is similar to saying, "I don't know, but Alice might know." With `__proto__`, you instruct JavaScript to "ask another object."

To check your understanding so far, write down your answers:

```
let human = {
  teeth: 32
};

let gwen = {
  __proto__: human,
  age: 19
};

console.log(human.age); // ?
console.log(gwen.age); // ?

console.log(human.teeth); // ?
console.log(gwen.teeth); // ?

console.log(human.tail); // ?
console.log(gwen.tail); // ?
```

Answer

Don't reveal until you have written the answers to six questions

Reveal

Now [illegible]

The `human` variable points t[illegible] doesn't have an `age` property, so `human.age` is `undefined`. The `gwen` variable points to an object that *does* have an `age` property. That wire points to `19`, so the value of `gwen.age` is `19`:

The `human` variable points to an object that has a `teeth` property, so the value of `human.teeth` is `32`. The `gwen` variable points to an object that doesn't have a `teeth` property. However, that object has a prototype, which *does* have a `teeth` property. This is why the value of `gwen.teeth` is also `32`.

Neither of our objects has a `tail` property, so we get `undefined` for both:

Note how although the value of `gwen.teeth` is `32`, it doesn't mean `gwen` has a `teeth` property! Indeed, in this example, the object that `gwen` points to *does not* have a `teeth` property. But its prototype object—the same one `human` points to—does.

This serves to remind us that `gwen.teeth` is an *expression*—a question to our JavaScript universe—and JavaScript will follow a sequence of steps to answer it. Now we know these steps involve looking at the prototype.

The Prototype Chain

A prototype isn't a special "thing" in JavaScript. A prototype is more like a *relationship*. An object may point to another object as its prototype.

This naturally leads to a question: but what if my object's prototype has its own prototype? And that prototype has *its* own prototype? Would that work?

The answer is yes—this is *exactly* how it works!

```
let mammal = {
  brainy: true,
};

let human = {
  __proto__: mammal,
  teeth: 32
};

let gwen = {
  __proto__: human,
  age: 19
};

console.log(gwen.brainy); // true
```

We can see that JavaScript will search for the property on our object, then on its prototype, then on *that* object's prototype, and so on. We would only get `undefined` if we ran out of prototypes and still hadn't found our property.

This is similar to saying, "I don't know, but Alice might know." Then Alice might say, "Actually, I don't know either—ask Bob." Eventually, you will either arrive at the answer or run out of people to ask!

This sequence of objects to "visit" is known as our object's *prototype chain*. (However, unlike a chain you might wear, prototype chains can't be circular!)

Shadowing

Consider this slightly modified example:

```
let human = {
  teeth: 32
};

let gwen = {
  __proto__: human,
  // This object has its own teeth property:
  teeth: 31
};
```

Both objects define a property called `teeth`, so the results are different:

```
console.log(human.teeth); // 32
console.log(gwen.teeth); // 31
```

Note that `gwen.teeth` is `31`. If `gwen` didn't have its own `teeth` property, we would look at the prototype. But because the object that `gwen` points to has its *own* `teeth` property, we don't need to keep searching for the answer.

In other words, once we find our property, **we stop the search**.

If you ever want to check if an object has its *own* property wire with a certain name, you can call a built-in function called `hasOwnProperty`. It returns `true` for "own" properties, and does not look at the prototypes. In our last example, both objects have their own `teeth` wires, so it is `true` for both:

```
console.log(human.hasOwnProperty('teeth')); // true
console.log(gwen.hasOwnProperty('teeth')); // true
```

Assignment

Consider this example:

```
let human = {
  teeth: 32
};

let gwen = {
  __proto__: human,
  // Note: no own teeth property
};

gwen.teeth = 31;

console.log(human.teeth); // ?
console.log(gwen.teeth); // ?
```

Before the assignment, both expressions result in `32`:

Then we need to execute this assignment:

```
gwen.teeth = 31;
```

Now the question is which wire does `gwen.teeth` correspond to? The answer is that, generally saying, assignments happen on the *object itself*.

So `gwen.teeth = 31` *creates a new own property* called `teeth` on the object that `gwen` points to. It doesn't have any effect on the prototype:

As a result, `human.teeth` is still `32`, but `gwen.teeth` is now `31`:

```
console.log(human.teeth); // 32
console.log(gwen.teeth); // 31
```

We can summarize this behavior with a simple rule of thumb.

When we *read* a property that doesn't exist on our object, we'll keep looking for it on the prototype chain. If we don't find it, we get `undefined`.

But when we *write* a property that doesn't exist on our object, that *creates* the property on our object. Generally speaking, prototypes will *not* play a role.

The Object Prototype

This object doesn't have a prototype, right?

```
let obj = {};
```

Try running this in your browser's console:

```
let obj = {};
console.log(obj.__proto__); // Play with it!
```

Surprisingly, `obj.__proto__` is not `null` or `undefined`! Instead, you'll see a curious object with a bunch of properties, including `hasOwnProperty`.

We're going to call that special object the Object Prototype:

At first, this might be a bit mind-blowing. Let that sink in. All this time, we were thinking `{}` created an "empty" object, but it's not so empty after all! It has a hidden `__proto__` wire that points to the Object Prototype by default.

This explains why JavaScript objects seem to have "built-in" properties:

```
let human = {
  teeth: 32
};
console.log(human.hasOwnProperty); // (function)
console.log(human.toString); // // (function)
```

These "built-in" properties are nothing more than normal properties on the Object Prototype. Because our object's prototype is the Object Prototype, we can access them.

An Object With No Prototype

We've just learned that all objects created with the `{}` syntax have the special `__proto__` wire pointing to a default Object Prototype. But we also know that we can customize the `__proto__`. You might wonder: can we set it to `null`?

```
let weirdo = {
  __proto__: null
};
```

The answer is yes—this will produce an object that truly doesn't have a prototype at all. As a result, it doesn't even have built-in object methods:

```
console.log(weirdo.hasOwnProperty); // undefined
console.log(weirdo.toString); // undefined
```

You probably won't want to create objects like this, however the Object Prototype is exactly that—an object with no prototype.

Polluting the Prototype

Now we know that all JavaScript objects get the same prototype by default. Let's briefly revisit our example from the module about mutation:

This picture gives us an interesting insight. If JavaScript searches for missing properties on the prototype, and most objects share the same prototype, can we make new properties "appear" on all objects by mutating that prototype?

Let's add these two lines of code:

```
let obj = {};
obj.__proto__.smell = 'banana';
```

We mutated the Object Prototype by adding a `smell` property to it. As a result, both detectives now appear to be using a banana-flavored perfume:

```
console.log(sherlock.smell); // "banana"
console.log(watson.smell); // "banana"
```

Mutating a shared prototype like we just did is called *prototype pollution*.

In the past, prototype pollution was a popular way to extend JavaScript with custom features. However, over the years, the web community realized that it is fragile and makes it hard to add new language features, so we prefer to avoid it.

Now you can solve the pineapple pizza puzzle from the beginning of this module! Check your solution in your console.

Fun Fact

`__proto__` vs. `prototype`

You might be wondering: what in the world is the `prototype` property? You might have seen it in MDN documentation.

The story around this is confusing. Before JavaScript added classes, it was common to write them as functions that produce objects, for example:

```
function Donut() {
  return { shape: 'round' };
}

let donut = Donut();
```

You'd want all donuts to share a prototype with some shared methods. However, manually adding `__proto__` to every object looks gross:

```
function Donut() {
  return { shape: 'round' };
}

let donutProto = {
  eat() {
    console.log('Nom nom nom');
  }
};

let donut1 = Donut();
donut1.__proto__ = donutProto;
let donut2 = Donut();
donut2.__proto__ = donutProto;

donut1.eat();
donut2.eat();
```

As a shortcut, adding `.prototype` *on the function itself* and adding `new` before your function calls would automatically attach the `__proto__`:

```
function Donut() {
  return { shape: 'round' };
}
Donut.prototype = {
  eat() {
    console.log('Nom nom nom');
  }
};

let donut1 = new Donut(); // __proto__: Donut.prototype
let donut2 = new Donut(); // __proto__: Donut.prototype

donut1.eat();
donut2.eat();
```

Now this pattern has mostly fallen into obscurity, but you can still see `prototype` property on the built-in functions and even on classes. To conclude, a function's `prototype` specifies the `__proto__` of the objects created by calling that function with a `new` keyword.

Why Does This Matter?

In practice, you probably won't use prototypes in your code directly. (In fact, even using the `__proto__` syntax is discouraged.)

Prototypes are unusual—most frameworks never embraced them as a paradigm. Still, you will notice prototypes hiding "beneath the surface" of other JavaScript features. For example, people often use prototypes to create a traditional "class inheritance" model that's popular in other programming languages.

This became so common that JavaScript added a class syntax as a convention that "hides" prototypes out of sight. To see it in action, look at this snippet of a JavaScript class rewritten with `__proto__` for a comparison.

Personally, I don't use a lot of classes in my daily coding, and I rarely deal with prototypes directly either. However, it helps to know how those features build on each other, and it's important to know what happens when I read or set a property on an object.

Recap

- When reading `obj.something`, if `obj` doesn't have a `something` property, JavaScript will look for `obj.__proto__.something`. Then it will look for `obj.__proto__.__proto__.something`, and so on, until it either finds our property or reaches the end of the prototype chain.
- When writing to `obj.something`, JavaScript will usually write to the object directly instead of traversing the prototype chain.
- We can use `obj.hasOwnProperty('something')` to determine whether our object has its *own* property called `something`.
- We can "pollute" a prototype shared by many objects by mutating it. We can even do this to the Object Prototype—the default prototype for `{}` objects! (But we shouldn't, unless we're pranking our colleagues.)
- You probably won't use prototypes much directly in practice. However, they are fundamental to JavaScript objects, so it is handy to understand their underlying mechanics. Some advanced JavaScript features, including classes, can be expressed in terms of prototypes.

Exercises

This module also has exercises for you to practice!

Don't skip them!

Even though you're likely familiar with the concept of prototypes, these exercises will help you cement the mental model we're building. We need this foundation before we can get to more complex topics.

Finished reading?

Mark this episode as learned to track your progress.

Mark as learned

UP NEXT

Take a Quiz →