

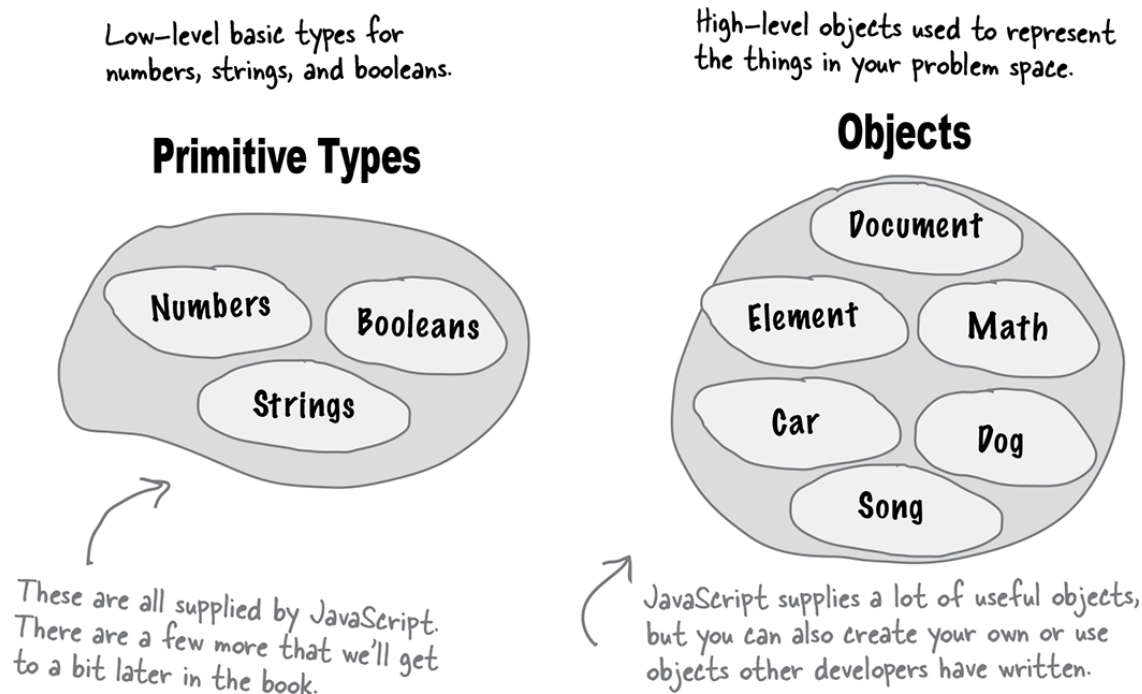
Chapter 7. Types, Equality, Conversion, and all that jazz: *Serious types*



It's time to get serious about our types. One of the great things about JavaScript is you can get a long way without knowing a lot of details of the language. But to truly **master the language**, get that promotion, and get on to the things you really want to do in life, you have to rock at **types**. Remember what we said about JavaScript back in [Chapter 1](#)? That it didn't have the luxury of a silver-spoon, academic, peer-reviewed language definition? Well, that's true, but the lack of an academic life didn't stop Steve Jobs and Bill Gates, and it didn't stop JavaScript either. It does mean that JavaScript doesn't have the...well, the most thought-out type system, and we'll find a few **idiosyncrasies** along the way. But don't worry, in this chapter we're going to nail all that down, and soon you'll be able to avoid all those embarrassing moments with types.

The truth is out there...

Now that you've had some experience working with JavaScript types—there's your primitives with numbers, strings, and booleans, and there's all the objects, some supplied by JavaScript (like the `Math` object), some supplied by the browser (like the `document` object), and some you've written yourself—aren't you just basking in the glow of JavaScript's simple, powerful, and consistent type system?



After all, what else would you expect from the official language of Webville? In fact, if you were a mere scripiter, you might think about sitting back, sipping on that Webville Martini, and taking a much-needed break...

But you're not a mere scripiter, and something is amiss. You have a sinking feeling that behind Webville's picket fences something bizarre is at work. You've heard the reports of sightings of strings that are acting like objects, you've read in the blogs about a (probably radioactive) null type, you've heard the rumors that the JavaScript interpreter as of late has been doing some weird type conversions...What does it all mean? We don't know, but the truth is out there, and we're going to uncover it in this

chapter—and when we do, we might just turn what you think of as true and false upside down.





A bunch of JavaScript values and party crashers, in full costume, are playing a party game: “Who am I?” They give you a clue, and you try to guess who they are, based on what they say. Assume they always tell the truth about themselves. Draw an arrow from each sentence to the name of one attendee. We’ve already guessed one of them for you. Check your answers at the end of the chapter before you go on.

If you find this exercise difficult, it’s okay to cheat and look at the answers.

I get returned from a function when there is no return statement.

zero

I'm the value of a variable when it hasn't been assigned a value.

empty object

null

I'm the value of an array item that doesn't exist in a sparse array.

undefined

NaN

I'm the value of a property that doesn't exist.

infinity

area 51

I'm the value of a property that's been deleted.

...---...

{ }

[]

→ Solution in ["Who am I?"](#)

Watch out, you might bump into undefined when you aren't expecting it...

Yikes!

As you can see, whenever things get shaky—you need a variable that's not been initialized yet, you want a property that doesn't exist (or has been deleted), you go after an array item that isn't there—you're going to encounter `undefined`.

But what the heck is it? It's not really that complicated. Think of `undefined` as the value assigned to things that don't yet have a value (in other words, they haven't been initialized).

So what good is it? Well, `undefined` gives you a way to test to see if a variable (or property, or array item) has been given a value. Let's look at a couple of examples, starting with an unassigned variable:

```
let x;  
  
if (x == undefined) {  
    // x isn't defined! just deal with it!  
}
```

You can check to see if a variable like x is undefined. Just compare it to the value undefined.

Note that we're using the value undefined here, not to be confused with the string "undefined".

Or how about an object property?

```
let customer = {  
    name: "Jenny"  
};  
  
if (customer.phoneNumber == undefined) {  
    // set the customer's phone number  
}
```

You can check to see if a property is undefined, again by comparing it to the value undefined.

Q: When do I need to check if a variable (or property or array item) is undefined?

A: Your code design will dictate this. If you've written code so that a property or variable may not have a value when a certain block of code is executed, then checking for undefined gives you a way to handle that situation rather than computing with undefined values.

Q: If undefined is a value, does it have a type?

A: Yes, it does. The type of undefined is undefined. Why? Well, our logic (work with us here) is this: it isn't an object, or a number or a string or a boolean, or really anything that is defined. So why not make the type undefined, too? This is one of those weird twilight zones of JavaScript you just have to accept.



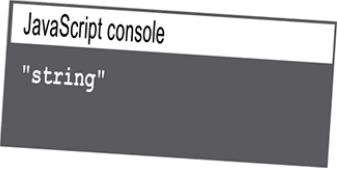
In the laboratory, we like to take things apart, look under the hood, poke and prod, hook up our diagnostic tools, and check out what is really going on. Today, we're investigating JavaScript's type system, and we've found a little diagnostic tool called **typeof** to examine variables. Put your lab coat and safety goggles on, and come on in and join us.

The **typeof** operator is built into JavaScript. You can use it to probe the type of its operand (the thing you use it to operate on). Here's an example:

```
let subject = "Just a string";  
let probe = typeof subject;  
console.log(probe);
```

The **typeof** operator takes an operand and evaluates to the type of the operand.

The type here is "string". Note that **typeof** uses strings to represent types, like "string", "boolean", "number", "object", "undefined", and so on.

A screenshot of a JavaScript console window. The title bar says "JavaScript console". The output area shows the string "string" in quotes.

Now it's your turn. Collect the data for the following experiments:


```
let test1 = "abcdef";  
let test2 = 123;  
let test3 = true;  
let test4 = {};  
let test5 = [];  
let test6;  
let test7 = {"abcdef": 123};  
let test8 = ["abcdef", 123];  
function test9() {return "abcdef"};  
  
console.log(typeof test1);  
console.log(typeof test2);  
console.log(typeof test3);  
console.log(typeof test4);  
console.log(typeof test5);  
console.log(typeof test6);  
console.log(typeof test7);  
console.log(typeof test8);  
console.log(typeof test9);
```

← Here's the test data,
and the tests.
↓

JavaScript console

↑ Put your results here. Are
there any surprises?

→ Solution in **"In the Laboratory Solution"**



Ah yes, this causes a lot of confusion. There are many languages that have the concept of a value that means “no object.” And it’s not a bad idea—take the `document.getElementById` method. It’s supposed to return an

object, right? So, what happens if it can't? Then we want to return something that says "I would have been an object if there was one, but we don't have one." And that's what `null` is.

You can also set a variable to `null` directly:

```
let killerObjectSomeday = null;
```

What does it mean to assign the value `null` to a variable? How about "We intend to assign an object to this variable at some point, but we haven't yet."


Now, if you're scratching your head and saying "Hmm, why didn't they just use `undefined` for that?" then you're in good company. The answer comes from the very beginnings of JavaScript. The idea was to have one value for variables that haven't been initialized to anything yet, and another that means the lack of an object. It isn't pretty, and it's a little redundant, but it is what it is at this point. Just remember the intent of each (`undefined` and `null`), and know that it is most common to use `null` in places where an object should be but one can't be created or found, and it is most common to find `undefined` when you have a variable that hasn't been initialized, or an object with a missing property, or an array with a missing value.

BACK IN THE LABORATORY

Oops, we forgot `null` in our test data. Here's the missing test case:

```
let test10 = null;
```

```
console.log(typeof test10);
```

 Put your results here. 

JavaScript console

➡ Solution in ["Back in the Laboratory Solution"](#)

How to use null

There are many functions and methods out there in the world that return objects, and you'll often want to make sure what you're getting back is a full-fledged object, and not `null`, just in case the function wasn't able to find one or make one to return to you. You've already seen examples from the DOM where a test is needed:

```
let header = document.getElementById("header");  
if (header == null) {  
    // okay, something is seriously wrong if we have no header  
}
```

Let's look for the all-important header element.

Uh-oh, it doesn't exist. Abandon ship!

Keep in mind that getting `null` doesn't necessarily mean something is wrong. It may just mean something doesn't exist yet and needs to be created, or something doesn't exist and you can skip it. Let's say users have the ability to open or close a weather widget on your site. If a user has it open there's a `<div>` with the id of "weatherDiv", and if not, there isn't. All of a sudden `null` becomes quite useful:

```
let weather = document.getElementById("weatherDiv");  
if (weather != null) {  
    // create content for the weather div  
}
```

Let's see if the element with id "weatherDiv" exists.

If the result of `getElementById` isn't null, then there is such an element in the page. Let's create a nice weather widget for it (presumably getting the weather for the local area).

We can use null to check
to see if an object exists
yet or not.

Blaine, Missouri

It is always 67
degrees with a 40%
chance of rain.



Remember, null is intended to represent an object that isn't there.

The Number that isn't a Number



It's easy to write JavaScript statements that result in numeric values that are not well defined.

Here are a few examples:

```
let a = 0/0;
```

↑ In mathematics this has no direct answer, so we can't expect JavaScript to know the answer either!

```
let b = "food" * 1000;
```

↑ We don't know what this evaluates to, but it is certainly not a number!

```
let c = Math.sqrt(-9);
```

↑ If you remember high school math, the square root of a negative number is an imaginary number, which you can't represent in JavaScript.

Believe it or not, there are numeric values that are impossible to represent in JavaScript! JavaScript can't express these values, so it has a stand-in value that it uses:

NaN

JavaScript uses the value NaN, more commonly known as "Not a Number," to represent numeric results that, well, can't be represented. Take $0 / 0$ for instance. $0 / 0$ evaluates to something that just can't be represented in a computer, so it is represented by NaN in JavaScript.



NaN MAY BE THE WEIRDEST VALUE IN THE WORLD. Not only does it represent all the numeric values that can't be represented, it is the only value in JavaScript that isn't equal to itself!

You heard that right. If you compare NaN to NaN, they aren't equal!

NaN != NaN

Dealing with NaN

Now, you might think that dealing with NaN is a rare event, but if you're working with any kind of code that uses numbers, you'll be surprised how often it shows up. The most common thing you'll need to do is test for NaN, and given everything you've learned about JavaScript, how to do this might seem obvious:

```
if (myNum == NaN) {  
    myNum = 0;  
}
```

You'd think this would work,
but it doesn't.

WRONG!

Any sensible person would assume that's how you test to see if a variable holds a `NaN` value, but it doesn't work. Why? Well, `NaN` isn't equal to anything—not even itself—so, any kind of test for equality with `NaN` is off the table. Instead, you need to use a special function, `isNaN`. Like this:

```
if (isNaN(myNum)) {  
    myNum = 0;  
}
```

Use the `isNaN` function, which
returns true if the value
passed to it is not a number.

RIGHT!

It gets even weirder...

Let's think through this a bit more. If `NaN` stands for “Not a Number,” what is it? Wouldn't it be easier if it were named for what it is rather than what it isn't? What do you think it is? We can check its type for a hint:

```
let test11 = 0 / 0;  
console.log(typeof test11);
```

Here's what we got.

JavaScript console

number



If your mind isn't blown,
you should probably just
use this book for some
good kindling.

What on earth? NaN is of type number? How can something that's not a number have the type number? Okay, deep breath. Think of NaN as just a poorly named value. Someone should have called it something more like “number that can't be represented” (okay, we agree the acronym isn't quite as nice) instead of “Not a Number.” If you think about it like that, then you can think of NaN as being a value that is a number but can't be represented (at least, not by a computer).

Go ahead and add this one to your JavaScript twilight zone list.

Q: If I pass isNaN a string, which isn't a number, will it return true?

A: It sure will, just as you'd expect. You can expect a variable holding the value NaN, or any other value that isn't an actual number, to result in isNaN returning true (and false otherwise). There are a few caveats to this that you'll see when we talk about type conversion.

Q: But why isn't NaN equal to itself?

A: If you're deeply interested in this topic, you'll want to seek out the IEEE floating-point specification. However, the layman's insight into this is that NaN represents an unrepresentable numeric value, but not all unrepresentable numbers are equal. For instance, take $\sqrt{-1}$ and $\sqrt{-2}$. They are definitely not the same, but they both produce NaN.

Q: When we divide 0 / 0 we get NaN, but I tried dividing 10 / 0 and got Infinity. Is that different from NaN?

A: Good find. The Infinity (or -Infinity) value in JavaScript represents all numbers (to get a little technical) that exceed the upper limit on computer floating-point numbers, which is $1.7976931348623157 \times 10^{308}$ (or $-1.7976931348623157 \times 10^{308}$ for -Infinity). The type of Infinity is number, and you can test for it if you suspect one of your values is getting a little large:

```
if (tamale == Infinity) {  
    alert("That's a big tamale!");  
}
```

Q: You did blow my mind with that “NaN is a number” thing. Any other mind-blowing details?

A: Funny you should ask. How about Infinity minus Infinity equals...wait for it...NaN. We'll refer you to a good mathematician to understand that one.

Q: Just to cover every detail, did you say what the type of null is?

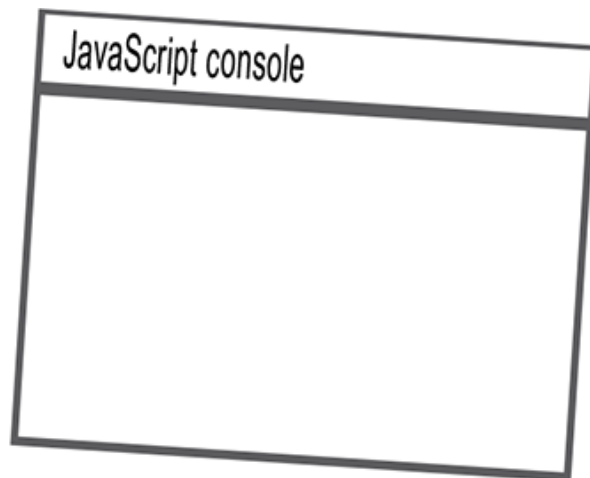
A: No, but a quick way to find out is by using the `typeof` operator on `null`. If you do that, you'll get back the result `"object"`. This makes sense, from the perspective that `null` is used to represent an object that isn't there. However, this point has been heavily debated, and the most recent spec defines the type of `null` as `null`. You'll find this an area where your browser's JavaScript implementation may not match the spec, but, in practice you'll rarely need to use the type of `null` in code.



We've been looking at some rather, um, interesting values so far in this chapter. Now, let's take a look at some interesting behavior. Try adding the code below to the `<script>` element in a basic web page and see what you get in the console when you load up the page. You probably won't get why yet, but see if you can take a guess about what might be going on.

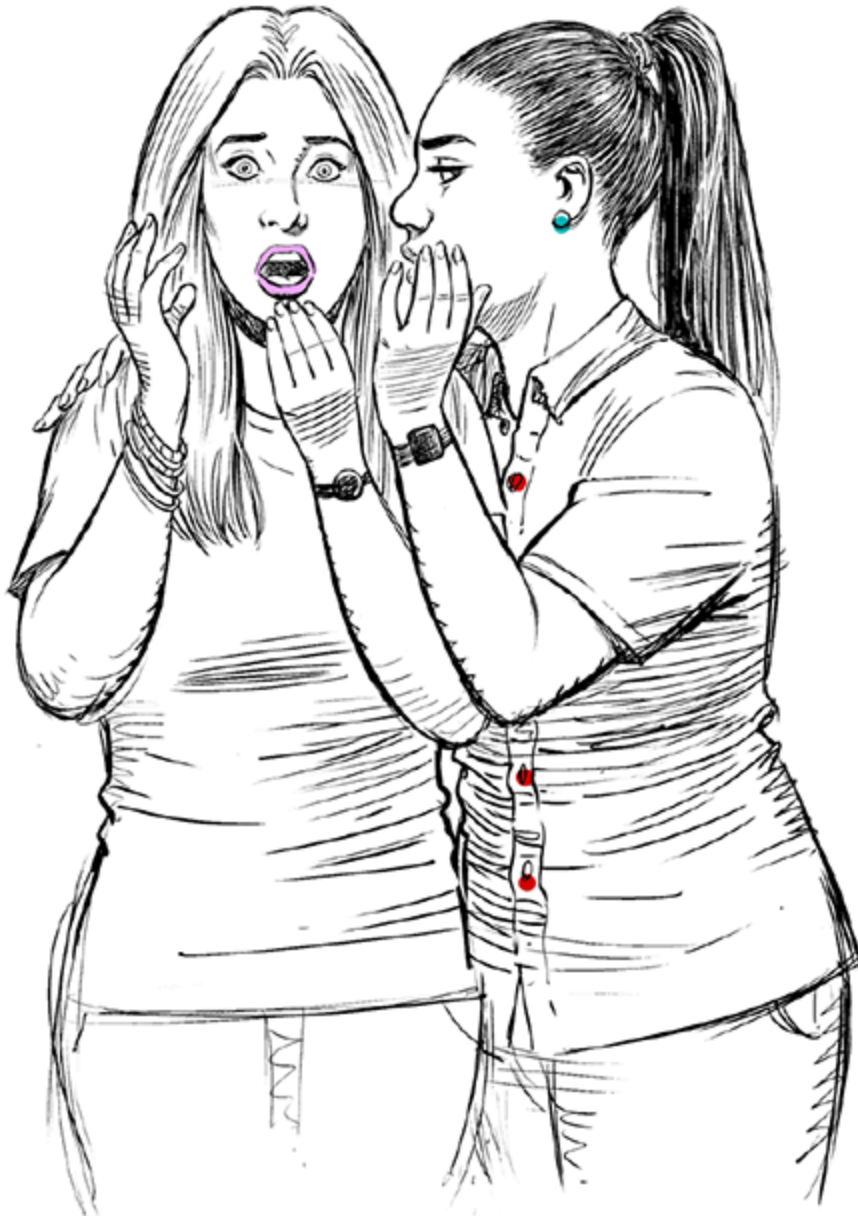
```
if (99 == "99") {  
    console.log("A number equals a string!");  
} else {  
    console.log("No way a number equals a string");  
}
```

Write what you get here.



➔ Solution in [“Exercise Solution”](#)

We have a confession to make



There is an aspect of JavaScript we've deliberately been holding back on. We could have told you up front, but it wouldn't have made as much sense as it will now.

It's not so much that we've been pulling the wool over your eyes, it's that there is more to the story than we've been telling you. And what is this topic? Here, let's take a look:

At some point a variable gets set, in this case to the number 99.

```
let testMe = 99;
```

And later it gets compared with a number in a conditional test.

```
if (testMe == 99) {  
    // good things happen  
}
```

Straightforward enough? Sure, what could be easier? However, one thing we've done at least once so far in this book, which you might not have noticed, is something like this:

At some point a variable gets set, in this case to the string "99".

Did we mention we're using a string this time?

```
let testMe = "99";
```

And later it gets compared with a number in a conditional test.

```
if (testMe == 99) {  
    // good things happen  
}
```

Now we have a string being compared to a number.

So what happens when we compare a number to a string? Mass chaos? Computer meltdown? Rioting in the streets?

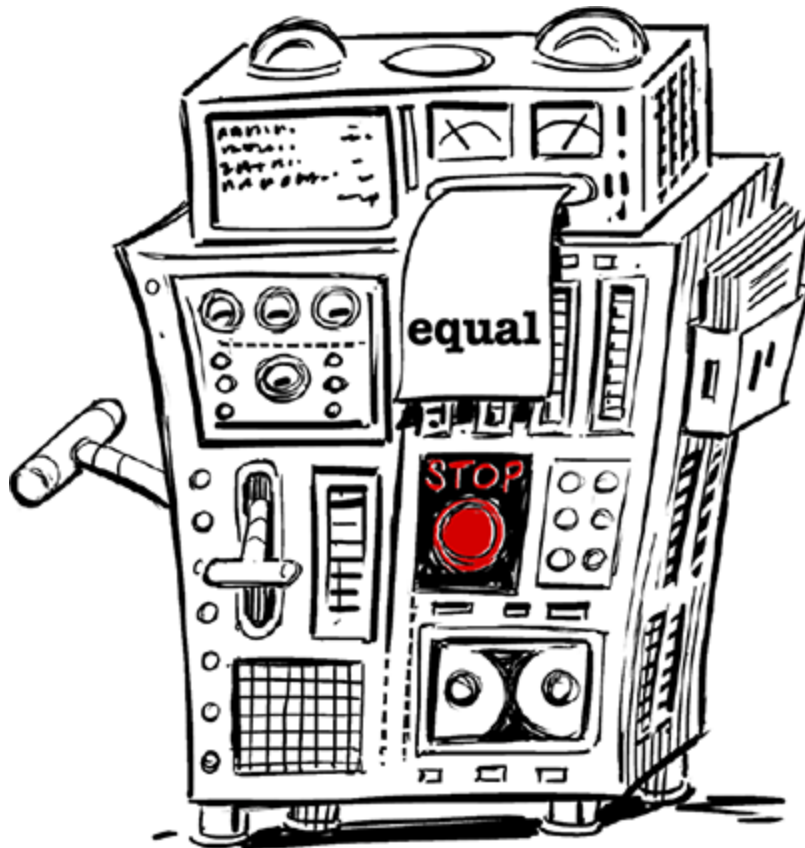
No, JavaScript is smart enough to determine that 99 and "99" are the same for all practical purposes. But what exactly is going on behind the scenes to make this work? Let's take a look...

BULLET POINTS

Just a quick reminder about the difference between assignment and equality:

- `let x = 99;` `=` is the assignment operator. It is used to assign a value to a variable.
 - `x == 99` `==` is a comparison operator. It is used to compare one value with another to see if they're equal.
-

Understanding the equality operator (otherwise known as ==)



You'd think that understanding equality would be a simple topic. After all, `1 == 1`, `"guacamole" == "guacamole"`, and `true == true`. But clearly there is more at work here if `"99" == 99`. What could be going on inside the equality operator to make that happen?

It turns out the `==` operator takes the types of its operands (that is, the two things you're comparing) into account when it does a comparison. You can break this down into two cases:

If the two values have the same type, just compare them

If the two values you are comparing have the same type, like two numbers or two strings, then the comparison works just like you would expect: the two values are compared against each other and the result is true if they are the same value. Easy enough.

If the two values have different types, try to convert them into the same type and then compare them

This is the more interesting case. Say you have two values with different types that you want to compare, like a number and a string. What JavaScript does is convert the string into a number and then compare the two values, like this:

NOTE

Note that the conversion is only temporary, so that the comparison can happen.

The diagram illustrates the process of comparing a number and a string in JavaScript. It shows two lines of code. The first line is `99 == "99"`. An arrow points from the string `"99"` to a handwritten note: "When you're comparing a number and a string, JavaScript converts the string to a number (if possible)...". A second arrow points from this note down to the second line of code, `99 == 99`. A third arrow points from this second line to another handwritten note: "...and then tries the comparison again. If they're equal the expression results in true, and otherwise false."

```
99 == "99"
```

When you're comparing a number and a string, JavaScript converts the string to a number (if possible)...

```
99 == 99
```

...and then tries the comparison again. If they're equal the expression results in true, and otherwise false.

Okay, that makes some intuitive sense, but what are the rules here? What if we compare a boolean to a number, or null to undefined, or some other combination of values? How do we know what's going to get converted into what? And why not convert the number into a string instead, or use some other scheme to test their equality? Well, this is defined by a fairly simple set of rules in the JavaScript specification that determine how the conversion happens when we compare two values with different types. This is one of those things you just need to internalize—once you've done that, you'll be on top of how comparisons work for the rest of your JavaScript career.

NOTE

This will also set you above your peers, and help you nail your next interview.

How equality converts its operands

(sounds more dangerous than it actually is)

You now know that when you compare two values that have different types, JavaScript will convert one type into another in order to compare them. If you're coming from another language this might seem strange, given that this is typically something you'd have to code explicitly rather than having it happen automatically. But no worries; in general, it's a useful thing in JavaScript *so long as you understand when and how it happens*. And that's what we've got to figure out now—when it happens and how it happens.

Here we go (in four simple cases):

Case #1: Comparing a number and a string

If you're comparing a string and a number, the same thing happens every time: the string is converted into a number, and the two numbers are then compared. This doesn't always go well, because not all strings can be converted to numbers. Let's see what happens in that case:

`99 == "vanilla"`

Once again, we're comparing a number and a string. But this time, when we try to convert the string to a number, we fail.

`99 == NaN`

When we try to convert "vanilla" to a number, we get NaN, and NaN isn't equal to anything. So, the result is false.

`false`

Case #2: Comparing a boolean with any other type

In this case, we convert the boolean to a number and compare. This might seem a little strange, but it's easier to digest if you just remember that true converts to 1 and false converts to 0. You also need to understand that sometimes this case requires doing more than one type conversion. Let's look at a few examples:

`1 == true`

We're comparing a number and a boolean. The true value is converted to the number 1.

`1 == 1`

And then we compare 1 to 1, which is true.

`true`

Here's another case, this time comparing a boolean to a string. Notice how more steps are needed:

`"1" == true`

We're comparing a string and a boolean. The true value is first converted to the number 1.

`"1" == 1`

And then we compare the string "1" to 1.

`1 == 1`

Now we use the rule from case #1, and the string is converted into a number.

`true`

And now we can finally compare a number to a number; in this case, the result is true.

Case #3: Comparing null and undefined

Comparing these values evaluates to true. That might seem odd as well, but it's the rule. For some insight, these values both essentially represent "no value" (that is, a variable with no value, or an object with no value), so they are considered to be equal:

undefined == null

true

← The values undefined and null are always equal.

Case #4: Oh, actually there is no case #4

That's it. You can pretty much determine the value of any equality with the previous rules. That said, there are a few edge cases and caveats. One caveat is that we still need to talk about comparing objects, which we'll get to in a bit. The other is around conversions that might catch you off guard. Here's one example:

1 == ""

← We're comparing a number and a string (case #1).

1 == 0

← The empty string is converted to the number 0...Believe it or not!

false

← Ah, too bad, 1 and 0 are not the same. So this evaluates to false.



If only I could find a way to test two values for **equality** without having to worry about their types being converted. A way to just test if two values are equal only if they have the same value *and* the same type. A way to not have to worry about all these rules and the mistakes they might cause. That would be dreamy. But I know it's just a fantasy...

How to get strict with equality

While we're making confessions, here's another one: there is not one but *two equality operators*. You've already been introduced to `==` (equality), and the other operator is `===` (strict equality).

That's right, three equals. You can use `===` in place of `==` anytime you want, but before you start doing that, let's make sure you understand how they differ.

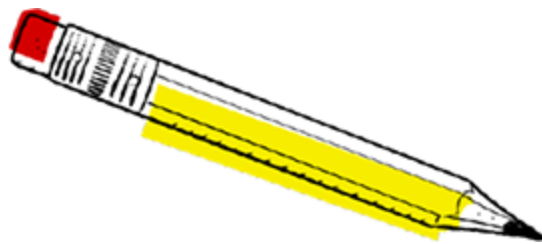
With `==`, you now know all the complex rules around how the operands are converted (if they're different types) when they're compared. With `===`, the rules are even more complicated.

Just kidding; actually, there is *only one rule* with `===`:

Two values are strictly equal only if they have the same type and the same value.

Read that again. What that means is that if two values have the same type, we compare them. If they don't, forget it, we're calling it false no matter what—no conversion, no figuring out complex rules, none of that. All you need to remember is that `===` will find two values equal *only if they are the same type and the same value*.

SHARPEN YOUR PENCIL



Write true or false below the operators `==` and `===` to represent the result of each of the following comparisons.

	<code>==</code>	<code>===</code>
<code>"42" == 42</code>	<u>true</u>	<u> </u>
<code>"0" == 0</code>	<u> </u>	<u> </u>
<code>"0" == false</code>	<u> </u>	<u> </u>
<code>"true" == true</code>	<u> </u>	<u> </u>
<code>true == (1 == "1")</code> Tricky!	<u> </u>	<u> </u>
		<code>"42" === 42</code>
		<code>"0" === 0</code>
		<code>"0" === false</code>
		<code>"true" === true</code>
		<code>true === (1 === "1")</code> Tricky!

➔ Solution in [“Sharpen your pencil Solution”](#)

Q: What happens if I compare a number, like 99, to a string, like “ninety-nine”, that can’t be converted to a number?

A: JavaScript will try to convert “ninety-nine” to a number, and it will fail, resulting in NaN. So the two values won’t be equal, and the result will be false.

Q: How does JavaScript convert strings to numbers?

A: It uses an algorithm to parse the individual characters of a string and try to turn each one of them into a number. So if you write “34”, it will look at “3,” and see that can be a 3, and then it will look at “4” and see that can be a 4. You can also convert strings like “1.2” to floating-point numbers—JavaScript is smart enough to recognize a string like this can still be a number.

Q: So, what if I try something like “true” == true?

A: That is comparing a string and a boolean, so according to the rules, JavaScript will first convert true to 1, and then compare “true” and 1. It will then try to convert “true” to a number and fail, so you’ll get false.

Q: So if there is both an == and a === operator, does that mean we have <= and <==, and >= and >==?

A: No. There are no <== and >== operators. You can use only <= and >=. These operators only know how to compare strings and numbers (true <= false doesn’t really make sense), so if you try to compare any values other than two strings or two numbers (or a string and a number), JavaScript will attempt to convert the types using the rules we’ve discussed.

Q: If I write 99 <= “100”, what happens?

A: Use the rules: “100” is converted to a number and then compared with 99. Because 99 is less than or equal to 100 (it’s less than), the result is true.

Q: Is there a !== operator?

A: Yes, and just like `===` is stricter than `==`, `!==` is stricter than `!=`. You use the same rules for `!=` as you do for `==`, except that you're checking for inequality instead of equality.

Q: Do we use the same rules when we're comparing, say, a boolean and a number with `<` and `>`, like `0 < true`?

A: Yup! And in that case, `true` gets converted to `1`, so you'll get `true` because `0` is less than `1`.

Q: It makes sense for a string to be equal to another string, but how can a string be less than or greater than another string?

A: Good question. What does it mean to say `"banana" < "mango"`? Well, with strings, you can use alphabetical order to tell if one string is less than or greater than another. Since `"banana"` begins with a `"b"` and `"mango"` with an `"m"`, `"banana"` is less than `"mango"` because `"b"` comes before `"m"` in the alphabet. And `"mango"` is less than `"melon"` because, while the first letters are the same, when we compare the second letters, `"a"` comes before `"e"`.

This alphabetical comparison can trip you up, however; for instance, `"Mango" < "mango"` is true, even though you might think that `"M"` is greater than `"m"` because its `"M"` is capitalized. The ordering of strings has to do with the ordering of the Unicode values that are used to represent each character in the computer (Unicode is a standard for representing characters digitally), and that ordering might not always be what you expect! For all the details, try googling "Unicode." But most of the time, the basic alphabetical ordering is all you need to know if one string is less than or greater than another.



Tonight's talk: **The equality and strict equality operators let us know who is boss.**

==

===

Ah, look who it is, Mr. Uptight.

Just keep in mind that several leading JavaScript gurus say that developers should use me, and only me. They think you should be taken out of the language altogether.

I'm up for a count of == versus === across all JavaScript code out in the world. You're going to come in way behind. It won't even be close.

You know, you might be right, but folks are slowly starting to get it, and those numbers are changing.

I don't think so. I provide a valuable service. Who doesn't want to, say, compare user input in the form of a string to a number every once in a while?

And with it come all the rules you have to keep in mind to even use ==. Keep life and code simple; use ===, and if you need to convert user input to a number there are methods for that.

When you were in grade school did you have to walk to school in the snow, every day, uphill, in both directions? Do you always have to do things the hard way?

Very funny. There's nothing wrong with being strict and

having clear-cut semantics around your comparisons. Bad, unexpected things can happen if you don't keep all the rules in mind.

The thing is, not only can I do the same comparisons you do, but I add value on top of that by doing some nice conversions of types.

Every time I look at your rules I throw up in my mouth a little. I mean, comparing a boolean to anything means I convert the boolean to a number? That doesn't seem very sensical to me.

You'd rather just throw your hands up, call it false, and go home?

No, but one can get a little too lax around your complex rules.

It's working so far. Look at all the code out there, a lot written by mere...well, *scripters*.

That's fine, but pages are getting more complex, more sophisticated. It's time to take on some best practices.

You mean like taking a shower after one of these conversations with you?

No, like sticking to `===`. It makes your code clearer and removes the potential for weird edge cases in comparisons.

Hmm. Well, ever considered just buying me out? I'd be happy to go spend my days on the beach, kicking back with a margarita in hand.

I didn't see that coming, I thought you'd defend your position as THE equality operator until the end. What gives?

Arguing about == versus === gets old. I mean, there are more interesting things to do in life.

I don't even know how to respond.

Look, here's the thing you have to deal with: people aren't going to just stop using ==. Sometimes it's really convenient. And people can use it in an educated way, taking advantage of it when it makes sense. Like the user input example—why the heck not use ==?

Well like I said, you never know when something is going to happen.

My new attitude is if people want to use you, great. I'm still here when they need me; and by the way, I still get a check every month no matter what they do! There's enough legacy code with == in the world—I'm never going off payroll.

Deep breath. There's a lot of debate around this topic, and different experts will tell you different things. Here's our take: traditionally, coders have used mostly `==` (equality) because, well, there wasn't a great awareness of the two operators and their differences. Today, we're more educated, and for most purposes `===` (strict equality) works just fine and is in some ways the safer route because you know what you're getting. With `==`, of course, you also know what you're getting, but with all the conversions it's hard sometimes to think through all the possibilities.

Now, there are times when `==` provides some nice convenience (like when you're comparing numbers to strings), and of course you should feel free to use `==` in those cases—especially now that, unlike many JavaScript coders, you know exactly what `==` does. Now that we've talked about `===`, you'll see us mostly shift gears in this book and predominantly use `===`, but we won't get dogmatic about it if there's a case where `==` makes our life easier and doesn't introduce issues.

NOTE

You'll also hear developers refer to `===` (strict equality) as the “identity” operator.

WHO DOES WHAT?

We had our descriptions of these operators all figured out, and then they got all mixed up. Can you help us figure out who does what? Be careful; we're not sure if each contender matches zero, one, or more descriptions. We've already figured one out, which is marked below.

Solution in [**“Who Does What Solution?”**](#)

Even more type conversions

Conditional expressions aren't the only place you're going to see type conversion. There are a few other operators that like to convert types when they get the chance. While these conversions are meant to be a convenience for you, the coder, and often they are, it's good to understand exactly where and when they might happen. Let's take a look.

Another look at concatenation, and addition

You've probably figured out that when you use the `+` operator with numbers you get *addition*, and when you use it with strings you get *concatenation*. But what happens when we mix the types of `+`'s operands? Let's find out.

If you try to add a number and a string, JavaScript converts the number to a string and concatenates the two. Kind of the opposite of what it does with equality:

If you put the string first and then use the `+` operator with a number, the same thing happens: the number is converted to a string and the two are joined by concatenation.

What about the other arithmetic operators?

When it comes the other arithmetic operators—like multiplication, division, and subtraction—JavaScript prefers to treat those as arithmetic operations, not string operations:

Q: Is + always interpreted as string concatenation when one of the operands is a string?

A: Yes. However, because + has what is called left-to-right associativity, if you have a situation like this:

```
let order = 1 + 2 + " pizzas";
```

you'll get "3 pizzas", not "12 pizzas", because, moving left to right, 1 is added to 2 first (and both are numbers), which results in 3. Next, we add 3 and a string, so 3 is converted to a string and concatenated with "pizza". To make sure you get the results you want, you can always use parentheses to force an operator to be evaluated first. This ensures you'll get 3 pizzas:

```
let order = (1 + 2) + " pizzas";
```

and this ensures you'll get 12 pizzas

```
let order = 1 + (2 + " pizzas");
```

Q: Is that it? Or are there more conversions?

A: There are some other places where conversion happens. For instance, the unary operator - (to make a negative number) will turn -true into -1. And concatenating a boolean with a string will create a string (like true + " love" is "true love"). These cases are fairly rare, and we've personally never needed these in practice, but now you know they exist.

Q: So if I want JavaScript to convert a string into a number to add it to another number, how can I do that?

A: There's a function that does this named Number (yes, it has a upper-case N). Use it like this:

```
let num = 3 + Number("4");
```

This statement results in num being assigned the value 7. The Number function takes an argument and, if possible, creates a number from it. If the argument can't be converted to a number, Number returns...wait for it...NaN.

SHARPEN YOUR PENCIL

Time to test that conversion knowledge. For each expression below, write the result in the blank next to it. We've done one for you. Check your answers at the end of the chapter before you go on.

Solution in **“Sharpen your pencil Solution”**

We're glad you're thinking about it. When it comes to object equality, there's a simple answer and there's a long, deep answer. The simple answer tackles the question: is this object equal to that object? That is, if I have two variables referencing objects, do they point to precisely the same object? We'll walk through that on the next page. The complex question involves object types, and the question of how two objects might or might not be the same type. We've seen that we can create objects that look like the same type, say two cars, but how do we ensure they really are? It's an important question, and one we're going to tackle head on in a later chapter.

How to determine if two objects are equal

Your first question might be: are we talking about `==` or `===`? Here's the good news: *if you're comparing two objects, it doesn't matter!* That is, if both operands are objects, then you can use either `==` or `===` because they work in exactly the same way. Here's what happens when we test two objects for equality.

When we test equality of two object variables, we compare the references to those objects.

Remember, variables hold references to objects, and so whenever we compare two objects, we're comparing object references.

Two references are equal only if they reference the same object.

The only time a test for equality between two variables containing object references returns true is when the two references point to the *same* object.

Here's a little code that helps find cars in the Earl's Autos parking lot.
Trace through this code and write the values of loc1 through loc4 below.

```
function findCarInLot(car) {  
  for (let i = 0; i < lot.length; i++) {  
    if (car === lot[i]) {  
      return i;  
    }  
  }  
  return -1;  
}  
  
let chevy = {  
  make: "Chevy",  
  model: "Bel Air"  
};  
  
let taxi = {  
  make: "Webville Motors",  
  model: "Taxi"  
};  
  
let fiat1 = {  
  make: "Fiat",  
  model: "500"  
};  
  
let fiat2 = {  
  make: "Fiat",  
  model: "500"  
};  
  
let lot = [chevy, taxi, fiat1, fiat2];  
  
let loc1 = findCarInLot(fiat2);  
let loc2 = findCarInLot(taxi);  
let loc3 = findCarInLot(chevy);  
let loc4 = findCarInLot(fiat1);
```

NOTE

Your answers here.

Solution in [“Sharpen your pencil Solution”](#)

The truthy is out there...

That’s right, we said truthy, not truth. We’ll say falsey too. What on earth are we talking about? Well, some languages are rather precise about true and false. JavaScript, not so much. In fact, JavaScript is kind of loose about true and false. How is it loose? Well, there are values in JavaScript that aren’t true or false, but that are nevertheless treated as true or false in a conditional. We call these values truthy and falsey precisely because they aren’t technically true or false, but they behave like they are (again, inside a conditional).

Now, here’s the secret to understanding truthy and falsey: *concentrate on knowing what is falsey, and then everything else you can consider truthy*. Let’s look at some examples of using these falsey values in a conditional:

NOTE

Some people write it “falsy.”

What JavaScript considers falsey

Again, the secret to learning what is truthy and what is falsey is to learn what’s falsey, and then consider everything else truthy.

There are five falsey values in JavaScript:

undefined is falsey.

null is falsey.

0 is falsey.

The empty string is falsey.

NaN is falsey.

To remember which values are truthy and which are falsey, just memorize the five falsey values—undefined, null, 0, “”, and NaN—and remember that everything else is truthy.

NOTE

Except false, which is...well, false.

So, every conditional test on the previous page evaluated to false. Did we mention every other value is truthy (except for false, of course)? Here are some examples of truthy values:

Time for a quick lie detector test. Figure out how many lies the perp tells, and whether the perp is guilty as charged, by determining which values are truthy and which values are falsey. Check your answer at the end of the chapter before you go on—and of course, feel free to try these out in the browser yourself.

```
function lieDetectorTest() {
  let lies = 0;

  let stolenDiamond = { };
  if (stolenDiamond) {
    console.log("You stole the diamond");
    lies++;
  }
  let car = {
    keysInPocket: null
  };
  if (car.keysInPocket) {
    console.log("Uh-oh, guess you stole the car!");
    lies++;
  }
  if (car.emptyGasTank) {
    console.log("You drove the car after you stole it!");
    lies++;
  }
  let foundYouAtTheCrimeScene = [ ];
  if (foundYouAtTheCrimeScene) {
    console.log("A sure sign of guilt");
    lies++;
  }
  if (foundYouAtTheCrimeScene[0]) {
    console.log("Caught with a stolen item!");
    lies++;
  }
  let yourName = " ";
  if (yourName) {
    console.log("Guess you lied about your name");
  }
}
```

```
        lies++;
    }
    return lies;
}
let numberOfLies = lieDetectorTest();
console.log("You told " + numberOfLies + " lies!");
if (numberOfLies >= 3) {
    console.log("Guilty as charged");
}
```

NOTE

A string with one space.

Solution in [“Sharpen your pencil Solution”](#)

BRAIN POWER

What do you think this code does? Do you see anything odd about the code, especially given what you know about primitive types?

```
let text = "YOU SHOULD NEVER SHOUT WHEN TYPING";
let presentableText = text.toLowerCase();
if (presentableText.length > 0) {
    alert(presentableText);
}
```

The Secret Life of Strings

Types always belong to one of two camps: they’re either primitive types or objects. Primitives live out fairly simple lives, while objects keep state

and have behavior (or, said another way, have properties and methods). Right?

Well, actually, while all that is true, it's not the whole story. As it turns out, strings are a little more mysterious. Check out this code:

How a string can look like a primitive and an object

How can a string masquerade as both a primitive and an object? Because JavaScript supports both. That is, with JavaScript you can create a string that is a primitive, and you can also create one that is an object (which supports lots of useful string manipulation methods). Now, we've never talked about how to create a string that is an object, and in most cases you won't need to explicitly do it yourself, because the JavaScript interpreter *will create string objects for you*, as needed.

Where and why might it do that? Let's look at the life of a string:

You don't need to. In general, you can just think of your strings as objects that have lots of great methods to help you manipulate the text in your strings. JavaScript will take care of all the details. So, look at it this way: you now have a better understanding of what is under the covers of JavaScript, but in your day-to-day coding you can just rely on JavaScript to do the right thing (and it does).

Q: Just making sure: do I ever have to keep track of where my string is a primitive and where it's an object?

A: Most of the time, no. The JavaScript interpreter will handle all the conversions for you. You just write your code, assuming a string supports the object properties and methods, and things will work as expected.

Q: Why does JavaScript support a string as both a primitive and an object?

A: Think about it this way: you get the efficiency of the simple string primitive type as long as you are doing basic string operations like comparison, concatenation, writing strings to the DOM, and so on. But if you need to do more sophisticated string processing, then you have the string object quickly at your disposal.

Q: Given an arbitrary string, how do I know if it is an object or a primitive?

A: A string is always a primitive unless you create it in a special way using an object constructor. We'll talk about object constructors later. And you can always use the `typeof` operator on your variable to see if it is of type string or object.

Q: Can other primitives act like objects?

A: Yes, numbers and booleans can also act like objects at times. However, neither of these has nearly as many useful properties as strings do, so you won't find you'll use this feature nearly as often as you do with strings. And remember, this all happens for you behind the scenes, so you don't really have to think about it much. Just use a property if you need to and let JavaScript handle the temporary conversion for you.

Q: How can I know all the methods and properties that are available for string objects?

A: That's where a good reference comes in handy. There are lots of online references that are helpful, and if you want a book, David Flanagan's *JavaScript: The Definitive Guide* has information about every string property and method in JavaScript. A search engine works pretty well too.

We're glad you asked. You're referring to a *template literal*, which is a fancy way of saying it's a convenient way to output variables or expressions within strings. It's super handy. As you noticed, template literals are enclosed in backticks (```) rather than standard quotes. Within the backticks you can include standard characters, just like any string, and you can also enclose expressions within `${}`, like `${name}` or `${people * cakeslices}`, to have those values included in the string:

```
let str = `My name is ${ name }`;
```

NOTE

Example of a template literal.

Let's take a moment to explore template literals, because they are so useful. After we've done that, we'll use them in the rest of the book.

How template literals work

Along with single and double quotes, you can use backticks, ```, to delimit strings. Like this:

When you use a backticks, you can also include expressions in your strings. To signify an expression, we surround it by a starting `${` and a

closing `}`. When the string is evaluated, any expressions are evaluated and converted to string values to output along with the rest of the string.

Like this:

Within template literals, what appears between the curly braces can be any expression that evaluates to or can be coerced into a string:

You'll be seeing a lot of template literals throughout the rest of the book, as it is a readable and convenient syntax that helps us create complex strings without a bunch of concatenation operators.

A five-minute tour of string properties and methods

Given that we're in the middle of talking about strings and you've just discovered that strings also support methods, let's take a little break from discussing weirdo types and look at a few of the more common string properties and methods you might want to use. A few string methods get used over and over, and it is highly worth your time to get to know them. So, on with the tour.

NOTE

A little pep talk: we could have written an entire chapter on each method and property that strings support. But that would have made this book 40 lbs and 2,000 pages long, and at this point, you really don't need it—you already get the basics of methods and objects, and all you need is a good reference if you really want to dive into the details of string processing.

String Soup

There's really no end to learning all the things you can do with strings. Here are a few more methods available to you. Just get a passing familiarity right now, and when you really need them you can look up the details.

Chair Wars

(or How Really Knowing Types Can Change Your Life)

Once upon a time in a software shop, two programmers were given the same spec and told to “build it.” The Really Annoying Project Manager forced the two coders to compete, by promising whoever delivered first gets one of those cool Aeron™ chairs all the Silicon Valley guys have. Brad, the hardcore hacker scripter, and Larry, the college grad, both knew this would be a piece of cake.

Larry, sitting in his cube, thought to himself, “What are the things this code has to *do*? It needs to make sure the string is long enough, it needs to make sure the fourth character is a dash, and it needs to make sure every other character is a number. I can use the string's `length` property, and I know how to access its characters using the `charAt` method.”

Brad, meanwhile, kicked back at the café and asked himself the same question. He first thought, “A string is an object, and there are lots of methods I can use to help validate the phone number. I’ll brush up on those and get this implemented quickly. After all, an object is an object.” Read on to see how Brad and Larry built their programs, and for the answer to the burning question: ***who got the Aeron?***

In Brad’s cube

Larry set about writing code based on the string methods. He wrote the code in no time:

In Brad’s cube

Brad wrote code to check for two numbers separated by a dash:

But wait! There’s been a spec change.

“Okay, *technically* you were first, Larry, because Brad was looking up how to use all those methods,” said the Manager, “but we have to add just one tiny thing to the spec. It’ll be no problem for crack programmers like you two.”

“If I had a dime for every time I’ve heard *that* one,” thought Larry, knowing that spec-change-no-problem was a fantasy. “And yet Brad looks strangely serene. What’s up with that?” Still, Larry held tight to his core belief that Brad’s fancy way, while cute, was just showing off, and that he’d win again in this next round and produce the code first.

Wait, can you think of any bugs Brad might have introduced with his use of `isNaN`?

Back in Larry's cube

Larry thought he could use most of his existing code; he just had to work in the edge case of the missing dash in the number. Either the number would be only seven digits, or it would be eight digits with a dash at index 3. Quickly, Larry coded the additions (which took a little testing to get right):

At the beach with Brad

Brad smiled, sipped his margarita, and quickly made his changes. He simply got the second part of the number using the length of the phone number minus four as the starting point for the substring, instead of hardcoding the starting point at a position that assumes a dash. That almost did it, but he did need to rewrite the test for the dash because it applies only when the phone number has a length of eight.

You can see Brad's updated code on the next page.

Err, we think Brad still has a bug. Can you find it?

Larry snuck in just ahead of Brad.

But the smirk on Larry's face melted when the Really Annoying Project Manager said, "Brad, your code is very readable and maintainable. Good job."

Larry shouldn't be too worried though, because as we know, good code has to do more than look pretty. The code still needs to get through QA, and we're not quite sure Brad's code works in all cases. What about you? Who do you think deserves the chair?

How would you rewrite Brad's code to use the split method instead?

The suspense is killing me. Who got the chair?

Amy from the second floor.

(Unbeknownst to all, the Project Manager had given the spec to *three* programmers.)

The lab crew continues to probe JavaScript using the **typeof** operator, and they're uncovering some more interesting things deep within the language. In the process, they've discovered a new operator, **instanceof**. With this one, they're truly on the cutting edge. Put your lab coat and safety goggles back on and see if you can help decipher this JavaScript and the results. *Warning: this is definitely going to be the weirdest code you've seen so far.*

Solution in [“IN THE LABORATORY, AGAIN SOLUTION”](#)

- There are two groups of types in JavaScript: **primitives** and objects. Any value that isn't a primitive type is an **object**.
 - The primitives are numbers, strings, booleans, null, and undefined (and BigInt and Symbol; check out the [Appendix A](#)). Everything else is an object.
 - **undefined** means that a variable (or property or array item) hasn't yet been initialized to a value.
 - **null** means "no object."
 - "NaN" stands for "Not a Number," although a better way to think of NaN is as a number that can't be represented in JavaScript. The type of NaN is number.
 - NaN never equals any other value, including itself, so to test for NaN use the function **isNaN**.
 - Test two values for equality using `==` or `===`.
 - If two operands have different types, the equality operator (`==`) will try to convert one of the operands into another type before testing for equality.
 - If two operands have different types, the strict equality operator (`===`) returns false.
 - You can use `===` if you want to be sure no type conversion happens; however, sometimes the type conversion of `==` can come in handy.
 - Type conversion is also used with other operators, like the arithmetic operators and string concatenation.
 - JavaScript has five **falsey** values: undefined, null, 0, "" (the empty string), and NaN. (False is also falsey, of course.) All other values are **truthy**.
 - Strings sometimes behave like objects. If you use a property or method on a primitive string, JavaScript will convert the string to an object temporarily, use the property, and then convert it back to a primitive string. This happens behind the scenes so you don't have to think about it.
 - Strings have many methods that are useful for string manipulation.
 - Two objects are equal only if the variables containing the object references point to the same object.
-

You're really expanding your JavaScript skills. Do a crossword to help it all sink in—all the answers are from this chapter.

ACROSS

2. The _____ operator tests two values to see if they're equal, after trying to convert the operands to the same type.
4. Sometimes strings masquerade as _____.
10. The only value in JavaScript that doesn't equal anything.
11. Two variables containing object references are equal only if they _____ the same object.
13. The value returned when you're expecting an object, and that object doesn't exist.
15. The _____ method is a string method that returns an array.
18. The _____ equality operator returns true only if the operands have the same type and the same value.
19. The type of Infinity is _____.
20. `null == undefined`

DOWN

1. The type of null in the JavaScript specification.
3. The value of a property that doesn't exist.

5. The _____ operator can be used to get the type of a value.
6. Your Fiat is parked at _____ Autos.
7. A handy way to include the value of an expression in a string is a _____ literal.
8. There are lots of handy string _____ you can use.
9. The weirdest value in the world.
12. To find a specific character at an index in a string, use the _____ method.
14. It's always 67 degrees in _____, Missouri.
16. There are _____ falsey values in JavaScript.
17. Who got the Aeron?

Solution in **“JavaScript cross Solution”**

WHO AM I?

From **“Who am I?”**

A bunch of JavaScript values and party crashers, in full costume, are playing a party game: “Who am I?” They give you a clue, and you try to guess who they are, based on what they say. Assume they always tell the truth about themselves. Draw an arrow from each sentence to the name of one attendee. We’ve already guessed one of them for you.

From [“In the Laboratory”](#)

In the laboratory, we like to take things apart, look under the hood, poke and prod, hook up our diagnostic tools, and check out what is really going on. Today, we’re investigating JavaScript’s type system, and we’ve found a little diagnostic tool called **typeof** to examine variables. Put your lab coat and safety goggles on, and come on in and join us.

The **typeof** operator is built into JavaScript. You can use it to probe the type of its operand (the thing you use it to operate on). Here’s an example:

Now it’s your turn. Collect the data for the following experiments:

BACK IN THE LABORATORY SOLUTIONFrom [“Back in the Laboratory”](#)

Oops, we forgot null in our test data. Here’s the missing test case:

```
let test10 = null;  
console.log(typeof test10);
```


From “Exercise”

We’ve been looking at some rather, um, interesting values so far in this chapter. Now, let’s take a look at some interesting behavior. Try adding the code below to the `<script>` element in a basic web page and see what you get in the console when you load up the page. You probably won’t get why yet, but see if you can take a guess about what might be going on.

```
if (99 == "99") {  
    console.log("A number equals a string!");  
} else {  
    console.log("No way a number equals a string");  
}
```

SHARPEN YOUR PENCIL SOLUTION**From “Sharpen your pencil”**

Write true or false below the operators `==` or `===` to represent the result of each of the following comparisons. Here’s our solution.

WHO DOES WHAT SOLUTION?

From **“Who Does What?”**

We had our descriptions of these operators all figured out, and then they got all mixed up. Can you help us figure out who does what? Be careful; we’re not sure if each contender matches zero, one, or more descriptions. Here’s our solution.

SHARPEN YOUR PENCIL SOLUTION

From **“Sharpen your pencil”**

Time to test that conversion knowledge. For each expression below, write the result in the blank next to it. Here’s our solution.

SHARPEN YOUR PENCIL SOLUTION

From **“Sharpen your pencil”**

Time for a quick lie detector test. Figure out how many lies the perp tells, and whether the perp is guilty as charged, by determining which values are truthy and which values are falsey. Here’s our solution. Did you try these out in the browser yourself?

From “Sharpen your pencil”

Here’s a little code that helps find cars in the Earl’s Autos parking lot. Trace through this code and write the values of loc1 through loc4 below.

IN THE LABORATORY, AGAIN SOLUTION

From “IN THE LABORATORY, AGAIN”

The lab crew continues to probe JavaScript using the **typeof** operator, and they’re uncovering some more interesting things deep within the language. In the process, they’ve discovered a new operator, **instanceof**. With this one, they’re truly on the cutting edge. Put your lab coat and safety goggles back on and see if you can help decipher this JavaScript and the results. *Warning: this is definitely going to be the weirdest code you’ve seen so far.*

JAVASCRIPT CROSS SOLUTION

From “JavaScript cross”

You’re really expanding your JavaScript skills. Do a crossword to help it all sink in—all the answers are from this chapter. Here’s our solution.
