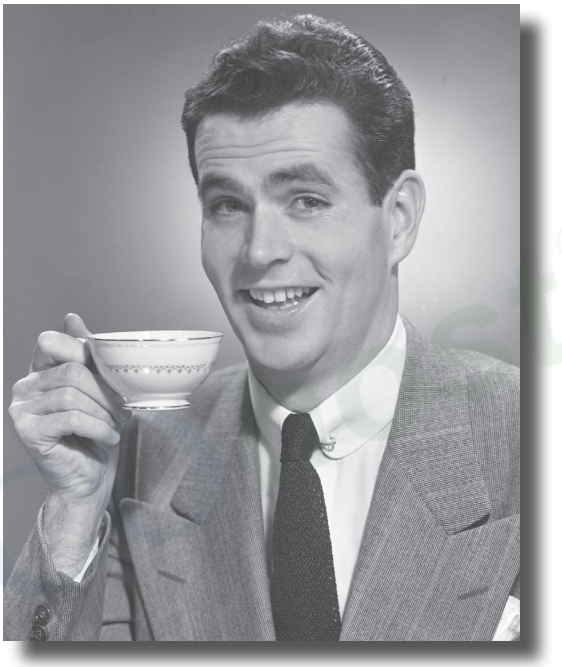
7 types, equality, conversion and all that jazz

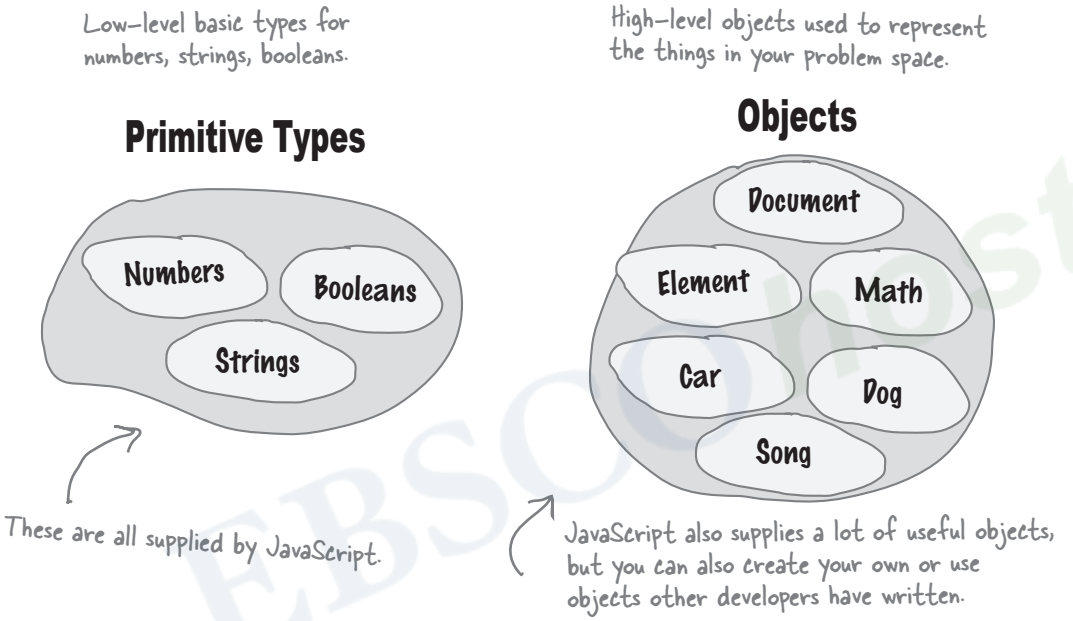
Serious types



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 way back about JavaScript? That it didn't have the luxury of a silver-spoon, academic, peer-reviewed language definition? Well that's true, but the 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 a lot of 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 scripter, you might think about sitting back, sipping on that Webville Martini, and taking a much needed break…

But you're not a mere scripter, and something is amiss. You have that 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 conversion. 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 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.

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

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

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

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

I'm the value that can't be assigned to a property when you create an object.

Tonight's attendees:

zero

empty object

null

undefined

NaN

infinity

area 51

...---...

{}

[]

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

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:

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

```
var x;

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

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

Or, how about an object property:

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

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

there are no Dumb Questions

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

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:

```
var subject = "Just a string";

var 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.

JavaScript console

```
string
```

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

```
var test1 = "abcdef";
var test2 = 123;
var test3 = true;
var test4 = {};
var test5 = [];
var test6;
var test7 = {"abcdef": 123};
var 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?

I remember from the DOM chapter that getElementById returns null, not undefined, if the id doesn't exist. What exactly is null, and why doesn't getElementById return undefined instead?

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:

```
var 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:

```
var test10 = null;

console.log(typeof test10);
```

Put your results here.

JavaScript console

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's look for the all-important header element.

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

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

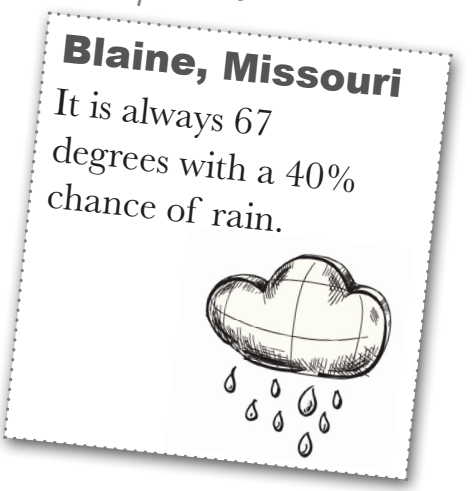
Keep in mind that getting `null` doesn't necessarily mean something is wrong. It may just mean something doesn't exist yet and needs to be created, or something doesn't exist and you can skip it. Let's say users have the ability to open or close a weather widget on your site. If a user has it open there's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

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

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

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.



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

WICKEDLYSMART'S
Believe It or Not!!

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:

```
var a = 0/0;
```

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

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

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

```
var 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'd 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.

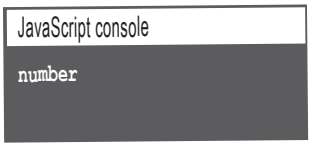
RIGHT!

It gets even weirder

So, 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:

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

Here's what we got.



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.

there are no Dumb Questions

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 just because NaN represents an unrepresentable numeric value, does not mean that those unrepresentable numbers are equal. For instance, take the 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 1.7976931348623157E+10308 (or -1.7976931348623157E+10308 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.

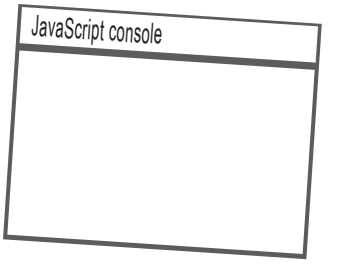
Q: Just to cover every detail, did we say what the type of null is?

A: 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". And 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.

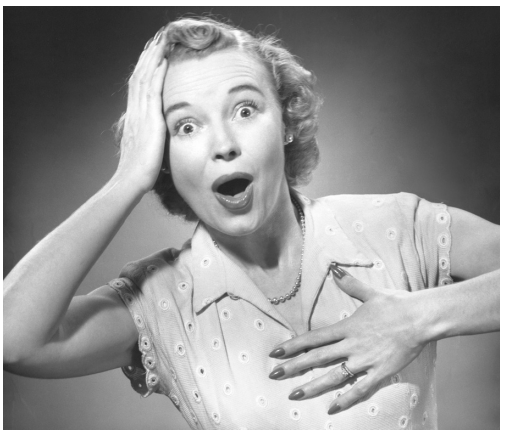
We've been looking at some rather, um, interesting, values so far in this chapter. Now, let's take a look at some interesting behavior. Try adding the code below to the <script> element in a basic web page and see what you get in the console when you load up the page. You won't get why yet, but see if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



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.

```
var 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, that 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?

```
var 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…

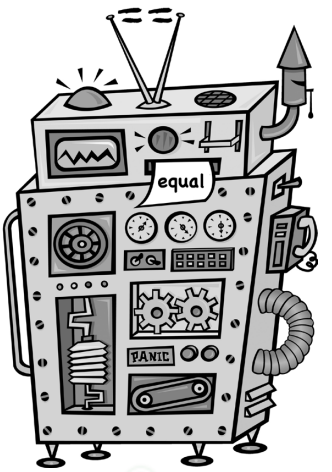
BULLET POINTS

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

- `var 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 compares the two values. Like this:

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

```
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. Now, if they're equal, the expression results in true, false otherwise.

Okay, that makes some intuitive sense, but what are the rules here? What if I compare a boolean to a number, or null to undefined, or some other combination of values? How do I 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 the rest of your JavaScript career.

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)

So what we know is 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 this is typically something you'd have to code explicitly rather than have 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. And 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
```

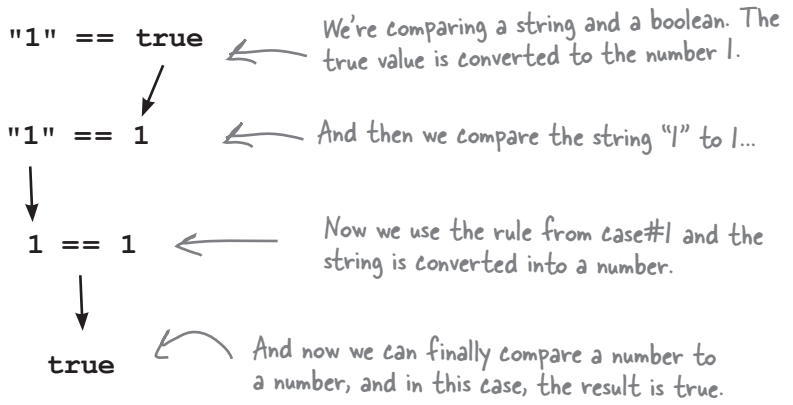
We're comparing a number and a boolean. The true value is converted to the number 1.

```
1 == 1
```

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

```
true
```

Here's another case; this time a boolean is compared to a string. Notice how more steps are needed.

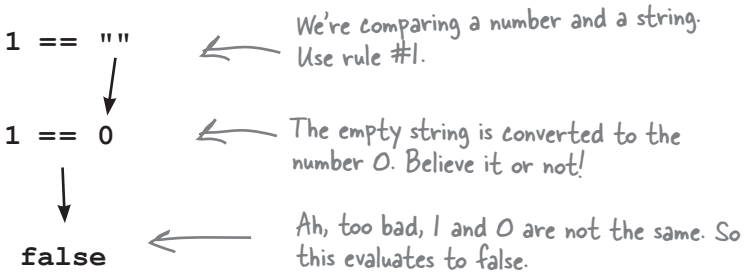


CASE#3: Comparing null and undefined.

Comparing these values evalutates 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.

CASE#4: Oh, actually there is no case #4.

That's it. You can pretty much determine the value of any equality with these 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 talk about in a bit. The other is around conversions that might catch you off guard. Here's one example:



How to get strict with equality

Editor's note: Make sure we have a photo release on file from Doug Crockford.

While we're making confessions, here's another one: there are 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, 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

For each comparison below write true or false below the operators == and === to represent the result of the comparison:

	==	===	
`"42" == 42`	true		`"42" === 42`
`"0" == 0`			`"0" === 0`
`"0" == false`			`"0" === false`
`"true" == true`			`"true" === true`
`true == (1 == "1")`			`true === (1 === "1")`

Tricky!

Tricky!

there are no Dumb Questions

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 a == 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: So 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 !==?

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?

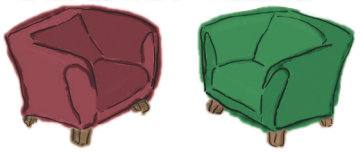
A: Yup! And in that case, true gets converted to 1, so you'll get true because 0 is less than 1.

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 know if one string is less than or greater than another. Because "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.

Fireside Chats



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

==

Ah look who it is, Mr. Uptight.

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.

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?

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?

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

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

===

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.

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

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.

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.

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.

==

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

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

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.

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

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 ==?

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.

===

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

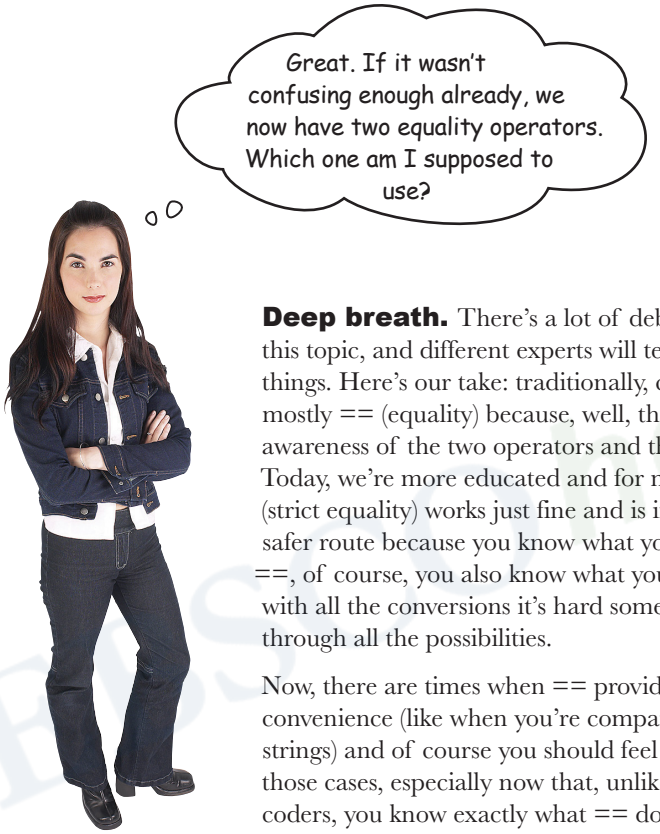
That's fine but pages are getting more complex, more sophisticated. It's time to take on some best practices.

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

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

I don't even know how to respond.

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



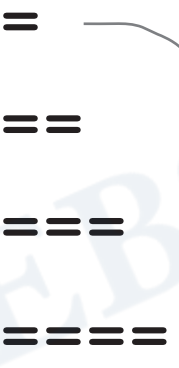
Deep breath. There's a lot of debate around this topic, and different experts will tell you different things. Here's our take: traditionally, coders have used mostly == (equality) because, well, there wasn't a great awareness of the two operators and their differences. Today, we're more educated and 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.

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

WHO DOES WHAT?

We had our descriptions for 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:



Compares values to see if they are equal. This is the considerate equality operator. He'll go to the trouble of trying to convert your types to see if you are really equal.

Compares values to see if they are equal. This guy won't even consider values that have different types.

Assigns a value to a variable.

Compares object references and returns true if they are the same and false otherwise.

Even more type conversions...

Conditional statements 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:

```
var addi = 3 + "4";
```

When we have a string added to a number, we get concatenation, not addition.

The result variable is set to "34" (not 7).

```
var plusi = "4" + 3;
```

Same here... we get "43".

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.

```
var multi = 3 * "4";
```

Here, JavaScript converts the string "4" to the number 4, and multiplies it by 3, resulting in 12.

```
var divi = 80 / "10";
```

Here the string "10" is converted to the number 10. Then 80 is divided by the number 10, resulting in 8.

```
var mini = "10" - 5;
```

With minus, the "10" is converted to the number 10, so we have 10 minus 5, which is 5.

there are no Dumb Questions

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:

```
var 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:

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

ensures you'll get 3 pizzas, and

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

ensures you'll get 12 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 would I do that?

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

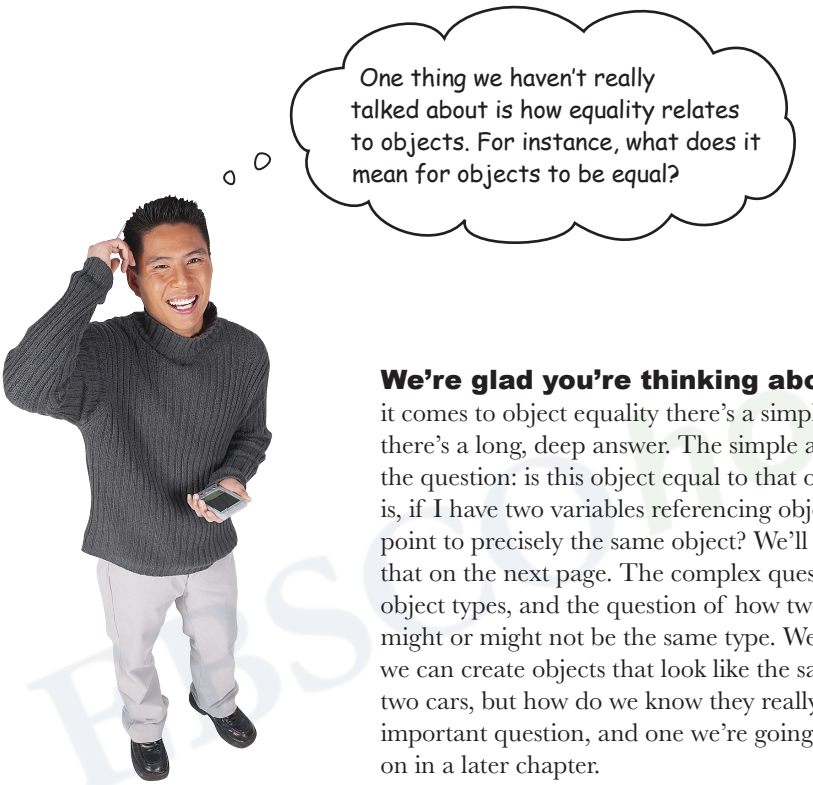
```
var num = 3 + Number("4");
```

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.

```
Infinity - "1"                    _____
"42" + 42                         "4242"
2 + "1 1"                         _____
99 + 101                          _____
"1" - "1"                         _____
console.log("Result: " + 10/2)    _____
3 + " bananas " + 2 + " apples"   _____
```



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 know 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

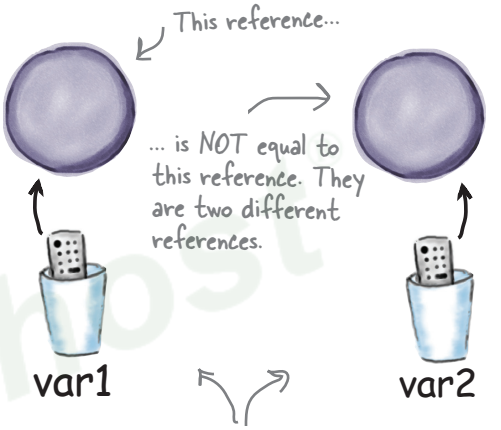
Your first question might be: are we talking about == or ===? Here's the good news: *if you're comparing two objects, it doesn't matter*! That is, if both operands are objects, then you can use either == or === because they work in exactly the same way. Here's what happens when you test two objects for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

```
if (var1 === var2) {
    // wow, these are the same object!
}
```

Not in this case!

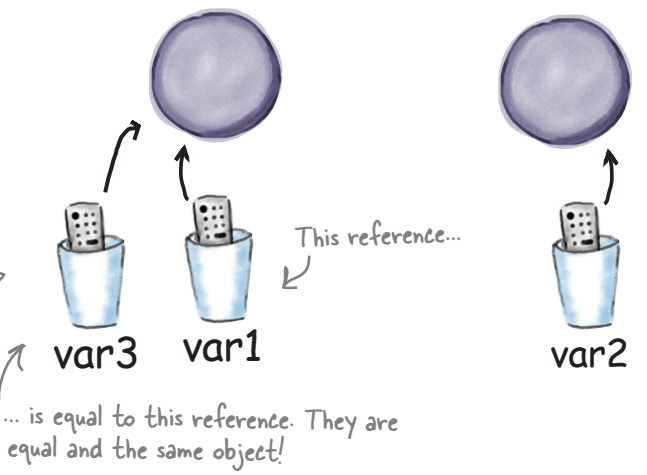


Two references are equal only if they reference the same object

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

```
if (var1 === var3) {
    // wow, these are the same object!
}
```

Finally, two object references that are equal.



Sharpen your pencil

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

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

var lot = [chevy, taxi, fiat1, fiat2];

var loc1 = findCarInLot(fiat2);
var loc2 = findCarInLot(taxi);
var loc3 = findCarInLot(chevy);
var loc4 = findCarInLot(fiat1);
```

Your answers here.

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).

Some people write it "falsy."

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:

```
var testThis;
if (testThis) {
   // do something
}
```

Okay that's weird, we know this variable will be undefined in the conditional test. Does this work? Is this legal JavaScript? (Answer: yes.)

```
var element = document.getElementById("elementThatDoesntExist");
if (element) {
   // do something
}
```

Here the value of element is null. What's that going to do?

```
if (0) {
   // do another thing
}
```

We're testing 0?

Now we're doing a conditional test on an empty string. Anyone want to place bets?

```
if ("") {
   // does code here ever get evaluated? Place your bets.
}
```

Wait, now we're using NaN in a boolean condition? What's that going to evaluate to?

```
if (NaN) {
   // Hmm, what's NaN doing in a boolean test?
}
```

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.

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:

This is an array. It's not undefined, null, zero, "" or NaN. It has to be true!

```
if ([]) {
   // this will happen
}

var element = document.getElementById("elementThatDoesExist");
if (element) {
   // so will this
}

if (1) {
   // gonna happen
}

var string = "mercy me";
if (string) {
   // this will happen too
}
```

This time we have an actual element object. That's not falsy either, so it's truthy.

Only the number 0 is falsey, all others are truthy.

Only the empty string is falsey, all other strings are truthy.

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. 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() {
    var lies = 0;

    var stolenDiamond = { };
    if (stolenDiamond) {
        console.log("You stole the diamond");
        lies++;
    }
    var 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++;
    }
    var foundYouAtTheCrimeScene = [ ];
    if (foundYouAtTheCrimeScene) {
        console.log("A sure sign of guilt");
        lies++;
    }
    if (foundYouAtTheCrimeScene[0]) {
        console.log("Caught with a stolen item!");
        lies++;
    }
    var yourName = " ";  A string with one space.
    if (yourName) {
        console.log("Guess you lied about your name");
        lies++;
    }
    return lies;
}
var numberOfLies = lieDetectorTest();
console.log("You told " + numberOfLies + " lies!");
if (numberOfLies >= 3) {
    console.log("Guilty as charged");
}
```

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

```
var text = "YOU SHOULD NEVER SHOUT WHEN TYPING";
var 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 a primitive type or an object. 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:

```
var emot = "XOxxOO";
var hugs = 0;
var kisses = 0;

emot = emot.trim();
emot = emot.toUpperCase();

for(var i = 0; i < emot.length ; i++) {
    if (emot.charAt(i) === "X") {
       hugs++;
    } else if (emot.charAt(i) == "O") {
       kisses++;
    }
}
```

This looks like a normal, primitive string.

Wait a sec, calling a method on a string?

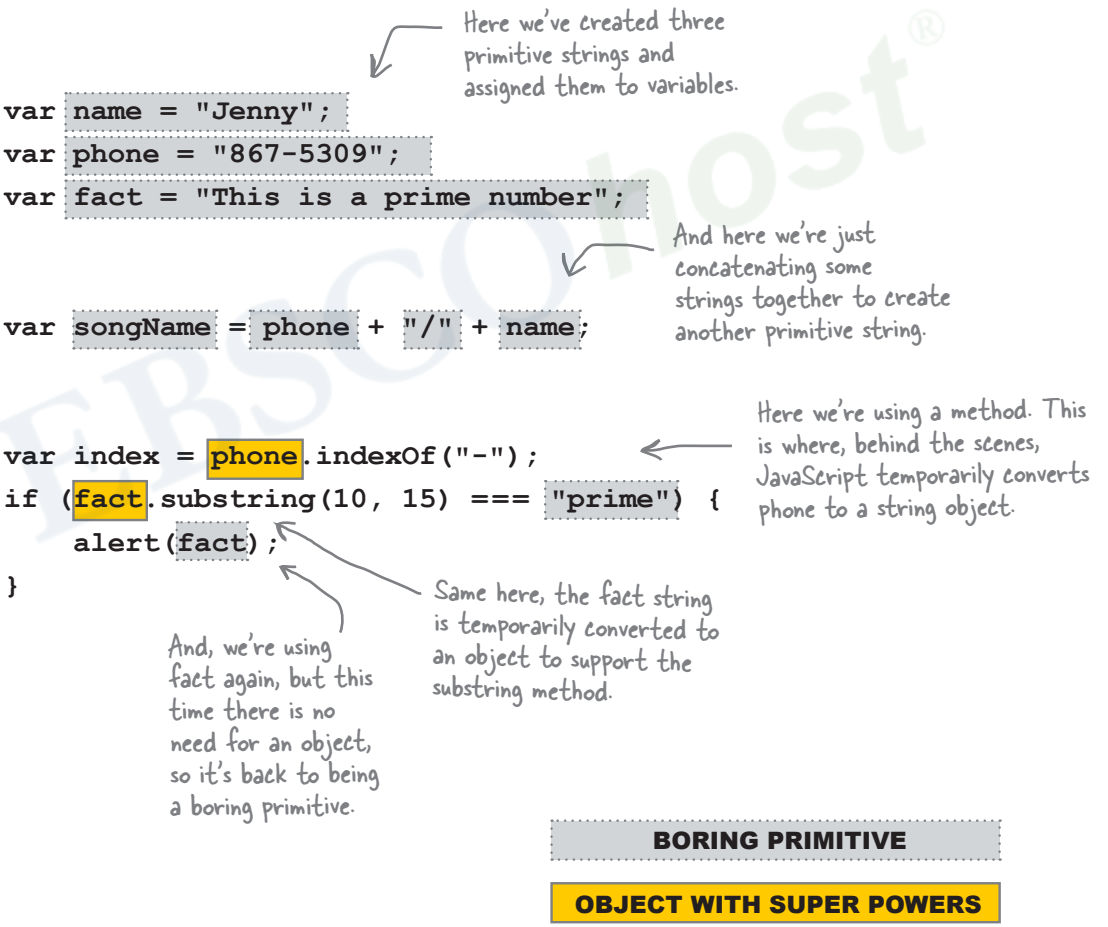
And a string with a property?

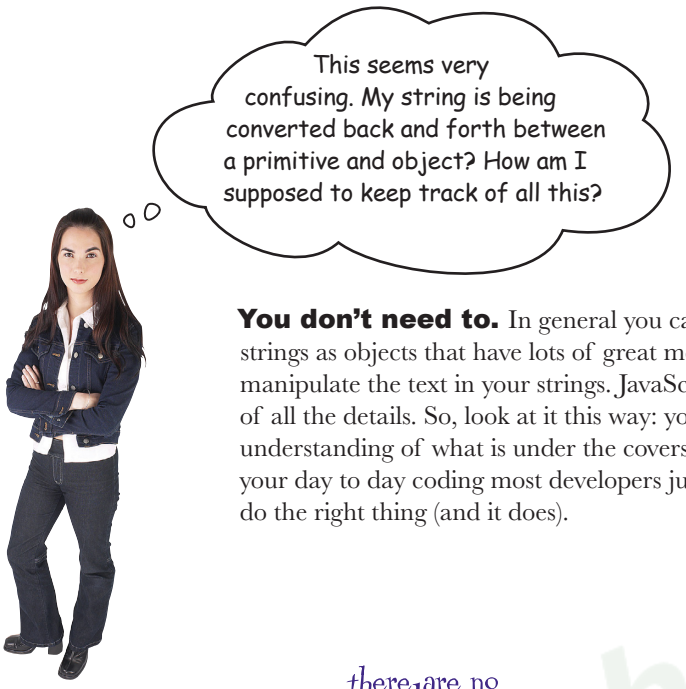
More methods?

How a string can look like a primitive and an object

How does 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 don't need to explicitly do it yourself, because the JavaScript interpreter *will create string objects for you*, as needed.

Now, 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 most developers just rely on JavaScript to do the right thing (and it does).

there are no Dumb Questions

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 conversion 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 string 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 arbitary string, how do I know if it is an object or 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, *JavaScript: The Definitive Guide* has a reference guide with information about every string property and method in JavaScript. Google works pretty well too.

A five-minute tour of string methods (and properties)

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 talking about weirdo types and look at a few of the more common string 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.

A little pep talk: we could pull you aside and write an entire chapter on every method and property that strings support. Not only would that make this book 40 lbs and 2000 pages long, but 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.

the length property

The length property holds the number of characters in the string. It's quite handy for iterating through the characters of the string.

We use the length property to iterate over each character in the string.

```
var input = "jenny@wickedlysmart.com";
for(var i = 0; i < input.length; i++) {
    if (input.charAt(i) === "@") {
        console.log("There's an @ sign at index " + i);
    }
}
```

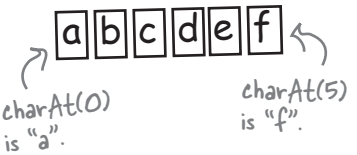
And the charAt method to get the character at a particular index in the string.

```
JavaScript console
There's an @ sign at index 5
```

the charAt method

The charAt method takes an integer number between zero and the length of the string (minus one), and returns a string containing the single character at that position of the string. Think of the string a bit like an array, with each character at an index of the string, with the indices starting at 0 (just like an array). If you give it an index that is greater than or equal to the length of the string, it returns the empty string.

Note that JavaScript doesn't have a character type. So characters are returned as new strings containing one character.



the indexOf method

This method takes a string as an argument and returns the index of the first character of the first occurrence of that argument in the string.

Here's the string we're going to call indexOf on.

```
var phrase = "the cat in the hat";
```

And our goal is to find the first occurence of "cat" in phrase.

```
var index = phrase.indexOf("cat");
console.log("there's a cat sitting at index " + index);
```

The index of the first cat is returned.

JavaScript console

```
There's a cat sitting at index 4
```

You can also add a second argument, which is the starting index for the search.

```
index = phrase.indexOf("the", 5);
console.log("there's a the sitting at index " + index);
```

Because we're starting the search at index 5, we're skipping the first "the" and finding the second "the" at index 11.

JavaScript console

```
There's a the sitting at index 11
```

```
index = phrase.indexOf("dog");
console.log("there's a dog sitting at index " + index);
```

Note if the string can't be found, then -1 is returned as the index.

JavaScript console

```
There's a dog sitting at index -1
```

the substring method

Give the substring method two indices, and it will extract and return the string contained within them.

```
var data = "name|phone|address";
var val = data.substring(5, 10);
console.log("Substring is " + val);
```

Here's the string we're going to call substring on.

We'd like the string from index 5 and up to (but not including) 10 returned.

We get back a new string with the characters from index 5 to 10.

```
JavaScript console
Substring is phone
```

You can omit the second index and substring will extract a string that starts at the first index and then continues until the end of the original string.

```
val = data.substring(5);
console.log("Substring is now " + val);
```

```
JavaScript console
Substring is now phone|address
```

the split method

The split method takes a character that acts as a delimiter, and breaks the string into parts based on the delimiter.

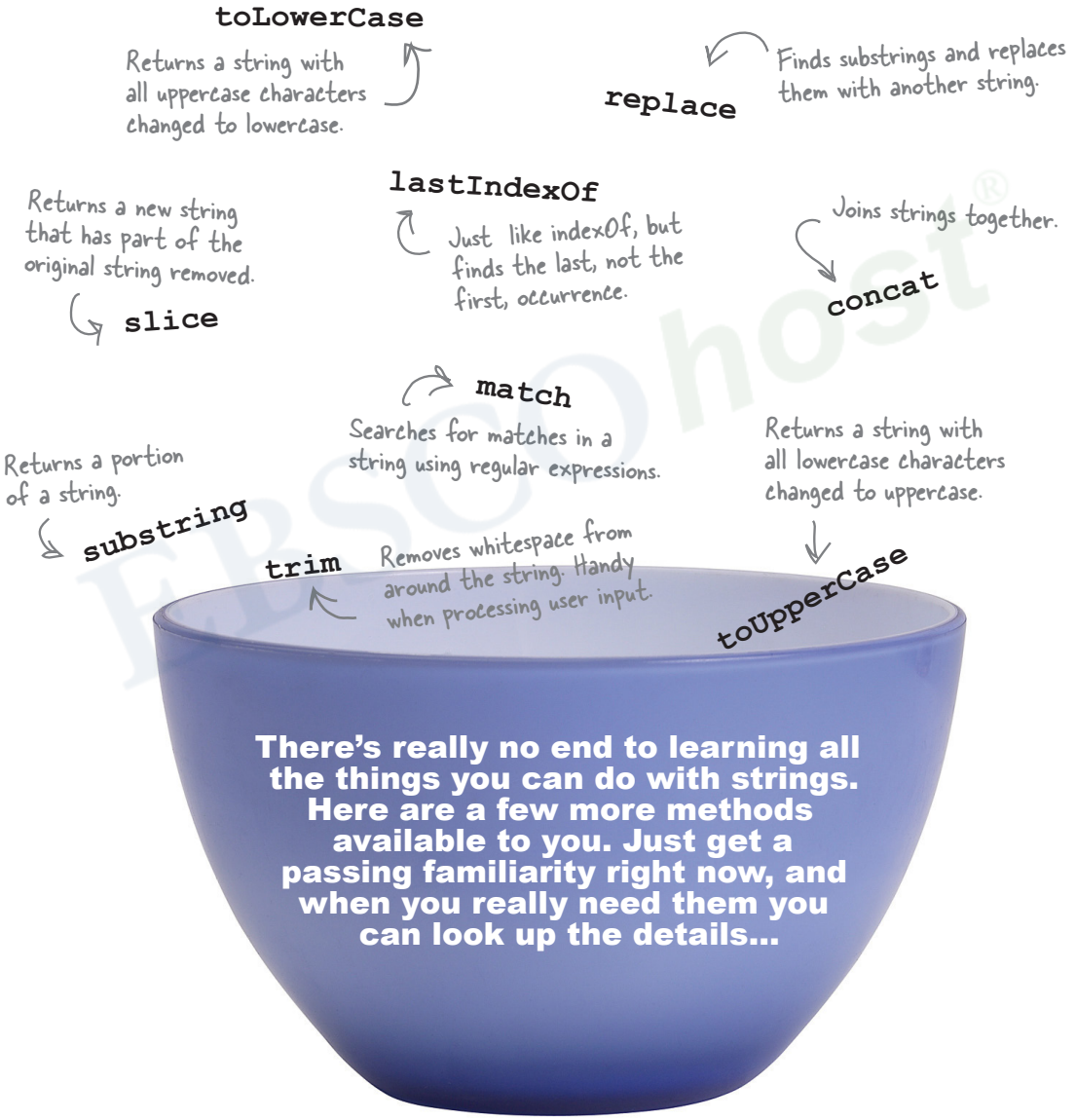
```
var data = "name|phone|address";
var vals = data.split("|");
console.log("Split array is ", vals);
```

Split uses the delimiter to break the original string into pieces, which are returned in an array.

Notice here we're passing two arguments to console.log separated by a comma. This way, the vals array doesn't get converted to a string before it's displayed in the console.

```
JavaScript console
Split array is ["name", "phone", "address"]
```

String Soup



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 that whoever delivers 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 middle 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 cafe and thought to himself, "What are the things this code has to do?" 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 your burning question: ***who got the Aeron?***

The spec

Take a phone number of the form:

"123-4567"

and write code to accept or reject it. To be accepted the number should have seven digits, 0 through 9, with a dash in the middle.

The chair

In Larry's cube

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

```
function validate(phoneNumber) {
    if (phoneNumber.length !== 8) {
       return false;
    }
    for (var i = 0; i < phoneNumber.length; i++) {
       if (i === 3) {
           if (phoneNumber.charAt(i) !== '-') {
              return false;
           }
       } else if (isNaN(phoneNumber.charAt(i))) {
          return false;
       }
    }
    return true;
}
```

Larry uses the length property of the string object to see how many characters it has.

He uses the charAt method to examine each character of the string.

First, he makes sure character three has a dash.

Then he makes sure each character zero through two and four through six has a number in it.

In Brad's cube

Brad wrote code to check for two numbers and a dash:

```
function validate(phoneNumber) {
   if (phoneNumber.length !== 8) {
      return false;
   }
   var first = phoneNumber.substring(0,3);
   var second = phoneNumber.substring(4);
   if (phoneNumber.charAt(3) !== "-" || isNaN(first) || isNaN(second)) {
      return false;
   }
   return true;
}
```

Brad starts just like Larry...

But he uses his knowledge of the string methods.

He uses the substring method to create a string containing three characters from zero up to character three.

And again to start at character index four up to the end of the string.

Then he tests all the conditions for being a correct phone number in one conditional.

And interestingly, knowing it or not, he's depending on some type conversions here to convert a string to a number, and then making sure it's a number with isNaN. Clever!

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.

BRAIN POWER

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

Take a phone number of the form:

"123-4567"

and write code to accept or reject it. To be accepted the number should have seven digits, 0 through 9, with **an optional** dash in the middle.

What got added to the spec

Back in Larry's cube

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

```
function validate(phoneNumber) {
    if (phoneNumber.length  > 8 ||
          phoneNumber.length < 7) {
       return false;
    }
    for (var i = 0; i < phoneNumber.length; i++) {
       if (i === 3) {
           if (phoneNumber.length === 8 &&
                    phoneNumber.charAt(i) !== '-') {
              return false;
           } else if (phoneNumber.length === 7 &&
                    isNaN(phoneNumber.charAt(i))) {
              return false;
           }
       } else if (isNaN(phoneNumber.charAt(i))) {
          return false;
       }
    }
    return true;
}
```

Larry had to make a few additions to his logic. Not a lot of code, but it's getting a bit hard to decipher.

At Brad's laptop at the beach

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.

About the same number of changes as Larry, but Brad's code is still easier to read.

```
function validate(phoneNumber) {
    if (phoneNumber.length  > 8 ||
        phoneNumber.length < 7) {
        return false;
    }
    var first = phoneNumber.substring(0,3);
    var second = phoneNumber.substring(phoneNumber.length - 4);

    if (isNaN(first) || isNaN(second)) {
       return false;
    }
    if (phoneNumber.length === 8) {
       return (phoneNumber.charAt(3) === "-");
    }
    return true;
}
```

Now Brad's getting the second number using the total length of the phone number to get the starting point.

And he's validating the dash only if the number is eight characters.

Here we're returning the result of evaluating the conditional, which will be true or false.

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

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

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."

But Larry shouldn't be too worried, because, as we know, there is more than just code beauty at work. This 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?

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.)

Wow, a one-liner! Check out how this works in the appendix!

Here's Amy's code.

```
function validate(phoneNumber) {
    return phoneNumber.match(/^\d{3}-?\d{4}$/);
}
```

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.*

Here's the code. Read it, run it, alter it, massage it, see what it does...

How strange. Doesn't this look a bit like a mix of a function and an object?

```
function Duck(sound) {
    this.sound = sound;
    this.quack = function() {console.log(this.sound);}
}

var toy = new Duck("quack quack");

toy.quack();

console.log(typeof toy);
console.log(toy instanceof Duck);
```

Hmm "new". We've haven't seen that before. But we're guessing we should read this as, create a new Duck and assign it to the toy variable.

If it looks like an object, and walks like an object... let's test it.

Okay, and here is instanceof...

Be sure to check your output with the answers at the end of the chapter. But just what does this all mean? Ah, we'll be getting to all that in just a couple of chapters. And, in case you didn't notice, you are well on your way to being a pretty darn advanced JavaScript coder. This is serious stuff!

JavaScript console

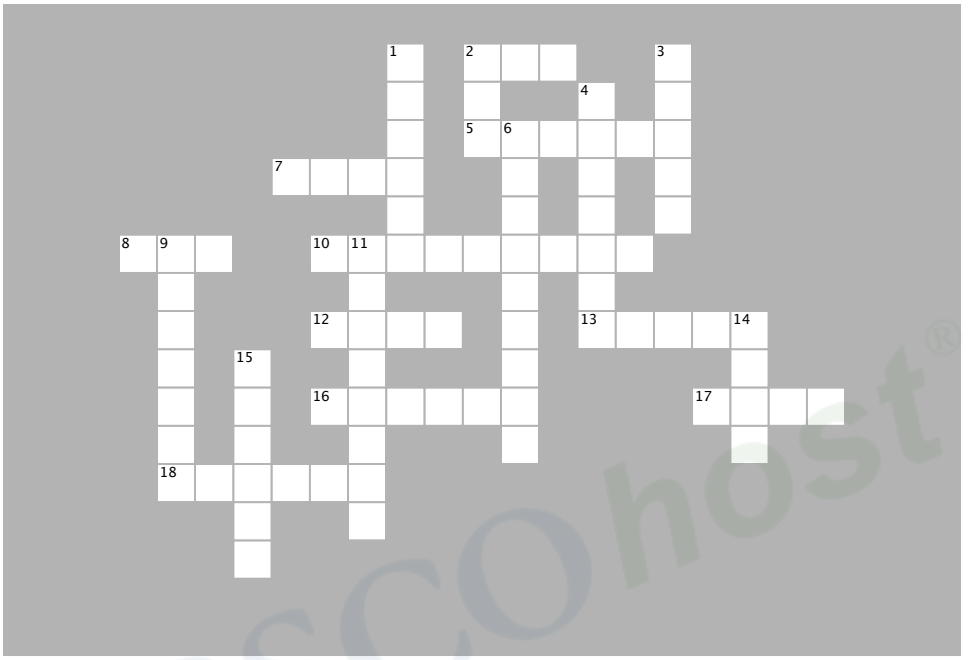
Put your results here. Are there any surprises?

BULLET POINTS

- 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. 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 false. 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.
- The string has 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.

JavaScript cross

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



ACROSS

2. The only value in JavaScript that doesn't equal anything.
5. The type of Infinity is _____.
7. There are _____ falsey values in JavaScript.
8. Who got the Aeron?
10. Two variables containing object references are equal only if they _____ the same object.
12. The value returned when you're expecting an object, and that object doesn't exist.
13. The _____ method is a string method that returns an array.
16. It's always 67 degrees in _____, Missouri.
17. The type of null in the JavaScript specification.
18. The _____ equality operator returns true only if the operands have the same type and the same value.

DOWN

1. The _____ operator can be used to get the type of a value.
2. The weirdest value in the world.
3. Your Fiat is parked at _____ Autos.
4. Sometimes strings masquerade as _____.
6. The value of a property that doesn't exist.
9. There are lots of handy string _____ you can use.
11. The _____ operator tests two values to see if they're equal, after trying to convert the operands to the same type.
14. null == undefined
15. To find a specific character at an index in a string, use the _____ method.

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. Fill in the blank next to each sentence with the name of one attendee. We've already guessed one of them.

Here's our solution:

Tonight's attendees:

- I get returned from a function when there is no return statement. → zero
- I'm the value of a variable when I haven't been assigned a value. → null
- I'm the value of an array item that doesn't exist in a sparse array. → undefined
- I'm the value of a property that doesn't exist. → NaN
- I'm the value of a property that's been deleted. → infinity
- I'm the value that can't be assigned to a property when you create an object. → area 51
- ...---...
- {}
- []

IN THE LABORATORY SOLUTION

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:

```
var subject = "Just a string";

var 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.

JavaScript console

```
string
```

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

```
var test1 = "abcdef";
var test2 = 123;
var test3 = true;
var test4 = {};
var test5 = [];
var test6;
var test7 = {"abcdef": 123};
var 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

```
string
number
boolean
object
object
undefined
object
object
function
```

Here are our results.

BACK IN THE LABORATORY SOLUTION

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

```
var test10 = null;

console.log(typeof test10);
```

Here's our result.

JavaScript console

```
object
```

Exercise Solution

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 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");
}
```

Here's what we got.

JavaScript console

```
A number equals a string!
```

Sharpen your pencil Solution

For each comparison below write true or false below the operators == or === to represent the result of the comparison:

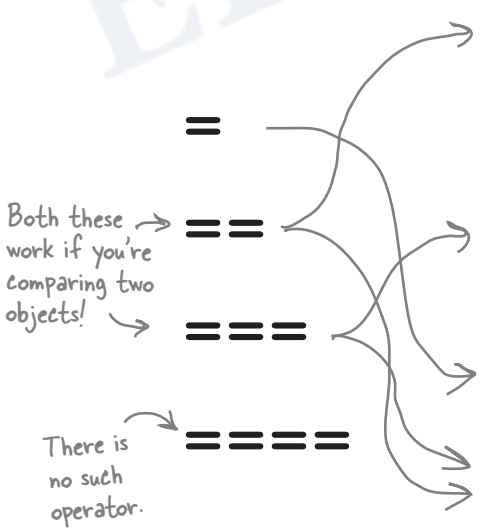
	==	===	
`"42" == 42`	true	false	`"42" === 42`
`"0" == 0`	true	false	`"0" === 0`
`"0" == false`	true	false	`"0" === false`
`"true" == true`	false	false	`"true" === true`
`true == (1 == "1")`	true	false	`true === (1 === "1")`

Tricky!

If you replace both == with ===, then the result is false.

WHO DOES WHAT? SOLUTION

We had our descriptions for 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:



Compares values to see if they are equal. This is the considerate equality operator. He'll go to the trouble of trying to convert your types to see if you are really equal.

Compares values to see if they are equal. This guy won't even consider values that have different types.

Assigns a value to a variable.

Compares object references and returns true if they are the same and false otherwise.

Sharpen your pencil Solution

For each expression below, write the result in the blank next to it. We've done one for you. Here's our solution.

`Infinity - "1"`	Infinity
`"42" + 42`	"4242"
`2 + "1 1"`	"21 1"
`99 + 101`	200
`"1" - "1"`	0
`console.log("Result: " + 10/2)`	"Result: 5"
`3 + " bananas " + 2 + " apples"`	"3 bananas 2 apples"

"1" is converted to 1, and Infinity – 1 is Infinity.

Both strings are converted to 1, and 1–1 is 0.

10/2 happens first, and the result is concatenated to the string "Result: "

Each + is concatenation because for both, one operand is a string.

Sharpen your pencil Solution

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?

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

    var stolenDiamond = { };
    if (stolenDiamond) {
        console.log("You stole the diamond");
        lies++;
    }
    var 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++;
    }
    var foundYouAtTheCrimeScene = [ ];
    if (foundYouAtTheCrimeScene) {
        console.log("A sure sign of guilt");
        lies++;
    }
    if (foundYouAtTheCrimeScene[0]) {
        console.log("Caught with a stolen item!");
        lies++;
    }
    var yourName = " ";
    if (yourName) {
        console.log("Guess you lied about your name");
        lies++;
    }
    return lies;
}
var numberOfLies = lieDetectorTest();
console.log("You told " + numberOfLies + " lies!");
if (numberOfLies >= 3) {
    console.log("Guilty as charged");
}
```

Any object is truthy, even an empty one.

This perp didn't steal the car because the value of the keysInPocket property is null, which is falsey.

And the perp didn't drive the car either, because the emptyGasTank property is undefined, which is falsey.

But [ ] (an empty array) is truthy, so the perp was caught on the scene.

There is no item in the array, so the array item at 0 is undefined, which is falsey. Hmm, the perp must have hidden the stash already.

A string with one space.

Any non-empty string is truthy, even if it just has one space!

The number of lies is 3 so we think the perp is guilty.

JavaScript console

```
You stole the diamond
A sure sign of guilt
Guess you lied about your name
You told 3 lies!
Guilty as charged
```

Sharpen your pencil Solution

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

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

var lot = [chevy, taxi, fiat1, fiat2];

var loc1 = findCarInLot(fiat2);
var loc2 = findCarInLot(taxi);
var loc3 = findCarInLot(chevy);
var loc4 = findCarInLot(fiat1);
```

Here are our answers.

3
1
0
2

Earl, from Earl's Autos.

IN THE LABORATORY, AGAIN

SOLUTION

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.*

Here's the code. Read it, run it, alter it, massage it, see what it does...

How strange. Doesn't this look a bit like a mix of a function and an object?

```
function Duck(sound) {
    this.sound = sound;
    this.quack = function() {console.log(this.sound);}
}

var toy = new Duck("quack quack");

toy.quack();

console.log(typeof toy);
console.log(toy instanceof Duck);
```

Hmm "new". We've haven't seen that before. But we're guessing we should read this as, create a new Duck and assign it to the toy variable.

If it looks like an object, and walks like an object... let's test it.

Okay, and here is instanceof...

Just what does this all mean? Ah, we'll be getting to all that in just a few chapters. And, in case you didn't notice, you are well on your way to being a pretty darn advanced JavaScript coder. This is serious stuff!

JavaScript console

```
quack quack
object
true
```

The toy acts like an object... we can call its method.

And the type is object.

But it is an "instanceof" a Duck, whatever that means... Hmm.

Here are our results.

JavaScript cross Solution

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