You Don't Know JS: Types & Grammar

Chapter 2: Values

`array`s, `string`s, and `number`s are the most basic building-blocks of any program, but JavaScript has some unique characteristics with these types that may either delight or confound you.
Let's look at several of the built-in value types in JS, and explore how we can more fully understand and correctly leverage their behaviors.

Arrays

As compared to other type-enforced languages, JavaScript `array`s are just containers for any type of value, from `string` to `number` to `object` to even another `array` (which is how you get multidimensional `array`s).

```
var a = [ 1, "2", [3] ];

a.length;               // 3
a[0] === 1;             // true
a[2][0] === 3;   // true
```

You don't need to presize your `array`s (see "Arrays" in Chapter 3), you can just declare them and add values as you see fit:

```
var a = [ ];

a.length;       // 0

a[0] = 1;
a[1] = "2";
a[2] = [ 3 ];

a.length;       // 3
```

Warning: Using `delete` on an `array` value will remove that slot from the `array`, but even if you remove the final element, it does **not** update the `length` property, so be careful! We'll cover the `delete` operator itself in more detail in Chapter 5.
Be careful about creating "sparse" `array`s (leaving or creating empty/missing slots):

```
var a = [ ];

a[0] = 1;
// no `a[1]` slot set here
a[2] = [ 3 ];

a[1];           // undefined
```

```
a.length;       // 3
```

While that works, it can lead to some confusing behavior with the "empty slots" you leave in between. While the slot appears to have the `undefined` value in it, it will not behave the same as if the slot is explicitly set (`a[1] = undefined`). See "Arrays" in Chapter 3 for more information.

`array`s are numerically indexed (as you'd expect), but the tricky thing is that they also are objects that can have `string` keys/properties added to them (but which don't count toward the `length` of the `array`):

```
var a = [ ];

a[0] = 1;
a["foobar"] = 2;

a.length;               // 1
a["foobar"];    // 2
a.foobar;               // 2
```

However, a gotcha to be aware of is that if a `string` value intended as a key can be coerced to a standard base-10 `number`, then it is assumed that you wanted to use it as a `number` index rather than as a `string` key!

```
var a = [ ];

a["13"] = 42;

a.length; // 14
```

Generally, it's not a great idea to add `string` keys/properties to `array`s. Use `object`s for holding values in keys/properties, and save `array`s for strictly numerically indexed values.

Array-Likes

There will be occasions where you need to convert an `array`-like value (a numerically indexed collection of values) into a true `array`, usually so you can call array utilities (like `indexOf(..)`, `concat(..)`, `forEach(..)`, etc.) against the collection of values.

For example, various DOM query operations return lists of DOM elements that are not true `array`s but are `array`-like enough for our conversion purposes. Another common example is when functions expose the `arguments` (`array`-like) object (as of ES6, deprecated) to access the arguments as a list.

One very common way to make such a conversion is to borrow the `slice(..)` utility against the value:

```
function foo() {
        var arr = Array.prototype.slice.call( arguments );
        arr.push( "bam" );
        console.log( arr );
}
```

```
foo( "bar", "baz" ); // ["bar","baz","bam"]
```

If `slice()` is called without any other parameters, as it effectively is in the above snippet, the default values for its parameters have the effect of duplicating the `array` (or, in this case, `array`-like).
As of ES6, there's also a built-in utility called `Array.from(..)` that can do the same task:

```
...
var arr = Array.from( arguments );
...
```

Note: `Array.from(..)` has several powerful capabilities, and will be covered in detail in the *ES6 & Beyond* title of this series.

Strings

It's a very common belief that `string`s are essentially just `array`s of characters. While the implementation under the covers may or may not use `array`s, it's important to realize that JavaScript `string`s are really not the same as `array`s of characters. The similarity is mostly just skin-deep.
For example, let's consider these two values:

```
var a = "foo";
var b = ["f","o","o"];
```

Strings do have a shallow resemblance to `array`s -- `array`-likes, as above -- for instance, both of them having a `length` property, an `indexOf(..)` method (`array` version only as of ES5), and a `concat(..)` method:

```
a.length;                                                   // 3
b.length;                                                   // 3

a.indexOf( "o" );                                   // 1
b.indexOf( "o" );                                   // 1

var c = a.concat( "bar" );                  // "foobar"
var d = b.concat( ["b","a","r"] );      // ["f","o","o","b","a","r"]

a === c;                                            // false
b === d;                                            // false

a;                                                              // "foo"
b;                                                              //
["f","o","o"]
```

So, they're both basically just "arrays of characters", right? **Not exactly**:

```
a[1] = "O";
b[1] = "O";

a; // "foo"
```

```
b; // ["f","O","o"]
```

JavaScript `string`s are immutable, while `array`s are quite mutable. Moreover, the `a[1]` character position access form was not always widely valid JavaScript. Older versions of IE did not allow that syntax (but now they do). Instead, the *correct* approach has been `a.charAt(1)`.

A further consequence of immutable `string`s is that none of the `string` methods that alter its contents can modify in-place, but rather must create and return new `string`s. By contrast, many of the methods that change `array` contents actually *do* modify in-place.

```
c = a.toUpperCase();
a === c;// false
a;                      // "foo"
c;                      // "FOO"

b.push( "!" );
b;                      // ["f","O","o","!"]
```

Also, many of the `array` methods that could be helpful when dealing with `string`s are not actually available for them, but we can "borrow" non-mutation `array` methods against our `string`:

```
a.join;                 // undefined
a.map;                  // undefined

var c = Array.prototype.join.call( a, "-" );
var d = Array.prototype.map.call( a, function(v){
        return v.toUpperCase() + ".";
} ).join( "" );

c;                              // "f-o-o"
d;                              // "F.O.O."
```

Let's take another example: reversing a `string` (incidentally, a common JavaScript interview trivia question!). `array`s have a `reverse()` in-place mutator method, but `string`s do not:

```
a.reverse;              // undefined

b.reverse();    // ["!","o","O","f"]
b;                              // ["!","o","O","f"]
```

Unfortunately, this "borrowing" doesn't work with `array` mutators, because `string`s are immutable and thus can't be modified in place:

```
Array.prototype.reverse.call( a );
// still returns a String object wrapper (see Chapter 3)
// for "foo" :(
```

Another workaround (aka hack) is to convert the `string` into an `array`, perform the desired operation, then convert it back to a `string`.

```
var c = a
        // split `a` into an array of characters
        .split( "" )
```

```
    // reverse the array of characters
    .reverse()
    // join the array of characters back to a string
    .join( "" );

c; // "oof"
```

If that feels ugly, it is. Nevertheless, *it works* for simple `string`s, so if you need something quick-n-dirty, often such an approach gets the job done.

Warning: Be careful! This approach **doesn't work** for `string`s with complex (unicode) characters in them (astral symbols, multibyte characters, etc.). You need more sophisticated library utilities that are unicode-aware for such operations to be handled accurately. Consult Mathias Bynens' work on the subject: *Esrever* (https://github.com/mathiasbynens/esrever).

The other way to look at this is: if you are more commonly doing tasks on your "strings" that treat them as basically *arrays of characters*, perhaps it's better to just actually store them as `array`s rather than as `string`s. You'll probably save yourself a lot of hassle of converting from `string` to `array` each time. You can always call `join("")` on the `array` *of characters* whenever you actually need the `string` representation.

Numbers

JavaScript has just one numeric type: `number`. This type includes both "integer" values and fractional decimal numbers. I say "integer" in quotes because it's long been a criticism of JS that there are not true integers, as there are in other languages. That may change at some point in the future, but for now, we just have `number`s for everything.

So, in JS, an "integer" is just a value that has no fractional decimal value. That is, `42.0` is as much an "integer" as `42`.

Like most modern languages, including practically all scripting languages, the implementation of JavaScript's `number`s is based on the "IEEE 754" standard, often called "floating-point." JavaScript specifically uses the "double precision" format (aka "64-bit binary") of the standard.

There are many great write-ups on the Web about the nitty-gritty details of how binary floating-point numbers are stored in memory, and the implications of those choices. Because understanding bit patterns in memory is not strictly necessary to understand how to correctly use `number`s in JS, we'll leave it as an exercise for the interested reader if you'd like to dig further into IEEE 754 details.

Numeric Syntax

Number literals are expressed in JavaScript generally as base-10 decimal literals. For example:

```
var a = 42;
var b = 42.3;
```

The leading portion of a decimal value, if `0`, is optional:

```
var a = 0.42;
var b = .42;
```

Similarly, the trailing portion (the fractional) of a decimal value after the `.`, if `0`, is optional:

```
var a = 42.0;
var b = 42.;
```

Warning: `42.` is pretty uncommon, and probably not a great idea if you're trying to avoid confusion when other people read your code. But it is, nevertheless, valid.

By default, most `number`s will be outputted as base-10 decimals, with trailing fractional `0`s removed. So:

```
var a = 42.300;
var b = 42.0;

a; // 42.3
b; // 42
```

Very large or very small `number`s will by default be outputted in exponent form, the same as the output of the `toExponential()` method, like:

```
var a = 5E10;
a;                                      // 50000000000
a.toExponential();       // "5e+10"

var b = a * a;
b;                                      // 2.5e+21

var c = 1 / a;
c;                                      // 2e-11
```

Because `number` values can be boxed with the `Number` object wrapper (see Chapter 3), `number` values can access methods that are built into the `Number.prototype` (see Chapter 3). For example, the `toFixed(..)` method allows you to specify how many fractional decimal places you'd like the value to be represented with:

```
var a = 42.59;

a.toFixed( 0 ); // "43"
a.toFixed( 1 ); // "42.6"
a.toFixed( 2 ); // "42.59"
a.toFixed( 3 ); // "42.590"
a.toFixed( 4 ); // "42.5900"
```

Notice that the output is actually a `string` representation of the `number`, and that the value is `0`-padded on the right-hand side if you ask for more decimals than the value holds.

`toPrecision(..)` is similar, but specifies how many *significant digits* should be used to represent the value:

```
var a = 42.59;

a.toPrecision( 1 ); // "4e+1"
a.toPrecision( 2 ); // "43"
a.toPrecision( 3 ); // "42.6"
a.toPrecision( 4 ); // "42.59"
a.toPrecision( 5 ); // "42.590"
a.toPrecision( 6 ); // "42.5900"
```

You don't have to use a variable with the value in it to access these methods; you can access these methods directly on `number` literals. But you have to be careful with the `.` operator. Since `.` is a valid numeric character, it will first be interpreted as part of the `number` literal, if possible, instead of being interpreted as a property accessor.

```
// invalid syntax:
42.toFixed( 3 ); // SyntaxError

// these are all valid:
(42).toFixed( 3 );       // "42.000"
0.42.toFixed( 3 );       // "0.420"
42..toFixed( 3 );        // "42.000"
```

`42.toFixed(3)` is invalid syntax, because the `.` is swallowed up as part of the `42.` literal (which is valid -- see above!), and so then there's no `.` property operator present to make the `.toFixed` access.

`42..toFixed(3)` works because the first `.` is part of the `number` and the second `.` is the property operator. But it probably looks strange, and indeed it's very rare to see something like that in actual JavaScript code. In fact, it's pretty uncommon to access methods directly on any of the primitive values. Uncommon doesn't mean *bad* or *wrong*.

Note: There are libraries that extend the built-in `Number.prototype` (see Chapter 3) to provide extra operations on/with `number`s, and so in those cases, it's perfectly valid to use something like `10..makeItRain()` to set off a 10-second money raining animation, or something else silly like that.

This is also technically valid (notice the space):

```
42 .toFixed(3); // "42.000"
```

However, with the `number` literal specifically, **this is particularly confusing coding style** and will serve no other purpose but to confuse other developers (and your future self). Avoid it.

`number`s can also be specified in exponent form, which is common when representing larger `number`s, such as:

```
var onethousand = 1E3;                          // means 1 * 10^3
var onemilliononehundredthousand = 1.1E6; // means 1.1 * 10^6
```

`number` literals can also be expressed in other bases, like binary, octal, and hexadecimal.

These formats work in current versions of JavaScript:

```
0xf3; // hexadecimal for: 243
0Xf3; // ditto

0363; // octal for: 243
```

Note: Starting with ES6 + `strict` mode, the `0363` form of octal literals is no longer allowed (see below for the new form). The `0363` form is still allowed in non-`strict` mode, but you should stop using it anyway, to be future-friendly (and because you should be using `strict` mode by now!).
As of ES6, the following new forms are also valid:

```
0o363;          // octal for: 243
0O363;          // ditto

0b11110011;     // binary for: 243
0B11110011; // ditto
```

Please do your fellow developers a favor: never use the `0O363` form. `0` next to capital `O` is just asking for confusion. Always use the lowercase predicates `0x`, `0b`, and `0o`.

Small Decimal Values

The most (in)famous side effect of using binary floating-point numbers (which, remember, is true of **all** languages that use IEEE 754 -- not *just* JavaScript as many assume/pretend) is:

```
0.1 + 0.2 === 0.3; // false
```

Mathematically, we know that statement should be `true`. Why is it `false`?
Simply put, the representations for `0.1` and `0.2` in binary floating-point are not exact, so when they are added, the result is not exactly `0.3`. It's **really** close: `0.30000000000000004`, but if your comparison fails, "close" is irrelevant.
Note: Should JavaScript switch to a different `number` implementation that has exact representations for all values? Some think so. There have been many alternatives presented over the years. None of them have been accepted yet, and perhaps never will. As easy as it may seem to just wave a hand and say, "fix that bug already!", it's not nearly that easy. If it were, it most definitely would have been changed a long time ago.
Now, the question is, if some `number`s can't be *trusted* to be exact, does that mean we can't use `number`s at all? **Of course not.**
There are some applications where you need to be more careful, especially when dealing with fractional decimal values. There are also plenty of (maybe most?) applications that only deal with whole numbers ("integers"), and moreover, only deal

with numbers in the millions or trillions at maximum. These applications have been, and always will be, **perfectly safe** to use numeric operations in JS.

What if we *did* need to compare two `number`s, like `0.1 + 0.2` to `0.3`, knowing that the simple equality test fails?
The most commonly accepted practice is to use a tiny "rounding error" value as the *tolerance* for comparison. This tiny value is often called "machine epsilon," which is commonly `2^-52` (`2.220446049250313e-16`) for the kind of `number`s in JavaScript.
As of ES6, `Number.EPSILON` is predefined with this tolerance value, so you'd want to use it, but you can safely polyfill the definition for pre-ES6:

```
if (!Number.EPSILON) {
	Number.EPSILON = Math.pow(2,-52);
}
```

We can use this `Number.EPSILON` to compare two `number`s for "equality" (within the rounding error tolerance):

```
function numbersCloseEnoughToEqual(n1,n2) {
	return Math.abs( n1 - n2 ) < Number.EPSILON;
}

var a = 0.1 + 0.2;
var b = 0.3;

numbersCloseEnoughToEqual( a, b );					// true
numbersCloseEnoughToEqual( 0.0000001, 0.0000002 );	// false
```

The maximum floating-point value that can be represented is roughly `1.798e+308` (which is really, really, really huge!), predefined for you as `Number.MAX_VALUE`. On the small end, `Number.MIN_VALUE` is roughly `5e-324`, which isn't negative but is really close to zero!

Safe Integer Ranges

Because of how `number`s are represented, there is a range of "safe" values for the whole `number` "integers", and it's significantly less than `Number.MAX_VALUE`.
The maximum integer that can "safely" be represented (that is, there's a guarantee that the requested value is actually representable unambiguously) is `2^53 - 1`, which is `9007199254740991`. If you insert your commas, you'll see that this is just over 9 quadrillion. So that's pretty darn big for `number`s to range up to.
This value is actually automatically predefined in ES6, as `Number.MAX_SAFE_INTEGER`.
Unsurprisingly, there's a minimum value, `-9007199254740991`, and it's defined in ES6 as `Number.MIN_SAFE_INTEGER`.
The main way that JS programs are confronted with dealing with such large numbers is when dealing with 64-bit IDs from databases, etc. 64-bit numbers cannot be

represented accurately with the `number` type, so must be stored in (and transmitted to/from) JavaScript using `string` representation.
Numeric operations on such large ID `number` values (besides comparison, which will be fine with `string`s) aren't all that common, thankfully. But if you *do* need to perform math on these very large values, for now you'll need to use a *big number* utility. Big numbers may get official support in a future version of JavaScript.

Testing for Integers

To test if a value is an integer, you can use the ES6-specified `Number.isInteger(..)`:

```
Number.isInteger( 42 );          // true
Number.isInteger( 42.000 );      // true
Number.isInteger( 42.3 );        // false
```

To polyfill `Number.isInteger(..)` for pre-ES6:

```
if (!Number.isInteger) {
	Number.isInteger = function(num) {
		return typeof num == "number" && num % 1 == 0;
	};
}
```

To test if a value is a *safe integer*, use the ES6-specified `Number.isSafeInteger(..)`:

```
Number.isSafeInteger( Number.MAX_SAFE_INTEGER );  // true
Number.isSafeInteger( Math.pow( 2, 53 ) );                        // false
Number.isSafeInteger( Math.pow( 2, 53 ) - 1 );            // true
```

To polyfill `Number.isSafeInteger(..)` in pre-ES6 browsers:

```
if (!Number.isSafeInteger) {
	Number.isSafeInteger = function(num) {
		return Number.isInteger( num ) &&
			Math.abs( num ) <= Number.MAX_SAFE_INTEGER;
	};
}
```

32-bit (Signed) Integers

While integers can range up to roughly 9 quadrillion safely (53 bits), there are some numeric operations (like the bitwise operators) that are only defined for 32-bit `number`s, so the "safe range" for `number`s used in that way must be much smaller.
The range then is `Math.pow(-2,31)` (`-2147483648`, about -2.1 billion) up to `Math.pow(2,31)-1` (`2147483647`, about +2.1 billion).
To force a `number` value in `a` to a 32-bit signed integer value, use `a | 0`. This works because the `|` bitwise operator only works for 32-bit integer values (meaning it can only pay attention to 32 bits and any other bits will be lost). Then, "or'ing" with zero is essentially a no-op bitwise speaking.

Note: Certain special values (which we will cover in the next section) such as `NaN` and `Infinity` are not "32-bit safe," in that those values when passed to a bitwise operator will pass through the abstract operation `ToInt32` (see Chapter 4) and become simply the `+0` value for the purpose of that bitwise operation.

Special Values

There are several special values spread across the various types that the *alert* JS developer needs to be aware of, and use properly.

The Non-value Values

For the `undefined` type, there is one and only one value: `undefined`. For the `null` type, there is one and only one value: `null`. So for both of them, the label is both its type and its value.
Both `undefined` and `null` are often taken to be interchangeable as either "empty" values or "non" values. Other developers prefer to distinguish between them with nuance. For example:

- `null` is an empty value
- `undefined` is a missing value

Or:

- `undefined` hasn't had a value yet
- `null` had a value and doesn't anymore

Regardless of how you choose to "define" and use these two values, `null` is a special keyword, not an identifier, and thus you cannot treat it as a variable to assign to (why would you!?). However, `undefined` *is* (unfortunately) an identifier. Uh oh.

Undefined

In non-`strict` mode, it's actually possible (though incredibly ill-advised!) to assign a value to the globally provided `undefined` identifier:

```
function foo() {
	undefined = 2; // really bad idea!
}

foo();

function foo() {
```

```
	"use strict";
	undefined = 2; // TypeError!
}

foo();
```

In both non-`strict` mode and `strict` mode, however, you can create a local variable of the name `undefined`. But again, this is a terrible idea!

```
function foo() {
	"use strict";
	var undefined = 2;
	console.log( undefined ); // 2
}

foo();
```

Friends don't let friends override `undefined`. Ever.

`void` Operator

While `undefined` is a built-in identifier that holds (unless modified -- see above!) the built-in `undefined` value, another way to get this value is the `void` operator.

The expression `void ___` "voids" out any value, so that the result of the expression is always the `undefined` value. It doesn't modify the existing value; it just ensures that no value comes back from the operator expression.

```
var a = 42;

console.log( void a, a ); // undefined 42
```

By convention (mostly from C-language programming), to represent the `undefined` value stand-alone by using `void`, you'd use `void 0` (though clearly even `void true` or any other `void` expression does the same thing). There's no practical difference between `void 0`, `void 1`, and `undefined`.

But the `void` operator can be useful in a few other circumstances, if you need to ensure that an expression has no result value (even if it has side effects).

For example:

```
function doSomething() {
	// note: `APP.ready` is provided by our application
	if (!APP.ready) {
		// try again later
		return void setTimeout( doSomething, 100 );
	}

	var result;

	// do some other stuff
	return result;
}

// were we able to do it right away?
if (doSomething()) {
```

```
        // handle next tasks right away
}
```

Here, the `setTimeout(..)` function returns a numeric value (the unique identifier of the timer interval, if you wanted to cancel it), but we want to `void` that out so that the return value of our function doesn't give a false-positive with the `if` statement.
Many devs prefer to just do these actions separately, which works the same but doesn't use the `void` operator:

```
if (!APP.ready) {
        // try again later
        setTimeout( doSomething, 100 );
        return;
}
```

In general, if there's ever a place where a value exists (from some expression) and you'd find it useful for the value to be `undefined` instead, use the `void` operator. That probably won't be terribly common in your programs, but in the rare cases you do need it, it can be quite helpful.

Special Numbers

The `number` type includes several special values. We'll take a look at each in detail.

The Not Number, Number

Any mathematic operation you perform without both operands being `number`s (or values that can be interpreted as regular `number`s in base 10 or base 16) will result in the operation failing to produce a valid `number`, in which case you will get the `NaN` value.
`NaN` literally stands for "not a `number`", though this label/description is very poor and misleading, as we'll see shortly. It would be much more accurate to think of `NaN` as being "invalid number," "failed number," or even "bad number," than to think of it as "not a number."
For example:

```
var a = 2 / "foo";              // NaN

typeof a === "number";   // true
```

In other words: "the type of not-a-number is 'number'!" Hooray for confusing names and semantics.

`NaN` is a kind of "sentinel value" (an otherwise normal value that's assigned a special meaning) that represents a special kind of error condition within the `number` set. The error condition is, in essence: "I tried to perform a mathematic operation but failed, so here's the failed `number` result instead."

So, if you have a value in some variable and want to test to see if it's this special failed-number `NaN`, you might think you could directly compare to `NaN` itself, as you can with any other value, like `null` or `undefined`. Nope.

```
var a = 2 / "foo";

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

`NaN` is a very special value in that it's never equal to another `NaN` value (i.e., it's never equal to itself). It's the only value, in fact, that is not reflexive (without the Identity characteristic `x === x`). So, `NaN !== NaN`. A bit strange, huh?

So how *do* we test for it, if we can't compare to `NaN` (since that comparison would always fail)?

```
var a = 2 / "foo";

isNaN( a ); // true
```

Easy enough, right? We use the built-in global utility called `isNaN(..)` and it tells us if the value is `NaN` or not. Problem solved!

Not so fast.

The `isNaN(..)` utility has a fatal flaw. It appears it tried to take the meaning of `NaN` ("Not a Number") too literally -- that its job is basically: "test if the thing passed in is either not a `number` or is a `number`." But that's not quite accurate.

```
var a = 2 / "foo";
var b = "foo";

a; // NaN
b; // "foo"

window.isNaN( a ); // true
window.isNaN( b ); // true -- ouch!
```

Clearly, `"foo"` is literally *not a `number`*, but it's definitely not the `NaN` value either! This bug has been in JS since the very beginning (over 19 years of *ouch*).

As of ES6, finally a replacement utility has been provided: `Number.isNaN(..)`. A simple polyfill for it so that you can safely check `NaN` values *now* even in pre-ES6 browsers is:

```
if (!Number.isNaN) {
	Number.isNaN = function(n) {
		return (
			typeof n === "number" &&
			window.isNaN( n )
		);
	};
}

var a = 2 / "foo";
var b = "foo";
```

```
Number.isNaN( a ); // true
Number.isNaN( b ); // false -- phew!
```

Actually, we can implement a `Number.isNaN(..)` polyfill even easier, by taking advantage of that peculiar fact that `NaN` isn't equal to itself. `NaN` is the *only* value in the whole language where that's true; every other value is always **equal to itself**.
So:

```
if (!Number.isNaN) {
	Number.isNaN = function(n) {
		return n !== n;
	};
}
```

Weird, huh? But it works!

`NaN`s are probably a reality in a lot of real-world JS programs, either on purpose or by accident. It's a really good idea to use a reliable test, like `Number.isNaN(..)` as provided (or polyfilled), to recognize them properly.
If you're currently using just `isNaN(..)` in a program, the sad reality is your program *has a bug*, even if you haven't been bitten by it yet!

Infinities

Developers from traditional compiled languages like C are probably used to seeing either a compiler error or runtime exception, like "Divide by zero," for an operation like:

```
var a = 1 / 0;
```

However, in JS, this operation is well-defined and results in the value `Infinity` (aka `Number.POSITIVE_INFINITY`). Unsurprisingly:

```
var a = 1 / 0;   // Infinity
var b = -1 / 0;  // -Infinity
```

As you can see, `-Infinity` (aka `Number.NEGATIVE_INFINITY`) results from a divide-by-zero where either (but not both!) of the divide operands is negative.
JS uses finite numeric representations (IEEE 754 floating-point, which we covered earlier), so contrary to pure mathematics, it seems it *is* possible to overflow even with an operation like addition or subtraction, in which case you'd get `Infinity` or `-Infinity`.
For example:

```
var a = Number.MAX_VALUE;        // 1.7976931348623157e+308
a + a;                                  // Infinity
a + Math.pow( 2, 970 );          // Infinity
a + Math.pow( 2, 969 );          // 1.7976931348623157e+308
```

According to the specification, if an operation like addition results in a value that's too big to represent, the IEEE 754 "round-to-nearest" mode specifies what the result should be. So, in a crude sense, `Number.MAX_VALUE + Math.pow( 2, 969 )` is closer to `Number.MAX_VALUE` than to `Infinity`, so it "rounds down," whereas `Number.MAX_VALUE + Math.pow( 2, 970 )` is closer to `Infinity` so it "rounds up".
If you think too much about that, it's going to make your head hurt. So don't. Seriously, stop!

Once you overflow to either one of the *infinities*, however, there's no going back. In other words, in an almost poetic sense, you can go from finite to infinite but not from infinite back to finite.

It's almost philosophical to ask: "What is infinity divided by infinity". Our naive brains would likely say "1" or maybe "infinity." Turns out neither is true. Both mathematically and in JavaScript, `Infinity / Infinity` is not a defined operation. In JS, this results in `NaN`. But what about any positive finite `number` divided by `Infinity`? That's easy! `0`. And what about a negative finite `number` divided by `Infinity`? Keep reading!

Zeros

While it may confuse the mathematics-minded reader, JavaScript has both a normal zero `0` (otherwise known as a positive zero `+0`) *and* a negative zero `-0`. Before we explain why the `-0` exists, we should examine how JS handles it, because it can be quite confusing.
Besides being specified literally as `-0`, negative zero also results from certain mathematic operations. For example:

```
var a = 0 / -3; // -0
var b = 0 * -3; // -0
```

Addition and subtraction cannot result in a negative zero.

A negative zero when examined in the developer console will usually reveal `-0`, though that was not the common case until fairly recently, so some older browsers you encounter may still report it as `0`.
However, if you try to stringify a negative zero value, it will always be reported as `"0"`, according to the spec.

```
var a = 0 / -3;

// (some browser) consoles at least get it right
a;                                                  // -0

// but the spec insists on lying to you!
a.toString();                          // "0"
a + "";                                         // "0"
```

```
String( a );                             // "0"

// strangely, even JSON gets in on the deception
JSON.stringify( a );            // "0"
```

Interestingly, the reverse operations (going from `string` to `number`) don't lie:

```
+"-0";                          // -0
Number( "-0" );         // -0
JSON.parse( "-0" );     // -0
```

Warning: The `JSON.stringify( -0 )` behavior of `"0"` is particularly strange when you observe that it's inconsistent with the reverse: `JSON.parse( "-0" )` reports `-0` as you'd correctly expect.

In addition to stringification of negative zero being deceptive to hide its true value, the comparison operators are also (intentionally) configured to *lie*.

```
var a = 0;
var b = 0 / -3;

a == b;         // true
-0 == 0;// true

a === b;// true
-0 === 0;       // true

0 > -0;         // false
a > b;          // false
```

Clearly, if you want to distinguish a `-0` from a `0` in your code, you can't just rely on what the developer console outputs, so you're going to have to be a bit more clever:

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

isNegZero( -0 );        // true
isNegZero( 0 / -3 );    // true
isNegZero( 0 );                 // false
```

Now, why do we need a negative zero, besides academic trivia?

There are certain applications where developers use the magnitude of a value to represent one piece of information (like speed of movement per animation frame) and the sign of that `number` to represent another piece of information (like the direction of that movement).

In those applications, as one example, if a variable arrives at zero and it loses its sign, then you would lose the information of what direction it was moving in before it arrived at zero. Preserving the sign of the zero prevents potentially unwanted information loss.

Special Equality

As we saw above, the `NaN` value and the `-0` value have special behavior when it comes to equality comparison. `NaN` is never equal to itself, so you have to use ES6's `Number.isNaN(..)` (or a polyfill). Similarly, `-0` lies and pretends that it's equal (even `===` strict equal -- see Chapter 4) to regular positive `0`, so you have to use the somewhat hackish `isNegZero(..)` utility we suggested above.
As of ES6, there's a new utility that can be used to test two values for absolute equality, without any of these exceptions. It's called `Object.is(..)`:

```
var a = 2 / "foo";
var b = -3 * 0;

Object.is( a, NaN );	// true
Object.is( b, -0 );		// true

Object.is( b, 0 );		// false
```

There's a pretty simple polyfill for `Object.is(..)` for pre-ES6 environments:

```
if (!Object.is) {
	Object.is = function(v1, v2) {
		// test for `-0`
		if (v1 === 0 && v2 === 0) {
			return 1 / v1 === 1 / v2;
		}
		// test for `NaN`
		if (v1 !== v1) {
			return v2 !== v2;
		}
		// everything else
		return v1 === v2;
	};
}
```

`Object.is(..)` probably shouldn't be used in cases where `==` or `===` are known to be *safe* (see Chapter 4 "Coercion"), as the operators are likely much more efficient and certainly are more idiomatic/common. `Object.is(..)` is mostly for these special cases of equality.

Value vs. Reference

In many other languages, values can either be assigned/passed by value-copy or by reference-copy depending on the syntax you use.

For example, in C++ if you want to pass a `number` variable into a function and have that variable's value updated, you can declare the function parameter like `int& myNum`, and when you pass in a variable like `x`, `myNum` will be a **reference to `x`**; references are like a special form of pointers, where you obtain a pointer to another variable (like an *alias*). If

you don't declare a reference parameter, the value passed in will *always* be copied, even if it's a complex object.
In JavaScript, there are no pointers, and references work a bit differently. You cannot have a reference from one JS variable to another variable. That's just not possible.

A reference in JS points at a (shared) **value**, so if you have 10 different references, they are all always distinct references to a single shared value; **none of them are references/pointers to each other.**

Moreover, in JavaScript, there are no syntactic hints that control value vs. reference assignment/passing. Instead, the *type* of the value *solely* controls whether that value will be assigned by value-copy or by reference-copy.

Let's illustrate:

```
var a = 2;
var b = a; // `b` is always a copy of the value in `a`
b++;
a; // 2
b; // 3

var c = [1,2,3];
var d = c; // `d` is a reference to the shared `[1,2,3]` value
d.push( 4 );
c; // [1,2,3,4]
d; // [1,2,3,4]
```

Simple values (aka scalar primitives) are *always* assigned/passed by value-copy: `null`, `undefined`, `string`, `number`, `boolean`, and ES6's `symbol`.
Compound values -- `object`s (including `array`s, and all boxed object wrappers -- see Chapter 3) and `function`s -- *always* create a copy of the reference on assignment or passing.
In the above snippet, because `2` is a scalar primitive, `a` holds one initial copy of that value, and `b` is assigned another *copy* of the value. When changing `b`, you are in no way changing the value in `a`.
But **both `c` and `d`** are separate references to the same shared value `[1,2,3]`, which is a compound value. It's important to note that neither `c` nor `d` more "owns" the `[1,2,3]` value -- both are just equal peer references to the value. So, when using either reference to modify (`.push(4)`) the actual shared `array` value itself, it's affecting just the one shared value, and both references will reference the newly modified value `[1,2,3,4]`.
Since references point to the values themselves and not to the variables, you cannot use one reference to change where another reference is pointed:

```
var a = [1,2,3];
```

```
var b = a;
a; // [1,2,3]
b; // [1,2,3]

// later
b = [4,5,6];
a; // [1,2,3]
b; // [4,5,6]
```

When we make the assignment `b = [4,5,6]`, we are doing absolutely nothing to affect *where* `a` is still referencing (`[1,2,3]`). To do that, `b` would have to be a pointer to `a` rather than a reference to the `array` -- but no such capability exists in JS!
The most common way such confusion happens is with function parameters:

```
function foo(x) {
	x.push( 4 );
	x; // [1,2,3,4]

	// later
	x = [4,5,6];
	x.push( 7 );
	x; // [4,5,6,7]
}

var a = [1,2,3];

foo( a );

a; // [1,2,3,4]  not  [4,5,6,7]
```

When we pass in the argument `a`, it assigns a copy of the `a` reference to `x`. `x` and `a` are separate references pointing at the same `[1,2,3]` value. Now, inside the function, we can use that reference to mutate the value itself (`push(4)`). But when we make the assignment `x = [4,5,6]`, this is in no way affecting where the initial reference `a` is pointing -- still points at the (now modified) `[1,2,3,4]` value.
There is no way to use the `x` reference to change where `a` is pointing. We could only modify the contents of the shared value that both `a` and `x` are pointing to.
To accomplish changing `a` to have the `[4,5,6,7]` value contents, you can't create a new `array` and assign -- you must modify the existing `array` value:

```
function foo(x) {
	x.push( 4 );
	x; // [1,2,3,4]

	// later
	x.length = 0; // empty existing array in-place
	x.push( 4, 5, 6, 7 );
	x; // [4,5,6,7]
}

var a = [1,2,3];
```

```
foo( a );

a; // [4,5,6,7]  not  [1,2,3,4]
```

As you can see, `x.length = 0` and `x.push(4,5,6,7)` were not creating a new `array`, but modifying the existing shared `array`. So of course, `a` references the new `[4,5,6,7]` contents.

Remember: you cannot directly control/override value-copy vs. reference -- those semantics are controlled entirely by the type of the underlying value.

To effectively pass a compound value (like an `array`) by value-copy, you need to manually make a copy of it, so that the reference passed doesn't still point to the original. For example:

```
foo( a.slice() );
```

`slice(..)` with no parameters by default makes an entirely new (shallow) copy of the `array`. So, we pass in a reference only to the copied `array`, and thus `foo(..)` cannot affect the contents of `a`.

To do the reverse -- pass a scalar primitive value in a way where its value updates can be seen, kinda like a reference -- you have to wrap the value in another compound value (`object`, `array`, etc) that *can* be passed by reference-copy:

```
function foo(wrapper) {
        wrapper.a = 42;
}

var obj = {
        a: 2
};

foo( obj );

obj.a; // 42
```

Here, `obj` acts as a wrapper for the scalar primitive property `a`. When passed to `foo(..)`, a copy of the `obj` reference is passed in and set to the `wrapper` parameter. We now can use the `wrapper` reference to access the shared object, and update its property. After the function finishes, `obj.a` will see the updated value `42`.

It may occur to you that if you wanted to pass in a reference to a scalar primitive value like `2`, you could just box the value in its `Number` object wrapper (see Chapter 3).

It *is* true a copy of the reference to this `Number` object *will* be passed to the function, but unfortunately, having a reference to the shared object is not going to give you the ability to modify the shared primitive value, like you may expect:

```
function foo(x) {
        x = x + 1;
        x; // 3
}
```

```
var a = 2;
var b = new Number( a ); // or equivalently `Object(a)`

foo( b );
console.log( b ); // 2, not 3
```

The problem is that the underlying scalar primitive value is *not mutable* (same goes for `String` and `Boolean`). If a `Number` object holds the scalar primitive value `2`, that exact `Number` object can never be changed to hold another value; you can only create a whole new `Number` object with a different value.
When `x` is used in the expression `x + 1`, the underlying scalar primitive value `2` is unboxed (extracted) from the `Number` object automatically, so the line `x = x + 1` very subtly changes `x` from being a shared reference to the `Number` object, to just holding the scalar primitive value `3` as a result of the addition operation `2 + 1`. Therefore, `b` on the outside still references the original unmodified/immutable `Number` object holding the value `2`.
You *can* add properties on top of the `Number` object (just not change its inner primitive value), so you could exchange information indirectly via those additional properties. This is not all that common, however; it probably would not be considered a good practice by most developers.

Instead of using the wrapper object `Number` in this way, it's probably much better to use the manual object wrapper (`obj`) approach in the earlier snippet. That's not to say that there's no clever uses for the boxed object wrappers like `Number` -- just that you should probably prefer the scalar primitive value form in most cases.
References are quite powerful, but sometimes they get in your way, and sometimes you need them where they don't exist. The only control you have over reference vs. value-copy behavior is the type of the value itself, so you must indirectly influence the assignment/passing behavior by which value types you choose to use.

Review

In JavaScript, `array`s are simply numerically indexed collections of any value-type. `string`s are somewhat "`array`-like", but they have distinct behaviors and care must be taken if you want to treat them as `array`s. Numbers in JavaScript include both "integers" and floating-point values.
Several special values are defined within the primitive types.

The `null` type has just one value: `null`, and likewise the `undefined` type has just the `undefined` value. `undefined` is basically the default value in any variable or property if no other value is present. The `void` operator lets you create the `undefined` value from any other value.

`number`s include several special values, like `NaN` (supposedly "Not a Number", but really more appropriately "invalid number"); `+Infinity` and `-Infinity`; and `-0`.
Simple scalar primitives (`string`s, `number`s, etc.) are assigned/passed by value-copy, but compound values (`object`s, etc.) are assigned/passed by reference-copy. References are not like references/pointers in other languages -- they're never pointed at other variables/references, only at the underlying values.