MDN Web Docs
moz://a

A re-introduction to JavaScript (JS tutorial)

Why a re-introduction? Because JavaScript is notorious for being misunderstood. It is often derided as being a toy, but beneath its layer of deceptive simplicity, powerful language features await. JavaScript is now used by an incredible number of high-profile applications, showing that deeper knowledge of this technology is an important skill for any web or mobile developer.

It's useful to start with an overview of the language's history. JavaScript was created in 1995 by Brendan Eich while he was an engineer at Netscape. JavaScript was first released with Netscape 2 early in 1996. It was originally going to be called LiveScript, but it was renamed in an ill-fated marketing decision that attempted to capitalize on the popularity of Sun Microsystem's Java language — despite the two having very little in common. This has been a source of confusion ever since.

Several months later, Microsoft released JScript with Internet Explorer 3. It was a mostly-compatible JavaScript work-alike. Several months after that, Netscape submitted JavaScript to Ecma International , a European standards organization, which resulted in the first edition of the ECMAScript standard that year. The standard received a significant update as ECMAScript edition 3 in 1999, and it has stayed pretty much stable ever since. The fourth edition was abandoned, due to political differences concerning language complexity. Many parts of the fourth edition formed the basis for ECMAScript edition 5, published in

December of 2009, and for the 6th major edition of the standard, published in June of 2015.

> **Note:** Because it is more familiar, we will refer to ECMAScript as "JavaScript" from this point on.

Unlike most programming languages, the JavaScript language has no concept of input or output. It is designed to run as a scripting language in a host environment, and it is up to the host environment to provide mechanisms for communicating with the outside world. The most common host environment is the browser, but JavaScript interpreters can also be found in a huge list of other places, including Adobe Acrobat, Adobe Photoshop, SVG images, Yahoo's Widget engine, server-side environments such as Node.js , NoSQL databases like the open source Apache CouchDB , embedded computers, complete desktop environments like GNOME (one of the most popular GUIs for GNU/Linux operating systems), and others.

Overview

JavaScript is a multi-paradigm, dynamic language with types and operators, standard built-in objects, and methods. Its syntax is based on the Java and C languages — many structures from those languages apply to JavaScript as well. JavaScript supports object-oriented programming with object prototypes, instead of classes (see more about prototypical inheritance and ES2015 classes). JavaScript also supports functional programming — because they are objects, functions may be stored in variables and passed around like any other object.

Let's start off by looking at the building blocks of any language: the types. JavaScript programs manipulate values, and those values all belong to a type. JavaScript's types are:

- Number
- BigInt
- String

- Boolean
- `Function`
- Object
- Symbol (new in ES2015)

... oh, and undefined and null, which are ... slightly odd. And `Array`, which is a special kind of object. And `Date` and `RegExp`, which are objects that you get for free. And to be technically accurate, functions are just a special type of object. So the type diagram looks more like this:

- Number
- BigInt
- String
- Boolean
- Symbol (new in ES2015)
- Object
    - `Function`
    - `Array`
    - `Date`

    - `RegExp`
- null
- undefined

And there are some built-in `Error` types as well. Things are a lot easier if we stick with the first diagram, however, so we'll discuss the types listed there for now.

Numbers

ECMAScript has two built-in numeric types: **Number** and **BigInt**.

The Number type is a double-precision 64-bit binary format IEEE 754 value (numbers between -(2^53 − 1) and 2^53 − 1). And where this article and other MDN articles refer to “integers”, what’s usually meant is a *representation* of an integer using a Number value. But because such Number values aren’t real integers, you have to be a little careful. For example:

```
console.log(3 / 2);              // 1.5, not 1
console.log(Math.floor(3 / 2)); // 1
```

So an *apparent integer* is in fact *implicitly a float*.

Also, watch out for stuff like:

```
0.1 + 0.2 == 0.30000000000000004;
```

In practice, integer values are treated as 32-bit ints, and some implementations even store it that way until they are asked to perform an instruction that's valid on a Number but not on a 32-bit integer. This can be important for bit-wise operations.

The standard arithmetic operators are supported, including addition, subtraction, modulus (or remainder) arithmetic, and so forth. There's also a built-in object that we did not mention earlier called `Math` that provides advanced mathematical functions and constants:

```
Math.sin(3.5);
var circumference = 2 * Math.PI * r;
```

You can convert a string to an integer using the built-in `parseInt()` function. This takes the base for the conversion as an optional second argument, which you should always provide:

```
parseInt('123', 10); // 123
parseInt('010', 10); // 10
```

In older browsers, strings beginning with a "0" are assumed to be in octal (radix 8), but this hasn't been the case since 2013 or so. Unless you're certain of your string format, you can get surprising results on those older browsers:

```
parseInt('010');  //  8
parseInt('0x10'); // 16
```

Here, we see the `parseInt()` function treat the first string as octal due to the leading 0, and the second string as hexadecimal due to the leading "0x". The *hexadecimal notation is still in place*; only octal has been

removed.

If you want to convert a binary number to an integer, just change the base:

```
parseInt('11', 2); // 3
```

Similarly, you can parse floating point numbers using the built-in `parseFloat()` function. Unlike its `parseInt()` cousin, `parseFloat()` always uses base 10.

You can also use the unary `+` operator to convert values to numbers:

```
+ '42';   // 42
+ '010';  // 10
+ '0x10'; // 16
```

A special value called `NaN` (short for "Not a Number") is returned if the string is non-numeric:

```
parseInt('hello', 10); // NaN
```

`NaN` is toxic: if you provide it as an operand to any mathematical operation, the result will also be `NaN`:

```
NaN + 5; // NaN
```

You can reliably test for `NaN` using the built-in `Number.isNaN()` function, which behaves just as its name implies:

```
Number.isNaN(NaN); // true
Number.isNaN('hello'); // false
Number.isNaN('1'); // false
```

```
Number.isNaN( 1 ); // false
Number.isNaN(undefined); // false
Number.isNaN({}); // false
Number.isNaN([1]) // false
Number.isNaN([1,2]) // false
```

But don't test for `NaN` using the global `isNaN()` function, which has unintuitive behavior:

```
isNaN('hello'); // true
isNaN('1'); // false
isNaN(undefined); // true
isNaN({}); // true
isNaN([1]) // false
isNaN([1,2]) // true
```

JavaScript also has the special values `Infinity` and `-Infinity`:

```
 1 / 0; //  Infinity
-1 / 0; // -Infinity
```

You can test for `Infinity`, `-Infinity` and `NaN` values using the built-in `isFinite()` function:

```
isFinite(1 / 0); // false
isFinite(-Infinity); // false
isFinite(NaN); // false
```

> **Note:** The `parseInt()` and `parseFloat()` functions parse a string until they reach a character that isn't valid for the specified number format, then return the number parsed up to that point. However the "+" operator converts the string to `NaN` if there is an invalid character contained within it. Just try parsing the

> operator converts the string to `NaN` if there is an invalid character contained within it. Just try parsing the string "10.2abc" with each method by yourself in the console and you'll understand the differences better.

Strings

Strings in JavaScript are sequences of Unicode characters. This should be welcome news to anyone who has had to deal with internationalization. More accurately, they are sequences of UTF-16 code units; each code unit is represented by a 16-bit number. Each Unicode character is represented by either 1 or 2 code units.

If you want to represent a single character, you just use a string consisting of that single character.

To find the length of a string (in code units), access its `length` property:

```
'hello'.length; // 5
```

There's our first brush with JavaScript objects! Did we mention that you can use strings like objects too? They have methods as well that allow you to manipulate the string and access information about the string:

```
'hello'.charAt(0); // "h"
'hello, world'.replace('world', 'mars'); // "hello, mars"
'hello'.toUpperCase(); // "HELLO"
```

Other types

JavaScript distinguishes between `null`, which is a value that indicates a deliberate non-value (and is only accessible through the `null` keyword), and `undefined`, which is a value of type `undefined` that

accessible through the `null` keyword), and `undefined`, which is a value of type `undefined` that indicates an uninitialized variable — that is, a value hasn't even been assigned yet. We'll talk about variables later, but in JavaScript it is possible to declare a variable without assigning a value to it. If you do this, the variable's type is `undefined`. `undefined` is actually a constant.

JavaScript has a boolean type, with possible values `true` and `false` (both of which are keywords.) Any value can be converted to a boolean according to the following rules:

1. `false`, `0`, empty strings (`""`), `NaN`, `null`, and `undefined` all become `false`.
2. All other values become `true`.

You can perform this conversion explicitly using the `Boolean()` function:

```
Boolean('');  // false
Boolean(234); // true
```

However, this is rarely necessary, as JavaScript will silently perform this conversion when it expects a boolean, such as in an `if` statement (see below). For this reason, we sometimes speak of "true values" and "false values," meaning values that become `true` and `false`, respectively, when converted to booleans. Alternatively, such values can be called "truthy" and "falsy", respectively.

Boolean operations such as `&&` (logical *and*), `||` (logical *or*), and `!` (logical *not*) are supported; see below.

Variables

New variables in JavaScript are declared using one of three keywords: `let`, `const`, or `var`.

`let` allows you to declare block-level variables. The declared variable is available from the *block* it is enclosed in.

```
let a;
let name = 'Simon';
```

The following is an example of scope with a variable declared with `let`:

```
// myLetVariable is *not* visible out here

for (let myLetVariable = 0; myLetVariable < 5; myLetVariable++) {
  // myLetVariable is only visible in here
}

// myLetVariable is *not* visible out here
```

`const` allows you to declare variables whose values are never intended to change. The variable is available from the *block* it is declared in.

```
const Pi = 3.14; // variable Pi is set
Pi = 1; // will throw an error because you cannot change a constant variable.
```

`var` is the most common declarative keyword. It does not have the restrictions that the other two keywords have. This is because it was traditionally the only way to declare a variable in JavaScript. A variable declared with the `var` keyword is available from the *function* it is declared in.

```
var a;
var name = 'Simon';
```

An example of scope with a variable declared with **`var`:**

```
// myVarVariable *is* visible out here

for (var myVarVariable = 0; myVarVariable < 5; myVarVariable++) {
  // myVarVariable is visible to the whole function
}

// myVarVariable *is* visible out here
```

If you declare a variable without assigning any value to it, its type is `undefined`.

An important difference between JavaScript and other languages like Java is that in JavaScript, blocks do not have scope; only functions have a scope. So if a variable is defined using `var` in a compound statement (for example inside an `if` control structure), it will be visible to the entire function. However, starting with ECMAScript 2015, `let` and `const` declarations allow you to create block-scoped variables.

Operators

JavaScript's numeric operators are `+`, `-`, `*`, `/` and `%` which is the remainder operator (which is the same as modulo.) Values are assigned using `=`, and there are also compound assignment statements such as `+=` and `-=`. These extend out to `x = x operator y`.

```
x += 5;
x = x + 5;
```

You can use `++` and `--` to increment and decrement respectively. These can be used as a prefix or postfix operators.

operators.

The + operator also does string concatenation:

```
'hello' + ' world'; // "hello world"
```

If you add a string to a number (or other value) everything is converted into a string first. This might trip you up:

```
'3' + 4 + 5;  // "345"
 3 + 4 + '5'; // "75"
```

Adding an empty string to something is a useful way of converting it to a string itself.

Comparisons in JavaScript can be made using `<`, `>`, `<=` and `>=`. These work for both strings and numbers. Equality is a little less straightforward. The double-equals operator performs type coercion if you give it different types, with sometimes interesting results:

```
123 == '123'; // true
1 == true; // true
```

To avoid type coercion, use the triple-equals operator:

```
123 === '123'; // false
1 === true;    // false
```

There are also `!=` and `!==` operators.

JavaScript also has bitwise operations. If you want to use them, they're there.

Control structures

JavaScript has a similar set of control structures to other languages in the C family. Conditional statements are supported by `if` and `else`; you can chain them together if you like:

```
var name = 'kittens';
if (name === 'puppies') {
  name += ' woof';
} else if (name === 'kittens') {
  name += ' meow';
} else {
  name += '!';
}
name === 'kittens meow';
```

JavaScript has `while` loops and `do-while` loops. The first is good for basic looping; the second for loops where you wish to ensure that the body of the loop is executed at least once:

```
while (true) {
  // an infinite loop!
}

var input;
do {
  input = get_input();
} while (inputIsNotValid(input));
```

JavaScript's `for` loop is the same as that in C and Java: it lets you provide the control information for your loop on a single line.

```
for (var i = 0; i < 5; i++) {
  // Will execute 5 times
}
```

JavaScript also contains two other prominent for loops: `for ... of`

```
for (let value of array) {
  // do something with value
}
```

and `for ... in`:

```
for (let property in object) {
  // do something with object property
}
```

The `&&` and `||` operators use short-circuit logic, which means whether they will execute their second operand is dependent on the first. This is useful for checking for null objects before accessing their attributes:

```
var name = o && o.getName();
```

Or for caching values (when falsy values are invalid):

```
var name = cachedName || (cachedName = getName());
```

```
var name = cachedName || (cachedName = getName());
```

JavaScript has a ternary operator for conditional expressions:

```
var allowed = (age > 18) ? 'yes' : 'no';
```

The `switch` statement can be used for multiple branches based on a number or string:

```
switch (action) {
  case 'draw':
    drawIt();
    break;
  case 'eat':
    eatIt();
    break;
  default:
    doNothing();
}
```

If you don't add a `break` statement, execution will "fall through" to the next level. This is very rarely what you want — in fact it's worth specifically labeling deliberate fallthrough with a comment if you really meant it to aid debugging:

```
switch (a) {
  case 1: // fallthrough
  case 2:
    eatIt();
    break;
  default:
    doNothing();
```

```
}
```

The default clause is optional. You can have expressions in both the switch part and the cases if you like; comparisons take place between the two using the `===` operator:

```
switch (1 + 3) {
  case 2 + 2:
    yay();
    break;
  default:
    neverhappens();
}
```

Objects

JavaScript objects can be thought of as simple collections of name-value pairs. As such, they are similar to:

- Dictionaries in Python.
- Hashes in Perl and Ruby.
- Hash tables in C and C++.
- HashMaps in Java.
- Associative arrays in PHP.

The fact that this data structure is so widely used is a testament to its versatility. Since everything (bar core types) in JavaScript is an object, any JavaScript program naturally involves a great deal of hash table lookups. It's a good thing they're so fast!

The "name" part is a JavaScript string, while the value can be any JavaScript value — including more objects. This allows you to build data structures of arbitrary complexity.

There are two basic ways to create an empty object:

```
var obj = new Object();
```

And:

```
var obj = {};
```

These are semantically equivalent; the second is called object literal syntax and is more convenient. This syntax is also the core of JSON format and should be preferred at all times.

Object literal syntax can be used to initialize an object in its entirety:

```
var obj = {
  name: 'Carrot',
  _for: 'Max', // 'for' is a reserved word, use '_for' instead.
  details: {
    color: 'orange',
    size: 12
  }
};
```

Attribute access can be chained together:

```
obj.details.color; // orange
```

```
obj['details']['size']; // 12
```

The following example creates an object prototype(`Person`) and an instance of that prototype(`you`).

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}

// Define an object
var you = new Person('You', 24);
// We are creating a new person named "You" aged 24.
```

Once created, an object's properties can again be accessed in one of two ways:

```
// dot notation
obj.name = 'Simon';
var name = obj.name;
```

And...

```
// bracket notation
obj['name'] = 'Simon';
var name = obj['name'];
// can use a variable to define a key
var user = prompt('what is your key?')
obj[user] = prompt('what is its value?')
```

These are also semantically equivalent. The second method has the advantage that the name of the property is provided as a string, which means it can be calculated at run-time. However, using this method

prevents some JavaScript engine and minifier optimizations being applied. It can also be used to set and get properties with names that are reserved words:

```
obj.for = 'Simon'; // Syntax error, because 'for' is a reserved word
obj['for'] = 'Simon'; // works fine
```

> **Note:** Starting in ECMAScript 5, reserved words may be used as object property names "in the buff". This means that they don't need to be "clothed" in quotes when defining object literals. See the ES5 Spec .

For more on objects and prototypes see Object.prototype. For an explanation of object prototypes and the object prototype chains see Inheritance and the prototype chain.

> **Note:** Starting in ECMAScript 2015, object keys can be defined by the variable using bracket notation upon being created. `{[phoneType]: 12345}` is possible instead of just `var userPhone = {}; userPhone[phoneType] = 12345`.

Arrays

Arrays in JavaScript are actually a special type of object. They work very much like regular objects (numerical properties can naturally be accessed only using `[]` syntax) but they have one magic property called '`length`'. This is always one more than the highest index in the array.

One way of creating arrays is as follows:

```
var a = new Array();
```

```
a[0] = 'dog';
a[1] = 'cat';
a[2] = 'hen';
a.length; // 3
```

A more convenient notation is to use an array literal:

```
var a = ['dog', 'cat', 'hen'];
a.length; // 3
```

Note that `array.length` isn't necessarily the number of items in the array. Consider the following:

```
var a = ['dog', 'cat', 'hen'];
a[100] = 'fox';
a.length; // 101
```

Remember — the length of the array is one more than the highest index.

If you query a non-existent array index, you'll get a value of `undefined` in return:

```
typeof a[90]; // undefined
```

If you take the above about `[]` and `length` into account, you can iterate over an array using the following `for` loop:

```
for (var i = 0; i < a.length; i++) {
  // Do something with a[i]
}
```

ES2015 introduced the more concise `for ... of` loop for iterable objects such as arrays:

```
for (const currentValue of a) {
  // Do something with currentValue
}
```

You could also iterate over an array using a `for ... in` loop, however this does not iterate over the array elements, but the array indices. Furthermore, if someone added new properties to `Array.prototype`, they would also be iterated over by such a loop. Therefore this loop type is not recommended for arrays.

Another way of iterating over an array that was added with ECMAScript 5 is `forEach()`:

```
['dog', 'cat', 'hen'].forEach(function(currentValue, index, array) {
  // Do something with currentValue or array[index]
});
```

If you want to append an item to an array do it like this:

```
a.push(item);
```

Arrays come with a number of methods. See also the full documentation for array methods.

Method name	Description
`a.toString()`	Returns a string with the `toString()` of each element separated by commas.

Method name	Description
`a.toLocaleString()`	Returns a string with the `toLocaleString()` of each element separated by commas.
`a.concat(item1[, item2[, ...[, itemN]]])`	Returns a new array with the items added on to it.

`a.join(sep)`	Converts the array to a string — with values delimited by the `sep` param
`a.pop()`	Removes and returns the last item.
`a.push(item1, ..., itemN)`	Appends items to the end of the array.
`a.shift()`	Removes and returns the first item.
`a.unshift(item1[, item2[, ...[, itemN]]])`	Prepends items to the start of the array.
`a.slice(start[, end])`	Returns a sub-array.
`a.sort([cmpfn])`	Takes an optional comparison function.
`a.splice(start, delcount[, item1[, ...[, itemN]]])`	Lets you modify an array by deleting a section and replacing it with more items.
`a.reverse()`	Reverses the array.

Method name	Description

Functions

Along with objects, functions are the core component in understanding JavaScript. The most basic function couldn't be much simpler:

```
function add(x, y) {
  var total = x + y;
  return total;
}
```

This demonstrates a basic function. A JavaScript function can take 0 or more named parameters. The function body can contain as many statements as you like and can declare its own variables which are local to that function. The `return` statement can be used to return a value at any time, terminating the function. If no return statement is used (or an empty return with no value), JavaScript returns `undefined`.

The named parameters turn out to be more like guidelines than anything else. You can call a function without passing the parameters it expects, in which case they will be set to `undefined`.

```
add(); // NaN
// You can't perform addition on undefined
```

You can also pass in more arguments than the function is expecting:

```
add(2, 3, 4); // 5
```

```
// added the first two; 4 was ignored
```

That may seem a little silly, but functions have access to an additional variable inside their body called `arguments`, which is an array-like object holding all of the values passed to the function. Let's re-write the add function to take as many values as we want:

```
function add() {
  var sum = 0;
  for (var i = 0, j = arguments.length; i < j; i++) {
    sum += arguments[i];
  }
  return sum;
}

add(2, 3, 4, 5); // 14
```

That's really not any more useful than writing `2 + 3 + 4 + 5` though. Let's create an averaging function:

```
function avg() {
  var sum = 0;
  for (var i = 0, j = arguments.length; i < j; i++) {
    sum += arguments[i];
  }
  return sum / arguments.length;
}

avg(2, 3, 4, 5); // 3.5
```

This is pretty useful, but it does seem a little verbose. To reduce this code a bit more we can look at substituting the use of the arguments array through Rest parameter syntax. In this way, we can pass in any

number of arguments into the function while keeping our code minimal. The **rest parameter operator** is used in function parameter lists with the format: **...variable** and it will include within that variable the entire list of uncaptured arguments that the function was called with. We will also replace the **for** loop with a **for...of** loop to return the values within our variable.

```
function avg(...args) {
  var sum = 0;
  for (let value of args) {
    sum += value;
  }
  return sum / args.length;
}

avg(2, 3, 4, 5); // 3.5
```

In the above code, the variable **args** holds all the values that were passed into the function.

It is important to note that wherever the rest parameter operator is placed in a function declaration it will store all arguments *after* its declaration, but not before. i.e. *function avg(**firstValue,** ...args)* will store the first value passed into the function in the **firstValue** variable and the remaining arguments in **args**. That's another useful language feature but it does lead us to a new problem. The `avg()` function takes a comma-separated list of arguments — but what if you want to find the average of an array? You could just rewrite the function as follows:

```
function avgArray(arr) {
  var sum = 0;
  for (var i = 0, j = arr.length; i < j; i++) {
    sum += arr[i];
```

```
  }
  return sum / arr.length;
}

avgArray([2, 3, 4, 5]); // 3.5
```

But it would be nice to be able to reuse the function that we've already created. Luckily, JavaScript lets you call a function with an arbitrary array of arguments, using the `apply()` method of any function object.

```
avg.apply(null, [2, 3, 4, 5]); // 3.5
```

The second argument to `apply()` is the array to use as arguments; the first will be discussed later on. This emphasizes the fact that functions are objects too.

You can achieve the same result using the spread operator in the function call.

For instance: `avg(...numbers)`

Anonymous functions

JavaScript lets you create anonymous functions — that is, functions without names:

```
function() {
  var sum = 0;
  for (var i = 0, j = arguments.length; i < j; i++) {
    sum += arguments[i];
  }
  return sum / arguments.length;
};
```

```
};
```

But such an anonymous function isn't useful in isolation — because without a name, there's no way to call the function. So in practice, anonymous functions are typically used as arguments to other functions or are made callable by immediately assigning them to a variable that can be used to invoke the function:

```
var avg = function() {
  var sum = 0;
  for (var i = 0, j = arguments.length; i < j; i++) {
    sum += arguments[i];
  }
  return sum / arguments.length;
};
```

That makes the anonymous function invocable by calling `avg()` with some arguments — that is, it's semantically equivalent to declaring the function using the `function avg()` named-function form.

But there's a way that anonymous functions can be useful even without ever being assigned to variables or passed as arguments to other functions: JavaScript provides a mechanism for simultaneously declaring and invoking a function using a single expression. It's called an Immediately invoked function expression (IIFE), and the syntax for using it with an anonymous function looks like this:

```
(function() {
  // …
})();
```

Further details on IIFEs are out of scope for this introductory article — but a good example of what they're particularly useful for is in the Emulating private methods with closures section of the Closures article.

Recursive functions

JavaScript allows you to call functions recursively. This is particularly useful for dealing with tree structures, such as those found in the browser DOM.

```
function countChars(elm) {
  if (elm.nodeType == 3) { // TEXT_NODE
    return elm.nodeValue.length;
  }
  var count = 0;
  for (var i = 0, child; child = elm.childNodes[i]; i++) {
    count += countChars(child);
  }
  return count;
}
```

This highlights a potential problem with anonymous functions: how do you call them recursively if they don't have a name? JavaScript lets you name function expressions for this. You can use named IIFEs (Immediately Invoked Function Expressions) as shown below:

```
var charsInBody = (function counter(elm) {
  if (elm.nodeType == 3) { // TEXT_NODE
    return elm.nodeValue.length;
  }
  var count = 0;
  for (var i = 0, child; child = elm.childNodes[i]; i++) {
    count += counter(child);
  }
  return count;
```

```
})(document.body);
```

The name provided to a function expression as above is only available to the function's own scope. This allows more optimizations to be done by the engine and results in more readable code. The name also shows up in the debugger and some stack traces, which can save you time when debugging.

Note that JavaScript functions are themselves objects — like everything else in JavaScript — and you can add or change properties on them just like we've seen earlier in the Objects section.

Custom objects

> **Note:** For a more detailed discussion of object-oriented programming in JavaScript, see Introduction to Object-Oriented JavaScript.

In classic Object Oriented Programming, objects are collections of data and methods that operate on that data. JavaScript is a prototype-based language that contains no class statement, as you'd find in C++ or Java (this is sometimes confusing for programmers accustomed to languages with a class statement). Instead, JavaScript uses functions as classes. Let's consider a person object with first and last name fields. There are two ways in which the name might be displayed: as "first last" or as "last, first". Using the functions and objects that we've discussed previously, we could display the data like this:

```
function makePerson(first, last) {
  return {
    first: first,
    last: last
  };
}
```

```
function personFullName(person) {
  return person.first + ' ' + person.last;
}
function personFullNameReversed(person) {
  return person.last + ', ' + person.first;
}

var s = makePerson('Simon', 'Willison');
personFullName(s); // "Simon Willison"
personFullNameReversed(s); // "Willison, Simon"
```

This works, but it's pretty ugly. You end up with dozens of functions in your global namespace. What we really need is a way to attach a function to an object. Since functions are objects, this is easy:

```
function makePerson(first, last) {
  return {
    first: first,
    last: last,
    fullName: function() {
      return this.first + ' ' + this.last;
    },
    fullNameReversed: function() {
      return this.last + ', ' + this.first;
    }
  };
}

var s = makePerson('Simon', 'Willison');
s.fullName(); // "Simon Willison"
s.fullNameReversed(); // "Willison, Simon"
```

Note on the `this` keyword. Used inside a function, `this` refers to the current object. What that actually means is specified by the way in which you called that function. If you called it using dot notation or bracket notation on an object, that object becomes `this`. If dot notation wasn't used for the call, `this` refers to the global object.

Note that `this` is a frequent cause of mistakes. For example:

```
var s = makePerson('Simon', 'Willison');
var fullName = s.fullName;
fullName(); // undefined undefined
```

When we call `fullName()` alone, without using `s.fullName()`, `this` is bound to the global object. Since there are no global variables called `first` or `last` we get `undefined` for each one.

We can take advantage of the `this` keyword to improve our `makePerson` function:

```
function Person(first, last) {
  this.first = first;
  this.last = last;
  this.fullName = function() {
    return this.first + ' ' + this.last;
  };
  this.fullNameReversed = function() {
    return this.last + ', ' + this.first;
  };
}
var s = new Person('Simon', 'Willison');
```

We have introduced another keyword: `new`. `new` is strongly related to `this`. It creates a brand new empty

We have introduced another keyword: `new`. `new` is strongly related to `this`. It creates a brand new empty object, and then calls the function specified, with `this` set to that new object. Notice though that the function specified with `this` does not return a value but merely modifies the `this` object. It's `new` that returns the `this` object to the calling site. Functions that are designed to be called by `new` are called constructor functions. Common practice is to capitalize these functions as a reminder to call them with `new`.

The improved function still has the same pitfall with calling `fullName()` alone.

Our person objects are getting better, but there are still some ugly edges to them. Every time we create a person object we are creating two brand new function objects within it — wouldn't it be better if this code was shared?

```
function personFullName() {
  return this.first + ' ' + this.last;
}
function personFullNameReversed() {
  return this.last + ', ' + this.first;
}
function Person(first, last) {
  this.first = first;
  this.last = last;
  this.fullName = personFullName;
  this.fullNameReversed = personFullNameReversed;
}
```

That's better: we are creating the method functions only once, and assigning references to them inside the constructor. Can we do any better than that? The answer is yes:

```
function Person(first, last) {
```

```
  this.first = first;
  this.last = last;
}
Person.prototype.fullName = function() {
  return this.first + ' ' + this.last;
};
Person.prototype.fullNameReversed = function() {
  return this.last + ', ' + this.first;
};
```

`Person.prototype` is an object shared by all instances of `Person`. It forms part of a lookup chain (that has a special name, "prototype chain"): any time you attempt to access a property of `Person` that isn't set, JavaScript will check `Person.prototype` to see if that property exists there instead. As a result, anything assigned to `Person.prototype` becomes available to all instances of that constructor via the `this` object.

This is an incredibly powerful tool. JavaScript lets you modify something's prototype at any time in your program, which means you can add extra methods to existing objects at runtime:

```
var s = new Person('Simon', 'Willison');
s.firstNameCaps(); // TypeError on line 1: s.firstNameCaps is not a function

Person.prototype.firstNameCaps = function() {
  return this.first.toUpperCase();
};
s.firstNameCaps(); // "SIMON"
```

Interestingly, you can also add things to the prototype of built-in JavaScript objects. Let's add a method to `String` that returns that string in reverse:

```
var s = 'Simon';
s.reversed(); // TypeError on line 1: s.reversed is not a function

String.prototype.reversed = function() {
  var r = '';
  for (var i = this.length - 1; i >= 0; i--) {
    r += this[i];
  }
  return r;
};

s.reversed(); // nomiS
```

Our new method even works on string literals!

```
'This can now be reversed'.reversed(); // desrever eb won nac sihT
```

As mentioned before, the prototype forms part of a chain. The root of that chain is `Object.prototype`, whose methods include `toString()` — it is this method that is called when you try to represent an object as a string. This is useful for debugging our `Person` objects:

```
var s = new Person('Simon', 'Willison');
s.toString(); // [object Object]

Person.prototype.toString = function() {
  return '<Person: ' + this.fullName() + '>';
}

s.toString(); // "<Person: Simon Willison>"
```

Remember how `avg.apply()` had a null first argument? We can revisit that now. The first argument to `apply()` is the object that should be treated as '`this`'. For example, here's a trivial implementation of `new`:

```
function trivialNew(constructor, ...args) {
  var o = {}; // Create an object
  constructor.apply(o, args);
  return o;
}
```

This isn't an exact replica of `new` as it doesn't set up the prototype chain (it would be difficult to illustrate). This is not something you use very often, but it's useful to know about. In this snippet, `...args` (including the ellipsis) is called the "rest arguments" — as the name implies, this contains the rest of the arguments.

Calling

```
var bill = trivialNew(Person, 'William', 'Orange');
```

is therefore almost equivalent to

```
var bill = new Person('William', 'Orange');
```

`apply()` has a sister function named `call`, which again lets you set `this` but takes an expanded argument list as opposed to an array.

```
function lastNameCaps() {
  return this.last.toUpperCase();
```

```
  return this.last.toUpperCase();
}
var s = new Person('Simon', 'Willison');
lastNameCaps.call(s);
// Is the same as:
s.lastNameCaps = lastNameCaps;
s.lastNameCaps(); // WILLISON
```

Inner functions

JavaScript function declarations are allowed inside other functions. We've seen this once before, with an earlier `makePerson()` function. An important detail of nested functions in JavaScript is that they can access variables in their parent function's scope:

```
function parentFunc() {
  var a = 1;

  function nestedFunc() {
    var b = 4; // parentFunc can't use this
    return a + b;
  }
  return nestedFunc(); // 5
}
```

This provides a great deal of utility in writing more maintainable code. If a called function relies on one or two other functions that are not useful to any other part of your code, you can nest those utility functions inside it. This keeps the number of functions that are in the global scope down, which is always a good thing.

This is also a great counter to the lure of global variables. When writing complex code it is often tempting to use global variables to share values between multiple functions — which leads to code that is hard to

use global variables to share values between multiple functions — which leads to code that is hard to maintain. Nested functions can share variables in their parent, so you can use that mechanism to couple functions together when it makes sense without polluting your global namespace — "local globals" if you like. This technique should be used with caution, but it's a useful ability to have.

Closures

This leads us to one of the most powerful abstractions that JavaScript has to offer — but also the most potentially confusing. What does this do?

```
function makeAdder(a) {
  return function(b) {
    return a + b;
  };
}
var add5 = makeAdder(5);
var add20 = makeAdder(20);
add5(6); // ?
add20(7); // ?
```

The name of the `makeAdder()` function should give it away: it creates new 'adder' functions, each of which, when called with one argument, adds it to the argument that it was created with.

What's happening here is pretty much the same as was happening with the inner functions earlier on: a function defined inside another function has access to the outer function's variables. The only difference here is that the outer function has returned, and hence common sense would seem to dictate that its local variables no longer exist. But they *do* still exist — otherwise, the adder functions would be unable to work.

What's more, there are two different "copies" of `makeAdder()`'s local variables — one in which `a` is 5 and the other one where `a` is 20. So the result of that function calls is as follows:

```
add5(6); // returns 11
add20(7); // returns 27
```

Here's what's actually happening. Whenever JavaScript executes a function, a 'scope' object is created to hold the local variables created within that function. It is initialized with any variables passed in as function parameters. This is similar to the global object that all global variables and functions live in, but with a couple of important differences: firstly, a brand new scope object is created every time a function starts executing, and secondly, unlike the global object (which is accessible as `this` and in browsers as `window`) these scope objects cannot be directly accessed from your JavaScript code. There is no mechanism for iterating over the properties of the current scope object, for example.

So when `makeAdder()` is called, a scope object is created with one property: `a`, which is the argument passed to the `makeAdder()` function. `makeAdder()` then returns a newly created function. Normally JavaScript's garbage collector would clean up the scope object created for `makeAdder()` at this point, but the returned function maintains a reference back to that scope object. As a result, the scope object will not be garbage-collected until there are no more references to the function object that `makeAdder()` returned.

Scope objects form a chain called the scope chain, similar to the prototype chain used by JavaScript's object system.

A **closure** is the combination of a function and the scope object in which it was created. Closures let you save state — as such, they can often be used in place of objects. You can find several excellent introductions to closures .

Last modified: Jul 20, 2021, by MDN contributors