A re-introduction to JavaScript (JS tutorial)

by 117 contributors: Show all...

Introduction

Why a re-introduction? Because JavaScript is notorious for being the world's most misunderstood programming language. While it is often derided as a toy, beneath its deceptive simplicity lie some powerful language features. 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, an engineer at Netscape, and first released with Netscape 2 early in 1996. (It was originally going to be called LiveScript, but was renamed in an ill-fated marketing decision in an attempt 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, a mostly-compatible JavaScript work-alike, with Internet Explorer 3. 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 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**: For familiarity, 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 an object-oriented dynamic language with types and operators, standard built-in objects, and methods. Its syntax is based on the Java and C languages — so many structures from those languages apply to JavaScript as well. One of the key differences is that JavaScript does not have classes; instead, the class functionality is accomplished by object prototypes (see more about ES6 `Classes`).The other main difference is that functions are objects, giving functions the capacity to hold executable code and be 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`
- `String`
- `Boolean`
- `Function`
- `Object`
- `Symbol` (new in Edition 6)

... 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`
- `String`
- `Boolean`
- `Symbol` (new in Edition 6)
- `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

Numbers in JavaScript are "double-precision 64-bit format IEEE 754 values", according to the spec. This has some interesting consequences. There's no such thing as an integer in JavaScript, so you have to be a little careful with your arithmetic if you're used to math in C or Java. Watch out for stuff like:

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

In practice, integer values are treated as 32-bit ints (and are stored that way in some browser implementations), which 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 forgot to mention earlier called `Math` that provides advanced mathematical functions and constants:

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

If you don't provide the base, you can get surprising results in older browsers (pre-2013):

```
parseInt("010"); // 8
```

That happened because the `parseInt()` function decided to treat the string as octal due to the leading 0.

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, which unlike its `parseInt()` cousin always uses base 10.

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

```
+ "42"; // 42
```

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 input to any mathematical operation the result will also be `NaN`:

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

You can test for `NaN` using the built-in `isNaN()` function:

```
isNaN(NaN); // 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 simply 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 characters. More accurately, they are sequences of Unicode characters, with each character represented by a 16-bit number. This should be welcome news to anyone who has had to deal with internationalization.

If you want to represent a single character, you just use a string of length 1.

To find the length of a string, 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("hello", "gooc
"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 indicates an uninitialized value — 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 simply 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 the `var` keyword:

```
var a;
var name = "simon";
```

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 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 Edition 6, `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 not 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 prefix or postfix 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 in to a string first. This might catch 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 and make sure your comparisons are always accurate, you should always 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 += "!";
} else if (name == "kittens") {
  name += "!!";
} else {
  name = "!" + name;
}
name == "kittens!!"
```

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
}
```

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 setting default values:

```
var name = otherName || "default";
```

JavaScript has a ternary operator for conditional expressions:

```
var allowed = (age > 18) ? "yes" : "
```

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",
  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 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
// (that was the first parameter, and
```

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

```
obj.name = "Simon";
var name = obj.name;
```

And...

```
obj["name"] = "Simon";
var name = obj["name"];
```

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 though 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, b
obj["for"] = "Simon"; // works fine
```

Note: Starting from 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.

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` returned:

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

If you take the above into account, you can iterate over an array using the following:

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

This is slightly inefficient as you are looking up the length property once every loop. An improvement is this:

```
for (var i = 0, len = a.length; i < 
  // Do something with a[i]
}
```

A nicer-looking but limited idiom is:

```
for (var i = 0, item; item = a[i++];)
  // Do something with item
}
```

Here we are setting up two variables. The assignment in the middle part of the `for` loop is also tested for truthfulness — if it succeeds, the loop continues. Since `i` is incremented each time, items from the array will be assigned to item in sequential order. The loop stops when a "falsy" item is found (such as `undefined`).

This trick should only be used for arrays which you know do not contain "falsy" values (arrays of objects or DOM nodes for example). If you are iterating over numeric data that might include a 0 or string data that might include the empty string you should use the `i, len` idiom instead.

You can iterate over an array using a `for...in` loop. Note that if someone added new properties to `Array.prototype`, they will also be iterated over by this loop. Therefore this method is "not" recommended.

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

```
["dog", "cat", "hen"].forEach(functi
  // Do something with currentValue 
});
```

If you want to append an item to an array simply 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.
`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)`	Adds one or more items to the end.
`a.reverse()`	Reverses the array.
`a.shift()`	Removes and returns the first item.
`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.unshift(item1[,`	

`item2[, ...[, itemN]]])`	Prepends items to the start of the array.

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 unde
```

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

```
add(2, 3, 4); // 5
// added the first two; 4 was ignorec
```

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.lengt
    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.lengt
    sum += arguments[i];
  }
  return sum / arguments.length;
}

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

This is pretty useful, but introduces 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
    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 and call it with an arbitrary array of arguments, using the `apply()` method of any function object.

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

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.

JavaScript lets you create anonymous functions.

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

This is semantically equivalent to the `function avg()` form. It's extremely powerful, as it lets you put a full function definition anywhere that you would normally put an expression. This enables all sorts of clever tricks. Here's a way of "hiding" some local variables — like block scope in C:

```
var a = 1;
var b = 2;

(function() {
  var b = 3;
  a += b;
})();

a; // 4
b; // 2
```

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_N
    return elm.nodeValue.length;
  }
  var count = 0;
  for (var i = 0, child; child = elm.
    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(
  if (elm.nodeType == 3) { // TEXT_N
    return elm.nodeValue.length;
  }
  var count = 0;
  for (var i = 0, child; child = elm.
    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.
}
function personFullNameReversed(perso
  return person.last + ', ' + person.
}

s = makePerson("Simon", "Willison");
personFullName(s); // "Simon Willisor
personFullNameReversed(s); "Willis
```

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.
    },
    fullNameReversed: function() {
      return this.last + ', ' + this.
    }
  };
}

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

There's something here we haven't seen before: 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:

```
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.la
  };
  this.fullNameReversed = function()
    return this.last + ', ' + this.fi
  };
}
var s = new Person("Simon", "Willisor
```

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.firs
}
function Person(first, last) {
  this.first = first;
  this.last = last;
  this.fullName = personFullName;
  this.fullNameReversed = personFullN
}
```

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 = f
  return this.last + ', ' + this.firs
```

```
};
```

`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:

```
s = new Person("Simon", "Willison");
s.firstNameCaps(); // TypeError on li

Person.prototype.firstNameCaps = func
  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:

String.prototype.reversed = function
  var r = "";
  for (var i = this.length - 1; i >=
    r += this[i];
  }
  return r;
```

```
};

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

Our new method even works on string literals!

```
"This can now be reversed".reversed(
```

As I 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", "Williso
s; // [object Object]

Person.prototype.toString = function(
  return '<Person: ' + this.fullName(
}

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

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, ...a
  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, "Willia
```

is therefore almost equivalent to

```
var bill = new Person("William", "Ora
```

`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();
}
var s = new Person("Simon", "Willison
lastNameCaps.call(s);
// Is the same as:
s.lastNameCaps = lastNameCaps;
s.lastNameCaps();
```

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 betterExampleNeeded() {
  var a = 1;
  function oneMoreThanA() {
    return a + 1;
  }
  return oneMoreThanA();
}
```

This provides a great deal of utility in writing more maintainable code. If a 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 the function that will be called from elsewhere. 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 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 x = makeAdder(5);
var y = makeAdder(20);
```

```
x(6); // ?
y(7); // ?
```

The name of the `makeAdder` function should give it away: it creates a new 'adder' functions, which when called with one argument adds it to the argument that they were 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 one in which a is 20. So the result of those function calls is as follows:

```
x(6); // returns 11
y(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 initialised 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.

Memory leaks

An unfortunate side effect of closures is that they make it trivially easy to leak memory in Internet Explorer. JavaScript is a garbage collected language — objects are allocated memory upon their creation and that memory is reclaimed by the browser when no references to an object remain. Objects provided by the host environment are handled by that environment.

Browser hosts need to manage a large number of objects representing the HTML page being presented — the objects of the DOM. It is up to the browser to manage the allocation and recovery of these.

Internet Explorer uses its own garbage collection scheme for this, separate from the mechanism used for JavaScript. It is the interaction between the two that can cause memory leaks.

A memory leak in IE occurs any time a circular reference is formed between a JavaScript object and a native object. Consider the following:

```
function leakMemory() {
  var el = document.getElementById('e
  var o = { 'el': el };
  el.o = o;
}
```

The circular reference formed above creates a memory leak; IE will not free the memory used by `el` and `o` until the browser is completely restarted.

The above case is likely to go unnoticed; memory leaks only become a real concern in long running applications or applications that leak large amounts of memory due to large data structures or leak patterns within loops.

Leaks are rarely this obvious — often the leaked data structure can have many layers of references, obscuring the circular reference.

Closures make it easy to create a memory leak without meaning to. Consider this:

```
function addHandler() {
  var el = document.getElementById('e
  el.onclick = function() {
    el.style.backgroundColor = 'red'
  };
}
```

The above code sets up the element to turn red when it is clicked. It also creates a memory leak.

Why? Because the reference to `el` is inadvertently caught in the closure created for the anonymous inner function. This creates a circular reference between a JavaScript object (the function) and a native object (`el`).

There are a number of workarounds for this problem. The simplest is not to use the `el` variable:

```
function addHandler(){
  document.getElementById('el').oncli
    this.style.backgroundColor = 'rec
  };
}
```

Surprisingly, one trick for breaking circular references introduced by a closure is to add another closure:

```
function addHandler() {
  var clickHandler = function() {
    this.style.backgroundColor = 'rec
  };
  (function() {
    var el = document.getElementById(
    el.onclick = clickHandler;
  })();
}
```

The inner function is executed straight away, and hides its contents from the closure created with `clickHandler`.

Another good trick for avoiding closures is breaking circular references during the `window.onunload` event. Many event libraries will do this for you. Note that doing so disables the back-forward cache in Firefox, so you should not register an `unload` listener in Firefox, unless you have other reasons to do so.