

1. Understand Javascript Array Reduce in 1 Minute

<https://www.airpair.com/javascript/javascript-array-reduce>

JavaScript Reducing vs Looping

Everybody knows what **looping** over a collection is. But do you know what **reducing** a collection means?

No? Are you sure?

It sounds scary. But it is in fact pretty simple. You do it all the time. Here. Let me show you an example:

```
var total = 0;
var numbers = [1, 5, 7, 3, 8, 9];
for ( var i = 0; i < numbers.length; i++ ){
    total += numbers[i];
}
```

CODE

That's it. You just **reduced the "numbers" collection into the "total" variable**.

Pretty simple huh?

JavaScript Reduce Example

Let me show you another example:

```
var message = "";
var words = ["reducing", "is", "simple"];
for ( var i = 0; i < words.length; i++ ){
    message += words[i];
}
```

CODE

You just **reduced the "words" collection into the "message" variable**.

You see the pattern right?

You start with a **collection** (words) and a **variable** (message) with an **initial value** (the empty string in this case). You then iterate over the collection and append (or add) the values to the variable.

JavaScript Array.prototype.reduce

So what's the big deal, you ask? Well. Check this out:

```
var sum = [1, 2, 3].reduce(
    function(total, num){ return total + num }
, 0);
```

CODE

Whoa! So the Array object already knows how to reduce?

Indeed!

Every time you find yourself going from a list of values to one value (reducing) ask yourself if you could leverage the built-in [Array.prototype.reduce\(\)](#) function. You can reduce with any sort of operation that combines two values. Not just addition. And not just arithmetic operations.

So you know what Array reduce means. Isn't that awesome? It only took you one minute.

You can now head on to the [Mozilla Docs for Array.prototype.reduce](#) for a more in depth overview.

Map Reduce

When you hear people talking about "Map Reduce" they are just talking about a "pattern": Mapping over a collection and then reducing it.

Yeah. Really. Now you know.

2. JavaScript's Map, Reduce, and Filter

As engineers we build and manipulate arrays holding numbers, strings, booleans and objects almost everyday. We use them to crunch numbers, collect objects, split strings, search, sort, and more. So what's the preferred way to traverse arrays? For years it's been the trusty `for` loop which ensures iterating over the elements in order by index.

In 2011, JavaScript introduced `map`, `reduce`, and `filter` as powerful alternatives when translating elements, finding cumulative values, or building subsets based on conditions. These methods help the developer manage less complexity, work without side effects, and often make code more readable.

In this post we'll cover the usefulness of Array's `map`, `reduce`, and `filter` methods. You'll see use cases, code samples, behavior, and parameters of each method.

`reduce()` can find a Fibonacci sequence in $O(n)$

Drawbacks of Looping

Say I'm working on a bug fix and wading thru the 1,000s of lines of JavaScript in our codebase. I come across a `for` loop. Not exactly sure what it's doing, I take a closer look.

```
for(var i = 0; i < array.length; i++) {
    if(array.indexOf(array[i]) === i) {
        models.push(array[i]);
    }
}
```

CODE

We can see it traverses an array and inserts non-duplicate elements into a new array. But to figure this out we had to glean these 5 things.

code piece	meaning
<code>var i = 0</code>	starts at left side of array

<code>i < array.length</code>	finishes at right side of array
<code>i++</code>	increments by one
<code>array.indexOf(array[i]) === i</code>	if value is first instance in array, it'll match index. okay this means it's checking if it's a duplicate
<code>models.push(...)</code>	<code>models</code> must be a list. But what data's in it? What are their data types? I must search the file for "models". Rinse. Repeat.

We needed to check 5 pieces of information to determine what was going on. And this is a single `for` loop!

A Functional Approach

This same effect could be written using JavaScript's built-in `filter()` method.

```
var uniqueProducts = array.filter(function(elem, i, array) {
    return array.indexOf(elem) === i;
});
```

CODE

Simple and elegant.

Seeing `filter` communicates code behavior so I know exactly what it's doing. Compared to the looping approach above:

- Checking #1, #2, #3 is unnecessary because `filter()` does these automatically.
- #4 is the same but without the additional `if(...)` block.
- A big hurdle was #5. I had to search thru the code base to find what `models` even was. Did it already have data? Was it targeting specific data types? `map`, `reduce`, and `filter` solves this problem by not depending on code outside the callbacks, called side-effects.

In sum, `map`, `reduce` and `filter` makes code less complex, without side effects, and often more readable.

Let's look at each.

map()

Use it when: You want to translate/map all elements in an array to another set of values.

Example: convert Fahrenheit temps to Celsius.

CODE

```

var fahrenheit = [0, 32, 45, 50, 75, 80, 99, 120];

var celcius = fahrenheit.map(function(elem) {
    return Math.round((elem - 32) * 5 / 9);
});

// ES6
// fahrenheit.map(elem => Math.round((elem - 32) * 5 / 9));

celcius // [-18, 0, 7, 10, 24, 27, 37, 49]

```

What it does: Traverses the array from left to right invoking a callback function on each element with parameters (below). For each callback the value returned becomes the element in the new array. After all elements have been traversed map() returns the new array with all the translated elements^[1].

parameters:

CODE

```

array.map(function(elem, index, array) {
    ...
}, thisArg);

```

param	meaning
elem	element value
index	index in each traversal, moving from left to right
array	original array invoking the method
thisArg	(Optional) object that will be referred to as <code>this</code> in callback

filter()

Use it when: You want to remove unwanted elements based on a condition.

Example: remove duplicate elements from an array.

CODE

```

var uniqueArray = array.filter(function(elem, index, array) {
    return array.indexOf(elem) === index;
});

// ES6
// array.filter((elem, index, arr) => arr.indexOf(elem) === index);

```

What it does: Like `map()` it traverses the array from left to right invoking a callback function on each element. The returned value must be a boolean identifying whether the element will be kept or discarded. After all elements have been traversed `filter()` returns a new array with all elements that returned true^[2].

It has the same parameters as `map()`

parameters:

```
array.filter(function(elem, index, array) {
    ...
}, thisArg);
```

CODE

param	meaning
elem	element value
index	index in each traversal, moving from left to right
array	original array invoking the method
thisArg	(Optional) object that will be referred to as <code>this</code> in callback

[reduce\(\)](#)

Use it when: You want to find a cumulative or concatenated value based on elements across the array.

Example: Sum up orbital rocket launches in 2014.

```
var rockets = [
    { country:'Russia', launches:32 },
    { country:'US', launches:23 },
    { country:'China', launches:16 },
    { country:'Europe(ESA)', launches:7 },
    { country:'India', launches:4 },
    { country:'Japan', launches:3 }
];

var sum = rockets.reduce(function(prevVal, elem) {
    return prevVal + elem.launches;
}, 0);

// ES6
// rockets.reduce((prevVal, elem) => prevVal + elem.launches, 0);

sum // 85
```

CODE

What it does: Like `map()` it traverses the array from left to right invoking a callback function on each element. The value returned is the cumulative value passed from callback to callback. After all elements have been traversed `reduce()` returns the cumulative value[3].

parameters:

```
array.reduce(function(prevVal, elem, index, array) {
    ...
}, initialValue);
```

CODE

param	meaning
prevValue	cumulative value returned thru each callback
elem	element value
index	index of the traversal, moving from left to right
array	original array invoking the method
initialValue	(Optional) object used as first argument in the first (leftmost) callback.

Extras

- Each are methods on the Array's prototype object.
- Changing an element in the `array` parameter in any callback will persist across all remaining callbacks but has no effect on the returned array.
- Callback functions are invoked on indexes with any value, even `undefined`, but not ones deleted or never assigned a value.

It's worth noting `for` loops still definitely have a place when working with large arrays (e.g. over 1,000 elements) or needing to break the traversal if a condition is met.

In this post we covered how `map()`, `reduce()`, and `filter()` more easily communicate code behavior and likely reduces the need to track side effects. We also covered use cases, code samples, behavior, and parameters of each method. I hope this was useful for you.

3. Functional programming in Javascript: map, filter and reduce

This article is meant to be an introduction to functional programming in Javascript - specifically, it will explain the `map`, `filter` and `reduce` methods.

While these are natively available in any recent browser and in Node.js, most articles on them are far too technical to understand, while the concept of these functions is actually really simple, and will benefit *any* developer - even those working on simple scripts.

Note that I'll be pretty verbose in this article, to make the concept understandable to developers of any skill level. If you're already familiar with some of the concepts described, you can safely skip over those bits. Similarly, if you're a more advanced developer, and examples alone are enough for you, just skip over the text entirely.

I won't address other aspects of functional programming here - these three functions fit well into most workflows on their own. I might explain other concepts in a later post.

So let's start with `map`, arguably the most common one in a typical project.

So, what is this `map` business, then?

Think of `map` as a "for each" loop, that is specifically for *transforming* values - one input value corresponds to one 'transformed' output value.

You may have found yourself writing code that looks something like this:

CODE

```
var numbers = [1, 2, 3, 4];
var newNumbers = [];

for(var i = 0; i < numbers.length; i++) {
    newNumbers[i] = numbers[i] * 2;
}

console.log("The doubled numbers are", newNumbers); // [2, 4, 6, 8]
```

While this code *works*, it's not very nice. It takes a lot of work to do a pretty simple task - double all the numbers in an array.

What if we could simply write our *intention* of doubling every number in an array?

CODE

```
var numbers = [1, 2, 3, 4];

var newNumbers = numbers.map(function(number){
    return number * 2;
});

console.log("The doubled numbers are", newNumbers); // [2, 4, 6, 8]
```

Suddenly, we don't need a loop anymore, and we don't have to manually add numbers to an array either! We can simply define our *intention*, and let `map` do the work. This can also be chained easily:

CODE

```
var numbers = [1, 2, 3, 4];

var newNumbers = numbers.map(function(number){
    return number * 2;
}).map(function(number){
    return number + 1;
});

console.log("The doubled and incremented numbers are", newNumbers); // [3, 5, 7, 9]
```

Every number is now doubled, and has 1 added to it. We don't have to manually manage any arrays - we simply specify functions that do some kind of *transformation* on a single value.

There are a few 'rules' for `map` functions, though. While these are not always *strictly* enforced, you should keep to them regardless - they make it a lot easier to understand what your code is doing. If you think these rules will get in your way, just keep reading - your questions will be answered.

The amount of input elements is equal to the amount of output elements

You can't give `map` 4 values, and only receive 3 back. If the 'source array' has an X amount of elements, then the resulting array will also have X elements. Every output element corresponds to the input element in the same position - they're never shuffled around.

Your callbacks shouldn't 'mutate' values

What this means, is really just that you shouldn't modify objects or arrays directly from within your callbacks - if the input value is an object or an array, *clone* it instead, and modify the copy.

This way, there's a guarantee that your callback doesn't cause 'side effects' - that is, no matter what happens in your callback, it will only affect the specific value you're working with. This makes it much easier to write reliable code.

You can clone an array in Javascript by doing `array.slice(0)`. Note that this is a 'shallow clone' - if the values in the array are themselves arrays or objects, they will still be the same values in both arrays.

Shallow-cloning an object is a little more complex. If you're using a CommonJS environment (Node.js, Webpack, Browserify, ...), you can simply use the [xtend](#) module. Otherwise, using a function like this should suffice:

```
function cloneObject(obj) {
  var newObj = {};

  for (var key in obj) {
    newObj[key] = obj[key];
  }

  return newObj;
}

var clonedObject = cloneObject(originalObject);
```

CODE

There is an exception to this rule, but we'll get to that later - just assume that this rule always applies, unless told otherwise.

Don't cause side-effects!

You should never do anything in a `map` call that modifies 'state' elsewhere. For example, while making a HTTP GET request is fine (although not really possible with plain `Array.map`), changing another array outside of the callback is not. Your callback can *only* modify the new value you're returning *from* that callback.

But... isn't this slow? All those callbacks and the cloning...

Don't worry about it. Such operations are actually rather fast in Javascript, especially in V8-based engines like Node.js, and it's extremely unlikely that it'll ever cause performance issues for you.

Always write code for readability first, and for performance second - it's much easier to optimize readable code, than it is to make optimized code readable.

But what if I only want to transform some of the values?

Perhaps your source array has some values that you want to transform, and some values that you just want to throw away entirely. That's not possible with `map` alone, as the number of input values and the number of output values for a `map` call is always equal.

Say you want to double the odd numbers, but throw away the even numbers. Your code may look something like this:

CODE

```

var numbers = [1, 2, 3, 4];
var newNumbers = [];

for(var i = 0; i < numbers.length; i++) {
    if(numbers[i] % 2 !== 0) {
        newNumbers[i] = numbers[i] * 2;
    }
}

console.log("The doubled numbers are", newNumbers); // [2, 6]

```

This is what it'd look like using `map` and `filter`:

CODE

```

var numbers = [1, 2, 3, 4];

var newNumbers = numbers.filter(function(number){
    return (number % 2 !== 0);
}).map(function(number){
    return number * 2;
});

console.log("The doubled numbers are", newNumbers); // [2, 6]

```

The `filter` callback is subject to the same "don't mutate" and "don't cause side-effects" rules as `map`, but the mechanics are different.

The return value from the `filter` callback should be a boolean, indicating whether to include the original value in the result (`true`) or whether to leave it out (`false`). You should *not* return the value itself, just a boolean - you can't modify the value from within the callback. The return value just decides whether the value will be included in the result, nothing else.

Chaining

As you might have noticed, you can chain `map` and `filter` calls indefinitely. Each of them just returns an array, and every array has these methods by default, so you can build a jQuery-like chain. There is no limit to how many of these calls you can chain, and it's usually pretty simple to build complex array transformations from just these functions alone.

An array goes in, an array comes out, and the callback operates on each value individually.

So... what if I just want to combine the values in an array?

'Combining' the values can mean a lot of different things. For example, you might want to sum all the numbers, doubled:

CODE

```

var numbers = [1, 2, 3, 4];
var totalNumber = 0;

for(var i = 0; i < numbers.length; i++) {
    totalNumber += numbers[i] * 2;
}

console.log("The total number is", totalNumber); // 20

```

But what if we want to do that in a `map` / `filter` chain? Easy enough, with `reduce`:

CODE

```

var numbers = [1, 2, 3, 4];

var totalNumber = numbers.map(function(number){
    return number * 2;
}).reduce(function(total, number){
    return total + number;
}, 0);

console.log("The total number is", totalNumber); // 20

```

It works like this:

1. The second argument to the function call is considered to be the 'starting value' for the total - this is what you start out with.
2. For each item in the array, it calls the callback, with the total value up to that point, and the item itself. For the first item, the 'total value' is the starting value.
3. You return a new 'total value'. In this case, it's the sum of all previous numbers plus the current number. This return value is used as the 'total value' for the *next* item.
4. After running out of items, the 'cumulative' total value is returned.

Again, the "don't mutate" and "don't cause side-effects" rules apply. Otherwise, it doesn't matter what kind of value you're working with - you could be summing a number, concatenating a string, putting together an object, and so on.

Just keep in mind that `reduce` always returns *just* the 'total value' - unless you've specifically made it an array, it won't be an array like for `map` and `filter`. That means that unless you're explicitly returning an array from it, you can't chain off it any further. But then again, it wouldn't really make sense to run `map` on a number!

But what if I want to add items? Like `filter`, but the opposite.

If you use Node.js, you may be familiar with transform streams - these are kind of like a `map` callback, but besides letting you 'push' 0 or 1 values, they also let you push *multiple* values. This can be useful if you are, for example, 'flattening' an array of arrays down to a single array with all the values.

While this can't be done with `map` - after all, one input element means one output element - it *can* be done with `reduce`, despite what the name implies. It's a slightly unusual trick, but you'll occasionally find yourself needing it.

Let's say you want to add the even numbers to the resulting array twice, but the odd numbers only once. This is what it could look like:

CODE

```

var numbers = [1, 2, 3, 4];
var newNumbers = [];

for(var i = 0; i < numbers.length; i++) {
    var number = numbers[i];
    newNumbers.push(number);

    if(number % 2 == 0) {
        /* Add it a second time. */
        newNumbers.push(number);
    }
}

console.log("The final numbers are", newNumbers); // [1, 2, 2, 3, 4, 4]

```

This is how you'd do it with `reduce`:

CODE

```

var numbers = [1, 2, 3, 4];

var newNumbers = numbers.reduce(function(newArray, number){
    newArray.push(number);

    if(number % 2 == 0) {
        /* Add it a second time. */
        newArray.push(number);
    }

    return newArray; /* This is important! */
}, []);

console.log("The final numbers are", newNumbers); // [1, 2, 2, 3, 4, 4]

```

Note how we start out with `[]` as initial value, and just add the value to it one or more times. While this *does* violate the "no mutation" rule, it's okay in this particular situation - the array object is created explicitly for this `reduce` call, so it can't cause side-effects anywhere else in the code.

When you use this trick, *don't forget to return the array!* It's easy to forget this, and you'll end up with an unexpected output. The `reduce` callback always expects the *new* total value to be returned, even if that value is an array, and even if it's the same one as before!

The same trick also works with objects - however, in that case you'd specify `{}` as starting value, and you'd use property assignment rather than `.push`.

Bonus: `forEach`

Perhaps you really *do* want a 'for each' loop - you're not transforming and returning any values, you just want to cause some kind of side-effect (eg. creating a DOM element). It would be nice if you could chain it at the end of a `map / reduce / filter` 'pipeline', and indeed you can!

This is what your `for` loop might normally look like:

CODE

```
var numbers = [1, 2, 3, 4];

for(var i = 0; i < numbers.length; i++) {
    doSomethingWith(numbers[i]);
}
```

And this is what it would look like using `forEach`:

CODE

```
var numbers = [1, 2, 3, 4];

numbers.forEach(function(number){
    doSomethingWith(number);
});
```

You can do the same for an object, using `Object.keys`:

```
var numbers = {one: 1, two: 2, three: 3, four: 4}; Object.keys(numbers).forEach(function(key){ var value =
numbers[key]; doSomethingWith(value); /* For example, key == "one" and value == 1 */});
```

You should really always consider using `forEach` instead of a `for` loop. Because `forEach` uses callbacks, it prevents scope problems with asynchronous operations in a loop - that is, you don't have to worry that the value of the `key` variable changes inbetween the start and the completion of your operations.

Indexes

At some point, you may want to get the index of the current item in the source array - however, I've only demonstrated the use of the value so far.

Getting the index of the current element is trivial: it's the second argument for `map`, `filter` and `forEach`, and the third argument for `reduce`. In other words, it comes after the arguments I've shown in the examples so far.

If you need a reference to the original array for some reason, that is the argument after *that* - however, needing this usually means that you're doing something wrong. You should never directly modify the source array from any of these functions, for example.

Further reference

MDN has further documentation on [map](#), [filter](#), [reduce](#), and [forEach](#).

Other libraries

Some utility libraries like `lodash` and `underscore` offer similar methods, but you almost certainly don't need them. All modern Javascript runtimes and browsers have these functions [available natively](#) on array objects.

jQuery offers similar methods, but keep in mind that the jQuery equivalents **switch around the value and the index** - that is, the order of arguments for the callback is `(index, value)`. A more reliable way is to use `this` within your callback where possible - it is always set to the value in jQuery.

My next Javascript and Node.js article will be about how Promises can help you escape callback hell. I will be

discussing the bluebird library specifically, including its equivalents of `map`, `filter`, `reduce` and `forEach` (named `each` in bluebird).

*I am currently offering **Node.js code review and tutoring** services, at affordable rates. More details can be found [here](#).*

4. Build Better Apps with ES6 Modules

ES6 is packed with features like iterators, generators, maps, sets, symbols, template strings and more. One of the biggest game changers to app architecture is the new modules, and different than the syntactic sugar of `class` and `extends` they're an entirely new construct^[1]. In this post I'll cover ES6's new module syntax and how it provides better performance, encapsulation, and error handling.

But first a little backstory. Engineers have long defined modules with workarounds on the global `window` object using object literals, the module pattern, and any combination of IIFEs one can imagine. Each of these comes with a set of challenges from lack of encapsulation to forced singletons. In 2010 RequireJS became a popular tool to define modules client-side, being used on sites like PayPal, Dropbox and The New York Times. CommonJS, it's counterpart, was adopted server-side by the Node.js community.

So why even use modules?

Why Modules?

As JavaScript code scales developers often organize it into multiple source files. Makes sense, right? But if we take these files and simply load them in our HTML there are issues.

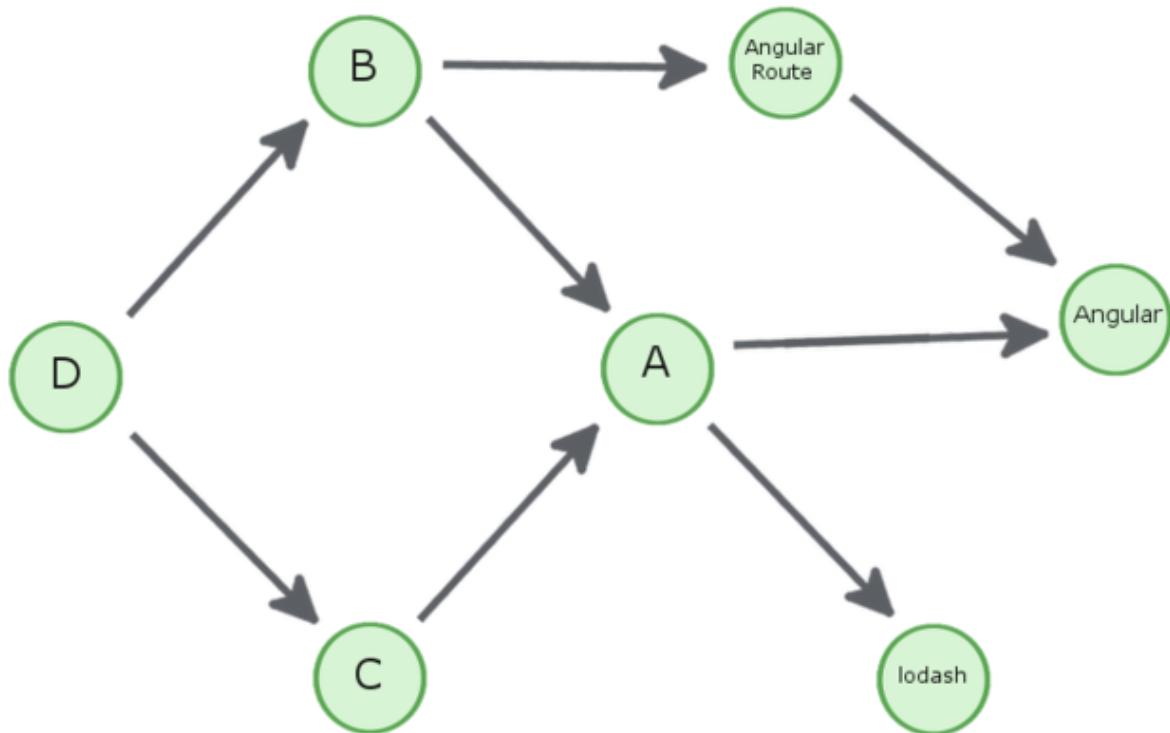
- **Performance** - Loading multiple source files at runtime as `<script>` tags can block execution.

```
<body>
  ...
  <script src="lib/angular.js"></script>
  <script src="lib/angular-route.js"></script>
  <script src="lib/lo-dash.js"></script>
  <script src="src/A.js"></script>
  <script src="src/B.js"></script>
  <script src="src/C.js"></script>
  <script src="src/D.js"></script>
</body>
```

CODE

Even though modern browsers load all scripts in parallel, separate network requests are made. And these scripts must execute in sequence, where one loading script will block a subsequent script from executing.

- **Unclear Dependencies** - In `<script>`s above what if `D.js` depends on `B.js` and `C.js` while `B.js` and `C.js` depend on `A.js`. We could remove `D.js` on its own but if we removed `C.js` we'd need to remove `D.js` too. But how would you know? A list can't identify these relationships because they're better modeled as *directed acyclic graphs*.



- **Heavy reliance on `window object`** - All function declarations and variables in top level scope become attached to a single global `window` object. We want concise, snappy modules but instead get something akin to this.

Thankfully, a better approach has arrived.

ES6 Modules

Modules support a one-module-per-file architecture using `export` and `import` declarations to communicate dependencies. They're fast, encapsulated, catch errors earlier, and avoid the global `window` object altogether.

A module can export any type of primitive or native object value, whether a string, number, boolean, object, array, function or more. A single module can import/export one primary value, called the **default**, along with multiple values called **named** imports/exports. We'll focus on default first.

Default Imports/Exports

ES6 prefers default imports/exports over named by giving them the most concise syntax.

Export

Examples	Export Name	Module Requested	Import Name	Local Variable
<code>export default function func() {...}</code>	"default"			"func"
<code>export default function() {...}</code>	"default"			"default"
<code>export default 42</code>	"default"			"default"

[3]

The `default` keyword creates the default export. Easy, right? Remember though that each file can have only one default export.

Import

Example	Module Requested	Import Name	Local Variable Name
<code>import myDefault from "mod";</code>	"mod"	"default"	"myDefault"

The default exported value from `mod.js` is imported as local variable `myDefault`.

Named Imports/Exports

Named import/export syntax is slightly more verbose. We either export variables inline, use `{}`, or `*`.

Export

Examples	Export Name	Module Requested	Import Name	Local Variable
<code>export var num = 100000</code>	"num"			"num"
<code>export { str }</code>	"str"			"str"
<code>export { str as myStr }</code>	"myStr"			"str"
<code>export * from "someMod"</code>		"someMod"	"*"	

1st: Exports declaration `var num = 100000` as a named export.

2nd: Curly braces export a previously declared variable. `str` was declared earlier as a `var`, `function`, `class`, or `let`.

3rd: Same as 2nd except `str` variable is renamed as `export myStr`.

4th: Re-exports all named exports from `someMod.js`.

Import

Curly braces `{ }` or an `*` will import a *named export*.

Example	Module Requested	Import Name	Local Variable
<code>import { num } from "mod";</code>	"mod"	"num"	"num"
<code>import { num as myNum } from "mod";</code>	"mod"	"num"	"myNum"
<code>import * as myObj from "mod";</code>	"mod"	"*"	"myObj"

1st: The `num` export from `mod.js` is imported to local variable `num`.

2nd: Same as 1st except the local variable `myNum` references named export `num`.

3rd: The `*` imports all named exports from module `mod.js` into a single object literal called `myObj`

More examples

myMod.js

CODE

```
var str = 'booyah';
var num = 100000;

// named exports
export str;
export num;
export function func() {
    return 'do stuff';
}
// OR -- shorthand for named exports
// export { str, num, func }

// default export
export default function() {
    return 'default';
}
```

main.js

CODE

```
// import 2 named exports
import { str, num } from "myMod"

// import default
import myDefault from "myMod"

// OR -- shorthand for default and named exports
// import myDefault, { str, num } from "myMod"
```

Note named export `func` wasn't imported into `main.js`.

Beyond the syntax, ES6 modules were designed with higher level goals.

1. Declarative Structure

Module definitions use **declarative syntax** which is a bit foreign at first but becomes quite readable.

CODE

```
/** 
 * import named exports myNum, myObj from mod.js
 */
import { myNum, myObj } from 'mod';
```

Notice there's no assignment. Most importantly, this simple syntax sets the stage for *compile-time resolution*.

2. Compile-Time Resolution

When a page loads:

1. JS text is parsed, and the "import", and "export" syntax is found.
2. Any dependencies are fetched and parsed.
3. Once the dependency tree has been fetched the ES module loader maps all dependency exports to the module's imports. [4]

All this occurs at compile-time, before any runtime code executes. Benefits include:

- **faster lookups.** Instead of assigning modules dynamic objects created at runtime like with RequireJS and CommonJS, static references resolve quicker in benchmarks because they don't require PIC Guards[2].
- Developer specified imports and exports provide better **encapsulation** by using built-in language constructs over RequireJS and CommonJS's use of mutable runtime objects.
- The interpreter can **handle errors** at compile-time before the script starts executing. e.g. trying to import a not-yet-created export.

3. Collections of Code

ES6 modules are defined broader than objects with properties. They're instead "collections of code" where both default and named exports can be exported. But by allowing *pieces* of modules to be imported critics think it defies the essence of a module as a *single unit* of functionality. Do they merely pave the way for bags-of-methods modules?

Isaac Schlueter, who created NPM, echoed this concern.

If the author wants [the module] to export multiple things, then let them take on the burden of creating an object and deciding what goes on it. The syntax should suggest that a "module" ought to be a single "thing", so that these util-bags appear as warty as they are. [2]

Dave Herman, the lead architect of ES6 modules replied.

If you want to export multiple functions, for example, you can export a single object with multiple function properties. But an object is a dynamic value. There's no way to say create a compile-time aggregation of declarative things analogous to an object. Such as a collection of macros, or a collection of type definitions, or a collection of sub-modules that themselves contain static things. It's one thing to encourage single-export modules, it's another to require it. [2]

At least ES6 favors default imports/exports, giving them the best syntax compared to named exports.

4. Cyclic Dependencies & Async Loading

Modules support cyclic dependencies, allowing module A to import module B while module B imports module A. Cyclic dependancies are usually a hallmark of poor design. But they may be useful in edge cases. For example, tree structures can use cyclic dependancies when child nodes refer to their parents (e.g. the DOM).

Browsers will likely be adding a `Loader` with `.get()` functionality for asynch loading but it's only in a draft phase[5].

Modules are one of the best features of ES6. Despite the clunky syntax they provide better performance, encapsulation and error handling, while setting the stage for greater things like macros and types to come. I'd enjoy

any questions or comments. Please feel free to reach out on
[LinkedIn](#).

5. Events, Concurrency and JavaScript

//danmartensen.svble.com/events-concurrency-and-javascript

Modern web apps are inherently event-driven yet much of the browser internals for triggering, executing, and handling events can seem as black box. Browsers model asynchronous I/O thru events and callbacks, enabling users to press keys and click mouses while XHR requests and timers trigger in code. Understanding how events work is critical for crafting high performance JavaScript. In this post we'll focus on the browser's built-in Web APIs, callback queues, event loops and JavaScript's run-time.

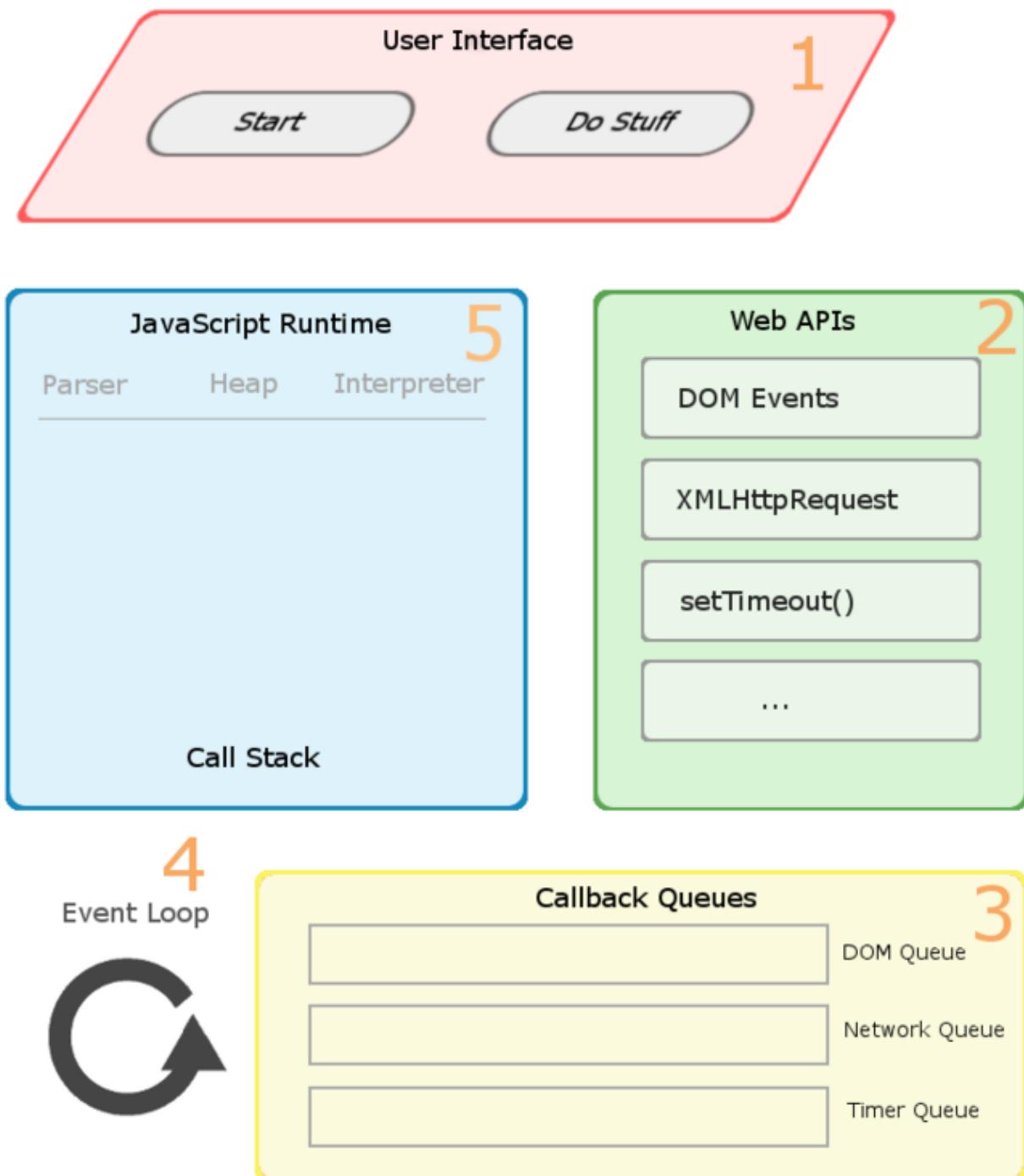
Code in action. A button and event handler.

```
<button id="doStuff">Do Stuff</button>

<script>
    document.getElementById('doStuff')
        .addEventListener('click', function() {
            console.log('Do Stuff');
        }
    );
</script>
```

CODE

Let's trace a Do Stuff click event thru browser and describe the components along the way.



From Philip Robert's diagram

Browser Runtime

- User Interface** - User clicks the Do Stuff button. Simple enough.
- Web APIs** - The click event propagates thru the DOM's Web API triggering click handlers during the capture and bubble phases on parent and child elements. Web APIs are a multi-threaded area of the browser that allows many events to trigger at once. They become accessible to JavaScript code thru the familiar window object on page load. Examples beyond the DOM's document are AJAX's XMLHttpRequest, and timers setTimeout() function[1].
- Event Queue** - Next the event's callback is pushed into one of many event queues (also called task queues). Just as there are multiple Web APIs, browsers have event queues for things like network requests, DOM

events, rendering, and more^[2].

4. **Event loop** - Then a single event loop chooses which callback to push onto the JavaScript call stack^[3]. Here's C++ pseudo code for Firefox's event loop.

```
while(queue.waitForMessage()){
    queue.processNextMessage();
}
```

CODE

[4]

Finally the event callback enters the JavaScript's runtime within the browser.

JavaScript Runtime

The JavaScript engine has many components such as a parser for script loading, heap for object memory allocation, garbage collection system, interpreter, and more. Like other code event handlers execute on it's call stack.

5. **Call Stack** - Every function invocation including event callbacks creates a new stack frame (also called execution object). These stack frames are pushed and popped from the top of the call stack, the top being the currently executing code^[5]. When the function is returned it's stack frame is popped from the stack.

Chrome's V8 C++ source code of single stack frame:

```
/*
 * v8.h line 1372 -- A single JavaScript stack frame.
 */
class V8_EXPORT StackFrame {
public:
    int GetLineNumber() const;
    int GetColumn() const;
    int GetscriptId() const;
    Local GetScriptName() const;
    Local GetScriptNameOrSourceURL() const;
    Local GetFunctionName() const;
    bool IsEval() const;
    bool IsConstructor() const;
};
```

CODE

[6]

Three characteristics of JavaScript's call stack.

Single threaded - Threads are basic units of CPU utilization. As lower level OS constructs they consist of a thread ID, program counter, register set, and stack^[7]. While the JavaScript engine itself is multi-threaded it's call stack is single threaded allowing only one piece of code to execute at a time^[8].

Synchronous - JavaScript call stack carries out tasks to completion instead of task switching and the same holds for events. This isn't a requirement by the ECMAScript or WC3 specs. But there are some exceptions like `window.alert()` interrupts the current executing task.

Non-blocking - Blocking occurs when the application state is suspended as a thread runs^[7]. Browsers are non-

blocking, still accepting events like mouse clicks even though they may not execute immediately.

CPU Intensive Tasks

CPU intensive tasks can be difficult because the single-threaded and synchronous run-time queues up other callbacks and threads into a wait state, e.g. the UI thread.

Let's add a CPU intensive task.

```
<button id="bigLoop">Big Loop</button>
<button id="doStuff">Do Stuff</button>
<script>
    document.getElementById('bigLoop')
        .addEventListener('click', function() {
            // big loop
            for (var array = [], i = 0; i < 100000000; i++) {
                array.push(i);
            }
        });
    document.getElementById('doStuff')
        .addEventListener('click', function() {
            // message
            console.log('do stuff');
        });
</script>
```

CODE

Click Big Loop then Do Stuff. When Big Loop handler runs the browser appears frozen. We know JavaScript's call stack is synchronous so Big Loop executes on the call stack until completion. It's also non-blocking where Do Stuff clicks are still received even if they didn't execute immediately.

Checkout this [CodePen](#) to see.

Solutions

- 1) Break the big loop into smaller loops and use `setTimeout()` on each loop.

```

...
document.getElementById('bigLoop')
    .addEventListener('click', function() {
        var array = []
        // smaller loop
        setTimeout(function() {
            for (i = 0; i < 5000000; i++) {
                array.push(i);
            }
        }, 0);
        // smaller loop
        setTimeout(function() {
            for (i = 0; i < 5000000; i++) {
                array.push(i);
            }
        }, 0);
    });
}

```

`setTimeout()` executes in the WebAPI, then sends the callback to an event queue and allows the event loop to repaint before pushing it's callback into the JavaScript call stack.

2) Use Web Workers, designed for CPU intensive tasks.

Summary

Events trigger in a multi-threaded area of the browser called **Web APIs**. After an event (e.g. XHR request) completes, the Web API passes it's callback to the **event queues**. Next an **event loop** synchronously selects and pushes the event callback from the callback queues onto JavaScript's single-threaded **call stack** to be executed. In short, events trigger asynchronously but their handlers execute on the call stack synchronously.

In this post we covered browser's concurrency model for events including Web APIs, event queues, event loop, and JavaScript's runtime. I'd enjoy questions or comments. Feel free to reach out via [LinkedIn](#).

6. What's "new" in JavaScript?

<https://danmartensen.svbtle.com/whats-new-in-javascript>

Few syntactic features highlight JavaScript's object oriented nature more than the `new` operator. Many of us have been using it for years but do we really know what it does? In languages like Java it allocates memory for an instance object, invokes a constructor function, and returns a reference to the object. And the same holds for JavaScript. But with the language's use of first class functions and prototypal inheritance, there's much more to `new`.

In this post I'll cover the relationship between function instance objects, prototype objects and instance objects, what happens when `new` invokes functions, and which functions can be used as constructors.

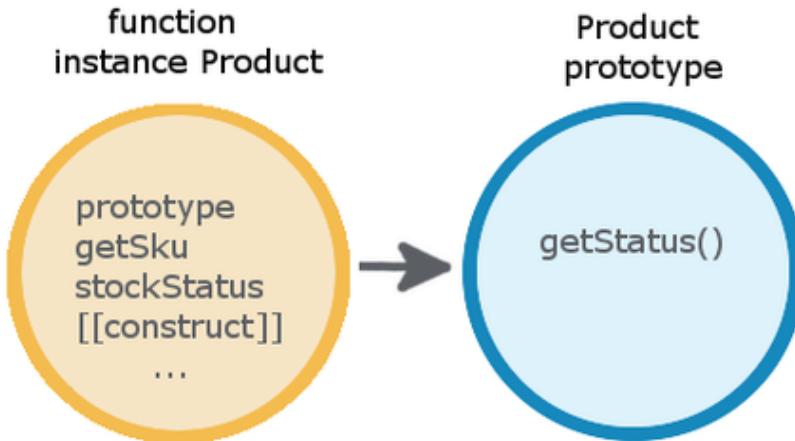
Function Instances & Prototypes

Before we explore how `new` works let's quickly review JavaScript functions. When a function is defined a *function instance object* is created under the hood, having a *prototype* property that references it's *prototype object*^[1]. Here's an example.

CODE

```
function Product(sku, status) {
    this.getSku = function() { return sku; };
    this.stockStatus = status;
}
```

In memory, the two objects look like:



Note I defined a method on the prototype object itself by assigning to function's `prototype` property,

```
Product.prototype.getStatus = function() {
    return this.stockStatus;
};
```

CODE

new Operator

So meet 3 objects.

```
var product1 = new Product('ABC1234', 'InStock');
var product2 = new Product('DEF5678', 'BackOrder');
var product3 = new Product('GHI9012', 'Discontinued');
```

CODE

For each `new Product` the following steps occur:

1. An empty instance object is created.
2. The new object is implicitly assigned many internal properties common to all objects. See details below.
3. The new object is passed into constructor `Product` as `this`.
4. The `Product` constructor's `this` properties and methods are copied to the new object. In our example it's `status` and `getSKU()`.
5. The new object's memory address is returned unless a `return` keyword is supplied. If so, a specified object may be returned instead.

[2]

Taking a closer look at step 2, internal properties on the new object are

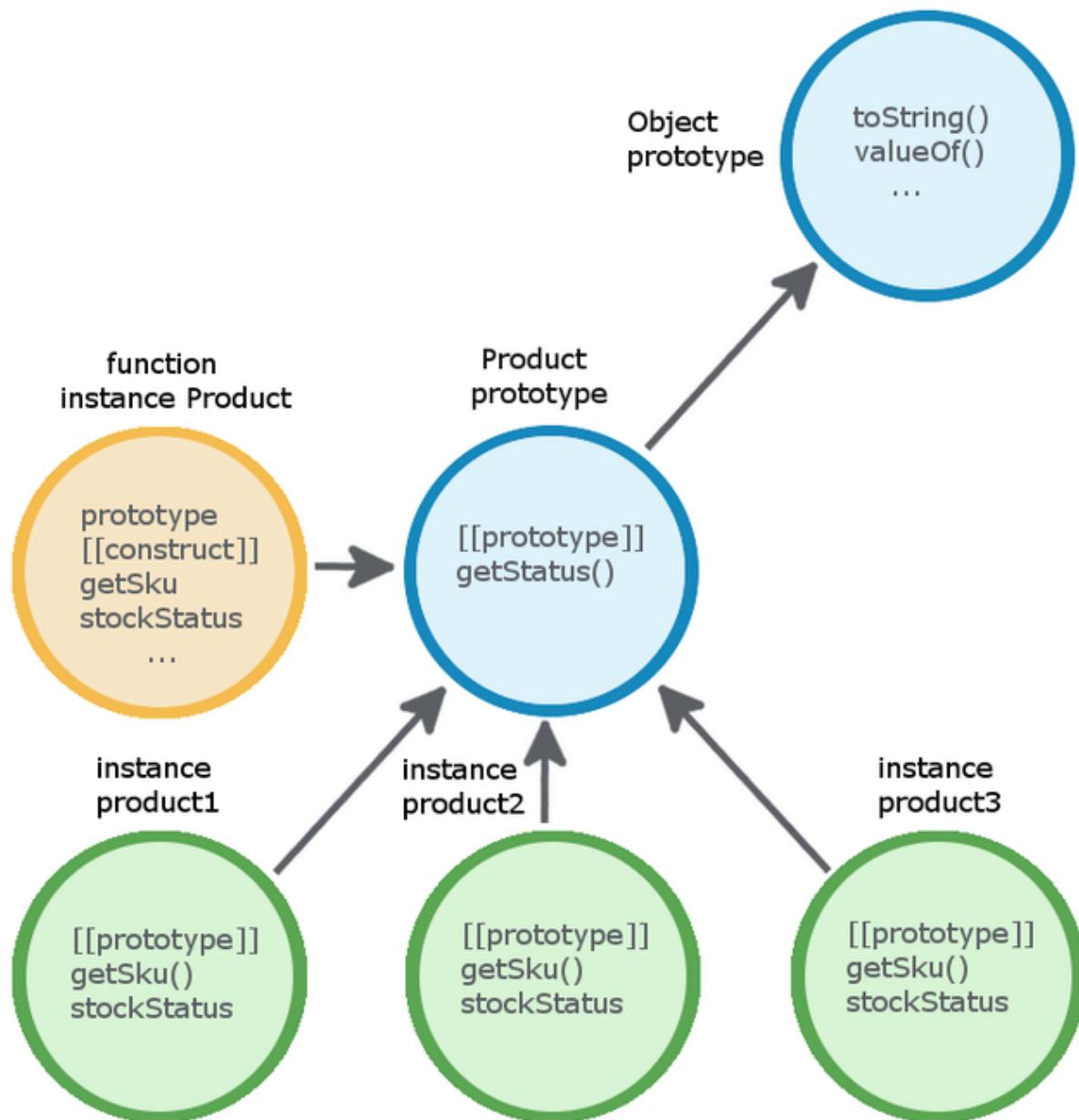
Property	Value Description
[[Prototype]]	The reference to the prototype object.
[[Class]]	String indicating kind of object
[[Extensible]]	If true, own properties may be added to the object.
[[Put]]	Sets a property value
[[Get]]	Returns a property value
...	...

[3]

I want to focus on the `[[Prototype]]` property. The `Product` constructor function's `prototype` property which references its prototype object is copied to the new object's `[[prototype]]` property.

All `prototype objects` also have a `[[prototype]]` property. By default it references `Object.prototype`, inheriting all of `Object`'s properties and methods.

We can visualize these objects in memory.



In sum, `new` invoked `Product` as a constructor function (Yellow), creating an object (Green), assigning it `instance` and internal properties including `[[prototype]]` which point its prototype object (Blue), and returning references to `product1`, `product2`, and `product3`. The instance objects now link to the same prototype object thru their internal `[[prototype]]` property, giving them access to the prototype's shared methods.

Using `new`

The syntax

```
new ConstructorFunction([args]);
```

`args` is an optional comma separated list of values. When there are no arguments, the parenthesis themselves are optional. Constructor functions usually begin with a capital letter.

So why does using `new` work for some functions but throws `TypeError` for others?

CODE

```
new Math()      // TypeError on built-in object
new Element()   // TypeError on host object
new HTMLElement() // TypeError on host object
```

Functions can be invoked with `new` when they have an internal `[[construct]]` property. We'll look thru the lens of object data types to identify which functions have this property. Objects can be grouped into three categories: *custom objects* are defined by us developers, *built-in* objects defined by the JavaScript spec, and *host objects* provided by the host (browser) environment^[4].

All custom functions defined by us developers have the `[[construct]]` property, being able to be invoked with `new`. Plain and simple.

```
new Product('QRS1234', 'InStock'); // works
```

Most built-ins objects can be invoked with `new`, such as these familiar ones.

CODE

```
new Object() // works
new Array()  // works
new RegExp() // works
```

But what about `Math` and `JSON` ?

CODE

```
new Math()    // TypeError
new JSON()   // TypeError
```

The trick with built-in objects is most have associated constructor functions that can be invoked with `new`, but those two along with the `Global` object don't. There are also built-in functions such as `parseInt()` and `decodeURI()` but it's clear they can't be invoked with `new` because they begin with a lowercase letter.

For host objects, only a few have the `[[construct]]` property.

CODE

```
new XMLHttpRequest() // works
new Screen()         // TypeError
new Node()           // TypeError
```

In this post we covered it by focusing on function objects, their prototypes and instance objects, the actions taken when a function is invoked using `new`, and what functions can be used as constructors.

I hope there was something of value for you in this post. I'd greatly enjoy your comments or questions. Please feel free to reach out on [LinkedIn](#)

7. The Anatomy of a JavaScript Function

January 16, 2015

[The Anatomy of a JavaScript Function](#)

Functions in JavaScript are like classes to Java. They're the fundamental modular unit: the cell in life, the note in music, the word in language, the funky chicken in dance. JavaScript functions are full fledged objects, often called *first-class objects*[1], having properties and methods, mutable values, and dynamic memory. Douglas Crockford, one of the early gurus and longtime critic of the language wrote[2]:

The best thing about JavaScript is its implementation of functions.

In this post I cover the qualities of functions as objects, how function objects are created, the difference between constructor, prototype, and instance objects, how the `new` operator works, and useful properties of functions.

First Class Objects

Functions have the same capabilities as other objects. So it's no secret they can be:

passed as arguments

```
function a() { ... }
function b(a) { ... }
b(a);
```

CODE

returned from functions

```
function a() {
  function b() { ... }
  return b;
}
```

CODE

assigned as data values

```
var a = function() { ... };
```

CODE

have properties and methods

```
function a() { ... }
var obj = {};
a.call(obj);
```

CODE

defined anywhere an expression can be, allowing them to be nested

```

function a() {
  function b() {
    function c() {
      ...
    }
    c();
  }
  b();
}
a();

```

Constructors, Prototypes, & Instances

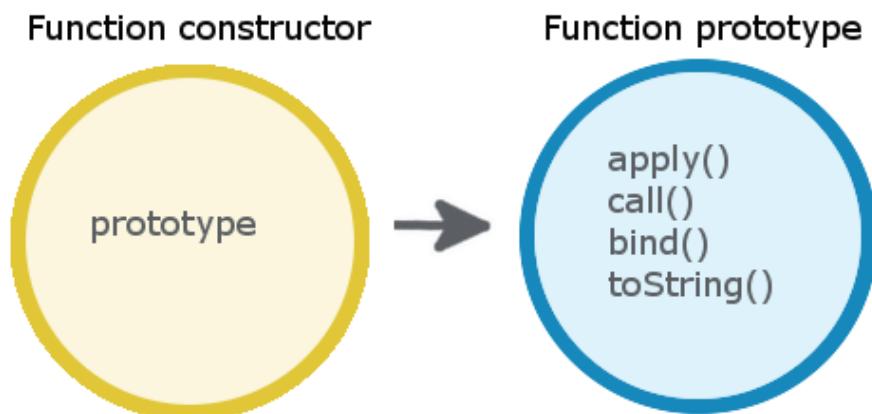
Before we look at function internals let's quickly review the basics of prototypes. Every function carries a *prototype* property linked to an internal *prototype object*[3]. When a function is invoked as a *constructor* to create *instance objects*, which we'll cover shortly, those instance objects carry the link to the same prototype object, giving them shared behavior across instances.

The key to understanding functions is in the relationship between *three objects*:

- the Function constructor object
- its prototype object
- function instance objects

Let's focus on the first two. The built-in Function *constructor object* is itself a function, having a prototype property that references it's prototype object thru Function.prototype [3]. These two objects are built into the JavaScript runtime before any of our code starts executing.

We can visualize them in memory.



Creating Function Instances

Meet 2 functions.

```

function Alpha() { ... }
function Bravo() { ... }

```

The simple function keyword is a workhorse. When these two functions are defined:

- The function keyword implicitly creates a brand new instance object and passes it into constructor `Function()`.
- The new instance object is implicitly assigned many internal properties, one being the `[[prototype]]` property. The `Function` constructor's `prototype` property referencing its prototype object is copied into this new object's `[[prototype]]` property^[3].
- The new instance object is also a function, so it too has a `prototype` property that refers to a prototype object that is automatically created, allowing the possibility that it will be used as a constructor.
- The new object is returned and assigned to its reference.

[3]

Augmenting Prototypes

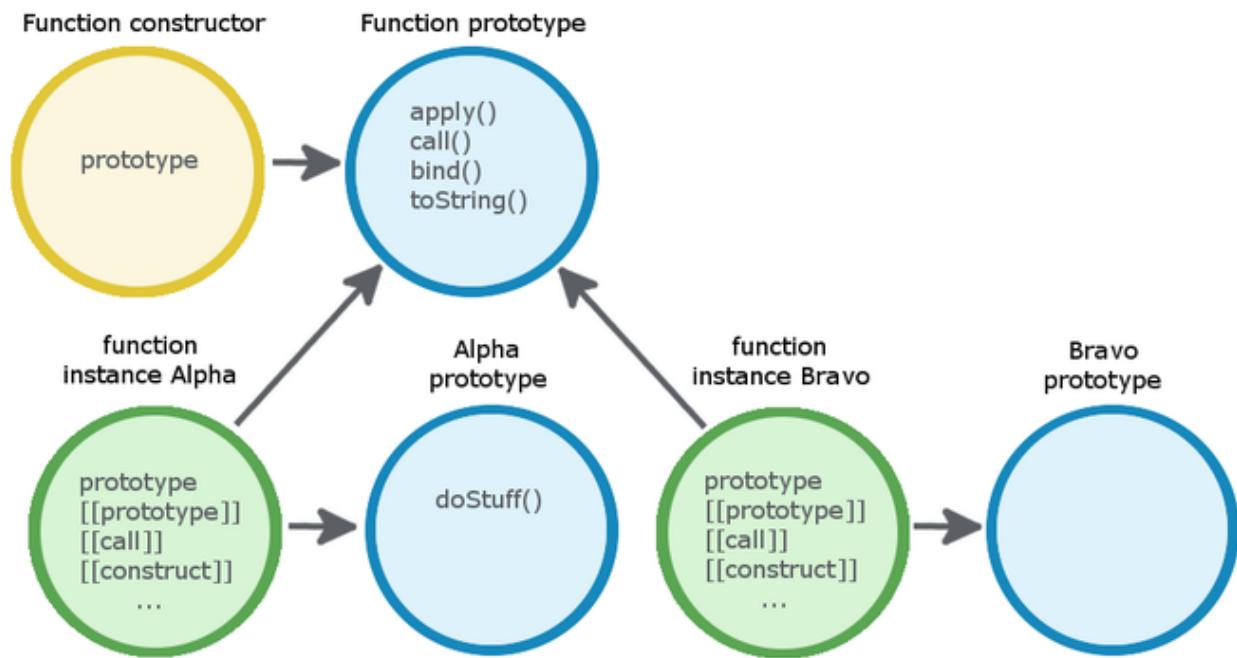
We can also define methods on their prototype objects by referencing the constructor function's `prototype` property,

```
Alpha.prototype.doStuff = function() { ... };
```

CODE

Our `Alpha` instance and any object instances created by it have access the prototype's shared methods.

After defining these two functions and augmenting `Alpha.prototype`, we have:



[3]

Note, internal properties are displayed inside double brackets.

The result is two function *instance objects*, referenced thru `Alpha` and `Bravo`. They link to the single `Function.prototype` thru their `[[prototype]]` property, giving them access to the prototype's useful methods, such as `.call()` and `.apply()`.

new Operator

Let's take this a step further. We invoke function instance object Alpha with the new operator, treating it as a *constructor function*.

```
var alpha1 = new Alpha();
var alpha2 = new Alpha();
```

CODE

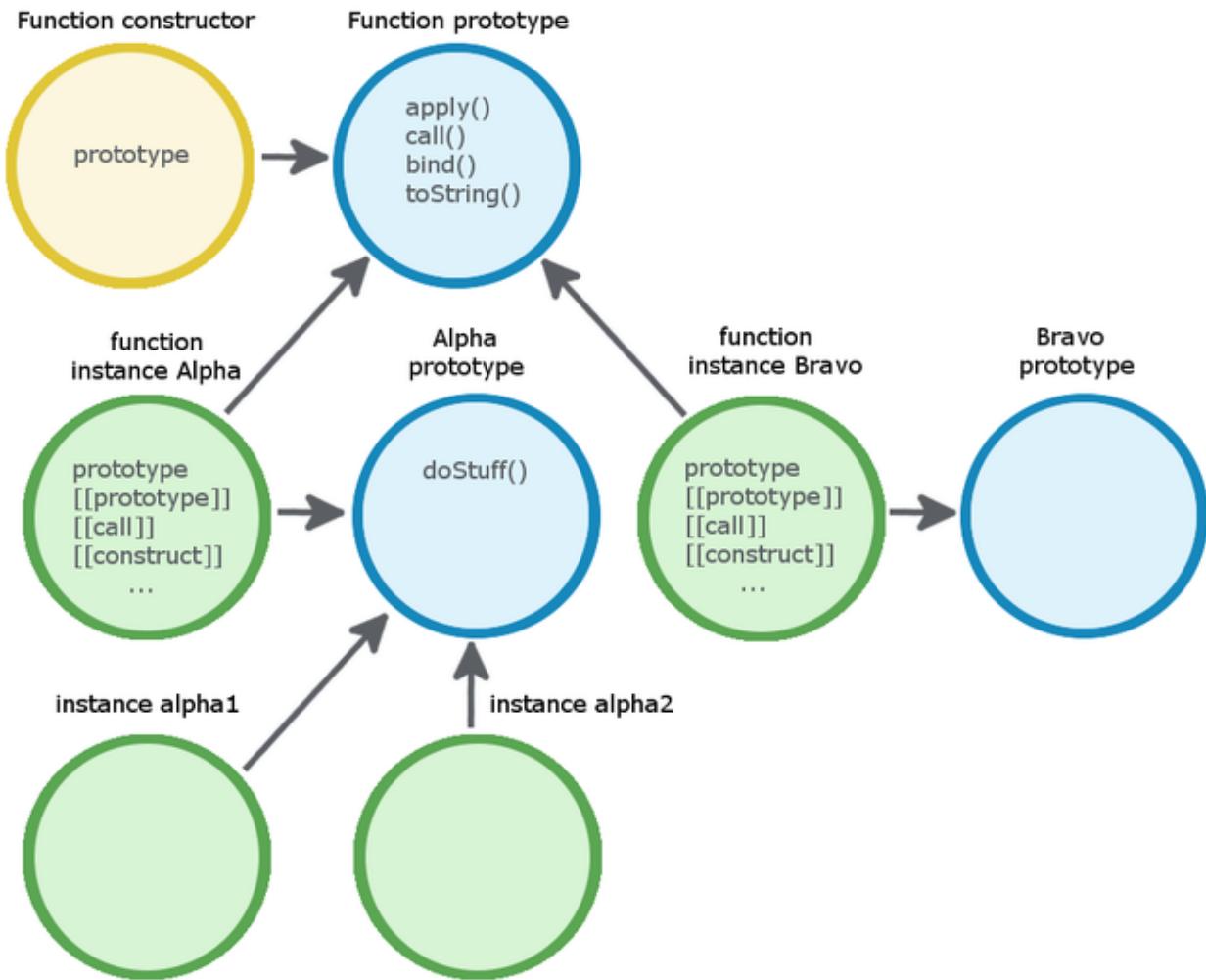
Similar steps as above occur, with some variation,

- A new empty instance object is created and passed into constructor Alpha as this .
- The new instance object is implicitly assigned many internal properties common to all objects, one being the [[prototype]] property. The Function constructor's **prototype** property referencing its prototype object is copied into the new object's [[prototype]] property^[3].
- Any this assignments are made to the new object.
- The new object is returned and assigned to its reference alpha1 and alpha2 .

[3]

In short, we used new to invoke Alpha as a constructor function, creating two instance objects referenced as alpha1 and alpha2 . The key differences here from when function object Alpha was created is alpha1 and alpha2 are objects but are not functions, so they don't have prototype objects.

And our objects in memory now look like:



Function Instance Properties

We briefly covered how function instance objects contains several internal properties and methods designed for its execution. It's worthwhile to have a general understanding of these:

Property	Value Description
prototype	Automatically created for every function, providing the possibility that the function will be used as a constructor[3]
[[Code]]	Function body code between { and }
[[FormalParameters]]	Function's parameters
[[BoundThis]]	this parameter referencing calling object
[[Call]]	Only functions have this property. It's invoked via () to internally execute function body code
[[Scope]]	Scope chain. Used for closures.
[[Class]]	Identifies object's internal type. Assigned "Function" [5]
[[Construct]]	Creates an object. Implemented in functions that can be invoked via the new operator such as all custom functions, and many built-in and host object

functions.

[6][7]

In this post we covered the depths of functions as objects, how function objects are created, what the `new` operator does, and internal properties and methods used in their execution.

I'd much enjoy your comments and questions. Feel free to reach out on [LinkedIn](#).

References

[1] - Resig, J., & Bibeault, B. (2013). Secrets of the JavaScript Ninja (p. 32). Shelter Island, NY: Manning.

[2] Crockford, D. (2008). JavaScript: The good parts (p. 3). Beijing: O'Reilly.

[3] ECMAScript 5.1 <http://www.ecma-international.org/ecma-262/5.1/#sec-13.2>

[4] - Flanagan, D. (2006). JavaScript: The definitive guide (5th ed., pp. 54-55). Cambridge: O'Reilly.

[5] ECMAScript 5.1 <http://www.ecma-international.org/ecma-262/5.1/#sec-15.3.3>

[6] ECMAScript 5.1 <http://www.ecma-international.org/ecma-262/5.1/#sec-15.3.5>

[7] ECMAScript 5.1 <http://www.ecma-international.org/ecma-262/5.1/#sec-8.6.2>

Type Checking Techniques in JavaScript: Part 1 of 2

<https://danmartensen.svbtle.com/type-checking-techniques-in-javascript-part-1>

January 5, 2015

Type Checking Techniques in JavaScript: Part 1 of 2

One of the early milestones in grocking any programming language is knowing how to use its data types. Should be easy for a small scripting language like JavaScript, right? And yet because of the elusive `var` keyword, implicit type conversion system, and dubious type checking tools, working with types can be a confusing task for even experienced developers.

Before we cover type checking in detail we must thoroughly *review data types in JavaScript*, the purpose of Part 1. In this short post I'll quickly cover the conceptual differences between primitive and object types, identify the types, and compare the *built-in*, *custom*, and *host* objects. Because this topic has been so muddled over the years I'm going to profusely site sources.

Primitives vs Objects

Like many languages, JavaScript has both primitive and object types. Let's explore the fundamental differences between these two:

	Primitive types	Object types
Has properties / methods?	No	Yes

Mutable values?	No	Yes
Memory allocation	Fixed size	Dynamic size
Variable holds	Actual data value	Memory address to object

[1]

Primitive Types

The current JavaScript language spec defines 6 primitive types [2]

- Number
- String
- Boolean
- Null
- Undefined
- Symbol

In action, examples are

```
var val;          // undefined is default variable value
val = 10000;      // number
val = 'woohoo';  // string
val = true;       // boolean
val = null;       // null
val = undefined; // can also be assigned
val = Symbol();   // symbol
```

CODE

It's worth mentioning here that variables are *dynamically typed*, able to be reassigned different types without explicit type casting.

Object type

All objects belong to a single type: the object type. JavaScript has many objects that fall into *three* categories within this single "object" type [2]:

Built-in Objects have properties and methods defined within the JavaScript spec. Most are instantiated through constructor functions using the `new` operator, while three are static: JSON, Math, and Global. All the built-ins are stored on the Global object during page load before any JavaScript code executes [3]. They are [2]

- Object object
- Array object
- Function object
- Date object
- RegExp object
- Wrapper objects: String object, Boolean object, Number object, Symbol
- Static objects: Math object, JSON object
- Error objects: Error, EvalError, RangeError, ReferenceError, SyntaxError, TypeError and URIError

- Global object

We use these often.

```
// instantiate built-in object
var date = new Date();
```

CODE

Custom Objects are the objects we define as developers. We can define them in 3 ways:

Constructor functions and prototypes

```
function Product(sku, status) {
    this.getSku = function() { return sku; };
    this.stockStatus = status;
}
Product.prototype.getStatus = function() {
    return this.stockStatus;
};
var product = new Product('ABC1234', 'InStock');
```

CODE

Object literals

```
var product = {
    sku: 'ABC1234',
    status: 'InStock',
    getSKU: function() { return this.sku; },
    getStatus: function() { return this.status; }
};
```

CODE

Object.create()

```
var product = Object.create(null, {
    sku: {
        value: 'ABC1234',
    },
    status: {
        value: 'InStock',
    },
    getSKU: {
        value: function() { return this.sku; },
        enumerable: true
    },
    getStatus: {
        value: function() { return this.status; },
        enumerable: true
    }
});
```

CODE

It's worth noting built-in and custom objects are also together called *Native objects* in the spec^[4]. Objects that are not native are called *Host objects*.

Host Objects are provided by the host (browser) environment. Like built-in objects they're stored on the Global window object. Most are static, unable to be instantiated with `new`, while others can be. Examples of host objects are:

- Document object (e.g. DOM)
- History object
- Location object
- Screen object
- XMLHttpRequest object (e.g. Ajax)

```
// instantiate Host object, from AngularJS source code
function createXhr() {
    return new window.XMLHttpRequest();
}
```

CODE

Primitive Wrapper Objects

You may have seen primitive types invoke object properties. Like it's distant cousin Java, JavaScript has the concept of **wrapper objects**. This allows variables holding number, string, or boolean primitive types to temporarily access their corresponding object type's properties and methods^[5].

```
var str = 'booyah'; // primitive string
str.length;        // accesses String object's property, returns 6
str.charAt(0);     // accesses String object's property, returns 'b'
```

CODE

We've covered the differences between primitive and object types, the three different categories of objects, and identified what those types are. Pretty straightforward, right? Now we're ready to explore the crooked paths of JavaScript's type checking tools.

If you have comments or questions, please feel free to reach out on [LinkedIn](#). I'd enjoy hearing from you.

References:

- [1] - Flanagan, D. (2006). JavaScript: The definitive guide (5th ed., pp. 54-55). Cambridge: O'Reilly.
- [2] - ECMAScript 5.1 spec, <http://www.ecma-international.org/ecma-262/5.1/#sec-4.2>
- [3] - ECMAScript 5.1 <http://www.ecma-international.org/ecma-262/5.1/#sec-15>
- [4] - ECMAScript 5.1 <http://www.ecma-international.org/ecma-262/5.1/#sec-4.3.6>
- [5] - Flanagan, D. (2006). JavaScript: The definitive guide (5th ed., pp. 40-41). Cambridge: O'Reilly.

19

Kudos

8. Type Checking Techniques in JavaScript Part 2 of 2

January 7, 2015

Type Checking Techniques in JavaScript Part 2 of 2

To get the most out of this article you'll want a solid grasp of JavaScript's data types including built-in, custom, and host objects. If you need a little refresher see [Part 1](#).

As front-end developers today are working with larger code bases, more 3rd party frameworks, and in teams, using effective type checking techniques is critical to writing bug free code. And you know how it goes, right? Less bugs means less Jira tickets reopened, less QA regressions, less work for project leads, and happier managers.

But the JavaScript language seems to be working against us. With its untyped `var` keyword, absence of typed parameters in function signatures, and implicit type conversions, combined with the lack of a unified `getType()` function, our need for reliable approaches is only increased.

In this article I'll cover three effective techniques for type checking and identify each of their advantages and disadvantages. Let's get started.

typeof operator

This first type checking technique is useful for *checking primitives types*. The `typeof` operator returns a string representation of the variable's type^[1].

The syntax is

```
typeof variable
```

CODE

In action

CODE

```

var val;
typeof val; // returns "undefined"

val = 10000;
typeof val; // returns "number"

val = 'woohoo';
typeof val; // returns "string"

val = true;
typeof val; // returns "boolean"

val = null;
typeof val; // returns "object", an ERROR

```

The laughable error is `typeof` on a `null` variable returns "object" when instead it should return "null", but it's too late to correct. Because of this we're forced to check for `null` by

CODE

```

if(val === null) {
  ...
}

```

`typeof` is of no use for distinguishing objects.

CODE

```

var val = {a:1, b:2}; // creates new Object object
typeof val; // returns "object"

val = [1,2,3]; // creates new Array object
typeof val; // returns "object"

val = new RegExp('a+b'); // creates new RegExp object
typeof val; // returns "object"

```

To muddy up the waters even more, it gives us a bonus value.

CODE

```

function doStuff(){} // function declaration
typeof doStuff; // returns "function"

```

But don't be misled, functions are not their own type. They're of the single "object" type^[2]. But this "function" value is still useful. We know that invoking an unknown variable can be risky because it'll throw a `TypeError` if it's not a function. `typeof` provides the safeguard to know whether the variable is a function before you invoke it.

CODE

```

function someFunc(param) {
  if(typeof param === 'function') {
    param();
  }
}

```

typeof summary

Pros	Cons
Simple way to identify primitive types	typeof on null variable returns "object".
Can be used as a safeguard to identify "functions" before invoking them	Misleads that function is a type when it's not
Differentiates primitive types from their object equivalent (e.g. 5 is not same as new Number(5))	All object types return "object". It doesn't distinguish between objects, such as an Array and RegExp

instanceof operator

This second technique is the *only way to identify custom objects* and *can be useful for identifying built-in objects*. It checks if a given object was created by a given constructor function[3].

The syntax is

```
object_reference instanceof ConstructorFunction
```

CODE

In action

```
// MyObj constructor function
function MyObj(){}
// create new MyObj
var myObj = new MyObj();

myObj instanceof MyObj; // returns true
myObj instanceof RegExp; // returns false
```

CODE

For better or worse, instanceof traverses up an object's prototype chain, checking if the object in question is an instance of the given constructor function or any of its parent objects.

CODE

```

function ParentObj(){} // parent constructor function
function ChildObj(){} // child constructor function

// set child to inherit from parent
ChildObj.prototype = new ParentObj;
ChildObj.prototype.constructor = ChildObj;

// instantiate instance of child
var child = new ChildObj();

// test if instance of child
child instanceof ChildObj // true

// test if instance of parent
child instanceof ParentObj // true

```

At times this behavior may be useful, but it's problematic. It leaves us with no way to determine which constructor function instantiated the object within its inheritance chain. By this one check we don't know if `child` was created by `ChildObj` or `ParentObj`.

instanceof summary

Pros	Cons
Only way to identify custom objects	Must know object's constructor function to check against
Also works for non-static built-in objects	Doesn't work for primitive types
Traverses prototype chain to check if instance of parent object	(Also a negative) Can't verify which constructor function instantiated object within inheritance chain

Object.prototype.toString

The third type checking technique is useful for *identifying built-in objects*.

Whenever an object is created it's assigned several internal properties under the hood. One of these is the `[[Class]]` property whose value is the kind of object it is. For built-in objects this value is the name of the object's constructor function^[5].

CODE

```

var array = [] // new array's internal [[Class]] is Array
var date = new Date(); // new date's internal [[Class]] is Date

```

We can use this unique, read-only property for type checking. The built-in Object's `toString()` method will return this internal `[[Class]]` property^[4]. We can invoke it with our object in question using the `call()` method located on every Function prototype.

CODE

```

var arr = [1,2,3];
Object.prototype.toString.call(arr); // returns "[object Array]"

var reg = new RegExp('a+b');
Object.prototype.toString.call(reg); // returns "[object RegExp]"

Object.prototype.toString.call(JSON); // returns "[object JSON]"

```

The string format `[[Class]]` is returned.

Interestingly, primitive types can be also identified using this approach.

CODE

```

var val = 1000; // primitive number
Object.prototype.toString.call(val); // returns "[object Number]"

val = 'abc'; // primitive string
Object.prototype.toString.call(val); // returns "[object String]"

val = true; // primitive boolean
Object.prototype.toString.call(val); // returns "[object Boolean]"

val = null; // primitive null
Object.prototype.toString.call(val); // returns "[object Null]"

val = undefined; // primitive undefined
Object.prototype.toString.call(val); // returns "[object Undefined]"

```

It's worth noting custom objects always return `[object Object]`, which isn't very useful. Use `instanceof` when checking those kinds of objects. The `[[Class]]` property values of host (browser) objects varies greatly per browser as it isn't specified in the language spec.

Object.prototype.toString() summary

Pros	Cons
Only way to identify all built-in objects	Verbose
Normalizes primitive types and their corresponding wrapper objects	(Also a negative) Does not differentiate whether a primitive or object

Summary

In this post we identified three reliable approaches to type checking, all based on what you're working with.

Primitives are best identified using `typeof` with the exception of the `null` value. It's checked by `val === null`.

`Object.prototype.toString` is used when wanting to normalize primitive numbers, strings, booleans with their wrapper object equivalents as being the same. **Built-in objects** are best identified using

`Object.prototype.toString` because it identifies types for both static and non-static built-in objects. **Custom objects**, which are objects defined by the developer, can only be identified using `instanceof`, though unfortunately there isn't a way to distinguish whether the object was constructed by a given constructor or its parent object's constructor. **Host objects** are best identified using `Object.prototype.toString` because most are static. Even

so, their values vary greatly between browser implementations.

I hope there was something of value for you in this post. I'd greatly enjoy your comments or questions. Please feel free to reach out on [LinkedIn](#).

References:

[1] - ECMAScript 5.1 spec, <http://www.ecma-international.org/ecma-262/5.1/#sec-11.4.3>

[2] - ECMAScript 5.1 spec, <http://www.ecma-international.org/ecma-262/5.1/#sec-4.3.24>

[3] - Resig, J., & Bibeault, B. (2013). Secrets of the JavaScript Ninja (p. 127). Shelter Island, NY: Manning.

[4] - ECMAScript 5.1 spec <http://www.ecma-international.org/ecma-262/5.1/#sec-15.2.4.2>

[5] - ECMAScript 5.1 spec <http://www.ecma-international.org/ecma-262/5.1/#sec-8.6.2>

Becoming a JavaScript ninja

July 28, 2014

[Becoming a JavaScript ninja](#)

I'm training really hard to become what some people call a "JavaScript Ninja". in this post I will share with you some important things that I have learned so far.

1. Use code conventions

Coding conventions are a set of guidelines for a specific programming language that recommend programming style, practices and methods for each aspect of a piece program written in this language. These conventions usually cover file organization, indentation, comments, declarations, statements, white space, naming conventions, programming practices, programming principles, programming rules of thumb, architectural best practices, etc. These are guidelines for software structural quality. Software programmers are highly recommended to follow these guidelines to help improve the readability of their source code and make software maintenance easier.

There are tools that will help you to ensure that you and your team follow JavaScript code conventions:

Code conventions tool	Description
JSLint	JSLint is a JavaScript program that looks for problems in JavaScript programs. It is a code quality tool. It was developed by Douglas Crockford. JSLint takes a JavaScript source and scans it. If it finds a problem, it returns a message describing the problem and an approximate location within the source. The problem is not necessarily a syntax error, although it often is. JSLint looks at some style conventions as well as structural problems. It does not prove that your program is correct. It just provides another set of eyes to help spot problems. You can download it from www.jslint.com .
JSHint	JSHint is fork of JSLint that was created by Anton Kovalyov because he believes that a code quality tool should be community-driven and because he thinks that

JSHint

sometimes we should be able to decide if we want to follow or not one code convention. As a result of this JS Hint is much more configurable than JS Lint. You can download it from www.jshint.com.

2. Document your code

I'm sure you are tired of listening to people that tell you that you should document your code. I'm sure you are doing it but sometimes it is not that easy to find value in doing it but if after creating the comments you end up with a documentation website, something like MSDN or the Java Documentation it seems to be more valuable, fortunately we also have tools that will generate the documentation for us:

Documentation generation tool	Description
JsDoc Toolkit	Is an application, written in JavaScript, for automatically generating template-formatted, multi-page HTML (or XML, JSON, or any other text-based) documentation from commented JavaScript source code. You can download it from here.

3. Separate concerns

In computer science, separation of concerns (SoC) is a design principle for separating a computer program into distinct sections, such that each section addresses a separate concern. A concern is a set of information that affects the code of a computer program.

The value of separation of concerns is simplifying development and maintenance of computer programs. When concerns are well separated, individual sections can be developed and updated independently. Of especial value is the ability to later improve or modify one section of code without having to know the details of other sections, and without having to make corresponding changes to those sections.

In a JavaScript application your concerns are HTML, CSS, JavaScript configuration code and JavaScript logic code. To keep them separated you just have to stick to the following rules:

a) Keep your HTML out of JavaScript: Embedding HTML strings inside your JavaScript is a bad practice.

Embedding HTML strings inside your JavaScript is a bad practice.

```
var div = document.getElementById("myDiv");
div.innerHTML = "<h3>Error</h3><p>Invalid email address.</p>";
```

CODE

The best way to avoid this problem is to load the HTML from the server via AJAX or even better load HTML client-side templates from the server. There are tools like handlebars.js that will help you to generate HTML in the client-side without embedding HTML strings inside your JavaScript.

CODE

```
// rendering logic
RenderJson = function(json, template, container) {
    var compiledTemplate = Handlebars.compile(template);
    var html = compiledTemplate(json);
    $(container).html('');
    $(container).html(html);
};

// template
var template = "{{#each hero}}<tr><td>{{this.name}}</td></tr>{{/each}}";

// model
var json = {
    hero : [
        { name : 'Batman' },
        { name : 'Superman' },
        { name : 'Spiderman' }
    ]
}

// Invoke rendering logic
RenderJson(json, template, $('#heroes_tbody'));

// DOM where we will insert HTML on rendering
<table>
    <thead><tr><th>Hero</th></tr></thead>
    <tbody id="heroes_tbody">
        <!-- rows added with JS -->
    </tbody>
</table>
```

b) Keep your CSS out of JavaScript

Don't change CSS rules from JavaScript, try to only work with CSS classes.

CODE

```
// bad
$(this).css( "color", "red" );

// good
$(this).addClass('redFont');
```

c) Keep your JavaScript out of CSS

Don't use CSS expressions, if you don't know what is a CSS expression (An IE8 and earlier feature) then you are in the right way.

d) Keep your CSS out of HTML

Don't use style tags to apply styles, use always class.

e) Keep your configuration code out of your logic code

Try to encapsulate all the hard coded variables and constants in a configuration object..

CODE

```

var CONFIG = {
  MESSAGE : {
    SUCCESS :"Success!"
  },
  DOM : {
    HEROE_LIST : "#heroes_tbody"
  }
};

// bad
RenderJson(json, template, $('#heroes_tbody'));

// good
RenderJson(json, template, $(CONFIG.DOM.HEROE_LIST));

```

If you do this finding the cause of issues will be much easier, imagine an incorrect background error, if you know that there is no JavaScript or HTML touching your CSS, then you automatically know that the issue must be in one of the CSS files, you just need to find the CSS class affected and you are done.

4. Avoid global variables

In computer programming, a global variable is a variable that is accessible in every scope. Global variables are a bad practices because it can lead to situations when one method is overriding and existing global variable and because code is harder to understand and mantain when we don't know where the variables has been declared. The best JavaScript code is the one where no global variable shas been declared. There are a few techniques that you can use to keep things local:

To avoid global one of the first things that you have to ensure is that all your JavaScript code is wrapped by a function. The easiest way of doing this is by using an inmmediate function and placing all of your script inside of the function.

CODE

```

(function(win) {
  "use strict"; // further avoid creating globals
  var doc = window.document;
  // declare your variables here
  // other code goes here
})(window);

```

The most common approcach to avoid globals is to create one unique global for your entire application think for example the \$ in Jquery. You can then use then a technique known as namespacing. Namespacing is simply a logical grouping of functions under a singleproperty of the global.

Sometimes each JavaScript file is sismply adding to a namespace, in this case, you should ensure that the namespace already exists.

CODE

```

var MyApp = {
    namespace: function(ns) {
        var parts = ns.split(".");
        object = this, i, len;
        for(i = 0, len = parts.length; i < len; i++) {
            if(!object[parts[i]]) {
                object[parts[i]] = {};
            }
            object = object[parts[i]];
        }
        return object;
    }
};

// declare a namespace
MyApp.namespace("Helpers.Parsing");

// you can now start using the namespace
MyApp.Helpers.Parsing.DateParser = function() {
    //do something
};

```

Another technique used by developers to avoid globals is the encapsulation in modules. A module is a generic piece of functionality that creates no new globals or namespaces. Instead all the code takes place within a single function that is responsible for executing a task or publishing an interface. The most common type of JavaScript modules is Asynchronous Module Definition (AMD).

CODE

```

//define
define( "parsing", //module name
    [ "dependency1", "dependency2" ], // module dependencies
    function( dependency1, dependency2 ) { //factory function

        // Instead of creating a namespace AMD modules
        // are expected to return their public interface
        var Parsing = {};
        Parsing.DateParser = function() { //do something };
        return Parsing;
    }
);

// load a AMD module with Require.js
require(["parsing"], function(Parsing) {
    Parsing.DateParser(); // use module
});

```

5. Avoid Null comparisons

The special value null is often misunderstood and confused with undefined. This value should be used in just a few cases:

- a) To initialize a variable that may later be assigned an object value
- b) To compare against an initialized variable that may or may not have an object value

c) To pass into a function where an object is expected**d) To return from a function where an object is expected**

There are cases in which null should not be used:

a) Do not use null to test whether an argument was supplied.**b) Do not test an uninitialized variable for the value null.**

The special value undefined is frequently confused with null. Part of the confusion is that null == undefined is true. However, these two values have two very different uses. Variables that are not initialized have an initial value of undefined.

```
//bad
var person;
console.log(person == undefined); //true
```

CODE

The general recommendation is to avoid using undefined at all times.

I guess that you must be know wondering how should you do the following without using undefined or null?

```
//bad
function doSomething(arg){
    if (arg != null) {
        soSomethingElse();
    }
}
```

CODE

Comparing a variable against null doesn't give you enough information about the value to determine whether is safe to proceed. Fortunately, JavaScript gives you a few ways to determine the true value of a variable:

a) Primitive values: If your expecting a value to be a primitive type (string, number, boolean) then the typeof operator is your best option.

```
// detect a number
if(typeof count === "number") {
    //do something
}
```

CODE

b) Reference values: The instanceof operator is the best way to detect objects of a particular reference type.

```
// detect a date
if(value instanceof Date) {
    //do something
}
```

CODE

****c) Functions:** Using typeof is the best way to detect functions.

CODE

```
// detect a function
if(MyApp.Helpers.Parsing.DateParser typeof === "function") {
    MyApp.Helpers.Parsing.DateParser();
}
```

d) Arrays: The best way to detect arrays is to use the isArray() function. The only problem is that isArray does not work in old versions of internet explorer. But you can sue the following code for cross browser support.

CODE

```
function isArray(value) {
    if (typeof Array.isArray === "function") {
        return Array.isArray(value);
    } else {
        return Object.prototype.toString.call(value) === "[object array]";
    }
}
```

e) Properties: The best way to detect a property is to use the in operator or the hasOwnProperty() function.

CODE

```
var hero = { name : 'superman '};
//check property
if (name in hero) {
    // do something
}
//check NOT inherited property
if (hero.hasOwnProperty('name')) {
    // do something
}
```

6. Handle errors

Throwing custom errors in JavaScript can help you to decrease your debugging time. It is not easy to know when you should throw a custom error but in general errors should be thrown only in the deepest part of your application stack. Any code that handles application-especific logig should have error-handling capabilities. You can use the following code to create your custom errors:

CODE

```
function MyError(message){
    this.message = message;
}
MyError.prototype = new Error(); //extending base error class
```

Is also a good idea to chek for specifict error types (Error, EvalError, RangeError, ReferenceError, SyntaxError, TypeError, URIError) to have a more robust error handling:

CODE

```

try {
    // Do something
} catch(ex) {
    if(ex instanceof TypeError) {
        // Handle error
    } else if (ex instanceof ReferenceError) {
        // Handle error
    } else {
        //Handle all others
    }
}

```

7. Don't modify objects that you don't own

There are things that you don't own (objects that has not been written by you or your team)

- a) **Native objects** (e.g. Object, Array, etc.)
- b) **DOM objects** (e.g. document)
- c) **Browser object model** (e.g. window)
- d) **Library objects** (e.g. Jquery, \$, _, Handlebars, etc.)

And thing that you should not try on objects that you don't own:

- a) **Don't override methods**
- b) **Don't add new methods**
- c) **Don't remove existing methods**

If you really need to extend or modify an object that you don't own, you should create a class that inherits from it and modify your new object. You can use one of the JavaScript inheritance basic forms:

a) Object-based Inheritance: an object inherits from another without calling a constructor function.

CODE

```

var person = {
    name : "Bob",
    sayName : function() {
        alert(this.name);
    }
}
var myPerson = Object.create(person);
myPerson.sayName(); // Will display "Bob"

```

a) Type-based inheritance: typically requires two steps: prototypal inheritance and then constructor inheritance.

```
function Person(name) {  
    this.name = name;  
}  
  
function Author(name) {  
    Person.call(this, name); // constructor inheritance  
}  
  
Author.prototype = new Person(); // prototypal inheritance
```

8. Test everything

As the complexity of JavaScript applications grows we must introduce the same design patterns and practices that we have been using for years in our server-side and desktop applications code to ensure high quality solutions. So it is time to start writing tests (unit, performance, integration...) for your JavaScript code. The good news is that there are several tools that you can use to help you:

- a) [Jasmine](#)
- b) [JsTestDrive](#)
- c) [PhantomJS](#)
- d) [QUnit](#)
- e) [Selenium](#)
- f) [Yeti](#)
- g) [YUI Test](#)

9. Automate everything

Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly.

Continuous Integration doesn't get rid of bugs, but it does make them dramatically easier to find and remove. In this respect it's rather like self-testing code. If you introduce a bug and detect it quickly it's far easier to get rid of.

Continuous Integration has been used for a while together with TDD (Test Driven Development). But it was more traditionally linked to server-side code. In the previous point we said that it is time to test our JavaScript code. In this point I want to highlight that it is also time to continuously integrate it.

Build

In a professional development environment you will normally find the following types of build:

a) Development Build: Run by developers as they are working, it should be as fast as possible to not interrupt productivity.

b) Integration Build: An automated build that runs on a regular schedule. These are sometimes run for each commit, but on large projects, they tend to run on intervals for a few minutes.

c) Release Build: an on-demand build that is run prior to production push.

Continuous integration

There are several continuous integration servers but the most popular is Jenkins, the most popular build tool is Apache Ant. The process of building your JavaScript code includes several tasks:

a) Test automation As discussed on point 8 you should use test automation tools for testing your JavaScript code.

b) Validation You should add code quality validation to your build process. You can use tools like JSLint or JSHint.

c) Concatenation You should join all your JavaScript files in one single file.

d) Baking You can perform tasks like adding a code the license or version code as part of the build.

e) Minification You should integrate as part the build the minification by tools like uglify.js.

f) Compression You should gzip your JavaScript as part of your build.

g) Documentation You should integrate as part of your build the auto-generation of the documentation by tools like JS Doc Toolkit.

10. Find a great master

A real ninja never stops learning, so you better find a great master!

Where can you find one? The answer is of course the Internet. I recommend to read the blogs of some of the chrome or mozilla developers about javascript as well as other js libraries like jquery.

75

Kudos

75

Kudos

About object-oriented design and the "class" & "extends" keywords in TypeScript / ES6

May 23, 2016

About object-oriented design and the "class" & "extends" keywords in TypeScript / ES6

A few weeks ago I found an interesting article titled [In Defense of JavaScript Classes](#). The article exposed some concerns about the `class` keyword in ES6 / TypeScript:

These days it feels like everyone is attacking classes in JavaScript. Developers I respect say [ES6](#)

[classes are a virus](#). We've compiled long lists on the reasons that [ES6 classes are not awesome](#). Apparently, if we're still brave enough to try them, we need advice on [how to use classes and still sleep at night](#).

The problems that I see with the `class` and `extends` keywords in ES6 / TypeScript are not something new. I believe that these problems are caused by bad object-oriented (OO) design and I'm sure that the source of most of the criticism is coming from programmers with a strong interest in functional programming and I understand their fears. They are afraid of some of the OOP "monsters":

- **Inheritance**
- **Internal classes state**
- **Object composition**

I'm going to write a really small TypeScript application to showcase some of these "monsters" and try to explain how to "fight" them.

Let's imagine that our business domain has three entities: `Samurai`, `Sword` and `Material`.

We also have some business rules:

- `Samurai` students use a `Bokken` (wood sword).
- `Samurai` master use a `Katana` (iron sword).

Inheritance

Dan Abramov wrote the following in his article [How to Use Classes and Sleep at Night](#):

Classes encourage inheritance but you should prefer composition.

When we don't follow this recommendation we end up with really bad code like the following:

CODE

```
class Material {
    public name: string;
}

class Iron extends Material {
    constructor() {
        super();
        this.name = "iron";
    }
}

class Wood extends Material {
    constructor() {
        super();
        this.name = "wood";
    }
}

class Sword {
    material: Material;
}

class Katana extends Sword {
    constructor() {
        super();
        this.material = new Iron();
    }
}

class Bokken extends Sword {
    constructor() {
        super();
        this.material = new Iron();
    }
}

class Samurai {
    public sword: Sword;
}

class SamuraiMaster extends Samurai {
    constructor() {
        super();
        this.sword = new Katana();
    }
}

class SamuraiStudent extends Samurai {
    constructor() {
        super();
        this.sword = new Bokken();
    }
}

let master = new SamuraiMaster();
let student = new SamuraiStudent();
```

The preceding code snippet is really wrong because it doesn't follow the core OOP principles:

- [The SOLID principles](#)
- [The composite reuse principle](#)

We should be very careful with the `extends` keyword. Because when a class extends another we are coupling the two forever as inheritance is the strongest kind of coupling that we can encounter.

However, I believe that is not a bad thing that the `extends` keyword is available. We just need to be very careful and use it only in very limited cases.

Internal class state

Let's imagine that a material has a level of endurance from 0 to 100:

```
class Sword {
    material: Material;
    constructor(material: Material) {
        this.material = material;
    }
    use() {
        if (this.material.endurance > 0) {
            this.material.endurance = this.material.endurance - 10;
            return this.material.power;
        }
        return 0;
    }
}
```

CODE

As we use a weapon its endurance will decrease. The endurance is part of the class internal state. The problem of internal state in classes is that predicting the output of its method becomes a difficult task. Being able to predict the output of a method is important because predictability and testability are directly proportional.

For example, let imagine that you are looking to a piece of code in the application that invokes the `use` method:

```
function doSomething(weapon: IWeapon) {
    return weapon.use(); // Will it cause any damage?
}
```

CODE

We need to know the class internal state to be able to predict the value returned by its methods. We can re-write the method as a pure function:

```
function use(endurance, power) {
    if (endurance > 0) {
        endurance = endurance - 10;
        return power;
    }
    return 0;
}
```

CODE

Pure functions/methods don't use the internal class state. This makes much easier to predict their return type:

CODE

```
function doSomething(weapon: IWeapon, material: IMaterial) {
    return weapon.use(material.endurance, material.power);
}
```

In real life the classes would have some methods. We should **try to be as declarative as possible** when implementing these methods. How can you be more declarative? You need to learn about functional programming and use libraries like [Ramda](#).

Let's take a look to an example. The following code filters a list of tasks by one of its properties to find the list of incomplete tasks:

```
// Plain JS
var incompleteTasks = tasks.filter(function(task) {
    return !task.complete;
});

// Lo-Dash
var incompleteTasks = _.filter(tasks, {complete: false});
```

CODE

Both examples mention the actual data collection `tasks`. We have declared **how** to iterate the items and filter each item (imperative) as opposed to declare **what** we want the code to do (declarative). With Ramda it would look as the following:

```
let getIncomplete = R.filter(R.where({complete: false}));
```

CODE

As we can see the data collection is not mentioned and the API is much more declarative. This makes this function highly re-usable.

We should also try to **declare "pure methods"**. In other words, we should declare methods that are independent of the internal class state (properties) if possible. If we can't do this we should try to **make our classes immutable when possible**. This means that the properties of a class will not change after an instance has been created.

Object composition

The following example adheres to the composite reuse and SOLID principles:

CODE

```

class Material {
    public name: string;
    constructor(string) {
        this.name = string;
    }
}

class Sword {
    material: Material;
    constructor(material: Material) {
        this.material = material;
    }
}

class Samurai {
    public sword: Sword;
    constructor(sword: Sword) {
        this.sword = sword;
    }
}

function getSamurai(rank) {
    if(rank === "master") {
        return new Samurai(new Sword(new Material("iron")));
    } else {
        return new Samurai(new Sword(new Material("wood")));
    }
}

```

The problem with the preceding code snippet is that the `getSamurai` function is in charge of the object composition and it uses an imperative style.

The `getSamurai` function can be easily broken by small changes. Let's try to imagine that we have a new requirement:

- Samurai generals use Katana made of gold.

It is just a small change but the function will become even more imperative and harder to maintain:

CODE

```

function getSamurai(rank) {
    if (rank === "master") {
        return new Samurai(new Sword(new Material("iron")));
    } else if (rank === "general"){
        return new Samurai(new Sword(new Material("gold")));
    } else {
        return new Samurai(new Sword(new Material("wood")));
    }
}

```

This is one of the main reasons to use an IoC container. An IoC container is a tool that helps us to avoid coupling and control the way object composition works in our application. In this section I'm going to use [InversifyJS](#).

We are going to start by declaring the types available in our application as string literals. We do this because the typescript types are not available at run-time but we need them to identify the dependencies of a class:

CODE

```
let TYPES = {
    materialName: "materialName",
    IMaterial: "IMaterial",
    ISword: "ISword",
    ISamurai: "ISamurai"
};
```

We can then declare some interfaces:

CODE

```
interface IMaterial {
    name: string;
}

interface ISword {
    material: IMaterial;
}

interface ISamurai {
    sword: ISword;
}
```

At this point we can declare our classes:

CODE

```
@injectable()
class Material implements IMaterial {
    public name: string;
    constructor(@inject(TYPES.materialName) string) {
        this.name = string;
    }
}

@Injectable()
class Sword implements ISword {
    material: IMaterial;
    constructor(@inject(TYPES.IMaterial) material: IMaterial) {
        this.material = material;
    }
}

@Injectable()
class Samurai implements ISamurai {
    public sword: ISword;
    constructor(@inject(TYPES.ISword) sword: ISword) {
        this.sword = sword;
    }
}
```

Now that our entities are defined, it is time to use our IoC container to declare some rules that will define the way the object composition works in our application:

CODE

```

let kernel = new Kernel();

kernel.bind<ISword>(TYPES.ISword).to(Sword);
kernel.bind<ISamurai>(TYPES.ISamurai).to(Samurai);
kernel.bind<IMaterial>(TYPES.IMaterial).to(Samurai);

kernel.bind<string>(TYPES.materialName)
    .toConstantValue("iron")
    .whenAnyAncestorTagged("rank", "master");

kernel.bind<string>(TYPES.materialName)
    .toConstantValue("wood")
    .whenAnyAncestorTagged("rank", "student");

```

I believe that the preceding code snippet is much more maintainable than:

CODE

```

function getSamurai(rank) {
    if (rank === "master") {
        return new Samurai(new Sword(new Material("iron")));
    } else {
        return new Samurai(new Sword(new Material("wood")));
    }
}

```

It is also a bit more verbose, but I prefer maintainable/verbose over unmaintainable/concise. Of course, this is just my opinion...

Let's imagine once more that we have a new requirement:

- Samurai generals use a Katana made of gold.

We can just add a new contextual binding:

CODE

```

kernel.bind<string>(TYPES.materialName)
    .toConstantValue("iron")
    .whenAnyAncestorTagged("rank", "general");

```

The nice thing about this is that we don't need to modify the existing code! At this point you can create a new samurai master, student or general and it will own a sword with the right material.

CODE

```

let general = kernel.getTagged<ISamurai>(
    TYPES.ISamurai, "rank", "general");

```

I would also like to mention that we are working an [alternative binding API for InversifyJS based on decorators](#). This API is even more declarative. Instead of declaring the bindings using the fluent syntax:

CODE

```

kernel.bind<IMaterial>(TYPES.IMaterial)
    .to(Wood)
    .whenAnyAncestorTagged("rank", "student");

```

We can declare bindings using decorators:

```
@provideWhenAnyAncestorTagged(TYPES.IMaterial, "rank", "student")
class Wood {
    // ...
```

CODE

We could implement the application as follows:

```
@provideWhenAnyAncestorTagged(TYPES.IMaterial, "rank", "master")
class Iron implements IMaterial {
    // ....
}

@provideWhenAnyAncestorTagged(TYPES.IMaterial, "rank", "student")
class Wood implements IMaterial {
    // ....
}

@provide(TYPES.ISword)
class Sword implements ISword {
    material: IMaterial;
    constructor(@inject(TYPES.IMaterial) material: IMaterial) {
        this.material = material;
    }
}

@provide(TYPES.ISamurai)
class Samurai implements ISamurai {
    public sword: ISword;
    constructor(@inject(TYPES.ISword) sword: ISword) {
        this.sword = sword;
    }
}
```

CODE

Let's imagine that the above is our code and we are asked to introduce the "general rule" once more. We could solve this problem by adding a new material and no other changes would be required in the entire application:

```
@provideWhenAnyAncestorTagged(TYPES.IMaterial, "rank", "general")
class Gold implements IMaterial {
    // ....
}
```

CODE

Summary

I like the new JavaScript/TypeScript OOP features introduced by ES6 but we need to be careful with them (just like in any other OOP language...).

If you want to write good OO code I encourage you to:

- Master the core OOP design principles.
- Gain a basic understanding of functional programming principles (pure functions, side effects...).
- Be very careful with inheritance and use it only in very limited cases.

- Try to be as declarative as possible.
- Try to declare "pure methods".
- Try to use immutable classes.
- Learn how to use IoC containers.

I believe that an IoC container can help us to write better OO code and that is why I've been working on [InversifyJS](#).

Please feel free to share thoughts about this article with us via [@OweR_ReLoaDeD](#), [@WolkSoftwareLtd](#) or [@InversifyJS](#).

[Don't forget to subscribe](#) if you don't want to miss out future articles!

-
40

Kudos

-
40

Kudos

10 Interview Questions Every JavaScript Developer Should Know AKA: The Keys to JavaScript Mastery

At most companies, management must trust the developers to give technical interviews in order to assess candidate skills. If you do well as a candidate, you'll eventually need to interview. Here's how.

You Might Not Agree, and That's OK

I advise people to hire based on whether or not a developer believes in class inheritance. Why? Because people who love it are obstinately stubborn about it. They will go to their graves clutching to it. As Bruce Lee famously said:

*"Those who are unaware
they are walking in darkness
will never seek the light."*

I won't hire them. You shouldn't, either.

[I've seen classes wreak havoc on projects, companies, and lives](#). Many brilliant people disagree with my views.

However, before dismissing my opinion out of hand, consider these points:

I ran an app consulting firm in the early days of SaaS, starting before the term was coined. I have worked on hundreds of projects for both startups and fortune 500 companies. I have a background in C++/Java (and assembly, AutoLisp, Delphi, etc...), and I consulted on dozens of Java and C++ apps (both class-oriented OOP languages). I was once an avid supporter of classical inheritance, and even wrote multi-language Rapid Application Development (RAD) tools for it.

I saw up close the many ways in which the class paradigm invited unwary developers down the wrong road, and I saw the extremely costly effects. I've seen products abruptly discontinued to stop losses, programmers laid off, and

companies brought to their knees by brittle, tangled codebases caused primarily by the improper use of classes and class inheritance.

Class in JS is not harmless sugar for prototypal OO. Class is a virus that infects everything it touches. It came to us formally in JavaScript with ES6, and at the same time, React was taking off. Lots of people started using classes for React components (you don't have to: the new React 0.14 supports pure function components, or try [react-stamp](#)).

Many are unaware that you can build React components in a class-free style. This has caused confusion and incompatibility between React components and other composable elements in the React ecosystem (components, mixins, and component wrappers).

During the days of Backbone dominance (Backbone used its own flavor of classes), I watched a once malleable codebase transform into a brittle mess. I saw code complicated by abstractions necessary to unify the calling API of classes that require `new`, and factories that don't, forcing the use of dependency injection container that ended up coupling every dependent module tightly to the container API. Those of you with Angular experience will understand what I'm talking about. Angular has four different ways to create services, and features a dependency injection container to abstract them all.

A screenshot of a Google search results page. The search bar contains the query "object oriented considered harmful". Below the search bar, there are navigation links for "Web", "News", "Images", "Videos", "Shopping", "More", and "Search tools". A red banner below the search bar displays the text "About 171,000 results (0.29 seconds)". The first result is a link to a website titled "Object Oriented Programming is Inherently Harmful" with the URL "harmful.cat-v.org/software/OO_programming/". A snippet of the page content states: "'Object-oriented programming is an exceptionally bad idea which could only have originated in ... Object-Oriented Considered Harmful by Frans Faase.'". The second result is a link to a website titled "Object-Oriented Considered Harmful - I write, therefore I am" with the URL "www.iwriteiam.nl/AoP_OOCH.html". A snippet of the page content states: "Object-Oriented Considered Harmful. By Frans. Most people praise Object-Oriented".

Many brilliant people warned of the perils of class inheritance decades ago, before JavaScript was invented. You can find their warnings in essays with titles like "Object Oriented Programming Considered Harmful", "Class Considered Harmful", and "New Considered Harmful". Such warnings have been published on Usenet, in academic papers, and respected publications such as Doctor Dobb's Journal: a magazine that gave us nearly 40 years of software development wisdom before its sunset at the end of 2014. Incidentally, there's also an essay called "Considered Harmful Considered Harmful", but I digress.

Joe Armstrong, the creator of Erlang summed up one of the famous problems with class in what has become known as The Gorilla Banana Problem from the great book, "[Coders at Work](#)" (buy it, there's lots of other great stuff in it):

"The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle."

The seminal tome of OO design, "[Design Patterns: Elements of Reusable Object-Oriented Software](#)"--the book from which design patterns get their name--implores:

*"Favor object composition
over class inheritance."*

Edit:

Some readers are asking about the class rant. Learn more about why I don't like classes in JavaScript:

- * [The Two Pillars of JavaScript: How to Escape the 7th Circle of Hell](#)
- * [A Simple Challenge to Classical Inheritance Fans](#)
- * [Common Misconceptions About Inheritance in JavaScript](#)
- * [How to Fix the ES6 `class` Keyword](#)
- * [Introducing the Stamp Specification](#)

If you're one of those obstinate classical inheritance fans, you may disagree with lots of this article, but there are still some gems here for you, too. Take what you like, toss the rest. I won't be offended.

It Starts With People

In ["How to Build a High Velocity Development Team"](#), I made a couple points worth repeating:

"Nothing predicts business outcomes better than an exceptional team. If you're going to beat the odds, you need to invest here, first."

As Marcus Lemonis says, focus on the 3 P's:

"People, Process, Product"

Your early hires should be very strong, senior-level candidates. People who can hire and mentor other developers, and help the mid-level and junior developers you'll eventually want to hire down the road.

Read ["Why Hiring is So Hard in Tech"](#) for a good breakdown of the general do's and don'ts of candidate evaluation.

*The best way to evaluate a candidate
is a pair programming exercise.*

Pair program with the candidate. Let the candidate drive. Watch and listen more than you talk. A good project might be to pull tweets from the Twitter API and display them on a timeline.

That said, no single exercise will tell you everything you need to know. An interview can be a very useful tool as well, but don't waste time asking about syntax or language quirks. You need to see the big picture. Ask about architecture and paradigms--the big decisions that can have a major impact on the whole project.

Syntax and features are easy to Google. It's much harder to Google for software engineering wisdom or the common paradigms and idioms JavaScript developers pick up with experience.

JavaScript is special, and it plays a critical role in almost every large application. What is it about JavaScript that makes it meaningfully different from other languages?

Here are some questions that will help you explore the stuff that really matters:

1. Can you name two programming paradigms important for JavaScript app developers?

JavaScript is a multi-paradigm language, supporting **imperative/procedural** programming along with **OOP** (Object-Oriented Programming) and **functional programming**. JavaScript supports OOP with **prototypal inheritance**.

Good to hear:

- Prototypal inheritance (also: prototypes, OOO).
- Functional programming (also: closures, first class functions, lambdas).

Red flags:

- No clue what a paradigm is, no mention of prototypal oo or functional programming.

Learn More:

- [The Two Pillars of JavaScript Part 1](#)--Prototypal OO.
- [The Two Pillars of JavaScript Part 2](#)--Functional Programming.

2. What is functional programming?

Functional programming produces programs by composing mathematical functions and avoids shared state & mutable data. Lisp (specified in 1958) was among the first languages to support functional programming, and was heavily inspired by lambda calculus. Lisp and many Lisp family languages are still in common use today.

Functional programming is an essential concept in JavaScript (one of the two pillars of JavaScript). Several common functional utilities were added to JavaScript in ES5.

Good to hear:

- Pure functions / function purity.
- Avoid side-effects.
- Simple function composition.
- Examples of functional languages: Lisp, ML, Haskell, Erlang, Clojure, Elm, F Sharp, OCaml, etc...
- Mention of features that support FP: first-class functions, higher order functions, functions as arguments/values.

Red flags:

- No mention of pure functions / avoiding side-effects.
- Unable to provide examples of functional programming languages.
- Unable to identify the features of JavaScript that enable FP.

Learn More:

- [The Two Pillars of JavaScript Part 2](#).
- [The Dao of Immutability](#).
- [Professor Frisby's Mostly Adequate Guide to Functional Programming](#).
- [The Haskell School of Music](#).

3. What is the difference between classical inheritance and prototypal inheritance?

Class Inheritance: instances inherit from classes (like a blueprint--a description of the class), and create sub-class

relationships: hierarchical class taxonomies. Instances are typically instantiated via constructor functions with the `new` keyword. Class inheritance may or may not use the `class` keyword from ES6.

Prototypal Inheritance: instances inherit directly from other objects. Instances are typically instantiated via factory functions or `Object.create()`. Instances may be composed from many different objects, allowing for easy selective inheritance.

In JavaScript, prototypal inheritance is simpler & more flexible than class inheritance.

Good to hear:

- Classes: create tight coupling or hierarchies/taxonomies.
- Prototypes: mentions of concatenative inheritance, prototype delegation, functional inheritance, object composition.

Red Flags:

- No preference for prototypal inheritance & composition over class inheritance.

Learn More:

- [The Two Pillars of JavaScript Part 1](#)--Prototypal OO.
- [Common Misconceptions About Inheritance in JavaScript](#).

4. What are the pros and cons of functional programming vs object-oriented programming?

OOP Pros: It's easy to understand the basic concept of objects and easy to interpret the meaning of method calls. OOP tends to use an imperative style rather than a declarative style, which reads like a straight-forward set of instructions for the computer to follow.

OOP Cons: OOP Typically depends on shared state. Objects and behaviors are typically tacked together on the same entity, which may be accessed at random by any number of functions with non-deterministic order, which may lead to undesirable behavior such as race conditions.

FP Pros: Using the functional paradigm, programmers avoid any shared state or side-effects, which eliminates bugs caused by multiple functions competing for the same resources. With features such as the availability of point-free style (aka tacit programming), functions tend to be radically simplified and easily recomposed for more generally reusable code compared to OOP.

FP also tends to favor declarative and denotational styles, which do not spell out step-by-step instructions for operations, but instead concentrate on **what** to do, letting the underlying functions take care of the **how**. This leaves tremendous latitude for refactoring and performance optimization, even allowing you to replace entire algorithms with more efficient ones with very little code change. (e.g., memoize, or use lazy evaluation in place of eager evaluation.)

Computation that makes use of pure functions is also easy to scale across multiple processors, or across distributed computing clusters without fear of threading resource conflicts, race conditions, etc...

FP Cons: Over exploitation of FP features such as point-free style and large compositions can potentially reduce readability because the resulting code is often more abstractly specified, more terse, and less concrete.

More people are familiar with OO and imperative programming than functional programming, so even common idioms in functional programming can be confusing to new team members.

FP has a much steeper learning curve than OOP because the broad popularity of OOP has allowed the language and learning materials of OOP to become more conversational, whereas the language of FP tends to be much more academic and formal. FP concepts are frequently written about using idioms and notations from lambda calculus, algebras, and category theory, all of which requires a prior knowledge foundation in those domains to be understood.

Good to hear:

- Mentions of trouble with shared state, different things competing for the same resources, etc...
- Awareness of FP's capability to radically simplify many applications.
- Awareness of the differences in learning curves.
- Articulation of side-effects and how they impact program maintainability.
- Awareness that a highly functional codebase can have a steep learning curve.
- Awareness that a highly OOP codebase can be extremely resistant to change and very brittle compared to an equivalent FP codebase.
- Awareness that immutability gives rise to an extremely accessible and malleable program state history, allowing for the easy addition of features like infinite undo/redo, rewind/replay, time-travel debugging, and so on. Immutability can be achieved in either paradigm, but a proliferation of shared stateful objects complicates the implementation in OOP.

Red flags:

- Unable to list disadvantages of one style or another--Anybody experienced with either style should have bumped up against some of the limitations.

Learn More:

- [The Two Pillars of JavaScript Part 1](#)--Prototypal OO.
- [The Two Pillars of JavaScript Part 2](#)--Functional Programming.

5. When is classical inheritance an appropriate choice?

This is a trick question. The answer is never. I've been issuing this challenge for years, and the only answers I've ever heard fall into one of several [common misconceptions](#). More frequently, the challenge is met with silence.

*"If a feature is sometimes useful
and sometimes dangerous
and if there is a better option
then always use the better option."
~ Douglas Crockford*

Good to hear:

- Rarely, almost never, or never.
- "Favor object composition over class inheritance."

Red flags:

- Any other response.
- "React Components"--no, **the pitfalls of class inheritance don't change just because a new framework**

comes along and embraces the `class` keyword. Contrary to popular awareness, you don't need to use `class` to use React. This answer reveals a misunderstanding of both `class` and React.

Learn More:

- [The Two Pillars of JavaScript Part 1](#)--Prototypal OO.
- [JS Objects--Inherited a Mess](#).

6. When is prototypal inheritance an appropriate choice?

There is more than one type of prototypal inheritance:

- **Delegation** (i.e., the prototype chain).
- **Concatenative** (i.e. mixins, `Object.assign()`).
- **Functional** (Not to be confused with functional programming. A function used to create a closure for private state/encapsulation).

Each type of prototypal inheritance has its own set of use-cases, but all of them are equally useful in their ability to enable **composition**, which creates **has-a** or **uses-a** or **can-do** relationships as opposed to the **is-a** relationship created with class inheritance.

Good to hear:

- In situations where modules or functional programming don't provide an obvious solution.
- When you need to compose objects from multiple sources.
- Any time you need inheritance.

Red flags:

- No knowledge of when to use prototypes.
- No awareness of mixins or `Object.assign()`.

Learn More:

- ["Programming JavaScript Applications": Prototypes section](#).

7. What does "favor object composition over class inheritance" mean?

This is a quote from ["Design Patterns: Elements of Reusable Object-Oriented Software"](#). It means that code reuse should be achieved by assembling smaller units of functionality into new objects instead of inheriting from classes and creating object taxonomies.

In other words, use **can-do**, **has-a**, or **uses-a** relationships instead of **is-a** relationships.

Good to hear:

- Avoid class hierarchies.
- Avoid brittle base class problem.
- Avoid tight coupling.
- Avoid rigid taxonomy (forced is-a relationships that are eventually wrong for new use cases).
- Avoid the gorilla banana problem ("what you wanted was a banana, what you got was a gorilla holding the banana, and the entire jungle").
- Make code more flexible.

Red Flags:

- Fail to mention any of the problems above.
- Fail to articulate the difference between composition and class inheritance, or the advantages of composition.

Learn More:

[Introducing
the Stamp Specification](#)
[Move Over, `class`:](#)
[Composable Factory Functions Are Heremedium.com](#)

8. What are two-way data binding and one-way data flow, and how are they different?

Two way data binding means that UI fields are bound to model data dynamically such that when a UI field changes, the model data changes with it and vice-versa.

One way data flow means that the model is the single source of truth. Changes in the UI trigger messages that signal user intent to the model (or "store" in React). Only the model has the access to change the app's state. The effect is that data always flows in a single direction, which makes it easier to understand.

One way data flows are deterministic, whereas two-way binding can cause side-effects which are harder to follow and understand.

Good to hear:

- React is the new canonical example of one-way data flow, so mentions of React are a good signal. Cycle.js is another popular implementation of uni-directional data flow.
- Angular is a popular framework which uses two-way binding.

Red flags:

- No understanding of what either one means. Unable to articulate the difference.

Learn more:**9. What are the pros and cons of monolithic vs microservice architectures?**

A monolithic architecture means that your app is written as one cohesive unit of code whose components are designed to work together, sharing the same memory space and resources.

A microservice architecture means that your app is made up of lots of smaller, independent applications capable of running in their own memory space and scaling independently from each other across potentially many separate machines.

Monolithic Pros: The major advantage of the monolithic architecture is that most apps typically have a large number of cross-cutting concerns, such as logging, rate limiting, and security features such audit trails and DOS protection.

When everything is running through the same app, it's easy to hook up components to those cross-cutting concerns.

There can also be performance advantages, since shared-memory access is faster than inter-process communication (IPC).

Monolithic cons: Monolithic app services tend to get tightly coupled and entangled as the application evolves,

making it difficult to isolate services for purposes such as independent scaling or code maintainability.

Monolithic architectures are also much harder to understand, because there may be dependencies, side-effects, and magic which are not obvious when you're looking at a particular service or controller.

Microservice pros: Microservice architectures are typically better organized, since each microservice has a very specific job, and is not concerned with the jobs of other components. Decoupled services are also easier to recompose and reconfigure to serve the purposes of different apps (for example, serving both the web clients and public API).

They can also have performance advantages depending on how they're organized because it's possible to isolate hot services and scale them independent of the rest of the app.

Microservice cons: As you're building a new microservice architecture, you're likely to discover lots of cross-cutting concerns that you did not anticipate at design time. A monolithic app could establish shared magic helpers or middleware to handle such cross-cutting concerns without much effort.

In a microservice architecture, you'll either need to incur the overhead of separate modules for each cross-cutting concern, or encapsulate cross-cutting concerns in another service layer that all traffic gets routed through.

Eventually, even monolithic architectures tend to route traffic through an outer service layer for cross-cutting concerns, but with a monolithic architecture, it's possible to delay the cost of that work until the project is much more mature.

Microservices are frequently deployed on their own virtual machines or containers, causing a proliferation of VM wrangling work. These tasks are frequently automated with container fleet management tools.

Good to hear:

- Positive attitudes toward microservices, despite the higher initial cost vs monolithic apps. Aware that microservices tend to perform and scale better in the long run.
- Practical about microservices vs monolithic apps. Structure the app so that services are independent from each other at the code level, but easy to bundle together as a monolithic app in the beginning. Microservice overhead costs can be delayed until it becomes more practical to pay the price.

Red flags:

- Unaware of the differences between monolithic and microservice architectures.
- Unaware or impractical about the additional overhead of microservices.
- Unaware of the additional performance overhead caused by IPC and network communication for microservices.
- Too negative about the drawbacks of microservices. Unable to articulate ways in which to decouple monolithic apps such that they're easy to split into microservices when the time comes.
- Underestimates the advantage of independently scalable microservices.

10. What is asynchronous programming, and why is it important in JavaScript?

Synchronous programming means that, barring conditionals and function calls, code is executed sequentially from top-to-bottom, blocking on long-running tasks such as network requests and disk I/O.

Asynchronous programming means that the engine runs in an event loop. When a blocking operation is needed, the request is started, and the code keeps running without blocking for the result. When the response is ready, an interrupt is fired, which causes an event handler to be run, where the control flow continues. In this way, a single

program thread can handle many concurrent operations.

User interfaces are asynchronous by nature, and spend most of their time waiting for user input to interrupt the event loop and trigger event handlers.

Node is asynchronous by default, meaning that the server works in much the same way, waiting in a loop for a network request, and accepting more incoming requests while the first one is being handled.

This is important in JavaScript, because it is a very natural fit for user interface code, and very beneficial to performance on the server.

Good to hear:

- An understanding of what blocking means, and the performance implications.
- An understanding of event handling, and why its important for UI code.

Red flags:

- Unfamiliar with the terms asynchronous or synchronous.
- Unable to articulate performance implications or the relationship between asynchronous code and UI code.

Learn more:

Conclusion

That's it for the interview. Stick to high-level topics. If they can answer these questions, that typically means that they have enough programming experience to pick up language quirks & syntax in a few weeks, even if they don't have a lot of JavaScript experience.

Don't disqualify candidates based on stuff that's easy to learn (including classic CS-101 algorithms, or any type of puzzle problem).

What you really need to know is, "does this candidate understand how to put an application together?"

[Level up your skills with courses & webcasts on ES6, TDD, prototypal OO, React. \\$495 Lifetime Access](#)

Eric Elliott is the author of "[Programming JavaScript Applications](#)" (O'Reilly), and "[Learn JavaScript Universal App Development with Node, ES6, & React](#)". He has contributed to software experiences for **Adobe Systems, Zumba Fitness, The Wall Street Journal, ESPN, BBC**, and top recording artists including **Usher, Frank Ocean, Metallica, and many more**.

He spends most of his time in the San Francisco Bay Area with the most beautiful woman in the world.

9. Common Misconceptions About Inheritance in JavaScript

WAT? [wat]--interjection: A sound a programmer makes when something violates the principle of least astonishment by astonishing them with counter-intuitive behavior.

```
> .1 + .2
0.3000000000000004
> WAT? OMG! STFU! STUPID JAVASCRIPT!!!
...

```

Also, WAT? is the sound I make when I talk to many seasoned JavaScript developers who have neglected to learn the basic mechanics of prototypal inheritance: one of the most important innovations in CS history, and one of the [Two Pillars of JavaScript](#).

To me, this is like a professional photographer who has yet to learn the exposure triangle--the basic formula for controlling much of the visual style of a photograph. Put simply:

Photo by Anthony Aceldy (CC BY-NC 2.0)

*If you don't understand prototypes,
you don't understand JavaScript.*

Aren't classical inheritance and prototypal inheritance really the same thing, just a stylistic preference?

No.

Classical and prototypal inheritance are **fundamentally and semantically distinct**.

There are some **defining characteristics** between classical inheritance and prototypal inheritance. For any of this article to make sense, you must keep these points in mind:

In **class inheritance**, **instances inherit from a blueprint** (the class), and **create sub-class relationships**. In other words, you can't use the class like you would use an instance. **You can't invoke instance methods on a class definition itself**. You must first create an instance and then invoke methods on that instance.

In prototypal inheritance, **instances inherit from other instances**. Using **delegate prototypes** (setting the prototype of one instance to refer to an **exemplar object**), it's literally **Objects Linking to Other Objects**, or **OLOO**, as Kyle Simpson calls it. Using **concatenative inheritance**, you just **copy properties** from an **exemplar object** to a new instance.

It's really important that you understand these differences. Class inheritance by virtue of its mechanisms **create class hierarchies as a side-effect of sub-class creation**. Those hierarchies lead to **arthritic code** (hard to change) and **brittleness** (easy to break due to rippling side-effects when you modify base classes).

Prototypal inheritance does not necessarily create similar hierarchies. I recommend that you keep prototype chains as shallow as possible. It's easy to flatten many prototypes together to form a **single delegate prototype**.

TL;DR:

- A **class** is a **blueprint**.
- A **prototype** is an **object** instance.

Aren't classes the right way to create objects in JavaScript?

No.

There are several **right ways** to create objects in JavaScript. The first and most common is an object literal. It looks like this (in ES6):

```
// ES6 / ES2015, because 2015.

let mouse = {
  furColor: 'brown',
  legs: 4,
  tail: 'long, skinny',
  describe () {
    return `A mouse with ${this.furColor} fur,
      ${this.legs} legs, and a ${this.tail}.`;
  }
};
```

CODE

Of course, object literals have been around a lot longer than ES6, but they lack the method shortcut seen above, and you have to use `var` instead of `let`. Oh, and that template string thing in the `describe()` method won't work in ES5, either.

You can attach delegate prototypes with `Object.create()` (an ES5 feature):

```
let animal = {
  animalType: 'animal',

  describe () {
    return `An ${this.animalType}, with ${this.furColor} fur,
      ${this.legs} legs, and a ${this.tail}.`;
  }
};

let mouse = Object.assign(Object.create(animal), {
  animalType: 'mouse',
  furColor: 'brown',
  legs: 4,
  tail: 'long, skinny'
});
```

CODE

Let's break this one down a little. `animal` is a **delegate prototype**. `mouse` is an instance. When you try to access a property on `mouse` that isn't there, the JavaScript runtime will look for the property on `animal` (the delegate).

`**Object.assign()**` is a new ES6 feature championed by Rick Waldron that was previously implemented in a few dozen libraries. You might know it as `\$.extend()` from jQuery or `_.extend()` from Underscore. Lodash has a version of it called `assign()`. You pass in a destination object, and as many source objects as you like, separated by commas. It will copy all of the **enumerable own properties** by *assignment* from the source objects to the destination objects with **last in priority**. If there are any property name conflicts, the version from the last object passed in wins.

`**Object.create()**` is an ES5 feature that was championed by Douglas Crockford so that we could attach delegate

prototypes without using constructors and the `new` keyword.

I'm skipping the constructor function example because I can't recommend them. I've seen them abused a lot, and I've seen them cause [a lot of trouble](#). It's worth noting that a lot of smart people disagree with me. Smart people will do whatever they want.

Wise people will **take Douglas Crockford's advice**:

"If a feature is sometimes dangerous, and there is a better option, then always use the better option."

Don't you need a constructor function to specify object instantiation behavior and handle object initialization?

No.

Any function can create and return objects. When it's not a constructor function, it's called a **factory function**.

The Better Option

CODE

```
let animal = {
  animalType: 'animal',
  describe () {
    return `An ${this.animalType} with ${this.furColor} fur,
      ${this.legs} legs, and a ${this.tail} tail.`;
  }
};

let mouseFactory = function mouseFactory () {
  return Object.assign(Object.create(animal), {
    animalType: 'mouse',
    furColor: 'brown',
    legs: 4,
    tail: 'long, skinny'
  });
};

let mickey = mouseFactory();
```

I usually don't name my factories "factory"--that's just for illustration. Normally I just would have called it `mouse()`.

Don't you need constructor functions for privacy in JavaScript?

No.

In JavaScript, any time you export a function, that function has access to the outer function's variables. When you use them, the JS engine creates a **closure**. Closures are a common pattern in JavaScript, and they're commonly used for data privacy.

Closures are not unique to constructor functions. Any function can create a closure for data privacy:

```

let animal = {
  animalType: 'animal',
  describe () {
    return `An ${this.animalType} with ${this.furColor} fur,
    ${this.legs} legs, and a ${this.tail} tail.`;
  }
};

let mouseFactory = function mouseFactory () {
  let secret = 'secret agent';

  return Object.assign(Object.create(animal), {
    animalType: 'mouse',
    furColor: 'brown',
    legs: 4,
    tail: 'long, skinny',
    profession () {
      return secret;
    }
  });
};

let james = mouseFactory();

```

Does `new` mean that code is using classical inheritance?

No.

The `new` keyword is used to invoke a constructor. What it actually does is:

- Create a new instance
- Bind `this` to the new instance
- Reference the new object's delegate [[Prototype]] to the object referenced by the constructor function's `prototype` property.
- Names the object type after the constructor, which you'll notice mostly in the debugging console. You'll see `[Object Foo]`, for example, instead of `[Object object]`.
- Allows `instanceof` to check whether or not an object's prototype reference is the same object referenced by the `.prototype` property of the constructor.

`instanceof` lies

Let's pause here for a moment and reconsider the value of `instanceof`. You might change your mind about its usefulness.

Important: `instanceof` does not do type checking the way that you expect similar checks to do in strongly typed languages. Instead, it does an identity check on the prototype object, and it's easily fooled. It won't work across execution contexts, for instance (a common source of bugs, frustration, and unnecessary limitations). For reference, an [example in the wild, from bacon.js](#).

It's also easily tricked into false positives (and more commonly) false negatives from another source. Since it's an identity check against a target object's `.prototype` property, it can lead to strange things:

```
> function foo() {}
> var bar = { a: 'a'};
> foo.prototype = bar; // Object {a: "a"}
> baz = Object.create(bar); // Object {a: "a"}
> baz instanceof foo // true. oops.
```

That last result is completely in line with the JavaScript specification. Nothing is broken--it's just that `instanceof` can't make any guarantees about type safety. It's easily tricked into reporting both **false positives**, and **false negatives**.

Besides that, trying to force your JS code to behave like strongly typed code can block your functions from being lifted to generics, which are much more reusable and useful.

`instanceof` limits the reusability of your code, and potentially introduces bugs into the programs that use your code.

`instanceof` lies.

Ducktype instead.

`'new'` is weird

WAT? `'new'` also does some *weird stuff* to return values. If you try to return a primitive, it won't work. If you return any other arbitrary object, that **does** work, but `this` gets thrown away, breaking all references to it (including `.call()` and `.apply()`), and breaking the link to the constructor's `.prototype` reference.

Is There a Big Performance Difference Between Classical and Prototypal Inheritance?

No.

You may have heard of **hidden classes**, and think that constructors dramatically outperform objects instantiated with `Object.create()`. Those performance differences are **dramatically overstated**.

A small fraction of your application's time is spent running JavaScript, and a minuscule fraction of that time is spent accessing properties on objects. In fact, the slowest laptops being produced today can access *millions of properties per second*.

That's not your app's bottleneck. Do yourself a favor and [profile your app](#) to **discover your real performance bottlenecks**. I'm sure there are a million things you should fix before you spend another moment thinking about micro-optimizations.

Not convinced? For a micro-optimization to have any appreciable impact on your app, you'd have to loop over the operation **hundreds of thousands of times**, and the only differences in micro-optimization you should ever be concerned about are the ones that are **orders of magnitude apart**.

Rule of thumb: Profile your app and eliminate as many loading, networking, file I/O, and rendering bottlenecks as you can find. **Then and only then should you start to think about a micro-optimization.**

Can you tell the difference between `.000000001` seconds and `.000000001` seconds? Neither can I, but I sure can tell the difference between loading 10 small icons or loading one web font, instead!

If you do **profile your app** and find that object creation really is a bottleneck, the fastest way to do it is not by using `'new'` and classical OO. **The fastest way is to use object literals.** You can do so in-line with a loop and add objects to an object pool to avoid thrashing from the garbage collector. If it's worth abandoning prototypal OO over perf, it's worth ditching the prototype chain and inheritance altogether to crank out object literals.

But Google said class is fast...

WAT? Google is building a JavaScript engine. You are building an application. Obviously what they care about and what you care about should be **very different things**. Let Google handle the micro-optimizations. You worry about **your app's real bottlenecks**. I promise, you'll get a whole lot better ROI focusing on just about anything else.

Is There a Big Difference in Memory Consumption Between Classical and Prototypal?

No.

Both can use delegate prototypes to share methods between many object instances. Both can use or avoid wrapping a bunch of state into closures.

In fact, if you start with factory functions, it's easier to switch to object pools so that you can manage memory more carefully and avoid being blocked periodically by the garbage collector. For more on why that's awkward with constructors, see the **WAT?** note under "*Does 'new' mean that code is using classical inheritance?*"

In other words, if you want the most flexibility for memory management, use factory functions instead of constructors and classical inheritance.

*"...if you want the most flexibility for memory management,
use factory functions..."*

The Native APIs use Constructors. Aren't they More Idiomatic than Factories?

No.

Factories are extremely common in JavaScript. For instance, the most popular JavaScript library of all time, **jQuery** exposes a factory to users. John Resig has written about the choice to use a factory and prototype extension rather than a class. Basically, it boils down to the fact that he didn't want callers to have to type `'new'` every time they made a selection. What would that have looked like?

```
/*
classy jQuery - an alternate reality where jQuery really sucked and never took off
OR
Why nobody would have liked jQuery if it had exported a class instead of a factory.
*/

// This just looks stupid. Are we creating a new DOM element
// with id="foo"? Nope. We're selecting an existing DOM element
// with id="foo", and wrapping it in a jQuery object instance.
var $foo = new $('#foo');

// Besides, it's a lot of extra typing with literally ZERO gain.
var $bar = new $('.bar');
var $baz = new $('.baz');

// And this is just... well. I don't know what.
var $bif = new $('.foo').on('click', function () {
  var $this = new $(this);
  $this.html('clicked!');
});
```

What else exposes factories?

- **React** `React.createClass()` is a factory.
- **Angular** uses classes & factories, but wraps them all with a factory in the Dependency Injection container. All providers are sugar that use the `provider()` factory. There's even a `factory()` provider, and even the `service()` provider wraps normal constructors and exposes ... you guessed it: A **factory** for DI consumers.
- **Ember** `Ember.Application.create();` is a factory that produces the app. Rather than creating constructors to call with `new`, the `extend()` methods augment the app.
- **Node** core services like `http.createServer()` and `net.createServer()` are factory functions.
- **Express** is a factory that creates an express app.

As you can see, virtually all of the most popular libraries and frameworks for JavaScript make heavy use of factory functions. **The only object instantiation pattern more common than factories in JS is the object literal.**

JavaScript built-ins started out using constructors because Brendan Eich was told to make it look like Java. JavaScript continues to use constructors for self-consistency. It would be awkward to try to change everything to factories and deprecate constructors now.

That doesn't mean that your APIs have to suck.

Isn't Classical Inheritance More Idiomatic than Prototypal Inheritance?

No.

Every time I hear this misconception I am tempted to say, "**do u even JavaScript?**" and move on... but I'll resist the urge and set the record straight, instead.

Don't feel bad if this is your question, too. It's **not your fault**. [JavaScript Training Sucks!](#)

The answer to this question is a big, gigantic

No... (but)

Prototypes are the idiomatic inheritance paradigm in JS, and `class` is the marauding invasive species.

A brief history of popular JavaScript libraries:

In the beginning, everybody wrote their own libs, and open sharing wasn't a big thing. And then **Prototype** came along. (The name is a big hint here). Prototype did its magic by extending built-in **delegate prototypes** using **concatenative inheritance**.

Later we all realized that modifying built-in prototypes was an anti-pattern when native alternatives and conflicting libs broke the internet. But that's a different story.

Next on the JS lib popularity roller coaster was **jQuery**. jQuery's big claim to fame was **jQuery plugins**. They worked by extending jQuery's **delegate prototype** using **concatenative inheritance**.

Are you starting to sense a pattern here?

jQuery remains the most popular JavaScript library ever made. By a HUGE margin. HUGE.

This is where things get muddled and class extension starts to sneak into the language... John Resig (author of jQuery) wrote about *Simple Class Inheritance in JavaScript*, and people started *actually using it*, even though John Resig himself didn't think it belonged in jQuery (because **prototypal OO did the same job better**).

Semi-popular Java-esque frameworks like ExtJS appeared, ushering in the first kinda, sorta, not-really mainstream uses of class in JavaScript. This was 2007. JavaScript was **12 years old** before a somewhat popular lib started exposing JS users to classical inheritance.

Three years later, Backbone **exploded** and had an `extend()` method that mimicked class inheritance, including all its nastiest features such as brittle object hierarchies. That's when *all hell broke loose*.

~100kloc app starts using Backbone. A few months in I'm debugging a 6-level hierarchy trying to find a bug. Stepped through every line of constructor code up the `super` chain. Found and fixed the bug in the top level base class. Then had to fix a lot of child classes because they depended on the buggy behavior of the base class. Hours of frustration that should have been a 5 minute fix.

This is not JavaScript. I was suddenly living in *Java hell* again. That lonely, dark, scary place where any quick movements could cause entire hierarchies to shudder and collapse in coalescing, tight-coupled convulsions.

These are the monsters rewrites are made of.

But, squirreled away in the Backbone docs, a ray of golden sunshine:

```
// A ray of sunshine in the belly of
// the beast...

var object = {};

_.extend(object, Backbone.Events);

object.on("alert", function(msg) {
  alert("Triggered " + msg);
});

object.trigger("alert", "an event");
```

Our old friend, **concatenative inheritance** saving the day with a `Backbone.Events` mixin.

It turns out, if you look at **any non-trivial JavaScript library** closely enough, **you're going to find examples of concatenation and delegation**. It's so common and automatic for JavaScript developers to do these things that **they don't even think of it as inheritance**, even though it **accomplishes the same goal**.

*Inheritance in JS is so easy
it confuses people who expect it to take effort.*

To make it harder, we added 'class'.

And how did we add class? **We built it on top of prototypal inheritance using delegate prototypes and object concatenation**, of course!

That's like driving your Tesla Model S to a car dealership and trading it in for a rusted out 1983 Ford Pinto.

Doesn't the Choice Between Classical and Prototypal Inheritance Depend on the Use Case?

No.

Prototypal OO is simpler, more flexible, and a lot less error prone. I have been making this claim and challenging people to come up with a compelling class use case *for many years*. Hundreds of thousands of people have heard the call. The few answers I've received depended on one or more of the misconceptions addressed in this article.

I was once a classical inheritance fan. I bought into it completely. I built object hierarchies everywhere. I built visual OO Rapid Application Development tools to help software architects design object hierarchies and relationships that made sense. It took a visual tool to truly map and graph the object relationships in enterprise applications using classical inheritance taxonomies.

Soon after my transition from C++ and Java to JavaScript, I stopped doing all of that. Not because I was building less complex apps (the opposite is true), but because JavaScript was so much simpler, I had no more need for all that OO design tooling.

I used to do application design consulting and frequently recommend sweeping rewrites. Why? Because **all object**

hierarchies are eventually wrong for new use cases.

I wasn't alone. In those days, **complete rewrites** were very common for new software versions. Most of those rewrites were necessitated by legacy lock-in caused by [arthritic, brittle class hierarchies](#). Entire books were written about OO design mistakes and how to avoid them or refactor away from them. It seemed like every developer had a copy of "[Design Patterns](#)" on their desk.

I recommend that you follow the [Gang of Four's advice](#) on this point:

"Favor object composition over class inheritance."

In Java, that was harder than class inheritance because you actually had to use classes to achieve it.

In JavaScript, we don't have that excuse. It's actually **much easier** in JavaScript to simply create the object that you need by assembling various **prototypes** together than it is to manage object hierarchies.

WAT? Seriously. Want the jQuery object that can turn any date input into a `megaCalendarWidget`? You don't have to `extend` a `class`. JavaScript has dynamic object extension, and jQuery exposes its own prototype so you can just extend that--without an extend keyword! **WAT?:**

```
/*
How to extend the jQuery prototype:
So difficult.
Brain hurts.
ouch.
*/
jQuery.fn.megaCalendarWidget = megaCalendarWidget;

// omg I'm so glad that's over.
```

CODE

The next time you call the jQuery factory, you'll get an instance that can make your date inputs mega awesome.

Similarly, you can use `Object.assign()` to compose any number of objects together with last-in priority:

```
import ninja from 'ninja'; // ES6 modules
import mouse from 'mouse';

let ninjamouse = Object.assign({}, mouse, ninja);
```

CODE

No, really--**any number of objects**:

```
// I'm not sure Object.assign() is available (ES6)
// so this time I'll use Lodash. It's like Underscore,
// with 200% more awesome. You could also use
// jQuery.extend() or Underscore's _.extend()
var assign = require('lodash/object/assign');

var skydiving = require('skydiving');
var ninja = require('ninja');
var mouse = require('mouse');
var wingsuit = require('wingsuit');

// The amount of awesome in this next bit might be too much
// for seniors with heart conditions or young children.
var skydivingNinjaMouseWithWingsuit = assign({}, // create a new object
skydiving, ninja, mouse, wingsuit); // copy all the awesome to it.
```

This technique is called **concatenative inheritance**, and the prototypes you inherit from are sometimes referred to as **exemplar prototypes**, which differ from delegate prototypes in that you **copy from them**, rather than delegate to them.

ES6 Has the `class` keyword. Doesn't that mean we should all be using it?

No.

There are lots of [compelling reasons](#) to [avoid the ES6 `class` keyword](#), not least of which because it's an awkward fit for JavaScript.

We already have an **amazingly powerful and expressive object system** in JavaScript. The concept of class as it's implemented in JS today is more restrictive (in a bad way, not in a cool type-correctness way), and obscures the [very cool prototypal OO system](#) that was built into the language a long time ago.

You know what would really be good for JavaScript? Better sugar and abstractions built on top of prototypes **from the perspective of a programmer familiar with prototypal OO**.

That could be [really cool](#).

*I'm creating a whole on-line class on
Prototypal OO in JavaScript.*

[Preorder Now](#)
*for Lifetime Access to all
my JavaScript courses.*

Eric Elliott is the author of "[Programming JavaScript Applications](#)" (O'Reilly), & host of the documentary film-in-production, "[Programming Literacy](#)". He has contributed to software experiences for **Adobe Systems**, **Zumba Fitness**, **The Wall Street Journal**, **ESPN**, **BBC**, and top recording artists including **Usher**, **Frank Ocean**, **Metallica**, and many more.

He spends most of his time in the San Francisco Bay Area with the most beautiful woman in the world.

10. JavaScript Modules: A Beginner's Guide

If you're a newcomer to JavaScript, jargon like "module bundlers vs. module loaders," "Webpack vs. Browserify" and "AMD vs. CommonJS" can quickly become overwhelming.

The JavaScript module system may be intimidating, but understanding it is vital for web developers.

In this post, I'll unpack these buzzwords for you in plain English (and a few code samples). I hope you find it helpful!

Note: for simplicity's sake, this will be divided into two sections: Part 1 will dive into explaining what modules are and why we use them. Part 2 (posted next week) will walk through what it means to bundle modules and the different ways to do so.

Part 1: Can someone please explain what modules are again?

Good authors divide their books into chapters and sections; good programmers divide their programs into modules.

Like a book chapter, modules are just clusters of words (or code, as the case may be).

Good modules, however, are highly self-contained with distinct functionality, allowing them to be shuffled, removed, or added as necessary, without disrupting the system as a whole.

Why use modules?

There are a lot of benefits to using modules in favor of a sprawling, interdependent codebase. The most important ones, in my opinion, are:

1) Maintainability: By definition, a module is self-contained. A well-designed module aims to lessen the dependencies on parts of the codebase as much as possible, so that it can grow and improve independently. Updating a single module is much easier when the module is decoupled from other pieces of code.

Going back to our book example, if you wanted to update a chapter in your book, it would be a nightmare if a small change to one chapter required you to tweak every other chapter as well. Instead, you'd want to write each chapter in such a way that improvements could be made without affecting other chapters.

2) Namespacing: In JavaScript, variables outside the scope of a top-level function are global (meaning, everyone can access them). Because of this, it's common to have "namespace pollution", where completely unrelated code shares global variables.

Sharing global variables between unrelated code is a big [no-no in development](#).

As we'll see later in this post, modules allow us to avoid namespace pollution by creating a private space for our variables.

3) Reusability: Let's be honest here: we've all copied code we previously wrote into new projects at one point or another. For example, let's imagine you copied some utility methods you wrote from a previous project to your current project.

That's all well and good, but if you find a better way to write some part of that code you'd have to go back and

remember to update it everywhere else you wrote it.

This is obviously a huge waste of time. Wouldn't it be much easier if there was--wait for it--a module that we can reuse over and over again?

How can you incorporate modules?

There are many ways to incorporate modules into your programs. Let's walk through a few of them:

Module pattern

The Module pattern is used to mimic the concept of classes (since JavaScript doesn't natively support classes) so that we can store both public and private methods and variables inside a single object--similar to how classes are used in other programming languages like Java or Python. That allows us to create a public facing API for the methods that we want to expose to the world, while still encapsulating private variables and methods in a closure scope.

There are several ways to accomplish the module pattern. In this first example, I'll use an anonymous closure. That'll help us accomplish our goal by putting all our code in an anonymous function. (Remember: in JavaScript, functions are the only way to create new scope.)

Example 1: Anonymous closure

CODE

```
(function () {
    // We keep these variables private inside this closure scope

    var myGrades = [93, 95, 88, 0, 55, 91];

    var average = function() {
        var total = myGrades.reduce(function(accumulator, item) {
            return accumulator + item}, 0);

        return 'Your average grade is ' + total / myGrades.length + '.';
    }

    var failing = function(){
        var failingGrades = myGrades.filter(function(item) {
            return item < 70;});

        return 'You failed ' + failingGrades.length + ' times.';
    }

    console.log(failing());

}());

// 'You failed 2 times.'
```

With this construct, our anonymous function has its own evaluation environment or "closure", and then we immediately evaluate it. This lets us hide variables from the parent (global) namespace.

What's nice about this approach is that is that you can use local variables inside this function without accidentally overwriting existing global variables, yet still access the global variables, like so:

CODE

```
var global = 'Hello, I am a global variable :)';

(function () {
  // We keep these variables private inside this closure scope

  var myGrades = [93, 95, 88, 0, 55, 91];

  var average = function() {
    var total = myGrades.reduce(function(accumulator, item) {
      return accumulator + item}, 0);

    return 'Your average grade is ' + total / myGrades.length + '.';
  }

  var failing = function(){
    var failingGrades = myGrades.filter(function(item) {
      return item < 70;});

    return 'You failed ' + failingGrades.length + ' times.';
  }

  console.log(failing());
  console.log(global);
}());

// 'You failed 2 times.'
// 'Hello, I am a global variable :)'
```

Note that the parenthesis around the anonymous function are required, because statements that begin with the keyword *function* are always considered to be function declarations (remember, you can't have unnamed function declarations in JavaScript.) Consequently, the surrounding parentheses create a function expression instead. If you're curious, you can [read more here](#).

Example 2: Global import

Another popular approach used by libraries like [jQuery](#) is global import. It's similar to the anonymous closure we just saw, except now we pass in globals as parameters:

CODE

```
(function (globalVariable) {

    // Keep this variables private inside this closure scope
    var privateFunction = function() {
        console.log('Shhhh, this is private!');
    }

    // Expose the below methods via the globalVariable interface while
    // hiding the implementation of the method within the
    // function() block

    globalVariable.each = function(collection, iterator) {
        if (Array.isArray(collection)) {
            for (var i = 0; i < collection.length; i++) {
                iterator(collection[i], i, collection);
            }
        } else {
            for (var key in collection) {
                iterator(collection[key], key, collection);
            }
        }
    };
}

globalVariable.filter = function(collection, test) {
    var filtered = [];
    globalVariable.each(collection, function(item) {
        if (test(item)) {
            filtered.push(item);
        }
    });
    return filtered;
};

globalVariable.map = function(collection, iterator) {
    var mapped = [];
    globalUtils.each(collection, function(value, key, collection) {
        mapped.push(iterator(value));
    });
    return mapped;
};

globalVariable.reduce = function(collection, iterator, accumulator) {
    var startingValueMissing = accumulator === undefined;

    globalVariable.each(collection, function(item) {
        if(startingValueMissing) {
            accumulator = item;
            startingValueMissing = false;
        } else {
            accumulator = iterator(accumulator, item);
        }
    });

    return accumulator;
};

})(globalVariable));
```

In this example, ***globalVariable*** is the only variable that's global. The benefit of this approach over anonymous closures is that you declare the global variables upfront, making it crystal clear to people reading your code.

Example 3: Object interface

Yet another approach is to create modules using a self-contained object interface, like so:

CODE

```
var myGradesCalculate = (function () {

    // Keep this variable private inside this closure scope
    var myGrades = [93, 95, 88, 0, 55, 91];

    // Expose these functions via an interface while hiding
    // the implementation of the module within the function() block

    return {
        average: function() {
            var total = myGrades.reduce(function(accumulator, item) {
                return accumulator + item;
            }, 0);

            return 'Your average grade is ' + total / myGrades.length + '.';
        },

        failing: function() {
            var failingGrades = myGrades.filter(function(item) {
                return item < 70;
            });

            return 'You failed ' + failingGrades.length + ' times.';
        }
    }();
}

myGradesCalculate.failing(); // 'You failed 2 times.'
myGradesCalculate.average(); // 'Your average grade is 70.33333333333333.'
```

As you can see, this approach lets us decide what variables/methods we want to keep private (e.g. ***myGrades***) and what variables/methods we want to expose by putting them in the return statement (e.g. ***average*** & ***failing***).

Example 4: Revealing module pattern

This is very similar to the above approach, except that it ensures all methods and variables are kept private until explicitly exposed:

```

var myGradesCalculate = (function () {

    // Keep this variable private inside this closure scope
    var myGrades = [93, 95, 88, 0, 55, 91];

    var average = function() {
        var total = myGrades.reduce(function(accumulator, item) {
            return accumulator + item;
        }, 0);

        return 'Your average grade is ' + total / myGrades.length + '.';
    };

    var failing = function() {
        var failingGrades = myGrades.filter(function(item) {
            return item < 70;
        });

        return 'You failed ' + failingGrades.length + ' times.';
    };

    // Explicitly reveal public pointers to the private functions
    // that we want to reveal publicly

    return {
        average: average,
        failing: failing
    }
})();

myGradesCalculate.failing(); // 'You failed 2 times.'
myGradesCalculate.average(); // 'Your average grade is 70.333333333333333.'

```

That may seem like a lot to take in, but it's just the tip of the iceberg when it comes to module patterns. Here are a few of the resources I found useful in my own explorations:

- [Learning JavaScript Design Patterns](#) by Addy Osmani: a treasure trove of details in an impressively succinct read
- [Adequately Good by Ben Cherry](#): a useful overview with examples of advanced usage of the module pattern
- [Blog of Carl Danley](#): module pattern overview and resources for other JavaScript patterns.

CommonJS and AMD

The approaches above all have one thing in common: the use of a single global variable to wrap its code in a function, thereby creating a private namespace for itself using a closure scope.

While each approach is effective in its own way, they have their downsides.

For one, as a developer, you need to know the right dependency order to load your files in. For instance, let's say you're using Backbone in your project, so you include the script tag for Backbone's source code in your file.

However, since Backbone has a hard dependency on Underscore.js, the script tag for the Backbone file can't be placed before the Underscore.js file.

As a developer, managing dependencies and getting these things right can sometimes be a headache.

Another downside is that they can still lead to namespace collisions. For example, what if two of your modules have the same name? Or what if you have two versions of a module, and you need both?

So you're probably wondering: can we design a way to ask for a module's interface without going through the global scope?

Fortunately, the answer is yes.

There are two popular and well-implemented approaches: CommonJS and AMD.

CommonJS

CommonJS is a volunteer working group that designs and implements JavaScript APIs for declaring modules.

A CommonJS module is essentially a reusable piece of JavaScript which exports specific objects, making them available for other modules to *require* in their programs. If you've programmed in Node.js, you'll be very familiar with this format.

With CommonJS, each JavaScript file stores modules in its own unique module context (just like wrapping it in a closure). In this scope, we use the *module.exports* object to expose modules, and *require* to import them.

When you're defining a CommonJS module, it might look something like this:

```
function myModule() {
  this.hello = function() {
    return 'hello!';
  }

  this.goodbye = function() {
    return 'goodbye!';
  }
}

module.exports = myModule;
```

CODE

We use the special object *module* and place a reference of our function into *module.exports*. This lets the CommonJS module system know what we want to expose so that other files can consume it.

Then when someone wants to use *myModule*, they can require it in their file, like so:

```
var myModule = require('myModule');

var myModuleInstance = new myModule();
myModuleInstance.hello(); // 'hello!'
myModuleInstance.goodbye(); // 'goodbye!'
```

CODE

There are two obvious benefits to this approach over the module patterns we discussed before:

1. Avoiding global namespace pollution
2. Making our dependencies explicit

Moreover, the syntax is very compact, which I personally love.

Another thing to note is that CommonJS takes a server-first approach and synchronously loads modules. This matters because if we have three other modules we need to `require`, it'll load them one by one.

Now, that works great on the server but, unfortunately, makes it harder to use when writing JavaScript for the browser. Suffice it to say that reading a module from the web takes a *lot* longer than reading from disk. For as long as the script to load a module is running, it blocks the browser from running anything else until it finishes loading. It behaves this way because the JavaScript thread stops until the code has been loaded. (I'll cover how we can work around this issue in Part 2 when we discuss module bundling. For now, that's all we need to know).

AMD

CommonJS is all well and good, but what if we want to load modules asynchronously? The answer is called Asynchronous Module Definition, or AMD for short.

Loading modules using AMD looks something like this:

```
define(['myModule', 'myOtherModule'], function(myModule, myOtherModule) {
    console.log(myModule.hello());
});
```

CODE

What's happening here is that the `define` function takes as its first argument an array of each of the module's dependencies. These dependencies are loaded in the background (in a non-blocking manner), and once loaded `define` calls the callback function it was given.

Next, the callback function takes, as arguments, the dependencies that were loaded--in our case, `myModule` and `myOtherModule`--allowing the function to use these dependencies. Finally, the dependencies themselves must also be defined using the `define` keyword.

For example, `myModule` might look like this:

```
define([], function() {

    return {
        hello: function() {
            console.log('hello');
        },
        goodbye: function() {
            console.log('goodbye');
        }
    };
});
```

CODE

So again, unlike CommonJS, AMD takes a browser-first approach alongside asynchronous behavior to get the job done. (Note, there are a lot of people who strongly believe that dynamically loading files piecemeal as you start to run code isn't favorable, which we'll explore more when in the next section on module-building).

Aside from asynchronicity, another benefit of AMD is that your modules can be objects, functions, constructors, strings, JSON and many other types, while CommonJS only supports objects as modules.

That being said, AMD isn't compatible with io, filesystem, and other server-oriented features available via CommonJS, and the function wrapping syntax is a bit more verbose compared to a simple `require` statement.

UMD

For projects that require you to support both AMD and CommonJS features, there's yet another format: Universal Module Definition (UMD).

UMD essentially creates a way to use either of the two, while also supporting the global variable definition. As a result, UMD modules are capable of working on both client and server.

Here's a quick taste of how UMD goes about its business:

```
(function (root, factory) {
    if (typeof define === 'function' && define.amd) {
        // AMD
        define(['myModule', 'myOtherModule'], factory);
    } else if (typeof exports === 'object') {
        // CommonJS
        module.exports = factory(require('myModule'), require('myOtherModule'));
    } else {
        // Browser globals (Note: root is window)
        root.returnExports = factory(root.myModule, root.myOtherModule);
    }
})(this, function (myModule, myOtherModule) {
    // Methods
    function notHelloOrGoodbye(){};
    function hello(){};
    function goodbye(){};

    // Exposed public methods
    return {
        hello: hello,
        goodbye: goodbye
    }
});
```

CODE

For more examples of UMD formats, check out this [enlightening repo](#) on GitHub.

Native JS

Phew! Are you still around? I haven't lost you in the woods here? Good! Because we have *one more* type of module to define before we're done.

As you probably noticed, none of the modules above were native to JavaScript. Instead, we've created ways to *emulate* a modules system by using either the module pattern, CommonJS or AMD.

Fortunately, the smart folks at TC39 (the standards body that defines the syntax and semantics of ECMAScript) have introduced built-in modules with ECMAScript 6 (ES6).

ES6 offers up a variety of possibilities for importing and exporting modules which others have done a great job explaining--here are a few of those resources:

- jsmodules.io
- exploringjs.com

What's great about ES6 modules relative to CommonJS or AMD is how it manages to offer the best of both worlds:

compact and declarative syntax *and* asynchronous loading, plus added benefits like better support for cyclic dependencies.

Probably my favorite feature of ES6 modules is that imports are *live* read-only views of the exports. (Compare this to CommonJS, where imports are copies of exports and consequently not alive).

Here's an example of how that works:

CODE

```
// lib/counter.js

var counter = 1;

function increment() {
  counter++;
}

function decrement() {
  counter--;
}

module.exports = {
  counter: counter,
  increment: increment,
  decrement: decrement
};

// src/main.js

var counter = require('../lib/counter');

counter.increment();
console.log(counter.counter); // 1
```

In this example, we basically make two copies of the module: one when we export it, and one when we require it.

Moreover, the copy in main.js is now disconnected from the original module. That's why even when we increment our counter it still returns 1--because the counter variable that we imported is a disconnected copy of the counter variable from the module.

So, incrementing the counter will increment it in the module, but won't increment your copied version. The only way to modify the copied version of the counter variable is to do so manually:

CODE

```
counter.counter++;
console.log(counter.counter); // 2
```

On the other hand, ES6 creates a live read-only view of the modules we import:

```
// lib/counter.js
export let counter = 1;

export function increment() {
  counter++;
}

export function decrement() {
  counter--;
}

// src/main.js
import * as counter from '../lib/counter';

console.log(counter.counter); // 1
counter.increment();
console.log(counter.counter); // 2
```

Cool stuff, huh? What I find really compelling about live read-only views is how they allow you to split your modules into smaller pieces without losing functionality.

Then you can turn around and merge them again, no problem. It just "works."

Looking forward: bundling modules

Wow! Where does the time go? That was a wild ride, but I sincerely hope it gave you a better understanding of modules in JavaScript.

In the next section I'll walk through module bundling, covering core topics including:

- Why we bundle modules
- Different approaches to bundling
- ECMAScript's module loader API
- ...and more. :)

NOTE: To keep things simple, I skipped over some of the nitty-gritty details (think: cyclic dependencies) in this post. If I left out anything important and/or fascinating, please let me know in the comments!

11. What is the Execution Context & Stack in JavaScript?

In this post I will take an in-depth look at one of the most fundamental parts of JavaScript, the `Execution Context`. By the end of this post, you should have a clearer understanding about what the interpreter is trying to do, why some functions / variables can be used before they are declared and how their value is really determined.

What is the Execution Context?

When code is run in JavaScript, the environment in which it is executed is very important, and is evaluated as 1 of the following:

- **Global code** -- The default environment where your code is executed for the first time.

- **Function code** -- Whenever the flow of execution enters a function body.
- **Eval code** -- Text to be executed inside the internal eval function.

You can read a lot of resources online that refer to `scope`, and for the purpose of this article to make things easier to understand, let's think of the term `execution context` as the environment / scope the current code is being evaluated in. Now, enough talking, let's see an example that includes both `global` and `function / local` context evaluated code.

```
// global context

var sayHello = 'Hello';

function person() {           // execution context

    var first = 'David',
        last = 'Shariff';

    function firstName() { // execution context
        return first;
    }

    function lastName() { // execution context
        return last;
    }

    alert(sayHello + firstName() + ' ' + lastName());
}

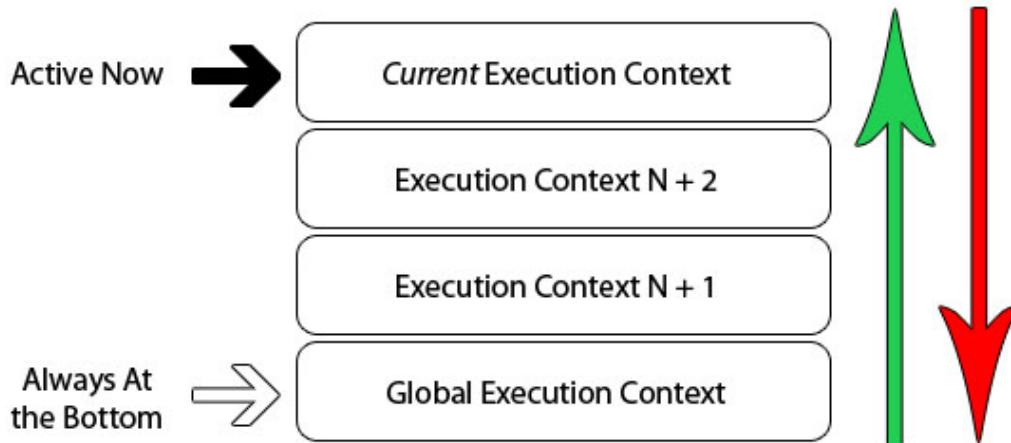
}
```

Nothing special is going on here, we have 1 `global context` represented by the purple border and 3 different `function contexts` represented by the green, blue and orange borders. There can only ever be 1 `global context`, which can be accessed from any other context in your program.

You can have any number of `function contexts`, and each function call creates a new context, which creates a private scope where anything declared inside of the function can not be directly accessed from outside the current function scope. In the example above, a function can access a variable declared outside of its current context, but an outside context can not access the variables / functions declared inside. Why does this happen? How exactly is this code evaluated?

Execution Context Stack

The JavaScript interpreter in a browser is implemented as a single thread. What this actually means is that only 1 thing can ever happen at one time in the browser, with other actions or events being queued in what is called the `Execution Stack`. The diagram below is an abstract view of a single threaded stack:

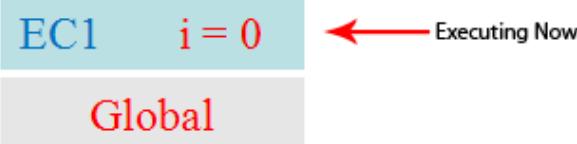


As we already know, when a browser first loads your script, it enters the `global execution context` by default. If, in your global code you call a function, the sequence flow of your program enters the function being called, creating a new `execution context` and pushing that context to the top of the `execution stack`.

If you call another function inside this current function, the same thing happens. The execution flow of code enters the inner function, which creates a new `execution context` that is pushed to the top of the existing stack. The browser will always execute the `current execution context` that sits on top of the stack, and once the function completes executing the `current execution context`, it will be popped off the top of the stack, returning control to the context below in the current stack. The example below shows a recursive function and the program's `execution stack`:

```
(function foo(i) {
    if (i === 3) {
        return;
    }
    else {
        foo(++i);
    }
}(0));
```

CODE



The code simply calls itself 3 times, incrementing the value of `i` by 1. Each time the function `foo` is called, a new execution context is created. Once a context has finished executing, it pops off the stack and control returns to the

context below it until the `global` context is reached again.

There are 5 key points to remember about the execution stack :

- Single threaded.
- Synchronous execution.
- 1 Global context.
- Infinite function contexts.
- Each function call creates a new execution context , even a call to itself.

Execution Context in Detail

So we now know that everytime a function is called, a new execution context is created. However, inside the JavaScript interpreter, every call to an execution context has 2 stages:

1. Creation Stage [when the function is called, but before it executes any code inside]:

- Create the [Scope Chain](#).
- Create variables, functions and arguments.
- Determine the value of [`"this"`](#) .

2. Activation / Code Execution Stage:

- Assign values, references to functions and interpret / execute code.

It is possible to represent each execution context conceptually as an object with 3 properties:

```
executionContextObj = {
  scopeChain: { /* variableObject + all parent execution context's variableObject */ },
  variableObject: { /* function arguments / parameters, inner variable and function declaration
s */ },
  this: {}
}
```

CODE

Activation / Variable Object [AO/VO]

This `executionContextObj` is created when the function is invoked, but *before* the actual function has been executed. This is known as stage 1, the `Creation Stage` . Here, the interpreter creates the `executionContextObj` by scanning the function for parameters or arguments passed in, local function declarations and local variable declarations. The result of this scan becomes the `variableObject` in the `executionContextObj` .

Here is a pseudo-overview of how the interpreter evaluates the code:

1. Find some code to invoke a function.
2. Before executing the function code, create the execution context .
3. Enter the creation stage:
 - Initialize the [Scope Chain](#) .
 - Create the variable object :
 - Create the `arguments` object , check the context for parameters, initialize the name and value and create a reference copy.
 - Scan the context for function declarations:

- For each function found, create a property in the `variable object` that is the exact function name, which has a reference pointer to the function in memory.
- If the function name exists already, the reference pointer value will be overwritten.
- Scan the context for variable declarations:
 - For each variable declaration found, create a property in the `variable object` that is the variable name, and initialize the value as `undefined`.
 - If the variable name already exists in the `variable object`, do nothing and continue scanning.
- Determine the value of `"this"` inside the context.

4. Activation / Code Execution Stage:

- Run / interpret the function code in the context and assign variable values as the code is executed line by line.

Let's look at an example:

```
function foo(i) {
  var a = 'hello';
  var b = function privateB() {
    };
    function c() {
    }
}

foo(22);
```

CODE

On calling `foo(22)`, the creation stage looks as follows:

```
fooExecutionContext = {
  scopeChain: { ... },
  variableObject: {
    arguments: {
      0: 22,
      length: 1
    },
    i: 22,
    c: pointer to function c()
    a: undefined,
    b: undefined
  },
  this: { ... }
}
```

CODE

As you can see, the creation stage handles defining the names of the properties, not assigning a value to them, with the exception of formal arguments / parameters. Once the creation stage has finished, the flow of execution enters the function and the activation / code execution stage looks like this after the function has finished execution:

CODE

```

fooExecutionContext = {
    scopeChain: { ... },
    variableObject: {
        arguments: {
            0: 22,
            length: 1
        },
        i: 22,
        c: pointer to function c()
        a: 'hello',
        b: pointer to function privateB()
    },
    this: { ... }
}

```

A Word On Hoisting

You can find many resources online defining the term `hoisting` in JavaScript, explaining that variable and function declarations are *hoisted* to the top of their function scope. However, none explain in detail why this happens, and armed with your new knowledge about how the interpreter creates the `activation object`, it is easy to see why. Take the following code example:

CODE

```

(function() {

    console.log(typeof foo); // function pointer
    console.log(typeof bar); // undefined

    var foo = 'hello',
        bar = function() {
            return 'world';
        };

    function foo() {
        return 'hello';
    }

})();â€¢

```

The questions we can now answer are:

- **Why can we access `foo` before we have declared it?**
 - If we follow the `creation stage`, we know the variables have already been created before the `activation / code execution stage`. So as the function flow started executing, `foo` had already been defined in the `activation object`.
- **Foo is declared twice, why is `foo` shown to be function and not undefined or string ?**
 - Even though `foo` is declared twice, we know from the `creation stage` that functions are created on the `activation object` before variables, and if the property name already exists on the `activation object`, we simply bypass the declaration.
 - Therefore, a reference to `function foo()` is first created on the `activation object`, and when we get interpreter gets to `var foo`, we already see the property name `foo` exists so the code does nothing and proceeds.

- Why is bar undefined ?
 - bar is actually a variable that has a function assignment, and we know the variables are created in the creation stage but they are initialized with the value of undefined .

Summary

Hopefully by now you have a good grasp about how the JavaScript interpreter is evaluating your code. Understanding the execution context and stack allows you to know the reasons behind why your code is evaluating to different values that you had not initially expected.

Do you think knowing the inner workings of the interpreter is too much overhead or a necessity to your JavaScript knowledge ? Does knowing the execution context phase help you write better JavaScript ?

Note: Some people have been asking about closures, callbacks, timeout etc which I will cover in the [next post](#), focusing more on the [Scope Chain](#) in relation to the execution context .

Further Reading

- [ECMA-262 5th Edition](#)
- [ECMA-262-3 in detail. Chapter 2. Variable object](#)
- [Identifier Resolution, Execution Contexts and scope chains](#)

12. Understanding Scope and Context in JavaScript

JavaScript's implementation of scope and context is a unique feature of the language, in part because it is so flexible. Functions can be adopted for various contexts and scope can be encapsulated and preserved. These concepts lend to some of the most powerful design patterns JavaScript has to offer. However, this is also a tremendous source of confusion amongst developers, and for good reason. The following is a comprehensive explanation of scope and context, the difference between them, and how various design patterns make use of them.

Context vs. Scope

The first important thing to clear up is that context and scope are not the same. I have noticed many developers over the years often confuse the two terms (myself included), incorrectly describing one for the other. To be fair, the terminology has become quite muddled over the years.

Every function invocation has both a scope and a context associated with it. Fundamentally, scope is function-based while context is object-based. In other words, scope pertains to the variable access of a function when it is invoked and is unique to each invocation. Context is always the value of the `this` keyword which is a reference to the object that "owns" the currently executing code.

Variable Scope

A variable can be defined in either local or global scope, which establishes the variables' accessibility from different scopes during runtime. Any defined global variable, meaning any variable declared outside of a function body will live throughout runtime and can be accessed and altered in any scope. Local variables exist only within the function body of which they are defined and will have a different scope for every call of that function. There it is subject for value

assignment, retrieval, and manipulation only within that call and is not accessible outside of that scope.

JavaScript presently does not support block scope which is the ability to define a variable to the scope of an if statement, switch statement, for loop, or while loop. This means the variable will not be accessible outside the opening and closing curly braces of the block. Currently any defined variables inside a block are accessible outside the block. However, this is soon to change, the `let` keyword has officially been added to the ES6 specification. It can be used alternatively to the `var` keyword and supports the declaration of block scope local variables.

What is "this" Context

Context is most often determined by how a function is invoked. When a function is called as a method of an object, `this` is set to the object the method is called on:

```
var obj = {
  foo: function(){
    alert(this === obj);
  }
};

obj.foo(); // true
```

CODE

The same principle applies when invoking a function with the `new` operator to create an instance of an object. When invoked in this manner, the value of `this` within the scope of the function will be set to the newly created instance:

```
function foo(){
  alert(this);
}

foo() // window
new foo() // foo
```

CODE

When called as an unbound function, `this` will default to the global context or `window` object in the browser. However, if the function is executed in *strict mode*, the context will default to `undefined`.

Execution Context

JavaScript is a single threaded language, meaning only one task can be executed at a time. When the JavaScript interpreter initially executes code, it first enters into a global execution context by default. Each invocation of a function from this point on will result in the creation of a new execution context.

This is where confusion often sets in, the term "execution context" is actually for all intents and purposes referring more to scope and not context as previously discussed. It is an unfortunate naming convention, however it is the terminology as defined by the ECMAScript specification, so we're kinda stuck with it.

Each time a new execution context is created it is appended to the top of the *execution stack*. The browser will always execute the current execution context that is atop the execution stack. Once completed, it will be removed from the top of the stack and control will return to the execution context below.

An execution context can be divided into a creation and execution phase. In the creation phase, the interpreter will first create a *variable object* (also called an *activation object*) that is composed of all the variables, function

declarations, and arguments defined inside the execution context. From there the *scope chain* is initialized next, and the value of `this` is determined last. Then in the execution phase, code is interpreted and executed.

The Scope Chain

For each execution context there is a scope chain coupled with it. The scope chain contains the variable object for every execution context in the execution stack. It is used for determining variable access and identifier resolution. For example:

```
function first(){
    second();
    function second(){
        third();
        function third(){
            fourth();
            function fourth(){
                // do something
            }
        }
    }
first();
```

CODE

Running the preceding code will result in the nested functions being executed all the way down to the `fourth` function. At this point the scope chain would be, from top to bottom: `fourth`, `third`, `second`, `first`, global. The `fourth` function would have access to global variables and any variables defined within the `first`, `second`, and `third` functions as well as the functions themselves.

Name conflicts amongst variables between different execution contexts are resolved by climbing up the scope chain, moving locally to globally. This means that local variables with the same name as variables higher up the scope chain take precedence.

To put it simply, each time you attempt to access a variable within a function's execution context, the look-up process will always begin with its own variable object. If the identifier is not found in the variable object, the search continues into the scope chain. It will climb up the scope chain examining the variable object of every execution context looking for a match to the variable name.

Closures

Accessing variables outside of the immediate lexical scope creates a closure. In other words, a closure is formed when a nested function is defined inside of another function, allowing access to the outer function's variables.

Returning the nested function allows you to maintain access to the local variables, arguments, and inner function declarations of its outer function. This encapsulation allows us to hide and preserve the execution context from outside scopes while exposing a public interface and thus is subject to further manipulation. A simple example of this looks like the following:

CODE

```

function foo(){
    var localVariable = 'private variable';
    return function bar(){
        return localVariable;
    }
}

var getLocalVariable = foo();
getLocalVariable() // private variable

```

One of the most popular types of closures is what is widely known as the *module pattern*; it allows you to emulate public, private, and privileged members:

CODE

```

var Module = (function(){
    var privateProperty = 'foo';

    function privateMethod(args){
        // do something
    }

    return {

        publicProperty: '',

        publicMethod: function(args){
            // do something
        },

        privilegedMethod: function(args){
            return privateMethod(args);
        }
    };
})();

```

The module acts as if it were a singleton, executed as soon as the compiler interprets it, hence the opening and closing parenthesis at the end of the function. The only available members outside of the execution context of the closure are your public methods and properties located in the return object (`Module.publicMethod` for example). However, all private properties and methods will live throughout the life of the application as the execution context is preserved, meaning variables are subject to further interaction via the public methods.

Another type of closure is what is called an immediately-invoked function expression (IIFE) which is nothing more than a self-invoked anonymous function executed in the context of the window:

CODE

```
(function(window){

    var foo, bar;

    function private(){
        // do something
    }

    window.Module = {

        public: function(){
            // do something
        }
    };

})(this);
```

This expression is most useful when attempting to preserve the global namespace as any variables declared within the function body will be local to the closure but will still live throughout runtime. This is a popular means of encapsulating source code for applications and frameworks, typically exposing a single global interface in which to interact with.

Call and Apply

These two methods inherent to all functions allow you to execute any function in any desired context. This makes for incredibly powerful capabilities. The `call` function requires the arguments to be listed explicitly while the `apply` function allows you to provide the arguments as an array:

```
function user(firstName, lastName, age){
    // do something
}

user.call(window, 'John', 'Doe', 30);
user.apply(window, ['John', 'Doe', 30]);
```

CODE

The result of both calls is exactly the same, the `user` function is invoked in the context of the `window` and provided the same three arguments.

ECMAScript 5 (ES5) introduced the `Function.prototype.bind` method that is used for manipulating context. It returns a new function which is permanently bound to the first argument of `bind` regardless of how the function is being used. It works by using a closure that is responsible for redirecting the call in the appropriate context. See the following polyfill for unsupported browsers:

CODE

```

if(!('bind' in Function.prototype)){
    Function.prototype.bind = function(){
        var fn = this,
            context = arguments[0],
            args = Array.prototype.slice.call(arguments, 1);
        return function(){
            return fn.apply(context, args.concat([].slice.call(arguments)));
        }
    }
}

```

It is commonly used where context is commonly lost; object-orientation and event handling. This is necessary because the `addEventListener` method of a node will always execute the callback in the context of the node the event handler is bound to, which is the way it should be. However if you're employing advanced object-oriented techniques and require your callback to be a method of an instance, you will be required to manually adjust the context, this is where `bind` comes in handy:

CODE

```

function MyClass(){
    this.element = document.createElement('div');
    this.element.addEventListener('click', this.onClick.bind(this), false);
}

MyClass.prototype.onClick = function(e){
    // do something
};

```

While reviewing the source of the polyfill for `Function.prototype.bind` function, you may have noticed 2 invocations involving the `slice` method of an `Array`:

CODE

```

Array.prototype.slice.call(arguments, 1);
[].slice.call(arguments);

```

What is interesting to note here is that the `arguments` object is not actually an array at all, however it is often described as an array-like object much like a nodelist (anything returned by `element.childNodes`). They contain a `length` property and indexed values but they are still not arrays, and subsequently don't support any of the native methods of arrays such as `slice` and `push`. However, because of their similar behavior, the methods of `Array` can be adopted or hijacked, if you will, and executed in the context of an array-like object as is the case above.

This technique of adopting another object's methods also applies to object-orientation when emulating classical based inheritance in JavaScript:

CODE

```

MyClass.prototype.init = function(){
    // call the superclass init method in the context of the "MyClass" instance
    MySuperClass.prototype.init.apply(this, arguments);
}

```

By invoking the method of the superclass (`MySuperClass`) in the context of an instance of a subclass (`MyClass`), we can mimic the ability of calling a method's super to fully exploit this powerful design pattern.

Conclusion

It is important to understand these concepts before you begin to approach advanced design patterns, as scope and context play a fundamental role in modern JavaScript. Whether we're talking about closures, object-orientation and inheritance, or various native implementations, context and scope play a significant role in all of them. If your goal is to master the JavaScript language and better understand all it encompasses then scope and context should be one of your starting points.

13. JavaScript's 'this' Keyword

A commonly used feature of JavaScript is the "this" keyword, but it is often also one of the most confused and misinterpreted features of the language. What does "this" actually mean and how is it decided?

This article tries to clear up the confusion and explain the answer in a clear fashion.

The "this" keyword is not new to those who have programmed in other languages, and more often than not it refers to the new object created when instantiating a class via its constructor. For example, if I have a class Boat(), which has a method moveBoat(), when referring to "this" inside of the moveBoat() method, we are actually accessing the newly created object of Boat().

In JavaScript, we also have this concept inside a Function constructor when it is invoked using the "new" keyword, however it is not the only rule and "this" can often refer to a different object from a different execution context. If you are not familiar with JavaScript's execution context, I recommend you read my other post on the topic [here](#). Enough talking, let's see some JavaScript examples:

```
// global scope  
  
foo = 'abc';  
alert(foo); // abc  
  
this.foo = 'def';  
alert(foo); // def
```

CODE

Whenever you use the keyword "this" in the global context (not inside a function), it always refers to the global object.

Now let's look at the value of "this" inside a function:

CODE

```

var boat = {
    size: 'normal',
    boatInfo: function() {
        alert(this === boat);
        alert(this.size);
    }
};

boat.boatInfo(); // true, 'normal'

var bigBoat = {
    size: 'big'
};

bigBoat.boatInfo = boat.boatInfo;
bigBoat.boatInfo(); // false, 'big'

```

So how is "this" determined above? We can see an object boat which has a property size and a method boatInfo(). Inside boatInfo(), it alerts if the value of this is the actual boat object, and also alerts the size property of this. So, we invoke the function using boat.boatInfo() and can see that this is the boat object and the size property of the boat is normal.

We then create another object, bigBoat, which has a size property of big. However, the bigBoat object does not have a boatInfo() method, so we copy the method from boat using bigBoat.boatInfo = boat.boatInfo. Now, when we call bigBoat.boatInfo() and enter the function, we see that this is not equal to boat and that size property is now big. Why did this happen? How did the value of this change inside boatInfo()?

The first thing you must realise is that the value of this inside any function is never static, it is always determined every time you call a function, but before the function actually executes its code. The value of this inside a function is actually provided by the parent scope in which the function was called, and more importantly, how the actual function syntax was written.

Whenever a function is called, we must look at the immediate left side of the brackets / parentheses "()". If on the left side of the parentheses we can see a reference, then the value of "this" passed to the function call is exactly of which that object belongs to, otherwise it is the global object. Let's see some examples:

CODE

```

function bar() {
    alert(this);
}

bar(); // global - because the method bar() belongs to the global object when invoked

var foo = {
    baz: function() {
        alert(this);
    }
}

foo.baz(); // foo - because the method baz() belongs to the object foo when invoked

```

If things are clear this far, then the above code obviously makes sense. We can further complicate things by changing the value of "this" inside the very same function, by writing the call / invoke syntax in 2 different ways:

CODE

```

var foo = {
    baz: function() {
        alert(this);
    }
}
foo.baz(); // foo - because baz belongs to the foo object when invoked

var anotherBaz = foo.baz;
anotherBaz(); // global - because the method anotherBaz() belongs to the global object when invoked, NOT foo

```

Here, we see the value of "this" inside baz() was different each time, as it was syntactically called in 2 different ways.

Now, let's see the value of "this" inside a deeply nested object:

CODE

```

var anum = 0;

var foo = {
    anum: 10,
    baz: {
        anum: 20,
        bar: function() {
            console.log(this.anum);
        }
    }
}
foo.baz.bar(); // 20 - because left side of () is bar, which belongs to baz object when invoked

var hello = foo.baz.bar;
hello(); // 0 - because left side of () is hello, which belongs to global object when invoked

```

Another question often asked is how is the keyword "this" determined inside an event handler? The answer is "this" inside of an event handler always refers to the element it was triggered on. Let's see an example:

CODE

```

<div id="test">I am an element with id #test</div>

function doAlert() {
    alert(this.innerHTML);
}

doAlert(); // undefined

var myElem = document.getElementById('test');
myElem.onclick = doAlert;

alert(myElem.onclick === doAlert); // true
myElem.onclick(); // I am an element

```

Here we can see that when doAlert() is first called, it alerts undefined, as doAlert() belongs to the global object. We then write myElem.onclick = doAlert, which copies the function doAlert() to the onclick() event of myElem. This basically means that whenever onclick() is fired, it is a method of myElem, meaning the value of "this" will be the myElem object exactly.

One last thing I want to note on this topic, is that the value of "this" can also be manually set using call() and

`apply()`, overriding what we have discussed here today. Also of interest is that when calling "this" inside a function constructor, "this" refers to the newly created object in all instances inside the constructor. The reason for that is the function constructor is invoked with the "new" keyword, which creates a new object where "this" inside the constructor always refers to the new object just created.

Summary

Hopefully today's blog post has cleared up any misunderstanding of the "this" keyword and you can go forth always knowing the correct value of "this". We now know that the value of "this" is never static and has a different value depending on how the function was invoked.

14. JavaScript Inheritance Patterns

In this post, I am going to introduce to you 3 different ways of how you can implement inheritance in JavaScript. You will see inheritance implemented in languages such as Java by allowing a class to inherit state and behavior from a superclass, where each superclass can have many subclasses.

This means that in Java an object is an instance of a class, which can inherit other classes. Now in JavaScript, being prototypal by nature, an object can inherit from an object.

For the rest of this post, I will introduce the Pseudoclassical, Functional and Prototypal inheritance patterns in JavaScript.

Pseudoclassical pattern

The Pseudoclassical pattern tries to replicate inheritance in a way that is familiar to those who come from a Java or C like background. By using Pseudoclassical inheritance, we attempt to recreate classic programming language's behavior by using class wide inheritance and where *objects are instances of those classes*.

A pattern which uses a `constructor` function and the `new` operator, combined with a prototype added to the `constructor` is said to be Pseudoclassical.

In JavaScript, one way to do this inheritance is:

1. Invoke a constructor function.
2. Point a child's prototype to the parent's prototype for inheritance to occur.

```
/*
 * Point a child's prototype to a parent's prototype
 */
var extendObj = function(childObj, parentObj) {
    childObj.prototype = parentObj.prototype;
};

// base human object
var Human = function() {};
// inheritable attributes / methods
Human.prototype = {
    name: '',
    gender: '',
    planetOfBirth: 'Earth',
    sayGender: function () {
        alert(this.name + ' says my gender is ' + this.gender);
    },
    sayPlanet: function () {
        alert(this.name + ' was born on ' + this.planetOfBirth);
    }
};

// male
var Male = function (name) {
    this.gender = 'Male';
    this.name = 'David';
};
// inherits human
extendObj(Male, Human);

// female
var Female = function (name) {
    this.name = name;
    this.gender = 'Female';
};
// inherits human
extendObj(Female, Human);

// new instances
var david = new Male('David');
var jane = new Female('Jane');

david.sayGender(); // David says my gender is Male
jane.sayGender(); // Jane says my gender is Female

Male.prototype.planetOfBirth = 'Mars';
david.sayPlanet(); // David was born on Mars
jane.sayPlanet(); // Jane was born on Mars
```

As expected, we have achieved inheritance in a Pseudoclassical manner, however, this solution has a problem. If you look at the last line, you will see the alert says Jane was born on Mars , but what we really want it to say is Jane was born on Earth . The reason for this is the Male prototype was changed to "Mars".

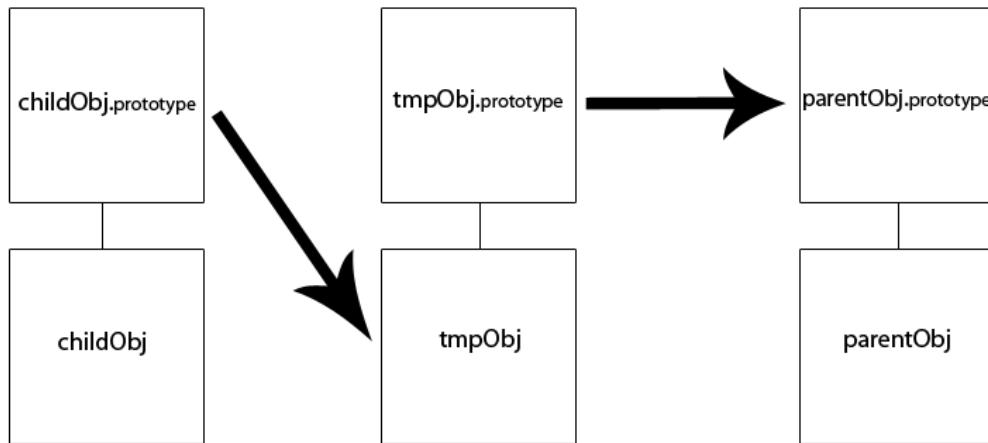
Given the direct link between the Male and Human prototype, if Human has many children inheriting from it, any change on a child's prototype properties will affect Human , and thus all children inheriting from Human . Changing a child's prototype should not affect other children inheriting from the same parent. The reason for this is because JavaScript passes objects by reference, not by value, meaning all children of Human inherit changes occurred on

other children's prototypes.

`childObj.prototype = parentObj.prototype` does give us inheritance. However, if you want to fix the issue above, you need to replace the `extendObj` function to take the child's prototype and link it to a temporary object, whose prototype is the parent object's prototype. In this way, by creating a temporary "middle" object, you allow the temporary object to be empty and inherit its properties from `Human`.

By doing this, you have solved the pass by reference issue with a new instance of an empty object, which still inherits from the parent, but is not affected by other children.

To understand this clearly, the image below shows the flow of the `extendObj` function.



Now, if you ran the same code again, but with the changes in `extendObj` below, you would see "Jane was born on Earth" was alerted.

```
/*
 * Create a new constructor function, whose prototype is the parent object's prototype.
 * Set the child's prototype to the newly created constructor function.
 */
var extendObj = function(childObj, parentObj) {
    var tmpObj = function () {}
    tmpObj.prototype = parentObj.prototype;
    childObj.prototype = new tmpObj();
    childObj.prototype.constructor = childObj;
};

// base human object
var Human = function () {};
// inheritable attributes / methods
Human.prototype = {
    name: '',
    gender: '',
    planetOfBirth: 'Earth',
    sayGender: function () {
        alert(this.name + ' says my gender is ' + this.gender);
    },
    sayPlanet: function () {
        alert(this.name + ' was born on ' + this.planetOfBirth);
    }
};

// male
var Male = function (name) {
    this.gender = 'Male';
    this.name = 'David';
};
// inherits human
extendObj(Male, Human);

// female
var Female = function (name) {
    this.name = name;
    this.gender = 'Female';
};
// inherits human
extendObj(Female, Human);

// new instances
var david = new Male('David');
var jane = new Female('Jane');

david.sayGender(); // David says my gender is Male
jane.sayGender(); // Jane says my gender is Female

Male.prototype.planetOfBirth = 'Mars';
david.sayPlanet(); // David was born on Mars
jane.sayPlanet(); // Jane was born on Earth
```

Functional pattern

Another pattern you can use to achieve inheritance in JavaScript is by [Douglas Crockford](#), called Functional inheritance. This pattern allows one object to inherit from another, take the result and augment it at the child level to

achieve inheritance. What this really means, is you create an object as your parent, pass the child object to the parent to inherit / apply its properties, and return the resulting object back to the child, who can then augment its own properties to the object returned from the parent.

Below is the same example used above to explain Pseudoclassical inheritance, but written in a functional nature.

CODE

```

var human = function(name) {
    var that = {};

    that.name = name || '';
    that.gender = '';
    that.planetOfBirth = 'Earth';
    that.sayGender = function () {
        alert(that.name + ' says my gender is ' + that.gender);
    };
    that.sayPlanet = function () {
        alert(that.name + ' was born on ' + that.planetOfBirth);
    };

    return that;
}

var male = function (name) {
    var that = human(name);
    that.gender = 'Male';
    return that;
}

var female = function (name) {
    var that = human(name);
    that.gender = 'Female';
    return that;
}

var david = male('David');
var jane = female('Jane');

david.sayGender(); // David says my gender is Male
jane.sayGender(); // Jane says my gender is Female

david.planetOfBirth = 'Mars';
david.sayPlanet(); // David was born on Mars
jane.sayPlanet(); // Jane was born on Earth

```

As you can see by using this pattern, there is no need to use the prototype chain, constructors or the "new" keyword. Functional inheritance achieves this by passing a unique object around every time an instance of the function is called.

This however, has a downside for performance because each object is unique, meaning each function call creates a new object, so the JavaScript interpreter has to assign new memory to the function in order to recompile everything inside of it as unique again.

There are also benefits to this approach, as the closures of each function allow for good use of public and private methods / attributes. Let's take this code for example, which shows a parent class of vehicle and children classes of motorbike and boat .

CODE

```
var vehicle = function(attrs) {
    var _privateObj = {
        hasEngine: true
    },
    that = {};

    that.name = attrs.name || null;
    that.engineSize = attrs.engineSize || null;
    that.hasEngine = function () {
        alert('This ' + that.name + ' has an engine: ' + _privateObj.hasEngine);
    };

    return that;
}

var motorbike = function () {

    // private
    var _privateObj = {
        numWheels: 2
    },

    // inherit
    that = vehicle({
        name: 'Motorbike',
        engineSize: 'Small'
    });

    // public
    that.totalNumWheels = function () {
        alert('This Motobike has ' + _privateObj.numWheels + ' wheels');
    };

    that.increaseWheels = function () {
        _privateObj.numWheels++;
    };

    return that;
};

var boat = function () {

    // inherit
    that = vehicle({
        name: 'Boat',
        engineSize: 'Large'
    });

    return that;
};

myBoat = boat();
myBoat.hasEngine(); // This Boat has an engine: true
alert(myBoat.engineSize); // Large

myMotorbike = motorbike();
myMotorbike.hasEngine(); // This Motorbike has an engine: true
myMotorbike.increaseWheels();
```

```
myMotorbike.totalNumWheels(); // This Motorbike has 3 wheels
alert(myMotorbike.engineSize); // Small

myMotorbike2 = motorbike();
myMotorbike2.totalNumWheels(); // This Motorbike has 2 wheels

myMotorbike._privateObj.numWheels = 0; // undefined
myBoat.totalNumWheels(); // undefined
```

You can see that it is fairly easy to provide encapsulation. The `_privateObj` can not be modified from outside of the object, unless exposed by a public method like `increaseWheels()`. Similarly, private values can also only be read when exposed by a public method, such as `motorbike's totalNumWheels()` function.

Prototypal pattern

You can also implement inheritance in JavaScript using a pure prototypal approach which is more suited to the language.

As of ECMAScript 5, it is possible to create an inherited object by simply doing the following:

```
var male = Object.create(human);
```

CODE

However, support is not so good for older browsers, thankfully you can augment the `Object` with a `create` method should it not exist already, which will have the same behavior as that of ECMAScript 5.

CODE

```
(function () {
    'use strict';

    //***** Helper functions for older browsers *****/
    if (!Object.hasOwnProperty('create')) {
        Object.create = function (parentObj) {
            function tmpObj() {}
            tmpObj.prototype = parentObj;
            return new tmpObj();
        };
    }
    if (!Object.hasOwnProperty('defineProperties')) {
        Object.defineProperties = function (obj, props) {
            for (var prop in props) {
                Object.defineProperty(obj, prop, props[prop]);
            }
        };
    }
    //***** */

    var human = {
        name: '',
        gender: '',
        planetOfBirth: 'Earth',
        sayGender: function () {
            alert(this.name + ' says my gender is ' + this.gender);
        },
        sayPlanet: function () {
            alert(this.name + ' was born on ' + this.planetOfBirth);
        }
    };

    var male = Object.create(human, {
        gender: {value: 'Male'}
    });

    var female = Object.create(human, {
        gender: {value: 'Female'}
    });

    var david = Object.create(male, {
        name: {value: 'David'},
        planetOfBirth: {value: 'Mars'}
    });

    var jane = Object.create(female, {
        name: {value: 'Jane'}
    });

    david.sayGender(); // David says my gender is Male
    david.sayPlanet(); // David was born on Mars

    jane.sayGender(); // Jane says my gender is Female
    jane.sayPlanet(); // Jane was born on Earth
})();
```

Summary

So today we have covered 3 different ways that you can implement inheritance in JavaScript. Most people are aware of prototypes, but as we have seen today, the Pseudoclassical and Functional patterns are just as valid.

Which pattern you should use varies depending on your project, there is no real "1 fits all" solution, so you are best to choose 1 you feel is the most suitable.

15. Identifier Resolution and Closures in the JavaScript Scope Chain

From my [previous post](#), we now know that every function has an associated [execution context](#) that contains a [variable object \[VO\]](#), which is composed of all the variables, functions and parameters defined inside that given local function.

The `scope chain` property of each [execution context](#) is simply a collection of the current context's [\[VO\]](#) + all parent's lexical [\[VO\]](#)s.

```
Scope = V0 + All Parent V0s
Eg: scopeChain = [ [V0] + [V01] + [V02] + [V0 n+1] ];
```

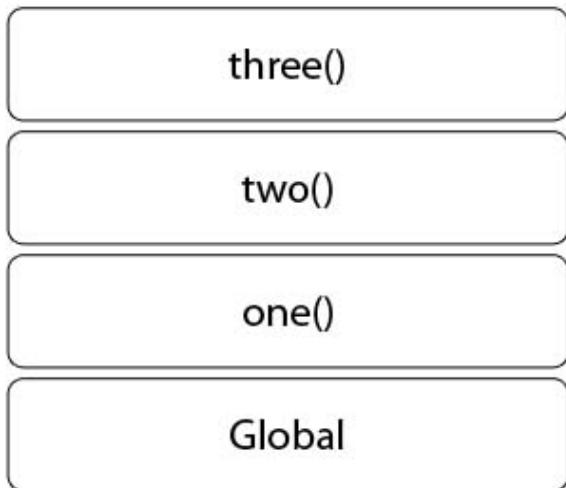
CODE

Determining a Scope Chain's Variable Objects [VO]s

We now know that the first [\[V0\]](#) of the `scope chain` belongs to the current [execution context](#), and we can find the remaining parent [\[VO\]](#)s by looking at the parent context's `scope chain`:

```
function one() {
    two();
    function two() {
        three();
        function three() {
            alert('I am at function three');
        }
    }
    one();
}
```

CODE



Execution Context Stack

The example is straight forward, starting from the global context we call `one()`, `one()` calls `two()`, which in turn calls `three()`, thus alerting that it is at function three. The image above shows the call stack at function `three` at the time `alert('I am at function three')` is fired. We can see that the scope chain at this point in time looks as follows:

```
three() Scope Chain = [ [three() v0] + [two() v0] + [one() v0] + [Global v0] ];
```

CODE

Lexical Scope

An important feature of JavaScript to note, is that the interpreter uses **Lexical Scoping**, as opposed to [Dynamic Scoping](#). This is just a complicated way of saying all inner functions, are statically (lexically) bound to the parent context in which the inner function was physically defined in the program code.

In our previous example above, it does not matter in which sequence the inner functions are called. `three()` will always be statically bound to `two()`, which in turn will always be bound to `one()` and so on and so forth. This gives a chaining effect where all inner functions can access the outer functions `v0` through the statically bound Scope Chain .

This lexical scope is the source of confusion for many developers. We know that every invocation of a function will create a new execution context and associated `v0`, which holds the values of variables evaluated in the current context.

It is this dynamic, runtime evaluation of the `v0` paired with the lexical (static) defined scope of each context that leads unexpected results in program behaviour. Take the following classic example:

CODE

```

var myAlerts = [];

for (var i = 0; i < 5; i++) {
    myAlerts.push(
        function inner() {
            alert(i);
        }
    );
}

myAlerts[0](); // 5
myAlerts[1](); // 5
myAlerts[2](); // 5
myAlerts[3](); // 5
myAlerts[4](); // 5

```

At first glance, those new to JavaScript would assume `alert(i);` to be the value of `i` on each increment where the function was physically defined in the source code, alerting 1, 2, 3, 4 and 5 respectively.

This is the most common point of confusion. Function `inner` was created in the global context, therefore its scope chain is statically bound to the global context.

Lines 11 ~ 15 invoke `inner()`, which looks in `inner.ScopeChain` to resolve `i`, which is located in the global context. At the time of each invocation, `i`, has already been incremented to 5, giving the same result every time `inner()` is called. The statically bound scope chain, which holds [V0s] from each context containing live variables, often catches developers by surprise.

Resolving the value of variables

The following example alerts the value of variables `a`, `b` and `c`, which gives us a result of 6.

CODE

```

function one() {

    var a = 1;
    two();

    function two() {

        var b = 2;
        three();

        function three() {

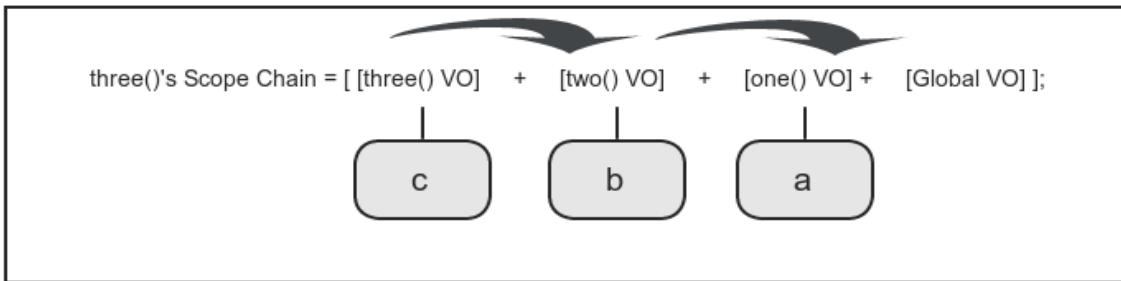
            var c = 3;
            alert(a + b + c); // 6

        }
    }
}

one();

```

Line 14 is intriguing, at first glance it seems that `a` and `b` are not "inside" function three, so how can this code still work? To understand how the interpreter evaluates this code, we need to look at the scope chain of function three at the time line 14 was executed:



When the interpreter executes line 14: `alert(a + b + c)`, it resolves `a` first by looking into the scope chain and checking the first variable object, `three's [VO]`. It checks to see if `a` exists inside `three's [VO]` but can not find any property with that name, so moves on to check the next `[VO]`.

The interpreter keeps checking each `[VO]` in sequence for the existence of the variable name, in which case the value will be returned to the original evaluated code, or the program will throw a `ReferenceError` if none is found. Therefore, given the example above, you can see that `a`, `b` and `c` are all resolvable given function three's scope chain.

How does this work with closures?

In JavaScript, closures are often regarded as some sort of magical unicorn that only advanced developers can really understand, but truth be told it is just a simple understanding of the scope chain. A closure, as [Crockford](#) says, is simply:

An inner function always has access to the vars and parameters of its outer function, even after the outer function has returned...

The code below is an example of a closure:

```
function foo() {
  var a = 'private variable';
  return function bar() {
    alert(a);
  }
}

var callAlert = foo();

callAlert(); // private variable
```

CODE

The global context has a function named `foo()` and a variable named `callAlert`, which holds the returned value of `foo()`. What often surprises and confuses developers is that the private variable, `a`, is still available even after `foo()` has finished executing.

However, if we look at each of the context in detail, we will see the following:

```
// Global Context when evaluated
global.V0 = {
  foo: pointer to foo(),
  callAlert: returned value of global.V0.foo
  scopeChain: [global.V0]
}

// Foo Context when evaluated
foo.V0 = {
  bar: pointer to bar(),
  a: 'private variable',
  scopeChain: [foo.V0, global.V0]
}

// Bar Context when evaluated
bar.V0 = {
  scopeChain: [bar.V0, foo.V0, global.V0]
}
```

Now we can see by invoking `callAlert()`, we get the function `foo()`, which returns the pointer to `bar()`. On entering `bar()`, `bar.V0.scopeChain` is `[bar.V0, foo.V0, global.V0]`.

By alerting `a`, the interpreter checks the first VO in the `bar.V0.scopeChain` for a property named `a` but can not find a match, so promptly moves on to the next VO, `foo.V0`.

It checks for the existence of the property and this time finds a match, returning the value back to the `bar` context, which explains why the `alert` gives us '`private variable`' even though `foo()` had finished executing sometime ago.

By this point in the article, we have covered the details of the `scope chain` and its `lexical environment`, along with how `closures` and `variable resolution` work. The rest of this article looks at some interesting situations in relation to those covered above.

Wait, how does the prototype chain affect variable resolution?

JavaScript is [prototypal](#) by nature and almost everything in the language, except for `null` and `undefined`, are objects. When trying to access a property on an object, the interpreter will try to resolve it by looking for the existence of the property in the object. If it can't find the property, it will continue to look up the [prototype chain](#), which is an inherited chain of objects, until it finds the property, or traversed to the end of the chain.

This leads to an interesting question, does the interpreter resolve an object property using the `scope chain` or `prototype chain` first? It uses both. When trying to resolve a property or identifier, the `scope chain` will be used first to locate the object. Once the object has been found, the `prototype chain` of that object will then be traversed looking for the property name. Let's look at an example:

CODE

```

var bar = {};

function foo() {

    bar.a = 'Set from foo()';

    return function inner() {
        alert(bar.a);
    }
}

foo()(); // 'Set from foo()'

```

Line 5 creates the property `a` on the global object `bar`, and sets its value to '`Set from foo()`'. The interpreter looks into the scope chain and as expected finds `bar.a` in the global context. Now, let's consider the following:

```

var bar = {};

function foo() {

    Object.prototype.a = 'Set from prototype';

    return function inner() {
        alert(bar.a);
    }
}

foo()(); // 'Set from prototype()'

```

At runtime, we invoke `inner()`, which tries to resolve `bar.a` by looking in its scope chain for the existence of `bar`. It finds `bar` in the global context, and proceeds to search `bar` for a property named `a`. However, `a` was never set on `bar`, so the interpreter traverses the object's prototype chain and finds `a` was set on `Object.prototype`.

It is this exact behavior which explains identifier resolution; locate the object in the scope chain, then proceed up the object's prototype chain until the property is found, or returned `undefined`.

When to use Closures?

Closures are a powerful concept given to JavaScript and some of the most common situations to use them are:

- **Encapsulation**

Allows us to hide the implementation details of a context from outside scopes, while exposing a controlled public interface. This is commonly referred to as the [module pattern](#) or [revealing module pattern](#).

- **Callbacks**

Perhaps one of the most powerful uses for closures are callbacks. JavaScript, in the browser, typically runs in a single threaded event loop, blocking other events from starting until one event has finished. Callbacks allow us

to defer the invocation of a function, typically in response to an event completing, in a non-blocking manner. An example of this is when making an AJAX call to the server, using a callback to handle the response, while still maintaining the bindings in which it was created.

- **Closures as arguments**

We can also pass closures as arguments to a function, which is a powerful functional paradigm for creating more graceful solutions for complex code. Take for example a minimum sort function. By passing closures as parameters, we could define the implementation for different types of data sorting, while still reusing a single function body as a schematic.

When not to use Closures ?

Although closures are powerful, they should be used sparingly due to some performance concerns:

- **Large scope lengths**

Multiple nested functions are a typical sign that you might run into some performance issues. Remember, every time you need to evaluate a variable, the Scope Chain must be traversed to find the identifier, so it goes without saying that the further down the chain the variable is defined, the longer to lookup time.

Garbage collection

JavaScript is a garbage collected language, which means developers generally don't have to worry about memory management, unlike lower level programming languages. However, this automatic garbage collection often leads developers application to suffer from poor performance and memory leaks.

Different JavaScript engines implement garbage collection slightly different, since ECMAScript does not define how the implementation should be handled, but the same philosophy can apply across engines when trying to create high performance, leak free JavaScript code. Generally speaking, the garbage collector will try to free the memory of objects when they can not be referenced by any other live object running in the program, or are unreachable.

Circular references

This leads us to closures, and the possibility of circular references in a program, which is a term used to describe a situation where one object references another object, and that object points back to the first object. Closures are especially susceptible to leaks, remember that an inner function can reference a variable defined further up the scope chain even after the parent has finished executing and returned. Most JavaScript engines handle these situations quite well (damn you IE), but it's still worth noting and taking into consideration when doing your development.

For older versions of IE, referencing a DOM element would often cause you memory leaks. Why? In IE, the JavaScript (JScript ?) engine and DOM both have their own individual garbage collector. So when referencing a DOM element from JavaScript, the native collector hands off to the DOM and the DOM collector points back to native, resulting in neither collector knowing about the circular reference.

Summary

From working with many developers over the past few years, I often found that the concepts of scope chain and

closures were known about, but not truly understood in detail. I hope this article has helped to take you from knowing the basic concept, to an understanding in more detail and depth.

Going forward, you should be armed with all the knowledge you need to determine how the resolution of variables, in any situation, works when writing your JavaScript. Happy coding !

16. JavaScript's Undefined Explored

It sounds a simple concept, but how do you actually check that a variable or property in JavaScript really exists? What is the best way to do this? How do we cover all of the edge cases? First, let's look at what is undefined...

Overview of undefined

The value of a variable is given a type, and there are several built-in native types in JavaScript:

1. Undefined
2. Null
3. Boolean
4. String
5. Number
6. Object
7. Reference
8. etc...

Looking at 1, the built-in [Undefined](#) type can only ever have a single value, which is called *undefined*. This value is a primitive, and whenever a variable is declared it is assigned this *undefined* value, until you programmatically assign it a different value.

Also, whenever a function finishes executing and returns without a given value, it returns *undefined* by default.

```
var foo,
    bar = (function() {
        // do some stuff
    }()),
    baz = (function() {
        var hello;
        return hello;
    }());

typeof foo; // undefined
typeof bar; // undefined
typeof baz; // undefined
```

CODE

So when a variable is declared but not assigned a value, it is given a value of *undefined*. We should also note that *undefined* is a variable / property that is available in the global scope, that also has the value of *undefined*.

CODE

```
typeof undefined; // undefined

var foo;

foo === undefined; // true
```

However, the global variable *undefined* is *not* a reserved word and therefore can be redefined. Luckily as of ECMA 5, *undefined* is not permitted to be redefined, but in previous versions and older browsers it was possible to do the following:

CODE

```
typeof undefined; // undefined
undefined = 99;
typeof undefined; // number
```

What is this null business all about?

Take the following:

CODE

```
null == undefined // true
null !== undefined // true
```

Many people are confused by the above, the explanation is quite simple. The only real relationship between *null* and *undefined* is that they both evaluate to false during type coercion.

`So null == undefined // true` is because the `==` is not performing a strict comparison, whereas using the `!==` is more strict when comparing types. Whenever you see *null* as the value, it has always been programmatically assigned and never set by default.

Accessing properties on an object

When you try to use a property on an object that does not exist you will also get *undefined*, except for if you try to use the non-existent property as a function will sometimes raise an error.

CODE

```
var foo = {};

foo.bar; // undefined
foo.bar(); // TypeError
```

What happens if you want to tell the difference between a property that has a value *undefined* and a property that does not exist at all? Both `typeof` and `==` will give you a value of *undefined*.

Using the `in` operator will check does a certain property exist in an object:

CODE

```

var foo = {};

// undefined (Not good, bar has never been declared in the window object)
typeof foo.bar;

// false (Use this if you don't care about the prototype chain)
'bar' in foo;

// false (use this if you do care about the prototype chain)
foo.hasOwnProperty('bar');

```

Should you use `typeof` or `in` / `hasOwnProperty`?

It depends. Generally, if you want to test for the *existence* of a property then use `in` / `hasOwnProperty` and if you want to check for the value of a property / variable use `typeof` instead.

Let's recap with some examples

Check if a variable exists:

```
if (typeof foo !== 'undefined') {}
```

CODE

Check if a property on an object exists, regardless if it has been assigned a value or not:

```

// exists on the object, checks the prototype too
if ('foo' in bar) {}

// exists directly on the object, don't check the prototype
if (bar.hasOwnProperty('foo')) {}

```

CODE

Check if a property on an object exists, and the property has a value set (truthy or falsy)

```

var bar = {
  foo: false
};

if ('foo' in bar && typeof bar.foo !== 'undefined'){
  // bar.foo exists, and it contains a value which was programatically assigned
}

```

CODE

17. This

<https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/this>

A function's `this` keyword behaves a little differently in JavaScript compared to other languages. It also has some differences between [strict mode](#) and non-strict mode.

In most cases, the value of `this` is determined by how a function is called. It can't be set by assignment during execution, and it may be different each time the function is called. ES5 introduced the [bind](#) method to [set the value of a function's this regardless of how it's called](#), and ECMAScript 2015 introduced [arrow functions](#) whose `this` is lexically scoped (it is set to the `this` value of the enclosing execution context).

Syntax[Edit](#)

```
this
```

CODE

Global context[Edit](#)

In the global execution context (outside of any function), `this` refers to the global object, whether in strict mode or not.

```
console.log(this.document === document); // true

// In web browsers, the window object is also the global object:
console.log(this === window); // true

this.a = 37;
console.log(window.a); // 37
```

CODE

Function context[Edit](#)

Inside a function, the value of `this` depends on how the function is called.

Simple call

```
function f1(){
  return this;
}

f1() === window; // global object
```

CODE

In this case, the value of `this` is not set by the call. Since the code is not in strict mode, the value of `this` must always be an object so it defaults to the global object.

```
function f2(){
  "use strict"; // see strict mode
  return this;
}

f2() === undefined;
```

CODE

In strict mode, the value of `this` remains at whatever it's set to when entering the execution context. If it's not

defined, it remains undefined. It can also be set to any value, such as null or 42 or "I am not this".

Note: In the second example, this should be undefined, because f2 was called directly and not as a method or property of an object (e.g. window.f2()). This feature wasn't implemented in some browsers when they first started to support strict mode. As a result, they incorrectly returned the window object.

Arrow functions

In [arrow functions](#), this is set lexically, i.e. it's set to the value of the enclosing execution context's this . In global code, it will be set to the global object:

```
CODE
var globalObject = this;
var foo = (() => this);
console.log(foo() === globalObject); // true
```

It doesn't matter how foo is called, its this will stay as the global object. This also holds if it's called as a method of an object (which would usually set its this to the object), with call or apply or bind is used:

```
CODE
// Call as a method of an object
var obj = {foo: foo};
console.log(obj.foo() === globalObject); // true

// Attempt to set this using call
console.log(foo.call(obj) === globalObject); // true

// Attempt to set this using bind
foo = foo.bind(obj);
console.log(foo() === globalObject); // true
```

No matter what, foo 's this is set to what it was when it was created (in the example above, the global object). The same applies for arrow functions created inside other functions: their this is set to that of the outer execution context.

CODE

```
// Create obj with a method bar that returns a function that
// returns its this. The returned function is created as
// an arrow function, so its this is permanently bound to the
// this of its enclosing function. The value of bar can be set
// in the call, which in turn sets the value of the
// returned function.
var obj = { bar : function() {
            var x = (() => this);
            return x;
        }
    };

// Call bar as a method of obj, setting its this to obj
// Assign a reference to the returned function to fn
var fn = obj.bar();

// Call fn without setting this, would normally default
// to the global object or undefined in strict mode
console.log(fn() === obj); // true
```

In the above, the function(call it anonymous function A) assigned to `obj.bar` returns another function(call it anonymous function B) that is created as an arrow function. As a result, function B's `this` is permanently set to the `this` of `obj.bar` (function A)when called. When the returned function(function B) is called, its `this` will always be what it was set to initially. In the above code example, function B's `this` is set to function A's `this` which is `obj`, so it remains set to `obj` even when called in a manner that would normally set its `this` to `undefined` or the global object (or any other method as in the previous example in the global execution context).

As an object method

When a function is called as a method of an object, its `this` is set to the object the method is called on.

In the following example, when `o.f()` is invoked, inside the function `this` is bound to the `o` object.

CODE

```
var o = {
    prop: 37,
    f: function() {
        return this.prop;
    }
};

console.log(o.f()); // logs 37
```

Note that this behavior is not at all affected by how or where the function was defined. In the previous example, we defined the function inline as the `f` member during the definition of `o`. However, we could have just as easily defined the function first and later attached it to `o.f`. Doing so results in the same behavior:

CODE

```

var o = {prop: 37};

function independent() {
  return this.prop;
}

o.f = independent;

console.log(o.f()); // logs 37

```

This demonstrates that it matters only that the function was invoked from the `f` member of `o`.

Similarly, the `this` binding is only affected by the most immediate member reference. In the following example, when we invoke the function, we call it as a method `g` of the object `o.b`. This time during execution, `this` inside the function will refer to `o.b`. The fact that the object is itself a member of `o` has no consequence; the most immediate reference is all that matters.

CODE

```

o.b = {g: independent, prop: 42};
console.log(o.b.g()); // logs 42

```

this on the object's prototype chain

The same notion holds true for methods defined somewhere on the object's prototype chain. If the method is on an object's prototype chain, `this` refers to the object the method was called on, as if the method was on the object.

CODE

```

var o = {f:function(){ return this.a + this.b; }};
var p = Object.create(o);
p.a = 1;
p.b = 4;

console.log(p.f()); // 5

```

In this example, the object assigned to the variable `p` doesn't have its own `f` property, it inherits it from its prototype. But it doesn't matter that the lookup for `f` eventually finds a member with that name on `o`; the lookup began as a reference to `p.f`, so `this` inside the function takes the value of the object referred to as `p`. That is, since `f` is called as a method of `p`, its `this` refers to `p`. This is an interesting feature of JavaScript's prototype inheritance.

this with a getter or setter

Again, the same notion holds true when a function is invoked from a getter or a setter. A function used as getter or setter has its `this` bound to the object from which the property is being set or gotten.

```
function sum(){
    return this.a + this.b + this.c;
}

var o = {
    a: 1,
    b: 2,
    c: 3,
    get average(){
        return (this.a + this.b + this.c) / 3;
    }
};

Object.defineProperty(o, 'sum', {
    get: sum, enumerable:true, configurable:true});

console.log(o.average, o.sum); // logs 2, 6
```

As a constructor

When a function is used as a constructor (with the [new](#) keyword), its `this` is bound to the new object being constructed.

Note: while the default for a constructor is to return the object referenced by `this`, it can instead return some other object (if the return value isn't an object, then the `this` object is returned).

CODE

```
/*
 * Constructors work like this:
 *
 * function MyConstructor(){
 *   // Actual function body code goes here.
 *   // Create properties on |this| as
 *   // desired by assigning to them. E.g.,
 *   this.fum = "nom";
 *   // et cetera...
 *
 *   // If the function has a return statement that
 *   // returns an object, that object will be the
 *   // result of the |new| expression. Otherwise,
 *   // the result of the expression is the object
 *   // currently bound to |this|
 *   // (i.e., the common case most usually seen).
 * }
 */

function C(){
  this.a = 37;
}

var o = new C();
console.log(o.a); // logs 37

function C2(){
  this.a = 37;
  return {a:38};
}

o = new C2();
console.log(o.a); // logs 38
```

In the last example (C2), because an object was returned during construction, the new object that `this` was bound to simply gets discarded. (This essentially makes the statement "`this.a = 37;`" dead code. It's not exactly dead, because it gets executed, but it can be eliminated with no outside effects.)

call and apply

Where a function uses the `this` keyword in its body, its value can be bound to a particular object in the call using the [call](#) or [apply](#) methods that all functions inherit from `Function.prototype`.

CODE

```

function add(c, d){
  return this.a + this.b + c + d;
}

var o = {a:1, b:3};

// The first parameter is the object to use as
// 'this', subsequent parameters are passed as
// arguments in the function call
add.call(o, 5, 7); // 1 + 3 + 5 + 7 = 16

// The first parameter is the object to use as
// 'this', the second is an array whose
// members are used as the arguments in the function call
add.apply(o, [10, 20]); // 1 + 3 + 10 + 20 = 34

```

Note that with `call` and `apply`, if the value passed as `this` is not an object, an attempt will be made to convert it to an object using the internal `ToObject` operation. So if the value passed is a primitive like `7` or `'foo'`, it will be converted to an Object using the related constructor, so the primitive number `7` is converted to an object as if by `new Number(7)` and the string `'foo'` to an object as if by `new String('foo')`, e.g.

CODE

```

function bar() {
  console.log(Object.prototype.toString.call(this));
}

bar.call(7); // [object Number]

```

The bind method

ECMAScript 5 introduced [`Function.prototype.bind`](#). Calling `f.bind(someObject)` creates a new function with the same body and scope as `f`, but where `this` occurs in the original function, in the new function it is permanently bound to the first argument of `bind`, regardless of how the function is being used.

CODE

```

function f(){
  return this.a;
}

var g = f.bind({a:"azerty"});
console.log(g()); // azerty

var o = {a:37, f:f, g:g};
console.log(o.f(), o.g()); // 37, azerty

```

As a DOM event handler

When a function is used as an event handler, its `this` is set to the element the event fired from (some browsers do not follow this convention for listeners added dynamically with methods other than `addEventListener`).

CODE

```
// When called as a listener, turns the related element blue
function bluify(e){
  // Always true
  console.log(this === e.currentTarget);
  // true when currentTarget and target are the same object
  console.log(this === e.target);
  this.style.backgroundColor = '#A5D9F3';
}

// Get a list of every element in the document
var elements = document.getElementsByTagName('*');

// Add bluify as a click listener so when the
// element is clicked on, it turns blue
for(var i=0 ; i<elements.length ; i++){
  elements[i].addEventListener('click', bluify, false);
}
```

In an in-line event handler

When code is called from an in-line [on-event handler](#), its `this` is set to the DOM element on which the listener is placed:

CODE

```
<button onclick="alert(this.tagName.toLowerCase());">
  Show this
</button>
```

The above alert shows `button`. Note however that only the outer code has its `this` set this way:

CODE

```
<button onclick="alert((function(){return this;}))">
  Show inner this
</button>
```

In this case, the inner function's `this` isn't set so it returns the global/window object (i.e. the default object in non-strict mode where `this` isn't set by the call).

18. Changing The Execution Context Of JavaScript Functions Using Call() And Apply()

Yesterday, as I writing about [changing the execution context of self-executing functions in JavaScript](#), I realized that I didn't have a good general post on what execution context was or how it can be changed. I've talked about it a number of times, but never in a really cohesive way. In JavaScript, all function bodies have access to the "this" keyword. The "this" keyword is the context of the function execution. By default, "this" is a reference to the object on which a particular function is called (in JavaScript all functions are bound to an object). We can, however, use `call()` and `apply()` to change this binding at runtime.

First, let's clarify the fact that all functions in JavaScript are actually "methods." That is, they are all properties of

some object. We can define functions that appear to be free-floating; however, these free-floating functions are implicitly created as properties of the global scope (ie. the Window object in the web browser). This means that functions defined without an explicit context can be accessed as a window property:

```
// Define the free-floating function.  
function someMethod(){ ... }  
  
// Access it as a property of Window.  
window.someMethod();
```

CODE

Now that we understand that all functions in JavaScript are properties of an object, we can talk about the default binding of the execution context. By default, a JavaScript function is executed in the context of the object for which it is a property. That is, within the function body, the "this" keyword is a reference to the parent object. As such, in the following code:

```
sarah.sayHello();
```

CODE

... the "this" keyword within the sayHello() method is a reference to the sarah object.

Similarly, in this code:

```
getScreenResolution();
```

CODE

... the "this" keyword within the getScreenResolution() function is a reference to the window object (since unbound functions are implicitly bound to the global scope).

If we want to, we can dynamically change the execution context of any method by using either call() or apply(). Both of these functions can be used to bind the "this" keyword to an explicit context. While similar in outcome, these two functions differ in their method signature:

- method.**call**(newThisContext, Param1, ..., Param N)
- method.**apply**(newThisContext, [Param1, ..., Param N]);

As you can see, call() takes a new context binding and N inline arguments. apply(), on the other hand, takes a new context binding and then an array of N arguments. The difference here is simply how the arguments get passed to the method in question. In both cases, the "this" keyword of the function body will refer to the newThisContext object provided during invocation.

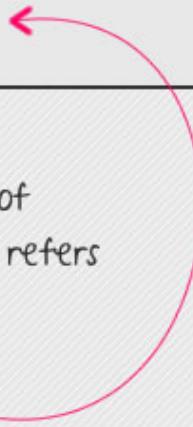
Window

Object

Method()

When called as a property of Object, the "this" keyword refers to the Object instance.

`Object.method() ==> THIS`

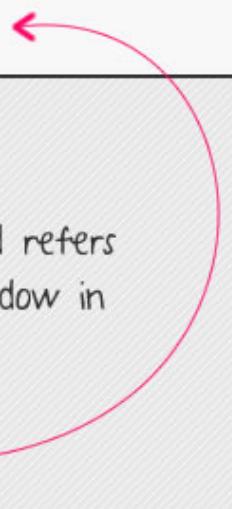


Window

Method()

When called as an unbound function, the "this" keyword refers to the global scope (ie. Window in the browser context).

`method() ==> THIS`



Window

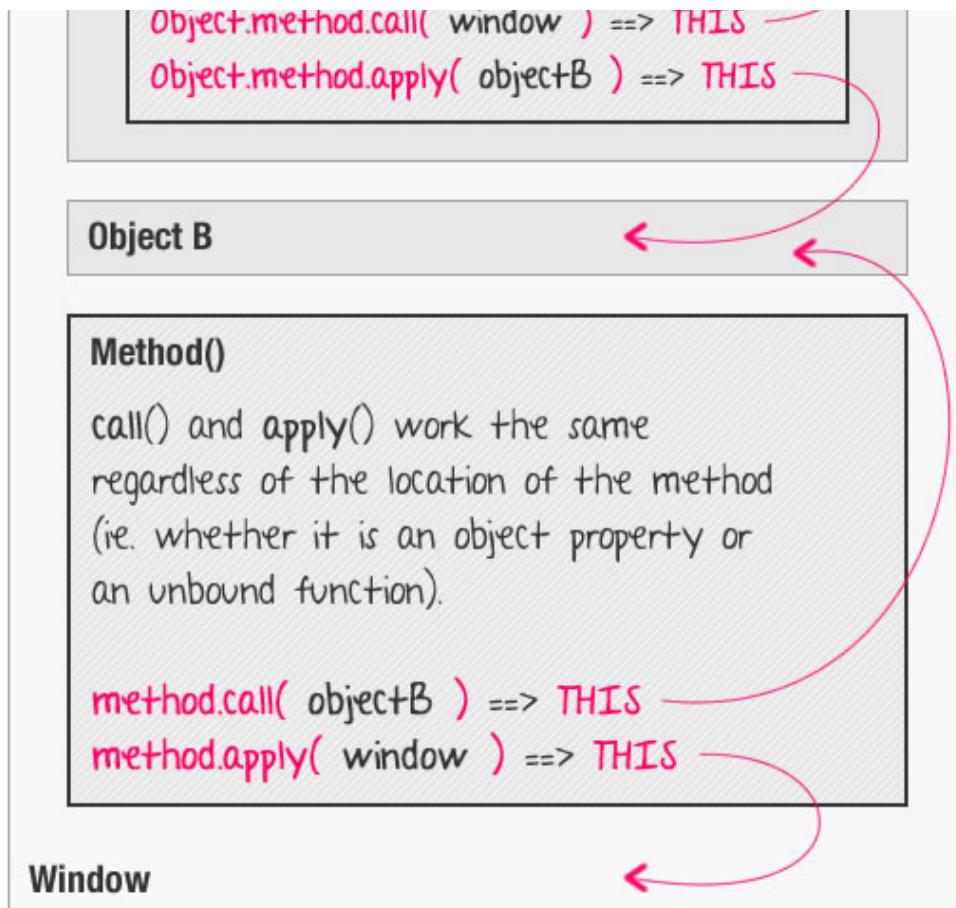
Object

Method()

`Object.method() ==> THIS`

When invoked with `call()` or `apply()`, the "this" keyword refers to the specified context (ie. the first argument).





Let's explore this concept with some code. In the following demo, we've got a few objects and a function. We're going to use `call()` and `apply()` to dynamically change the execution context of the given function at runtime, regardless of where the function happens to live.

CODE

```
<!DOCTYPE html>
<html>
<head>
<title>Changing Execution Context In JavaScript</title>

<script type="text/javascript">

// Create a global variable for context (this lives in the
// global scope - window).
var context = "Global (ie. window)";

// Create an object.
var objectA = {
context: "Object A"
};

// Create another object.
var objectB = {
context: "Object B"
};

// -----
// ----- //


// Define a function that uses an argument AND a reference
// to this THIS scope. We will be invoking this function
// using a variety of approaches.
function testContext( approach ){

console.log( approach, "==> THIS ==>", this.context );

}

// -----
// ----- //


// Invoke the unbound method with standard invocation.
testContext( "testContext()" );

// Invoke it in the context of Object A using call().
testContext.call(
objectA,
".call( objectA )"
);

// Invoke it in the context of Object B using apply().
testContext.apply(
objectB,
[ ".apply( objectB )" ]
);

// ----- //
```

```
// ----- //

// Now, let's set the test method as an actual property
// of the object A.
objectA.testContext = testContext;

// -----
// ----- //

// Invoke it as a property of object A.
objectA.testContext( "objectA.testContext()" );

// Invoke it in the context of Object B using call.
objectA.testContext.call(
objectB,
"objectA.testContext.call( objectB )"
);

// Invoke it in the context of Window using apply.
objectA.testContext.apply(
window,
[ "objectA.testContext.apply( window )" ]
);

</script>
</head>
<body>

</body>
</html>
```

As you can see, we have a property, context, defined on window, objectA, and objectB. We then use call() and apply() to execute the testContext() method in the context of the above objects. Doing so results in the following console output:

```
testContext() ==> THIS ==> Global (ie. window)
.call( objectA ) ==> THIS ==> Object A
.apply( objectB ) ==> THIS ==> Object B
objectA.testContext() ==> THIS ==> Object A
objectA.testContext.call( objectB ) ==> THIS ==> Object B
objectA.testContext.apply( window ) ==> THIS ==> Global (ie. window)
```

As you can see, the "this" keyword within the function body is bound to the first argument of either the call() or apply() functions. This is true regardless of where the invoked method resides.

.... hmm, not sure how to end this. I'm feeling like this blog post wasn't very well articulated. I wrote, however, so I might as well post it.

19. Using Method Chaining With The Revealing Module Pattern In JavaScript

Yesterday, I looked at [creating null-prototype objects in Node.js](#). As part of that exploration, I created a very simple

cache class with public methods that could be chained. However, since the public methods were exposed using the revealing module pattern, it dawned on me that the "this" reference works; but, it works in a very interesting way. When using method chaining, in conjunction with the revealing module pattern, you can only use "this" to refer to public methods.

To explain this in better detail, I've re-written my simple cache class to make the syntax a little more obvious:

```
// Create an instance of our cache and set some keys. Notice that the [new] operator
// is optional since the SimpleCache (and revealing module pattern) doesn't use
// prototypical inheritance. And, we can use method-chaining to set the cache keys.
var cache = SimpleCache()
.set( "foo", "Bar" )
.set( "hello", "world" )
.set( "beep", "boop" )
;

console.log( cache.has( "beep" ) );

// -----
// ----- //



// I provide a super simple cache container.
function SimpleCache() {

    // Create an object without a prototype so that we don't run into any cache-key
    // conflicts with native Object.prototype properties.
    var cache = Object.create( null );

    // Reveal the public API.
    return({
        get: get,
        has: has,
        remove: remove,
        set: set
    });
}

// ---
// PUBLIC METHODS.
// ---


// I get the value cached at the given key; or, undefined.
function get( key ) {

    return( cache[ key ] );
}

// I check to see if the given key has a cached value.
function has( key ) {

    return( key in cache );
}

// I remove the given key (and associated value) from the cache.
// --
// NOTE: Returns [this] for method chaining.
function remove( key ) {

    delete( cache[ key ] );
}
```

```
// CAUTION: In this context, [this] does not refer to the SimpleCache instance;
// rather, it refers to the public API that was "revealed". As such, method
// chaining can only work in conjunction with "public" methods.
return( this );

}

// I cache the given value at the given key.
// --
// NOTE: Returns [this] for method chaining.
function set( key, value ) {

cache[ key ] = value;

// CAUTION: In this context, [this] does not refer to the SimpleCache instance;
// rather, it refers to the public API that was "revealed". As such, method
// chaining can only work in conjunction with "public" methods.
return( this );

}

}
```

Notice that the two methods, `.remove()` and `.set()`, both return "this". In a typical "prototypal context," you might expect "this" to refer to the instance of `SimpleCache`. However, since we're using the revealing module pattern, there may not be an "instance" of `SimpleCache` - at least not in the traditional sense. In this case, since we're "revealing" a public API by returning a completely new object literal, [the "this" will refer specifically to the object literal](#), aka, our public API.

```
// I provide a super simple cache container.
function SimpleCache() {

    // Create an object without a prototype so that we don't run into any cache-key
    // conflicts with native Object.prototype properties.
    var cache = Object.create( null );

    // Reveal the public API.
    return({
        get: get,
        has: has,
        remove: remove,
        set: set
    });
}
```

```
// ---
// PUBLIC METHODS.
// ---
```

```
// I get the value cached
function get( key ) {
    return( cache[ key ] );
}

// I check to see if the given key exists.
function has( key ) {
    return( key in cache );
}
```

```
// I remove the given key (and associated value) from the cache.
```

```
// --
// NOTE: Returns [this] for method chaining.
function remove( key ) {

    delete( cache[ key ] );

    // CAUTION: In this context, [this] does not refer to the SimpleCache instance
    // rather, it refers to the public API that was "revealed". As such, method
    // chaining can only work in conjunction with "public" methods.
    return( this );
}
```

```
// I cache the given value at the given key.
```

```
// --
// NOTE: Returns [this] for method chaining.
function set( key, value ) {

    cache[ key ] = value;

    // CAUTION: In this context, [this] does not refer to the SimpleCache instance
    // rather, it refers to the public API that was "revealed". As such, method
```

When you return "this" in the revealing module pattern, it refers to the revealed API — not to the SimpleCache "instance". In fact, there may not even be a SimpleCache "instance", in the traditional sense, only a function closure.

```
// chaining can only work in conjunction with "public" methods.
return( this );
}

}
```

While I don't have any private methods in this example, I hope you can see that method chaining won't work with private methods. In the revealing module pattern, private methods (and variables) are only available due to [the lexical binding of the SimpleCache\(\) function which "closes over" the function references](#). As such, if I attempted to return "this" from a private function, I would lose access to everything but the public API.

That said, I would assume that the vast majority of use-cases for method chaining sit with public methods rather than with private methods. As such, I don't really consider this a problem. But, if you're going to be combining the revealing module patterns with method chaining, you really are getting knee-deep into the wonderfully flexible world of JavaScript! Proceed with much joy and some caution!

20. Functions and execution contexts in JavaScript

[Functions and execution contexts in JavaScript](#)

Posted on 2/24/2011 by

Sergio Cinos Senior Architecture Engineer

Functions are the main building block of JavaScript. Functions define the behaviour of things like closures, 'this', global variables vs. local variables... Understanding the functions is the first step to truly understand how JavaScript works.

As we already know, functions can access variables declared 'outside' the current function's scope, global variables, and as well as variables declared inside the function and those passed in as arguments. Also, the variable 'this' points to 'the container object'. All of these form an 'environment' for our function that defines which variables are accessible by the function and their values. Some parts of this 'environment' are defined when the function is defined and others when the function is called.

Understanding what happens internally when a function is called may be a little difficult the first time, mainly due to the technical details and nomenclature. In order to be clear, some parts of this article are simplifications of the technical explanation. The specific details can be found in section 10.1.6 of [ECMA 262 \(3rd edition\)](#).

When a function is **called**, an `ExecutionContext` is created. This context defines a big part of the function's 'environment', so let's see how this is constructed (the order is important):

1. The `arguments` property is created. This is an array-like object with integer keys, each one referencing a value passed into the function call, in that same order. This object also contains `length` (number of values passed in the function call) and `callee` (reference to the function being called) properties.
2. Function's scope is created, using `[[scope]]` property and this `ExecutionContext`. More details on this later.
3. Variable instantiation takes place now. It has 3 substeps (also in order):
 1. `ExecutionContext` gets a property for each argument defined in the function signature. If there is a value for that position in the `arguments` object, the value is assigned to the new created property. Otherwise, the property will have the value `undefined`.

2. The function's body is scanned to detect `FunctionDeclarations`. Then, those functions are created and assigned as a property to `ExecutionContext` using defined names.
3. The function's body is scanned to detect variable declarations. Those variables are saved as a property in `ExecutionContext` and initialized as `undefined`.
4. The `this` property is created. Its value depends on how the function was called:
 1. Regular function (`myFunction(1,2,3)`). The value of `this` points to the global object (i.e. `window`).
 2. Object method (`myObject.myFunction(1,2,3)`). The value of `this` points to the object containing the function (i.e. the object before the dot). The value is `myObject` in our example.
 3. Callback for `setTimeout()` or `setInterval()`. The value of `this` points to the global object (i.e. `window`).
 4. Callback for `call()` or `apply()`. The value of `this` is the first argument of `call()` / `apply()`.
 5. As constructor (`new myFunction(1,2,3)`). The value of `this` is an empty object with `myFunction.prototype` as prototype.

Let's see an example of this process in pseudo-code:

JavaScript code:

```
function foo (a, b, c) {
    function z(){alert(`Z!`);}
    var d = 3;
}
foo(`foo','bar');
```

CODE

`ExecutionContext` in the `foo()` call: Step 1: `arguments` is created

```
ExecutionContext: {
  arguments: {
    0: `foo`, 1: `bar`,
    length: 2, callee: function() //Points to foo function
  }
}
```

CODE

Step 3a: variable instantiation, `arguments`

```
ExecutionContext: {
  arguments: {
    0: `foo`, 1: `bar`,
    length: 2, callee: function() //Points to foo function
  },
  a: `foo`, b: `bar`, c: undefined
}
```

CODE

Step 3b: variable instantiation, functions

CODE

```
ExecutionContext: {
    arguments: {
        0: `foo', 1: `bar',
        length: 2, callee: function() //Points to foo function
    },
    a: `foo', b: `bar', c: undefined,
    z: function() //Created z() function
}
```

Step 3c: variable instantiation, variables

CODE

```
ExecutionContext: {
    arguments: {
        0: `foo', 1: `bar',
        length: 2, callee: function() //Points to foo function
    },
    a: `foo', b: `bar', c: undefined,
    z: function(), //Created z() function,
    d: undefined
}
```

Step 4: set this value

CODE

```
ExecutionContext: {
    arguments: {
        0: `foo', 1: `bar',
        length: 2, callee: function() //Points to foo function
    },
    a: `foo', b: `bar', c: undefined,
    z: function(), //Created z() function,
    d: undefined,
    this: window
}
```

After the creation of `ExecutionContext`, the function starts running its code from the first line until it finds a `return` or the function ends. Every time this code tries to access a variable, it is read from the `ExecutionContext` object.

In JavaScript, every single instruction is executed in an `ExecutionContext`. As we have seen, all code within any function will have an `ExecutionContext` associated, no matter how the function was created or invoked. Therefore, every single statement inside any function is executed in that function's `ExecutionContext`. The code that does not belong to any function but the Global code (code executed inline, loaded via `<script>`, executed through `eval()` ...) is associated to a special context called `GlobalExecutionContext`. This context works very much like `ExecutionContext`, but since we do not have function arguments, only step 3) and 4) take place (`this` points to global object, usually `window`). In conclusion, every JavaScript statement runs 'inside' an `ExecutionContext`.

As the execution of our program goes on, it will 'jump' from one function to another (via direct calls, DOM events, timers...). As each function has its own `ExecutionContext`, these function calls will create a stack of contexts. For example, let's see the following code.

```
<script>
    function a() {
        function b() {
            var c = {
                d: function() {
                    alert(1);
                }
            };
            c.d();
        }
        b.call({});
    }
    a();
</script>
```

When the JavaScript engine is about to execute the `alert()` function, the `ExecutionContext` stack is:

1. d() Execution context
2. b() Execution context
3. a() Execution context
4. Global execution context

The most important part of the `ExecutionContext` stack happens when the function is defined. The key idea to fully understand JavaScript contexts is that every function declaration is executed inside a `ExecutionContext` (in the previous example, function `b()` is declared and created inside `a()` `ExecutionContext`). Everytime a function is created, the current `ExecutionContext` stack is saved in the `[[scope]]` property of the function itself. All of this happens at function creation. This stack is preserved and tied to the newly created function, even if the original function has finished (this can happen if the original function returns the created function as result, for example).

Now we can explain the step 2) of `ExecutionContext` creation. At this point, a new `ExecutionContext` stack is created, pushing the function's `ExecutionContext` on top of the mentioned `[[scope]]` property. This stack is also called 'scope chain'. Note that the `ExecutionContext` stack can be (and usually is) different from the calling stack. The later is defined when the functions are called, and the former when they are defined. For example: function `a()` calls function `b()` which calls function `c()`. This would be the calling stack that can be inspected in any debug tool. However, function `c()` might have been created inside a function `d()`. The `ExecutionContext` related to `d()` is part of the scope chain, but not of the calling stack.

When the code within the function is looking for a variable, the scope chain is examined. The engine starts searching for it in the first `ExecutionContext` in the chain. As it is the function's `ExecutionContext` itself, it corresponds to the functions arguments, declared variables, etc. If it is not found, the engine will then search for the variable in the next `ExecutionContext` in the stack and so on until it reaches the end of the chain. If it still not found, it returns `undefined` as the variable value.

And that is all about function internals: just creating `ExecutionContexts` and stacking them. Let's sum up the bullet points about functions and execution contexts:

- The value of `this` is not coupled to the function nor is a 'special' property, but behaves more like a regular argument. It is defined when the function is called, so the same function can be executed with different values for `this`.
- `arguments` is not an array, just a regular object with numbers as property names. So it does not inherit the array methods like `push()`, `concat()`, `slice()` ...

- Variables are actually defined in step 3c), no matter where they are defined in the function code. However, initialization takes place when the execution flow reaches the instruction where they are initialized. That is why in our example `d` points to `undefined`. It will point to 3 when the execution code reaches the second line of the function code.
- You can call a function before it is defined. It is allowed by step 3b) in `ExecutionContext` creation (not true for `FunctionExpressions`)
- All inner functions declarations are created at the `ExecutionContext` step. So an unreachable function declaration will be always created. For example:

```
function foo() {
    if (false) {
        function bar() {alert(1);}
    }
    bar();
}
```

CODE

It will work (working in IE8, Chrome and Safari5, not in Firefox) because `bar()` is created at `ExecutionContext`, before starting function's code and evaluating the `if`.

- Variables can be hidden. As all the steps take place in order, later steps can overwrite the job done by previous ones. For example, if we define an argument called `foo` at function signature, and inside that function we declare another function called `foo` too, the later will 'hide' the former when the `ExecutionContext` is finally created.
- Closures: a function can access its 'parent' function's variables. When asking for a variable, the value is not found in our current `ExecutionContext` but on the next context in the stack: the context from our 'parent' function. You can even build 'multilevel' closures that uses the data from its parent, grandparent... functions.
- JavaScript has global variables. In this case, the value is found in the last item of the chain, the `GlobalExecutionContext` (this is the reason why global variable access is slow; the engine must search for the variable in each context in the stack trace before reaching the global context). Also, you can use semi-global variables: if different functions have a common `ExecutionContext` in their stacks, any variable declared in that common `ExecutionContext` will be available for all those functions like a global variable.

JavaScript core is actually `ExecutionContexts` and scope chains, as most of the language features raise from the contexts behaviour. If you get used to designing your program as an interaction of contexts, your code will be much simpler and more natural. For example, with contexts in mind, a mixin-based inheritance system is very easy to implement (as opposed to many other languages). Most of the JavaScript loading libraries rely on context management to load modules without polluting the global context. To sum up, it is by thinking in contexts and scope chains (and not in functions and/or objects) that JavaScript unleashes its power. Make sure to understand them as deeply as you can.

Further reading:

- [Javascript Closures, comp.lang.javascript FAQ](#)
- [Understanding JavaScript's this keyword](#)
- [The JavaScript arguments object...and beyond](#)
- [ECMA-262-3 in detail. Chapter 1. Execution Contexts](#)
- [JavaScript Scoping and Hoisting](#)

21. Understanding JavaScript's this keyword

([In Portuguese](#))

The JavaScript `this` keyword is ubiquitous yet misconceptions abound.

What you need to know

Every execution context has an associated `ThisBinding` whose lifespan is equal to that of the execution context and whose value is constant. There are three types of execution context: global, function and evaluation. Here's a tabular summary followed by a little more detail, and some examples:

<i>Execution Context</i>	<i>Syntax of function call</i>	<i>Value of this</i>
Global	n/a	global object (e.g. <code>window</code>)
Function	Method call: <code>myObject.foo();</code>	<code>myObject</code>
Function	Baseless function call: <code>foo();</code>	global object (e.g. <code>window</code>) (<code>undefined</code> in strict mode)
Function	Using call: <code>foo.call(context, myArg);</code>	<code>context</code>
Function	Using apply: <code>foo.apply(context, [myArgs]);</code>	<code>context</code>
Function	Constructor with new: <code>var newFoo = new Foo();</code>	the new instance (e.g. <code>newFoo</code>)
Evaluation	n/a	value of <code>this</code> in parent context

1. Global context

`this` is bound to the global object (`window` in a browser)

```
1 alert( this ); //window
```

2. Function context

There are at least 5 ways to invoke a function. The value of `this` depends on the method of invocation

a) Invoke as a method

`this` is the [baseValue](#) of the property reference

CODE

```

var a = {
    b : function () {
        return this;
    }
};
a.b(); //a;
a['b'](); //a;
var c= {};
c.d = a.b;
c.d(); //c

```

b) Invoke as baseless function call

`this` is the global object (or `undefined` in strict mode)

CODE

```

var a = {
    b: function() {
        return this;
    }
};

var foo = a.b;
foo(); //window

var a = {
    b: function() {
        var c = function() {
            return this;
        };
        return c();
    }
};

a.b(); //window

```

The same applies to self invoking functions:

CODE

```

var a = {
    b: function() {
        return (function() {return this;})();
    }
};

a.b(); //window

```

c) Invoke using Function.prototype.call

`this` is passed by argument

d) Invoke using Function.prototype.apply

`this` is passed by argument

CODE

```

var a = {
  b: function() {
    return this;
  }
};

var d = {};

a.b.apply(d); //d

```

e) Invoke a constructor using new

`this` is the newly created object

CODE

```

var A = function() {
  this.toString = function(){return "I'm an A"};
};

new A(); //"I'm an A"

```

3. Evaluation context

`this` value is taken from the `this` value of the calling execution context

CODE

```

alert(eval('this==window')); //true - (except firebug, see above)

var a = {
  b: function() {
    eval('alert(this==a)');
  }
};

a.b(); //true;

```

What you might want to know

This section explores the process by which `this` gets its value in the functional context -- using [ECMA-262 version 5.1](#) as a reference.

Lets start with the ECMAScript definition of `this` :

The `this` keyword evaluates to the value of the `ThisBinding` of the current execution context.

from [ECMA 5.1, 11.1.1](#)

How is `ThisBinding` set?

Each function defines a `[[Call]]` internal method ([ECMA 5.1, 13.2.1 \[\[Call\]\]](#)) which passes invocation values to the function's execution context:

The following steps are performed when control enters the execution context for function code contained in function object F, a caller provided `thisValue`, and a caller provided `argumentsList`:

1. If the function code is strict code, set the `ThisBinding` to `thisValue`.

2. Else if `thisValue` is null or undefined, set the `ThisBinding` to the global object.
3. Else if `Type(thisValue)` is not Object, set the `ThisBinding` to `ToObject(thisValue)`.
4. Else set the `ThisBinding` to `thisValue`

from [ECMA 5.1, 10.4.3 Entering Function Code](#) (slightly edited)

In other words `ThisBinding` is set to the object coercion of the abstract argument `thisValue`, or if `thisValue` is undefined, the global object (unless running in strict mode in which case `thisValue` is assigned to `ThisBinding` as-is)

So where does `thisValue` come from?

Here we need to go back to our 5 types of function invocation:

- 1. Invoke as a method**
- 2. Invoke as baseless function call**

in ECMAScript parlance these are *Function Calls* and have two components: a *MemberExpression* and an *Arguments* list.

1. Let `ref` be the result of evaluating `MemberExpression`.
2. Let `func` be `GetValue(ref)`.
6. If `Type(ref)` is Reference, then
 - a. If `IsPropertyReference(ref)` is true
 - i. Let `thisValue` be `GetBase(ref)`.
 - b. Else, the base of `ref` is an Environment Record
 - i. Let `thisValue` be the result of calling the `ImplicitThisValue` concrete method of `GetBase(ref)`.
8. Return the result of calling the `[[Call]]` internal method on `func`, providing `thisValue` as the `this` value and providing the list `argList` as the argument values

from [ECMA 5.1, 11.2.3 Function Calls](#)

So, in essence, `thisValue` becomes the baseValue of the function expression (see step 6, above).

In a method call the function is expressed as a property, so the `baseValue` is the identifier preceding the dot (or square bracket).

```
foo.bar(); // foo assigned to thisValue
foo['bar'](); // foo assigned to thisValue
```

CODE

```
var foo = {
  bar:function() {
    //Comments apply to example invocation only)
    //MemberExpression = foo.bar
    //thisValue = foo
    //ThisBinding = foo
    return this;
  }
};
foo.bar(); //foo
```

A baseless function is either a function declaration or a variable -- in either case the `baseValue` is the *Environment Record* (specifically a *Declarative Environment Record*). [ES 5.1, 10.2.1.1.6](#) tells us that the `ImplicitThisValue` of a

Declarative Environment Record is undefined.

Revisiting [10.4.3 Entering Function Code](#) (see above) we see that unless in strict mode, an undefined `thisValue` results in a `ThisBinding` value of global object. So `this` in a baseless function invocation will be the global object. In strict mode the `ThisBinding` remains undefined.

In full...

```
var bar = function() {
    //Comments apply to example invocation only)
    //MemberExpression = bar
    //thisValue = undefined
    //ThisBinding = global object (e.g. window)
    return this
};
bar(); //window
```

CODE

3. Invoke using `Function.prototype.apply`

4. Invoke using `Function.prototype.call`

(specifications at [15.3.4.3 Function.prototype.apply](#) and [15.3.4.4 Function.prototype.call](#))

These sections describe how, in call and apply invocations, the actual value of the function's `this` argument (i.e. its first argument) is passed as the `thisValue` to [10.4.3 Entering Function Code](#). (Note this differs from ECMA 3 where primitive `thisArg` values undergo a `ToObject` transformation, and null or undefined values are converted to the global object -- but the difference will normally be negligible since the value will undergo identical transformations in the target function invocation (as we've already seen in [10.4.3 Entering Function Code](#)))

5. Invoke a constructor using `new`

When the `[[Construct]]` internal method for a Function object `F` is called with a possibly empty list of arguments, the following steps are taken:

1. Let `obj` be a newly created native ECMAScript object.
2. Let `result` be the result of calling the `[[Call]]` internal property of `F`, providing `obj` as the `thisValue` and providing the argument list passed into `[[Construct]]` as `args`.
3. Return `obj`.

from [ECMA 5.1, 13.2.2 \[\[Construct\]\]](#)

This is pretty clear. Invoking the constructor with `new` creates an object that gets assigned as the `thisValue`. It's also a radical departure from any other usage of `this`.

House Cleaning

Strict mode

In ECMAScript's strict mode, the `thisValue` is not coerced to an object. A `this` value of `null` or `undefined` is not converted to the global object and primitive values are not converted to wrapper objects

The bind function

`Function.prototype.bind` is new in ECMAScript 5 but will already be familiar to users of major frameworks. Based on `call/apply` it allows you to prebake the `thisValue` of an execution context using simple syntax. This is especially

useful for event handling code, for example a function to be invoked by a button click, where the `ThisBinding` of the handler will default to the `baseValue` of the property being invoked -- i.e. the button element:

```
//Bad Example: fails because ThisBinding of handler will be button
var sorter = {
    sort: function() {
        alert('sorting');
    },
    requestSorting: function() {
        this.sort();
    }
}
$('sortButton').onclick = sorter.requestSorting;
```

CODE

```
//Good Example: sorter baked into ThisBinding of handler
var sorter = {
    sort: function() {
        alert('sorting');
    },
    requestSorting: function() {
        this.sort();
    }
}
$('sortButton').onclick = sorter.requestSorting.bind(sorter);
```

CODE

22. The JavaScript arguments object...and beyond

Spare a thought for JavaScript's `arguments` object. It wants so desperately to be an array. It walks like an array, quacks like an array but flies like a turkey. During the early years of the language Brendan Eich came close to rewriting `arguments` as an array until ECMA came along and clipped its wings forever.

In spite of all this (or maybe because of it) we love the `arguments` object. In this article I'll explore its niftyness and its quirkiness and I'll finish up by looking at the likely successors: `rest` and `spread` ...

The arguments object

When control enters the execution context of a function an `arguments` object is created. The `arguments` object has an array-like structure with an indexed property for each passed argument and a `length` property equal to the total number of parameters supplied by the caller. Thus the `length` of the `arguments` object can be greater than, less than or equal to the number of formal parameters in the function definition (which we can get by querying the function's `length` property):

CODE

```

function echoArgs(a,b) {
    return arguments;
}

//number of formal parameters...
echoArgs.length; //2

//length of argument object...
echoArgs().length; //0
echoArgs(5,7,8).length; //3

```

Binding with named function parameters

Each member of the `arguments` object shares its value with the corresponding named parameter of the function -- so long as its index is less than the number of formal parameters in the function.

[ES5 clause 10.6 \(note 1\)](#) puts it like this:

For non-strict mode functions the array index [...] named data properties of an arguments object whose numeric name values are less than the number of formal parameters of the corresponding function object initially share their values with the corresponding argument bindings in the function's execution context. This means that changing the property changes the corresponding value of the argument binding and vice-versa

CODE

```

(function(a) {
    console.log(arguments[0] === a); //true
    console.log(a); //1

    //modify argument property
    arguments[0] = 10;
    console.log(a); //10

    //modify named parameter variable
    a = 20;
    console.log(arguments[0]); //20
})(1,2)

```

Argument properties whose index is greater than or equal to the number of formal parameters (i.e. additional arguments which do not correspond to named parameters) are not bound to any named parameter value. Similarly if a function call does not supply an argument for every named parameter, the unfilled parameters should not be bound to the `arguments` object and their values cannot be updated by modifying the `arguments` objects...

CODE

```
//Invoke a three argument function but only pass two arguments
(function(a, b, c) {
    //arguments' has two members
    console.log(arguments.length); //2

    //Updating arguments[2] should do not modify named param
    arguments[2] = 10;
    console.log(c); //undefined
})(1,2);

(function(a, b, c) {
    //Assigning to 'c' should not populate 'arguments' object
    c = 10;
    console.log('2' in arguments); //false
})(1,2)
```

...well according to the ES5 spec at least. Unfortunately the Chrome browser doesn't comply. It creates an `arguments` member for every named parameter, regardless of whether the argument was actually passed (this is a known [issue](#))

CODE

```
//CHROME BROWSER ONLY...

(function(a, b, c) {
    //Updating arguments[2] should do not modify named param
    arguments[2] = 10;
    console.log(c); //10!!
})(1,2);

(function(a, b, c) {
    //Assigning to 'c' should not populate 'arguments' object
    c = 10;
    console.log('2' in arguments); //true!!
})(1,2)
```

There's another [bug](#) related to Chrome's over-reaching `arguments` object. Deleting supposedly non-existent members of the `arguments` object will cause the corresponding named (but not passed) parameter value to be wiped out:

CODE

```
var cParam = (function(a, b, c) {
    c = 3;
    delete arguments[2];
    return c;
})(1,2);

cParam;
// Chrome -> undefined
// Other browsers -> 3
```

arguments.callee

Every instance of `arguments` has a `callee` property which references the currently invoking function. The strict mode of ES5 disallows access to `arguments.callee`

arguments.caller

In supported implementations, every instance of `arguments` has a `caller` property which references the function (if any) from which the current function was invoked. There is only patchy vendor support for `arguments.caller` and it is not standardized by ECMA except to explicitly disallow access in the strict mode.

More quirkiness

1) An `arguments` object will not be created if `arguments` is the name of a formal parameter or is used as a variable or function declaration within the function body:

```

function foo(a, arguments) {
    return arguments;
};

foo(1); //undefined

function foo(a, b) {
    var arguments = 43;
    return arguments
};

foo(1, 2); //43

```

CODE

2) The SpiderMonkey engine (used by Firefox) supplies a secret value at `arguments[0]` when invoking `valueOf`. The value will be "number" if the object is to be coerced to a number, otherwise undefined.

Thanks to [Andrea Giammarchi](#) for the following example

```

//FIREFOX BROWSER ONLY...
1
2 var o = {
3     push:[].push,
4     length:0,
5     toString:[].join,
6     valueOf: function (){
7         return arguments[0] == "number" ? this .length :
8         this .toString();
9     }
10 };
11
12 o.push(1, 2, 3);
13
14 o.toString(); // "1,2,3"
15 (o*1).toString(); // 3

```

Arrays vs. arguments

As noted, the `arguments` object is not an array. It is not a product of the `Array` constructor and it lacks all of the standard methods of `Array`. Moreover changing the `length` of `arguments` has no effect on its indexed properties:

```

1 var arr = [1,2,3];
2 var args = echoArgs(1,2,3);
3
4 Object.prototype.toString.apply(arr); //"[object Array]"
5 Object.prototype.toString.apply(args); //"[object Object]"
6
7 arr.push(4); //4
8 args.push(4); //TypeError: args.push is not a function
9
10 arr.length = 1;
11 arr[2]; //undefined
12 args.length = 1;
13 args[2]; //3

```

Leveraging methods of `Array.prototype`

Since all the methods of `Array.prototype` are designed to be generic they can be easily applied to the array-compatible `arguments` object:

```

1 var args = echoArgs(1,2,3);
2
3 [].push.apply(args,[4,5]);
4 args[4]; //5
5
6 var mapped = [].map.call(args, function (s) { return s/100});
7 mapped[2]; //0.03

```

A common approach is to go one better by using `Array.prototype.slice` to copy the entire `arguments` object into a real array:

```

1 var argsArray = [].slice.apply(echoArgs(1,2,3));
2
3 argsArray.push(4,5);
4 argsArray[4]; //5
5
6 var mapped = argsArray.map( function (s) { return s/100});
7 mapped[2]; //0.03

```

Practical Applications

1. Functions that take unlimited arguments

```

1 var average = function ( /*numbers*/ ) {
2     for ( var i=0, total = 0, len=arguments.length; i<len; i++ ) {
3         total += arguments[i];
4     }
5     return total / arguments.length;
6 }
7
8 average(50, 6, 5, -1); //15

```

2. Verifying all named arguments are supplied

JavaScript's liberal attitude to parameter passing is appealing but some functions will break if all named arguments are not supplied. We could write a function wrapper to enforce this when necessary:

```

1  var requireAllArgs=  function (fn) {
2      return function () {
3          if (arguments.length < fn.length) {
4              throw ([ "Expected" , fn.length, "arguments, got" ,
5  arguments.length].join( " " ));
6          }
7          return fn.apply( this , arguments);
8      }
9  }
10
11 var divide = requireAllArgs( function (a, b) { return a/b});
12
13 divide(2/5); //Expected 2 arguments, got 1"
14 divide(2,5); //0.4

```

3. A string formatter

(based on Dean Edwards' [Base 2 library](#))

```

1  function format(string) {
2      var args = arguments;
3      var pattern = RegExp( "%([1-" + (arguments.length-1) + "])" , "g" );
4      return string.replace(pattern, function (match, index) {
5          return args[index];
6      });
7  };
8
9  format( "a %1 and a %2" , "cat" , "dog" );
10 //a cat and a dog

```

4. Partial function application

The typical JavaScript implementations of [curry](#), [partial](#) and [compose](#) store the `arguments` object for later concatenation with the runtime arguments of the inner function.

```

Function.prototype.curry = function () {
1      if (arguments.length<1) {
2          return this ; //nothing to curry with - return function
3      }
4      var __method =  this ;
5      var args = [].slice.apply(arguments);
6      return function () {
7          return __method.apply( this ,
8          args.concat([].slice.apply(arguments)));
9      }
10 }
11
12 var converter =  function (ratio, symbol, input) {
13     return [(input*ratio).toFixed(1),symbol].join( " " );
14 }

```

```

15 }
16
17 var kilosToPounds = converter.curry(2.2, "lbs" );
18 var milesToKilometers = converter.curry(1.62, "km" );
19
20 kilosToPounds(4); //8.8 lbs
  milesToKilometers(34); //55.1 km

```

The Future...

Brendan Eich has stated that the `arguments` object will gradually disappear from JavaScript. In this fascinating ["minute with Brendan" excerpt](#) he ponders the future of arguments handling. Here's my take away:

rest parameters

Harmony (the next scheduled specification of ECMAScript) has already penciled in the design for a likely successor known as a [rest parameter](#) and it's scheduled to be prototyped in Firefox later this year (ActionScript already supports a similar feature).

The idea behind the `rest` parameter is disarmingly simple. If you prefix the last (or only) formal parameter name with `...', that parameter gets created as an array (a genuine array) which acts as a bucket for all passed arguments that do not match any of the other named parameters.

Here's a simple example...

```

1 //Proposed syntax....
2
3 var callMe(fn, ...args) {
4     return fn.apply(args);
5 }
6
7 callMe(Math.max, 4, 7, 6); //7

```

...and here's our curry function rewritten using `rest` arguments. This time there is no need to copy the outer `arguments` object, instead we make it a `rest` parameter with a unique name so that the inner function can simply reference it by closure. Also no need to apply array methods to either of our `rest` arguments.

```

1 //Proposed syntax....
2
3 Function.prototype.curry = function (...curryArgs) {
4     if (curryArgs.length < 1) {
5         return this; //nothing to curry with - return function
6     }
7     var __method = this;
8     return function (...args) {
9         return __method.apply(this, curryArgs.concat(args));
10    }
11 }

```

spread

Similar to Ruby's `splat` operator, `spread` will unpack an array into a formal argument list. Amongst other things

this allows the members of a `rest` parameter to be passed as a set of formal arguments to another function:

```

1 //Possible future syntax....
2
3 var stats = function (...numbers) {
4     for ( var i=0, total = 0, len=numbers.length; i<len; i++) {
5         total += numbers[i];
6     }
7     return {
8         average: total / arguments.length,
9         max: Math.max(...numbers); //spread array into formal params
10    }
11 }
12
13 stats(5, 6, 8, 5); //{average: 6, max: 8}

```

Notice that I'm assuming that there will be no need for a formal `spread` operator and that `spread` just describes the process of automatic coercion from an array into listed parameters.

For the above example we could have fallen back on the traditional `Math.max.apply(numbers)` instead, but unlike `apply` `spread` will also work with constructors and with multiple array arguments.

A Brave New (JavaScript) World awaits...enjoy!

23. Understanding JavaScript Closures

In JavaScript, a closure is a function to which the variables of the surrounding context are bound by reference.

```

1 function getMeAClosure() {
2     var canYouSeeMe = "here I am" ;
3     return ( function theClosure() {
4         return {canYouSeeIt: canYouSeeMe ? "yes!" : "no" };
5     });
6 }
7
8 var closure = getMeAClosure();
9 closure().canYouSeeIt; //"yes!"

```

Every JavaScript function forms a closure on creation. In a moment I'll explain why and walk through the process by which closures are created. Then I'll address some common misconceptions and finish with some practical applications. But first a brief word from our sponsors: JavaScript closures are brought to you by *lexical scope* and the *VariableEnvironment*...

Lexical Scope

The word *lexical* pertains to words or language. Thus the *lexical scope* of a function is statically defined by the function's physical placement within the written source code.

Consider the following example:

```

1 var x = "global" ;

```

```

2
3   function outer() {
4     var y = "outer" ;
5
6     function inner() {
7       var x = "inner" ;
8     }
9 }
```

Function `inner` is physically surrounded by function `outer` which in turn is wrapped by the global context. We have formed a lexical hierarchy:

```

global
outer
  inner
```

The outer lexical scope of any given function is defined by its ancestors in the lexical hierarchy. Accordingly, the outer lexical scope of function `inner` comprises the global object and function `outer`.

VariableEnvironment

The global object has an associated execution context. Additionally every invocation of a function establishes and enters a new execution context. The execution context is the dynamic counterpart to the static lexical scope. Each execution context defines a VariableEnvironment which is a repository for variables declared by that context. ([ES 5 10.4, 10.5](#))

[Note in EcmaScript 3, the VariableEnvironment of a function was known as the ActivationObject -- which is also the term I used in some older articles]

We could represent the VariableEnvironment with pseudo-code...

```

1 //variableEnvironment: {x: undefined, etc.};
2 var x = "global" ;
3 //variableEnvironment: {x: "global", etc.};
4
5 function outer() {
6   //variableEnvironment: {y: undefined};
7   var y = "outer" ;
8   //variableEnvironment: {y: "outer"};
9
10  function inner() {
11    //variableEnvironment: {x: undefined};
12    var x = "inner" ;
13    //variableEnvironment: {x: "inner"};
14  }
15}
```

However, it turns out this is only part of the picture. Each VariableEnvironment will also inherit the VariableEnvironment of its lexical scope. [The hero enters (stage-left)....]

The `[[scope]]` property

When a given *execution context* encounters a function definition in the code, a new function object is created with an internal property named `[[scope]]` (as in *lexical scope*) which references the current VariableEnvironment. ([ES 5 13.0](#)-)

2)

Every function gets a `[[scope]]` property, and when the function is invoked the value of the scope property is assigned to the *outer lexical environment reference* (or `outerLex`) property of its `VariableEnvironment`. ([ES 5 10.4.3.5-7](#)) In this way, each `VariableEnvironment` inherits from the `VariableEnvironment` of its lexical parent. This scope chaining runs the length of the lexical hierarchy starting from the global object.

Let's see how our pseudo-code looks now:

```

1  //VariableEnvironment: {x: undefined, etc.};
2  var x = "global" ;
3  //VariableEnvironment: {x: "global", etc.};
4
5  function outer() {
6      //VariableEnvironment: {y: undefined, outerLex: {x: "global", etc.}};
7      var y = "outer" ;
8      //VariableEnvironment: {y: "outer", outerLex: {x: "global", etc.}};
9
10     function inner() {
11         //VariableEnvironment: {x: undefined, outerLex: {y: "outer",
12         outerLex: {x:"global", etc.}});
13         var x = "inner" ;
14         //VariableEnvironment: {x: "inner", outerLex: {y: "outer", outerLex:
15         {x:"global", etc.}}};
16     }
17 }
```

The `[[scope]]` property acts as a bridge between nested `VariableEnvironments` and enables the process by which outer variables are embedded by inner `VariableEnvironments` (and prioritized by lexical proximity). The `[[scope]]` property also enables closures, since without it the variables of an outer function would be dereferenced and garbage collected once the outer function returned.

So there we have it -- closures are nothing but an unavoidable side-effect of lexical scoping

Dispelling the Myths

Now that we know how closures work, we can begin to address some of the more scurrilous rumors associated with them.

Myth 1. Closures are created only after an inner function has been returned

When the function is created, it is assigned a `[[scope]]` property which references the variables of the outer lexical scope and prevents them from being garbage collected. Therefore the closure is formed on function creation

There is no requirement that a function should be returned before it becomes a closure. Here's a closure that works without returning a function:

```

1  var callLater =  function (fn, args, context) {
2      setTimeout( function (){fn.apply(context, args)}, 2000);
3  }
4
5  callLater(alert,[ 'hello' ]);
```

Myth 2. The values of outer variables get copied or "baked in" to the closure

As we've seen, the closure references variables not values.

```

1 //Bad Example
2 //Create an array of functions that add 1,2 and 3 respectively
3 var createAdders = function () {
4     var fns = [];
5     for ( var i=1; i<4; i++) {
6         fns[i] = ( function (n) {
7             return i+n;
8         });
9     }
10    return fns;
11 }
12
13 var adders = createAdders();
14 adders[1](7); //11 ??
15 adders[2](7); //11 ??
16 adders[3](7); //11 ??

```

All three adder functions point to the same variable `i`. By the time any of these functions is invoked, the value of `i` is 4.

One solution is to pass each argument via a self invoking function. Since every function invocation takes place in a unique execution context, we guarantee the uniqueness of the argument variable across successive invocations.

```

1 //Good Example
2 //Create an array of functions that add 1,2 and 3 respectively
3 var createAdders = function () {
4     var fns = [];
5     for ( var i=1; i<4; i++) {
6         ( function (i) {
7             fns[i] = ( function (n) {
8                 return i+n;
9             });
10        })(i)
11    }
12    return fns;
13 }
14
15 var adders = createAdders();
16 adders[1](7); //8 (:
17 adders[2](7); //9 (:
18 adders[3](7); //10 (:

```

Myth 3. Closures only apply to inner functions

Admittedly closures created by outer functions are not interesting because the `[[scope]]` property only references the global scope, which is universally visible in any case. Nevertheless its important to note that the closure creation process is identical for every function, and every function creates a closure.

Myth 4. Closures only apply to anonymous functions

I've seen this claim in one too many articles. Enough said

Myth 5. Closures cause memory leaks

Closures do not of themselves create circular references. In our original example, function `inner` references outer variables via its `[[scope]]` property, but neither the referenced variables or function `outer` references function `inner` or its local variables.

Older versions of IE are notorious for memory leaks and these usually get blamed on closures. A typical culprit is a DOM element referenced by a function, while an attribute of that same DOM element references another object in the same lexical scope as the function. Between IE6 and IE8 these circular references have been mostly tamed.

Practical Applications

Function templates

Sometimes we want to define multiple versions of a function, each one conforming to a blueprint but modified by supplied arguments. For example, we can create a standard set of functions for converting units of measures:

```

1  function makeConverter(toUnit, factor, offset) {
2      offset = offset || 0;
3      return function (input) {
4          return [((offset+input)*factor).toFixed(2), toUnit].join( " " );
5      }
6  }
7
8  var milesToKm = makeConverter( 'km' ,1.60936);
9  var poundsToKg = makeConverter( 'kg' ,0.45460);
10 var farenheitToCelsius = makeConverter( 'degrees C' ,0.5556, -32);
11
12 milesToKm(10); // "16.09 km"
13 poundsToKg(2.5); // "1.14 kg"
14 farenheitToCelsius(98); // "36.67 degrees C"
```

If, like me, you're into functional abstraction the next logical step would be to [curry](#) this process (see below).

Functional JavaScript

Aside from the fact that JavaScript functions are first class objects, functional JavaScript's other best friend is closures.

The typical implementations of bind, [curry](#), [partial](#) and [compose](#) all rely on closures to provide the new function with a reference to the original function and arguments.

For example, here's curry:

```

Function.prototype.curry = function () {
1      if (arguments.length<1) {
2          return this ; //nothing to curry with - return function
3      }
4      var __method = this ;
5      var args = toArray(arguments);
6
```

```

7      return function () {
8          return __method.apply( this , args.concat([]).slice.apply( null ,
9 arguments));
10     }
11 }
```

And here's our previous example re-done using curry

```

1  function converter(toUnit, factor, offset, input) {
2      offset = offset || 0;
3      return [((offset+input)*factor).toFixed(2), toUnit].join( " " );
4  }
5
6  var milesToKm = converter.curry( 'km' ,1.60936,undefined);
7  var poundsToKg = converter.curry( 'kg' ,0.45460,undefined);
8  var farenheitToCelsius = converter.curry( 'degrees C' ,0.5556, -32);
9
10 milesToKm(10); // "16.09 km"
11 poundsToKg(2.5); // "1.14 kg"
12 farenheitToCelsius(98); // "36.67 degrees C"
```

There are plenty of other nifty function modifiers that use closures. This little gem comes courtesy of [Oliver Steele](#)

```

/** 
1   * Returns a function that takes an object, and returns the value of its
2 'name' property
3 */
4 var pluck =  function (name) {
5     return function (object) {
6         return object[name];
7     }
8 }
9
10 var getLength = pluck( 'length' );
11 getLength( "SF Giants are going to the World Series!" ); //40
```

The module pattern

This [well known technique](#) uses a closure to maintain a private, exclusive reference to a variable of the outer scope. Here I'm using the module pattern to make a "guess the number" game. Note that in this example, the closure (`guess`) has exclusive access to the `secretNumber` variable, while the `responses` object references a copy of the variable's value at the time of creation.

```

1  var secretNumberGame =  function () {
2      var secretNumber = 21;
3
4      return {
5          responses: {
6              true : "You are correct! Answer is " + secretNumber,
7              lower: "Too high!",
8              higher: "Too low!"
9          },
10
11          guess:  function (guess) {
12              var key =
```

```

13         (guess == secretNumber) ||
14             (guess < secretNumber ? "higher" : "lower" );
15             alert( this.responses[key] )
16     }
17 }
18 }
19
20 var game = secretNumberGame();
21 game.guess(45); //Too high!
22 game.guess(18); //Too low!
23 game.guess(21); //You are correct! Answer is 21"

```

Wrap up

In programming terms, closures represent the height of grace and sophistication. They make code more compact, readable and beautiful and promote functional re-use. Knowing how and why closures work eliminates the uncertainty around their usage. I hope this article helps in that regard. Please feel free to comment with questions, thoughts or concerns.

Further Reading

[ECMA-262 5th Edition](#)

- 10.4 Creating the VariableEnvironment
- 10.4.3.5-7 Referencing the [[scope]] property in the VariableEnvironment
- 10.5 Populating the VariableEnvironment
- 13.0-2 Assigning the [[scope]] property when a function is created

24. The module pattern (in a nutshell)

The module pattern (first publicized by the Yahoo! JavaScript team) makes use of [closures](#) to bake privacy and state into your objects.

This is the generic form...

```

1 function () {
2     //private state
3     //private functions
4     return {
5         //public state
6         //public variables
7     }
8 }
```

Now lets put some meat on the bones. Here is a poll manager responsible for tallying yes and no votes:-

```

1 var pollManager = function () {
2     //private state
3     var alreadyVoted = {};
4     var yesVotes = 0;
5     var noVotes = 0;
6
7     //return public interface
8 }
```

```

7      return {
8          vote : function (name, voteYes) {
9              if (alreadyVoted[name]) {
10                  alert(name + ", you can't vote twice" );
11              } else {
12                  alreadyVoted[name] = true ;
13                  voteYes ? yesVotes++ : noVotes++;
14              }
15          },
16      },
17      reportTally : function () {
18          var results = [];
19          results.push( "Yes = " );results.push(yesVotes);
20          results.push( ", No = " );results.push(noVotes);
21          return results.join( "" );
22      }
23  }
24 }
25 }
26
27 var doYouLikeBroccoli = pollManager();
28 doYouLikeBroccoli.vote( "Bob" , true );
29 doYouLikeBroccoli.vote( "Mary" , false );
30 doYouLikeBroccoli.vote( "Bob" , true ); //Bob, you can't vote twice
31 doYouLikeBroccoli.reportTally(); //"Yes = 1, No = 1"

```

We could have written this as an object literal ({}), but by enclosing it in a function instead we created a closure. This has the effect of protecting state (and potentially functionality) from the outside world. We return only the public API, everything else is private -- the names of voters cannot be listed, the vote tallies can only be updated by voting.

We can further crank up the privacy by rewriting the reportTally function to show only the percentages. We'll also create a helper function called asPercentage. Since asPercentage is only useful within this module we won't return it as part of the public API -- which means it becomes a private function -- the module now hides function access as well as state.

```

1  var pollManager = function () {
2      //private state
3      //...
4      var asPercentage = function (value) {
5          return Math.round((100*(value/(yesVotes + noVotes))));}
6      }
7
8      //return public interface
9      return {
10          //...
11          reportTally : function () {
12              return "Yes = " + asPercentage(yesVotes) + "%" +
13                  ", No = " + asPercentage(noVotes) + "%";
14          }
15      }
16  }
17
18 //...
19 doYouLikeBroccoli.reportTally(); //"Yes = 50%, No = 50%"

```

At the risk of stating the obvious, in JavaScript when you make a function private, the logic is not hidden. You won't

keep your encryption function a secret by hiding it in a module. The concept of privacy is restricted to runtime access. I can only invoke pollManager's asPercentage function or retrieve the value of the noVotes variable from within the pollManager closure.

An equally important benefit of modules is tidiness. Private objects only exist for the lifetime of the module function call -- after which they are available for garbage collection. Similarly, the module returns an API object (e.g. doYouLikeBroccoli) whose properties are functions (vote and reportTally). These function objects live and die with the API object.

Sometimes you might want to access part the publicly returned object from within your private methods. (For sometimes read *very occasionally* -- I couldn't really think of a convincing example that wouldn't work better with this behavior factored out) . In that case we can assign the public object to a variable (addressable from anywhere in the function) before returning it.

```

1  function () {
2      //private state
3      //private functions (can refer to publicObj)
4      var publicObj = { /*...public API...*/ };
5      return publicObj;
6  }

```

More often than not you will see modules wrapped in parentheses and invoked immediately to provide singletons. Churning out a large number of module instances would be clunky and inefficient but the implication that you would never ever need more than one polling manager or id generator in your app is a bit of a mystery to me.

I'm drawn to the module pattern for its elegance (and genius). You get to keep your code tidy by hiding the one-off grunt work in a box while highlighting the interface you intend others to interact with. Clearly stated intention is the bedrock of good programming.

So why don't I use it very often? There are some obvious answers: most of the tidiness of this pattern can be replicated with a simple object literal pattern, prototyping is more efficient for multiple instance creation and most grunt work is not one-off...you want to put in a utility file and re-use it.

But there are also more subtle reasons: privacy and form are probably overrated in Javascript where there is no formal concept of interface, class or strong typing. Objects can be effortlessly reshaped and transformed and this is liberating, especially for those brought up on the rigid constructs of Java et al. Hiding and partitioning introduces unfamiliar constraints to JavaScript developers who generally rely on self discipline over enforced discipline.

Wes, thanks for sharing your code. What you are doing is essentially the same as the module pattern. You are attaching public attributes to a variable and returning it. However there are a few things that are a bit confusing to me:-

Firstly you assign *this* to *self* and *self* is ultimately the public object that will be returned. You didn't include any usage examples but I'm assuming you are not using the *new Constructor* pattern, in which case *this* will be the invoker of *timepick*. Having the invoker of the function overwritten by the return value seems risky. It could be that I'm missing something in your use case

Secondly the module pattern is usually free from process, i.e. it just returns the public API as an object. However your example ends by calling init() which is process rich -- and finally returns the public API almost as an afterthought.

Thirdly public and private attributes are not clearly sectioned off from one another. You start with the public object *self* and a public attribute, then define a bunch of private variables, followed by public functions, a long and distracting

switch statement, an init function that is private but ends up returning the public API, followed by more private functions.

Apologies if this seems critical, I don't mean to be -- you clearly have a good understanding of the language -- I think your example could just use some better organization

Hope this helps

25. Understanding JavaScript's `undefined`

Compared to other languages, JavaScript's concept of undefined is a little confusing. In particular, trying to understand ReferenceErrors ("x is not defined") and how best to code against them can be frustrating.

This is my attempt to straighten things out a little. If you're not already familiar with the difference between variables and properties in JavaScript (including the internal VariableObject) now might be a good time to check out my [previous posting](#).

What is undefined?

In JavaScript there is Undefined (type), undefined (value) and undefined (variable).

Undefined (type) is a built-in JavaScript type.

undefined (value) is a primitive and is the sole value of the Undefined type. Any property that has not been assigned a value, assumes the undefined value. (ECMA 4.3.9 and 4.3.10). A function without a return statement, or a function with an empty return statement returns undefined. The value of an unsupplied function argument is undefined.

```

1  var a;
2  typeof a; // "undefined"
3
4  window.b;
5  typeof window.b; // "undefined"
6
7  var c = (function () {})();
8  typeof c; // "undefined"
9
10 var d = (function (e) { return e})();
11 typeof d; // "undefined"
```

undefined (variable) is a global property whose initial value is undefined (value). Since its a global property we can also access it as a variable. For consistency I'm always going to call it a variable in this article.

```

1  typeof undefined; // "undefined"
2
3  var f = 2;
4  f = undefined; // re-assigning to undefined (variable)
5  typeof f; // "undefined"
```

As of ECMA 3, its value can be reassigned :

```

1  undefined = "washing machine" ; //assign a string to undefined (variable)
2  typeof undefined //string
3
4  f = undefined;
5  typeof f; //string
6  f; //washing machine

```

Needless to say, re-assigning values to the undefined variable is very bad practice, and in fact its not allowed by ECMA 5 (though amongst the current set of full browser releases, only Safari enforces this).

And then there's null?

Yes, generally well understood but worth re-stating: `undefined` is distinct from `null` which is also a primitive value representing the *intentional* absence of a value. The only similarity between `undefined` and `null` is they both coerce to false.

So what's a ReferenceError?

A ReferenceError indicates that an invalid reference value has been detected (ECMA 5 15.11.6.3)

In practical terms, this means a ReferenceError will be thrown when JavaScript attempts to get the value of an unresolvable reference. (There are other cases where a ReferenceError will be thrown, most notably when running in ECMA 5 Strict mode. If you're interested check the reading list at the end of this article)

Note how the message syntax varies across browser. As we will see none of these messages is particularly enlightening:

```

1  alert(foo)
2  //FF/Chrome: foo is not defined
3  //IE: foo is undefined
4  //Safari: can't find variable foo

```

Still not clear..."unresolvable reference"?

In ECMA terms, a Reference consists of a base value and a reference name (ECMA 5 8.7 -- again I'm glossing over strict mode. Also note that ECMA 3 terminology varies slightly but the effect is the same)

If the Reference is a property, the base value and the reference name sit either side of the dot (or first bracket or whatever):

```

1  window.foo; //base value = window, reference name = foo;
2  a.b; //base value = a, reference name = b;
3  myObj[ 'create' ]; // base value = myObj, reference name = 'create';
4  //Safari, Chrome, IE8+ only
5  Object.defineProperty(window, "foo" , {value: "hello" }); //base value =
window, reference name = foo;

```

For variable References, the base value is the VariableObject of the current execution context. The VariableObject of the global context is the global object itself (`window` in a browser). Each functional context has an abstract VariableObject known as the ActivationObject.

```

1  var foo; //base value = window, reference name = foo

```

```
2 function a() {
3     var b; base value = <code>ActivationObject</code>, reference name = b
4 }
```

A Reference is considered unresolvable if its base value is undefined

Therefore a property reference is unresolvable if the value before the dot is undefined. The following example would throw a ReferenceError but it doesn't because TypeError gets there first. This is because the base value of a property is subject to CheckObjectCoercible (ECMA 5 9.10 via 11.2.1) which throws a TypeError when trying to convert Undefined type to an Object. (thanks to kangax for the pre-posting tip off via twitter)

```
1 var foo;
2 foo.bar; //TypeError (base value, foo, is undefined)
3 bar.baz; //ReferenceError (bar is unresolvable)
4 undefined.foo; //TypeError (base value is undefined)
```

A variable Reference will never be unresolvable since the var keyword ensures a VariableObject is always assigned to the base value.

References which are neither properties or variables are by definition unresolvable and will throw a ReferenceError:

```
1 foo; //ReferenceError
```

JavaScript sees no explicit base value and therefore looks up the VariableObject for a property with reference name 'foo'. Finding none it determines 'foo' has no base value and throws a ReferenceError

But isn't foo just an undeclared variable?

Technically no. Though we sometimes find "undeclared variable" a useful term for bug diagnostics, in reality a variable is not a variable until its declared.

What about implicit globals?

It's true, identifiers which were never declared with the var keyword will get created as global variables -- but only if they are the object of an assignment

```
1 function a() {
2     alert(foo); //ReferenceError
3     bar = [1,2,3]; //no error, foo is global
4 }
5 a();
6 bar; //"1,2,3"
```

This is, of course, annoying. It would be better if JavaScript consistently threw ReferenceErrors when it encountered unresolvable references (and in fact this is what it does in ECMA Strict Mode)

When do I need to code against ReferenceErrors?

If your code is sound, very rarely. We've seen that in typical usage there is only one way to get an unresolvable reference: use a syntactically correct Reference that is neither a property or a variable. In most cases this scenario is avoided by ensuring you remember the var keyword. The only time you might get a run-time surprise is when

referencing variables that only exist in certain browsers or 3rd party code.

A good example is the **console**. In Webkit browsers the console is built-in and the console property is always available. The Firefox console depends on Firebug (or other add-ons) being installed and switched on. IE7 has no console, IE8 has a console but the console property only exists when IE Developer Tools is started. Apparently Opera has a console but I've never got it to work

The upshot is that there's a good chance the following snippet will throw a ReferenceError when run in the browser:

```
1 console.log( new Date());
```

How do I code against variables that may not exist?

One way to inspect an unresolvable reference without throwing a ReferenceError is by using the `typeof` keyword

```
1 if ( typeof console != "undefined" ) {
2     console.log( new Date());
3 }
```

However this always seems verbose to me, not to mention dubious (its not the reference name that is undefined, its the base value), and anyway I prefer to reserve `typeof` for positive type checking.

Fortunately there's an alternative: we already know that undefined properties will not throw a ReferenceError providing their base value is defined- and since `console` belongs to the global object, we can just do this:

```
1 window.console && console.log( new Date());
```

In fact you should only ever need to check for variable existence within the global context (the other execution contexts exist within functions, and you control what variables exist in your own functions). So in theory at least you should be able to get away without ever using a `typeof` check against a ReferenceError

Where can I read more?

Mozilla Developer Center: [undefined](#)

Angus Croll: [Variables vs. properties in JavaScript](#)

Juriy Zaytsev ("kangax"): [Understanding Delete](#)

Dmitry A. Soshnikov: [ECMA-262-3 in detail. Chapter 2. Variable object.](#)

[ECMA-262 5th Edition](#)

undefined: 4.3.9, 4.3.10, 8.1

Reference Error: 8.7.1, 8.7.2, 10.2.1, 10.2.1.1.4, 10.2.1.2.4, and 11.13.1.

The Strict Mode of ECMAScript Annex C

26. JavaScript Scoping and Hoisting

Do you know what value will be alerted if the following is executed as a JavaScript program?

CODE

```

var foo = 1;
function bar() {
    if (!foo) {
        var foo = 10;
    }
    alert(foo);
}
bar();

```

If it surprises you that the answer is "10", then this one will probably really throw you for a loop:

CODE

```

var a = 1;
function b() {
    a = 10;
    return;
    function a() {}
}
b();
alert(a);

```

Here, of course, the browser will alert "1". So what's going on here? While it might seem strange, dangerous, and confusing, this is actually a powerful and expressive feature of the language. I don't know if there is a standard name for this specific behavior, but I've come to like the term "hoisting". This article will try to shed some light on this mechanism, but first lets take a necessary detour to understand JavaScript's scoping.

Scoping in JavaScript

One of the sources of most confusion for JavaScript beginners is scoping. Actually, it's not just beginners. I've met a lot of experienced JavaScript programmers who don't fully understand scoping. The reason scoping is so confusing in JavaScript is because it looks like a C-family language. Consider the following C program:

CODE

```

#include <stdio.h>
int main() {
    int x = 1;
    printf("%d, ", x); // 1
    if (1) {
        int x = 2;
        printf("%d, ", x); // 2
    }
    printf("%d\n", x); // 1
}

```

The output from this program will be 1, 2, 1. This is because C, and the rest of the C family, has **block-level scope**. When control enters a block, such as the `if` statement, new variables can be declared within that scope, without affecting the outer scope. This is not the case in JavaScript. Try the following in Firebug:

CODE

```

var x = 1;
console.log(x); // 1
if (true) {
    var x = 2;
    console.log(x); // 2
}
console.log(x); // 2

```

In this case, Firebug will show 1, 2, 2. This is because JavaScript has **function-level scope**. This is radically different from the C family. Blocks, such as `if` statements, **do not** create a new scope. Only functions create a new scope.

To a lot of programmers who are used to languages like C, C++, C#, or Java, this is unexpected and unwelcome. Luckily, because of the flexibility of JavaScript functions, there is a workaround. If you must create temporary scopes within a function, do the following:

CODE

```

function foo() {
    var x = 1;
    if (x) {
        (function () {
            var x = 2;
            // some other code
        })();
    }
    // x is still 1.
}

```

This method is actually quite flexible, and can be used anywhere you need a temporary scope, not just within block statements. However, I strongly recommend that you take the time to really understand and appreciate JavaScript scoping. It's quite powerful, and one of my favorite features of the language. If you understand scoping, hoisting will make a lot more sense to you.

Declarations, Names, and Hoisting

In JavaScript, a name enters a scope in one of four basic ways:

1. **Language-defined**: All scopes are, by default, given the names `this` and `arguments`.
2. **Formal parameters**: Functions can have named formal parameters, which are scoped to the body of that function.
3. **Function declarations**: These are of the form `function foo() {}`.
4. **Variable declarations**: These take the form `var foo;`.

Function declarations and variable declarations are always moved ("hoisted") invisibly to the top of their containing scope by the JavaScript interpreter. Function parameters and language-defined names are, obviously, already there. This means that code like this:

CODE

```
function foo() {
    bar();
    var x = 1;
}
```

is actually interpreted like this:

CODE

```
function foo() {
    var x;
    bar();
    x = 1;
}
```

It turns out that it doesn't matter whether the line that contains the declaration would ever be executed. The following two functions are equivalent:

CODE

```
function foo() {
    if (false) {
        var x = 1;
    }
    return;
    var y = 1;
}
function foo() {
    var x, y;
    if (false) {
        x = 1;
    }
    return;
    y = 1;
}
```

Notice that the assignment portion of the declarations were not hoisted. Only the name is hoisted. This is not the case with function declarations, where the entire function body will be hoisted as well. But remember that there are two normal ways to declare functions. Consider the following JavaScript:

CODE

```
function test() {
    foo(); // TypeError "foo is not a function"
    bar(); // "this will run!"
    var foo = function () { // function expression assigned to local variable 'foo'
        alert("this won't run!");
    }
    function bar() { // function declaration, given the name 'bar'
        alert("this will run!");
    }
}
test();
```

In this case, only the function declaration has its body hoisted to the top. The name 'foo' is hoisted, but the body is left behind, to be assigned during execution.

That covers the basics of hoisting, which is not as complex or confusing as it seems. Of course, this being JavaScript, there is a little more complexity in certain special cases.

Name Resolution Order

The most important special case to keep in mind is name resolution order. Remember that there are four ways for names to enter a given scope. The order I listed them above is the order they are resolved in. In general, if a name has already been defined, it is never overridden by another property of the same name. This means that a function declaration takes priority over a variable declaration. This does not mean that an assignment to that name will not work, just that the declaration portion will be ignored. There are a few exceptions:

- The built-in name `arguments` behaves oddly. It seems to be declared following the formal parameters, but before function declarations. This means that a formal parameter with the name `arguments` will take precedence over the built-in, even if it is undefined. This is a bad feature. Don't use the name `arguments` as a formal parameter.
- Trying to use the name `this` as an identifier anywhere will cause a `SyntaxError`. This is a good feature.
- If multiple formal parameters have the same name, the one occurring latest in the list will take precedence, even if it is undefined.

Named Function Expressions

You can give names to functions defined in function expressions, with syntax like a function declaration. This does not make it a function declaration, and the name is not brought into scope, nor is the body hoisted. Here's some code to illustrate what I mean:

CODE

```
foo(); // TypeError "foo is not a function"
bar(); // valid
baz(); // TypeError "baz is not a function"
spam(); // ReferenceError "spam is not defined"

var foo = function () {} // anonymous function expression ('foo' gets hoisted)
function bar() {} // function declaration ('bar' and the function body get hoisted)
var baz = function spam() {} // named function expression (only 'baz' gets hoisted)

foo(); // valid
bar(); // valid
baz(); // valid
spam(); // ReferenceError "spam is not defined"
```

How to Code With This Knowledge

Now that you understand scoping and hoisting, what does that mean for coding in JavaScript? The most important thing is to always declare your variables with a `var` statement. I **strongly** recommend that you have *exactly one* `var` statement per scope, and that it be at the top. If you force yourself to do this, you will never have hoisting-related confusion. However, doing this can make it hard to keep track of which variables have actually been declared in the current scope. I recommend using [JSLint](#) with the `onevar` option to enforce this. If you've done all of this, your code should look something like this:

CODE

```
/*jslint onevar: true [...] */
function foo(a, b, c) {
    var x = 1,
        bar,
        baz = "something";
}
```

What the Standard Says

I find that it's often useful to just consult the [ECMAScript Standard \(pdf\)](#) directly to understand how these things work. Here's what it has to say about variable declarations and scope (section 12.2.2 in the older version):

If the variable statement occurs inside a FunctionDeclaration, the variables are defined with function-local scope in that function, as described in section 10.1.3. Otherwise, they are defined with global scope (that is, they are created as members of the global object, as described in section 10.1.3) using property attributes { DontDelete }. Variables are created when the execution scope is entered. A Block does not define a new execution scope. Only Program and FunctionDeclaration produce a new scope. Variables are initialised to undefined when created. A variable with an Initialiser is assigned the value of its AssignmentExpression when the VariableStatement is executed, not when the variable is created.

I hope this article has shed some light on one of the most common sources of confusion to JavaScript programmers. I have tried to be as thorough as possible, to avoid creating more confusion. If I have made any mistakes or have large omissions, please let me know.

27. Spying Constructors in JavaScript

When writing unit-tests for code, a common technique is **spying**, where you set expectations on a method's invocation, run some code, and verify that the method was invoked as expected. This is pretty straightforward. Here's a simple example using [JsMockito](#):

CODE

```
function foo(a) { return a; }
foo = spy(foo);
foo(1);
verify(foo)(1); // verified!
verify(foo)(2); // never run
```

Here, we're spying on the `foo` method, and checking that it was invoked at least once with the parameter `1`, and once with the parameter `2`. As it turns out, this `spy` method does not work well with JavaScript constructors, in [JsMockito](#), [Jasmine](#), or many other testing frameworks. The basic problem is that the prototype is not transferred appropriately, so code like this will fail:

CODE

```
function Foo(a) {
    this.a = a;
}
Foo.prototype = {
    bar: function () {
        console.log(this.a);
    }
};

var f = new Foo(1);
f.bar(); // 1

Foo = spy(Foo);
var g = new Foo(1);
g.bar(); // error

verify(Foo)(1); // not reached
```

It turns out it's really easy to write a constructor-safe spying function, and it doesn't even take very many lines of code.

CODE

```
function spy(F) {
    function G() {
        var args = Array.prototype.slice.call(arguments);
        G.calls.push(args);
        F.apply(this, args);
    }

    G.prototype = F.prototype;
    G.calls = [];

    return G;
}
```

This `spy` function works just like the one in JsMockito, but it doesn't fail with constructors. For completeness, here's an implementation of a simple `verify` function:

CODE

```
function verify(F) {
    return function () {
        var args = Array.prototype.slice.call(arguments),
            i,
            j,
            call,
            count = 0,
            matched;

        for (i = 0; i < F.calls.length; i += 1) {
            call = F.calls[i];
            matched = true;
            for (j = 0; j < args.length; j += 1) {
                if (args[j] !== call[j]) {
                    matched = false;
                    break;
                }
            }
            if (matched) {
                count += 1;
            }
        }

        return count > 0;
    };
}
```

It would be easy to extend this `verify` implementation to allow more types of verify like `.once()` or `.never()`, working off the `count` variable.

And that's it! Here's an example of code that will work with this `spy` implementation:

CODE

```

function Foo(name, id) {
    this.name = name;
    this.id = id;
}

Foo.prototype = {
    log: function () {
        console.log("Foo %o:%o", this.id, this.name);
    }
};

var f = new Foo("test", 1);
f.log();

Foo = spy(Foo);

var f2 = new Foo("spied", 2);
f2.log();

console.log("verify Foo(\"spied\", 2): %o", verify(Foo)("spied", 2));
console.log("verify Foo(\"something\", 2): %o", verify(Foo)("something", 2));

var baz = {
    spam: function (a) {
        console.log("calling baz.spam(%o), this.other=%o", a, this.other);
    },
    other: 10
};

baz.spam = spy(baz.spam);

baz.spam(1);
console.log("verify baz.spam(1)", verify(baz.spam)(1));
console.log("verify baz.spam(2)", verify(baz.spam)(2));

```

The other neat thing is that, so long as you're not trapping stale references to the original constructor function before it got spied, JavaScript's `instanceof` operator should work just fine:

CODE

```

function F() {}
F = spy(F)
new F() instanceof F; // true

```

You can find the complete code (and a bit more) for this exercise at www.bcherry.net/playground/spying-constructors. I hope this was informative. I think I'll probably end up either contributing a patch to JsMockito with this, or building my own bare-bones set of mocking/spying functions for use with [QUnit](#).

— P.S. It's been some time since I've updated, but I'm hoping this will be the first of many new, interesting JavaScript posts inspired by the work I'm doing at Twitter with @bs, @hoverbird, @ded, and @dsa. —

28. JavaScript Module Pattern: In-Depth

The module pattern is a common JavaScript coding pattern. It's generally well understood, but there are a number of advanced uses that have not gotten a lot of attention. In this article, I'll review the basics and cover some truly

remarkable advanced topics, including one which I think is original.

The Basics

We'll start out with a simple overview of the module pattern, which has been well-known since Eric Miraglia (of YUI) first [blogged about it](#) three years ago. If you're already familiar with the module pattern, feel free to skip ahead to "Advanced Patterns".

Anonymous Closures

This is the fundamental construct that makes it all possible, and really is the single **best feature of JavaScript**. We'll simply create an anonymous function, and execute it immediately. All of the code that runs inside the function lives in a **closure**, which provides **privacy** and **state** throughout the lifetime of our application.

```
(function () {
    // ... all vars and functions are in this scope only
    // still maintains access to all globals
})();
```

CODE

Notice the `()` around the anonymous function. This is required by the language, since statements that begin with the token `function` are always considered to be **function declarations**. Including `()` creates a **function expression** instead.

Global Import

JavaScript has a feature known as **implied globals**. Whenever a name is used, the interpreter walks the scope chain backwards looking for a `var` statement for that name. If none is found, that variable is assumed to be global. If it's used in an assignment, the global is created if it doesn't already exist. This means that using or creating global variables in an anonymous closure is easy. Unfortunately, this leads to hard-to-manage code, as it's not obvious (to humans) which variables are global in a given file.

Luckily, our anonymous function provides an easy alternative. By passing globals as parameters to our anonymous function, we **import** them into our code, which is both **clearer** and **faster** than implied globals. Here's an example:

```
(function ($, YAHOO) {
    // now have access to globals jQuery (as $) and YAHOO in this code
}(jQuery, YAHOO));
```

CODE

Module Export

Sometimes you don't just want to *use* globals, but you want to *declare* them. We can easily do this by exporting them, using the anonymous function's **return value**. Doing so will complete the basic module pattern, so here's a complete example:

CODE

```

var MODULE = (function () {
    var my = {},
        privateVariable = 1;

    function privateMethod() {
        // ...
    }

    my.moduleProperty = 1;
    my.moduleMethod = function () {
        // ...
    };

    return my;
}());

```

Notice that we've declared a global module named `MODULE`, with two public properties: a method named `MODULE.moduleMethod` and a variable named `MODULE.moduleProperty`. In addition, it maintains **private internal state** using the closure of the anonymous function. Also, we can easily import needed globals, using the pattern we learned above.

Advanced Patterns

While the above is enough for many uses, we can take this pattern farther and create some very powerful, extensible constructs. Let's work through them one-by-one, continuing with our module named `MODULE`.

Augmentation

One limitation of the module pattern so far is that the entire module must be in one file. Anyone who has worked in a large code-base understands the value of splitting among multiple files. Luckily, we have a nice solution to **augment modules**. First, we import the module, then we add properties, then we export it. Here's an example, augmenting our `MODULE` from above:

CODE

```

var MODULE = (function (my) {
    my.anotherMethod = function () {
        // added method...
    };

    return my;
})(MODULE));

```

We use the `var` keyword again for consistency, even though it's not necessary. After this code has run, our module will have gained a new public method named `MODULE.anotherMethod`. This augmentation file will also maintain its own private internal state and imports.

Loose Augmentation

While our example above requires our initial module creation to be first, and the augmentation to happen second, that isn't always necessary. One of the best things a JavaScript application can do for performance is to load scripts

asynchronously. We can create flexible multi-part modules that can load themselves in any order with **loose augmentation**. Each file should have the following structure:

```
var MODULE = (function (my) {  
    // add capabilities...  
  
    return my;  
}(MODULE || {}));
```

CODE

In this pattern, the `var` statement is always necessary. Note that the import will create the module if it does not already exist. This means you can use a tool like [LABjs](#) and load all of your module files in parallel, without needing to block.

Tight Augmentation

While loose augmentation is great, it does place some limitations on your module. Most importantly, you cannot override module properties safely. You also cannot use module properties from other files during initialization (but you can at run-time after initialization). **Tight augmentation** implies a set loading order, but allows **overrides**. Here is a simple example (augmenting our original `MODULE`):

```
var MODULE = (function (my) {  
    var old_moduleMethod = my.moduleMethod;  
  
    my.moduleMethod = function () {  
        // method override, has access to old through old_moduleMethod...  
    };  
  
    return my;  
}(MODULE));
```

CODE

Here we've overridden `MODULE.moduleMethod`, but maintain a reference to the original method, if needed.

Cloning and Inheritance

CODE

```

var MODULE_TWO = (function (old) {
    var my = {},
        key;

    for (key in old) {
        if (old.hasOwnProperty(key)) {
            my[key] = old[key];
        }
    }

    var super_moduleMethod = old.moduleMethod;
    my.moduleMethod = function () {
        // override method on the clone, access to super through super_moduleMethod
    };

    return my;
})(MODULE);

```

This pattern is perhaps the **least flexible** option. It does allow some neat compositions, but that comes at the expense of flexibility. As I've written it, properties which are objects or functions will *not* be duplicated, they will exist as one object with two references. Changing one will change the other. This could be fixed for objects with a recursive cloning process, but probably cannot be fixed for functions, except perhaps with `eval`. Nevertheless, I've included it for completeness.

Cross-File Private State

One severe limitation of splitting a module across multiple files is that each file maintains its own private state, and does not get access to the private state of the other files. This can be fixed. Here is an example of a loosely augmented module that will **maintain private state** across all augmentations:

CODE

```

var MODULE = (function (my) {
    var _private = my._private = my._private || {},
        _seal = my._seal = my._seal || function () {
            delete my._private;
            delete my._seal;
            delete my._unseal;
        },
        _unseal = my._unseal = my._unseal || function () {
            my._private = _private;
            my._seal = _seal;
            my._unseal = _unseal;
        };

    // permanent access to _private, _seal, and _unseal

    return my;
})(MODULE || {});

```

Any file can set properties on their local variable `_private`, and it will be immediately available to the others. Once this module has loaded completely, the application should call `MODULE._seal()`, which will prevent external access to the internal `_private`. If this module were to be augmented again, further in the application's lifetime, one of the internal methods, in any file, can call `_unseal()` before loading the new file, and call `_seal()` again after it has

been executed. This pattern occurred to me today while I was at work, I have not seen this elsewhere. I think this is a very useful pattern, and would have been worth writing about all on its own.

Sub-modules

Our final advanced pattern is actually the simplest. There are many good cases for creating sub-modules. It is just like creating regular modules:

```
MODULE.sub = (function () {
    var my = {};
    // ...

    return my;
}());
```

CODE

While this may have been obvious, I thought it worth including. Sub-modules have all the advanced capabilities of normal modules, including augmentation and private state.

Conclusions

Most of the advanced patterns can be combined with each other to create more useful patterns. If I had to advocate a route to take in designing a complex application, I'd combine **loose augmentation**, **private state**, and **sub-modules**.

I haven't touched on performance here at all, but I'd like to put in one quick note: The module pattern is **good for performance**. It minifies really well, which makes downloading the code faster. Using **loose augmentation** allows easy non-blocking parallel downloads, which also speeds up download speeds. Initialization time is probably a bit slower than other methods, but worth the trade-off. Run-time performance should suffer no penalties so long as globals are imported correctly, and will probably gain speed in sub-modules by shortening the reference chain with local variables.

To close, here's an example of a sub-module that loads itself dynamically to its parent (creating it if it does not exist). I've left out private state for brevity, but including it would be simple. This code pattern allows an entire complex hierarchical code-base to be loaded completely in parallel with itself, sub-modules and all.

```
var UTIL = (function (parent, $) {
    var my = parent.ajax = parent.ajax || {};
    my.get = function (url, params, callback) {
        // ok, so I'm cheating a bit :)
        return $.getJSON(url, params, callback);
    };
    // etc...

    return parent;
}(UTIL || {}, jQuery));
```

CODE

I hope this has been useful, and please leave a comment to share your thoughts. Now, go forth and write better, more modular JavaScript!

This post was [featured on Ajaxian.com](#), and there is a little bit more discussion going on there as well, which is worth reading in addition to the comments below.

29. Top 13 JavaScript Mistakes

Top 13 JavaScript Mistakes

Posted on 10/13/2010 by **Prem Gurbani Frontend Architect**

Sergio Cinos Architecture Engineer

Recently we've defined a list of most common Javascript mistakes that developers make. These cover a wide variety of topics and I'm sure you will find among them at least one that you've committed yourself. We describe the theory behind each of the problems/bad practices and show concrete solution(s).

Do you think we're missing something obvious? Leave a comment and let us know about it.

1. Usage of for..in to iterate Arrays

Example:

CODE

```
var myArray = [ "a", "b", "c" ];
var totalElements = myArray.length;
for (var i = 0; i < totalElements; i++) {
    console.log(myArray[i]);
}
```

The main problem here is that the for..in statement does not guarantee the order. Effectively this means that you could obtain different results at different executions. Moreover, if someone augmented Array.prototype with some other custom functions, your loop will iterate over those functions as well as the original array items.

Solution: always use regular for loops to iterate arrays.

CODE

```
var myArray = [ "a", "b", "c" ];
for (var i=0; i<myArray.length; i++) {
    console.log(myArray[i]);
}
```

2. Array dimensions

Example:

CODE

```
var myArray = new Array(10);
```

There are two different problems here. First, the developer is trying to create an array already containing 10 items, and it will create an array with 10 empty slots. However, if you try to get an array item, you will get 'undefined' as result. In other words, the effect is the same as if you did not reserved that memory space. There is no really good reason to predefined the array length.

The second problem is that the developer is creating an array using Array constructor. This is technically correct. However it's slower than the literal notation.

Solution: Use literal notation to initialize arrays. Do not predefined array length.



```
var myArray = [];
```

3. Undefined properties

Example:



```
var myObject = {  
    someProperty: "value",  
    someOtherProperty: undefined  
}
```

Undefined properties (such as someOtherProperty in the above example) will create an element in the object with key 'someOtherProperty' and value 'undefined'. If you loop through your array checking the existence of an element the following 2 statements will both return 'undefined':

```
typeof myObject['someOtherProperty'] // undefined  
typeof myObject['unknownProperty'] // undefined
```

Solution: if you want to explicitly declare a uninitialized properties inside an object, mark them as null



```
var myObject = {  
    someProperty: "value",  
    someOtherProperty: null  
}
```

4. Misuse of Closures

Example:

CODE

```

function(a, b, c) {
    var d = 10;
    var element = document.getElementById('myID');
    element.onclick = (function(a, b, c, d) {
        return function() {
            alert (a + b + c + d);
        }
    })(a, b, c, d);
}

```

Here the developer is using two functions to pass the arguments a, b and c to the onclick handler. The double function is not needed only adding complexity to the code.

The variables a, b and c are already defined in the inner function because they are already declared as parameters in the main function. Any function inside the inner function will create a closure with all variables defined by main function. This includes 'regular' variables (like d) and arguments (like a, b and c). Thus It is not necessary to pass them again as parameters using a auto-executable function.

See [JavaScript Closures FAQ](#) for an awesome explanation about closures and contexts.

Solution: use closures to simplify your code

CODE

```

function (a, b, c) {
    var d = 10;
    var element = document.getElementById('myID');
    element.onclick = function() {
        //a, b, and c come from the outer function arguments.
        //d come from the outer function variable declarations.
        //and all of them are in my closure
        alert (a + b + c + d);
    };
}

```

5. Closures in loops

Example:

CODE

```

var elements = document.getElementsByTagName('div');
for (var i = 0; i<elements.length; i++) {
    elements[i].onclick = function() {
        alert("Div number " + i);
    }
}

```

In this example, we want to trigger an action (display "Div number 1", "Div number 2"... etc) when the user clicks on the different divs on the page. However, if we have 10 divs in the page, all of them will display "Div number 10".

The problem is that we are creating a closure with the inner function, so the code inside the function has access to variable i. The point is that i inside the function and i outside the function refers to the same variable (i.e.: the same position in memory). When our loop ends, i points to the value 10. So the value of i inside the inner function will be 10.

See [JavaScript Closures FAQ](#) for an awesome explanation about closures and contexts.

Solution: use a second function to pass the correct value.

CODE

```
var elements = document.getElementsByTagName('div');
for (var i = 0; i<elements.length; i++) {
    elements[i].onclick = (function(idx) { //Outer function
        return function() { //Inner function
            alert("Div number " + idx);
        }
    })(i);
}
```

The outer function is a function that executes immediately, receiving i as a parameter. That parameter is called idx inside the outer function, thus inner function creates a closure with idx (instead of i). Therefore idx is completely different from across iterations (i.e. they point to different memory address). This example is very important to understand how closures work. Be sure to read it and play with the code in your browser until you fully understand what's going on there.

6. Memory leaks with DOM objects

Example:

CODE

```
function attachEvents() {
    var element = document.getElementById('myID');
    element.onclick = function() {
        alert("Element clicked");
    }
};
attachEvents();
```

This code creates a reference loop. The variable element contains a reference to the function (it is assigned to onclick properties). Also, function is keeping a reference to the DOM element (note that inside the function you have access to element because of the closure). So JavaScript garbage collector cannot clean neither element nor the function, because both are referenced by each other. Most JavaScript engines aren't clever enough to clean circular references.

Solution: avoid those closures or undo the circular reference inside the function

CODE

```

function attachEvents() {
    var element = document.getElementById('myID');
    element.onclick = function() {
        //Remove element, so function can be collected by GC
        delete element;
        alert("Element clicked");
    }
};

attachEvents();

```

7. Differentiate float numbers from integer numbers

Example:

CODE

```

var myNumber = 3.5;
var myResult = 3.5 + 1.0; //We use .0 to keep the result as float

```

In JavaScript, there is no difference between float and integers. Actually, every number in JavaScript is represented using double-precision 64-bits format IEEE 754. In plain words, all numbers are floats.

Solution: don't use decimals to "convert" numbers to floats.

CODE

```

var myNumber = 3.5;
var myResult = 3.5 + 1; //Result is 4.5, as expected

```

8. Usage of with() as a shortcut

Example:

CODE

```

team.attackers.myWarrior = { attack: 1, speed: 3, magic: 5};
with (team.attackers.myWarrior){
    console.log ( "Your warrior power is " + (attack * speed));
}

```

Before talking about `with()`, let's see how JavaScript contexts works. Each function has an execution context that, put in simple words, holds all the variables that the function can access. Thus the context contain arguments and defined variables. Also a context points to a "parent" context (the context that our caller function has). For example, if `functionA()` calls `functionB()`, `functionB`'s context points to `functionA`'s context as its parent context. When accessing any variable inside a function, the engine first search in his own context. If not found, it switches to the parent context and so on until it finds the variable or it reaches the end of the context chain. Execution contexts are what makes closures work.

What `with()` actually does, is to insert an object into our context chain. It injects between my current context and my

parent's context. In this way, the engine first searches in the current context for the requested variable, and then it searches for it in the recently injected object, and finally in the "real" parent context. As you can see, the shortcut used in the example code above is a side effect of using context injection. However usage of `with()` is very slow, and thus using it for shortcuts is just insane.

Just a side note. Every book recommends not to use `with()`. Nevertheless all of them are focused on its usage as a shortcut. Context injection is really useful and you may find that you need to use it in advanced JavaScript. In those cases, it's acceptable to use `with()` with its real meaning. Although be aware that it's still very slow from a performance perspective.

See [JavaScript Closures FAQ](#) for an awesome explanation about closures and contexts.

Solution: don't use `with()` for shortcuts. Only for context injection when you really need it.

CODE

```
team.attackers.myWarrior = { attack: 1, speed: 3, magic: 5};
var sc = team.attackers.myWarrior;
console.log("Your warrior power is " + (sc.attack * sc.speed));
```

9. Usage of strings with setTimeout/setInterval

Example:

CODE

```
function log1() { console.log(document.location); }
function log2(arg) { console.log(arg); }
var myValue = "test";
setTimeout("log1()", 100);
setTimeout("log2(" + myValue + ")", 200);
```

Both `setTimeout()` and `setInterval()` can accept either a function or a string as the first parameter. If you pass a string, the engine will create a new function using Function constructor. This is very slow in some browsers. Instead pass the function itself as the first argument; it's faster, more powerful and clearer.

Solution: never use strings for `setTimeout()` or `setInterval()`

CODE

```
function log1() { console.log(document.location); }
function log2(arg) { console.log(arg); }
var myValue = "test";
setTimeout(log1, 100); //Reference to a function
setTimeout(function(){ //Get arg value using closures
    log2(arg);
}, 200);
```

10. Usage of setInterval() for heavy functions

Example:

CODE

```
function domOperations() {
    //Heavy DOM operations, takes about 300ms
}
setInterval(domOperations, 200);
```

We can have a problem when using intervals where operation time is bigger than the interval step time. In the example we are doing a complex (i.e.: long) operation with DOM objects every 200ms. If the `domOperations()` function takes more than 200ms to complete, each step will overlap with the previous step and eventually some steps may get discarded. This can become a problem.

`setInterval()` schedules a function to be executed only if there isn't another execution already waiting in the main execution queue. The JavaScript engine only adds the next execution to the queue if there is no another execution already in the queue. This may yield to skip executions or run two different executions without waiting 200ms between them. To make it clear, `setInterval()` doesn't take in account how long it takes `domOperations()` to complete its job.

Solution: avoid `setInterval()`, use `setTimeout()`

CODE

```
function domOperations() {
    //Heavy DOM operations, takes about 300ms

    //After all the job is done, set another timeout for 200 ms
    setTimeout(domOperations, 200);
}
setTimeout(domOperations, 200);
```

11. Misuse of 'this' There is no example for this common mistake as it is very difficult to build one to illustrate it. The value of this in JavaScript is very different from other languages, where the this behaviour is usually clearer. However, in JavaScript this is a bit different.

First of all, the value of this inside a function is defined when the function is called, not when it is declared. Its very important to understand this, because the value of this depends on how the function is called. In the following cases, this has a different meaning inside myFunction

* Regular function: `myFunction('arg1');`

this points to the global object, which is window for all browsers.

* Method: `someObject.myFunction('arg1');`

this points to object before the dot, someObject in this case.

* Constructor: `var something = new myFunction('arg1');`

this points to an empty Object.

* Using `call()`/`apply()`: `myFunction.call(someObject, 'arg1');`

this points to the object passed as first argument.

In this way you can have the same function (myFunction in the example above), that internally use this. However, the value of this is not related to the function declaration itself, only to the way that function is called.

12. Usage of eval() to access dynamic properties

Example:

CODE

```
var myObject = { p1: 1, p2: 2, p3: 3};
var i = 2;
var myResult = eval(`myObject.p'+i);
```

The main problem lies in that starting a new execution context with eval() is extremely slow. The same can be achieved using square bracket notation instead of dot notation.

Solution: use square bracket notation instead of eval()

CODE

```
var myObject = { p1: 1, p2: 2, p3: 3};
var i = 2;
var myResult = myObject["p"+i];
```

13. Usage of undefined as a variable

Example:

CODE

```
if ( myVar === undefined ) {
    //Do something
}
```

This check usually works, but it does by pure chance. In the code above undefined is effectively a variable. All JavaScript engines will create the variable window.undefined initialized to undefined as its value. However note that variables isn't read-only, and any other code can change its value. It's very weird to find a scenario where window.undefined has a value different from undefined. (nobody will never actually do undefined = 10;). But why take the risk? It is better to use typeof checks.

Solution: use typeof when checking for undefined.

CODE

```
if ( typeof myVar === "undefined" ) {
    //Do something
}
```

30. Javascript Closures

[FAQ](#) > [FAQ Notes](#)

- [Introduction](#)
- [The Resolution of Property Names on Objects](#)
 - [Assignment of Values](#)
 - [Reading of Values](#)
- [Identifier Resolution, Execution Contexts and scope chains](#)
 - [The Execution Context](#)
 - [scope chains and \[\[scope\]\]](#)
 - [Identifier Resolution](#)
- [Closures](#)
 - [Automatic Garbage Collection](#)
 - [Forming Closures](#)
- [What can be done with Closures?](#)
 - [Example 1: setTimeout with Function References](#)
 - [Example 2: Associating Functions with Object Instance Methods](#)
 - [Example 3: Encapsulating Related Functionality](#)
 - [Other Examples](#)
- [Accidental Closures](#)
- [The Internet Explorer Memory Leak Problem](#)

Introduction

Closure

A "closure" is an expression (typically a function) that can have free variables together with an environment that binds those variables (that "closes" the expression).

Closures are one of the most powerful features of ECMAScript (javascript) but they cannot be properly exploited without understanding them. They are, however, relatively easy to create, even accidentally, and their creation has potentially harmful consequences, particularly in some relatively common web browser environments. To avoid accidentally encountering the drawbacks and to take advantage of the benefits they offer it is necessary to understand their mechanism. This depends heavily on the role of scope chains in identifier resolution and so on the resolution of property names on objects.

The simple explanation of a Closure is that ECMAScript allows inner functions; function definitions and function expressions that are inside the function bodies of other functions. And that those inner functions are allowed access to all of the local variables, parameters and declared inner functions within their outer function(s). A closure is formed when one of those inner functions is made accessible outside of the function in which it was contained, so that it may be executed after the outer function has returned. At which point it still has access to the local variables, parameters and inner function declarations of its outer function. Those local variables, parameter and function declarations (initially) have the values that they had when the outer function returned and may be interacted with by the inner function.

Unfortunately, properly understanding closures requires an understanding of the mechanism behind them, and quite a bit of technical detail. While some of the ECMA 262 specified algorithms have been brushed over in the early part of the following explanation, much cannot be omitted or easily simplified. Individuals familiar with object property name resolution may skip that section but only people already familiar with closures can afford to skip the following sections, and they can stop reading now and get back to exploiting them.

The Resolution of Property Names on Objects

ECMAScript recognises two categories of object, "Native Object" and "Host Object" with a sub-category of native objects called "Built-in Object" (ECMA 262 3rd Ed Section 4.3). Native objects belong to the language and host objects are provided by the environment, and may be, for example, document objects, DOM nodes and the like.

Native objects are loose and dynamic bags of named properties (some implementations are not that dynamic when it comes to the built in object sub-category, though usually that doesn't matter). The defined named properties of an object will hold a value, which may be a reference to another Object (functions are also Objects in this sense) or a primitive value: String, Number, Boolean, Null or Undefined. The Undefined primitive type is a bit odd in that it is possible to assign a value of Undefined to a property of an object but doing so does not remove that property from the object; it remains a defined named property, it just holds the value `undefined`.

The following is a simplified description of how property values are read and set on objects with the internal details brushed over to the greatest extent possible.

Assignment of Values

Named properties of objects can be created, or values set on existing named properties, by assigning a value to that named property. So given:-

```
var objectRef = new Object(); //create a generic javascript object.
```

CODE

A property with the name "testNumber" can be created as:-

```
objectRef.testNumber = 5;
/* - or:- */
objectRef["testNumber"] = 5;
```

CODE

The object had no "testNumber" property prior to the assignment but one is created when the assignment is made. Any subsequent assignment does not need to create the property, it just re-sets its value:-

```
objectRef.testNumber = 8;
/* - or:- */
objectRef["testNumber"] = 8;
```

CODE

Javascript objects have prototypes that can themselves be objects, as will be described shortly, and that prototype may have named properties. But this has no role in assignment. If a value is assigned and the actual object does not have a property with the corresponding name a property of that name is created and the value is assigned to it. If it has the property then its value is re-set.

Reading of Values

It is in reading values from object properties that prototypes come into play. If an object has a property with the property name used in the property accessor then the value of that property is returned:-

CODE

```
/* Assign a value to a named property. If the object does not have a
   property with the corresponding name prior to the assignment it
   will have one after it:-
*/
objectRef.testNumber = 8;

/* Read the value back from the property:- */

var val = objectRef.testNumber;
/* and - val - now holds the value 8 that was just assigned to the
   named property of the object. */
```

But all objects may have prototypes, and prototypes are objects so they, in turn, may have prototypes, which may have prototypes, and so on forming what is called the prototype chain. The prototype chain ends when one of the objects in the chain has a null prototype. The default prototype for the `Object` constructor has a null prototype so:-

CODE

```
var objectRef = new Object(); //create a generic javascript object.
```

Creates an object with the prototype `Object.prototype` that itself has a null prototype. So the prototype chain for `objectRef` contains only one object: `Object.prototype`. However:-

CODE

```

/* A "constructor" function for creating objects of a -
   MyObject1 - type.
*/
function MyObject1(formalParameter){
    /* Give the constructed object a property called - testNumber - and
       assign it the value passed to the constructor as its first
       argument:-*/
    this.testNumber = formalParameter;
}

/* A "constructor" function for creating objects of a -
   MyObject2 - type:-*/
function MyObject2(formalParameter){
    /* Give the constructed object a property called - testString -
       and assign it the value passed to the constructor as its first
       argument:-*/
    this.testString = formalParameter;
}

/* The next operation replaces the default prototype associated with
   all MyObject2 instances with an instance of MyObject1, passing the
   argument - 8 - to the MyObject1 constructor so that its -
   testNumber - property will be set to that value:-*/
MyObject2.prototype = new MyObject1( 8 );

/* Finally, create an instance of - MyObject2 - and assign a reference
   to that object to the variable - objectRef - passing a string as the
   first argument for the constructor:-*/
var objectRef = new MyObject2( "String_Value" );

```

The instance of `MyObject2` referred to by the `objectRef` variable has a prototype chain. The first object in that chain is the instance of `MyObject1` that was created and assigned to the `prototype` property of the `MyObject2` constructor. The instance of `MyObject1` has a prototype, the object that was assigned to the function `MyObject1`'s `prototype` property by the implementation. That object has a prototype, the default `Object` prototype that corresponds with the object referred to by `Object.prototype`. `Object.prototype` has a null prototype so the prototype chain comes to an end at this point.

When a property accessor attempts to read a named property from the object referred to by the variable `objectRef` the whole prototype chain can enter into the process. In the simple case:-

CODE

```
var val = objectRef.testString;
```

- the instance of `MyObject2` referred to by `objectRef` has a property with the name "testString" so it is the value of that property, set to "String_Value", that is assigned to the variable `val`. However:-

CODE

```
var val = objectRef.testNumber;
```

- cannot read a named property from the instance of `MyObject2` itself as it has no such property but the variable `val` is set to the value of `8` rather than `undefined` because having failed to find a corresponding named property on the object itself the interpreter then examines the object that is its prototype. Its prototype is the instance of `MyObject1` and it was created with a property named "testNumber" with the value `8` assigned to that property, so the property accessor evaluates as the value `8`. Neither `MyObject1` or `MyObject2` have defined a `toString` method, but if a property accessor attempts to read the value of a `toString` property from `objectRef` :-

```
var val = objectRef.toString;
```

CODE

- the `val` variable is assigned a reference to a function. That function is the `toString` property of `Object.prototype` and is returned because the process of examining the prototype of `objectRef`, when `objectRef` turns out not to have a "toString" property, is acting on an object, so when that prototype is found to lack the property its prototype is examined in turn. Its prototype is `Object.prototype`, which does have a `toString` method so it is a reference to that function object that is returned.

Finally:-

```
var val = objectRef.madeUpProperty;
```

CODE

- returns `undefined`, because as the process of working up the prototype chain finds no properties on any of the object with the name "madeUpProperty" it eventually gets to the prototype of `Object.prototype`, which is `null`, and the process ends returning `undefined`.

The reading of named properties returns the first value found, on the object or then from its prototype chain. The assigning of a value to a named property on an object will create a property on the object itself if no corresponding property already exists.

This means that if a value was assigned as `objectRef.testNumber = 3` a "testNumber" property will be created on the instance of `MyObject2` itself, and any subsequent attempts to read that value will retrieve that value as set on the object. The prototype chain no longer needs to be examined to resolve the property accessor, but the instance of `MyObject1` with the value of `8` assigned to its "testNumber" property is unaltered. The assignment to the `objectRef` object masks the corresponding property in its prototype chain.

Note: ECMAScript defines an internal `[[prototype]]` property of the internal Object type. This property is not directly accessible with scripts, but it is the chain of objects referred to with the internal `[[prototype]]` property that is used in property accessor resolution; the object's prototype chain. A public `prototype` property exists to allow the assignment, definition and manipulation of prototypes in association with the internal `[[prototype]]` property. The details of the relationship between the two are described in ECMA 262 (3rd edition) and are beyond the scope of this discussion.

Identifier Resolution, Execution Contexts and scope chains

The Execution Context

An *execution context* is an abstract concept used by the ECMAScript specification (ECMA 262 3rd edition) to define the behaviour required of ECMAScript implementations. The specification does not say anything about how *execution contexts* should be implemented but execution contexts have associated attributes that refer to specification defined

structures so they might be conceived (and even implemented) as objects with properties, though not public properties.

All javascript code is executed in an *execution context*. Global code (code executed inline, normally as a JS file, or HTML page, loads) gets executed in *global execution context*, and each invocation of a function (possibly as a constructor) has an associated *execution context*. Code executed with the `eval` function also gets a distinct execution context but as `eval` is never normally used by javascript programmers it will not be discussed here. The specified details of *execution contexts* are to be found in section 10.2 of ECMA 262 (3rd edition).

When a javascript function is called it enters an *execution context*, if another function is called (or the same function recursively) a new *execution context* is created and execution enters that context for the duration of the function call. Returning to the original execution context when that called function returns. Thus running javascript code forms a stack of *execution contexts*.

When an *execution context* is created a number of things happen in a defined order. First, in the *execution context* of a function, an "Activation" object is created. The activation object is another specification mechanism. It can be considered as an object because it ends up having accessible named properties, but it is not a normal object as it has no prototype (at least not a defined prototype) and it cannot be directly referenced by javascript code.

The next step in the creation of the *execution context* for a function call is the creation of an `arguments` object, which is an array-like object with integer indexed members corresponding with the arguments passed to the function call, in order. It also has `length` and `callee` properties (which are not relevant to this discussion, see the spec for details). A property of the Activation object is created with the name "arguments" and a reference to the `arguments` object is assigned to that property.

Next the *execution context* is assigned a *scope*. A *scope* consists of a list (or chain) of objects. Each function object has an internal `[[scope]]` property (which we will go into more detail about shortly) that also consists of a list (or chain) of objects. The *scope* that is assigned to the *execution context* of a function call consists of the list referred to by the `[[scope]]` property of the corresponding function object with the Activation object added at the front of the chain (or the top of the list).

Then the process of "variable instantiation" takes place using an object that ECMA 262 refers to as the "Variable" object. However, the Activation object is used as the Variable object (note this, it is important: they are the same object). Named properties of the Variable object are created for each of the function's formal parameters, and if arguments to the function call correspond with those parameters the values of those arguments are assigned to the properties (otherwise the assigned value is `undefined`). Inner function definitions are used to create function objects which are assigned to properties of the Variable object with names that correspond to the function name used in the function declaration. The last stage of variable instantiation is to create named properties of the Variable object that correspond with all the local variables declared within the function.

The properties created on the Variable object that correspond with declared local variables are initially assigned `undefined` values during variable instantiation, the actual initialisation of local variables does not happen until the evaluation of the corresponding assignment expressions during the execution of the function body code.

It is the fact that the Activation object, with its `arguments` property, and the Variable object, with named properties corresponding with function local variables, are the same object, that allows the identifier `arguments` to be treated as if it was a function local variable.

Finally a value is assigned for use with the `this` keyword. If the value assigned refers to an object then property accessors prefixed with the `this` keyword reference properties of that object. If the value assigned (internally) is null then the `this` keyword will refer to the global object.

The global execution context gets some slightly different handling as it does not have arguments so it does not need a defined Activation object to refer to them. The global execution context does need a *scope* and its *scope chain* consists of exactly one object, the global object. The global execution context does go through variable instantiation, its inner functions are the normal top level function declarations that make up the bulk of javascript code. The global object is used as the Variable object, which is why globally declared functions become properties of the global object. As do globally declared variables.

The global execution context also uses a reference to the global object for the `this` object.

scope chains and [[scope]]

The *scope chain* of the execution context for a function call is constructed by adding the execution context's Activation/Variable object to the front of the *scope chain* held in the function object's `[[scope]]` property, so it is important to understand how the internal `[[scope]]` property is defined.

In ECMAScript functions are objects, they are created during variable instantiation from function declarations, during the evaluation of function expressions or by invoking the `Function` constructor.

Function objects created with the `Function` constructor always have a `[[scope]]` property referring to a *scope chain* that only contains the global object.

Function objects created with function declarations or function expressions have the *scope chain* of the execution context in which they are created assigned to their internal `[[scope]]` property.

In the simplest case of a global function declaration such as:-

```
function exampleFunction(formalParameter){
    ... // function body code
}
```

CODE

- the corresponding function object is created during the variable instantiation for the global execution context. The global execution context has a *scope chain* consisting of only the global object. Thus the function object that is created and referred to by the property of the global object with the name "exampleFunction" is assigned an internal `[[scope]]` property referring to a *scope chain* containing only the global object.

A similar *scope chain* is assigned when a function expression is executed in the global context:-

```
var exampleFuncRef = function(){
    ... // function body code
}
```

CODE

- except in this case a named property of the global object is created during variable instantiation for the global execution context but the function object is not created, and a reference to it assigned to the named property of the global object, until the assignment expression is evaluated. But the creation of the function object still happens in the global execution context so the `[[scope]]` property of the created function object still only contains the global object in the assigned scope chain.

Inner function declarations and expressions result in function objects being created within the execution context of a function so they get more elaborate scope chains. Consider the following code, which defines a function with an inner

function declaration and then executes the outer function:-

```
function exampleOuterFunction(formalParameter){
    function exampleInnerFuncitonDec(){
        ... // inner function body
    }
    ... // the rest of the outer function body.
}

exampleOuterFunction( 5 );
```

CODE

The function object corresponding with the outer function declaration is created during variable instantiation in the global execution context so its `[[scope]]` property contains the one item scope chain with only the global object in it.

When the global code executes the call to the `exampleOuterFunction` a new execution context is created for that function call and an Activation/Variable object along with it. The `scope` of that new execution context becomes the chain consisting of the new Activation object followed by the chain referred to by the outer function object's `[[scope]]` property (just the global object). Variable instantiation for that new execution context results in the creation of a function object that corresponds with the inner function definition and the `[[scope]]` property of that function object is assigned the value of the `scope` from the execution context in which it was created. A *scope chain* that contains the Activation object followed by the global object.

So far this is all automatic and controlled by the structure and execution of the source code. The *scope chain* of the execution context defines the `[[scope]]` properties of the function objects created and the `[[scope]]` properties of the function objects define the `scope` for their execution contexts (along with the corresponding Activation object). But ECMAScript provides the `with` statement as a means of modifying the scope chain.

The `with` statement evaluates an expression and if that expression is an object it is added to the *scope chain* of the current execution context (in front of the Activation/Variable object). The `with` statement then executes another statement (that may itself be a block statement) and then restores the execution context's *scope chain* to what it was before.

A function declaration could not be affected by a `with` statement as they result in the creation of function objects during variable instantiation, but a function expression can be evaluated inside a `with` statement:-

```

/* create a global variable - y - that refers to an object:- */
var y = {x:5}; // object literal with an - x - property
function exampleFuncWith(){
    var z;
    /* Add the object referred to by the global variable - y - to the
       front of the scope chain:- */
    /*
    with(y){
        /* evaluate a function expression to create a function object
           and assign a reference to that function object to the local
           variable - z - :- */
        /*
        z = function(){
            ... // inner function expression body;
        }
    }
    ...
}

/* execute the - exampleFuncWith - function:- */
exampleFuncWith();

```

When the `exampleFuncWith` function is called the resulting execution context has a *scope chain* consisting of its Activation object followed by the global object. The execution of the `with` statement adds the object referred to by the global variable `y` to the front of that *scope chain* during the evaluation of the function expression. The function object created by the evaluation of the function expression is assigned a `[[scope]]` property that corresponds with the *scope* of the execution context in which it is created. A *scope chain* consisting of object `y` followed by the Activation object from the execution context of the outer function call, followed by the global object.

When the block statement associated with the `with` statement terminates the *scope* of the execution context is restored (the `y` object is removed), but the function object has been created at that point and its `[[scope]]` property assigned a reference to a *scope chain* with the `y` object at its head.

Identifier Resolution

Identifiers are resolved against the scope chain. ECMA 262 categorises `this` as a keyword rather than an identifier, which is not unreasonable as it is always resolved dependent on the `this` value in the execution context in which it is used, without reference to the scope chain.

Identifier resolution starts with the first object in the scope chain. It is checked to see if it has a property with a name that corresponds with the identifier. Because the *scope chain* is a chain of objects this checking encompasses the prototype chain of that object (if it has one). If no corresponding value can be found on the first object in the *scope chain* the search progresses to the next object. And so on until one of the objects in the chain (or one of its prototypes) has a property with a name that corresponds with the identifier or the scope chain is exhausted.

The operation on the identifier happens in the same way as the use of property accessors on objects described above. The object identified in the *scope chain* as having the corresponding property takes the place of the object in the property accessor and the identifier acts as a property name for that object. The global object is always at the end of the scope chain.

As execution contexts associated with function calls will have the Activation/Variable object at the front of the chain, identifiers used in function bodies are effectively first checked to see whether they correspond with formal

parameters, inner function declaration names or local variables. Those would be resolved as named properties of the Activation/Variable object.

Closures

Automatic Garbage Collection

ECMAScript uses automatic garbage collection. The specification does not define the details, leaving that to the implementers to sort out, and some implementations are known to give a very low priority to their garbage collection operations. But the general idea is that if an object becomes un-referable (by having no remaining references to it left accessible to executing code) it becomes available for garbage collection and will at some future point be destroyed and any resources it is consuming freed and returned to the system for re-use.

This would normally be the case upon exiting an execution context. The *scope chain* structure, the Activation/Variable object and any objects created within the execution context, including function objects, would no longer be accessible and so would become available for garbage collection.

Forming Closures

A closure is formed by returning a function object that was created within an execution context of a function call from that function call and assigning a reference to that inner function to a property of another object. Or by directly assigning a reference to such a function object to, for example, a global variable, a property of a globally accessible object or an object passed by reference as an argument to the outer function call. e.g:-

```
function exampleClosureForm(arg1, arg2){
    var localVar = 8;
    function exampleReturned(innerArg){
        return ((arg1 + arg2)/(innerArg + localVar));
    }
    /* return a reference to the inner function defined as -
       exampleReturned -:-
    */
    return exampleReturned;
}

var globalVar = exampleClosureForm(2, 4);
```

CODE

Now the function object created within the execution context of the call to `exampleClosureForm` cannot be garbage collected because it is referred to by a global variable and is still accessible, it can even be executed with `globalVar(n)`.

But something a little more complicated has happened because the function object now referred to by `globalVar` was created with a `[[scope]]` property referring to a scope chain containing the Activation/Variable object belonging to the execution context in which it was created (and the global object). Now the Activation/Variable object cannot be garbage collected either as the execution of the function object referred to by `globalVar` will need to add the whole *scope chain* from its `[[scope]]` property to the *scope* of the execution context created for each call to it.

A closure is formed. The inner function object has the free variables and the Activation/Variable object on the function's *scope chain* is the environment that binds them.

The Activation/Variable object is trapped by being referred to in the *scope chain* assigned to the internal `[[scope]]` property of the function object now referred to by the `globalVar` variable. The Activation/Variable object is preserved along with its state; the values of its properties. Scope resolution in the execution context of calls to the inner function will resolve identifiers that correspond with named properties of that Activation/Variable object as properties of that object. The value of those properties can still be read and set even though the execution context for which it was created has exited.

In the example above that Activation/Variable object has a state that represents the values of formal parameters, inner function definitions and local variables, at the time when the outer function returned (exited its execution context). The `arg1` property has the value `2`, the `arg2` property the value `4`, `localVar` the value `8` and an `exampleReturned` property that is a reference to the inner function object that was returned from the outer function. (We will be referring to this Activation/Variable object as "ActOuter1" in later discussion, for convenience.)

If the `exampleClosureForm` function was called again as:-

```
var secondGlobalVar = exampleClosureForm(12, 3);
```

CODE

- a new execution context would be created, along with a new Activation object. And a new function object would be returned, with its own distinct `[[scope]]` property referring to a scope chain containing the Activation object from this second execution context, with `arg1` being `12` and `arg2` being `3`. (We will be referring to this Activation/Variable object as "ActOuter2" in later discussion, for convenience.)

A second and distinct closure has been formed by the second execution of `exampleClosureForm`.

The two function objects created by the execution of `exampleClosureForm` to which references have been assigned to the global variable `globalVar` and `secondGlobalVar` respectively, return the expression `((arg1 + arg2)/(innerArg + localVar))`. Which applies various operators to four identifiers. How these identifiers are resolved is critical to the use and value of closures.

Consider the execution of the function object referred to by `globalVar`, as `globalVar(2)`. A new execution context is created and an Activation object (we will call it "ActInner1"), which is added to the head of the scope chain referred to the `[[scope]]` property of the executed function object. ActInner1 is given a property named `innerArg`, after its formal parameter and the argument value `2` assigned to it. The *scope chain* for this new execution context is: `ActInner1-> ActOuter1-> global object`.

Identifier resolution is done against the *scope chain* so in order to return the value of the expression `((arg1 + arg2)/(innerArg + localVar))` the values of the identifiers will be determined by looking for properties, with names corresponding with the identifiers, on each object in the scope chain in turn.

The first object in the chain is ActInner1 and it has a property named `innerArg` with the value `2`. All of the other 3 identifiers correspond with named properties of ActOuter1; `arg1` is `2`, `arg2` is `4` and `localVar` is `8`. The function call returns `((2 + 4)/(2 + 8))`.

Compare that with the execution of the otherwise identical function object referred to by `secondGlobalVar`, as `secondGlobalVar(5)`. Calling the Activation object for this new execution context "ActInner2", the scope chain becomes: `ActInner2-> ActOuter2-> global object`. ActInner2 returns `innerArg` as `5` and ActOuter2 returns `arg1`, `arg2` and `localVar` as `12`, `3` and `8` respectively. The value returned is `((12 + 3)/(5 + 8))`.

Execute `secondGlobalVar` again and a new Activation object will appear at the front of the *scope chain* but ActOuter2 will still be next object in the chain and the value of its named properties will again be used in the

resolution of the identifiers `arg1`, `arg2` and `localVar`.

This is how ECMAScript inner functions gain, and maintain, access to the formal parameters, declared inner functions and local variables of the execution context in which they were created. And it is how the forming of a closure allows such a function object to keep referring to those values, reading and writing to them, for as long as it continues to exist. The Activation/Variable object from the execution context in which the inner function was created remains on the scope chain referred to by the function object's `[[scope]]` property, until all references to the inner function are freed and the function object is made available for garbage collection (along with any now unneeded objects on its scope chain).

Inner function may themselves have inner functions, and the inner functions returned from the execution of functions to form closures may themselves return inner functions and form closures of their own. With each nesting the *scope chain* gains extra Activation objects originating with the execution contexts in which the inner function objects were created. The ECMAScript specification requires a scope chain to be finite, but imposes no limits on their length. Implementations probably do impose some practical limitation but no specific magnitude has yet been reported. The potential for nesting inner functions seems so far to have exceeded anyone's desire to code them.

What can be done with Closures?

Strangely the answer to that appears to be anything and everything. I am told that closures enable ECMAScript to emulate anything, so the limitation is the ability to conceive and implement the emulation. That is a bit esoteric and it is probably better to start with something a little more practical.

Example 1: setTimeout with Function References

A common use for a closure is to provide parameters for the execution of a function prior to the execution of that function. For example, when a function is to be provided as the first argument to the `setTimeout` function that is common in web browser environments.

`setTimeout` schedules the execution of a function (or a string of javascript source code, but not in this context), provided as its first argument, after an interval expressed in milliseconds (as its second argument). If a piece of code wants to use `setTimeout` it calls the `setTimeout` function and passes a reference to a function object as the first argument and the millisecond interval as the second, but a reference to a function object cannot provide parameters for the scheduled execution of that function.

However, code could call another function that returned a reference to an inner function object, with that inner function object being passed by reference to the `setTimeout` function. The parameters to be used for the execution of the inner function are passed with the call to the function that returns it. `setTimout` executes the inner function without passing arguments but that inner function can still access the parameters provided by the call to the outer function that returned it:-

```

function callLater(paramA, paramB, paramC){
    /* Return a reference to an anonymous inner function created
       with a function expression:- */
    return (function(){
        /* This inner function is to be executed with - setTimeout
           - and when it is executed it can read, and act upon, the
           parameters passed to the outer function:- */
        paramA[paramB] = paramC;
    });
}

...
/* Call the function that will return a reference to the inner function
   object created in its execution context. Passing the parameters that
   the inner function will use when it is eventually executed as
   arguments to the outer function. The returned reference to the inner
   function object is assigned to a local variable:- */
var functRef = callLater(elStyle, "display", "none");
/* Call the setTimeout function, passing the reference to the inner
   function assigned to the - functRef - variable as the first argument:- */
hideMenu=setTimeout(functRef, 500);

```

Example 2: Associating Functions with Object Instance Methods

There are many other circumstances when a reference to a function object is assigned so that it would be executed at some future time where it is useful to provide parameters for the execution of that function that would not be easily available at the time of execution but cannot be known until the moment of assignment.

One example might be a javascript object that is designed to encapsulate the interactions with a particular DOM element. It has `doOnClick`, `doMouseOver` and `doMouseOut` methods and wants to execute those methods when the corresponding events are triggered on the DOM element, but there may be any number of instances of the javascript object created associated with different DOM elements and the individual object instances do not know how they will be employed by the code that instantiated them. The object instances do not know how to reference themselves globally because they do not know which global variables (if any) will be assigned references to their instances.

So the problem is to execute an event handling function that has an association with a particular instance of the javascript object, and knows which method of that object to call.

The following example uses a small generalised closure based function that associates object instances with element event handlers. Arranging that the execution of the event handler calls the specified method of the object instance, passing the event object and a reference to the associated element on to the object method and returning the method's return value.

```

/*
 * A general function that associates an object instance with an event
 * handler. The returned inner function is used as the event handler.
 * The object instance is passed as the - obj - parameter and the name
 * of the method that is to be called on that object is passed as the -
 * methodName - (string) parameter.
*/
function associateObjWithEvent(obj, methodName){
    /* The returned inner function is intended to act as an event
     * handler for a DOM element:- */
    return (function(e){
        /* The event object that will have been parsed as the - e -
         * parameter on DOM standard browsers is normalised to the IE
         * event object if it has not been passed as an argument to the
         * event handling inner function:- */
        e = e||window.event;
        /* The event handler calls a method of the object - obj - with
         * the name held in the string - methodName - passing the now
         * normalised event object and a reference to the element to
         * which the event handler has been assigned using the - this -
         * (which works because the inner function is executed as a
         * method of that element because it has been assigned as an
         * event handler):- */
        return obj[methodName](e, this);
    });
}

/* This constructor function creates objects that associates themselves
 * with DOM elements whose IDs are passed to the constructor as a
 * string. The object instances want to arrange than when the
 * corresponding element triggers onclick, onmouseover and onmouseout
 * events corresponding methods are called on their object instance.
*/
function DhtmlObject(elementId){
    /* A function is called that retrieves a reference to the DOM
     * element (or null if it cannot be found) with the ID of the
     * required element passed as its argument. The returned value
     * is assigned to the local variable - el -:- */
    var el = getElementById(elementId);
    /* The value of - el - is internally type-converted to boolean for
     * the - if - statement so that if it refers to an object the
     * result will be true, and if it is null the result false. So that
     * the following block is only executed if the - el - variable
     * refers to a DOM element:- */
    if(el){
        /* To assign a function as the element's event handler this
         * object calls the - associateObjWithEvent - function
         * specifying itself (with the - this - keyword) as the object
         * on which a method is to be called and providing the name of
         * the method that is to be called. The - associateObjWithEvent
         * - function will return a reference to an inner function that
         * is assigned to the event handler of the DOM element. That
         * inner function will call the required method on the
         * javascript object when it is executed in response to
         * events:- */
    }
}

```

```

        el.onclick = associateObjWithEvent(this, "doOnClick");
        el.onmouseover = associateObjWithEvent(this, "doMouseOver");
        el.onmouseout = associateObjWithEvent(this, "doMouseOut");
        ...
    }
}

DhtmlObject.prototype.doOnClick = function(event, element){
    ... // doOnClick method body.
}

DhtmlObject.prototype.doMouseOver = function(event, element){
    ... // doMouseOver method body.
}

DhtmlObject.prototype.doMouseOut = function(event, element){
    ... // doMouseOut method body.
}

```

And so any instances of the `DhtmlObject` can associate themselves with the DOM element that they are interested in without any need to know anything about how they are being employed by other code, impacting on the global namespace or risking clashes with other instances of the `DhtmlObject`.

Example 3: Encapsulating Related Functionality

Closures can be used to create additional scopes that can be used to group interrelated and dependent code in a way that minimises the risk of accidental interaction. Suppose a function is to build a string and to avoid the repeated concatenation operations (and the creation of numerous intermediate strings) the desire is to use an array to store the parts of the string in sequence and then output the results using the `Array.prototype.join` method (with an empty string as its argument). The array is going to act as a buffer for the output, but defining it locally to the function will result in its re-creation on each execution of the function, which may not be necessary if the only variable content of that array will be re-assigned on each function call.

One approach might make the array a global variable so that it can be re-used without being re-created. But the consequences of that will be that, in addition to the global variable that refers to the function that will use the buffer array, there will be a second global property that refers to the array itself. The effect is to render the code less manageable, as, if it is to be used elsewhere, its author has to remember to include both the function definition and the array definition. It also makes the code less easy to integrate with other code because instead of just ensuring that the function name is unique within the global namespace it is necessary to ensure that the `Array` on which it is dependent is using a name that is unique within the global namespace.

A Closure allows the buffer array to be associated (and neatly packaged) with the function that is dependent upon it and simultaneously keep the property name to which the buffer array was assigned out of the global namespace and free of the risk of name conflicts and accidental interactions.

The trick here is to create one additional execution context by executing a function expression in-line and have that function expression return an inner function that will be the function that is used by external code. The buffer array is then defined as a local variable of the function expression that is executed in-line. That only happens once so the `Array` is only created once, but is available to the function that depends on it for repeated use.

The following code creates a function that will return a string of HTML, much of which is constant, but those constant character sequences need to be interspersed with variable information provided as parameter to the function call.

A reference to an inner function object is returned from the in-line execution of a function expression and assigned to a global variable so that it can be called as a global function. The buffer array is defined as a local variable in the

outer function expression. It is not exposed in the global namespace and does not need to be re-created whenever the function that uses it is called.

```

/*
 * A global variable - getImgInPositionedDivHtml - is declared and
 assigned the value of an inner function expression returned from
 a one-time call to an outer function expression.

That inner function returns a string of HTML that represents an
absolutely positioned DIV wrapped round an IMG element, such that
all of the variable attribute values are provided as parameters
to the function call:-

*/
var getImgInPositionedDivHtml = (function(){
    /* The - buffAr - Array is assigned to a local variable of the
       outer function expression. It is only created once and that one
       instance of the array is available to the inner function so that
       it can be used on each execution of that inner function.

Empty strings are used as placeholders for the date that is to
be inserted into the Array by the inner function:-

*/
var buffAr = [
    '<div id="',
    '' , //index 1, DIV ID attribute
    '" style="position:absolute;top:',
    '' , //index 3, DIV top position
    'px;left:',
    '' , //index 5, DIV left position
    'px;width:',
    '' , //index 7, DIV width
    'px;height:',
    '' , //index 9, DIV height
    'px;overflow:hidden;"></div>'
];
/* Return the inner function object that is the result of the
evaluation of a function expression. It is this inner function
object that will be executed on each call to -
getImgInPositionedDivHtml( ... ) ---

*/
return (function(url, id, width, height, top, left, altText){
    /* Assign the various parameters to the corresponding
       locations in the buffer array:-

    */
    buffAr[1] = id;
    buffAr[3] = top;
    buffAr[5] = left;
    buffAr[13] = (buffAr[7] = width);
    buffAr[15] = (buffAr[9] = height);
    buffAr[11] = url;
    buffAr[17] = altText;
    /* Return the string created by joining each element in the
       array using an empty string (which is the same as just
       joining the elements together):-

    */
    return buffAr.join('');
}

```

```
}); //:End of inner function expression.  
})();  
/*^- :The inline execution of the outer function expression. */
```

If one function was dependent on one (or several) other functions, but those other functions were not expected to be directly employed by any other code, then the same technique could be used to group those functions with the one that was to be publicly exposed. Making a complex multi-function process into an easily portable and encapsulated unit of code.

Other Examples

Probably one of the best known applications of closures is [Douglas Crockford's technique for the emulation of private instance variables in ECMAScript objects](#). Which can be extended to all sorts of structures of scope contained nested accessibility/visibility, including [the emulation of private static members for ECMAScript objects](#).

The possible application of closures are endless, understanding how they work is probably the best guide to realising how they can be used.

Accidental Closures

Rendering any inner function accessible outside of the body of the function in which it was created will form a closure. That makes closures very easy to create and one of the consequences is that javascript authors who do not appreciate closures as a language feature can observe the use of inner functions for various tasks and employ inner functions, with no apparent consequences, not realising that closures are being created or what the implications of doing that are.

Accidentally creating closures can have harmful side effects as the following section on the IE memory leak problem describes, but they can also impact of the efficiency of code. It is not the closures themselves, indeed carefully used they can contribute significantly towards the creation of efficient code. It is the use of inner functions that can impact on efficiency.

A common situation is where inner functions are used as event handlers for DOM elements. For example the following code might be used to add an onclick handler to a link element:-

CODE

```

/* Define the global variable that is to have its value added to the
 - href - of a link as a query string by the following function:-
*/
var quantaty = 5;
/* When a link passed to this function (as the argument to the function
 call - linkRef -) an onclick event handler is added to the link that
 will add the value of a global variable - quantaty - to the - href -
 of that link as a query string, then return true so that the link
 will navigate to the resource specified by the - href - which will
 by then include the assigned query string:-
*/
function addGlobalQueryOnClick(linkRef){
    /* If the - linkRef - parameter can be type converted to true
       (which it will if it refers to an object):-
    */
    if(linkRef){
        /* Evaluate a function expression and assign a reference to the
           function object that is created by the evaluation of the
           function expression to the onclick handler of the link
           element:-
        */
        linkRef.onclick = function(){
            /* This inner function expression adds the query string to
               the - href - of the element to which it is attached as
               an event handler:-
            */
            this.href += ('?quantaty=' + escape(quantaty));
            return true;
        };
    }
}

```

Whenever the `addGlobalQueryOnClick` function is called a new inner function is created (and a closure formed by its assignment). From the efficiency point of view that would not be significant if the `addGlobalQueryOnClick` function was only called once or twice, but if the function was heavily employed many distinct function objects would be created (one for each evaluation of the inner function expression).

The above code is not taking advantage of the fact that inner functions are becoming accessible outside of the function in which they are being created (or the resulting closures). As a result exactly the same effect could be achieved by defining the function that is to be used as the event handler separately and then assigning a reference to that function to the event handling property. Only one function object would be created and all of the elements that use that event handler would share a reference to that one function:-

CODE

```
/* Define the global variable that is to have its value added to the
 - href - of a link as a query string by the following function:-
*/
var quantaty = 5;

/* When a link passed to this function (as the argument to the function
 call - linkRef -) an onclick event handler is added to the link that
 will add the value of a global variable - quantaty - to the - href -
 of that link as a query string, then return true so that the link
 will navigate to the resource specified by the - href - which will
 by then include the assigned query string:-
*/
function addGlobalQueryOnClick(linkRef){
    /* If the - linkRef - parameter can be type converted to true
       (which it will if it refers to an object):-
    */
    if(linkRef){
        /* Assign a reference to a global function to the event
           handling property of the link so that it becomes the
           element's event handler:-
        */
        linkRef.onclick = forAddQueryOnClick;
    }
}
/* A global function declaration for a function that is intended to act
   as an event handler for a link element, adding the value of a global
   variable to the - href - of an element as an event handler:-
*/
function forAddQueryOnClick(){
    this.href += ('?quantaty='+escape(quantaty));
    return true;
}
```

As the inner function in the first version is not being used to exploit the closures produced by its use, it would be more efficient not to use an inner function, and thus not repeat the process of creating many essentially identical function objects.

A similar consideration applies to object constructor functions. It is not uncommon to see code similar to the following skeleton constructor:-

CODE

```

function ExampleConst(param){
    /* Create methods of the object by evaluating function expressions
       and assigning references to the resulting function objects
       to the properties of the object being created:- */
    this.method1 = function(){
        ... // method body.
    };
    this.method2 = function(){
        ... // method body.
    };
    this.method3 = function(){
        ... // method body.
    };
    /* Assign the constructor's parameter to a property of the object:- */
    this.publicProp = param;
}

```

Each time the constructor is used to create an object, with `new ExampleConst(n)`, a new set of function objects are created to act as its methods. So the more object instances that are created the more function objects are created to go with them.

Douglas Crockford's technique for emulating private members on javascript objects exploits the closure resulting from assigning references to inner function objects to the public properties of a constructed object from within its constructor. But if the methods of an object are not taking advantage of the closure that they will form within the constructor the creation of multiple function objects for each object instantiation will make the instantiation process slower and more resources will be consumed to accommodate the extra function objects created.

In that case it would be more efficient to create the function object once and assign references to them to the corresponding properties of the constructor's `prototype` so they may be shared by all of the objects created with that constructor:-

CODE

```

function ExampleConst(param){
    /* Assign the constructor's parameter to a property of the object:- */
    this.publicProp = param;
}
/* Create methods for the objects by evaluating function expressions
   and assigning references to the resulting function objects to the
   properties of the constructor's prototype:- */
ExampleConst.prototype.method1 = function(){
    ... // method body.
};
ExampleConst.prototype.method2 = function(){
    ... // method body.
};
ExampleConst.prototype.method3 = function(){
    ... // method body.
};

```

The Internet Explorer Memory Leak Problem

The Internet Explorer web browser (verified on versions 4 to 6 (6 is current at the time of writing)) has a fault in its garbage collection system that prevents it from garbage collecting ECMAScript and some host objects if those host objects form part of a "circular" reference. The host objects in question are any DOM Nodes (including the document object and its descendants) and ActiveX objects. If a circular reference is formed including one or more of them, then none of the objects involved will be freed until the browser is closed down, and the memory that they consume will be unavailable to the system until that happens.

A circular reference is when two or more objects refer to each other in a way that can be followed and lead back to the starting point. Such as object 1 has a property that refers to object 2, object 2 has a property that refers to object 3 and object 3 has a property that refers back to object 1. With pure ECMAScript objects as soon as no other objects refer to any of objects 1, 2 or 3 the fact that they only refer to each other is recognised and they are made available for garbage collection. But on Internet Explorer, if any of those objects happen to be a DOM Node or ActiveX object, the garbage collection cannot see that the circular relationship between them is isolated from the rest of the system and free them. Instead they all stay in memory until the browser is closed.

Closures are extremely good at forming circular references. If a function object that forms a closure is assigned as, for example, an event handler on a DOM Node, and a reference to that Node is assigned to one of the Activation/Variable objects in its *scope chain* then a circular reference exists. `DOM_Node.onevent -> function_object.
[[scope]] -> scope_chain -> Activation_object.nodeRef -> DOM_Node`. It is very easy to do, and a bit of browsing around a site that forms such a reference in a piece of code common to each page can consume most of the systems memory (possibly all).

Care can be taken to avoid forming circular references and remedial action can be taken when they cannot otherwise be avoided, such as using IE's `onunload` event to null event handling function references. Recognising the problem and understanding closures (and their mechanism) is the key to avoiding this problem with IE.

[comp.lang.javascript FAQ notes T.O.C.](#)

Written by Richard Cornford. March 2004.

With corrections and suggestions by:-

Martin Honnen.

Yann-Erwan Perio (Yep).

Lasse Reichstein Nielsen. ([definition of closure](#))

Mike Scirocco.

Dr John Stockton.

Garrett Smith.

31. Let's Learn JavaScript Closures

Closures are a fundamental JavaScript concept that every serious programmer should know inside-out.

The Internet is packed with great explanations of "what" closures are, but few deep-dives into the "why" side of things.

I find that understanding the internals ultimately gives developers a stronger grasp of their tools, so this post will be dedicated to the nuts and bolts of *how* and *why* closures work the way they do.

Hopefully you'll walk away better equipped to take advantage of closures in your day-to-day work. Let's get started!

What is a closure?

Closures are an extremely powerful property of JavaScript (and most programming languages). As defined on [MDN](#):

*Closures are **functions** that refer to independent (**free**) **variables**. In other words, the function defined in the closure 'remembers' the environment in which it was created.*

Note: Free variables are variables that are neither locally declared nor passed as parameter.

Let's look at some examples:

Example 1:

```
function numberGenerator() {
  // Local "free" variable that ends up within the closure
  var num = 1;
  function checkNumber() {
    console.log(num);
  }
  num++;
  return checkNumber;
}

var number = numberGenerator();
number(); // 2
```

CODE

In the example above, the function **numberGenerator** creates a local "free" variable **num** (a number) and **checkNumber** (a function which prints **num** to the console). The function **checkNumber** doesn't have any local variables of its own--however, it does have access to the variables within the outer function, **numberGenerator**, because of a closure. Therefore, it can use the variable **num** declared in **numberGenerator** to successfully log it to the console even after **numberGenerator** has returned.

Example 2:

In this example we'll demonstrate that a closure contains any and all local variables that were declared inside the outer enclosing function.

```
function sayHello() {
  var say = function() { console.log(hello); }
  // Local variable that ends up within the closure
  var hello = 'Hello, world!';
  return say;
}
var sayHelloClosure = sayHello();
sayHelloClosure(); // `Hello, world!'
```

CODE

Notice how the variable **hello** is defined *after* the anonymous function--but can still access the **hello** variable. This is because the **hello** variable has already been defined in the function "scope" at the time of creation, making it

available when the anonymous function is finally executed. (Don't worry, I'll explain what "scope" means later in the post. For now, just roll with it!)

Understanding the High Level

These examples illustrated "what" closures are on a high level. The general theme is this: *we have access to variables defined in enclosing function(s) even after the enclosing function which defines these variables has returned.* Clearly, something is happening in the background that allows those variables to still be accessible long after the enclosing function that defined them has returned.

To understand how this is possible, we'll need to touch on a few related concepts--starting 3000 feet up and slowly climbing our way back down to the land of closures. Let's start with the overarching *context* within which a function is run, known as "*Execution context*".

Execution Context

Execution context is an abstract concept used by the ECMAScript specification to track the runtime evaluation of code. This can be the global context in which your code is first executed or when the flow of execution enters a function body.

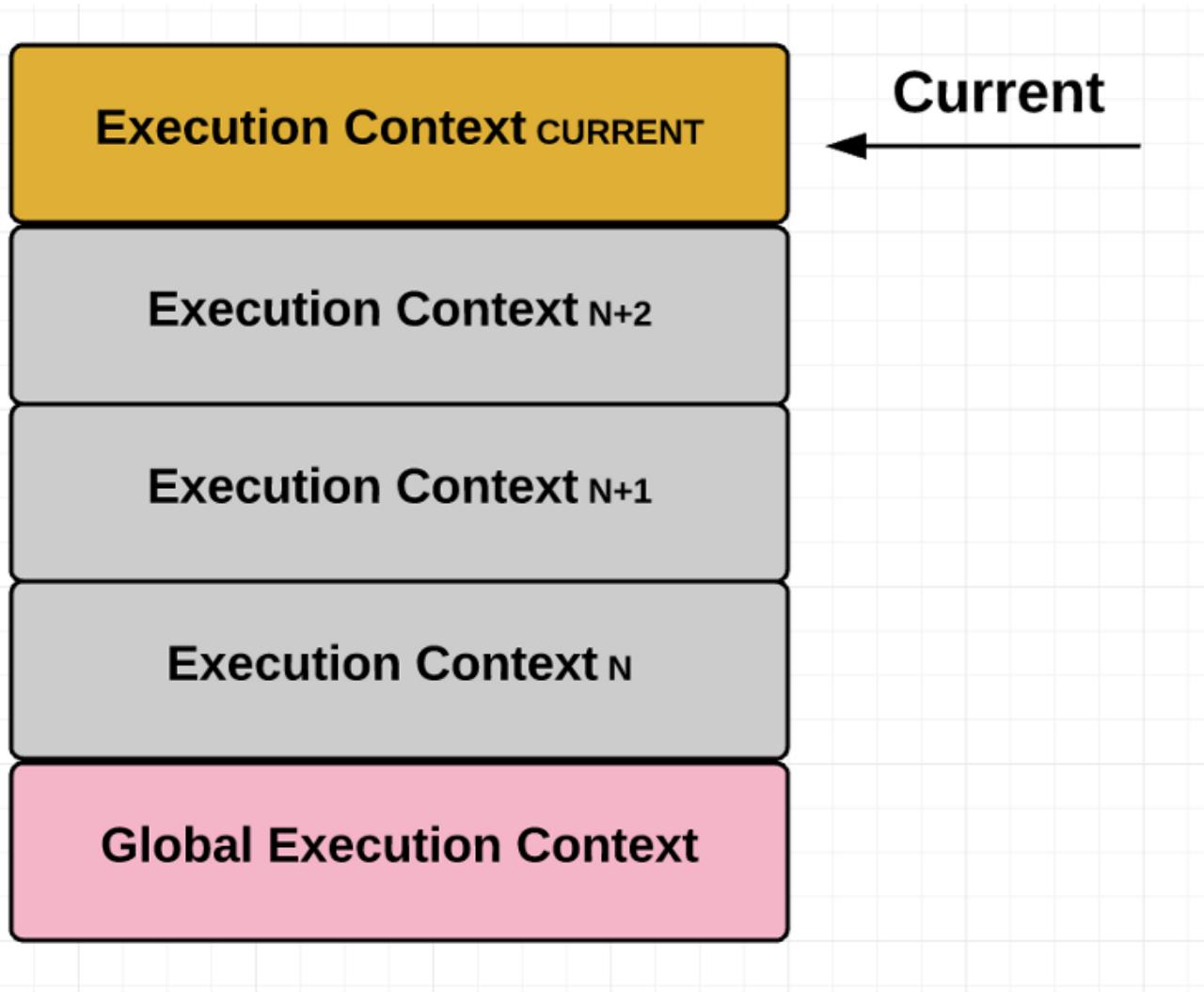
Global Execution Context

```
1  var x = 10;  
2  
3  function foo() {  
4      Execution Context (foo)  
5      var y = 20; // free variable  
6  
7      function bar() {  
8          Execution Context (bar)  
9          var z = 15; // free variable  
10         var output = x + y + z;  
11         return output;  
12     }  
13  
14     return bar;  
15 }
```

At any point in time, there can only be one execution context running. That's why JavaScript is "single threaded,"

meaning only one command can be processed at a time. Typically, browsers maintain this execution context using a "stack." A stack is a Last In First Out (LIFO) data structure, meaning the last thing that you pushed onto the stack is the first thing that gets popped off it. (This is because we can only insert or delete elements at the top of the stack.) The current or "running" execution context is always the top item in the stack. It gets popped off the top when the code in the running execution context has been completely evaluated, allowing the next top item to take over as running execution context.

Moreover, just because an execution context is running doesn't mean that it has to finish running before a different execution context can run. There are times when the running execution context is suspended and a different execution context becomes the running execution context. The suspended execution context might then at a later point pick back up where it left off. Anytime one execution context is replaced by another like this, a new execution context is created and pushed onto the stack, becoming the current execution context.



For a practical example of this concept in action in the browser, see the example below:

CODE

```

var x = 10;
function foo(a) {
    var b = 20;

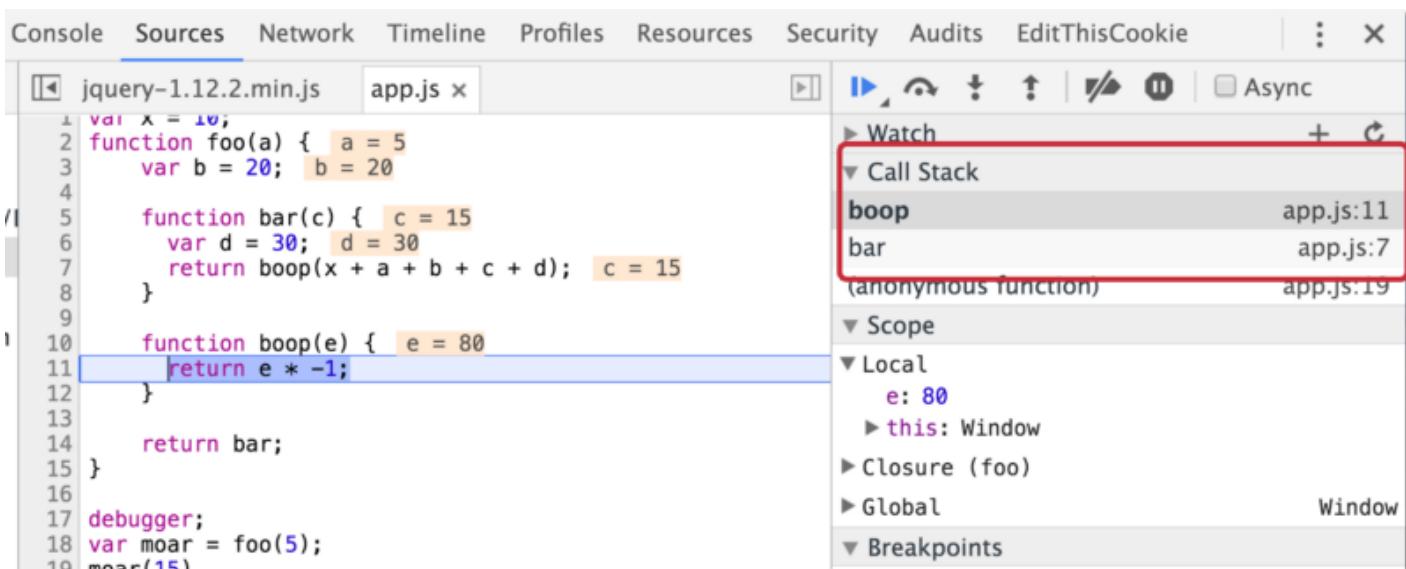
    function bar(c) {
        var d = 30;
        return boop(x + a + b + c + d);
    }

    function boop(e) {
        return e * -1;
    }

    return bar;
}

var moar = foo(5); // Closure
/*
The function below executes the function bar which was returned
when we executed the function foo in the line above. The function bar
invokes boop, at which point bar gets suspended and boop gets pushed
onto the top of the call stack (see the screenshot below)
*/
moar(15);

```



Then when **boop** returns, it gets popped off the stack and **bar** is resumed:

```

1 var x = 10;
2 function foo(a) {
3     var b = 20;
4
5     function bar(c) { c = 15
6         var d = 30; d = 30
7         return boop(x + a + b + c + d); c = 15
8     }
9
10    function boop(e) {
11        return e * -1;
12    }
13
14    return bar;
15 }
16
17 debugger;
18 var moar = foo(5);
19 moar(15)

```

When we have a bunch of execution contexts running one after another--often being paused in the middle and then later resumed--we need some way to keep track of state so we can manage the order and execution of these contexts. And that is in fact the case; as per the ECMAScript spec, each execution context has various state components that are used to keep track of the progress of the code in each context has made. These include:

- **Code evaluation state:** Any state needed to perform, suspend, and resume evaluation of the code associated with this execution context
- **Function:** The function object which the execution context is evaluating (or null if the context being evaluated is a *script or module*)
- **Realm:** A set of internal objects, an ECMAScript global environment, all of the ECMAScript code that is loaded within the scope of that global environment, and other associated state and resources
- **Lexical Environment:** Used to resolve identifier references made by code within this execution context.
- **Variable Environment:** Lexical Environment whose EnvironmentRecord holds bindings created by VariableStatements within this execution context.

If this sounds too confusing to you, don't worry. Of all these variables, the Lexical Environment variable is the one that's most interesting to us because it explicitly states that it resolves "*identifier references*" made by code within this execution context. You can think of "*identifiers*" as variables. Since our original goal was to figure out how it's possible for us to magically access variables even after a function (or "context") has returned, Lexical Environment looks like something we should dig into!

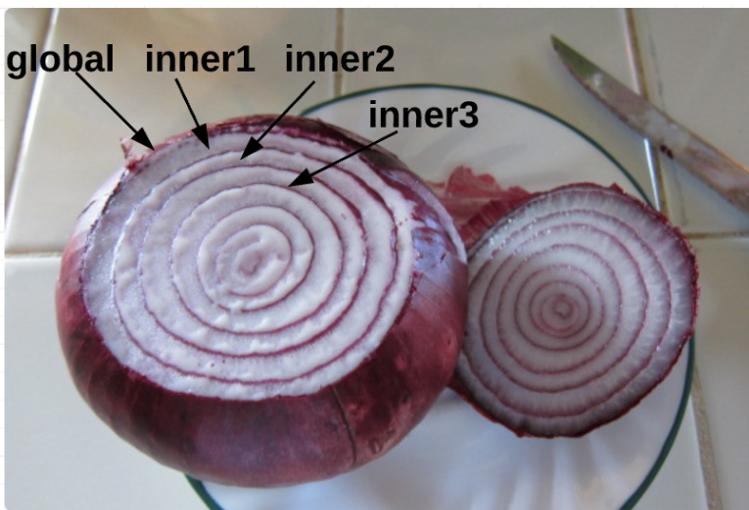
Note: Technically, both *Variable Environment* and *Lexical Environment* are used to implement closures. But for simplicity's sake, we'll generalize it to an "*Environment*". For a detailed explanation on the difference between *Lexical* and *Variable Environment*, see Dr. Alex Rauschmayer's excellent [article](#).

Lexical Environment

By definition: A *Lexical Environment* is a specification type used to define the association of *Identifiers* to specific *variables* and *functions* based upon the lexical nesting structure of ECMAScript code. A *Lexical Environment* consists of an *Environment Record* and a possibly null reference to an outer *Lexical Environment*. Usually a *Lexical Environment* is associated with some specific syntactic structure of ECMAScript code such as a *FunctionDeclaration*, a *BlockStatement*, or a *Catch clause* of a *TryStatement* and a new *Lexical Environment* is created each time such code is evaluated.--[ECMAScript-262/6.0](#)

Let's break this down.

- **"Used to define the association of Identifiers"**: The purpose of a Lexical Environment is to manage data (i.e. identifiers) within code. In other words, it gives meaning to identifiers. For instance, if we had a line of code "console.log(x / 10)", it's meaningless to have a variable (or "identifier") **x** without something that provides meaning for that variable. The Lexical Environments provides this meaning (or "association") via its Environment Record (see below).
- **"Lexical Environment consists of an Environment Record"**: An Environment Record is a fancy way to say that it keeps a record of all identifiers and their bindings that exist within a Lexical Environment. Every Lexical Environment has its own Environment Record.
- **"Lexical nesting structure"**: This is the interesting part, which is basically saying that an inner environment references the outer environment that surrounds it, and that this outer environment can have its own outer environment as well. As a result, an environment can serve as the outer environment for more than one inner environment. The global environment is the only Lexical environment that does not have an outer environment. The language here is tricky, so let's use a metaphor and think of lexical environments like layers of an onion: the global environment is the outermost layer of the onion; every subsequent layer below is nested within.



Abstractly, the environment looks like this in pseudocode:

```

CODE
LexicalEnvironment = {
  EnvironmentRecord: {
    // Identifier bindings go here
  },
  // Reference to the outer environment
  outer: < >
};

```

- **"A new Lexical Environment is created each time such code is evaluated"**: Each time an enclosing outer function is called, a new lexical environment is created. This is important--we'll come back to this point again at the end. (Side note: a function is not the only way to create a Lexical Environment. Others include a block statement or a catch clause. For simplicity's sake, I'll focus on environment created by functions throughout this post)

In short, every execution context has a Lexical Environment. This Lexical environments holds variables and their associated values, and also has a reference to its outer environment. The Lexical Environment can be the global environment, a module environment (which contains the bindings for the top level declarations of a Module), or a function environment (environment created due to the invocation of a function).

Scope Chain

Based on the above definition, we know that an environment has access to its parent's environment, and its parent environment has access to its parent environment, and so on. This set of identifiers that each environment has access to is called "scope." We can nest scopes into a hierarchical chain of environments known as the "scope chain".

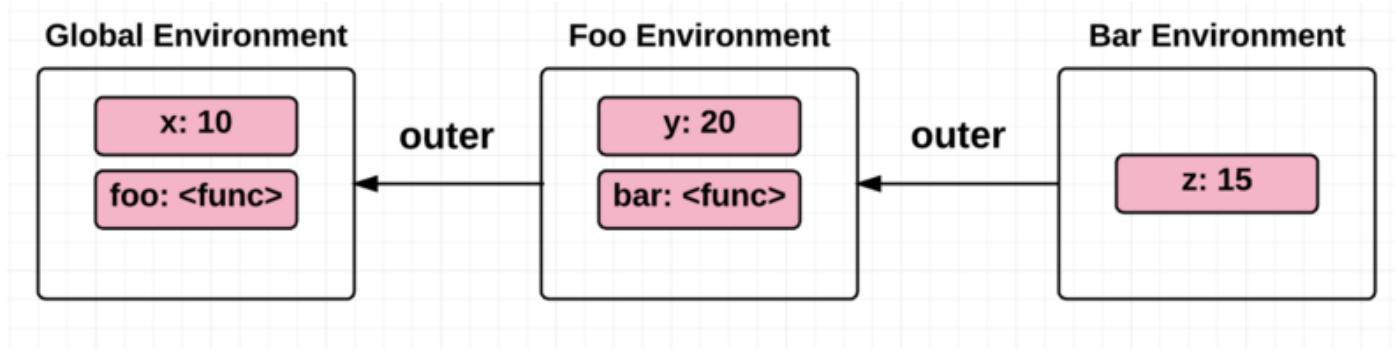
Let's look at an example of this nesting structure:

```
var x = 10;

function foo() {
  var y = 20; // free variable
  function bar() {
    var z = 15; // free variable
    return x + y + z;
  }
  return bar;
}
```

CODE

As you can see, **bar** is nested within **foo**. To help you visualize the nesting, see the diagram below:



We'll revisit this example later in the post.

This scope chain, or chain of environments associated with a function, is saved to the function object at the time of its creation. In other words, it's defined statically by location within the source code. (This is also known as "lexical scoping".)

Let's take a quick detour to understand the difference between "dynamic scope" and "static scope", which will help clarify why static scope (or lexical scope) is necessary in order to have closures.

Detour: Dynamic Scope vs. Static Scope

Dynamic scoped languages have "stack-based implementations", meaning that the local variables and arguments of functions are stored on a stack. Therefore, the runtime state of the program stack determines what variable you are referring to.

On the other hand, static scope is when the variables referenced in a context are recorded at the *time of creation*. In other words, the structure of the program source code determines what variables you are referring to.

At this point, you might be wondering how dynamic scope and static scope are different. Here's two examples to help

illustrate:

Example 1:

CODE

```
var x = 10;

function foo() {
    var y = x + 5;
    return y;
}

function bar() {
    var x = 2;
    return foo();
}

function main() {
    foo(); // Static scope: 15; Dynamic scope: 15
    bar(); // Static scope: 15; Dynamic scope: 7
    return 0;
}
```

We see above that the static scope and dynamic scope return different values when the function **bar** is invoked.

With static scope, the return value of **bar** is based on the value of **x** at the time of **foo**'s creation. This is because of the static and lexical structure of the source code, which results in **x** being 10 and the result being 15.

Dynamic scope, on the other hand, gives us a stack of variable definitions tracked at runtime--such that which **x** we use depends on what exactly is in scope and has been defined dynamically at runtime. Running the function **bar** pushes **x = 2** onto the top of the stack, making **foo** return 7.

Example 2:

CODE

```
var myVar = 100;

function foo() {
    console.log(myVar);
}

foo(); // Static scope: 100; Dynamic scope: 100

(function () {
    var myVar = 50;
    foo(); // Static scope: 100; Dynamic scope: 50
})();

// Higher-order function
(function (arg) {
    var myVar = 1500;
    arg(); // Static scope: 100; Dynamic scope: 1500
})(foo);
```

Similarly, in the dynamic scope example above the variable **myVar** is resolved using the value of **myVar** at the place where the function is called. Static scope, on the other hand, resolves **myVar** to the variable that was saved in the

scope of the two IIFE functions *at creation*.

As you can see, dynamic scope often leads to some ambiguity. It's not exactly made clear which scope the free variable will be resolved from.

Closures

Some of that may strike you as off-topic, but we've actually covered everything we need to know to understand closures:

Every function has an execution context, which comprises of an environment that gives meaning to the variables within that function and a reference to its parent's environment. A reference to the parent's environment makes all variables in the parent scope available for all inner functions, regardless of whether the inner function(s) are invoked outside or inside the scope in which they were created.

So, it appears as if the function "remembers" this environment (or scope) because the function literally has a reference to the environment (and the variables defined in that environment)!

Coming back to the nested structure example:

```
var x = 10;

function foo() {
  var y = 20; // free variable
  function bar() {
    var z = 15; // free variable
    return x + y + z;
  }
  return bar;
}

var test = foo();

test(); // 45
```

CODE

Based on our understanding of how environments work, we can say that the environment definitions for the above example look something like this (note, this is purely pseudocode):

CODE

```

GlobalEnvironment = {
  EnvironmentRecord: {
    // built-in identifiers
    Array: '',
    Object: '',
    // etc..
  }

  // custom identifiers
  x: 10
},
outer: null
};

fooEnvironment = {
  EnvironmentRecord: {
    y: 20,
    bar: ''
  }
  outer: GlobalEnvironment
};

barEnvironment = {
  EnvironmentRecord: {
    z: 15
  }
  outer: fooEnvironment
};

```

When we invoke the function **test**, we get 45, which is the return value from invoking the function **bar** (because **foo** returned **bar**). **bar** has access to the free variable **y** even after the function **foo** has returned because **bar** has a reference to **y** through its outer environment, which is **foo**'s environment! **bar** also has access to the global variable **x** because **foo**'s environment has access to the global environment. This is called "*scope-chain lookup*."

Returning to our discussion of dynamic scope vs static scope: for closures to be implemented, we can't use dynamic scoping via a dynamic stack to store our variables. The reason is because it would mean that when a function returns, the variables would be popped off the stack and no longer available--which contradicts our initial definition of a closure. What happens instead is that the closure data of the parent context is saved in what's known as the "heap," which allows for the data to persist after the function call that made them returns (i.e. even after the execution context is popped off the execution call stack).

Make sense? Good! Now that we understand the internals on an abstract level, let's look at a couple more examples:

Example 1:

One canonical example/mistake is when there's a for-loop and we try to associate the counter variable in the for-loop with some function in the for-loop:

CODE

```

var result = [];

for (var i = 0; i < 5; i++) {
  result[i] = function () {
    console.log(i);
  };
}

result[0](); // 5, expected 0
result[1](); // 5, expected 1
result[2](); // 5, expected 2
result[3](); // 5, expected 3
result[4](); // 5, expected 4

```

Going back to what we just learned, it becomes super easy to spot the mistake here! Abstractly, here's what the environment looks like this by the time the for-loop exits:

CODE

```

environment: {
  EnvironmentRecord: {
    result: [...],
    i: 5
  },
  outer: null,
}

```

The incorrect assumption here was that the scope is different for all five functions within the result array. Instead, what's actually happening is that the environment (or context/scope) is the same for all five functions within the result array. Therefore, every time the variable **i** is incremented, it updates scope--which is shared by all the functions. That's why any of the 5 functions trying to access **i** returns 5 (**i** is equal to 5 when the for-loop exits).

One way to fix this is to create an additional enclosing context for each function so that they each get their own execution context/scope:

CODE

```

var result = [];

for (var i = 0; i < 5; i++) {
  result[i] = (function inner(x) {
    // additional enclosing context
    return function() {
      console.log(x);
    }
  })(i);
}

result[0](); // 0, expected 0
result[1](); // 1, expected 1
result[2](); // 2, expected 2
result[3](); // 3, expected 3
result[4](); // 4, expected 4

```

Yay! That fixed it :)

Another, rather clever approach is to use **let** instead of **var**, since **let** is block-scoped and so a new identifier binding

is created for each iteration in the for-loop:

```
var result = [];

for (let i = 0; i < 5; i++) {
  result[i] = function () {
    console.log(i);
  };
}

result[0](); // 0, expected 0
result[1](); // 1, expected 1
result[2](); // 2, expected 2
result[3](); // 3, expected 3
result[4](); // 4, expected 4
```

CODE

Tada! :)

Example 2:

In this example, we'll show how each *call* to a function creates a new separate closure:

```
function iCantThinkOfAName(num, obj) {
    // This array variable, along with the 2 parameters passed in,
    // are 'captured' by the nested function 'doSomething'
    var array = [1, 2, 3];
    function doSomething(i) {
        num += i;
        array.push(num);
        console.log('num: ' + num);
        console.log('array: ' + array);
        console.log('obj.value: ' + obj.value);
    }

    return doSomething;
}

var referenceObject = { value: 10 };
var foo = iCantThinkOfAName(2, referenceObject); // closure #1
var bar = iCantThinkOfAName(6, referenceObject); // closure #2

foo(2);
/*
  num: 4
  array: 1,2,3,4
  obj.value: 10
*/

bar(2);
/*
  num: 8
  array: 1,2,3,8
  obj.value: 10
*/

referenceObject.value++;

foo(4);
/*
  num: 8
  array: 1,2,3,4,8
  obj.value: 11
*/

bar(4);
/*
  num: 12
  array: 1,2,3,8,12
  obj.value: 11
*/
```

In this example, we can see that each call to the function **iCantThinkOfAName** creates a new closure, namely **foo** and **bar**. Subsequent invocations to either closure functions updates the closure variables within that closure itself, demonstrating that the variables in each closure continue to be usable by **iCantThinkOfAName**'s **doSomething** function long after **iCantThinkOfAName** returns.

Example 3:

CODE

```
function mysteriousCalculator(a, b) {
    var mysteriousVariable = 3;
    return {
        add: function() {
            var result = a + b + mysteriousVariable;
            return toFixedTwoPlaces(result);
        },
        subtract: function() {
            var result = a - b - mysteriousVariable;
            return toFixedTwoPlaces(result);
        }
    }
}

function toFixedTwoPlaces(value) {
    return value.toFixed(2);
}

var myCalculator = mysteriousCalculator(10.01, 2.01);
myCalculator.add() // 15.02
myCalculator.subtract() // 5.00
```

What we can observe is that **mysteriousCalculator** is in the global scope, and it returns two functions. Abstractly, the environments for the example above look like this:

CODE

```
GlobalEnvironment = {
  EnvironmentRecord: {
    // built-in identifiers
    Array: '',
    Object: '',
    // etc...
  },
  // custom identifiers
  mysteriousCalculator: '',
 toFixedTwoPlaces: '',
},
outer: null,
};

mysteriousCalculatorEnvironment = {
  EnvironmentRecord: {
    a: 10.01,
    b: 2.01,
    mysteriousVariable: 3,
  }
  outer: GlobalEnvironment,
};

addEnvironment = {
  EnvironmentRecord: {
    result: 15.02
  }
  outer: mysteriousCalculatorEnvironment,
};

subtractEnvironment = {
  EnvironmentRecord: {
    result: 5.00
  }
  outer: mysteriousCalculatorEnvironment,
};
```

Because our **add** and **subtract** functions have a reference to the **mysteriousCalculator** function environment, they're able to make use of the variables in that environment to calculate the result.

Example 4:

One final example to demonstrate an important use of closures: to maintain a private reference to a variable in the outer scope.

```

function secretPassword() {
  var password = 'xh38sk';
  return {
    guessPassword: function(guess) {
      if (guess === password) {
        return true;
      } else {
        return false;
      }
    }
  }
}

var passwordGame = secretPassword();
passwordGame.guessPassword('heyisthisit?'); // false
passwordGame.guessPassword('xh38sk'); // true

```

This is a very powerful technique--it gives the closure function **guessPassword** exclusive access to the **password** variable, while making it impossible to access the **password** from the outside.

Tl;dr

- Execution context is an abstract concept used by the ECMAScript specification to track the runtime evaluation of code. At any point in time, there can only be one execution context that is executing code.
- Every execution context has a Lexical Environment. This Lexical environments holds identifier bindings (i.e. variables and their associated values), and also has a reference to its outer environment.
- The set of identifiers that each environment has access to is called "scope." We can nest these scopes into a hierarchical chain of environments, known as the "scope chain".
- Every function has an execution context, which comprises of a Lexical Environment that gives meaning to the variables within that function and a reference to its parent's environment. And so it appears as if the function "remembers" this environment (or scope) because the function literally has a reference to this environment. This is a closure.
- A closure is created every time an enclosing outer function is called. In other words, the inner function does not need to return for a closure to be created.
- The scope of a closure in JavaScript is lexical, meaning it's defined statically by its location within the source code.
- Closures have many practical use cases. One important use case is to maintain a private reference to a variable in the outer scope.

Clos(ure)ing remarks

I hope this post was helpful and gave you a mental model for how closures are implemented in JavaScript. As you can see, understanding the nuts and bolts of how they work makes it much easier to spot closures--not to mention saving a lot of headache when it's time to debug.

PS: I'm human and make mistakes--so if you find any mistakes I'd love for you to let me know!

Further Reading

For the sake of brevity I left out a few topics that might be interesting to some readers. Here are some links that I

wanted to share:

- **What's the VariableEnvironment within an execution context?** Dr. Axel Rauschmayer does a phenomenal job explaining it so I'll leave you off with a link to his blog post: <http://www.2ality.com/2011/04/ecmascript-5-spec-lexicalenvironment.html>
- **What are the different types of Environment Records?** Read the spec here: <http://www.ecma-international.org/ecma-262/6.0/#sec-environment-records>
- **Excellent article by MDN on closures:** <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures>
- Others? Please suggest and I'll add them!

32. ES6 arrow functions, syntax and lexical scoping

ES2015 (ES6) introduces a really nice feature that punches above its weight in terms of simplicity to integrate versus time saving and feature output. This feature is the arrow function.

Before we dive into the features of the arrow function and what it actually does for us, let's understand what an arrow function is *not*. It's not a replacement for the `function` keyword, at all. This means you can't do a find and replace on every single `function` keyword and everything works perfectly, because it likely won't.

If you're competent with the way [JavaScript scope](#) works, and have a great understanding of lexical scope, the `this` keyword and Prototype methods such as `.call()`, `.apply()` and `.bind()`, then you're in good hands to continue reading.

Syntax

Let's look at what the arrow function's construct is from [MDN](#):

```
// example 1
([param] [, param]) => {
  statements
}

// example 2
param => expression
```

CODE

The "normal JavaScript" (ES5) equivalents to help transition:

```
// example 1
function ([param] [, param]) {
  statements
}

// example 2
function (param) {
  return expression
}
```

CODE

The ES6 and ES5 differences in `example 1` are that the `function` keyword is omitted, and `=>` now exists *after* the arguments. In `example 2`, our function has been reduced to one line, this is great for single line function

expressions that get return `d.

Hint: arrows are anonymous

Arrow functions are always anonymous, which means we can't do this with ES6:

```
// ES5
function doSomething() {
    //...
}
```

CODE

Instead of this, we could assign our anonymous arrow function it to a variable (using var here instead of let as ES6 block scoping is another topic):

```
// ES6
var doSomething = () => {
    //...
}
```

CODE

Let's look at the syntaxes a little further and then the functionality differences when using arrow functions.

Syntax: single line expressions

We touched briefly above on single line expressions, let's look at a great use case for them.

Let's take some junky ES5 example that iterates over an Array using `Array.prototype.map`:

```
var numbers = [1,2,3,4,5];
var timesTwo = numbers.map(function (number) {
    return number * 2;
});
console.log(timesTwo); // [2, 4, 6, 8, 10]
```

CODE

We can reduce this down to a single line with an arrow function, which saves us a lot of typing and can actually enhance readability in my opinion as this piece of code has one clear role:

```
var numbers = [1,2,3,4,5];
var timesTwo = numbers.map((number) => number * 2);
console.log(timesTwo); // [2, 4, 6, 8, 10]
```

CODE

Syntax: single argument functions

Arrow functions also give us a small "sugar" syntax that allows us to remove parenthesis when only using a single argument in a function.

Taking the last piece of code for example we had this:

```
numbers.map((number) => number * 2);
```

CODE

When we could remove the parens from `(number)` to leave us with this:

```
numbers.map(number => number * 2);
```

CODE

This is great and a little clearer initially, but as we all know applications grow and code scales, and to save us headaches (be it forgetting syntaxes or lesser experienced developers "not knowing" to add parens back with more than one argument), I'd recommend always using the parens out of habit, even for single args:

```
// we still rock with ES6
numbers.map((number) => number * 2);
```

CODE

Functionality: lexical scoping "this"

Now we're past the sugar syntax excitement, we can dig into the benefits of the arrow function and its implications on execution context.

Typically if we're writing ES5, we'll use something like `Function.prototype.bind` to grab the `this` value from another scope to change a function's execution context. This will mainly be used in callbacks inside a different scope.

In Angular, I adopt the `controllerAs` syntax which allows me to use `this` inside the Controller to refer to itself (so here's an example). Inside a function the `this` value may change, so I could have a few options, use `that = this` or `.bind`:

```
function FooCtrl (FooService) {
  this.foo = 'Hello';
  FooService
    .doSomething(function (response) {
      this.foo = response;
    });
}
```

CODE

The `this.foo = response;` won't work correctly as it's been executed in a different context. To change this we could use `.bind(this)` to give our desired effect:

```
function FooCtrl (FooService) {
  this.foo = 'Hello';
  FooService
    .doSomething(function (response) {
      this.foo = response;
    }.bind(this));
}
```

CODE

Or you may be used to keeping a top level `this` reference, which can make more sense when dealing with many nested contexts, we don't want a gross tree of `.bind(this)`, `.bind(this)`, `.bind(this)` and a tonne of wasted

time binding those new functions (`.bind` is very slow). So we could look at `that = this` to save the day:

```
function FooCtrl (FooService) {
  var that = this;
  that.foo = 'Hello';
  FooService
    .doSomething(function (response) {
      that.foo = response;
    });
}
```

CODE

With arrow functions, we have a better option, which allows us to "inherit" the scope we're in if needed. Which means if we changed our initial example to the following, the `this` value would be bound correctly:

```
function FooCtrl (FooService) {
  this.foo = 'Hello';
  FooService
    .doSomething((response) => { // woo, pretty
      this.foo = response;
    });
}
```

CODE

We could then refactor some more into a nice single line expression, push to git and head home for the day:

```
function FooCtrl (FooService) {
  this.foo = 'Hello';
  FooService
    .doSomething((response) => this.foo = response);
}
```

CODE

The interesting thing to note is that the `this` value (internally) is not *actually* bound to the arrow function. Normal functions in JavaScript bind their own `this` value, however the `this` value used in arrow functions is actually fetched lexically from the scope it sits inside. It has no `this`, so when you use `this` you're talking to the outer scope.

33. Replacing switch statements with Object literals

In many programming languages, the `switch` statement exists - but should it any longer? If you're a JavaScript programmer, you're often jumping in and out of Objects, creating, instantiating and manipulating them. Objects are really flexible, they're at the heart of pretty much everything in JavaScript, and using them instead of the `switch` statement has been something I've been doing lately.

What is the `switch` statement?

If you've not used `switch` before or a little unsure what it does, let's walk through it. What `switch` does it take input and provide an output, such as code being run.

Let's look at a usual `switch` statement:

```

var type = 'coke';
var drink;
switch(type) {
  case 'coke':
    drink = 'Coke';
    break;
  case 'pepsi':
    drink = 'Pepsi';
    break;
  default:
    drink = 'Unknown drink!';
}
console.log(drink); // 'Coke'

```

CODE

It's similar to `if` and `else` statements, but it should evaluate a single value - inside the `switch` we use a `case` to evaluate against each value.

When you start seeing lots of `else if` statements, something is likely wrong and generally you shoud use something like `switch` as it's more suited for the purpose and intention. Here's some `else if` abuse:

```

function getDrink (type) {
  if (type === 'coke') {
    type = 'Coke';
  } else if (type === 'pepsi') {
    type = 'Pepsi';
  } else if (type === 'mountain dew') {
    type = 'Mountain Dew';
  } else if (type === 'lemonade') {
    type = 'Lemonade';
  } else if (type === 'fanta') {
    type = 'Fanta';
  } else {
    // acts as our "default"
    type = 'Unknown drink!';
  }
  return 'You\'ve picked a ' + type;
}

```

CODE

This implementation is too loose, there is room for error, plus it's a very verbose syntax to keep repeating yourself. There's also the room for hacks as you can evaluate multiple expressions inside each `else if`, such as `else if (type === 'coke' && somethingElse !== 'apples')`. The `switch` was the best tool for the job, albeit you need to keep adding `break;` statements to prevent cases falling through, one of its many issues.

Problems with switch

There are multiple issues with `switch`, from its procedural control flow to its non-standard-looking way it handles code blocks, the rest of JavaScript uses curly braces yet `switch` does not. Syntactically, it's not one of JavaScript's best, nor is its design. We're forced to manually add `break;` statements within each `case`, which can lead to difficult debugging and nested errors further down the case should we forget! Douglas Crockford has written and spoken about it numerous times, his recommendations are to treat it with caution.

We often use Object lookups for things in JavaScript, often for things we would never contemplate using `switch` for

- so why not use an Object literal to replace `switch`? Objects are much more flexible, have better readability and maintainability and we don't need to manually `break;` each "case". They're a lot friendlier on new JavaScript developers as well, as they're standard Objects.

As the number of "cases" increases, the performance of the object (hash table) gets better than the average cost of the switch (the order of the cases matter). The object approach is a hash table lookup, and the switch has to evaluate each case until it hits a match and a break.

Object Literal lookups

We use Object's all the time, either as constructors or literals. Often, we use them for Object lookup purposes, to get values from Object properties.

Let's setup a simple Object literal that returns a `String` value only.

```
function getDrink (type) {
    var drinks = {
        'coke': 'Coke',
        'pepsi': 'Pepsi',
        'lemonade': 'Lemonade',
        'default': 'Default item'
    };
    return 'The drink I chose was ' + (drinks[type] || drinks['default']);
}

var drink = getDrink('coke');
// The drink I chose was Coke
console.log(drink);
```

CODE

We've saved a few lines of code from the switch, and to me the data is a lot cleaner in presentation. We can even simplify it further, without a default case:

```
function getDrink (type) {
    return 'The drink I chose was ' + {
        'coke': 'Coke',
        'pepsi': 'Pepsi',
        'lemonade': 'Lemonade'
    }[type];
}
```

CODE

We might, however, need more complex code than a `String`, which could hang inside a function. For sake of brevity and easy to understand examples, I'll just return the above strings from the newly created function:

```

var type = 'coke';

var drinks = {
  'coke': function () {
    return 'Coke';
  },
  'pepsi': function () {
    return 'Pepsi';
  },
  'lemonade': function () {
    return 'Lemonade';
  }
};

```

CODE

The difference is we need to call the Object literal's function:

```
drinks[type]();
```

CODE

More maintainable and readable. We also don't have to worry about `break;` statements and cases falling through - it's just a plain Object.

Usually, we would put a `switch` inside a function and get a `return` value, so let's do the same here and turn an Object literal lookup it into a usable function:

```

function getDrink (type) {
  var drinks = {
    'coke': function () {
      return 'Coke';
    },
    'pepsi': function () {
      return 'Pepsi';
    },
    'lemonade': function () {
      return 'Lemonade';
    }
  };
  return drinks[type]();
}

// let's call it
var drink = getDrink('coke');
console.log(drink); // 'Coke'

```

CODE

Nice and easy, but this doesn't cater for a "default" case , so we can create that easily:

CODE

```

function getDrink (type) {
  var fn;
  var drinks = {
    'coke': function () {
      return 'Coke';
    },
    'pepsi': function () {
      return 'Pepsi';
    },
    'lemonade': function () {
      return 'Lemonade';
    },
    'default': function () {
      return 'Default item';
    }
  };
  // if the drinks Object contains the type
  // passed in, let's use it
  if (drinks[type]) {
    fn = drinks[type];
  } else {
    // otherwise we'll assign the default
    // also the same as drinks.default
    // it's just a little more consistent using square
    // bracket notation everywhere
    fn = drinks['default'];
  }
  return fn();
}

// called with "dr pepper"
var drink = getDrink('dr pepper');
console.log(drink); // 'Default item'

```

We could simplify the above `if` and `else` using the `or ||` operator inside an expression:

CODE

```

function getDrink (type) {
  var drinks = {
    'coke': function () {
      return 'Coke';
    },
    'pepsi': function () {
      return 'Pepsi';
    },
    'lemonade': function () {
      return 'Lemonade';
    },
    'default': function () {
      return 'Default item';
    }
  };
  return (drinks[type] || drinks['default'])();
}

```

This wraps the two Object lookups inside parenthesis `()`, treating them as an expression. The result of the expression is then invoked. If `drinks[type]` isn't found in the lookup, it'll default to `drinks['default']`, simple!

We don't have to always return inside the function either, we can change references to any variable then return it:

```
function getDrink (type) {  
    var drink;  
    var drinks = {  
        'coke': function () {  
            drink = 'Coke';  
        },  
        'pepsi': function () {  
            drink = 'Pepsi';  
        },  
        'lemonade': function () {  
            drink = 'Lemonade';  
        },  
        'default': function () {  
            drink = 'Default item';  
        }  
    };  
  
    // invoke it  
(drinks[type] || drinks['default'])();  
  
    // return a String with chosen drink  
    return 'The drink I chose was ' + drink;  
}  
  
var drink = getDrink('coke');  
// The drink I chose was Coke  
console.log(drink);
```

CODE

These are very basic solutions, and the Object literals hold a function that returns a String , in the case you only need a String , you could use a String as the key's value - some of the time the functions will contain logic, which will get returned from the function. If you're mixing functions with strings, it might be easier to use a function at all times to save looking up the type and invoking if it's a function - we don't want to attempt invoking a String .

Object Literal "fall through"

With switch cases, we can let them fall through (which means more than one case can apply to a specific piece of code):

```

var type = 'coke';
var snack;
switch(type) {
  case 'coke':
  case 'pepsi':
    snack = 'Drink';
    break;
  case 'cookies':
  case 'crisps':
    snack = 'Food';
    break;
  default:
    drink = 'Unknown type!';
}
console.log(snack); // 'Drink'

```

CODE

We let `coke` and `pepsi` "fall through" by not adding a `break` statement. Doing this for Object Literals is simple and more declarative - as well as being less prone to error. Our code suddenly becomes much more structured, readable and reusable:

```

function getSnack (type) {
  var snack;
  function isDrink () {
    return snack = 'Drink';
  }
  function isFood () {
    return snack = 'Food';
  }
  var snacks = {
    'coke': isDrink,
    'pepsi': isDrink,
    'cookies': isFood,
    'crisps': isFood,
  };
  return snacks[type]();
}

var snack = getSnack('coke');
console.log(snack); // 'Drink'

```

CODE

Summing up

Object literals are a more natural control of flow in JavaScript, `switch` is a bit old and clunky and prone to difficult debugging errors. Object's are more extensible, maintainable, and we can test them a lot better. They're also part of a design pattern and very commonly used day to day in other programming tasks. Object literals can contain functions as well as any other [Object type](#), which makes them really flexible! Each function in the literal has function scope too, so we can return the closure from the parent function we invoke (in this case `getDrink` returns the closure).

Some more interesting comments and feedback on [Reddit](#).

34. Web Components and concepts, ShadowDOM, imports, templates,

custom elements

Web Components, the future of the web, inspired from attending [Google I/O](#) I decided to pick up Web Components and actually build something. Since learning the basics around a year ago, it's changed and advanced a lot! Thought I'd write a post on it and share [my first web component](#) yesterday (built with Polymer).

Before I get into Polymer, we'll look at Web Components in this post, what it means for the web and how it completely changes things and our outlook on building for the web platform from today.

Gone are the days of actually creating HTML structures and "pages" (what're those?). The web is becoming "all about components", and those components are completely up to us thanks to Web Components.

We aren't really at a stage where we can use Web Components to it's fullest, browser support is still ongoing implementations and IE are [in consideration](#) of the entire spec (blows a single fanfare). But it's coming together, give it a few years and we'll get there. Or do we have to wait that long?...

Google are innovating in this area like no tomorrow with [Polymer.js](#), a polyfill and platform (that provides additional features such as data-binding, event callbacks and much more) for those missing pieces in modern browsers that don't fully support Web Components.

Building blocks of Web Components

Before we get over excited about this stuff though, let's actually understand what the [Web Components spec](#) really means. First thing's first, Web Components are a collection of building blocks, not a single thing. Let's look at each block to see what's up.

This will be a very high level view, otherwise this post could end up being three days worth of reading!

Templates

Templates are where we define reusable code, we even get an element for it with `<template>`. The first time you use it, don't panic - it's invisible in the visible interface output, until you view source you won't know anything is even there. It's merely a declarative element to create a new template for... anything you like.

An example of a `<template>` to populate a profile section for a user:

```
<template id="profileTemplate">
  <div class="profile">
    <img src="" class="profile__img">
    <div class="profile__name"></div>
    <div class="profile__social"></div>
  </div>
</template>
```

CODE

Sprinkle some JavaScript to populate it, and append it to the `<body>`:

```
var template = document.querySelector('#profileTemplate');
template.querySelector('.profile_img').src = 'toddmotto.jpg';
template.querySelector('.profile_name').textContent = 'Todd Motto';
template.querySelector('.profile_social').textContent = 'Follow me on Twitter';
document.body.appendChild(template);
```

CODE

You'll notice that this is just JavaScript, no new APIs or anything confusing. Nice! For me, a `<template>` is useless without its good buddy *Custom Elements*. We need this to do something useful with the tech, things are all global and disgusting as of now.

Custom Elements

Custom Elements allow us to define (you guessed it), our own element. This can be anything, but before you go crazy, your elements must have a dash, presumably to avoid any potential naming clashes with future HTML implementations - I think that's a good idea as well.

So, with our custom element, how do we do it? Simple really, we get the `<element>` element, so meta. Well, we *had* the `<element>` element. Read on, as `<element>` was recently deprecated and thus needs a JavaScript implementation, but this is the older way:

```
<element>
  <template id="profileTemplate">
    <div class="profile">
      <img src="" class="profile_img">
      <div class="profile_name"></div>
      <div class="profile_social"></div>
    </div>
  </template>
</element>
```

CODE

This example is still deprecated but worth showing. We would've given `<element>` a `name=""` attribute to define the custom element:

```
<element name="user-profile">
  <template id="profileTemplate">
    <div class="profile">
      <img src="" class="profile_img">
      <div class="profile_name"></div>
      <div class="profile_social"></div>
    </div>
  </template>
</element>

// usage
<user-profile></user-profile>
```

CODE

So what's replacing `<element>` ?

Use of `<element>` was [deprecated](#) towards the end of 2013, which means we simply use the JavaScript API instead, which I think offers more flexibility and less bloat on the markup:

```

<template id="profileTemplate">
  <div class="profile">
    <img src="" class="profile__img">
    <div class="profile__name"></div>
    <div class="profile__social"></div>
  </div>
</template>
<script>
var MyElementProto = Object.create(HTMLElement.prototype);
window.MyElement = document.registerElement('user-profile', {
  prototype: MyElementProto
  // other props
});
</script>

```

CODE

New elements must inherit from the `HTMLElement.prototype`. More on the above setup and callbacks etc [here](#), cheers [Zeno](#).

Extending and inheriting

What if we wanted to extend an existing element, such as an `<h1>` tag? There will be many cases of this, such as riding off an existing element and creating a "special" version of it, rather than a totally new element. We introduce the `{ extends: '' }` property to declare where what element we're extending. Using an extended element is simple, drop the `is=""` attribute on an existing element and it'll inherit its new extension. Pretty simple, I guess.

```

<template>
  // include random, funky things
</template>
<script>
var MyElementProto = Object.create(HTMLElement.prototype);
window.MyElement = document.registerElement('funky-heading', {
  prototype: MyElementProto,
  extends: 'h1' // extends declared here
});
</script>

<h1 is="funky-heading">
  Page title
</h1>

```

CODE

Using `extends=""` as an attribute on `<element>` was the way to do it before it was deprecated.

So what next? Enter the shadows...

ShadowDOM

ShadowDOM *is* as cool as it sounds, and provides a DOM encapsulation within DOM. Whaaat? Essentially, nested document fragments, that are shadow-y... In ShadowDOM, we're observing nested DOM trees/hierarchies. Typically in web documents, there is one DOM. Think about DOM hosting DOM, which hosts more DOM. You'll see something like this in Chrome inspector (note `#shadow-root`, which is completely encapsulated DOM):

```
<%><user-profile>
<%#shadow-root (user-agent)>
<div class="profile">
  <img src="" class="profile__img">
  <div class="profile__name"></div>
  <div class="profile__social"></div>
</div>
</user-profile>
```

CODE

There are a few different concepts with Shadow DOM, for me, it's that there is no "global" Object, no `window`, I can create a new document root. The "host" of my this new document root is either referred to as the root or host. We can create new ShadowDOM by invoking `.createShadowRoot()` on an element.

ShadowDOM already exists in the wild today though, as soon as you use `<input type=range>` in the browser, we get a nice input with a slider, guess what - that's ShadowDOM! It's a nested structure that's hidden inside our DOM tree. Now we can create it ourselves, this opens up an entire plethora of opportunities.

Why is this *really* cool?

ShadowDOM gives us *true* encapsulation, with scoped components. CSS is *scoped* (wow, although we tried this with `<style scoped>` but Blink have since removed it from the core to make way for Web Components). This means any CSS we write inside ShadowDOM only affects the DOM of that particular ShadowDOM!

```
<template>
  <style>
    :host {
      border: 1px solid red;
    }
  </style>
  // stuff
</template>
<script>
var MyElementProto = Object.create(HTMLElement.prototype);
window.MyElement = document.registerElement('funky-heading', {
  prototype: MyElementProto,
  extends: 'h1'
});
</script>
```

CODE

This also means each document can also have a unique `id`, and we can avoid crazy naming conventions for scaling our apps/websites (a minor bonus).

We can also put scripts in there too and talk to the current element:

```

<template>
  <style>
    :host {
      border: 1px solid red;
    }
  </style>
  // stuff
</template>
<script>
(function () {
  // stuff with JS...
})();

```



```

var MyElementProto = Object.create(HTMLElement.prototype);
window.MyElement = document.registerElement('funky-heading', {
  prototype: MyElementProto,
  extends: 'h1'
});
</script>

```

CODE

JavaScript events that are fired, also are encapsulated to the ShadowDOM tree.

How can I see this ShadowDOM?

In true shadow style, you need to enable it via the `Show user agent ShadowDOM` checkbox inside Chrome Dev Tools. Upon inspecting element, you can see the nested DOM trees. Chrome also allows you to edit the CSS, which is even more awesome.

HTML Imports

Importing dependencies into our language of choice comes in many shapes and sizes. For CSS, we have `@import`, for JavaScript in ES6 modules we have `import {Module} from './somewhere';`, and *finally*, HTML. We can import HTML components at the top of our document to define which ones we need to use in our app:

```

<link rel="import" href="user-profile.html">

<!--
  <user-profile> now available, ooo yeah!
-->

```

CODE

This is massive! Encapsulated components all in one file. Out of the box and working. Let's take Google Maps API for example, we need to include the Maps API v3, import the 'Hello world' code and then style a basic map. Wouldn't it be great to just do this:

```

<link rel="import" href="google-map.html">

<!-- boom! -->
<google-map></google-map>

```

CODE

All encapsulated, tested, I could just pass in values via attributes and job done:

```
<google-map coords="37.2350, 115.8111"></google-map>
```

CODE

Decorators

Decorators are part of Web Components, but actually have *no spec* (according to the [spec](#)). Apparently they might look something like this, with their intention to enhance or override the presentation of an existing element. So ignore them for now, I guess (*see Addy's comment on Decorators, they might even disappear from Web Components entirely*).

```
<decorator id="details-open">
  <template>
    <a id="summary">
      &blacktriangledown;
      <content select="summary"></content>
    </a>
    <content></content>
  </template>
</decorator>
```

CODE

Can I get started now? Enter Polymer.js

Yes. Web Components are going to be a little while before fully landing and being the next generation of the web, but they're certainly making fast traction. We can get to grips with the technology and concepts now and start building using a framework such as Polymer - which polyfills things for modern browsers to let us use Web Components now.

An example of using Polymer to define an element. Here, we simply swap out (was) `<element>` for `<polymer-element>` and that's it.

```
<polymer-element name="my-element">
  <template>
    // take it away!
  </template>
  <script>
    Polymer('my-element', {});
  </script>
</polymer-element>

<my-element></my-element>
```

CODE

Polymer has some really sweet features, such as data-binding (the Angular dev inside me loves this) and a tonne of simple events built in, from new instances of the element, to creation and injection callbacks that make it really simple to creating new elements.

Takeaways

This post isn't meant to be a full tutorial - these components are vast and best explored individually, but I wanted to provide an eye opener on the rapidly approaching technology that is Web Components.

For me, one of the biggest selling points of Web Components is to prevent the inclusion of a huge JavaScript file, a huge CSS file and a tonne of HTML to make our website or app. In such cases, we no doubt come back to it a few months later and have forgotten what each thing does and it's painful to get back up to speed again. We don't forget what the `<google-map>` element does though, or the `<fluid-vids>` element, they're declarative and self-explanatory, we know exactly where their logic is, and where the styles are.

The biggest win? Logic is *contained*. We've all struggled managing logic, markup and styles, and now the web has listened. Encapsulated behaviour and scoping, but a very powerful engine for componentising the web, anything from a navigation to google maps to an image slider.

Benefits of Web Components are very clear, and I'm interested to see where it takes us in the next few years. This post is by far from exhaustive, but I feel we should all take a dive into what the future of the web will bring us, we'll be there sooner than you think!

Links to definitely keep an eye on (any others feel free to share below):

- WebComponents.org
- Polymer
- customelements.io
- HTML5 Rocks
- Eric Bidelman, Google I/O 2013 [Tectonic shift for the web](#)
- Eric Bidelman, Google I/O 2014 [Polymer and Web Components](#)

35. Methods to determine if an Object has a given property

There are multiple ways to detect whether an Object has a property. You'd think it'd be as easy as `myObject.hasOwnProperty('prop');` - but no, there are a few different ways with their own problems and gotchas. Let's look at the few ways to check property existence, concepts that confuse JavaScript developers, prototype chain lookups and problems JavaScript might provide us.

Double bang !! property lookup

We've all seen it, probably in something such as Modernizr for simple feature detection, the infamous `!!` amongst our JS. Important note before we begin this one, it doesn't actually check if an Object has a property "as such", it checks the *value* of the Object property. Which means if the property value is false, or the object property doesn't even exist, they give the same `falsy` result - which can be really bad if you use it without knowing what it does and its limitations.

What does it mean?

The double-bang is a simple way to typecast something to `Boolean`. The `Boolean` will cast `true` for *truthy* values. Even things such as `undefined` and `null` (both falsy values, `!!null` is `false` when cast to `Boolean`). The *absolute key* here is that it casts *values*. I'll say it again, *values!* This is irrelevant to the shape and size of your Object. We convert *truthy* and *falsy* values to `Boolean`.

Examples

An empty `Array` is an example of a *truthy* value:

```
CODE
var a = []; // []
```

What if we want to convert it to a Boolean though? It's truthy, so we should expect true :

```
CODE
var a = !![]; // true
```

null is an example of a *falsey* value:

```
CODE
var a = null; // null
```

And the expected output of false :

```
CODE
var a = !!null; // false
```

This means that we can use it when looking up our Objects!

```
CODE
var toddObject = {
  name: 'Todd',
  cool: false
};
!!toddObject.name // true (correct result as it's a truthy value)
```

This method also looks up the Object's prototype chain to see if the property exists, which can cause unintended side effects if naming of properties is the same as a prototypes.

```
CODE
// Object.prototype.toString
!!toddObject.toString // true

// !!Array.prototype.forEach
!![]['forEach'] // true
```

Gotchas

Beware of using it for detecting your own Objects. We often create Objects and defaults such as this:

```
CODE
var toddObject = {
  name: 'Todd',
  favouriteDrink: null
};
```

If we're using the double-bang to check if an Object property exists using this method, then it's definitely a silly idea:

```

var toddObject = {
  name: 'Todd',
  favouriteDrink: null
};
if (!!toddObject.favouriteDrink) { // false
  // do something if it exists, I think...
}

```

CODE

That would be naive! The above code (to the new developer or non-double-banger) might say "*If toddObject.favouriteDrink exists, do something*". But no, because (I'll say it again...) this casts *values*, the value is `null` and falsy - even though the property exists. It's generally not a good idea in this case to use it for checking if a property exists incase it has a falsy value to begin with.

hasOwnProperty

We went as far as getting a native method for this, but it's not 100% reliable for a few reasons. Let's examine it first.

What does it mean?

Using `myObject.hasOwnProperty('prop')` is a great way of accessing the Object's keys directly, which *does not* look into the Object's prototype - hooray, this is great for specific use cases. `hasOwnProperty` returns a Boolean for us on whether a property exists.

Examples

```

var toddObject = {
  name: 'Todd',
  favouriteDrink: null
};
if (toddObject.hasOwnProperty('favouriteDrink')) { // true
  // do something if it exists, fo sho
}

```

CODE

But don't be sold on this exact implementation... read below for best practice.

Gotchas

IE messes up the `hasOwnProperty` method completely as it's painful with host Objects (host objects don't have the `hasOwnProperty` method).

JavaScript also decided not to protect the method's name, so we can infact do this:

```

var toddObject = {
  hasOwnProperty: 'hello...'
};

```

CODE

This makes it hard to fully trust it. What we can do however is access the `Object.prototype` directly to guarantee any `hasOwnProperty` calls haven't been tampered with or overridden.

Let's bulletproof the process:

```
var toddObject = {
  name: 'Todd',
  favouriteDrink: null
};
if (Object.prototype.hasOwnProperty.call(toddObject, 'favouriteDrink')) { // true
  // do something if it exists, fo sho sho!
}
```

CODE

The secret sauce here is `.call()` to change the context of `hasOwnProperty` (take that, IE) and ensure we've got the exact `hasOwnProperty` we want from the `Object.prototype`.

Obviously you'd want to wrap it inside a helper function or something to save writing out that `prototype` each time:

```
function hasProp (obj, prop) {
  return Object.prototype.hasOwnProperty.call(obj, prop);
}
if (hasProp(toddObject, 'favouriteDrink')) {}
```

CODE

‘prop’ in myObject

The `in` operator isn't so widely used as the former methods, but is probably worth using after reading this. It also returns a Boolean much like `!!myObject`, but *does not* evaluate the `value`, it evaluates the *existence* of the property!. This means if a property has a value of `false`, we get a correct reading that the property does in fact exist.

```
var toddObject = {
  name: 'Todd',
  favouriteDrink: null,
  cool: false
};
'cool' in toddObject; // true
```

CODE

The `in` operator is probably your best friend for checking the existence of a property, it's also pretty concise.

Gotchas

The `in` operator also looks up the `prototype`, which *may* cause unintended side effects:

```
// inherits Object.prototype.toString
'toString' in toddObject; // true
```

CODE

But we should know these property names and not create conflicts, right ;)

typeof

We can use `typeof` as well.

What does it mean?

The standard `typeof` operator returns a String ([not a very reliable one](#)), and we can evaluate it against something, such as `!== 'undefined'` - which indicates it exists.

```
if (typeof toddObject.name !== 'undefined') {
    // do something
}
```

CODE

It looks a little ugly, as well as being quite long to write out if we were to make multiple checks using this method. Also, `null` would fall under this check unless using `!= 'undefined'` (single `=`) as `null == undefined` anyway.

Gotchas

Only use it [if you know what you're doing](#) as it's very unreliable for standard type checking.

Feature detection

I can't recall exactly what was said, but someone (I think) once told me that some vendor once implemented a feature with a falsy value if it didn't exist (though I'm not even certain that's true, worth a mention though)... and as such the `in` operator is best for these such cases:

```
// just an example, not the one somebody mentioned...
if ('draggable' in document.createElement('div')) {
    // do something if prop exists
}
```

CODE

36. What `(function (window, document, undefined) {})(window, document);` really means

Interestingly enough I get asked about the IIFE (immediately-invoked function expression) a lot, which takes the following setup:

```
(function (window, document, undefined) {
    //
})(window, document);
```

CODE

So why not write a post about it? ;-)

First, this does a series of different things. From the top:

Scope

JavaScript has function scope, so first this creates some much needed "private scope". For example:

```
(function (window, document, undefined) {
    var name = 'Todd';
})(window, document);

console.log(name); // name is not defined, it's in a different scope
```

CODE

Simple.

How it works

A normal function looks like this:

```
var logMyName = function (name) {
    console.log(name);
};

logMyName('Todd');
```

CODE

We get to *invoke* it by choice, and wherever we want/can scope providing.

The reason "IIFE" was coined was because they're immediately-invoked function expressions. Which means they're immediately called at runtime - also we can't call them again they run just once like this:

```
var logMyName = (function (name) {
    console.log(name); // Todd
})('Todd');
```

CODE

The secret sauce here is this, (which I've assigned to a variable in the previous example):

```
(function () {
```

CODE

```
)()();
```

The extra pair of parentheses *is* necessary as this doesn't work:

```
function () {
```

CODE

```
}();
```

Though several tricks can be done to trick JavaScript into "making it work". These force the JavaScript parser to treat the code following the ! character as an expression:

```
CODE
!function () {
}();
```

There are also other variants:

```
CODE
+function () {
}();
-function () {
}();
~function () {
}();
```

But I wouldn't use them.

Check out [Disassembling JavaScript's IIFE Syntax](#) by [@mariusschulz](#) for a detailed explanation of the IIFE syntax and its variants.

Arguments

Now we know how it works, we can pass in arguments to our IIFE:

```
CODE
(function (window) {
}) (window);
```

How does this work? Remember, the closing `(window);` is where the function is invoked, and we're passing in the `window` Object. This then gets passed into the function, which I've named `window` also. You could argue this is pointless as we should name it something different - but for now we'll use `window` as well.

So what else can we do? Pass in all the things! Let's pass in the `document` Object:

```
CODE
(function (window, document) {
  // we refer to window and document normally
})(window, document);
```

Local variables are faster to resolve than the global variables, but this is on a huge scale and you'll never notice the speed increase - but also worth considering if we're referencing our globals a lot!

What about `undefined` ?

In ECMAScript 3, `undefined` is mutable. Which means its value could be reassigned, something like `undefined = true;` for instance, oh my! Thankfully in ECMAScript 5 strict mode (`'use strict'`) the parser will throw an error telling you you're an idiot. Before this, we started protecting our IIFE's by doing this:

```
(function (window, document, undefined) {
})(window, document);
```

CODE

Which means if someone came along and did this, we'd be okay:

```
undefined = true;
(function (window, document, undefined) {
    // undefined is a local undefined variable
})(window, document);
```

CODE

Minifying

Minifying your local variables is where the IIFE pattern's awesomeness really kicks in. Local variable names aren't really needed if they're passed in, so we can call them what we like.

Changing this:

```
(function (window, document, undefined) {
    console.log(window); // Object window
})(window, document);
```

CODE

To this:

```
(function (a, b, c) {
    console.log(a); // Object window
})(window, document);
```

CODE

Imagine it, all your references to libraries and `window` and `document` nicely minified. Of course you don't need to stop there, we can pass in `jQuery` too or whatever is available in the lexical scope:

```
(function ($, window, document, undefined) {
    // use $ to refer to jQuery
    // $(document).addClass('test');
})(jQuery, window, document);

(function (a, b, c, d) {
    // becomes
    // a(c).addClass('test');
})(jQuery, window, document);
```

CODE

This also means you don't need to call `jQuery.noConflict();` or anything as `$` is assigned locally to the module. Learning how scopes and global/local variables work will help you even further.

A good minifier will make sure to rename `undefined` to `c` (for example, and only if used) throughout your script too. Important to note, *the name undefined is irrelevant*. We just need to know that the referencing Object is `undefined`, as `undefined` has no special meaning - `undefined` is the value javascript gives to things that are

declared but have no value.

Non-browser global environments

Due to things such as Node.js, the browser isn't always the global Object which can be a pain if you're trying to create IIFE's that work across multiple environments. For this reason, I tend to stick with this as a base:

```
(function (root) {  
})(this);
```

CODE

In a browser, the global environment `this` refers to the `window` Object, so we don't need to pass in `window` at all, we could always shorten it to `this`.

I prefer the name `root` as it can refer to non-browser environments as well as the root of the browser.

If you're interested in a universal solution (which I use all the time nowadays when creating open source project modules) is the UMD wrapper:

```
(function (root, factory) {  
    if (typeof define === 'function' && define.amd) {  
        define(factory);  
    } else if (typeof exports === 'object') {  
        module.exports = factory;  
    } else {  
        root.MYMODULE = factory();  
    }  
})(this, function () {  
    //  
});
```

CODE

This is some sexy stuff. The function is being invoked with another function passed into it. We can then assign it to the relevant environment inside. In the browser, `root.MYMODULE = factory();` is our IIFE module, elsewhere (such as Node.js) it'll use `module.exports` or `requireJS` if `typeof define === 'function' && define.amd` resolves true.

But this stuff is another story, but I insist you check out the [UMD repo](#).

37. Understanding JavaScript types and reliable type checking

Type checking in JavaScript can often be a pain, especially for new JS developers. I want to show you how to reliably check types in JS and understand them a little more. This post digs through Objects, Primitives, shadow objects/coercion, the `typeof` operator and how we can reliably get a "real" JavaScript type.

Objects versus Primitives

"Everything in JavaScript is an Object". Remember it, then forget it. It's not true. JavaScript makes the subject very difficult to understand though - it presents everything as some form of "object" if we dive into their Prototypes (later).

For now, let's look at types.

To understand JavaScript types, we need a top level view of them:

- Number
- String
- Boolean
- Object
- Null
- Undefined

We have `Number`, `String`, `Boolean` - these are Primitives (not Objects!). This means their values are unable to be changed because they are merely *values*, they have no properties. The Primitive types are wrapped by their Object counterparts when called, JavaScript will dive between the `Number/String/Boolean` to an Object when needed (coercion). Underneath, it will in fact construct an Object, use it, then return the result (all the instance will be shipped out for garbage collection).

For example using `'someString'.trim()`; will spin up an Object underneath and call the `.trim()` method on it.

`Null` and `undefined` are weird (both Primitives too), and distinguish between *no* value or an *unknown* value (`null` is unknown value, `undefined` is *totally* not known or even declared). There is also an [Error object](#).

Object's however are a different story. You'll notice I've not mentioned `Array` or `RegExp`, these are *types* of Object, let's investigate. Under the `Object` tree we have:

- Object
 - Function
 - Array
 - Date
 - RegExp

Having broken it down, things seem a little simpler, we have Objects versus Primitives. That's it, right? No, JavaScript decided it wanted to complicate *everything* you'd assume logical from above.

Typeof operator

From [MDN](#): "The `typeof` operator returns a string indicating the type of the unevaluated operand".

Based on our newly acquired knowledge from the above, you wouldn't expect this to happen:

```
typeof []; // object
typeof {}; // object
typeof ''; // string
typeof new Date() // object
typeof 1; // number
typeof function () {} // function
typeof /test/i; // object
typeof true; // boolean
typeof null; // object
typeof undefined; // undefined
```

CODE

Whyyyyy?! `Function` is an Object, but tells us it's a `function`, `Array` is an Object and says it is. `null` is an

Object, and so is our `RegExp`. What happened?

The `typeof` operator is a bit strange. Unless you know how to *really* use it, simply avoid it to avoid headaches. We wouldn't want something like this to happen:

```
// EXPECTATION
var person = {
  getName: function () {
    return 'Todd';
  };
};

if (typeof person === 'object') {
  person.getName();
}

// THIS GETS LET THROUGH...
// because I stupidly refactored some code changing the names
// but the `if` still lets through `person`
var person = [];
var myPerson = {
  getName: function () {
    return 'Todd';
  }
};

if (typeof person === 'object') {
  person.getName(); // Uncaught TypeError: undefined is not a function
}
```

CODE

`typeof` let us down here, what we really wanted to know was that `person` was a *plain Object*.

True Object types

There's a really simple way, though to look at it looks like a hack:

```
Object.prototype.toString.call();
```

CODE

The `.toString()` method is accessed using `Object.prototype` because every object descending from `Object` prototypically inherits it. By default, we get `[object Object]` when calling `{}.toString()` (an `Object`).

We can use `.call()` to change the `this` context (as it converts its argument to a value of type) and, for example, if we use `.call(/test/i)` (a Regular Expression) then `[object Object]` becomes `[object RegExp]`.

Which means if we run our test again using all JS types:

CODE

```
Object.prototype.toString.call([]); // [object Array]
Object.prototype.toString.call({}); // [object Object]
Object.prototype.toString.call(''); // [object String]
Object.prototype.toString.call(new Date()); // [object Date]
Object.prototype.toString.call(1); // [object Number]
Object.prototype.toString.call(function () {}); // [object Function]
Object.prototype.toString.call(/test/i); // [object RegExp]
Object.prototype.toString.call(true); // [object Boolean]
Object.prototype.toString.call(null); // [object Null]
Object.prototype.toString.call(); // [object Undefined]
```

We can then push this into a function and more reliably validate our previous function:

CODE

```
var getType = function (elem) {
    return Object.prototype.toString.call(elem);
};

if (getType(person) === '[object Object]') {
    person.getName();
}
```

To keep things DRY and save writing `==> '[object Object]'` or whatever out each time, we can create methods to simply reference. I've used `.slice(8, -1)`; inside the `getType` function to remove the unnecessary `[object` and `]` parts of the String:

CODE

```
var getType = function (elem) {
    return Object.prototype.toString.call(elem).slice(8, -1);
};

var isObject = function (elem) {
    return getType(elem) === 'Object';
};

if (isObject(person)) {
    person.getName();
}
```

Snazzy.

I put together all the above methods into a micro-library called [Axis.js](#) which you can use:

CODE

```
axis.isArray([]); // true
axis.isObject({}); // true
axis.isString(''); // true
axis.isDate(new Date()); // true
axis.isRegExp(/test/i); // true
axisisFunction(function () {}); // true
axis.isBoolean(true); // true
axis.isNumber(1); // true
axis.isNull(null); // true
axis.isUndefined(); // true
```

The code powering that does some cool stuff for those interested:

CODE

```
/*! axis v1.1.0 | (c) 2014 @toddmotto | github.com/toddmotto/axis */
(function (root, factory) {
    if (typeof define === 'function' && define.amd) {
        define(factory);
    } else if (typeof exports === 'object') {
        module.exports = factory;
    } else {
        root.axis = factory();
    }
})(this, function () {

    'use strict';

    var exports = {};

    var types = 'Array Object String Date RegExp Function Boolean Number Null Undefined'.split(' ');

    var type = function () {
        return Object.prototype.toString.call(this).slice(8, -1);
    };

    for (var i = types.length; i--;) {
        exports['is' + types[i]] = (function (self) {
            return function (elem) {
                return type.call(elem) === self;
            };
        })(types[i]);
    }

    return exports;
});


```

[Download Fork on GitHub](#)

38. Simple forEach implementation for Objects/NodeLists/Arrays

Looping Objects is easy. Looping Arrays is also easy. Looping NodeLists is easy. They can be a little repetitive though and often take time to construct each loop and pass in the index, property, element or whatever...

There is no "standard" way of iterating over everything. We can use `Array.prototype.forEach` to loop over Arrays (or the regular `for` loop), a `for in` loop for Objects, and a regular `for` loop again for NodeLists or HTML collections. No, you're not going to use that `forEach.call(NodeList)` [hack](#).

Wouldn't it be nice to just forget about what type of collection we're looping, forget about browser support and write a nice little function that handles everything for us. Yes.

So I did...

[Download Fork on GitHub](#)

forEach.js

`forEach.js` is a simple script, it's not part of a library or even a module, it's just a function, here's its syntax and a quick example using an `Array`:

```
CODE
// syntax
forEach(collection[, callback[, context]]);

// example
var myArray = ['A', 'B', 'C', 'D'];
forEach(myArray, function (value, index) {
    // `this` will reference myArray: []
}, myArray); // note third param changing execution context
```

forEach() for Arrays/NodeLists

You can loop over an `Array` or `NodeList` using a standard `for` loop, however, `NodeLists` cannot be used in conjunction with the newer ECMAScript 5 `Array.prototype.forEach`. This script takes care of that in the same way it loops over an `Array`, you'll get the same stuff passed back:

```
CODE
// Array:
forEach(['A', 'B', 'C', 'D'], function (value, index) {
    console.log(index); // 0, 1, 2, 3
    console.log(value); // A, B, C, D
});
// NodeList:
forEach(document.querySelectorAll('div'), function (value, index) {
    console.log(index); // 0, 1, 2, 3
    console.log(value); // <div>, <div>, <div>...
});
```

forEach() for Objects

Object iteration is usually done via a `for in` loop, we can wrap this up by passing back values which makes our loops cleaner and easier to manage:

```
CODE
// Object:
forEach({ name: 'Todd', location: 'UK' }, function (value, prop, obj) {
    console.log(value); // Todd, UK
    console.log(prop); // name, location
    console.log(obj); // { name: 'Todd', location: 'UK' }, { name: 'Todd', location: 'U
K' }
});
```

collection

Type: `Array|Object|NodeList`

Collection of items to iterate, could be an `Array`, `Object` or `NodeList`.

callback

Type: `Function`

Callback function for each iteration.

context

Type: Array|Object|NodeList Default: null

Object/NodeList/Array that `forEach` is iterating over, to use as the `this` value when executing callback.

Code

For those interested, check out the code below, the latest version is available [on GitHub](#).

```
var forEach = function (collection, callback, scope) {
  if (Object.prototype.toString.call(collection) === '[object Object]') {
    for (var prop in collection) {
      if (Object.prototype.hasOwnProperty.call(collection, prop)) {
        callback.call(scope, collection[prop], prop, collection);
      }
    }
  } else {
    for (var i = 0; i < collection.length; i++) {
      callback.call(scope, collection[i], i, collection);
    }
  }
};
```

CODE

[Download Fork on GitHub](#)

39. Mastering the Module Pattern

I'm a massive fan of JavaScript's Module Pattern and I'd like to share some use cases and differences in the pattern, and why they're important. The Module Pattern is what we'd call a "[design pattern](#)," and it's extremely useful for a vast amount of reasons. My main attraction to the Module Pattern (and its variant, the Revealing Module Pattern) is it makes scoping a breeze and doesn't overcomplicate program design.

It also keeps things very simple and easy to read and use, uses Objects in a very nice way, and doesn't bloat your code with repetitive `this` and `prototype` declarations. I thought I'd share some insight as to the awesome parts of the Module, and how you can master it, its variants and features.

Creating a Module

To understand what a Module can give you, you'll need to understand what the following `function` concept does:

```
(function () {
  // code
})();
```

CODE

It declares a function, which then calls itself immediately. These are also known as [Immediately-Invoked-Function-Expressions](#), in which the `function` creates new scope and creates "privacy". JavaScript doesn't have privacy, but

creating new scope emulates this when we wrap all our function logic inside them. The idea then is to return only the parts we need, leaving the other code out of the `global` scope.

After creating new `scope`, we need to namespace our code so that we can access any methods we return. Let's create a namespace for our anonymous Module.

```
CODE
var Module = (function () {
    // code
})();
```

We then have `Module` declared in the global scope, which means we can call it wherever we like, and even pass it into another Module.

Private methods

You'll see and hear a lot about `private` methods in JavaScript. But Javascript doesn't *strictly* have `private` methods, but we *can* create a working equivalent.

What *are* private methods you might be asking? Private methods are anything you don't want users/devs/hackers to be able to see/call outside the scope they're in. We might be making server calls and posting sensitive data, we *don't* want to expose those functions publicly, they could post anything back then and take advantage of our code. So we can create closure and be more sensible (as best as we can with JavaScript) at protecting our code. It's not *all* about protection however, there are also naming conflicts. I bet when you first started out writing jQuery/JavaScript, that you dumped all your code in one file and it was just `function`, `function`, `function`. Little did you know these were all global, and you probably suffered the consequence at some point. If so, you'll learn why, and what to do to change it.

So let's use our newly created `Module` scope to make our methods inaccessible outside of that scope. For beginners to the Module Pattern, this example will help understand how a private method would be defined:

```
CODE
var Module = (function () {
    var privateMethod = function () {
        // do something
    };

    })();
```

The above example declares our function `privateMethod`, which is locally declared inside the new scope. If we were to attempt calling it anywhere outside of our module, we'll get an error thrown and our JavaScript program will break! We don't want anyone to be able to call our methods, especially ones that might manipulate data and go back and forth to a server.

Understanding "return"

Typical Modules will use `return` and return an `Object` to the Module, to which the methods bound to the `Object` will be accessible from the Module's namespace.

A real light example of returning an `Object` with a `function` as a property:

```
CODE
var Module = (function () {
    return {
        publicMethod: function () {
            // code
        }
    };
})();
```

As we're returning an `Object Literal`, we can call them exactly like Object Literals:

```
CODE
Module.publicMethod();
```

For those who haven't used the Object Literal syntax before, a standard Object Literal could look something like this:

```
CODE
var myObjLiteral = {
    defaults: { name: 'Todd' },
    someMethod: function () {
        console.log(this.defaults);
    }
};

// console.log: Object { name: 'Todd' }
myObjLiteral.someMethod();
```

But the issue with Object Literals is the pattern can be abused. Methods *intended* to be "private" will be accessible by users because they are part of the Object. This is where the Module comes in to save us, by allowing us to define all our "private" stuff locally and only return "the good parts".

Let's look at a more Object Literal syntax, and a perfectly good Module Pattern and the `return` keyword's role. Usually a Module will return an Object, but how that Object is defined and constructed is totally up to you. Depending on the project and the role/setup of the code, I may use one of a few syntaxes.

Anonymous Object Literal return

One of the easiest patterns is the same as we've declared above, the Object has no name declared locally, we just return an Object and that's it:

```

var Module = (function () {

    var privateMethod = function () {};

    return {
        publicMethodOne: function () {
            // I can call `privateMethod()` you know...
        },
        publicMethodTwo: function () {

        },
        publicMethodThree: function () {

        }
    };
})();

```

CODE

Locally scoped Object Literal

Local scope means a variable/function declared inside a scope. On the [Conditionizr](#) project, we use a locally scoped namespace as the file is over 100 lines, so it's good to be able to see what are the public and private methods without checking the `return` statement. In this sense, it's *much* easier to see what *is* public, because they'll have a locally scoped namespace attached:

```

var Module = (function () {

    // locally scoped Object
    var myObject = {};

    // declared with `var`, must be "private"
    var privateMethod = function () {};

    myObject.someMethod = function () {
        // take it away Mr. Public Method
    };

    return myObject;
})();

```

CODE

You'll then see on the last line inside the Module that `myObject` is returned. Our global `Module` doesn't care that the locally scoped `Object` has a name, we'll only get the actual Object sent back, not the name. It offers for better code management.

Stacked locally scoped Object Literal

This is pretty much identical as the previous example, but uses the "traditional" single Object Literal notation:

```

var Module = (function () {

    var privateMethod = function () {};

    var myObject = {
        someMethod: function () {

        },
        anotherMethod: function () {

        }
    };

    return myObject;

})();

```

CODE

I prefer the second approach we looked at, the *Locally scoped Object Literal*. Because here, we have to declare *other* functions before we use them (you should do this, using `function myFunction () {}` hoists your functions and can cause issues when used incorrectly). Using `var myFunction = function () {};` syntax lets us not worry about this, as we'll declare them all before we use them, this also makes debugging easier as the JavaScript interpreter will render our code in the order we declare, rather than hoisting `function` declarations. I also don't like this approach so much, because the "stacking" method can often get verbose looking, and there is no obvious locally scoped `Object` namespace for me to bolt public methods onto.

Revealing Module Pattern

We've looked at the Module, and there's a really neat variant which is deemed the "revealing" pattern, in which we reveal public pointers to methods inside the Module's scope. This again, can create a really nice code management system in which you can clearly see and define which methods are shipped *back* to the Module:

```

var Module = (function () {

    var privateMethod = function () {
        // private
    };

    var someMethod = function () {
        // public
    };

    var anotherMethod = function () {
        // public
    };

    return {
        someMethod: someMethod,
        anotherMethod: anotherMethod
    };

})();

```

CODE

I really like the above syntax, as it's very declarative. For bigger JavaScript Modules this pattern helps out a lot more,

using a standard "Module Pattern" can get out of control depending on the syntax you go for and how you structure your code.

Accessing "Private" Methods

You might be thinking at some stage during this article, "*So if I make some methods private, how can I call them?*". This is where JavaScript becomes even more awesome, and allows us to actually *invoke* private functions via our public methods. Observe:

```
CODE
var Module = (function () {

    var privateMethod = function (message) {
        console.log(message);
    };

    var publicMethod = function (text) {
        privateMethod(text);
    };

    return {
        publicMethod: publicMethod
    };
})();

// Example of passing data into a private method
// the private method will then `console.log()` 'Hello!'
Module.publicMethod('Hello!');
```

You're not just limited to methods, though. You've access to Objects, Arrays, anything:

```
CODE
var Module = (function () {

    var privateArray = [];

    var publicMethod = function (somethingOfInterest) {
        privateArray.push(somethingOfInterest);
    };

    return {
        publicMethod: publicMethod
    };
})();
```

Augmenting Modules

So far we've created a nice Module, and returned an Object. But what if we wanted to extend our Module, and include another smaller Module, which extends our original Module?

Let's assume the following code:

```

var Module = (function () {

    var privateMethod = function () {
        // private
    };

    var someMethod = function () {
        // public
    };

    var anotherMethod = function () {
        // public
    };

    return {
        someMethod: someMethod,
        anotherMethod: anotherMethod
    };

})();

```

CODE

Let's imagine it's part of our application, but by design we've decided to not include something into the core of our application, so we could include it as a standalone Module, creating an extension.

So far our Object for Module would look like:

```
Object {someMethod: function, anotherMethod: function}
```

CODE

But what if I want to add our Module extension, so it ends up with *another* public method, maybe like this:

```
Object {someMethod: function, anotherMethod: function, extension: function}
```

CODE

A third method is now available, but how do we manage it? Let's create an aptly named `ModuleTwo`, and pass in our `Module` namespace, which gives us access to our Object to extend:

```

var ModuleTwo = (function (Module) {

    // access to `Module`

}) (Module);

```

CODE

We could then create *another* method inside this module, have all the benefits of private scoping/functionality and then return our extension method. My pseudo code could look like this:

```
var ModuleTwo = (function (Module) {
    Module.extension = function () {
        // another method!
    };
    return Module;
})(Module || {});
```

CODE

Module gets passed into ModuleTwo , an extension method is added and then returned *again*. Our Object is getting thrown about, but that's the flexibility of JavaScript :D

I can then see (through something like Chrome's Dev Tools) that my initial Module now has a third property:

```
// Object {someMethod: function, anotherMethod: function, extension: function}
console.log(Module);
```

CODE

Another hint here, you'll notice I've passed in Module || {} into my second ModuleTwo , this is incase Module is undefined - we don't want to cause errors now do we ;). What this does is instantiate a *new* Object, and bind our extension method to it, and return it.

Private Naming Conventions

I personally love the Revealing Module Pattern, and as such, I have many functions dotting around my code that visually are all declared the same, and look the same when I'm scanning around. I sometimes create a locally scoped Object, but sometimes don't. When I don't, how can I distinguish between private variables/methods? The _ character! You've probably seen this dotted around the web, and now you know why we do it:

```
var Module = (function () {
    var _privateMethod = function () {
        // private stuff
    };

    var publicMethod = function () {
        _privateMethod();
    };

    return {
        publicMethod: publicMethod
    };
})();
```

CODE

40. Understanding the "this" keyword in JavaScript

It's safe to say that the this keyword is probably one of the misunderstood parts of JavaScript. Admittedly, I used to throw the this keyword around until my script worked and it confused the hell out of me (and still does many other

JS developers) around the world. Only when I learned about lexical scope, how functions are invoked, scope context, a few context changing methods - I really understood it.

Before you dive into this article, here's a few very important points to takeaway and remember about the `this` keyword:

- The `this` keyword's value has nothing to do with the function itself, how the function is called determines the `this` value
- It can be dynamic, based on how the function is called
- You can change the `this` context through `.call()`, `.apply()` and `.bind()`

Default this context

There are a few different ways the `this` value changes, and as we know it's usually the call-site that creates the context.

Window Object, global scope

Let's take a quick example at how simply calling regular functions binds the `this` value differently:

```
// define a function
var myFunction = function () {
    console.log(this);
};

// call it
myFunction();
```

CODE

What can we expect the `this` value to be? By default, this should always be the `window` Object, which refers to the root - the global scope. So when we `console.log(this);` from our function, as it's invoked by the window (simply just called), we should expect the `this` value to be our `window` Object:

```
// define a function
var myFunction = function () {
    console.log(this); // [object Window]
};

// call it
myFunction();
```

CODE

Object literals

Inside Object literals, the `this` value will always refer to its own Object. Nice and simple to remember. That is good news when invoking our functions, and one of the reasons I adopt patterns such as the module pattern for organising my objects.

Here's how that might look:

CODE

```
// create an object
var myObject = {};

// create a method on our object
myObject.someMethod = function () {
    console.log(this);
};

// call our method
myObject.someMethod();
```

Here, our `window` Object didn't invoke the function - our `Object` did, so `this` will refer to the `Object` that called it:

CODE

```
// create an object
var myObject = {};

// create a method on our object
myObject.someMethod = function () {
    console.log(this); // myObject
};

// call our method
myObject.someMethod();
```

Prototypes and Constructors

The same applies with Constructors:

CODE

```
var myConstructor = function () {
    this.someMethod = function () {
        console.log(this);
    };
};

var a = new myConstructor();
a.someMethod();
```

And we can add a Prototype Object as well:

CODE

```

var myConstructor = function () {
    this.someMethod = function () {
        console.log(this);
    };
};

myConstructor.prototype = {
    somePrototypeMethod: function () {
        console.log(this);
    }
};

var a = new myConstructor();
a.someMethod();
a.somePrototypeMethod();

```

Interestingly, in both cases the `this` value will refer to the Constructor object, which will be `myConstructor`.

Events

When we bind events, the same rule applies, the `this` value points to the owner. The owner in the following example would be the element.

CODE

```

// let's assume .elem is <div class="elem"></div>
var element = document.querySelector('.elem');
var someMethod = function () {
    console.log(this);
};
element.addEventListener('click', someMethod, false);

```

Here, `this` would refer to `<div class="elem"></div>`.

Dynamic this

The second point I made in the intro paragraph was that `this` is dynamic, which means the value could change. Here's a real simple example to show that:

CODE

```

// let's assume .elem is <div class="elem"></div>
var element = document.querySelector('.elem');

// our function
var someMethod = function () {
    console.log(this);
};

// when clicked, `this` will become the element
element.addEventListener('click', someMethod, false); // <div>

// if we just invoke the function, `this` becomes the window object
someMethod(); // [object Window]

```

Changing this context

There are often many reasons why we need to change the context of a function, and thankfully we have a few methods at our disposal, these being `.call()`, `.apply()` and `.bind()`.

Using any of the above will allow you to change the context of a function, which in effect will change the `this` value. You'll use this when you want `this` to refer to something different than the scope it's in.

Using `.call()`, `.apply()` and `.bind()`

"Functions are first class Objects" you'll often hear, this means they can also have their own methods!

The `.call()` method allows you to change the scope with a specific syntax [ref](#):

```
.call(thisArg[, arg1[, arg2[, ...]]]);
```

CODE

Usage would look something like this:

```
someMethod.call(anotherScope, arg1, arg1);
```

CODE

You'll notice further arguments are all comma separated - this is the only difference between `.call()` and `.apply()`:

```
someMethod.call(anotherScope, arg1, arg1); // commas
someMethod.apply(anotherScope, [arg1, arg1]); // array
```

CODE

With any of the above, they immediately invoke the function. Here's an example:

```
var myFunction = function () {
  console.log(this);
};

myFunction.call();
```

CODE

Without any arguments, the function is just invoked and `this` will remain as the `window` Object.

Here's a more practical usage, this script will always refer to the `window` Object:

```
var numbers = [
  {name: 'Mark'},
  {name: 'Tom'},
  {name: 'Travis'}
];
for (var i = 0; i < numbers.length; i++) {
  console.log(this); // window
}
```

CODE

The `forEach` method also has the same effect, it's a function so it creates new scope:

```
CODE
var numbers = [
  {
    name: 'Mark'
  },
  {
    name: 'Tom'
  },
  {
    name: 'Travis'
  }];
numbers.forEach(function () {
  console.log(this); // window
});
```

We could change each iteration's scope to the current element's value inside a regular `for` loop as well, and use `this` to access object properties:

```
CODE
var numbers = [
  {
    name: 'Mark'
  },
  {
    name: 'Tom'
  },
  {
    name: 'Travis'
  }];
for (var i = 0; i < numbers.length; i++) {
  (function () {
    console.log(this.name); // Mark, Tom, Travis
  }).call(numbers[i]);
}
```

This is especially extensible when passing around other Objects that you might want to run through the exact same functions.

forEach scoping

Not many developers using `forEach` know that you can change the initial scope context via the second argument:

```
CODE
numbers.forEach(function () {
  console.log(this); // this = Array [{ name: 'Mark' }, { name: 'Tom' }, { name: 'Travis' }]
}, numbers); // BOOM, scope change!
```

Of course the above example doesn't change the scope to how we want it, as it changes the functions scope for every iteration, not each individual one - though it has use cases for sure!

To get the *ideal* setup, we need:

```

var numbers = [
  {name: 'Mark'},
  {name: 'Tom'},
  {name: 'Travis'}
];
numbers.forEach(function (item) {
  (function () {
    console.log(this.name); // Mark, Tom, Travis
  }).call(item);
});

```

CODE

.bind()

Using `.bind()` is an ECMAScript 5 addition to JavaScript, which means it's not supported in all browsers (but can be polyfilled so you're all good if you need it). `Bind` has the same effect as `.call()`, but instead binds the function's context *prior* to being invoked, this is essential to understand the difference. Using `.bind()` *will not* invoke the function, it just "sets it up".

Here's a really quick example of how you'd setup the context for a function, I've used `.bind()` to change the context of the function, which by default the `this` value would be the window Object.

```

var obj = {};
var someMethod = function () {
  console.log(this); // this = obj
}.bind(obj);
someMethod();

```

CODE

This is a really simple use case, they can also be used in event handlers as well to pass in some extra information without a needless anonymous function:

```

var obj = {};
var element = document.querySelector('.elem');
var someMethod = function () {
  console.log(this);
};
element.addEventListener('click', someMethod.bind(obj), false); // bind

```

CODE

"Jumping scope"

I call this jumping scope, but essentially it's just some slang for accessing a lexical scope reference (also a bit easier to remember...).

There are many times when we need to access lexical scope. Lexical scope is where variables and functions are still accessible to us in parent scopes.

```

var obj = {};

obj.myMethod = function () {
    console.log(this); // this = `obj`
};

obj.myMethod();

```

CODE

In the above scenario, `this` binds perfectly, but what happens when we introduce another function. How many times have you encountered a scope challenge when using a function such as `setTimeout` inside another function? It totally screws up any `this` reference:

```

var obj = {};
obj.myMethod = function () {
    console.log(this); // this = obj
    setTimeout(function () {
        console.log(this); // window object :0!!!
    }, 100);
};
obj.myMethod();

```

CODE

So what happened there? As we know, [functions create scope](#), and `setTimeout` will be invoked by itself, defaulting to the `window` Object, and thus making the `this` value a bit strange inside that function.

Important note: `this` and the `arguments` Object are the only objects that *don't* follow the rules of lexical scope.

How can we fix it? There are a few options! If we're using `.bind()`, it's an easy fix, note the usage on the end of the function:

```

var obj = {};
obj.myMethod = function () {
    console.log(this); // this = obj
    setTimeout(function () {
        console.log(this); // this = obj
    }.bind(this), 100); // .bind() #ftw
};
obj.myMethod();

```

CODE

We can also use the jumping scope trick, `var that = this;`:

```

var obj = {};
obj.myMethod = function () {
    var that = this;
    console.log(this); // this = obj
    setTimeout(function () {
        console.log(that); // that (this) = obj
    }, 100);
};
obj.myMethod();

```

CODE

We've cut the `this` short and just simply pushed a reference of the scope into the new scope. It's kind of cheating, but works wonders for "jumping scope". With newcomers such as `.bind()`, this technique is sometimes frowned upon if used and abused.

One thing I dislike about `.bind()` is that you could end up with something like this:

```
CODE
var obj = {};
obj.myMethod = function () {
    console.log(this);
    setTimeout(function () {
        console.log(this);
        setTimeout(function () {
            console.log(this);
            setTimeout(function () {
                console.log(this);
                setTimeout(function () {
                    console.log(this);
                    }.bind(this), 100); // bind
                    }.bind(this), 100); // bind
                    }.bind(this), 100); // bind
                    }.bind(this), 100); // bind
    });
    obj.myMethod();
}
```

A tonne of `.bind()` calls, which look totally stupid. Of course this is an exaggerated issue, but it can happen very easily when switching scopes. In my opinion this would be easier - it will also be tonnes quicker as we're saving lots of function calls:

```
CODE
var obj = {};
obj.myMethod = function () {
    var that = this; // one declaration of that = this, no fn calls
    console.log(this);
    setTimeout(function () {
        console.log(that);
        setTimeout(function () {
            console.log(that);
            setTimeout(function () {
                console.log(that);
                setTimeout(function () {
                    console.log(that);
                    }, 100);
                    }, 100);
                    }, 100);
                    }, 100);
    });
    obj.myMethod();
}
```

Do what makes sense!

jQuery `$(this)`

Yes, the same applies, don't use `$(this)` unless you actually know what it's doing. What it *is* doing is passing the normal `this` value into a new jQuery Object, which will then inherit all of jQuery's prototypal methods (such as `addClass`), so you can instantly do this:

```
$('.elem').on('click', function () {  
    $(this).addClass('active');  
});
```

CODE

Happy scoping ;)

41. Instantiation Patterns In JavaScript

In this tutorial, we're going to talk a bit about JavaScript objects and dive into the four different object instantiation patterns.

In a lot of my demos, there are bits of JavaScript that pave the way for visual interactivity. The great thing about those scripts though is that they are reusable as "instances" that can exist around each other, without ever interfering with each other. In other words, if we can have many buttons that perform the same set of functionality, without ever having to repeat that functionality or worry about it conflicting. These are in fact JavaScript objects with added functionality, and each object instance is responsible for itself.

In JavaScript, there are 4 instantiation patterns available to us. We'll go through each of them here, and discuss the pros and cons of each. In the end, I'll give my opinion on my favourite one. Here are the 4 patterns:

1. Functional Instantiation
2. Functional-shared instantiation
3. Prototypal instantiation
4. Pseudoclassical instantiation

Let's look at each of them below.

Functional Instantiation

Functional instantiation is at the root of object instantiation in JavaScript. We create a function, and inside it, we can create an empty object, some scoped variables, some instance variables, and some instance methods. At the end of it all, we can return that instance, so that every time the function is called, we have access to those methods. Here it is in action:

CODE

```
// Set up
var Func = function() {
  var someInstance = {};
  var a = 0;
  var b = 1;

  someInstance.method1 = function() {
    // code for method1 here
  }

  someInstance.method2 = function() {
    // code for method2 here
  }

  someInstance.logA = function() {
    return a;
  }

  return someInstance;
}

// Usage
var myFunc = Func();
myFunc.logA(); // returns a
```

Pros: This pattern is the easiest to follow, as everything exists inside the function. It's instantly obvious that those methods and variables belong to that function.

Cons: This pattern creates a new set of functions in memory for each instance of the function `Func`. If you're creating a big app, it's ultimately just not suitable in terms of memory.

Functional Shared Instantiation

Functional-shared instantiation is similar to functional instantiation, but the methods are an extension of the function instead. Like before, we create an empty object inside our function and return that object. Before we return it though, we extend it with some function methods. For that, we'll need an extender function. Here's the code:

```
// Set up
var Func = function() {
  var someInstance = {};
  someInstance.a = 0;
  someInstance.b = 1;
  extend(someInstance, funcMethods);

  return someInstance;
}

var extend = function(to, from) {
  for (var key in from) {
    to[key] = from[key];
  }
}

var funcMethods = {};

funcMethods.method1 = function() {
  // code for method1 here...
}

funcMethods.method2 = function() {
  // code for method2 here...
}

funcMethods.logA = function() {
  return this.a;
}

// Usage
var myFunc = Func();
myFunc.logA(); // returns a
```

Pros: In this instantiation pattern, the object methods are referenced in memory, so object instances refer to those references when being called. This allows for great memory management.

Cons: If we choose to edit some of the `funcMethods` at some point and then create a new object instance, the old and new instances will refer to different references of the method in memory. This can get confusing for some, but is a minor caveat once you're aware of it.

Prototypal Instantiation

Prototypal instantiation (which doesn't actually use the keyword `prototype`) is done by attaching methods directly to the object's prototype using the `Object.create()` method. It's fairly similar to the previous pattern, except this time, the returned object has prototypal methods that directly reference the `funcMethods` object. Check it out:

```
// Set up
var Func = function() {
  var someInstance = Object.create(funcMethods);
  someInstance.a = 0;
  someInstance.b = 1;

  return someInstance;
}

var funcMethods = {};

funcMethods.method1 = function() {
  // code for method1 here...
}

funcMethods.method2 = function() {
  // code for method2 here...
}

funcMethods.logA = function() {
  return this.a;
}

// Usage
var myFunc = Func();
myFunc.logA(); // returns a
```

Pros: This pattern attaches methods directly to the object's prototype, rather than as attachments to the returned objects like before.

Cons: In my opinion, there is room for improvement on this pattern. Even though the pros are great, it's still a bit of a long-winded implementation.

Pseudoclassical Instantiation

The last pattern is the pseudoclassical pattern, which takes a bit of the long-windedness out of the prototypal pattern. It does, as before, attach methods directly to the object's prototype. Another point of interest is the fact that the object's constructor is automatically included. This means that we can create new instances using the `new` keyword. The reason for this is that some behind the scenes work goes down in this pattern. Before and after all code in the object's body, `this = Object.create(Object.prototype);` and `return this;` get run in the background. Here's the pseudoclassical style in action:

```
// Set up
var Func = function() {
  this.a = 0;
  this.b = 1;
}

Func.prototype.method1 = function() {
  // code for method1 here...
}

Func.prototype.method2 = function() {
  // code for method2 here...
}

Func.prototype.logA = function() {
  return this.a;
}

var myFunc = new Func();
myFunc.logA(); // returns a
```

Pros: Methods are attached directly to the prototype, instances are created with the `new` keyword, and the `this` keyword has a distinct scope. Rumour has it that it's the fastest instantiation pattern too, though that would be irrelevant unless you are creating upward of tens of thousands of objects at once.

Cons: Newcomers to JavaScript might find this pattern a bit difficult to understand, especially if their grasp on the `this` keyword is not up to scratch. It also can pose some problems when the meaning of the `this` keyword changes, for example calling an object method from an event listener of some sort.

Wrap Up

And that's a wrap! In this tutorial, we looked at the four different instantiation patterns available to us in JavaScript. Thanks for reading, and remember, if you have any questions, comments, or feedback, you can also leave them below.

42. JavaScript Objects & Building A JavaScript Component -- Part 1

Here's part 1 of a 2-part tutorial that dives into JavaScript objects, prototype programming, and building a simple reusable JavaScript component.

Introduction

In this two-part tutorial, we're going to take a look at JavaScript objects, prototype programming, and building our very first JavaScript component that we can reuse and extend in various parts of our application. Let's get started!

About Objects & JavaScript

Think of an object as a collection of *things*. For example, imagine you have a bicycle. That bike is your object, and has a collection of *things* called **properties**. An example of a property would be the bike's model, the year it was made, and its components. Its components might then have their own set of properties, like the brakes, seat, derailleur,

wheels, etc. These are all properties of the "bike" object. Pretty straightforward, yes?

JavaScript by nature is an object-based language, where everything has its own set of properties that we can access. Here's a more formal definition from the MDN:

JavaScript is designed on a simple object-based paradigm. An object is a collection of properties, and a property is an association between a name and a value. A property's value can be a function, in which case the property is known as a method. In addition to objects that are predefined in the browser, you can define your own objects.

That's neat...not only are we provided a set of predefined objects in the browser, but we can create our own. In JavaScript, we access an object's properties via the dot notation. But first, let's take a quick look at the creation and instantiation of an object.

Creation Of An Object

You've probably created or used objects in JavaScript without even knowing it. That's because almost everything in JavaScript is an object. Here's a note from the MDN:

All primitive types except `null` and `undefined` are treated as objects. They can be assigned properties (assigned properties of some types are not persistent), and they have all characteristics of objects.

While there are a vast number of predefined objects at our disposal, I won't be going through any of that. From this point on, and throughout the rest of this tutorial, we're going to build a very simple JavaScript component called SimpleAlert. With that in mind, let's create our own object.

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

CODE

And that's it! Pretty easy, huh? This is clearly useless though, unless we add some properties to it. We can do this by using the dot notation. Let's add some.

```
// create new object
var simpleAlert = new Object();

// add some properties
simpleAlert.sa_default = "Hello World!";
simpleAlert.sa_error = "Error...";
simpleAlert.sa_success = "Success!";

// output object in the console
console.log(simpleAlert);
```

CODE

From the console, we can see that our object now has three properties. Since we set these properties above, we can now access them anywhere in our script. For example, if we wanted to send an error alert to the user, we could do this:

CODE

```
// alert error message
alert(simpleAlert.sa_error);
```

This is just one way to create an object and access its properties.

Creating Objects With Object Initializers

Another way is using what's called "object initialisers". According to the MDN:

...you can create objects using an object initializer. Using object initializers is sometimes referred to as creating objects with literal notation. "Object initializer" is consistent with the terminology used by C++.

Restructuring the creation of our SimpleAlert object would be as easy as this:

```
// create new object
var simpleAlert = {
  sa_default    : "Hello World!",
  sa_error      : "Error...",
  sa_success    : "Success!"
}

// output object in the console
console.log(simpleAlert);
```

CODE

Property values can also be functions. For example:

```
// create new object
var simpleAlert = {
  sa_default    : "Hello World!",
  sa_error      : "Error...",
  sa_success    : "Success!",
  sa_fallback   : function(){
    console.log("Fallback");
  }
}

// run the fallback
simpleAlert.sa_fallback();
```

CODE

The above should output "Fallback" to the console. When a property's value is a function, it is known as a **method**.

Using A Constructor Function

Another great way to create objects in JavaScript is to use a "constructor function". From the MDN:

Define the object type by writing a constructor function. There is a strong convention, with good reason, to use a capital initial letter. Create an instance of the object with new .

You can read more about it [here](#). Using the constructor function, we can rewrite our object creation like this:

```
// constructor function
function SimpleAlert( sa_default, sa_error, sa_success ) {
  this.sa_default = sa_default;
  this.sa_error = sa_error;
  this.sa_success = sa_success;
}

// creation of new object
var my_alert = new SimpleAlert( "Hello World!", "Error...", "Success!" );
```

CODE

Now if we output to the console, we can see our entire object. We can also now get the properties of our object via dot notation.

```
console.log(my_alert); // outputs object
console.log(my_alert.sa_error); // outputs "Error..."
```

CODE

The beauty of this technique is that you can now create multiple instances of the object, and pass in different values (which can be later accessed as properties). Properties can also be objects and even functions.

A Whole Lot More About Objects

In JavaScript, there's a whole lot more to objects than I mentioned above, although the methods mentioned above are a great start. The "initialiser" and "constructor function" methods in particular are useful, and we'll use these two methods below to create our very first SimpleAlert JavaScript component. I highly recommend reading through the MDN document [Working With Objects](#) to boost your knowledge even more. Also, take a read through the [Object](#) document to see what properties/methods are readily available for us.

Building Blocks For Our Component

We're going to make a component called SimpleAlert, that posts an alert (or message) to the user's screen when they press a button. The message that gets displayed will depend on which button the user presses. Here's a look at the HTML:

```
<div id="component">
  <button id="default">Show Default Message</button>
  <button id="success">Show Success Message</button>
  <button id="error">Show Error Message</button>
</div>
```

CODE

For our component, we'll wrap all our code in a self executing function and then make it available in the global namespace. We'll start off like this:

CODE

```
; (function( window ) {
    'use strict';
})( window );
```

There are a couple points to note here:

1. I'm using strict mode. You can read more about strict mode [here](#).
2. We are passing `window` into our self-executing function so that we can later add our `SimpleAlert` component to the global namespace.

Let's build a bit on our component. We want to first create our function `SimpleAlert`. We also want to add it to the global namespace. Here's how we'll do this:

```
; (function( window ) {
    'use strict';

    /**
     * SimpleAlert function
     */
    function SimpleAlert( message ) {
        this.message = message;
    }

    // rest of code here...

    /**
     * Add SimpleAlert to global namespace
     */
    window.SimpleAlert = SimpleAlert;
})( window );
```

CODE

So far, if we create a new instance of our object, nothing happens. It just exists, and that's pretty sad. For example, if we do this:

```
(function() {
    /**
     * Show default
     */
    var default_btn = document.getElementById( "default" );
    default_btn.addEventListener( "click", function() {
        var default_alert = new SimpleAlert("Hello World!");
        default_alert;
    } );
})(());
```

CODE

Before I continue, I need to introduce to you briefly `Object.prototype`. This represents the Object prototype object. From the MDN:

All objects in JavaScript are descended from `Object`; all objects inherit methods and properties from `Object.prototype`, although they may be overridden (except an `Object` with a `null` prototype, i.e. `Object.create(null)`).

For more reading, go [here](#). We'll be utilising this in our component creation. Let's build onto our component a bit. Let's add an `init` function, and call that function as soon as our object is created. Inside this `init` function, let's send our message to the console. Now, our JavaScript looks like this:

```
; (function( window ) {
```

CODE

```
    'use strict';

    /**
     * SimpleAlert function
     */
    function SimpleAlert( message ) {
        this.message = message;
        this._init();
    }

    /**
     * Initialise the message
     */
    SimpleAlert.prototype._init = function() {
        console.log(this.message);
    }

    /**
     * Add SimpleAlert to global namespace
     */
    window.SimpleAlert = SimpleAlert;

})( window );
```

And now, our new instance of our object created above should output "Hello World!" to the console. We're getting somewhere! Let's actually show the notice to the user on screen, and not in the console. Our `_init` function may now look like this:

CODE

```

;(function( window ) {

    'use strict';

    /**
     * SimpleAlert function
     */
    function SimpleAlert( message ) {
        this.message = message;
        this._init();
    }

    /**
     * Initialise the message
     */
    SimpleAlert.prototype._init = function() {
        this.component = document.getElementById("component");
        this.box = document.createElement("div");
        this.box.className = "simple-alert";
        this.box.innerHTML = this.message;
        this.component.appendChild( this.box );
    }

    /**
     * Add SimpleAlert to global namespace
     */
    window.SimpleAlert = SimpleAlert;

})( window );

```

Let's go through the `_init` function step by step.

1. Inside our constructor function, we are making the variable `message` accessible throughout by writing `this.message = message`.
2. We then call our `_init` function by writing `this._init()`. Every time we create a new instance of our `SimpleAlert` object, these two steps automatically run, so we automatically go to `_init()`.
3. Inside `_init()`, we are assigning our variables by using the `this` keyword and a variable name of our choice. We fetch the `#component` dive, and create our new `div` element where we would have our new `SimpleAlert` show up. Then we append the message to this new `div` that we created.

Pretty neat, huh?

The Code In Full

Here is the code in full, including some CSS to tie it all together neatly. Firstly, the HTML:

CODE

```

<div id="component">
    <button id="default">Show Default Message</button>
    <button id="success">Show Success Message</button>
    <button id="error">Show Error Message</button>
</div>

```

Same as we've seen above. Here's some CSS to make our alerts look a bit nice on screen:

CODE

```
.simple-alert {  
    padding: 20px;  
    border: solid 1px #ebebeb;  
}
```

Simple. Edit at your will, of course! And finally, here's the JavaScript that covers our component, and the actual event listeners on all three buttons triggering new instances of our object and hence displaying alerts to the screen:

CODE

```
// COMPONENT
//
// The building blocks for our SimpleAlert component.
///////////////////////////////
;(function( window ) {

    'use strict';

    /**
     * SimpleAlert function
     */
    function SimpleAlert( message ) {
        this.message = message;
        this._init();
    }

    /**
     * Initialise the message
     */
    SimpleAlert.prototype._init = function() {
        this.component = document.getElementById("component");
        this.box = document.createElement("div");
        this.box.className = "simple-alert";
        this.box.innerHTML = this.message;
        this.component.appendChild( this.box );
        this._initUIActions;
    }

    SimpleAlert.prototype._initUIActions = function() {

    }

    /**
     * Add SimpleAlert to global namespace
     */
    window.SimpleAlert = SimpleAlert;

})( window );

// EVENTS
//
// The code for the creation of new object instances,
// depending on which button is pressed.
///////////////////////////////
;(function() {

    /**
     * Show default
     */
    var default_btn = document.getElementById( "default" );
    default_btn.addEventListener( "click", function() {
        var default_alert = new SimpleAlert("Hello World!");
        default_alert;
    } );

    /**
     * Show success
     */
    var default_btn = document.getElementById( "success" );
    default_btn.addEventListener( "click", function() {


```

```

var default_alert = new SimpleAlert("Success!!!");
default_alert;
} );

/** 
 * Show error
 */
var default_btn = document.getElementById( "error" );
default_btn.addEventListener( "click", function() {
    var default_alert = new SimpleAlert("Error...");
    default_alert;
} );

})();

```

Wrap Up

Here, we looked into JavaScript objects and we've seen how to create a very basic JavaScript component using objects. This demonstration is simple, but provides us with some fundamental knowledge for creating reusable and extendable components. Think of how you might add a "dismiss" button to the alert, triggering a new function `hide()` when pressed.

Next time, we expand a bit more on our `SimpleAlert` component, and we learn how to set default options and pass in our own set of options. This is where we will see the beauty of this type of programming coming into play. I hope you enjoyed this tutorial, and if you have any questions, comments or feedback, leave it below. Stay tuned for part 2, and thanks for reading!

43. JavaScript Objects & Building A JavaScript Component -- Part 2

Here's part 2 of a 2-part tutorial that dives into JavaScript objects, prototype programming, and building a simple reusable JavaScript component.

[Get Source](#) [View Demo](#)

Introduction

In [part one](#) of this two-part tutorial, we went into some detail about JavaScript objects, and created a very basic component. Our component, however, only took on one parameter, and isn't easily extendable. In part two (this tutorial), we're going to look into some interesting functions, and give our component some options. These options can be specified each time we create a new instance of our component, giving us more control over the output.

Simple Alert Outline

The outline for our component will be as follows:

1. We want the user to wire up a Simple Alert to whatever element he/she desires. For this tutorial, I'll be wiring up Simple Alerts to buttons.
2. We want the user to specify what type of alert will be shown. The user will have 4 options - error, success, warning, and default.
3. We want to give different styles to each of these alerts, so that the user experience is a bit more in line with

what users are accustomed to (i.e. red is error, green is success, etc).

4. We want users to be able to dismiss the alerts. This functionality will be written inside our Simple Alert component.

So off the bat, we know we need some HTML and CSS structure. Because the JavaScript is actually creating the HTML for us on each instance of the alert, I like to style it in some test markup before so I know how it would look. With that in mind, let's take a look at what the markup should be like.

Simple Alert Test Markup

In general, our alerts would look something like this after the JS outputs it:

```
<div class="simple-alert TYPE">
  <span class="simple-alert__content">Message here</span>
  <a href="#" class="simple-alert__dismiss"></a>
</div>
```

CODE

"TYPE" is the type of alert, and this class, as mentioned above, will be represented by either of 4 types:

- .error
- .success
- .warning
- .default

Knowing all this, we now have some classes to work with. So let's dig into the CSS.

The CSS For Our JS Object Output

As seen in the markup, we have some classes to work with that will give our alerts some style. I mentioned before that each of our four types of alerts will have different colour schemes to invoke some familiarity to users, so let's style with that in mind. Here's the CSS I came up with:

```
.simple-alert {  
    display: inline-block;  
    position: relative;  
    margin: 0 5px 5px 0;  
    padding: 5px 45px 5px 5px;  
}  
  
.simple-alert.error {  
    background-color: #f06464;  
    border: solid 1px #d91515;  
}  
  
.simple-alert.success {  
    background-color: lightgreen;  
    border: solid 1px #38e038;  
}  
  
.simple-alert.warning {  
    background-color: moccasin;  
    border: solid 1px #ffb44f;  
}  
  
.simple-alert.default {  
    background-color: lightblue;  
    border: solid 1px #5fb3ce;  
}  
  
.simple-alert__content {  
    color: rgba(0, 0, 0, 0.5);  
}  
  
.simple-alert__dismiss {  
    display: block;  
    position: absolute;  
    top: 0;  
    right: 0;  
    bottom: 0;  
    margin: auto 0;  
    width: 20px;  
    height: 20px;  
    background-image: url("../img/close.png");  
    font-size: 0;  
    text-indent: -9999px;  
}
```

These styles should now give us a nice aesthetic that's pleasing to the user and representative of the various message types. Now, without further ado, let's get into some JavaScript!

A Quick Re-Introduction

As before, we'll wrap our component in a self-executing function. Note the ";" before our function, which is a precaution in case of auto-concatenated JavaScript files. Also note that we pass in `window` in order to add our Simple Alert component to the global namespace. Here's our function wrapper:

CODE

```
; ( function( window ) {

  'use strict';

})( window );
```

If you remember part 1, you'll remember how we started out our component. If not, it was as easy as defining `SimpleAlert` as a new function, and adding it to the global namespace. Here's how our JS should look now:

CODE

```
; ( function( window ) {

  'use strict';

  /**
   * SimpleAlert
   */
  function SimpleAlert( options ) {
    // function body...
  }

  /**
   * Add to global namespace
   */
  window.SimpleAlert = SimpleAlert;

})( window );
```

Component Building & Extending

Now, let's get our feet into some new stuff. As I mentioned before, we want to be able to pass in some options to our new instances of our object. In the demo, three options are available:

1. wrapper - the wrapper to append alerts to
2. type - the type of alert
3. message - the alert message

So we want to be able to accept an options object that looks like this:

CODE

```
options = {
  wrapper : document.body,
  type : "default",
  message : "Default message."
}
```

Note that I've set some default options above. This is interesting though, because we actually want default options inside our object in case the user decides to keep some of the defaults and only set one or two options. This brings us to another issue, however. How do we see if the user has passed in no options, some options, or all options, and override them where necessary? For this, we have an "extend object" function. It looks like this:

```
function extend( a, b ) {
  for( var key in b ) {
    if( b.hasOwnProperty( key ) ) {
      a[key] = b[key];
    }
  }
  return a;
}
```

Don't get all confused! This function is simple, and performs three main things:

1. Firstly, it accepts two objects as parameters.
2. Secondly, it loops through the second object, `b`, and looks at each `key` in it. It checks to see if `b` has that property by calling on the method `hasOwnProperty`, and you can view documentation on that method [here](#).
3. Finally, it updates and sets the corresponding `key` in the other object, `a`, and then when the loop is complete, it returns object `a`.

This is all well and good, but what exactly do we pass into this `extend` function for our options object to be ready for use in our component? We want to follow a couple steps to get there. So here's what we need to do:

1. We want `this.options` accessible in our entire component, so we first update it to be equal to our defaults. To do this, we pass in an empty object as `a` and the default options (initially represented by `this.options`) as `b` in our `extend` function.
2. We then extend `this.options` with our passed in object `options`, overriding the defaults where specified.

Our component, from start to finish, now looks something like this:

```

;( function( window ) {

    'use strict';

    /**
     * Extend obj function
     *
     * This is an object extender function. It allows us to extend an object
     * by passing in additional variables and overwriting the defaults.
     */
    function extend( a, b ) {
        for( var key in b ) {
            if( b.hasOwnProperty( key ) ) {
                a[key] = b[key];
            }
        }
        return a;
    }

    /**
     * SimpleAlert
     *
     * @param {object} options - The options object
     */
    function SimpleAlert( options ) {
        this.options = extend( {}, this.options );
        extend( this.options, options );
        // start the functionality...
    }

    /**
     * SimpleAlert options Object
     *
     * @type {HTMLElement} wrapper - The wrapper to append alerts to.
     * @param {string} type - The type of alert.
     * @param {string} message - The alert message.
     */
    SimpleAlert.prototype.options = {
        wrapper : document.body,
        type : "default",
        message : "Default message."
    }

    /**
     * Add to global namespace
     */
    window.SimpleAlert = SimpleAlert;

})( window );

```

Adding The Actual Functionality

Now that we're all set up, we need to add some actual functionality. Firstly, after a new instance of the object is instantiated, we want to run some code. I'll separate this first chunk of code into an `_init()` function. The reason for underscoring these functions is just a matter of taste. It doesn't actually do anything special, but lets other developers, or implementers of the component, know that this function should be reserved for private use inside the

component. Here's what we want our `_init()` function to do for us:

1. We want to create our actual alert. We do this by creating a `div` element, and adding the correct class names to it.
2. We want to then construct our inner HTML, i.e. the alert message and dismiss-alert button, and give that inner HTML to our newly created `div`.
3. We then want to listen for events on the dismiss button, and have some functions for showing and dismissing the alert

Initial Function

Our `_init()` function now should look like this:

```
/*
 * SimpleAlert _init
 *
 * This is the initializer function. It builds the HTML and gets the alert
 * ready for showing.
 *
 * @type {HTMLElement} this.sa - The Simple Alert div
 * @param {string} strinner - The inner HTML for our alert
 */
SimpleAlert.prototype._init = function() {
    // create element
    this.sa = document.createElement('div');
    this.sa.className = 'simple-alert ' + this.options.type;

    // create html
    var strinner = '';
    strinner += '<span class="simple-alert__content">';
    strinner += this.options.message;
    strinner += '</span>';
    strinner += '<a href="#" class="simple-alert__dismiss">close</a>';
    this.sa.innerHTML = strinner;

    // run the events
    this._events();
};

CODE
```

Note the usage of `this.options.type` and `this.options.message` which are of course, inherited values from our `this.options` object that we set up before. Also note that at the end of this function, we're calling on a new function, `_events()`, which will handle our events.

Events Function

The `_events()` function takes care of dismissing our alert when a user clicks the dismiss button. It's a simple function, and here's what it looks like:

CODE

```
/**
 * SimpleAlert _events
 *
 * This is our events function, and its sole purpose is to listen for
 * any events inside our Simple Alert.
 *
 * @type {HTMLElement} btn_dismiss - The dismiss-alert button
 */
SimpleAlert.prototype._events = function() {
    // cache vars
    var btn_dismiss = this.sa.querySelector('.simple-alert__dismiss'),
        self = this;

    // listen for dismiss
    btn_dismiss.addEventListener( "click", function(e) {
        e.preventDefault();
        self.dismiss();
    });
}
```

Notice that we set `var self = this` at the start of the function. This is because we want to access that instance of `this` inside the click event, but once inside the click event, `this` refers to the node that was clicked. Setting `this` to `self` before hand avoids any confusion. Note the call on the `dismiss()` function from before.

The Show Function

The `show()` function does at its name says - it shows the alert. This is a very basic function, and just appends our newly created `div` in the `_init()` function to the wrapper passed in to the `options` object. Here's what the function looks like:

CODE

```
/**
 * SimpleAlert show
 *
 * This function simply shows our Simple Alert by appending it
 * to the wrapper in question.
 */
SimpleAlert.prototype.show = function() {
    this.options.wrapper.appendChild(this.sa);
}
```

The Dismiss Function

The `dismiss()` function is equally as basic as the `show()` function, and simply removes the alert from the wrapper. Here's the function:

```
/**  
 * SimpleAlert dismiss  
 *  
 * This function simply hides our Simple Alert by removing it  
 * from the wrapper in question.  
 */  
SimpleAlert.prototype.dismiss = function() {  
    this.options.wrapper.removeChild(this.sa);  
};
```

Why Separate Two Simple Functions?

Surely, the functionality of `show()` and `dismiss()` could have all happened inside the `_init()` function, but there's a reason we separate them.

- We want to instantiate our alert, and show the alert whenever we want. In a more complex component, adding a "public method" like `show()` will allow us to show the actual alert whenever we want, after it is created.
- In the case of dismissing the alert, we added an event listener to the dismiss button which is default behaviour. What if an implementer of the component wanted to add his/her own functionality, and allow dismissing the alerts by clicking on a custom button?
- Finally, by doing this, we can (if we wanted) add two callbacks as part of our options object, and run the callbacks on alert creation, and alert dismissal. We would run these callbacks inside each of the functions `show()` and `dismiss()` respectively.

The Final Component Code

Here at last, a look at the final component code:

```
/**  
 * Simple Alert  
 *  
 * This little function allows us to display alerts to the user. The alerts  
 * append to a wrapper of choice, and takes on two other variables.  
 *  
 * Licensed under the MIT license.  
 * http://www.opensource.org/licenses/mit-license.php  
 *  
 * Copyright 2014, Call Me Nick  
 * http://callmenick.com  
 */  
  
;( function( window ) {  
  
    'use strict';  
  
    /**  
     * Extend obj function  
     *  
     * This is an object extender function. It allows us to extend an object  
     * by passing in additional variables and overwriting the defaults.  
     */  
    function extend( a, b ) {  
        for( var key in b ) {  
            if( b.hasOwnProperty( key ) ) {  
                a[key] = b[key];  
            }  
        }  
        return a;  
    }  
  
    /**  
     * SimpleAlert  
     *  
     * @param {object} options - The options object  
     */  
    function SimpleAlert( options ) {  
        this.options = extend( {}, this.options );  
        extend( this.options, options );  
        this._init();  
    }  
  
    /**  
     * SimpleAlert options Object  
     *  
     * @type {HTMLElement} wrapper - The wrapper to append alerts to.  
     * @param {string} type - The type of alert.  
     * @param {string} message - The alert message.  
     */  
    SimpleAlert.prototype.options = {  
        wrapper : document.body,  
        type : "default",  
        message : "Default message."  
    }  
  
    /**  
     * SimpleAlert _init  
     *  
     * This is the initializer function. It builds the HTML and gets the alert  
    */
```

```
* ready for showing.  
*  
* @type {HTMLElement} this.sa - The Simple Alert div  
* @param {string} strinner - The inner HTML for our alert  
*/  
SimpleAlert.prototype._init = function() {  
    // create element  
    this.sa = document.createElement('div');  
    this.sa.className = 'simple-alert ' + this.options.type;  
  
    // create html  
    var strinner = '';  
    strinner += '<span class="simple-alert__content">';  
    strinner += this.options.message;  
    strinner += '</span>';  
    strinner += '<a href="#" class="simple-alert__dismiss">close</a>';  
    this.sa.innerHTML = strinner;  
  
    // run the events  
    this._events();  
};  
  
/**  
 * SimpleAlert _events  
 *  
 * This is our events function, and its sole purpose is to listen for  
 * any events inside our Simple Alert.  
 *  
 * @type {HTMLElement} btn_dismiss - The dismiss-alert button  
 */  
SimpleAlert.prototype._events = function() {  
    // cache vars  
    var btn_dismiss = this.sa.querySelector('.simple-alert__dismiss'),  
        self = this;  
  
    // listen for dismiss  
    btn_dismiss.addEventListener( "click", function(e) {  
        e.preventDefault();  
        self.dismiss();  
    });  
}  
  
/**  
 * SimpleAlert show  
 *  
 * This function simply shows our Simple Alert by appending it  
 * to the wrapper in question.  
 */  
SimpleAlert.prototype.show = function() {  
    this.options.wrapper.appendChild(this.sa);  
}  
  
/**  
 * SimpleAlert dismiss  
 *  
 * This function simply hides our Simple Alert by removing it  
 * from the wrapper in question.  
 */  
SimpleAlert.prototype.dismiss = function() {  
    this.options.wrapper.removeChild(this.sa);  
};
```

```
/**
 * Add to global namespace
 */
window.SimpleAlert = SimpleAlert;

})( window );
```

An Example Of Execution

Now that our component is complete, let's take a look at an example execution.

CODE

```
<button id="simple-alert__button--error">Show Me An Error Alert</button>
<section class="demo-section" id="alerts"></section>

<script>
( function() {
  // cache vars
  var btn_error = document.getElementById("simple-alert__button--error");

  // show error
  btn_error.addEventListener( "click", function(e) {
    e.preventDefault();
    var sa = new SimpleAlert({
      wrapper : document.getElementById("alerts"),
      type : "error",
      message : "Show me an error!"
    });
    sa.show();
  });
})();
</script>
```

Simple and easy.

Wrap Up

And that's a wrap! In this tutorial, we dove deep into the creation of a JavaScript component, object extending, instantiating, and executing. The results may be simple, but it is a great insight into the wonderful world of JavaScript OOP, and a great starting point to building your own components. Don't forget, you can view the demo and download the source by clicking the links below, and if you have any questions, comments, or feedback, you can also leave them below. Also, you can [check out this project on GitHub](#) and fiddle as you see fit.

[Get Source](#) [View Demo](#)

44. Simple Inheritance with JavaScript

This article is part of a web dev tech series from Microsoft. Thank you for supporting the partners who make SitePoint possible.

A lot of my friends are C# or C++ developers. They are used to using inheritance in their projects and when they want

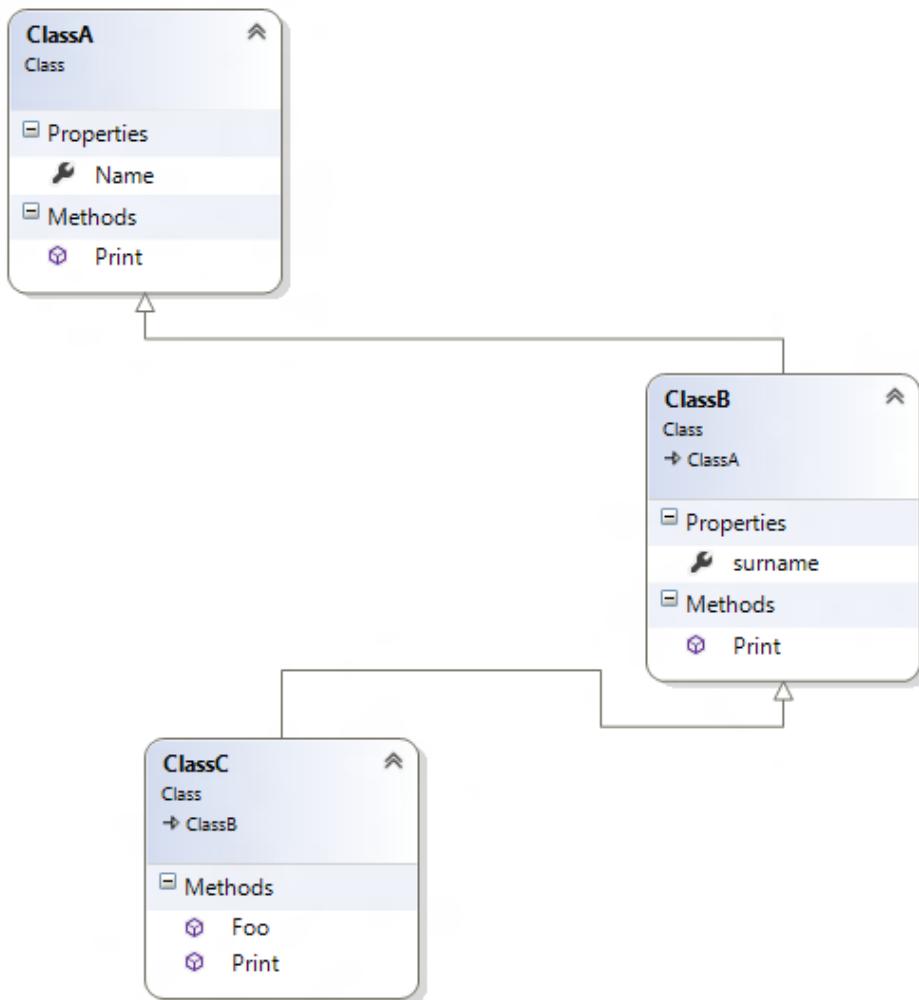
to learn or discover JavaScript, one of the first question they ask is: "But how can I do inheritance with JavaScript?"

Actually, JavaScript uses a different approach than C# or C++ to create an object-oriented language. It is a prototype-based language. The concept of prototyping implies that behavior can be reused by cloning existing objects that serve as prototypes. Every object in JavaScript descends from a prototype which defines a set of functions and members that the object can use. There is no class. Just objects. Every object can then be used as a prototype for another object.

This concept is extremely flexible and we can use it to simulate some concepts from OOP like inheritance.

Implementing Inheritance

Let's visualize what we want to create with this hierarchy using JavaScript:



First of all, we can create ClassA easily. Because there are no explicit classes, we can define a set of behavior (A class so...) by just creating a function like this:

```

var ClassA = function() {
    this.name = "class A";
}
  
```

CODE

This "class" can be instantiated using the `new` keyword:

CODE

```
var a = new ClassA();
ClassA.prototype.print = function() {
    console.log(this.name);
}
```

And to use it using our object:

CODE

```
a.print();
```

Fairly simple, right?

The complete sample is just 8 lines long:

CODE

```
var ClassA = function() {
    this.name = "class A";
}

ClassA.prototype.print = function() {
    console.log(this.name);
}

var a = new ClassA();

a.print();
```

Now let's add a tool to create "inheritance" between classes. This tool will just have to do one single thing: clone the prototype:

CODE

```
var inheritsFrom = function (child, parent) {
    child.prototype = Object.create(parent.prototype);
};
```

This is exactly where the magic happens! By cloning the prototype, we transfer all members and functions to the new class.

So if we want to add a second class that will be a child of the first one, we just have to use this code:

CODE

```
var ClassB = function() {
    this.name = "class B";
    this.surname = "I'm the child";
}

inheritsFrom(ClassB, ClassA);
```

Then because `ClassB` inherited the `print` function from `ClassA`, the following code is working:

CODE

```
var b = new ClassB();
b.print();
```

And produces the following output:

CODE

```
class B
```

We can even override the `print` function for `ClassB`:

CODE

```
ClassB.prototype.print = function() {
    ClassA.prototype.print.call(this);
    console.log(this.surname);
}
```

In this case, the produced output will look like this:

CODE

```
class B
I'm the child
```

The trick here is to the call `ClassA.prototype` to get the base `print` function. Then thanks to `call` function we can call the base function on the current object (`this`).

Creating `ClassC` is now obvious:

CODE

```
var ClassC = function () {
    this.name = "class C";
    this.surname = "I'm the grandchild";
}

inheritsFrom(ClassC, ClassB);

ClassC.prototype.foo = function() {
    // Do some funky stuff here...
}

ClassC.prototype.print = function () {
    ClassB.prototype.print.call(this);
    console.log("Sounds like this is working!");
}

var c = new ClassC();
c.print();
```

And the output is:

```
class C  
I'm the grandchild  
Sounds like this is working!
```

Or our team's learning series:

- [Practical Performance Tips to Make your HTML/JavaScript Faster](#) (a 7-part series from responsive design to casual games to performance optimization)
- [The Modern Web Platform JumpStart](#) (the fundamentals of HTML, CSS, and JS)
- [Developing Universal Windows Apps with HTML and JavaScript JumpStart](#) (use the JS you've already created to build an app)

And Some Philosophy...

To conclude, I just want to clearly state that JavaScript is not C# or C++. It has its own philosophy. If you are a C++ or C# developer and you really want to embrace the full power of JavaScript, the best tip I can give you is: Do not try to replicate your language into JavaScript. There is no best or worst language. Just different philosophies!

This article is part of the web dev tech series from Microsoft. We're excited to share [Project Spartan](#) and its [new rendering engine](#) with you. Get free virtual machines or test remotely on your Mac, iOS, Android, or Windows device at [modern.IE](#).

45. JavaScript inheritance patterns An overview and comparison

JavaScript inheritance patterns

An overview and comparison

JavaScript is a very powerful language. So powerful, in fact, that there are multiple different ways of designing prototypes and instantiating objects. There are tradeoffs when using each different method, and I aim to assist newcomers to the language by clearing up the mess. This is a follow-up to my previous post, [Stop Classifying JavaScript](#). I received many comments and responses asking for code examples, so here they are.

JavaScript has *Prototypal Inheritance*

This means that, in JavaScript, objects inherit from other objects. Basic objects in JavaScript, created with the {} curly braces, have only one prototype: `Object.prototype`. `Object.prototype` is, in itself, an object, and all members of `Object.prototype` are accessible from all objects.

Basic arrays, created with the [] square brackets, have multiple prototypes, including `Object.prototype` and `Array.prototype`. This means that all members of `Object.prototype` and all members of `Array.prototype` are accessible as members of arrays. Any members that overlap, like `.valueOf` and `.toString`, are overridden by the closest prototype, `Array.prototype` in this case.

Prototype definitions and object instantiation

Method 1: Constructor pattern

JavaScript has a special type of function called constructor functions, which act similarly to constructors in other languages. They are called mandatorily with the *new* keyword and bind the *this* keyword to the object being created by the constructor function. A typical constructor may look like this:

```
function Animal(type){
  this.type = type;
}
Animal.isAnimal = function(obj, type){
  if(!Animal.prototype.isPrototypeOf(obj)){
    return false;
  }
  return type ? obj.type === type : true;
};
```

CODE

```
function Dog(name, breed){
  Animal.call(this, "dog");
  this.name = name;
  this.breed = breed;
}
Object.setPrototypeOf(Dog.prototype, Animal.prototype);
Dog.prototype.bark = function(){
  console.log("ruff, ruff");
};
Dog.prototype.print = function(){
  console.log("The dog " + this.name + " is a " + this.breed);
};
```

CODE

```
Dog.isDog = function(obj){
  return Animal.isAnimal(obj, "dog");
};
```

CODE

The usage of this constructor looks like instantiation in other languages:

```
var sparkie = new Dog("Sparkie", "Border Collie");
```

CODE

```
sparkie.name; // "Sparkie"
sparkie.breed; // "Border Collie"
sparkie.bark(); // console: "ruff, ruff"
sparkie.print(); // console: "The dog Sparkie is a Border Collie"
```

CODE

```
Dog.isDog(sparkie); // true
```

CODE

bark and *print* are prototype methods which apply to all dogs. The *name* and *breed* properties are own properties that are set by the constructor. Usually, all methods are set in the prototype and all properties are set by the constructor.

Method 2: ES2015 (ES6) Class definitions

`class` has been a reserved keyword in JavaScript since the beginning, and now there is finally a use for it. Class definitions in JavaScript look a lot like they do in other languages.

```
class Animal {
  constructor(type){
    this.type = type;
  }
  static isAnimal(obj, type){
    if(!Animal.prototype.isPrototypeOf(obj)){
      return false;
    }
    return type ? obj.type === type : true;
  }
}
```

CODE

```
class Dog extends Animal {
  constructor(name, breed){
    super("dog");
    this.name = name;
    this.breed = breed;
  }
  bark(){
    console.log("ruff, ruff");
  }
  print(){
    console.log("The dog " + this.name + " is a " + this.breed);
  }
  static isDog(obj){
    return Animal.isAnimal(obj, "dog");
  }
}
```

CODE

A lot of people like this syntax because it combines the constructor, static, and the prototype method declarations into one nice block. The usage is exactly the same as the Constructor method.

```
var sparkie = new Dog("Sparkie", "Border Collie");
```

CODE

Method 3: Explicit prototype declaration, `Object.create`, method factory

This method really displays the prototypal inheritance behind the workings of the `class` syntax, and allows for the omission of the `new` keyword.

CODE

```

var Animal = {
  create(type){
    var animal = Object.create(Animal.prototype);
    animal.type = type;
    return animal;
  },
  isAnimal(obj, type){
    if(!Animal.prototype.isPrototypeOf(obj)){
      return false;
    }
    return type ? obj.type === type : true;
  },
  prototype: {}
};

```

CODE

```

var Dog = {
  create(name, breed){
    var dog = Object.create(Dog.prototype);
    Object.assign(dog, Animal.create("dog"));
    dog.name = name;
    dog.breed = breed;
    return dog;
  },
  isDog(obj){
    return Animal.isAnimal(obj, "dog");
  },
  prototype: {
    bark(){
      console.log("ruff, ruff");
    },
    print(){
      console.log("The dog " + this.name + " is a " + this.breed);
    }
  }
};

```

CODE

```
Object.setPrototypeOf(Dog.prototype, Animal.prototype);
```

This syntax is nice because the prototypes are very explicitly defined. It is very clear exactly which are members of the prototype and which are members of the object. `Object.create` is nice because it allows the creation of an object with a specific prototype. The `.isPrototypeOf` check still works in both cases. The usage is different, but not incredible different:

CODE

```
var sparkie = Dog.create("Sparkie", "Border Collie");
```

CODE

```

sparkie.name; // "Sparkie"
sparkie.breed; // "Border Collie"
sparkie.bark(); // console: "ruff, ruff"
sparkie.print(); // console: "The dog Sparkie is a Border Collie"

```

```
Dog.isDog(sparkie); // true
```

CODE

Method 4: Object.create, top-level factory, prototype post-declaration

This method is a slight variation of Method 3, where the factory is the class, versus the class being an object with a factory method. It looks like the constructor example (Method 1), but uses factories and `Object.create` instead.

```
function Animal(type){
  var animal = Object.create(Animal.prototype);
  animal.type = type;
  return animal;
}
Animal.isAnimal = function(obj, type){
  if(!Animal.prototype.isPrototypeOf(obj)){
    return false;
  }
  return type ? obj.type === type : true;
};
Animal.prototype = {};
```

CODE

```
function Dog(name, breed){
  var dog = Object.create(Dog.prototype);
  Object.assign(dog, Animal("dog"));
  dog.name = name;
  dog.breed = breed;
  return dog;
}
Dog.isDog = function(obj){
  return Animal.isAnimal(obj, "dog");
};
Dog.prototype = {
  bark(){
    console.log("ruff, ruff");
  },
  print(){
    console.log("The dog " + this.name + " is a " + this.breed);
  }
};
```

CODE

```
Object.setPrototypeOf(Dog.prototype, Animal.prototype);
```

CODE

This method is nice because it's usage looks like Method 1, but does not require the `new` keyword and works with `instanceof`. The usage is the same as the first method, but without the `new`:

```
var sparkie = Dog("Sparkie", "Border Collie");
```

CODE

CODE

```
sparkie.name; // "Sparkie"
sparkie.breed; // "Border Collie"
sparkie.bark(); // console: "ruff, ruff"
sparkie.print(); // console: "The dog Sparkie is a Border Collie"
```

CODE

```
Dog.isDog(sparkie); // true
```

Comparison

Method 1 vs Method 4

There is very little reason to use Method 1 over Method 4. Method 1 requires either the use of *new* in your code or a check like this in the constructor:

```
if(!(this instanceof Foo)){
    return new Foo(a, b, c);
}
```

CODE

At which point you may as well just use *Object.create* in a factory. You also can't use *Function#call* or *Function#apply* on constructor functions, because they mess up *this*. The check above could remedy that issue as well, but if you want to use an unknown amount of arguments, you have to use a factory.

Method 2 vs Method 3

The same arguments about constructors and *new* that applied above apply to this as well. The *instanceof* check is necessary for using the *class* syntax without *new* or with *Function#apply* and *Function#call*.

My opinion

A programmer should strive for code clarity. Method 3's explicit syntax very clearly shows exactly what is going on. It also allows for easy multiple inheritance and concatenative inheritance. Since using the *new* keyword violates the open-closed principle due to incompatibility with *apply* or *call*, it should be avoided. The *class* keyword hides the prototypal nature of JavaScript's inheritance behind the guise of a classical system.

"Simple is better than clever," and using classical syntax because it is considered to be more "sophisticated" is just unnecessary, technical overhead.

Object.create is more expressive and clearer than a bound *this* variable and *new*. Also, the prototype is stored in an object possibly outside the scope of the factory itself, so it can be modified and improved more easily and with method definition syntax, just like ES6 classes.

The class keyword will probably be the most harmful feature in JavaScript. I have enormous respect for the brilliant and hard-working people who have been involved in the standardization effort, but even brilliant people occasionally do the wrong thing.â€â€Eric Elliott

Adding something which is unnecessary, possibly damaging, and counter to the very nature of the language is a bad move.

If you choose to use `class`, I hope I never have to try to maintain your code. In my opinion, developers should avoid the use of constructors, `class`, and `new`, and use methods of inheritance which more closely follow the language architecture.

Glossary

`Object.assign(a, b)` copies all enumerable properties of object `b` onto object `a` and then returns object `a`

`Object.create(proto)` creates a new bare object with the prototype `proto`

`Object.setPrototypeOf(obj, proto)` sets the internal `[[Prototype]]` property of `obj` to `proto`

46. Understanding JavaScript Inheritance

So someone shoulder-taps you and asks you to explain the concepts behind JavaScript Inheritance to them. In my eyes you've got a few options.

The Terminology Play

You mention that it's **prototypal** inheritance, not **prototypical** and pretty much gloss over the rest, comfortable in your superiority in terminology. You may go as far as saying "Objects just come from other Objects because there aren't any classes." Then you just link to [Crock's Post](#) on it, and try to seem busy for the next few days.

Many years later you find out that **Prototypal** and **Prototypical** are synonyms, but you choose to ignore this.

The Like-Classical-Inheritance-But-Different Play aka the Run-On Sentence Play

"So in Java, like, you have classes or whatever, right? Well so imagine that you don't have those, but you still want to do that same type of thing or whatever, so then you just take another object instead of a class and you just kind of use it like it's a class, but it's not because it can change and it's just a normal object, and if it changes and you don't override the object, oh yea, so you can decide to override the parent object class thing, so if you dont do that and the parent changes the link is live..."

And so forth.

The Animal Play

This is a pretty popular one.

So let's say we want to make an `Animal` class in our code. As is often necessary in production JavaScript applications.

First we make a "constructor function," which acts kind of like a constructor method on the inside of a class in a classical language when it's invoked with the `new` operator. Except this one is on the outside.

CODE

```
function Animal (name) {
  this.name = name;
}

var myAnimal = new Animal('Annie');
```

Then we want to have actions that all animals can do.

CODE

```
Animal.prototype.walk = function () {
  console.log(this.name + ' is walking.');
};
```

But then you want to define a more specific *type* of animal. Things start to get weird.

CODE

```
// I think we need to define a new Animal type and extend from it somehow

function Dog (name) {
  this.name = name;
}

// BUT HOW DO WE EXTEND
// WITHOUT AN INSTANCE TO USE?
Dog.prototype = Animal.prototype; // ?? I HAVE NO IDEA
// Maybe that'll work for some stuff?
// ProHint: probably not much, once you start modifying one of them :D
```

Then you remember that Prototypal Inheritance doesn't really do 'classes' so much. So you do something like this:

CODE

```
var Dog = new Animal('Annie'); // ??? NO THATS NOT IT >:(

// Maybe we can try Object.create? I hear it's prototypal-y
var Dog = Object.create(Animal);

// Maybe that worked? Let's see...
var myDog = new Dog('Sparky');
// TypeError: object is not a function

// Shucks
```

And you eventually simply converge on the...

The Father/Son Analogy Play

Here we go. Finally a **real world** example of 'instances begetting instances.' It'll be a perfect analogy. It's even an interview question some places. Let's see how we might implement the relationship of a father and son (or a parent to its child) in JavaScript.

We'll start out like we did before, with a Human constructor

CODE

```
function Human( name ) {
  this.name = name;
}
```

Then we'll add in a common human shared action.

CODE

```
Human.prototype.sayHi = function () {
  console.log("Hello, I'm " + this.name);
};
```

So we'll create my dad first.

CODE

```
// Instantiate him
var myDad = new Human('Bill Sexton');

// Greet him
myDad.sayHi();
// "Hello, I'm Bill Sexton"
```

Score. Now let's create me.

CODE

```
// Let's use ES5 `object.create` in order to be as 'prototypal' as possible.
var me = Object.create(myDad);
me.sayHi();
// "Hello, I'm Bill Sexton"
```

It's a start! Seems like I inherited a little too much from my dad, but I inherited, none the less.

Let's try to smooth things out to make the analogy work better. So we'll instantiate objects without a name and have a parent name them after they're created.

CODE

```
// Wrap it all together
function makeBaby(parent, name) {
  // Instantiate a new object based on the parent
  var baby = Object.create(parent);

  // Set the name of the baby
  baby.name = name;

  // Give the baby away
  return baby;
}
```

Perfect. Now the baby can sayHi on its own.

CODE

```

var alex = makeBaby(myDad, 'Alex Sexton');

alex.sayHi();
// "Hello, I'm Alex Sexton"

```

Err. **yipes**. Babies can't talk. And what's this deal with a baby being made by **one** parent. Not to worry, we can fix all of this.

First we'll probably want to try to take two parents into the `makeBaby` function (no giggles).

COD		CODE
E 1	function makeBaby(father, mother, name){	
2	var baby = Object.create(...// fuuu	
3	}	

Multiple Inheritance! How did *you* get here? Ugh. Fine. We'll just simply mock the human chromosome pattern into our little inheritance example.

COD		CODE
E 1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		
16		
17	// Let's take a set of 4 genes for ease of	
18	// example here. We'll put them in charge	
19	// a few things.	
20	function Human (name, genes_mom, genes_dad) {	
21	this.name = name;	
22		
23	// Set the genes	
24	this.genes = {	
25	darkHair: this._selectGenes(genes_mom.darkHair, genes_dad.darkHair),	
26	smart: this._selectGenes(genes_mom.smart, genes_dad.smart),	
27	athletic: this._selectGenes(genes_mom.athletic, genes_dad.athletic),	
28	tall: this._selectGenes(genes_mom.tall, genes_dad.tall)	
29	};	

```
3      // Since genes affect you since birth we can set these as actual attrib
4      utes
5      this.attributes = {
6          darkHair: !!(~this.genes.darkHair.indexOf('D')),
7          smart: !!(~this.genes.smart.indexOf('D')),
8          athletic: !!(~this.genes.athletic.indexOf('D')),
9          tall: !!(~this.genes.tall.indexOf('D'))
10     };
11 }
12
13 // You don't have access to your own gene selection
14 // so we'll make this private (but in the javascript way)
15 Human.prototype._selectGenes = function (gene1, gene2) {
16     // Assume that a gene is a 2 length array of the following possibilite
17     s
18     // DD, Dr, rD, rr -- the latter being the only non "dominant" result
19
20     // Simple random gene selection
21     return [ gene1[Math.random() > 0.5 ? 1 : 0], gene2[Math.random() > 0.5
22 ? 1 : 0] ]
23 };
24
25 Human.prototype.sayHi = function () {
26     console.log("Hello, I'm " + this.name);
27 };
28
29 function makeBaby(name, mother, father) {
30     // Send in the genes of each parent
31     var baby = new Human(name, mother.genes, father.genes);
32     return baby;
33 }
34
35
36
37
38
39
40
41
42
```

Elementary. My only beef is that we no longer are using real prototypal inheritance. There is no live link between the parents and the child. If there was only one parent, we could use the `__proto__` property to set the parent as the prototype after the baby was instantiated. However we have two parents...

So we'll need to implement runtime getters that do a lookup for each parent via [ES Proxies](#).

```
CODE
1----- function makeBaby(name, mother, father) {
2      // Send in the genes of each parent
3      var baby = new Human(name, mother.genes, father.genes);
4
5      // Proxy the baby
6      return new Proxy(baby, {
7          get: function (proxy, prop) {
8              // shortcut the lookup
9              if (baby[prop]) {
10                  return baby[prop];
11              }
12
13              // Default parent
14              var parent = father;
15
16              // Spice it up
17              if (Math.random() > 0.5) {
18                  parent = mother;
19              }
20
21              // See if they have it
22              return parent[prop];
23          }
24      });
25 }
```

So now we support live lookups of parents, and, you know, some simplified genetics.

Isn't that just a simple, well-defined, example of how straightforward inheritance can be in JavaScript?

Conclusion

Sometimes these analogies get pretty crazy in my head, and I start to think that maybe instead of trying to apply known examples in the outside world in order to help people understand, it's often better to just let someone know why they might wanna use inheritance in their programs!

I personally find the best Prototypal Inheritance analogy to be:

```

CODE
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20

var defaults = {
  zero: 0,
  one: 1
};

var myOptions = Object.create(defaults);
var yourOptions = Object.create(defaults);

// When I want to change *just* my options
myOptions.zero = 1000;

// When you wanna change yours
yourOptions.one = 42;

// When we wanna change the **defaults** even after we've got our options
// even **AFTER** we've already created our instances
defaults.two = 2;

myOptions.two; // 2
yourOptions.two; // 2

```

So stop making everything so confusing and go program cool stuff, and ignore my old presentations when I used these analogies.

<3z

47. Javascript Inheritance Patterns: Functional and Psuedo-Classical

Introduction to Polymorphism

Polymorphism could be defined as an object's ability to manifest in different forms. The abstract notion of polymorphism manifest as the concrete application of subclassing. A class works great when you want to create a fleet on similar objects. Polymorphism allows us to subclass in order to augment different behavior onto different objects.

Functional Inheritance

Functional inheritance can be used with functional instantiation (aka: Factory Pattern) or shared functional instantiation. Functional inheritance is when you instantiate an object inside a function, and then augment that object with special properties before returning it.

CODE

```
function Shape(width, height) {
  var shape = {};
  shape.width = width;
  shape.height = height;
  return shape;
}

function Square(width, height) {
  var shape = Shape(width, height);
  shape.size = this.width * this.height;
  return shape;
}

square = Square(100, 100);
>>>Object {width: 100, height: 100, size: 10000}
```

We called the `Shape` factory and received a `Shape` object in return. We then augmented it with a `size` property and returned the augmented object.

Pseudo-classical Inheritance

Pseudo-classical inheritance is used with pseudo-classical instantiation (aka: Constructor Pattern). There are two different things that need to be inherited when using the pseudo-classical style. Firstly, you need to inherit properties from within the constructor. Secondly, you need to inherit properties from `ConstructorName.prototype`. Additionally, you also need to manually set the subclass's `constructor` property. The subclass inherits its prototype from the superclasses prototype, this works fine but it has the unfortunate side effect of destroying the reference the subclasses prototype constructor property has to itself. We need to set the subclasses `constructor` property manually as a result.

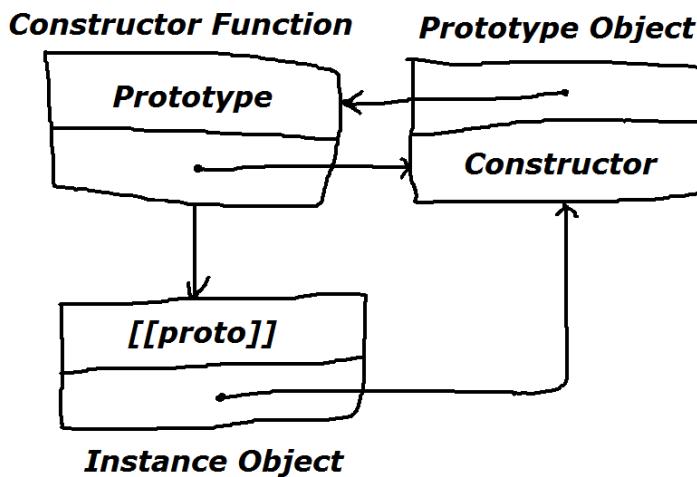
CODE

```
function Fruit(sweetness, freshness, organic) {
  this.sweetness = sweetness;
  this.freshness = freshness;
  this.organic = organic;
}

function Apple(sweetness, freshness, organic) {
  Fruit.call(this, sweetness, freshness, organic);
  this.color = "red";
  this.name = "apple";
}
```

Notice when we use pseudo-classical inheritance that the line `Fruit.call(this, sweetness, freshness, organic);` is what is doing the heavy lifting here. All we need to do is call the superclass `Fruit` from within subclass `Apple` for inheritance to occur. The reason we do not need to set any variables explicitly is because our

call to `Fruit` is occurring in the context of `Apple`. Therefore when `this` is used in superclass `Fruit` it is actually referring to the object that is being instantiated in `Apple` since `this` refers to the object being created when using the Constructor pattern.



As stated previously, on top of calling `Fruit.call(this, sweetness, freshness, organic)`; we also need to set inheritance in their Prototype chains correctly and set the subclasses constructor property manually on its prototype.

```
Apple.prototype = Object.create(Fruit.prototype);
Apple.prototype.constructor = Apple
```

CODE

If we do not manually set `Apple.prototype.constructor = Apple` then the `constructor` property will be lost due to inheritance overriding it. We would then get weird behavior when using functionality like `instanceof`.

48. Pseudo-classical pattern

1. [Pseudo-class declaration](#)
2. [Inheritance](#)
3. [Calling superclass constructor](#)
4. [Overriding a method \(polymorphism\)](#)
 1. [Calling a parent method after overriding](#)
 2. [Sugar: removing direct reference to parent](#)
5. [Private/protected methods \(encapsulation\)](#)
6. [Static methods and properties](#)
7. [Summary](#)

In pseudo-classical pattern, the object is created by a constructor function and its methods are put into the prototype.

Pseudo-classical pattern is used in frameworks, for example in Google Closure Library. Native JavaScript objects also follow this pattern.

Pseudo-class declaration

The term "*pseudo-class*" is chosen, because there are actually no classes in JavaScript, like those in C, Java, PHP etc. But the pattern is somewhat close to them.

The article assumes you are familiar with how the prototypal inheritance works.

That is described in the article [Prototypal inheritance](#).

A *pseudo-class* consists of the constructor function and methods.

For example, here's the Animal pseudo-class with single method `sit` and two properties.

CODE

```
function Animal(name) {
    this.name = name
}

Animal.prototype = {
    canWalk: true,
    sit: function() {
        this.canWalk = false
        alert(this.name + ' sits down.')
    }
}

var animal = new Animal('Pet') // (1)

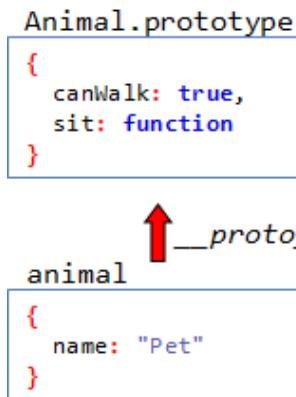
alert(animal.canWalk) // true

animal.sit()           // (2)

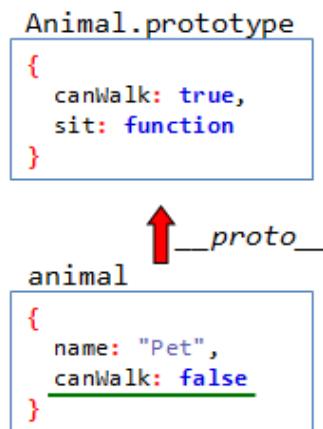
alert(animal.canWalk) // false
```

1. When `new Animal(name)` is called, the new object receives `__proto__` reference to `Animal.prototype`, see that on the left part of the picture.
2. Method `animal.sit` changes `animal.canWalk` in the instance, so now this animal object can't walk. But other animals still can.

Initially (1):



After step (2):



The scheme for a pseudo-class:

- Methods and default properties are in prototype.
- Methods in prototype use `this`, which is the *current object* because the value of `this` only depend on the calling context, so `animal.sit()` would set `this` to `animal`.

There are dangers in the scheme. See the task below.

You are a team lead on a hamster farm. A fellow programmer got a task to create Hamster constructor and prototype.

Hamsters should have a `food` storage and the `found` method which adds to it.

He brings you the solution (below). The code looks fine, but when you create two hamsters, then feed one of them - somehow, both hamsters become full.

What's up? How to fix it?

CODE

```
function Hamster() { }
Hamster.prototype = {
  food: [],
  found: function(something) {
    this.food.push(something)
  }
}

// Create two speedy and lazy hamsters, then feed the first one
speedy = new Hamster()
lazy = new Hamster()

speedy.found("apple")
speedy.found("orange")

alert(speedy.food.length) // 2
alert(lazy.food.length) // 2 (!???)
```

Solution

Let's get into details what happens in `speedy.found("apple")`:

1. The interpreter searches `found` in `speedy`. But `speedy` is an empty object, so it fails.
2. The interpreter goes to `speedy.__proto__` (`==Hamster.prototype`) and luckily gets `found` and runs it.
3. At the pre-execution stage, `this` is set to `speedy` object, because of dot-syntax: `speedy.found`.
4. `this.food` is not found in `speedy`, but is found in `speedy.__proto__.food`.
5. The "apple" is appended to `speedy.__proto__.food`.

Hamsters share the same belly! Or, in terms of JavaScript, the `food` is modified in `__proto__`, which is shared between all hamster objects.

Note that if there were a simple assignment in `found()`, like `this.food = something`, then step 4-5 would not lookup `food` anywhere, but assign `something` to `this.food` directly.

Fixing the issue

To fix it, we need to ensure that every hamster has its own belly. This can be done by assigning it in the constructor:

CODE

```

function Hamster() {
    this.food = []
}
Hamster.prototype = {
    found: function(something) {
        this.food.push(something)
    }
}

speedy = new Hamster()
lazy = new Hamster()

speedy.found("apple")
speedy.found("orange")

alert(speedy.food.length) // 2
alert(lazy.food.length) // 0(!)

```

Inheritance

Let's create a new class and inherit it from `Animal`.

Here you are.. A Rabbit!

CODE

```

function Rabbit(name) {
    this.name = name
}

Rabbit.prototype.jump = function() {
    this.canWalk = true
    alert(this.name + ' jumps!')
}

var rabbit = new Rabbit('John')

```

As you see, the same structure as `Animal`. Methods in prototype.

To inherit from `Animal`, we need `Rabbit.prototype.__proto__ == Animal.prototype`. This is a very natural requirement, because if a method is not find in `Rabbit.prototype`, it should be searched in the parental method store, which is `Animal.prototype`.

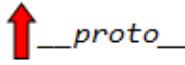
That's how it should look like:

Animal.prototype

```
{
  canWalk: true,
  sit: function
}
```

**Rabbit.prototype**

```
{
  jump: function
}
```

**rabbit**

```
{
  name: "John"
}
```

To implement the chain, we need to create initial `Rabbit.prototype` as an empty object inheriting from `Animal.prototype` and *then* add methods.

CODE

```
function Rabbit(name) {
  this.name = name
}

Rabbit.prototype = inherit(Animal.prototype)

Rabbit.prototype.jump = function() { ... }
```

'inherit'

In the code above, `inherit` is a function which creates an empty object with given `__proto__`.

CODE

```
function inherit(proto) {
  function F() {}
  F.prototype = proto
  return new F
}
```

See [Prototypal inheritance](#) for details.

And finally, the full code of two objects:

```
// Animal
function Animal(name) {
  this.name = name
}

// Animal methods
Animal.prototype = {
  canWalk: true,
  sit: function() {
    this.canWalk = false
    alert(this.name + ' sits down.')
  }
}

// Rabbit
function Rabbit(name) {
  this.name = name
}

// inherit
Rabbit.prototype = inherit(Animal.prototype)

// Rabbit methods
Rabbit.prototype.jump = function() {
  this.canWalk = true
  alert(this.name + ' jumps!')
}

// Usage
var rabbit = new Rabbit('Sniffer')

rabbit.sit() // Sniffer sits.
rabbit.jump() // Sniffer jumps!
```

Don't create new Animal to inherit it

There is a well-known, but *wrong* way of inheriting, when instead of `Rabbit.prototype = inherit(Animal.prototype)` people use:

```
// inherit from Animal
Rabbit.prototype = new Animal()
```

As a result, we get a new `Animal` object in `prototype`. Inheritance works here, because `new Animal` naturally inherits `Animal.prototype`.

... But who said that `new Animal()` can be called like without the `name`? The constructor may strictly require arguments and die without them.

Actually, the problem is more conceptual than that. **We don't want to create an `Animal`. We just want to inherit from it.**

That's why `Rabbit.prototype = inherit(Animal.prototype)` is preferred. The neat inheritance without side-effects.

Calling superclass constructor

The "superclass" constructor is not called automatically. We can call it manually by applying the `Animal` function to current object:

```
function Rabbit(name) {
    Animal.apply(this, arguments)
}
```

CODE

That executes `Animal` constructor in context of the current object, so it sets the `name` in the instance.

Overriding a method (polymorphism)

To override a parent method, replace it in the prototype of the child:

```
Rabbit.prototype.sit = function() {
    alert(this.name + ' sits in a rabbity way.')
}
```

CODE

A call to `rabbit.sit()` searches `sit` on the chain `rabbit -> Rabbit.prototype -> Animal.prototype` and finds it in `Rabbit.prototype` without ascending to `Animal.prototype`.

Of course, we can even more specific than that. A method can be overridden directly in the object:

```
rabbit.sit = function() {
    alert('A special sit of this very rabbit ' + this.name)
}
```

CODE

Calling a parent method after overriding

When a method is overwritten, we may still want to call the old one. It is possible if we directly ask parent prototype for it.

```
Rabbit.prototype.sit = function() {
    alert('calling superclass sit:')
    Animal.prototype.sit.apply(this, arguments)
}
```

CODE

All parent methods are called with `apply/call` to pass current object as `this`. A simple call `Animal.prototype.sit()` would use `Animal.prototype` as `this`.

Sugar: removing direct reference to parent

In the examples above, we call parent class directly. Either it's constructor: `Animal.apply...`, or methods: `Animal.prototype.sit.apply...`

Normally, we shouldn't do that. Refactoring may change parent name or introduce intermediate class in the hierarchy.

Usually programming languages allow to call parent methods using a special key word, like `parent.method()` or `super()`.

JavaScript doesn't have such feature, but we could emulate it.

The following function `extend` forms inheritance and also assigns `parent` and `constructor` to call parent without a direct reference:

```
function extend(Child, Parent) {
    Child.prototype = inherit(Parent.prototype)
    Child.prototype.constructor = Child
    Child.parent = Parent.prototype
}
```

CODE

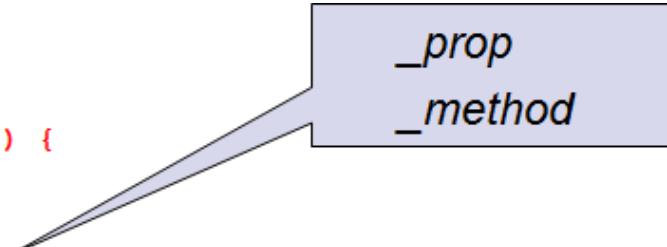
Usage:

```
function Rabbit(name) { Rabbit.parent.constructor.apply(this, arguments) // super constructor } extend(Rabbit,
Animal) Rabbit.prototype.run = function() { Rabbit.parent.run.apply(this, arguments) // parent method alert("fast") }
```

As the result, we can now rename `Animal`, or create an intermediate class `GrassEatingAnimal` and the changes will only touch `Animal` and `extend(...)`.

Private/protected methods (encapsulation)

Protected methods and properties are supported by naming convention. So, that a method, starting with underscore '`_`' should not be called from outside (technically it is callable).



```
function Animal(name) {
    this.name = name
}

Animal.prototype._doWalk = function() { // protected
    alert("running")
}

Animal.prototype.walk = function() { // public
    this._doWalk()
}
```

Private methods are usually not supported.

Static methods and properties

A static property/method are assigned directly to constructor:

CODE

```
function Animal() {  
    Animal.count++  
}  
Animal.count = 0  
  
new Animal()  
new Animal()  
  
alert(Animal.count) // 2
```

Summary

And finally, the whole suppa-mega-oop framework.

CODE

```
function extend(Child, Parent) {  
    Child.prototype = inherit(Parent.prototype)  
    Child.prototype.constructor = Child  
    Child.parent = Parent.prototype  
}  
function inherit(proto) {  
    function F() {}  
    F.prototype = proto  
    return new F  
}
```

Usage:

```
// ----- the base object -----
function Animal(name) {
    this.name = name
}

// methods
Animal.prototype.run = function() {
    alert(this + " is running!")
}

Animal.prototype.toString = function() {
    return this.name
}

// ----- the child object -----
function Rabbit(name) {
    Rabbit.parent.constructor.apply(this, arguments)
}

// inherit
extend(Rabbit, Animal)

// override
Rabbit.prototype.run = function() {
    Rabbit.parent.run.apply(this)
    alert(this + " bounces high into the sky!")
}

var rabbit = new Rabbit('Jumper')
rabbit.run()
```

Frameworks may add a bit more sugar, like function `mixin` which copies many properties from one object to another:

```
mixin(Animal.prototype, { run: ..., toString: ...})
```

But in fact you don't need much to use this OOP pattern. Just two tiny functions will do.

- [General concepts All-in-one constructor pattern](#) ▾

49. All-in-one constructor pattern

1. [Declaration](#)
2. [Inheritance](#)
3. [Overriding \(polymorphism\)](#)
4. [Private/protected methods \(encapsulation\)](#)
5. [Summary](#)
 1. [Comparison with pseudo-classical pattern](#)

All methods and properties of the object can be added in the constructor. This method doesn't use `prototype` at all.

Declaration

The object is declared solely by it's constructor. As an example, let's build the `Animal` with property `name` and method `run`:

```

1 function Animal(name) {
2   this.name = name
3   this.run = function() {
4     alert("running "+this.name)
5   }
6 }
7
8 var animal = new Animal('Foxie')
9 animal.run()

```

CODE

As you see, properties and methods are assigned to `this`. As the result, we have a full object.

Inheritance

To create a `Rabbit`, inheriting from `Animal`:

1. First apply `Animal` constructor to `this`. We've got an `Animal`
2. Modify `this`, add more methods to get a `Rabbit`.

For example:

```

function Rabbit(name) {
  Animal.apply(this, arguments) // inherit

  this.bounce = function() {
    alert("bouncing "+this.name)
  }
}

rabbit = new Rabbit("Rab")

rabbit.bounce() // own method

rabbit.run() // inherited method

```

CODE

The superclass constructor is called automatically when inhereting. There is no non-ugly way to first inherit, then do something, and then call the superclass constructor.

We can call constructor with custom parameters too:

CODE

```

1 function Rabbit(name) {
  Animal.call(this, "Mr. " + name.toUpperCase())
  // ..
}

rabbit = new Rabbit("Rab")

rabbit.run()

```

At the end of `new call` we always have a single object with both own and parent methods in it. The `prototype` is not used at all.

Overriding (polymorphism)

Overriding a parent method is as easy as overwriting it in `this`. Of course we may want to copy the old method and call it in the process.

CODE

```

01 function Rabbit(name) {

  Animal.apply(this, arguments)

  var parentRun = this.run // keep parent method

  this.run = function() {
    alert("bouncing "+this.name)
    parentRun.apply(this) // call parent method
  }
}

rabbit = new Rabbit("Rab")

rabbit.run() // inherited method

```

Here we use `apply` to provide right `this`.

Private/protected methods (encapsulation)

Private methods and properties are supported really good in this pattern.

A local function or variable are private. All constructor arguments are private automatically.

That's because all functions created in the scope of `Rabbit` can reference each other through closure, but the outside code can only access those assigned to `this`.

In the example below, `name` and `created` are private properties, used by private method `sayHi`:

CODE

```

01 function Rabbit(name) {
02
03   Animal.call(this, "Mr. " + name.toUpperCase())
04
05   var created = new Date() // private
06
07   function sayHi() { // private
08     alert("I'm talking rabbit " + name)
09   }
10
11   this.report = function() {
12     sayHi.apply(this)
13     alert("Created at " + created)
14   }
15 }
16
17 rabbit = new Rabbit("Rab")
18
19 rabbit.report()

```

It is quite inconvenient to use call methods through `apply`, like `sayHi.apply(this)`. So, the functions are usually bound to the object. Read more about that in [Early and Late Binding](#)

Local variables/functions become private, not protected. A child can't access them:

CODE

```

1 function Animal() {
2   var prop = 1
3 }
4
5 function Rabbit() {
6   Animal.call(this) // inherit
7   /* can't access prop from here */
8 }

```

Protected properties are implemented same way as in [pseudo-classical approach](#). That is, by naming convention: `"_prop"`.

CODE

```

01 function Animal() {
02   this._prop = 'test' // protected
03 }
04
05 function Rabbit() {
06   Animal.call(this) // inherit
07   alert(this._prop) // access
08 }
09
10 new Rabbit()

```

Summary

- The object is fully described by its constructor.
- Inheritance is done by calling the parent constructor in the context of current object.

- All local variables/functions become private, all assigned to `this` become public. Local functions are usually bound to the object.
- Protected properties can be prepended with underscore `'_'`, but their protection can't be forced on language level.
- Overriding is done as replacing the property in `this`. The old property may be copied and reused.

Comparison with pseudo-classical pattern

- `rabbit instanceof Animal` doesn't work here. That's because `Rabbit` does not inherit from `Animal` in prototype-sense.
- Slower creation, more memory for methods, because every object carries all methods in it, without prototype as shared storage. But on frontend programming we *shouldn't* create many objects. So that's not a big problem.
- Inheritance is joined with parent constructor call. That's architectural inflexibility, because one can't call parent method before parent constructor. Not so fearful though.
- Private methods and properties. That's safe and fast. Especially because JavaScript compressors shorten them.
- There are no "occasionally static" properties in prototype.

Actually, we have minor problems and advantages. Choose the method depending on how they refer to your application.

- [Pseudo-classical pattern](#) [Factory constructor pattern](#)

See also:

- [Early and Late Binding](#)

50. Extending Natives

1. [Modifying native prototypes](#)
2. [Inheriting from native objects](#)
3. [Method borrowing](#)
4. [Summary](#)

Native JavaScript objects store their methods in prototypes.

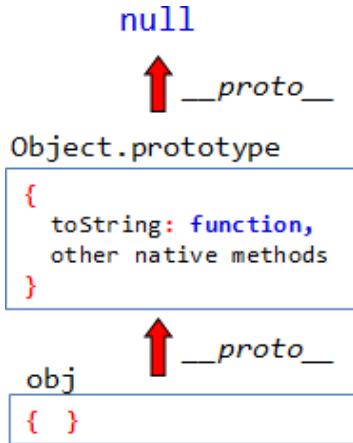
For example, When a new object is created, it doesn't have anything. Then how would `toString` work?

```
var obj = {}  
alert( obj.toString() )
```

CODE

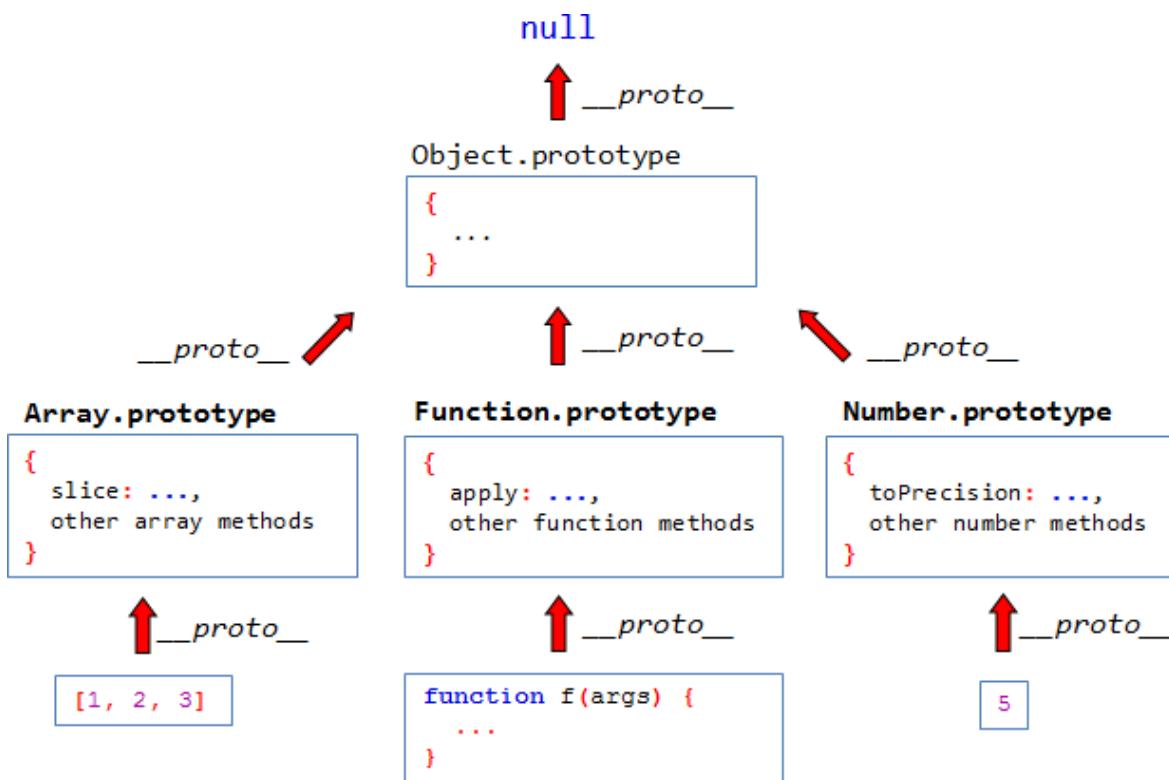
That's because `obj = {}` is a short for `obj = new Object`, where `Object` is a native JavaScript constructor function.

..And the object, made by `new Object` naturally receives `object.__proto__ == Object.prototype`, where `Object.prototype` is a native object with `Object.prototype.toString` property (and not only).



So, all properties of `Object.prototype` become available in objects. That's why all objects have `toString` method.

Same story with `Array`, `Function` and other objects. Their methods are in `Array.prototype`, `Function.prototype` etc.



When an object property is accessed, the interpreter searches it:

1. In the object itself,
2. Then in its `__proto__`,
3. Then in its `__proto__.__proto__` etc.

The chain is followed until the property is found or the next `__proto__ == null`.

The only object with `__proto__ = null` is `Object.prototype`. All chains eventually end at `Object.prototype` or, in other words, all objects inherit from `Object`.

Primitive values like `5` are implicitly converted to objects when a method is called, and the result is again a primitive.

Modifying native prototypes

Native prototypes can be modified. New methods can be introduced.

We could write a method to repeat a string many times (read [here](#new Array) about new Array syntax):

```
String.prototype.repeat = function(times) {
    return new Array(times+1).join(this)
}

alert( "a".repeat(3) ) // aaa
```

CODE

Same way we could add a method Object.prototype.each(func) to apply func to every property.

```
show clean source in new windowHide/show line numbersprint highlighted code
e.each = function(f) {
02   for(var prop in this) {
03     var value = this[prop]
04     f.call(value, prop, value) // calls f(prop, value), this=value
05   }
06 }
07
08 // Try it: (works wrong!)
09 var obj = { name: 'John', age: 25 }
10 obj.each(function(prop, val) {
11   alert(prop) // name -> age -> (!) each
12 })
```

CODE

As you can see from the comments, the code is wrong. It outputs extra each property, because for..in walks the prototype.

Fortunately, this functionality can be fixed if we add hasOwnProperty check to for..in .

```
show clean source in new windowHide/show line numbersprint highlighted code
e.each = function(f) {
02   for(var prop in this) {
03     if (Object.prototype.hasOwnProperty(prop)) continue // filter
04     var value = this[prop]
05     f.call(value, prop, value)
06   }
07 }
08
09 // Now correct
10 var obj = { name: 'John', age: 25 }
11 obj.each(function(prop, val) {
12   alert(prop) // name -> age
13 })
```

CODE

In this case it worked, now the code is correct. But generally we don't want to put hasOwnProperty in every loop...

New properties of Object.prototype appear in all for..in loops. Don't add them.

Native properties and `for..in`

Built-in properties and methods are not listed in for..in , because they have a special [[Enumerable]] flag set to

```
false .
```

In browsers which support modern JavaScript (IE from version 9), this flag can be set for custom properties too.

For other objects, which are not iterated in `for..in` loops (Strings , Functions etc), there are pro and contra against adding new methods:

- New methods allow to write shorter and cleaner code.
Compare `"a".repeat(5)` and `str_repeat("a", 5)`. The first looks better for most people.
- New properties may conflict. Imagine different people writing same named method in their libraries. Integration may be hard.

Prototypes are global for all your code, and modifying them is architecturally bad for same reason as introducing new global variables.

- But that's in theory. In practice, there is a well-known library [Prototype JS](#) which extends prototypes without terrible consequences



Modifying built-in prototypes is generally considered bad, but may be useful to implement modern ECMA-262 5th edition methods in older browsers.

For example, if there is no `Object.create`, then add it:

```
show clean source in new windowHide/show line numbersprint highlighted code 1 if (!Object.create)-----  
CODE  
{  
2  Object.create = function(proto) {  
3      function F() {}  
4      F.prototype = proto  
5      return new F  
6  }  
7 }
```

Inheriting from native objects

Native objects can be inherited. For example, `Array.prototype` keeps all methods for new `Array` instances.

If we want these methods be accessible in our `myArray`, then `myArray.__proto__ == Array.prototype` does the trick.

Let's build a function which creates such objects:

CODE

```

show clean source in new windowHide/show line numbersprint highlighted code 01 // constructor
02 function MyArray() {
03   this.stringify = function() {
04     return this.join(', ')
05   }
06 }
07 // Make sure that (new MyArray) has the right __proto__
08 MyArray.prototype = Array.prototype
09
10 // Test
11 var myArr = new MyArray()
12 myArr.push(1)
13 myArr.push(2)
14 alert( myArr.stringify() ) // 1, 2
15 alert( myArr.length ) // 2 in all except IE

```

That works great in all browsers except IE, which doesn't autoupdate the `length` property.

Method borrowing

If you want to use a piece of `Array` functionality, it is not required to inherit it.

Instead, you can *borrow* a method, and then *apply* it without inheriting:

CODE

```

var join = Array.prototype.join
// same but shorter
var join = [].join

```

.. And call it with a non-standard `this`:

CODE

```

show clean source in new windowHide/show line numbersprint highlighted code 1 var obj = {
2   0: 'first',
3   1: 'second',
4   length: 2
5 }
6
7 alert( [].join.call(obj, ', ') ) // first, second

```

Method `Array.prototype.join` is described in the specification ES-5, p.15.4.4.5. It doesn't check for object type. All it does is a joining loop over properties with indices `0..length-1`, so here it works fine.

`Array` methods are often borrowed for manipulating array-like objects.

Summary

- Methods of native objects are stored in their prototypes.
- Native prototypes can be inherited or extended.
- Modifying top-level `Object.prototype` breaks `for..in` loops. Other prototypes are less dangerous to modify, but most developers don't recommend it.

The notable exception is adding a support for modern methods for older browsers, like [Object.create](#),

[Object.keys](#), [Function.prototype.bind](#) and others.

- [Prototypal inheritance The "constructor" property](#) ↗

51. Object Oriented JavaScript Pattern Comparison

In this post, I explore various object oriented JavaScript design patterns. An intermediate level of JavaScript knowledge is required to get value from this post. Before we get started exploring the various patterns below, it's worth reviewing a couple key things about JavaScript objects.

The new Keyword in JavaScript

When the `new` keyword is placed in front of a function call, four things happen:

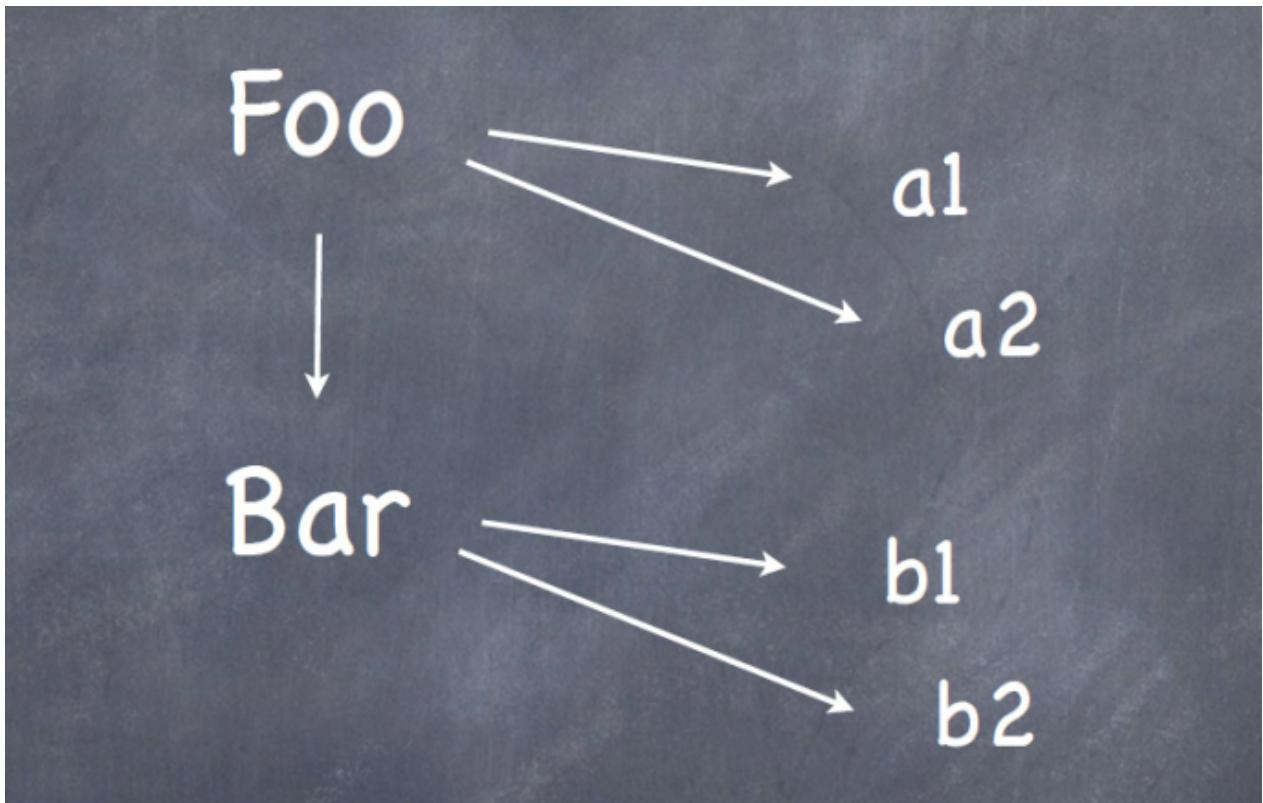
1. A new object gets created by the *constructor function*.
2. The new object gets linked to a different object.
3. The new object gets bound as the `this` keyword within the constructor function call.
4. If the constructor function does not return a value, JavaScript implicitly inserts `return this;` at the end of the constructor function's execution.

Prototypal Inheritance in JavaScript

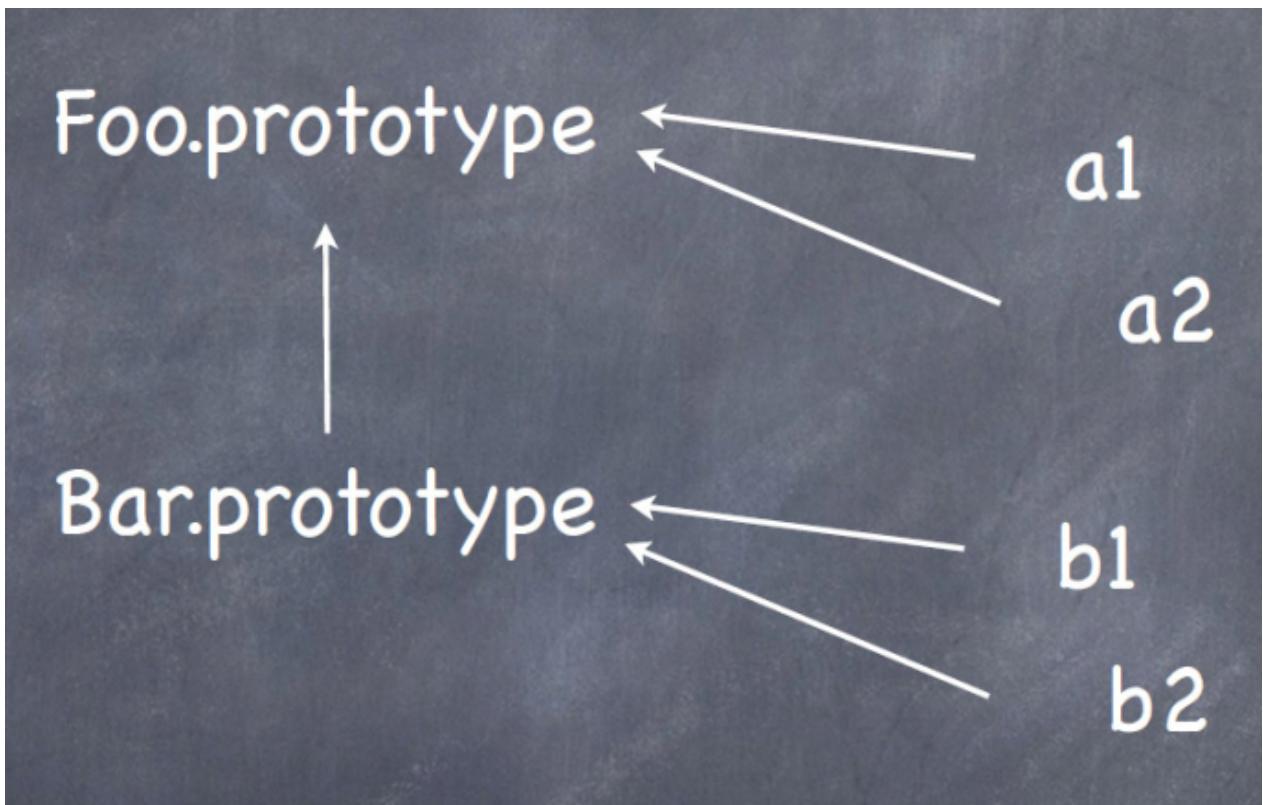
In JavaScript, the term "inheritance" in the form of prototypes is a misnomer. Prototypal Inheritance in JavaScript is much different than inheritance in traditional class-based languages, where a subclasses inherit from superclasses. I find it easiest to understand inheritance in JavaScript by abstracting the concept to human beings...

When your parents had you, you inherited their DNA — you received a *copy* of it. When they broke their leg, yours did not break. JavaScript is the opposite of this. In JavaScript, when your parents break their leg, yours breaks too. A term better suited than prototypal inheritance to JavaScript is prototypal *delegation*. When a new object is created from another object in JavaScript, it *links* back to the parent object's prototype properties and methods as opposed to copying them.

The following screenshots provide further clarification about how JavaScript manages inheritance differently than class-based languages.



Class Inheritance



Prototypal Inheritance

Screenshots taken from Kyle Simpson's [Advanced JavaScript course](#) on Frontend Masters

Factory Pattern

CODE

```
// -----
// JavaScript Factory Pattern
// -----



// factory function to create car objects
function createCar(make, model, year) {
    var o = new Object();

    o.make    = make;
    o.model   = model;
    o.year    = year;
    o.sayCar = function() {
        alert('I have a ' + this.year + ' ' + this.make + ' ' + this.model + '.');
    };

    return o;
}

// create 2 car objects for John and Jane
var johnCar = createCar('Ford', 'F150', '2011'),
    janeCar = createCar('Audi', 'A4', '2007');

// call method on Jane's car
janeCar.sayCar();
```

JavaScript's Factory Pattern employs a *factory function* to create new objects. It was conceived as a DRY means to abstract the process of creating objects. However, there are couple problems with it:

1. **Efficiency** â€“ Methods created on the factory function are copied to all new object instances. This is inefficient.
2. **Type Determination** â€“ Because the factory function returns a new object, it makes type determination of object instances difficult. New object instances are typed as "Object", with no indication of the instances' context.

The type determination problem led to the creation of the Constructor Pattern.

Constructor Pattern

CODE

```
// -----
// JavaScript Constructor Pattern
// -----  
  

// constructor function to create car objects
function Car(make, model, year) {
    this.make = make;
    this.model = model;
    this.year = year;
    this.sayCar = function() {
        alert('I have a ' + this.year + ' ' + this.make + ' ' + this.model + '.');
    };
}  
  

// create 2 car objects for John and Jane
var johnCar = new Car('Ford', 'F150', '2011'),
    janeCar = new Car('Audi', 'A4', '2007');  
  

// call method on Jane's car
janeCar.sayCar();
```

JavaScript's Constructor Pattern solves the Factory Pattern's type determination problem by replacing the factory function with a *constructor function* to create new objects.

In the Factory Pattern:

CODE

```
alert(johnCar.constructor === Car); // false
alert(johnCar.constructor === Car); // false
```

In the Constructor Pattern:

CODE

```
alert(johnCar.constructor === Car); // true
alert(johnCar.constructor === Car); // true
```

By convention, constructor functions always start with a capital letter, as opposed to the camelCase convention we see throughout JavaScript. This convention helps to distinguish constructor functions from other functions in JavaScript. Constructor functions must be preceded by the `new` keyword in JavaScript (this is what makes them a "constructor"). As stated above, [the `new` keyword performs several actions](#) when placed before a constructor function, shown in the example code above.

The main problem with the Constructor Pattern is, as in the Factory Pattern, inefficiency. In the Constructor Pattern, methods are [still] copied to all new object instances. This problem led to the creation of the Combination Constructor/Prototype Pattern.

Combination Constructor/Prototype Pattern

```
// -----
// Combination Constructor/Prototype Pattern
// -----  
  
// constructor function to create car objects
function Car(make, model, year) {
    this.make = make;
    this.model = model;
    this.year = year;
}  
  
// constructor prototype to share properties and methods
Car.prototype.sayCar = function() {
    alert('I have a ' + this.year + ' ' + this.make + ' ' + this.model + '.');
};  
  
// create 2 car objects for John and Jane
var johnCar = new Car('Ford', 'F150', '2011'),
    janeCar = new Car('Audi', 'A4', '2007');  
  
// call method on Jane's car
janeCar.sayCar();
```

JavaScript's Combination Constructor/Prototype Pattern solves the efficiency issues present in the Factory Pattern and Constructor Pattern by utilizing prototypal inheritance — or *prototypal delegation* [as explained above](#). It is among the most popular object oriented design patterns in JavaScript because it allows for unique (non-shared) instance properties to be created within a constructor function, as well as shared properties and methods on the constructor function's prototype.

One important thing to note about prototypes is that if you assign your constructor's prototype to an object literal instead of using the dot notation, you will overwrite the default `constructor` property. The object literal tells JavaScript to *replace* the prototype with a new object, rather than *augment* the existing prototype. Because `constructor` is simply a default property on all prototypes, it gets removed when you replace the prototype with a new object. If you wish to assign a new object to a prototype *and* maintain the constructor relationship, you will need to recreate the `constructor` property and assign it the proper value — see the examples below.

Create prototype using object literal without an explicit constructor

CODE

```

function Car(make, model, year) {
    this.make = make;
    this.model = model;
    this.year = year;
}

// create constructor prototype with object literal
Car.prototype = {
    sayCar: function() {
        alert('I have a ' + this.year + ' ' + this.make + ' ' + this.model + '.');
    }
};

var johnCar = new Car('Ford', 'F150', '2011');

// get constructor on John's car
alert(johnCar.constructor === Car); // false
alert(johnCar.constructor === Object); // true

```

Create prototype using object literal and create an explicit constructor

CODE

```

function Car(make, model, year) {
    this.make = make;
    this.model = model;
    this.year = year;
}

// create constructor prototype with object literal
Car.prototype = {
    constructor: Car,
    sayCar: function() {
        alert('I have a ' + this.year + ' ' + this.make + ' ' + this.model + '.');
    }
};

var johnCar = new Car('Ford', 'F150', '2011');

// get constructor on John's car
alert(johnCar.constructor === Car); // true
alert(johnCar.constructor === Object); // false

```

The obvious reason to use the object literal syntax is to make your code simpler — to cut down on the repetition necessitated by using the dot notation for adding properties and methods to a prototype. However, the reality is that assigning a new object to a prototype makes your code more complicated because you lose the ability to determine type. Let me explain...

All properties in JavaScript have four default attributes assigned to them. These attributes are not directly accessible in JavaScript, but they have a significant impact on how the language operates. The four attributes assigned to each property in JavaScript are:

`[[Configurable]]`

The `configurable` attribute determines whether a property's subsequent attributes may be modified, or if the property may be removed via the `delete` keyword. Its default value is `true` for all properties defined directly on an object.

[[Enumerable]]

The *enumerable* attribute determines if a property will be returned in `for-in` loop. Its default value is `true` for all properties defined directly on an object.

[[Writable]]

The *writable* attribute determines if a property is writable. Its default value is also `true` for all properties defined directly on an object.

[[Value]]

The *value* attribute contains the actual data value for a property. Its default value is `undefined`.

The native `constructor` property that exists on the constructor function's prototype has an `[[Enumerable]]` value of `false`. Therefore, you *should* ensure that when assigning a new object to your constructor function's prototype, you also define the enumerable value on your custom-created `constructor` property as `false`. While you can do this with JavaScript's `Object.defineProperty` method, it adds confusion and works against original goal of simplifying code by assigning a new object to a prototype. My advice is to stick with the dot notation `â€" to augment the default prototype object on a constructor function, rather than replacing it with a new object literal.`

Finally, the main complaint with the Combined Constructor/Prototype Pattern is that some developers perceive the separation of the constructor function and its prototype confusing. This problem led to the creation of the Dynamic Prototype Pattern.

Dynamic Prototype Pattern

CODE

```
// -----
// Dynamic Prototype Pattern
// -----


// constructor function to create car objects
function Car(make, model, year) {
    this.make = make;
    this.model = model;
    this.year = year;

    // constructor prototype to share properties and methods
    if ( typeof this.sayCar !== "function" ) {
        Car.prototype.sayCar = function() {
            alert('I have a ' + this.year + ' ' + this.make + ' ' + this.model + '.');
        }
    }
}

// create 2 car objects for John and Jane
var johnCar = new Car('Ford', 'F150', '2011'),
    janeCar = new Car('Audi', 'A4', '2007');

// call method on Jane's car
janeCar.sayCar();
```

JavaScript's Dynamic Prototype Pattern encapsulates all information within a constructor, and has the benefits of both unique instance properties and shared prototypal properties and methods. In the Dynamic Prototype Pattern, the prototype is initialized inside of a constructor function. Then, conditional logic is implemented to ensure the prototype is initialized *only* on the first object instance created by a constructor.

In the example above, we check if the `Car` prototype has been initialized by comparing the type of the `sayCar` method to a function. When the `johnCar` instance is created, the `if` statement evaluates to true and the prototype method `sayCar` is initialized. When the `janeCar` instance is created, the `if` statement evaluates to false because the `sayCar` method was already initialized.

The `if` statement can be used to test any of the properties or methods on a constructor function's prototype. In the example above, we use the `sayCar` method because that is the only method that exists on the `Car` constructor's prototype.

OLOO Pattern

CODE

```
// -----
// OLOO Pattern
// -----

// constructor object to create car objects
var Car = {
    init: function(make, model, year) {
        this.make = make;
        this.model = model;
        this.year = year;
    },
    sayCar: function() {
        alert('I have a ' + this.year + ' ' + this.make + ' ' + this.model + '.');
    }
};

// create 2 car objects for John and Jane
var johnCar = Object.create(Car),
    janeCar = Object.create(Car);

// call init method on John and Jane
johnCar.init('Ford', 'F150', '2011');
janeCar.init('Audi', 'A4', '2007');

// call method on Jane's car
janeCar.sayCar();
```

JavaScript's OLOO pattern is a relatively recent object oriented JavaScript design pattern created by Kyle Simpson. OLOO stands for Objects Linked to Other Objects. The OLOO pattern simplifies the predominant class-based design patterns (listed above and below) by creating objects directly from other objects, instead of using constructor functions.

[In the OLOO Pattern] there are just pure objects that delegate to one another.

Kyle Simpson, [Advanced JavaScript on Frontend Masters](#) (OLOO video: 5:10:08--5:16:22).

In the example above, the `johnCar` and `janeCar` objects delegate to the `Car` object. It's important to note that in OLOO there are no constructors. While the first letter of the `Car` object is still capitalized, it is done so *only* to remain consistent with the accepted convention.

In OOO, rather than the implicit initialization that occurs in constructor patterns, we have explicit initialization using a method (`init` in the example above). The result is that OOO allows/forces you to create and initialize your objects at separate times, rather than bundling both object creation and initialization together.

As far as efficiency through prototypal inheritance (or "[delegation](#)") is concerned, while it may appear absent in the OOO Pattern, prototypal inheritance *is* happening. In the example above, there is just one `sayCar` method that is shared by the `johnCar` and `janeCar` objects — shown in the screenshot below.

```
> console.dir(johnCar);
  ▼ Object ⓘ
    make: "Ford"
    model: "F150"
    year: "2011"
    ▼ proto : Object
      ► init: function (make, model, year) {
      ► sayCar: function () {
      ► __proto__: Object
  ← undefined
> console.dir(janeCar);
  ▼ Object ⓘ
    make: "Audi"
    model: "A4"
    year: "2007"
    ▼ __proto__: Object
      ► init: function (make, model, year) {
      ► sayCar: function () {
      ► proto : Object
  ← undefined
```

OOO Delegation

Because there is no constructor in the OOO Pattern, calling `johnCar instanceof Car` will throw an error. Also, calling `johnCar.constructor` will delegate up the prototype chain to `Object.prototype.constructor` and return the native method. You can test the `Car` object's relationship to `johnCar` with `Car.isPrototypeOf(johnCar)`, which returns `true`.

Parasitic Constructor Pattern

```
// -----
// Parasitic Constructor Pattern
// -----



// constructor function to create car objects
function Car(make, model, year) {
    var o = new Object();

    o.make    = make;
    o.model   = model;
    o.year    = year;
    o.sayCar = function() {
        alert('I have a ' + this.year + ' ' + this.make + ' ' + this.model + '.');
    };

    return o;
}

// create 2 car objects for John and Jane
var johnCar = new Car('Ford', 'F150', '2011'),
    janeCar = new Car('Audi', 'A4', '2007');

// call method on Jane's car
janeCar.sayCar();
```

JavaScript's Parasitic Constructor Pattern is similar to the [Factory Pattern](#) in that a function is used to create and return a new object. The key difference is that a *constructor function* is used instead of a factory function.

In the example above, the constructor function explicitly returns the new `o` object. This overrides the default behavior, which in the absence of the explicit `return o;` statement, would implicitly return a new `Car` object.

The Parasitic Constructor Pattern is useful in situations where you want to abstract and encapsulate code in a safe way, as in the example below taken from [Professional JavaScript for Web Developers](#)

CODE

```

function SpecialArray() {

    // create the array
    var values = new Array();

    // add the values
    values.push.apply( values, arguments);

    // assign the method
    values.toPipedString = function() {
        return this.join(' |');
    };

    // return it
    return values;

}

// create a new [special] array for colors
var colors = new SpecialArray(' red', 'blue', 'green');

alert(colors.toPipedString()); // ' red | blue | green'

```

In this example, a constructor called `SpecialArray` is created. In the constructor, a new array is created and initialized using the `push()` method (which has all of the constructor arguments passed in). Then a method called `toPipedString()` is added to the instance, which simply outputs the array values as a pipe-delimited list. The last step is to return the array as the function value. Once that is complete, the `SpecialArray` constructor can be called, passing in the initial values for the array, and `toPipedString()` can be called.

Zakas, Nicholas C. (2011-12-20). [Professional JavaScript for Web Developers](#)
 (Kindle Locations 6714-6717). Wiley. Kindle Edition.

It's important to note that in the Parasitic Constructor Pattern, there is no relationship between the returned object and its constructor. Continuing with the `SpecialArray` example above, we see that:

CODE

```
alert(colors instanceof SpecialArray); // false
```

Bottom line, the Parasitic Constructor Pattern should not be used when other patterns suffice.

Durable Constructor Pattern

CODE

```
// -----
// Durable Constructor Pattern
// -----


// constructor function to create car objects
function Car(make, model, year) {
    var o = new Object();

    // private variable
    var condition = 'used';

    // public method
    o.sayCar = function() {
        alert('I have a ' + condition + ' ' + year + ' ' + make + ' ' + model + '.');
    };

    return o;
}

// create 2 car objects for John and Jane
var johnCar = Car('Ford', 'F150', '2011'),
    janeCar = Car('Audi', 'A4', '2007');

// call method on Jane's car
janeCar.sayCar();
```

JavaScript's Durable Constructor Pattern is similar to the [Factory Pattern](#) and Parasitic Constructor Pattern in that it creates and explicitly returns a new object. The two key differences in the Durable Constructor Pattern are:

1. The `this` keyword is never used on constructor methods.
2. The `new` keyword is never called before the constructor.

The unique aspect of the Durable Constructor Pattern is that it allows for private data that can be accessed through the process of closure. When the `condition` variable is created, it is bound to the function scope of the `Car` constructor. While the `condition` variable remains accessible to public methods attached to the `o` object within the `Car` constructor, it is *not* accessible outside of those methods — you are not be able to assign a different value to `condition` on the `johnCar` or `janeCar` objects. The private nature of variables in the Durable Constructor Pattern is also evident for arguments passed to the constructor function — `make`, `model` and `year` are all private. For this reason, the Durable Constructor Pattern is most often used in environments which require more security than other design patterns provide.

Finally, as with the Parasitic Constructor Pattern, there is no relationship between the returned object and its constructor:

CODE

```
alert(johnCar instanceof Car); // false
```

If you made it all the way through this post, you deserve a medal. Below is a link to download code examples for each pattern listed above.

[JavaScript Design Patterns](#)

52. JavaScript Functions vs Methods

Since the start of the year I have been pretty good about my resolution to get better with JavaScript. One of the things that I initially stumbled on was the difference between *functions* and *methods* in JavaScript. I have watched several tutorials on JavaScript from [Lynda](#), [Udemy](#), [Nettuts](#), [CodeSchool](#), [AppendTo](#) and [Treehouse](#). Ultimately, the clearest answer to my question came from Treehouse.

One of the courses that I recently completed on Treehouse was on jQuery. After that course, I became even more confused about the difference between JavaScript functions and methods. I went ahead and posted a question in the recently launched forums at Treehouse and received an answer from the course's instructor, Andrew Chalkley. It was such a good/thorough response, that I requested permission to re-post the it here.

So without further adieu, here is the difference between functions and methods in JavaScript.

I've made several changes to the original response and included examples where I felt necessary. While this isn't a direct quote, the gist of Andrew's answer in the Treehouse forums is still present.

Hi John,

I tend to use the terms interchangeably because in most, if not all cases, they're identical in meaning.

I found a [stackoverflow](#) that discusses it more — note the replies to the answer on how the definitions slightly change language to language. But I don't think this definition helps either, so I'll try to explain.

You can do some pretty crazy-cool things in JavaScript like passing functions around from the global scope in to an object and vice versa. The key with JavaScript the scope in which a function is executed.

For example when you define the `whoIsThis` function below:

CODE

```
function whoIsThis() {
    console.log(this); // logs window to the console when executed
}
```

This function is now a method on the `window` object. Any function defined in the global scope (in browsers) gets bound to `window`. Whilst we haven't implicitly passed in `window`, we can access it and all the variables in the global scope. Within the context of the `whoIsThis` function, `this` points to the `window` object.

CODE

1. // Define a function in the global scope
2. function sayHi() {
3. alert("Hi");
4. }
5. // Which means I can call
6. window.sayHi(); // shows an alert with the string "Hi"

In the example above, there is a function declared on the second line. If you look at last line, we call this function on

the `window` object. So, you could say that the `sayHi()` function is a *method* since it's being explicitly called on the `window` object.

The screenshot shows a browser's developer tools console tab labeled "Console". It contains the following log entries:

- > `window.jQuery == jQuery`
- < `true`
- > |

Below the console output, the text "Calling jQuery on the window object" is displayed.

Similarly, jQuery is called on the `window` object. If you open your browser's JavaScript console and type in `window.jQuery`, you can verify this. So, I find that definition proposed on StackOverflow confusing.

To be honest, I wouldn't get too hung up on the definitions. Scope is what you should be focusing on, not the semantics of functions vs methods. The context of `this` within the function/method is the key.

In your day to day developer life, someone may call it the `fooBar()` function rather than the `fooBar` method. There is no issue. People understand these terms as interchangeable.

Eventually you'll be writing code that doesn't pollute the global scope unnecessarily. CoffeeScript does this straight out of the box wrapping all of your code in a self-executing anonymous function when it's compiled down to JavaScript.

Regards, Andrew

Fantastic answer! I hope this makes the JavaScript functions vs methods issue as clear to you as it does to me.

53. Demystifying Debounce in JavaScript

There are several situations in JavaScript where you want a function that is bound to an event to fire only once after a specific amount of time has passed. Functions bound to the `resize` and `scroll` events are the typical candidates. Invoking a function every time one of these events fires will significantly impact performance. Enter the JavaScript debounce function.

Debounce functions are included in many JavaScript libraries. The goal behind each implementation is to reduce overhead by preventing a function from being called several times in succession. Regardless of the library, all debounce functions are built on JavaScript's native `setTimeout` function. Understanding specifically how they work can be difficult. While typically small in size, debounce functions take advantage of some pretty advanced JavaScript concepts like closures and execution queues.

Similar to my [explanation of a jQuery plugin](#), I wrote a basic debounce function and littered it with code comments. I [try to] explain what each line of a JavaScript debounce function does. While it does require a basic understanding of scope, closures and `setTimeout`, hopefully it will help you to better understand how these concepts come together in a debounce function.

MDN has great [documentation and examples for `setTimeout`](#). If you are unfamiliar with it, check out the MDN docs. Also, it may help to understand [how JavaScript is compiled and executed](#) (links to my post).

JavaScript Debounce Examples

The pen below shows the debounce function in action as a method on a custom object named JD. Included are two examples where functions are called that log my name to the console when the window is resized. The first function passed to the debounce method logs my last name. The second function uses the *immediate* argument in the debounce method and logs my first name.

JavaScript Debounce Explanation

```

// Create JD Object
// -----
/*
  It's a good idea to attach helper methods like `debounce` to your own
  custom object. That way, you don't pollute the global space by
  attaching methods to the `window` object and potentially run in to
  conflicts.
*/
var JD = {};

// Debounce Method
// -----
/*
  Return a function, that, as long as it continues to be invoked, will
  not be triggered. The function will be called after it stops being
  called for `wait` milliseconds. If `immediate` is passed, trigger the
  function on the leading edge, instead of the trailing.
*/
JD.debounce = function(func, wait, immediate) {
  /*
    Declare a variable named `timeout` variable that we will later use
    to store the *timeout ID returned by the `setTimeout` function.

    *When setTimeout is called, it retuns a numeric ID. This unique ID
    can be used in conjunction with JavaScript's `clearTimeout` method
    to prevent the code passed in the first argument of the `setTimout`
    function from being called. Note, this prevention will only occur
    if `clearTimeout` is called before the specified number of
    milliseconds passed in the second argument of setTimout have been
    met.
  */
  var timeout;

  /*
    Return an anonymous function that has access to the `func`
    argument of our `debounce` method through the process of closure.
  */
  return function() {

    /*
      1) Assign `this` to a variable named `context` so that the
      `func` argument passed to our `debounce` method can be
      called in the proper context.

      2) Assign all *arugments passed in the `func` argument of our
      `debounce` method to a variable named `args`.

      *JavaScript natively makes all arguments passed to a function
      accessible inside of the function in an array-like variable
      named `arguments`. Assinging `arguments` to `args` combines
      all arguments passed in the `func` argument of our `debounce`
      method in a single variable.
    */
    var context = this, /* 1 */
        args = arguments; /* 2 */

    /*
      Assign an anonymous function to a variable named `later`.
      This function will be passed in the first argument of the
      `setTimeout` function below.
    */
  }
}

```

```

*/
var later = function() {

    /*
        When the `later` function is called, remove the numeric ID
        that was assigned to it by the `setTimeout` function.

        Note, by the time the `later` function is called, the
        `setTimeout` function will have returned a numeric ID to
        the `timeout` variable. That numeric ID is removed by
        assiging `null` to `timeout`.
    */

    timeout = null;

    /*
        If the boolean value passed in the `immediate` argument
        of our `debouce` method is falsy, then invoke the
        function passed in the `func` argument of our `debouce`
        method using JavaScript's *`apply`* method.

        *The `apply` method allows you to call a function in an
        explicit context. The first argument defines what `this`
        should be. The second argument is passed as an array
        containing all the arguments that should be passed to
        `func` when it is called. Previously, we assigned `this`
        to the `context` variable, and we assigned all arguments
        passed in `func` to the `args` variable.
    */

    if ( !immediate ) {
        func.apply(context, args);
    }
};

/*
    If the value passed in the `immediate` argument of our
    `debounce` method is truthy and the value assigned to `timeout`
    is falsy, then assign `true` to the `callNow` variable.
    Otherwise, assign `false` to the `callNow` variable.
*/
var callNow = immediate && !timeout;

/*
    As long as the event that our `debounce` method is bound to is
    still firing within the `wait` period, remove the numerical ID
    (returned to the `timeout` variable by `setTimeout`) from
    JavaScript's execution queue. This prevents the function passed
    in the `setTimeout` function from being invoked.

    Remember, the `debounce` method is intended for use on events
    that rapidly fire, ie: a window resize or scroll. The *first*
    time the event fires, the `timeout` variable has been declared,
    but no value has been assigned to it - it is `undefined`.
    Therefore, nothing is removed from JavaScript's execution queue
    because nothing has been placed in the queue - there is nothing
    to clear.

    Below, the `timeout` variable is assigned the numerical ID
    returned by the `setTimeout` function. So long as *subsequent*
    events are fired before the `wait` is met, `timeout` will be
    cleared, resulting in the function passed in the `setTimeout`
    function being removed from the execution queue. As soon as the

```

```

`wait` is met, the function passed in the `setTimeout` function
will execute.

*/
clearTimeout(timeout);

/*
Assign a `setTimout` function to the `timeout` variable we
previously declared. Pass the function assigned to the `later`
variable to the `setTimeout` function, along with the numerical
value assigned to the `wait` argument in our `debounce` method.
If no value is passed to the `wait` argument in our `debounce`
method, pass a value of 200 milliseconds to the `setTimeout`
function.

*/
timeout = setTimeout(later, wait || 200);

```

Typically, you want the function passed in the `func` argument of our `debounce` method to execute once **after** the `wait` period has been met for the event that our `debounce` method is bound to (the trailing side). However, if you want the function to execute once **before** the event has finished (on the leading side), you can pass `true` in the `immediate` argument of our `debounce` method.

If `true` is passed in the `immediate` argument of our `debounce` method, the value assigned to the `callNow` variable declared above will be `true` only after the **first** time the event that our `debounce` method is bound to has fired.

After the first time the event is fired, the `timeout` variable will contain a falsy value. Therefore, the result of the expression that gets assigned to the `callNow` variable is `true` and the function passed in the `func` argument of our `debounce` method is exected in the line of code below.

Every subsequent time the event that our `debounce` method is bound to fires within the `wait` period, the `timeout` variable holds the numerical ID returned from the `setTimout` function assigned to it when the previous event was fired, and the `debounce` method was executed.

This means that for all subsequent events within the `wait` period, the `timeout` variable holds a truthy value, and the result of the expression that gets assigned to the `callNow` variable is `false`. Therefore, the function passed in the `func` argument of our `debounce` method will not be executed.

Lastly, when the `wait` period is met and the `later` function that is passed in the `setTimeout` function executes, the result is that it just assigns `null` to the `timeout` variable. The `func` argument passed in our `debounce` method will not be executed because the `if` condition inside the `later` function fails.

```

*/
if ( callNow ) {
    func.apply(context, args);
}
};

};


```

[JavaScript Debounce Function](#)

54. Hoisting in JavaScript

Hoisting is one of the more confusing aspects of JavaScript. The concept of hoisting was created by developers to explain what happens during the compilation phase when variables and function declarations are moved " or *hoisted* " to the top of their containing scope. There are some good articles listed at the bottom of this post that explain JavaScript hoisting in detail. For me, the best explanation came from Kyle Simpson's [Advanced JavaScript course](#) on Frontend Masters.

In the course, Kyle explains the compilation and execution phases of JavaScript in a pseudocode-esque conversation. This abstraction made the concept of hoisting click for me. I knew about hoisting before taking the course, but knowing and *understanding* are two very different things. Below is a code snippet along with a recap of the conversations that take place during the compilation and execution phases.

CODE

```

1. var foo = "bar";
2.
3. function bar() {
4.     var foo = "baz";
5.
6.     function baz(foo) {
7.         foo = "bam";
8.         bam = "yay";
9.     }
10.    baz();
11. }
12.
13. bar();
14. foo;           // "bar"
15. bam;          // "yay"
16. baz();        // Error!

```

JavaScript Hoisting Example

JavaScript Compilation

The first step taken by the browser's JavaScript engine

1. Line 1: Hey *global* scope, I have a declaration for a variable named `foo`.
2. Line 3: Hey *global* scope, I have a declaration for a function¹ named `bar`.
3. Line 4: Hey `bar` scope, I have a declaration for a variable named `foo`.
4. Line 6: Hey `bar` scope, I have a declaration for a function named `baz`.
5. Line 6: Hey `baz` scope, I have a declaration for a parameter named `foo`.

1. Since `bar` is a function, we recursively descend into its scope and continue compilation.

JavaScript Execution

The second step take by the browser's JavaScript engine

There are two terms that you need to be familiar with as we enter the execution phase: LHS and RHS. LHS stands for *left hand side*, and RHS stands for *right hand side*. LHS references are located on the left hand side of the `=` assignment operator. RHS references are located on the right hand size of the of the `=` assignment operator, or implied when there is no LHS reference. If this seems a bit confusing, a good way to think about LHS versus RHS is *target* versus *source*. LHS is the target, and RHS is the source.

Let's continue the conversation during the execution phase...

Line 1: Hey *global* scope, I have an LHS reference for a variable named `foo`. Ever heard of it?

The global scope has because `foo` was registered on line 1 in the compilation phase, so the assignment occurs.

Line 13:¹ Hey *global* scope, I have an RHS² reference for a variable named `bar`. Ever heard of it?

The global scope has because `bar` was registered as a function on line 3 in the compilation phase, so the function executes.

Line 4: Hey `bar` scope, I have an LHS reference for a variable named `foo`. Ever heard of it?

The `bar` scope has because `foo` was registered on line 1 in the compilation phase, so the the assignment occurs.³

Line 10:⁴ Hey `bar` scope, I have an RHS reference for a variable named `baz`. Ever heard of it?

The `bar` scope has because `baz` was registered as a function on line 6 in the compilation phase, so the function executes.

Line 7: Hey `baz` scope, I have an LHS reference for a variable named `foo`. Ever heard of it?

The `baz` scope has because `foo` was declared as a parameter of the `baz` function on line 6 in the compilation phase, so the assignment occurs.

Line 8: Hey `baz` scope, I have an LHS reference for a variable named `bam`. Ever heard of it?

The `baz` scope has not. Therefore we look for `bam` in the next outer scope, the `bar` scope.

Line 8: Hey `bar` scope, I have an LHS reference for a variable named `bam`. Ever heard of it?

The `bar` scope has not. Therefore we look for `bam` in the next outer scope, the global scope.

Line 8: Hey *global* scope, I have an LHS reference for a variable named `bam`. Ever heard of it?

The global scope has not. Therefore the global scope automatically registers a variable named `bam`.⁵

Line 14: Hey *global* scope, I have an RHS reference for a variable named `foo`. Ever heard of it?

The global scope has because `foo` was declared on line 1 in the compilation phase, its value is the string "bar".

Line 15: Hey *global* scope, I have an RHS reference for a variable named `bam`. Ever heard of it?

The global scope has because `bar` was automatically created two steps back, its value is the string "yay".⁶

Line 16: Hey *global* scope, I have an RHS reference for a variable named `baz`. Ever heard of it?

The global scope has not because `baz` was exists in the function scope of `bar`. Therefore, `baz` is inaccessible to the global scope and a reference error is thrown.

1. Lines 3--11 don't exist in the execution phase because they were compiled away. So, we move to line 13.
2. The reason line 13 is an RHS is because there is no assignment. As such, we cannot establish a left/right reference. Therefore, we know that the value on line 13 represents the *source*.
3. Within the `bar` scope, `foo` will always refer to the value assigned to it on line 4. This is because the `foo` variable on line 4 is preceded with the `var` keyword, and will therefore be the first reference to `foo` inside the `bar` scope.
4. Lines 6--9 don't exist in the execution phase because they were compiled away. So, we move to line 10.

5. If you are in strict mode, the `bam` variable will *not* be registered. Therefore, because `bam` doesn't exist a reference error will be thrown.
6. Again, if you are in strict mode, a reference error will be thrown because `bam` doesn't exist.

JavaScript Hoisting

Puting it all together — compilation + execution

Hoisting is simply a mental construct. You saw hoisting in action during the compilation phase in the example above. Understanding how JavaScript is compiled and executed is the key to understanding hoisting. Let's go through one more simpler conversation in the context of hoisting and examine what happens with the code before, during and after compilation.

CODE

```
1. // Code as authored by developer
2. a;
3. b;
4. var a = b;
5. var b = 2;
6. b;
7. a;
```

Before Compilation

CODE

```
1. /*
2.      Notice: Variable declarations have been **hoisted** to the top of the
3.      containing scope. In this case, the global scope.
4. */
5. var a;
6. var b;
7. a;
8. b;
9. a = b;
10. b = 2;
11. b;
12. a;
```

During Compilation

CODE

```

1. /*
2.      Notice: The var keyword has been compiled away.
3. */
4. a;      // undefined
5. b;      // undefined
6. a = b;
7. b = 2;
8. b;      // 2
9. a;      // undefined

```

After Compilation

Now that you understand how variables get hoisted during the compilation phase, understanding the execution phase is much easier. Below is the conversation that occurs during the execution phase, as shown in the *After Compilation* figure above.

Line 4: Hey *global* scope, I have an RHS reference for a variable named `a`. Ever heard of it?

The global scope has because `a` was registered on line 5 in the compilation phase, its value is undefined.

Line 5: Hey *global* scope, I have an RHS reference for a variable named `b`. Ever heard of it?

The global scope has because `b` was registered on line 6 in the compilation phase, its value is undefined.

Line 6: Hey *global* scope, I have an LHS reference for a variable named `a`. Ever heard of it?

The global scope has because `a` was registered on line 5 in the compilation phase, so the assignment occurs.

Line 7: Hey *global* scope, I have an LHS reference for a variable named `b`. Ever heard of it?

The global scope has because `b` was registered on line 6 in the compilation phase, so the assignment occurs.

Line 8: Hey *global* scope, I have an RHS reference for a variable named `b`. Ever heard of it?

The global scope has because `b` was registered on line 6 in the compilation phase and a value was assigned to it on line 7 in the [current] execution phase, its value is the number 2.

Line 9: Hey *global* scope, I have an RHS reference for a variable named `a`. Ever heard of it?

The global scope has because `a` was registered on line 5 in the compilation phase and `b` was assigned to it on line 6 in the [current] execution phase. As `b` was undefined at the time of its assignment to `a`, the value of `a` is undefined.

To keep our example short and simple, I did not include functions. Note that functions are *hoisted* in the same way variables are. Functions get hoisted *above* variables during the compilation phase.

Hopefully this post helps to further your understanding of hoisting in JavaScript. Below are the resources on hoisting that I mentioned at the beginning of the post.

Hoisting Resources

- [JavaScript Hoisting Explained by Jeffrey Way](#)
- [You Don't Know JS: Scope & Closures by Kyle Simpson](#)
- [JavaScript Scoping and Hoisting](#)
- [JavaScript Variables on MDN](#)

55. JavaScript Hoisting Explained

Today's video quick tip comes in response to a question on [Twitter](#), concerning JavaScript "hoisting." What is it? How does it work? What do you need to know about it? All of that will be covered in this beginner-focused fundamentals lesson.

Hoisting Explained

Consider the following code:

```
1 var myvar = 'my value' ;
2 alert(myvar); // my value
```

Okay, of course the alert will display "my value." That's obvious; however, stick with me. Let's next create an immediate function, which alerts the same value.

```
1 var myvar = 'my value' ;
2
3 ( function () {
4     alert(myvar); // my value
5 })();
```

All right, all right. Still obvious, I know. Now, let's throw a wrench into the mix, and create a local variable within this anonymous function of the same name.

```
1     var myvar = 'my value' ;
2
3 ( function () {
4     alert(myvar); // undefined
5     var myvar = 'local value' ;
6 })();
```

Huh? Why is the alert now displaying `undefined`? Even though we've *declared* a new variable, it's still below the alert; so it shouldn't have an effect, right? **Wrong**.

Variable Declarations are Hoisted

Within its current scope, regardless of where a variable is declared, it will be, behind the scenes, hoisted to the top. However, only the `declaration` will be hoisted. If the variable is also `initialized`, the current value, at the top of the scope, will initially be set to `undefined`.

Okay, let's decipher the difference between the terms, `declaration` and `initialization`. Assume the following line: `var joe = 'plumber';`

Declaration

```
1 var joe; // the declaration
```

Initialization

```
1 joe = 'plumber' ; // the initialization
```

Now that we understand the terminology, we can more easily comprehend what's happening under the hood. Consider the following bogus function.

```
1 ( function () {
2     var a = 'a' ;
3     // lines of code
4     var b = 'b' ;
5     // more lines of code
6     var c= 'c' ; // antipattern
7     // final lines of scripting
8 })();
```

Declare all variables at the top.

Note that what's exemplified above is considered to be bad practice. Nonetheless, behind the scenes, all of those variable declarations — regardless of where they occur in the function scope — will be hoisted to the top, like so:

```
1 ( function () {
2     var a, b, c; // variables declared
3     a = 'a' ;
4     // lines of code
5     b = 'b' ; // initialized
6     // more lines of code
7     c= 'c' ; // initialized
8     // final lines of scripting
9 })();
```

Aha Moment

If we now return to the original confusing `undefined` piece of code, from above:

```
1 var myvar = 'my value' ;
2
3 ( function () {
4     alert(myvar); // undefined
5     var myvar = 'local value' ;
6 })();
```

It should now make perfect sense why `myvar` is alerting `undefined`. As we learned, as soon as the local variable, `myvar`, was declared, it was automatically hoisted to the top of the function scope...above the alert. As a result, the variable had already been declared at the time of the alert; however, because initializations aren't hoisted as well, the value of the variable is: `undefined`.

56. JavaScript Class Instantiation & Inheritance Patterns

At first glance, JavaScript may not seem like an object oriented programming (OOP) language, but it does have

powerful object oriented programming capabilities. OOP features such as inheritance, polymorphism, and encapsulation are all enabled in JavaScript. However, being a very flexible language, there are several different ways to define classes and use them. Here's an overview of all the different JavaScript class instantiation and inheritance patterns and how to implement them.

Sections

- [Functional](#)
- [Functional Shared](#)
- [Prototypal](#)
- [Pseudoclassical](#)
- [Summary](#)

Functional

A class is a set of characteristics that define a collection of things. Knowing that something belongs to a certain class should indicate that certain properties are exhibited by that thing. In OOP, the "thing" would be referred to as an **object**, which is an **instance** of a class. The simplest way to create objects that all exhibit the same properties, is through a function.

Consider the function `makeAnimal` that takes a `name` and a `color` and returns animal objects. Every invocation of `makeAnimal` will create an instance that has `name` and `color` properties. This function is a **constructor** in the sense that it creates objects of a particular class.

```
var makeAnimal = function(name, color) {  
    var animal = {};  
    animal.name = name;  
    animal.color = color;  
    return animal  
}  
  
var bob = makeAnimal('bob', 'blue'); // bob.name === 'bob'  
var yo = makeAnimal('yo', 'yellow'); // yo.color === 'yellow'
```

CODE

Methods (functions associated with a class) can also be added to each instance via the constructor function.

```
var makeAnimal = function(name, color) {  
    var animal = {};  
    animal.name = name;  
    animal.color = color;  
    animal.growl = function() {  
        console.log('grrr');  
    }  
    return animal  
}  
  
var bob = makeAnimal('bob', 'blue'); // bob.name === 'bob'  
  
bob.growl() // logs 'grrr'
```

CODE

However, not all animals are capable of growling, so it may not be appropriate to add that method on all animals. This is a good reason to subclass. Subclassing under the functional pattern is very straight forward. I can create a function that makes bears, employing `makeAnimal` in the process:

CODE

```
var makeAnimal = function(name, color) {
  var animal = {};
  animal.name = name;
  animal.color = color;
  return animal;
}

var makeBear = function(name, color, type) {
  var bear = makeAnimal(name, color);
  bear.type = type;
  bear.growl = function() {
    console.log('grrr');
  }
  return bear;
}

var yogi = makeBear('yogi', 'brown', 'thief');

yogi.growl(); // logs 'grrr'
```

This, in essence, is inheritance under the functional pattern. You would be correct to point out that this does not provide certain capabilities offered by inheritance in other OOP languages. For instance, there is no way to know that bears made by `makeBear` are animals without examining the code. This code is also not very efficient. Every instance of bear (and animal, for that matter) holds properties that point to its own function definitions. There is no convenient method to alter the functionality of the entire class.

CODE

```
var yogi = makeBear('yogi', 'brown', 'thief');
var pooh = makeBear('winnie', 'yellow', 'honey');

yogi.growl(); // logs 'grrr'
yogi.growl === pooh.growl; // false
```

If my app required the creation of thousands of bears, there would be a lot of redundant memory costs associated with this pattern.

There are benefits to using this pattern though - this code is very transparent and easy to understand. Depending on the expected use of this function, the redundant memory costs may be perfectly acceptable.

Functional Shared

A slightly improved version of the functional pattern involves eliminating that redundant memory cost mentioned above. Instead of defining a function on each instance of the class, the methods can be defined in some form of *method holder* for each instance to point at.

Using the same example as above:

CODE

```

var makeAnimal = function(name, color) { // no change to this
  var animal = {};
  animal.name = name;
  animal.color = color;
  return animal;
}

var bearMethods = { // the holder of all methods that bears should have
  growl: function() {
    console.log('grrr');
  }
}

var makeBear = function(name, color, type) {
  var bear = makeAnimal(name, color);
  bear.type = type;
  bear.growl = bearMethods.growl; // point at same function as bearMethods.growl
  return bear;
}

var yogi = makeBear('yogi', 'brown', 'thief');
var pooh = makeBear('winnie', 'yellow', 'honey');

yogi.growl(); // logs 'grrr'
yogi.growl === pooh.growl; // true

```

If more methods are needed for bears, simply define them within `bearMethods`, and use a function such as [extend](#) within `makeBear` to extend all the methods onto the bear instance.

Inheritance would work the same way as the plain Functional pattern. A subclass of bears would invoke `makeBear` to create a bear (which *extends* all methods from `bearMethods`) and then decorate the instance with additional properties and methods specific to the subclass (for example, extending it with methods from a method holder called `polarBearMethods`).

Aside from the potential memory cost savings, this pattern does not offer many other benefits over the first functional pattern. There is still no convenient way to alter the methods for all instances of a class. Altering the `bearMethods` object will only point the properties at new function objects. The bears that have already been instantiated will not receive the update, unless explicitly updated with these new method definitions.

Prototypal

The functional patterns do not create proper objects in the classical sense. They do not allow inheritance, since the instances of the "class" are very much their own object with access to only their own properties. The prototypal pattern allows for this behavior through an object's **prototype**.

Consider this example using pokemon under the functional shared pattern:

CODE

```
// functional shared pattern
var pokemonMethods = {
  growl: function() {
    console.log( this.name + '! ' + this.name + '!');
  }
}

var makePokemon = function(name, level, type) {
  var pokemon = {};
  pokemon.name = name;
  pokemon.level = level;
  pokemon.type = type;
  pokemon.growl = pokemonMethods.growl;
  return pokemon;
}

var zeni = makePokemon('squirtle', 15, 'water');
zeni.growl(); // logs 'squirtle! squirtle!'

pokemonMethods.fight = function() {
  console.log( this.name + ' used tackle!');
}

zeni.fight(); // fight is undefined!
```

A simple modification to this code transforms this into the prototypal instantiation pattern:

CODE

```
// prototypal pattern
var pokemonMethods = {
  growl: function() {
    console.log( this.name + '! ' + this.name + '!');
  }
}

var makePokemon = function(name, level, type) {
  var pokemon = Object.create(pokemonMethods); // *
  pokemon.name = name;
  pokemon.level = level;
  pokemon.type = type;
  // pokemon.growl = pokemonMethods.growl; // *
  return pokemon;
}

var zeni = makePokemon('squirtle', 15, 'water');
zeni.growl(); // logs 'squirtle! squirtle!'

pokemonMethods.fight = function() {
  console.log( this.name + ' used tackle!');
}

zeni.fight(); // logs 'squirtle used tackle!'
```

Note the very important difference within the `makePokemon` function. Instead of creating an empty object in the first line of `makePokemon`, `Object.create` is called with `pokemonMethods` as an argument. `Object.create` (introduced with ES5) is a method that returns a new object that will delegate failed property lookups to the object

passed in to the function. For this reason, extending methods from the method holder to the instance is no longer necessary. Calling `zeni.growl()` will do the following:

1. Search `zeni` for the method `growl`
2. Fail method lookup on `zeni`
3. Search `pokemonMethods` for the method `growl`
4. Successful method lookup
5. Invoke `growl`, with `this` referring to `zeni`

The most powerful aspect of this pattern is that new methods declared on `pokemonMethods` (the prototype) will be available to `zeni` without any additional action. Modifications to `pokemonMethods` will affect all those that inherit from it, so long as they rely on `pokemonMethods` for lookups. If a `growl` method got defined on `zeni`, it would effectively **mask** the `growl` method on the prototype.

```
zeni.growl = function() {
  console.log('squirrrrrrttlleeeeeeee!!!');
}

zeni.growl(); // logs 'squirrrrrrttlleeeeeeee!!!'
```

CODE

Inheritance under this pattern is also possible.

```
var squirtleMethods = Object.create(pokemonMethods); // *
squitleMethods.fight = function() {
  console.log( this.name + ' used Water Gun!' );
}

var makeSquirtle = function(level, color) {
  var squirtle = Object.create(squitleMethods);
  squirtle.name = 'squirtle';
  squirtle.level = level;
  squirtle.type = 'water'
  squirtle.color = color; // new property for squirtle class
  return squirtle;
}

var zeni = makeSquirtle(15, 'blue');

zeni.growl(); // logs 'squirtle! squirtle!'
zeni.fight(); // logs 'squirtle used Water Gun!'
```

CODE

Note that `growl` is not a method available on `squitleMethods`, but since `squitleMethods` delegates to `pokemonMethods`, it is still available on `zeni`.

Pseudoclassical

The pseudoclassical pattern is the pattern that most resembles the classic object instantiation pattern of other languages such as Java and C++. Here the function `Pokemon` is capitalized by convention to indicate that it is a **constructor** (the capitalization does nothing special besides indicate to readers that this is a constructor). Constructor functions need to be instantiated by using the `new` keyword.

The `new` keyword has the effect of letting `this` within the constructor function inherit from the prototype. The

prototype in this case is `Pokemon.prototype` instead of a `pokemonMethods`. `Pokemon.prototype` is nothing more than an object defined as a property of the constructor function (recall that functions are objects themselves, and can have arbitrary properties defined on them). For convenience, the `prototype` property is defined for you by JavaScript and is automatically the prototype of all instances created by the constructor.

CODE

```
var Pokemon = function(name, level, type) {
  this.name = name;
  this.level = level;
  this.type = type;
}

Pokemon.prototype.growl = function() {
  console.log( this.name + '! ' + this.name + '!' );
}

Pokemon.prototype.fight = function() {
  console.log( this.name + ' used tackle!' );
}
```

The pseudoclassical pattern really shines when it comes to inheritance. Here is an example of subclassing the `Pokemon` class to a `Squirtle` class.

CODE

```
var Squirtle = function(level, color) {
  Pokemon.call(this, 'squirtle', level, 'water'); // *
  this.color = color;
}

Squirtle.prototype = Object.create(Pokemon.prototype); // *
Squirtle.prototype.constructor = Squirtle; // *

Squirtle.prototype.fight = function() {
  console.log( this.name + ' used Water Gun!' );
}

var zeni = new Squirtle(15, 'blue');

zeni.growl(); // logs 'squirtle! squirtle!'
zeni.fight(); // logs 'squirtle used Water Gun!'
```

There are a couple of important things to note about subclassing. The first line within the `Squirtle` constructor merely invokes the `Pokemon` constructor and passes along the appropriate arguments. The `this` keyword, which references the instance of `Squirtle` being created, will have properties `name`, `level`, and `type` assigned via `Pokemon`.

More important are the lines that set up the proper prototype chain. Without these two lines, `Squirtle` would merely be a regular constructor function, where instances of `Squirtle` would delegate failed property lookups to `Squirtle.prototype`. As mentioned before, `Object.create` returns an object that inherits from the prototype passed in as the argument. Using the code below, `Squirtle.prototype` now inherits from `Pokemon.prototype`. The `growl` method from `Pokemon.prototype` can now be called from `zeni`:

CODE

```
Squirtle.prototype = Object.create(Pokemon.prototype);
```

Something I have not mentioned earlier is that each prototype property of a function has another property constructor that points back at the function itself. That is:

```
Pokemon.prototype.constructor === Pokemon; // true
```

CODE

So what happens when we set Squirtle.prototype to Object.create(Pokemon.prototype) ?

Squirtle.prototype actually does not have its own constructor property (Object.create does not conveniently set this up for you). As a result, it delegates failed lookups to Pokemon.prototype , which **does** have a constructor property...pointing to Pokemon . This line is required to set things back to normal.

```
Squirtle.prototype.constructor = Squirtle;  
  
// and now this makes sense  
zeni.constructor === Squirtle; // true
```

CODE

The pseudoclassical instantiation pattern is usually the most popular. However, it requires a thorough understanding of the pseudoclassical instantiation pattern. Furthermore, before ES5 introduced Object.create there were several other conventions used to substitute this LOC, such as setting it equal to a new instance of Pokemon.

```
Squirtle.prototype = Object.create(Pokemon.prototype); // correct  
Squirtle.prototype = new Pokemon(); // used to be common practice, not correct
```

CODE

Failed method lookups will fall through to the instance of Pokemon, and then to the prototype of Pokemon, as desired. However, this created problems if the Pokemon constructor required a lot of initialization parameters and could not accept undefined ones.

Know that Object.create is the only correct pattern to use, but there might still be established code bases out there with the old pattern. In fact, I often see blog posts and websites reference the old pattern as the proper way to set up inheritance in JS.

Summary

Even though the pseudoclassical pattern is the most powerful and is often optimized by many runtime engines, it may sometimes be beneficial to go with a functional pattern for simplicity and transparency. It all depends on the intended usage. In my personal opinion, the **functional** and **pseudoclassical** patterns provide the best value. Here's a summary:

- **Functional**
 - Most transparent and easy to understand
 - Higher memory cost because each instance retains its own properties
- Functional shared
 - Offers memory cost advantage over functional
 - Adds a little complexity over functional
- Prototypal
 - Not much easier to comprehend compared to the pseudoclassical pattern
 - Does not offer any obvious advantages over pseudoclassical pattern
- **Pseudoclassical**

- Often the most optimized pattern
- More complex, but is a very common pattern

57. Functional inheritance in javascript

People often ask me how do I do inheritance in javascript. Can javascript do inheritance? Even if javascript has classes?

In this post I will try to give answers to those questions and present inheritance pattern I use frequently in javascript.

Are there classes in javascript?

The short answer is no. In javascript the functions are a first class citizens and there is no concept of a class.

But before I explain what we can do about it let's first ask ourselves one important question: why do we need classes? Oh, that's easy, classes organize our code and promote code reuse. Knowing that we can improve our question: "Are there tools in javascript to help us organize our code and to promote code reuse?". Now this is much better question and the answer is yes, we can use design patterns to emulate constructs that are missing in language specification.

Although in ECMAScript 6 specification there is a "class" keyword, note that this hardly changes anything from the language point. Language is still functional and classes are merely a syntactic sugar.

Inheritance patterns

So if there are no classes can there be an inheritance? Again remember to think in terms of functionality rather than concepts you know from other programming languages. There are design patterns which we can exploit to get functionality of inheritance.

There are two well known inheritance design patterns in javascript (at least for me): Pseudoclassical and Functional. The terminology comes from Douglas Crockford's book "Javascript - The Good Parts" which I recommend reading.

The Pseudoclassical inheritance pattern is not the subject of this article, it's much more used then the functional pattern and you can find example of it almost everywhere.

Functional inheritance pattern

In the functional inheritance pattern we create our "classes" with functions that return objects. The pattern is similar and probably comes from the module design pattern. The properties you attach to the returning object are class public interface. Functions and properties you define inside function, but not as a part of the returning object, are private members accessible because of closure and invisible because they were not returned.

See the following example:

Defining a class

CODE

```

function TestClass(/*arguments*/)
{
    // return object
    var obj = {};

    var privateField;

    function privateFunction() {}

    obj.publicField = "";

    obj.publicFunction = function() {};

    // construction logic here

    return obj;
}

```

We have defined class function which returns object. Properties and functions defined on that object become public interface and the rest is private. One thing is important to note here, private properties are truly private and there is no way to force class to give up its secrets.

Ok, how do we inherit that class? See the following example:

Inheritance

CODE

```

function TestClassSpecific(/*arguments*/)
{
    // initial object is instance of our test class
    var obj = TestClass();

    // save reference to parent class function
    var super_publicFunction = obj.publicFunction;

    obj.publicFunction = function()
    {
        // call parent class function
        super_publicFunction();
    };

    // construction logic here

    return obj;
}

```

A few things to note here: our return object is instance of parent class, because of that fact it contains all methods defined on parent class. Second if we want to override parent class function we have to save reference to that function first, then we can easily invoke parent function inside our override.

It's simple as that. Very neat and easy on eyes. But wait it gets even better, you can also have protected members:

Protected members

```

function TestClass(my)
{
    var obj = {};

    my = my || {};

    my.protectedField = "";

    my.protectedFunction = function(){};

    return obj;
}

function TestClassSpecific()
{
    var my = {},
        obj = TestClass(my);

    return obj;
};

```

This one may be confusing at first but makes perfect sense once you understand it. If you take object as a argument of parent class and attach properties and functions to that objects then those fields become protected, shared secrets between parent and child classes. In the example above `TestClassSpecific` **can** access `protectedField` and `protectedFunction`, but code outside that class can't.

Although this inheritance pattern looks too good to be true you need to understand implications of its usage:

Pros

- Easy on eyes. This pattern is very straightforward and easy for beginners to learn.
- Offers truly private and protected members.
- Protects against common javascript pitfalls, specifically if you forget to specify "new" keyword during class creation or if you are using "this" keyword inside functions.

Cons

- Requires more memory than Pseudoclassical inheritance pattern. With every new instance of a class memory is reserved for all functions and fields inside that class.
- Types cannot be tested using "instanceof" keyword.
- Javascript minimizers might not perform as good as with Pseudoclassical pattern as there is no way for minimizer to safely rename members of parent class.

Conclusions

So, here we have some strong points both for and against functional inheritance pattern and one question comes by itself: Should I use it? There is no unique answer to that question, it depends on your situation. Carefully consider both pros and cons and decide if this pattern is good for you.

I'm successfully using it on large project (about 100000 lines of javascript code) with nothing but success. I choose it

mainly because of the small learning curve - the ease with which I can explain inheritance to new developers.

What do you think? Does it work for you?

58. Four Rules to Define this in JavaScript

Understanding how `this` works in JavaScript is fairly simple. Knowing what `this` points to however... that is *much* more difficult. Scope and context become important concepts to understand. Knowing exactly what `this` points to and *why* within each function/method in your JavaScript code can be utterly confusing. Hopefully the rules and corresponding examples below provide clarity on `this`.

TL;DR â€“ The four rules presented for determining what `this` in JavaScript points to boil down to one simple question: **What is the calling object?**

In JavaScript functions don't know where they live. They only know how they are called.

Alexis Abril, [Advanced JS Fundamentals to jQuery & Pure DOM Scripting](#) (What is "this": 9:09--9:13)

1. **Whenever a function is contained in the global scope, the value of `this` inside of that function will be the `window` object.**

```
// Basic Function
// -----
function greetMe(name) {
    console.log('Hello ' + name);
    console.log(this);
}

greetMe('John');
```

CODE

```
> // Basic Function
// -----
function greetMe(name) {
    console.log('Hello ' + name);
    console.log(this);
}

greetMe('John');
Hello John
▶ Window {top: Window, location: Location, document: document, window: Window, external: Object...} VM2522:6
```

VM2522:5

Functions within the global scope are in fact methods on the `window` object. So, calling `greetMe('john')`; is no different than calling `window.greetMe('john')`; Therefore, `this` inside of the `greetMe` function points to `window`.

2. **Whenever a function is called by a preceding dot, the object before that dot is `this`.**

CODE

```
// Object Method
// -----
var greetMe = {
  greeting: 'Hello ',
  speak: function(name) {
    console.log(this.greeting + name);
    console.log(this);
  }
}

greetMe.speak('John');
```

```
> // Object Method
// -----
var greetMe = {
  greeting: 'Hello ',
  speak: function(name) {
    console.log(this.greeting + name);
    console.log(this);
  }
}

greetMe.speak('John');
```

Hello John

VM2280:7

▶ Object {greeting: "Hello "}

VM2280:8

This example is similar to what we saw above with `window.greetMe`. Here, instead of `window` we have the `greetMe` object on the left of the dot. Therefore, `this` inside of the `speak` method points to `greetMe`.

CODE

```
// Method Assignment
// -----
var greeting = 'Salutations ',
greet = greetMe.speak;

greet('John');
```

```
> // Method Assignment
// -----
var greeting = 'Salutations ',
greet = greetMe.speak;

greet('John');
```

Salutations John

VM2280:7

▶ Window {top: Window, location: Location, document: document, window: Window, external: Object...} VM2280:8

While this example appears to break [rule 2](#), it doesn't because it actually applies to [rule 1](#). In this example, we establish two global variables: `greeting` and `greet`. Remember, since these variables are declared in the global scope, they are actually just properties on the `window` object. Therefore, when `greet` is called, while it points to the `greetMe.speak` method, it is executed in the context of the `window` object. It would be as if you executed `window.greet('John');`. Therefore, `this` inside of the `greet` function points to `window`.

3. Whenever a constructor function is used, this refers to the specific instance

of the object that is created and returned by the constructor function.

```
// Constructor Function
// -----
function GreetMe(name) {
    this.greeting = 'Hello ';
    this.name = name;
    this.speak = function() {
        console.log(this.greeting + this.name);
        console.log(this);
    }
};

var greetJohn = new GreetMe('John');
var greetJane = new GreetMe('Jane');

greetJohn.speak();
greetJane.speak();
```

CODE

```
> // Constructor Function
// -----
function GreetMe(name) {
    this.greeting = 'Hello ';
    this.name = name;
    this.speak = function() {
        console.log(this.greeting + this.name);
        console.log(this);
    }
};

var greetJohn = new GreetMe('John');
var greetJane = new GreetMe('Jane');
```

greetJohn.speak();
greetJane.speak();

Hello John

VM2521:8

▶ GreetMe {greeting: "Hello ", name: "John"}

VM2521:9

Hello Jane

VM2521:8

▶ GreetMe {greeting: "Hello ", name: "Jane"}

VM2521:9

In the example above, both the `greetJohn` and `greetJane` variables are assigned a unique object returned by the `GreetMe` constructor function. Therefore, `this` inside of the `speak` method points to the unique `GreetMe` object instance stored in the variable on which the `speak` method is being called. This is evident by looking at the `name` property on the `GreetMe` object.

The `new` keyword in JavaScript makes a standard function into a constructor function. It does several other things as well, detailed at the top of my post on [object oriented JavaScript patterns](#).

CODE

```
// Constructor Function Prototype Method
// -----
GreetMe.prototype.sayGoodbye = function() {
    console.log('Goodbye ' + this.name);
    console.log(this);
};

greetJohn.sayGoodbye();
greetJane.sayGoodbye();
```

```
› // Constructor Function Prototype Method
// -----
GreetMe.prototype.sayGoodbye = function() {
    console.log('Goodbye ' + this.name);
    console.log(this);
};

greetJohn.sayGoodbye();
greetJane.sayGoodbye();

Goodbye John
▶ GreetMe {greeting: "Hello ", name: "John"}  

Goodbye Jane
▶ GreetMe {greeting: "Hello ", name: "Jane"}
```

[VM2493:5](#)[VM2493:6](#)[VM2493:5](#)[VM2493:6](#)

This example is similar to the previous. The difference is that instead of a method being called that exists directly on the object instance, the method is called from the constructor function's prototype. However, the context of `this` does not change — it still points to the unique `GreetMe` object stored in the variable on which the `sayGoodbye` method is being called.

4. Whenever JavaScript's `call` or `apply` method is used, `this` is explicitly defined.

CODE

```
// Using call/apply
// -----
greetJohn.sayGoodbye.call(greetJane);
greetJane.sayGoodbye.apply(greetJohn);
```

```
› // Using call/apply
// -----
greetJohn.sayGoodbye.call(greetJane);
greetJane.sayGoodbye.apply(greetJohn);

Goodbye Jane
▶ GreetMe {greeting: "Hello ", name: "Jane"}  

Goodbye John
▶ GreetMe {greeting: "Hello ", name: "John"}
```

[VM2493:5](#)[VM2493:6](#)[VM2493:5](#)[VM2493:6](#)

JavaScript's `call` and `apply` methods allow you to execute a function in a different context. The first argument passed to either of these methods explicitly sets what `this` points to. Because `call` and `apply` are explicit, they present the clearest case of what `this` points to. This is evident by looking at the `name` property logged to the console by each invocation of the `sayGoodbye` method.

59. How does Hoisting Work in Javascript?

What is Hoisting mean anyways?

Before jumping into what hoisting means in the context of javascript, it's important to learn that 'hoisting' simply means to raise something up. What? Here are example usages of the word: She hoisted her backpack onto her shoulders or The tractor hoisted up the fallen tree. The same idea of lifting something up or 'hoisting' something in the natural world also occurs in javascript with variable declarations & function declarations.

Hoisting in Javascript

There is a conceptual model for how javascript code is interpreted by the browser & that conceptual model is called 'hoisting'. Interestingly enough, if you open up the javascript spec, you will not find the word hoisting anywhere, because hoisting actually isn't a thing. It's a mental construct that we as programmers, particularly in the javascript world, have invented to explain the behavior of javascript.

Let's say we have this file called `file1.js`. Take a look at it for a moment. It looks very simple right?

```
a;          // ???
b;
var a = 2;  // ???
var b = 2;  // ???
b;          // 2
a;          //???
```

CODE

By simply glancing at the code, it would reasonable to assume, that our code would just execute line-by-line, and that variables 'a' & 'b' would cause a reference error, meaning that they weren't defined. What javascript will actually do is go through our entire code first & compile our code before it's executes it. The code you see in `file.js` can more accurately be looked at like this:

```
var a;    // undefined
var b;    // undefined
a;        // undefined
b;        // undefined
a = b;   // undefined
b = 2;   // b=2
b;        // 2
a;        // undefined
```

CODE

Rather than thinking that variable "a" would be undeclared on line 1, in reality in javascript variable declarations get moved to the top of their scope. What's happening is the js engine will go through all of our code & find all of the variable declarations first. Variables get treated first during the compile phase & then the assignments are left in place. This code moving up to the top is what we refer to as hoisting in javascript.

Now, how about with functions? (Before):

CODE

```

var a = b();    //??
var c = d();    //??
a;            //??
c;            //??

function b(){
  return c;    //??
}

var d = function(){
  return b();  //??
}

```

The way it's actually executed (After):

CODE

```

function b(){  // this is function declaration
  return c;
}
var a;        // undefined
var c;        // undefined
var d;        // undefined
a = b();      // a = undefined
c = d();      // TypeError: d is not a function
a;            // undefined
c;            // undefined
d = function(){ // undefined
  return b();
}

```

Functions get moved to the top first of their scope first, then all of the variables, then code starts executing and then finally variables get assigned their values. Below are some more examples of hoisting.

Hoisting With Nested Scope (Before):

CODE

```

foo();          // ??

function foo() {
  console.log(a); // ??
  var a = 2;
}

```

How the code is actually interpreted (After):

CODE

```

function foo(){
  var a;        // undefined
  console.log(a); // undefined
  a = 2;
}
foo();

```

Another very important aspect of hoisting to learn is that function declarations are hoisted up to the top of their scope,

but function expressions are not. For example:

Before:

```
foo();      //??
var foo = function bar() {
    var a = 1;
    console.log(a)
};
```

CODE

After:

```
var foo;  //undefined
foo();    //trying to invoke undefined is a type error
foo = function(){
    var a;
    a = 1;
    console.log(a);
}
```

CODE

Summary:

- If there are functions(not function expressions), they get hoisted first to the top of the scope in which they were defined in.
- Next variables are hoisted
- Afterwards variables are assigned their values.
- Tip: use function expressions over function declarations, there will be less surprises.

Additional Resources

- [YDKJS: Scopes & Closures by Kyle Simpson](#)
- [Javscript is Sexy : Hoisting](#)

60. JavaScript Class/Instantiation Patterns

Like in pretty much all things in life, *there are several ways you do things*. However, sometimes it's worth exploring which solutions are popular, efficient, readable or what have you. Over the past few months as I've been deep diving into javascript. I've come across several ways you can create "classes" in the language. I recall the frustration I had when I first started learning about this topic, so today I'd like to briefly share with you the different class instantiation I've learned.

###Functional Pattern

CODE

```

var Car = function(color){
    var obj = {};
    obj.color = color;
    obj.door = 'closed';
    obj.doorOpen = function(){
        obj.door = "open";
    };
    obj.closeDoor = function(){
        obj.door = 'closed';
    };
    return obj;
};

var car = Car('red');

```

Pros:

- clear object construction
- everything that is being done is contained inside the function

Cons:

- Methods duplicated for every object we create, so new spot in memory created.
- Not ideal if your creating multiple instance of objects.

###Functional shared pattern

CODE

```

//Functional-shared pattern
var Car = function(color){
    var obj = {};
    obj.color = color;
    obj.door = 'closed';
    obj.openDoor = carMethods.openDoor;
    obj.closeDoor = carMethods.closeDoor;
    return obj;
};

var carMethods = {};
carMethods.openDoor = function(){
    this.door = 'open';
};
carMethods.closeDoor = function(){
    this.door = "closed";
};

var car = Car('red');

```

Pros:

- clear object construction
- no duplication of methods each time we create an instance

Cons:

- Setting method pointer is less efficient than delegating a fallback
- Not ideal, if you're creating multiple instances of objects.

###Prototypal Pattern

```

var Car = function(color){
    var obj = Object.create(Car.prototype);
    obj.color = "color";
    obj.door = 'open';
    return obj;
};

//Automatically created by interpreter
//Car.prototype = {};
Car.prototype.openDoor = function(){
    this.door = 'open';
};
Car.prototype.closeDoor = function(){
    this.door = 'closed';
};

var car = Car('red');

```

CODE

Pros:

- clear object construction
- no duplication of methods each time we create an instance

Cons:

- Setting method pointer is less efficient than delegating a fallback
- Not ideal, if you're creating multiple instances of objects.

###Pseudoclassical Pattern

```

var Car = function(color){
    //Automatically created by interpreter
    //var this = Object.create(Car.prototype);
    this.color = color;
    this.door = 'open';

    //return this;
};

var car = new Car('red');

```

CODE

###ES6 Class keyword

CODE

```

class Car {
  constructor(color){
    this.color = color;
  }
  openDoor(){
    this.door = 'open';
  }
  closeDoor(){
    this.door = 'closed';
  }
}
var car = new Car('red');

```

The ES6 class pattern is still relatively new, and there's still a lot of debate on whether to use this or the other patterns like the prototypal and pseudoclassical pattern.

61. 20 Must Know JavaScript Interview Q&As

JavaScript: Love it, hate it, chances are you'll need to know something about it if you're working as a developer. If you're like me, you probably avoid it at all costs and instead rely on one of the many fabulous JavaScript libraries to do your bidding. However, if the role you're going for is frontend-heavy, you'll likely need to do know a good bit more than that.

With front end developers pulling in a median of \$66/k per year (and Sr. developers pulling in much higher) [1](#), it might be worth your time to extend your knowledge of JavaScript before your next interview and make sure you at least have the basics down.



Median wage of JavaScript developer.

Most of the questions below are general and conceptual, but you should also be prepared for whiteboard questions, which will require you to write out code by hand. This certainly requires much more skill than discussing JavaScript concepts at a higher level, and it will immediately show if your skills aren't up to par. But being able to clearly discuss complex topics is a skill in and of itself, so be sure to take some time to prep for the verbal component of the interview as well as the technical.

Describe an instance of prototypal inheritance in JavaScript?

- Contrary to many object oriented languages, JavaScript is classless and does not support classical inheritance. Because there are no classes, each object is a prototype of another object, and inherits the properties defined in the prototype.
- In the example below, you can see how 'dog' is the prototype for 'beagle', which in turn is the prototype for 'spot'. Any properties defined in dog, such as number of legs, will be inherited by Spot the dog.

```
1 var dog = { legs: 4 };
2 var beagle = Object.create(dog);
3 var spot = Object.create(beagle);
```

What are closures and how are they used?

- This is a mouth full to try and explain, and you'll probably need to whiteboard an example if this question comes up. The simplest definition I could find comes from [Jibbering.com blog](#):
 - "A "closure" is an expression (typically a function) that can have free variables together with an environment that binds those variables (that "closes" the expression)."
- If you're like most people and that definition means nothing to you, [MDN](#) has an excellent section on closures that is chalk full of examples. Even if you've never recognized them as closures, you've probably seen or even used these patterns before.

Can you use x === "object" to test if x is an object?

- In short, yes, but you must take into account the fact that null is considered an object in JavaScript. Even if x is null, `console.log(typeof x === "object")` will log true instead of false.
- To account for this, you must also test whether or not x is null by including the following:
 - `console.log((x !== null) && (typeof x === "object"));`

What happens when you don't declare a variable in Javascript?

- If you don't explicitly declare a variable, you risk creating an implied global variable. This is a problem, as you will be unable to create multiple instances of that object, as each new instance will overwrite the data from the last.
- In general, global variables should be used only in very specific situations and are typically not recommended, as they can lead to a lot of odd side effects that may be difficult to track down.

What is the difference between == and === ?

- '==' evaluates equality of the value, while '===' evaluates equality of type and value.

What datatypes are supported in Javascript?

- Undefined
- Number
- String
- Boolean
- Object
- Function
- Null

What would "1"+2+3 and 1+2+"3" evaluate to, respectively?

- When the first character is a string, the remaining characters will be converted into a single string, rendering 123.
- If the string is at the end, the initial characters will perform normal mathematical functions before converting into a string, resulting in 33.

What is the difference between window.onload and the jQuery \$(document).ready() method?

- The window.onload method occurs after all the page elements have loaded(HTML, CSS, images), which can result in a delay.
- The \$(document).ready() method begins to run code as soon as the Document Object Model (DOM) is loaded, which should be faster and less prone to loading errors across different browsers.

Explain the concept of unobtrusive Javascript?

- Unobtrusive JavaScript is basically a JavaScript methodology that seeks to overcome browser inconsistencies by separating page functionality from structure. The basic premise of unobtrusive JavaScript is that page functionality should be maintained even when JavaScript is unavailable on a user's browser.

What is the difference between a null value and an undefined value?

- Null is used to assign an empty value to a variable and needs to be assigned manually.
- Undefined values result when you declare a variable without assigning it a value. Undefined will be the default whenever you don't explicitly assign a value.

Which conditional statements will JavaScript support?

- if statement
- if else statement
- if else if else statement
- switch statement

What is NaN?

- Nan is literally "Not-a-Number". NaN usually results when either the result or one of the values in an operation

is non-numeric. Even though NaN is not a number, `console.log(typeof NaN === "number");` logs true, while NaN compared to anything else (including NaN) logs false. The only real way to test if a value is equal to NaN is with the function `isNaN()`.

What types of pop-up boxes you can create in JavaScript and how can you create them?

- Alert, prompt, and confirm boxes are the three main pop-up boxes that can be created in JavaScript.
- Alert boxes are used to alert the user to important info regarding the site and require the user to click 'OK' in order to proceed.
- Alert boxes are created using the following syntax:
 - `window.alert("Text");`
- Prompt boxes prompt the user to input some form of data before proceeding.
- Prompt boxes are created using the following syntax:
 - `window.prompt("Prompt Text","Default value");`
- Confirm boxes are used when for verification, such as agreeing to a terms of service, rendering a TRUE value when the user clicks 'OK'.
- Confirm boxes are created with the following syntax:
 - `window.confirm("Confirm text here.");`

Which boolean operators are in JavaScript?

- 'and' operator: `&&`
- 'or' operator: `||`
- 'not' operator: `!`

Why would you include 'use strict' at the beginning of a JavaScript source file?

- Using strict mode enforces stricter error handling when running your code. It essentially raises errors that would have otherwise failed silently. Using strict mode can help you avoid simple mistakes like creating accidental globals, undefined values, or duplicate property names. This is typically a good idea when writing JavaScript, as such errors can create a lot of frustrating side effects and be difficult to track down.

Explain the meaning of the keyword 'this' in JavaScript functions

- The keyword 'this' in JavaScript refers to the object that a function is a method of. If it's not specified, it will default to the global object, the `window`.
- In the example below, you can see that 'this' refers to the `box` object when it is applied to the `width` function. When no object is passed in, it will default to the browser window.

```

1 var box = { outerWidth: 50 };
2
3 function width() {
4     alert(this.outerWidth);
5 }
6
7 width.apply(box); // Returns 50, the outerWidth of the box object
8
9 width(); // Returns the outerWidth of the browser window

```

What does a timer do and how would you implement one?

- Setting timers allows you to execute your code at predefined times or intervals.
- This can be achieved through two main methods: setInterval(); and setTimeout();
- setInterval() accepts a function and a specified number of milliseconds.
 - ex) setInterval(function(){alert("Hello, World!")},10000) will alert the "Hello, World!" function every 10 seconds.
- setTimeout() also accepts a function, followed by milliseconds. setTimeout() will only execute the function once after the specified amount of time, and will not reoccur in intervals.

Is there automatic type conversion in JavaScript?

- Yes, JavaScript supports automatic type conversion. In the example below, JavaScript is expecting a string. When it receives a fixnum as an operand, it's automatically converted to a string.
 - ex)
 - 5 + " cats" = "5 cats"

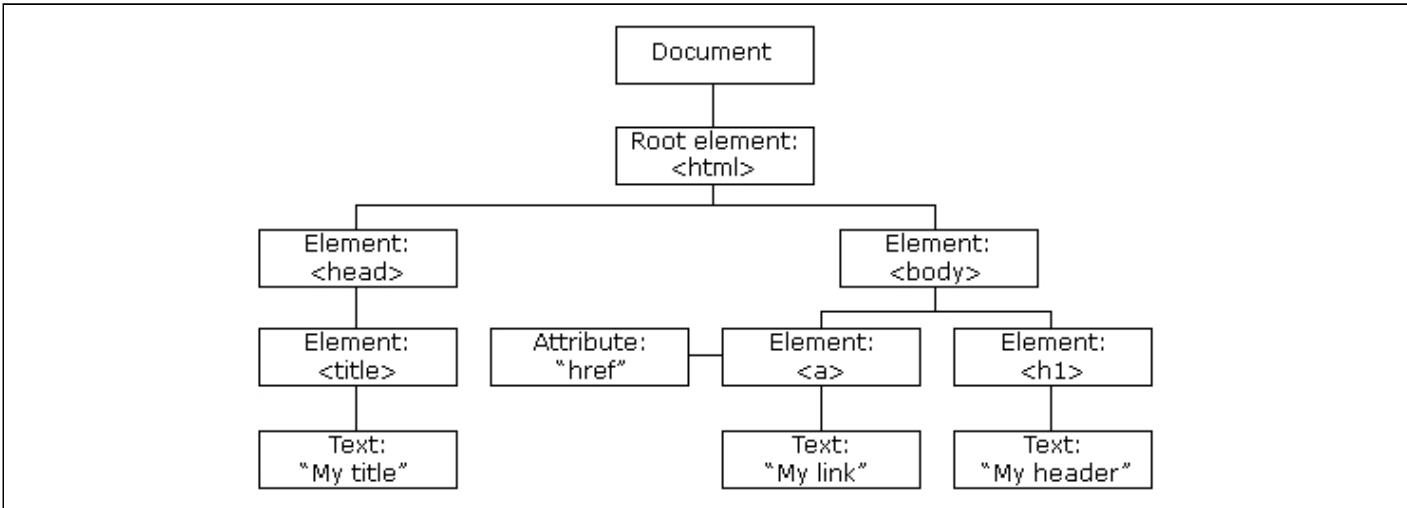
Explain the HTML DOM and its relevance to JavaScript

- The Document Object Model allows JavaScript to obtain information about a page and manipulate it by adding or removing HTML elements, or by altering their attributes. The DOM can be viewed as a programming interface for HTML, whereby you can access and alter its structure, attributes, styles, and events. According to Mozilla Developer Network (MDN), "The DOM provides a representation of the document as a structured group of nodes and objects that have properties and methods. Essentially, it connects web pages to scripts or programming languages."
- Using the below example from MDN, the code would return all <p> elements within the HTML document.

```

1 paragraphs = document.getElementsByTagName("P");
2 //paragraphs[0] is the first <p> element
3 //paragraphs[1] is the second <p> element, etc.
4 alert(paragraphs[0].nodeName);

```



A DOM tree structure, from MDN

What are your favorite JavaScript libraries?

- This is one of those general open-ended questions interviewers love to throw out. The only correct answer here is one that shows you're familiar with and have an opinion on the various JavaScript libraries. You could discuss the pros and cons of using raw JavaScript vs. jQuery, or describe an instance in which you'd use one library over another. Visit jsdb.io for a complete rundown.

Get Ready to Interact with Your Interviewer

This list is certainly not exhaustive, but is instead meant to act as a refresher and highlight any weaknesses. Aside from being able to discuss the topics listed above, you should also be prepared to create whiteboard examples for each one. If you're a pro and use JavaScript everyday, you'll breeze through these. If you're lazy like me and only rely on the libraries, you could probably benefit from a review or even a refresher course. While it's true that there's no substitute for experience, an excellent programmer can easily disqualify him or herself by giving a poor verbal interview. Likewise, a less experienced candidate can gain an edge by demonstrating solid communication skills and an eagerness to learn and be part of a team. Wherever you fall on this spectrum, the verbal component of an interview is always vital and shouldn't be discounted.

Footnotes

- [1. \[http://www.payscale.com/research/US/Job=Front_End_Developer_%2F_Engineer/Salary\]\(http://www.payscale.com/research/US/Job=Front_End_Developer_%2F_Engineer/Salary\)](http://www.payscale.com/research/US/Job=Front_End_Developer_%2F_Engineer/Salary)



62. 25 Essential JavaScript Interview Questions*

What is a potential pitfall with using `typeof bar === "object"` to determine if `bar` is an object? How can this pitfall be avoided?

Hide answer

Although `typeof bar === "object"` is a reliable way of checking if `bar` is an object, the surprising gotcha in JavaScript is that `null` is *also* considered an object!

Therefore, the following code will, to the surprise of most developers, log `true` (not `false`) to the console:

```
CODE
var bar = null;
console.log(typeof bar === "object"); // logs true!
```

As long as one is aware of this, the problem can easily be avoided by also checking if `bar` is `null`:

```
CODE
console.log((bar !== null) && (typeof bar === "object")); // logs false
```

To be entirely thorough in our answer, there are two other things worth noting:

First, the above solution will return `false` if `bar` is a function. In most cases, this is the desired behavior, but in situations where you want to also return `true` for functions, you could amend the above solution to be:

```
CODE
console.log((bar !== null) && ((typeof bar === "object") || (typeof bar === "function")));
```

Second, the above solution will return `true` if `bar` is an array (e.g., if `var bar = []`). In most cases, this is the desired behavior, since arrays are indeed objects, but in situations where you want to also `false` for arrays, you could amend the above solution to be:

```
CODE
console.log((bar !== null) && (typeof bar === "object") && (toString.call(bar) !== "[object Array]"));
```

Or, if you're using jQuery:

```
CODE
console.log((bar !== null) && (typeof bar === "object") && (! $.isArray(bar)));
```

[Comment](#)

What will the code below output to the console and why?

```
CODE
(function(){
  var a = b = 3;
})();

console.log("a defined? " + (typeof a !== 'undefined'));
console.log("b defined? " + (typeof b !== 'undefined'));
```

[View the answer ↗](#)

Since both `a` and `b` are defined within the enclosing scope of the function, and since the line they are on begins with the `var` keyword, most JavaScript developers would expect `typeof a` and `typeof b` to both be `undefined` in the above example.

However, that is *not* the case. The issue here is that most developers *incorrectly* understand the statement `var a = b = 3;` to be shorthand for:

CODE

```
var b = 3;
var a = b;
```

But in fact, `var a = b = 3;` is actually shorthand for:

CODE

```
b = 3;
var a = b;
```

As a result (if you are *not* using strict mode), the output of the code snippet would be:

CODE

```
a defined? false
b defined? true
```

But how can `b` be defined *outside* of the scope of the enclosing function? Well, since the statement `var a = b = 3;` is shorthand for the statements `b = 3;` and `var a = b;`, `b` ends up being a global variable (since it is not preceded by the `var` keyword) and is therefore still in scope even outside of the enclosing function.

Note that, in strict mode (i.e., with `use strict`), the statement `var a = b = 3;` will generate a runtime error of `ReferenceError: b is not defined`, thereby avoiding any headfakes/bugs that might otherwise result. (Yet another prime example of why you should use `use strict` as a matter of course in your code!)

[Comment](#)

What will the code below output to the console and why?

CODE

```
var myObject = {
  foo: "bar",
  func: function() {
    var self = this;
    console.log("outer func: this.foo = " + this.foo);
    console.log("outer func: self.foo = " + self.foo);
    (function() {
      console.log("inner func: this.foo = " + this.foo);
      console.log("inner func: self.foo = " + self.foo);
    })();
  }
};
myObject.func();
```

[View the answer ↗](#)

The above code will output the following to the console:

CODE

```
outer func: this.foo = bar
outer func: self.foo = bar
inner func: this.foo = undefined
inner func: self.foo = bar
```

In the outer function, both `this` and `self` refer to `myObject` and therefore both can properly reference and access `foo`.

In the inner function, though, `this` no longer refers to `myObject`. As a result, `this.foo` is undefined in the inner function, whereas the reference to the local variable `self` remains in scope and is accessible there. (Prior to ECMA 5, `this` in the inner function would refer to the global `window` object; whereas, as of ECMA 5, `this` in the inner function would be `undefined`.)

[Comment](#)

What is the significance of, and reason for, wrapping the entire content of a JavaScript source file in a function block?

View the answer 

This is an increasingly common practice, employed by many popular JavaScript libraries (jQuery, Node.js, etc.). This technique creates a closure around the entire contents of the file which, perhaps most importantly, creates a private namespace and thereby helps avoid potential name clashes between different JavaScript modules and libraries.

Another feature of this technique is to allow for an easily referenceable (presumably shorter) alias for a global variable. This is often used, for example, in jQuery plugins. jQuery allows you to disable the `$` reference to the jQuery namespace, using `jQuery.noConflict()`. If this has been done, your code can still use `$` employing this closure technique, as follows:

```
(function($) { /* jQuery plugin code referencing $ */ })(jQuery);
```

CODE

[Comment](#)

What is the significance, and what are the benefits, of including '`use strict`' at the beginning of a JavaScript source file?

View the answer 

The short and most important answer here is that `use strict` is a way to voluntarily enforce stricter parsing and error handling on your JavaScript code at runtime. Code errors that would otherwise have been ignored or would have failed silently will now generate errors or throw exceptions. In general, it is a good practice.

Some of the key benefits of strict mode include:

- **Makes debugging easier.** Code errors that would otherwise have been ignored or would have failed silently will now generate errors or throw exceptions, alerting you sooner to problems in your code and directing you more quickly to their source.
- **Prevents accidental globals.** Without strict mode, assigning a value to an undeclared variable automatically creates a global variable with that name. This is one of the most common errors in JavaScript. In strict mode, attempting to do so throws an error.
- **Eliminates this coercion.** Without strict mode, a reference to a `this` value of null or undefined is automatically coerced to the global. This can cause many headfakes and pull-out-your-hair kind of bugs. In strict mode, referencing a `this` value of null or undefined throws an error.
- **Disallows duplicate property names or parameter values.** Strict mode throws an error when it detects a duplicate named property in an object (e.g., `var object = {foo: "bar", foo: "baz"};`) or a duplicate named argument for a function (e.g., `function foo(val1, val2, val1){}`), thereby catching what is almost certainly a bug in your code that you might otherwise have wasted lots of time tracking down.

- **Makes eval() safer.** There are some differences in the way `eval()` behaves in strict mode and in non-strict mode. Most significantly, in strict mode, variables and functions declared inside of an `eval()` statement are *not* created in the containing scope (they *are* created in the containing scope in non-strict mode, which can also be a common source of problems).
- **Throws error on invalid usage of delete.** The `delete` operator (used to remove properties from objects) cannot be used on non-configurable properties of the object. Non-strict code will fail silently when an attempt is made to delete a non-configurable property, whereas strict mode will throw an error in such a case.

[Comment](#)

Consider the two functions below. Will they both return the same thing? Why or why not?

```
function foo1()
{
  return {
    bar: "hello"
  };
}

function foo2()
{
  return
  {
    bar: "hello"
  };
}
```

CODE

[View the answer ↗](#)

Surprisingly, these two functions will *not* return the same thing. Rather:

```
console.log("foo1 returns:");
console.log(foo1());
console.log("foo2 returns:");
console.log(foo2());
```

CODE

will yield:

```
foo1 returns:
Object {bar: "hello"}
foo2 returns:
undefined
```

CODE

Not only is this surprising, but what makes this particularly gnarly is that `foo2()` returns `undefined` without any error being thrown.

The reason for this has to do with the fact that semicolons are technically optional in JavaScript (although omitting them is generally really bad form). As a result, when the line containing the `return` statement (with nothing else on the line) is encountered in `foo2()`, a semicolon is automatically inserted immediately after the `return` statement.

No error is thrown since the remainder of the code is perfectly valid, even though it doesn't ever get invoked or do

anything (it is simply an unused code block that defines a property `bar` which is equal to the string "hello").

This behavior also argues for following the convention of placing an opening curly brace at the end of a line in JavaScript, rather than on the beginning of a new line. As shown here, this becomes more than just a stylistic preference in JavaScript.

[Comment](#)

What is `Nan`? What is its type? How can you reliably test if a value is equal to `Nan`?

[View the answer ↗](#)

The `Nan` property represents a value that is "not a number". This special value results from an operation that could not be performed either because one of the operands was non-numeric (e.g., `"abc" / 4`), or because the result of the operation is non-numeric (e.g., an attempt to divide by zero).

While this seems straightforward enough, there are a couple of somewhat surprising characteristics of `Nan` that can result in hair-pulling bugs if one is not aware of them.

For one thing, although `Nan` means "not a number", its type is, believe it or not, `Number`:

```
console.log(typeof Nan === "number"); // logs "true"
```

CODE

Additionally, `Nan` compared to anything — even itself! — is false:

```
console.log(Nan === Nan); // logs "false"
```

CODE

A *semi-reliable* way to test whether a number is equal to `Nan` is with the built-in function `isNaN()`, but even using [`isNaN\(\)` is an imperfect solution](#).

A better solution would either be to use `value !== value`, which would *only* produce true if the value is equal to `Nan`. Also, ES6 offers a new [`Number.isNaN\(\)`](#) function, which is a different and more reliable than the old global `isNaN()` function.

[Comment](#)

What will the code below output? Explain your answer.

```
console.log(0.1 + 0.2);
console.log(0.1 + 0.2 == 0.3);
```

CODE

[View the answer ↗](#)

An educated answer to this question would simply be: "You can't be sure. it might print out "0.3" and "true", or it might not. Numbers in JavaScript are all treated with floating point precision, and as such, may not always yield the expected results."

The example provided above is classic case that demonstrates this issue. Surprisingly, it will print out:

CODE

```
0.30000000000000004
false
```

Comment

Discuss possible ways to write a function `isInteger(x)` that determines if `x` is an integer.

[View the answer ↗](#)

This may sound trivial and, in fact, it is trivial with ECMAScript 6 which introduces a new `Number.isInteger()` function for precisely this purpose. However, prior to ECMAScript 6, this is a bit more complicated, since no equivalent of the `Number.isInteger()` method is provided.

The issue is that, in the ECMAScript specification, integers only exist conceptually; i.e., numeric values are *always* stored as floating point values.

With that in mind, the *simplest and cleanest* pre-ECMAScript-6 solution (which is also sufficiently robust to return `false` even if a non-numeric value such as a string or `null` is passed to the function) would be the following:

CODE

```
function isInteger(x) { return (x^0) === x; }
```

The following solution would also work, although not as elegant as the one above:

CODE

```
function isInteger(x) { return Math.round(x) === x; }
```

Note that `Math.ceil()` or `Math.floor()` could be used equally well (instead of `Math.round()`) in the above implementation.

Or alternatively:

CODE

```
function isInteger(x) { return (typeof x === 'number') && (x % 1 === 0); }
```

One fairly common **incorrect** solution is the following:

CODE

```
function isInteger(x) { return parseInt(x, 10) === x; }
```

While this `parseInt`-based approach will work well for *many* values of `x`, once `x` becomes quite large, it will fail to work properly. The problem is that `parseInt()` coerces its first parameter to a string before parsing digits. Therefore, once the number becomes sufficiently large, its string representation will be presented in exponential form (e.g., `1e+21`). Accordingly, `parseInt()` will then try to parse `1e+21`, but will stop parsing when it reaches the `e` character and will therefore return a value of `1`. Observe:

CODE

```
> String(100000000000000000000)
'1e+21'

> parseInt(100000000000000000000, 10)
1

> parseInt(100000000000000000000, 10) === 100000000000000000000
false
```

Comment

In what order will the numbers 1-4 be logged to the console when the code below is executed? Why?

CODE

```
(function() {
    console.log(1);
    setTimeout(function(){console.log(2)}, 1000);
    setTimeout(function(){console.log(3)}, 0);
    console.log(4);
})();
```

[View the answer ↗](#)

The values will be logged in the following order:

CODE

```
1
4
3
2
```

Let's first explain the parts of this that are presumably more obvious:

- 1 and 4 are displayed first since they are logged by simple calls to `console.log()` without any delay
- 2 is displayed after 3 because 2 is being logged after a delay of 1000 msecs (i.e., 1 second) whereas 3 is being logged after a delay of 0 msecs.

OK, fine. But if 3 is being logged after a delay of 0 msecs, doesn't that mean that it is being logged right away? And, if so, shouldn't it be logged *before* 4, since 4 is being logged by a later line of code?

The answer has to do with properly understanding [JavaScript events and timing](#).

The browser has an event loop which checks the event queue and processes pending events. For example, if an event happens in the background (e.g., a script `onload` event) while the browser is busy (e.g., processing an `onclick`), the event gets appended to the queue. When the onclick handler is complete, the queue is checked and the event is then handled (e.g., the `onload` script is executed).

Similarly, `setTimeout()` also puts execution of its referenced function into the event queue if the browser is busy.

When a value of zero is passed as the second argument to `setTimeout()`, it attempts to execute the specified function "as soon as possible". Specifically, execution of the function is placed on the event queue to occur on the next timer tick. Note, though, that this is *not* immediate; the function is not executed until the next tick. That's why in

In the above example, the call to `console.log(4)` occurs before the call to `console.log(3)` (since the call to `console.log(3)` is invoked via `setTimeout`, so it is slightly delayed).

[Comment](#)

Write a simple function (less than 80 characters) that returns a boolean indicating whether or not a string is a [palindrome](#).

[View the answer ↗](#)

The following one line function will return `true` if `str` is a palindrome; otherwise, it returns `false`.

```
function isPalindrome(str) {
    str = str.replace(/\W/g, '').toLowerCase();
    return (str == str.split('').reverse().join(''));
}
```

CODE

For example:

```
console.log(isPalindrome("level"));           // logs 'true'
console.log(isPalindrome("levels"));          // logs 'false'
console.log(isPalindrome("A car, a man, a maraca")); // logs 'true'
```

CODE

[Comment](#)

Write a `sum` method which will work properly when invoked using either syntax below.

```
console.log(sum(2,3));    // Outputs 5
console.log(sum(2)(3));  // Outputs 5
```

CODE

[View the answer ↗](#)

There are (at least) two ways to do this:

METHOD 1

```
function sum(x) {
    if (arguments.length == 2) {
        return arguments[0] + arguments[1];
    } else {
        return function(y) { return x + y; };
    }
}
```

CODE

In JavaScript, functions provide access to an `arguments` object which provides access to the actual arguments passed to a function. This enables us to use the `length` property to determine at runtime the number of arguments passed to the function.

If two arguments are passed, we simply add them together and return.

Otherwise, we assume it was called in the form `sum(2)(3)`, so we return an anonymous function that adds together the argument passed to `sum()` (in this case 2) and the argument passed to the anonymous function (in this case 3).

METHOD 2

```
function sum(x, y) {
  if (y !== undefined) {
    return x + y;
  } else {
    return function(y) { return x + y; };
  }
}
```

CODE

When a function is invoked, JavaScript does not require the number of arguments to match the number of arguments in the function definition. If the number of arguments passed exceeds the number of arguments in the function definition, the excess arguments will simply be ignored. On the other hand, if the number of arguments passed is less than the number of arguments in the function definition, the missing arguments will have a value of `undefined` when referenced within the function. So, in the above example, by simply checking if the 2nd argument is `undefined`, we can determine which way the function was invoked and proceed accordingly.

[Comment](#)

Consider the following code snippet:

```
for (var i = 0; i < 5; i++) {
  var btn = document.createElement('button');
  btn.appendChild(document.createTextNode('Button ' + i));
  btn.addEventListener('click', function(){ console.log(i); });
  document.body.appendChild(btn);
}
```

CODE

(a) What gets logged to the console when the user clicks on "Button 4" and why?

(b) Provide one or more alternate implementations that will work as expected.

View the answer ↗

(a) No matter what button the user clicks the number 5 will *always* be logged to the console. This is because, at the point that the `onclick` method is invoked (for *any* of the buttons), the `for` loop has already completed and the variable `i` already has a value of 5. (Bonus points for the interviewee if they know enough to talk about how execution contexts, variable objects, activation objects, and the internal "scope" property contribute to the closure behavior.)

(b) The key to making this work is to capture the value of `i` at each pass through the `for` loop by passing it into a newly created function object. Here are three possible ways to accomplish this:

CODE

```

for (var i = 0; i < 5; i++) {
  var btn = document.createElement('button');
  btn.appendChild(document.createTextNode('Button ' + i));
  btn.addEventListener('click', (function(i) {
    return function() { console.log(i); };
  })(i));
  document.body.appendChild(btn);
}

```

Alternatively, you could wrap the entire call to `btn.addEventListener` in the new anonymous function:

CODE

```

for (var i = 0; i < 5; i++) {
  var btn = document.createElement('button');
  btn.appendChild(document.createTextNode('Button ' + i));
  (function (i) {
    btn.addEventListener('click', function() { console.log(i); });
  })(i);
  document.body.appendChild(btn);
}

```

Or, we could replace the `for` loop with a call to the array object's native `forEach` method:

CODE

```

['a', 'b', 'c', 'd', 'e'].forEach(function (value, i) {
  var btn = document.createElement('button');
  btn.appendChild(document.createTextNode('Button ' + i));
  btn.addEventListener('click', function() { console.log(i); });
  document.body.appendChild(btn);
});

```

[Comment](#)

What will the code below output to the console and why?

CODE

```

var arr1 = "john".split('');
var arr2 = arr1.reverse();
var arr3 = "jones".split('');
arr2.push(arr3);
console.log("array 1: length=" + arr1.length + " last=" + arr1.slice(-1));
console.log("array 2: length=" + arr2.length + " last=" + arr2.slice(-1));

```

[View the answer ↗](#)

The logged output will be:

CODE

```

"array 1: length=5 last=j,o,n,e,s"
"array 2: length=5 last=j,o,n,e,s"

```

`arr1` and `arr2` are the same after the above code is executed for the following reasons:

- Calling an array object's `reverse()` method doesn't only *return* the array in reverse order, it also reverses the

order of the array *itself* (i.e., in this case, `arr1`).

- The `reverse()` method returns a reference to the array itself (i.e., in this case, `arr1`). As a result, `arr2` is simply a reference to (rather than a copy of) `arr1` . Therefore, when anything is done to `arr2` (i.e., when we invoke `arr2.push(arr3);`), `arr1` will be affected as well since `arr1` and `arr2` are simply references to the same object.

And a couple of side points here that can sometimes trip someone up in answering this question:

- Passing an array to the `push()` method of another array pushes that *entire* array as a *single* element onto the end of the array. As a result, the statement `arr2.push(arr3);` adds `arr3` in its entirety as a single element to the end of `arr2` (i.e., it does *not* concatenate the two arrays, that's what the `concat()` method is for).
- Like Python, JavaScript honors negative subscripts in calls to array methods like `slice()` as a way of referencing elements at the end of the array; e.g., a subscript of `-1` indicates the last element in the array, and so on.

[Comment](#)

What will the code below output to the console and why ?

```
CODE
console.log(1 + "2" + "2");
console.log(1 + +"2" + "2");
console.log(1 + -"1" + "2");
console.log(+ "1" + "1" + "2");
console.log( "A" - "B" + "2");
console.log( "A" - "B" + 2);
```

View the answer [â†'](#)

The above code will output the following to the console:

```
CODE
"122"
"32"
"02"
"112"
"NaN2"
NaN
```

Here's whyâ€¦

The fundamental issue here is that JavaScript (ECMAScript) is a loosely typed language and it performs automatic type conversion on values to accommodate the operation being performed. Let's see how this plays out with each of the above examples.

Example 1: `1 + "2" + "2"` Outputs: "122" **Explanation:** The first operation to be performed in `1 + "2"` .

Since one of the operands (`"2"`) is a string, JavaScript assumes it needs to perform string concatenation and therefore converts the type of `1` to `"1"` , `1 + "2"` yields `"12"` . Then, `"12" + "2"` yields `"122"` .

Example 2: `1 + +"2" + "2"` Outputs: "32" **Explanation:** Based on order of operations, the first operation to be performed is `+"2"` (the extra `+` before the first `"2"` is treated as a unary operator). Thus, JavaScript converts the type of `"2"` to numeric and then applies the unary `+` sign to it (i.e., treats it as a positive number). As a result, the

next operation is now `1 + 2` which of course yields `3`. But then, we have an operation between a number and a string (i.e., `3` and `"2"`), so once again JavaScript converts the type of the numeric value to a string and performs string concatenation, yielding `"32"`.

Example 3: `1 + -"1" + "2"` **Outputs:** `"02"` **Explanation:** The explanation here is identical to the prior example, except the unary operator is `-` rather than `+`. So `"1"` becomes `1`, which then becomes `-1` when the `-` is applied, which is then added to `1` yielding `0`, which is then converted to a string and concatenated with the final `"2"` operand, yielding `"02"`.

Example 4: `+"1" + "1" + "2"` **Outputs:** `"112"` **Explanation:** Although the first `"1"` operand is typecast to a numeric value based on the unary `+` operator that precedes it, it is then immediately converted back to a string when it is concatenated with the second `"1"` operand, which is then concatenated with the final `"2"` operand, yielding the string `"112"`.

Example 5: `"A" - "B" + "2"` **Outputs:** `"NaN2"` **Explanation:** Since the `-` operator can not be applied to strings, and since neither `"A"` nor `"B"` can be converted to numeric values, `"A" - "B"` yields `NaN` which is then concatenated with the string `"2"` to yield `"NaN2"`.

Example 6: `"A" - "B" + 2` **Outputs:** `NaN` **Explanation:** As explained in the previous example, `"A" - "B"` yields `NaN`. But any operator applied to `NaN` with any other numeric operand will still yield `NaN`.

[Comment](#)

The following recursive code will cause a stack overflow if the array list is too large. How can you fix this and still retain the recursive pattern?

```
CODE
var list = readHugeList();

var nextListItem = function() {
    var item = list.pop();

    if (item) {
        // process the list item...
        nextListItem();
    }
};
```

[View the answer ↗](#)

The potential stack overflow can be avoided by modifying the `nextListItem` function as follows:

```
CODE
var list = readHugeList();

var nextListItem = function() {
    var item = list.pop();

    if (item) {
        // process the list item...
        setTimeout( nextListItem, 0);
    }
};
```

The stack overflow is eliminated because the event loop handles the recursion, not the call stack. When

`nextListItem` runs, if `item` is not null, the timeout function (`nextListItem`) is pushed to the event queue and the function exits, thereby leaving the call stack clear. When the event queue runs its timed-out event, the next `item` is processed and a timer is set to again invoke `nextListItem`. Accordingly, the method is processed from start to finish without a direct recursive call, so the call stack remains clear, regardless of the number of iterations.

[Comment](#)

What is a "closure" in JavaScript? Provide an example.

[View the answer ↗](#)

A closure is an inner function that has access to the variables in the outer (enclosing) function's scope chain. The closure has access to variables in three scopes; specifically: (1) variable in its own scope, (2) variables in the enclosing function's scope, and (3) global variables.

Here is a simple example:

CODE

```
var globalVar = "xyz";

(function outerFunc(outerArg) {
    var outerVar = 'a';

    (function innerFunc(innerArg) {
        var innerVar = 'b';

        console.log(
            "outerArg = " + outerArg + "\n" +
            "innerArg = " + innerArg + "\n" +
            "outerVar = " + outerVar + "\n" +
            "innerVar = " + innerVar + "\n" +
            "globalVar = " + globalVar);

    })(456);
})(123);
```

In the above example, variables from `innerFunc`, `outerFunc`, and the global namespace are **all** in scope in the `innerFunc`. The above code will therefore produce the following output:

CODE

```
outerArg = 123
innerArg = 456
outerVar = a
innerVar = b
globalVar = xyz
```

[Comment](#)

What will be the output of the following code:

CODE

```
for (var i = 0; i < 5; i++) {
    setTimeout(function() { console.log(i); }, i * 1000 );
}
```

Explain your answer. How could the use of closures help here?

[View the answer ↗](#)

The code sample shown will **not** display the values 0, 1, 2, 3, and 4 as might be expected; rather, it will display 5, 5, 5, 5, and 5.

The reason for this is that each function executed within the loop will be executed *after* the entire loop has completed and *all* will therefore reference the *last* value stored in `i`, which was 5.

[**Closures**](#) can be used to prevent this problem by creating a unique scope for each iteration, storing each unique value of the variable within its scope, as follows:

```
for (var i = 0; i < 5; i++) {
    (function(x) {
        setTimeout(function() { console.log(x); }, x * 1000 );
    })(i);
}
```

CODE

This will produce the presumably desired result of logging 0, 1, 2, 3, and 4 to the console.

[Comment](#)

What would the following lines of code output to the console?

```
console.log("0 || 1 = "+(0 || 1));
console.log("1 || 2 = "+(1 || 2));
console.log("0 && 1 = "+(0 && 1));
console.log("1 && 2 = "+(1 && 2));
```

CODE

Explain your answer.

[View the answer ↗](#)

The code will output the following four lines:

```
0 || 1 = 1
1 || 2 = 1
0 && 1 = 0
1 && 2 = 2
```

CODE

In JavaScript, both `||` and `&&` are logical operators that return the first fully-determined "logical value" when evaluated from left to right.

The or (`||`) operator. In an expression of the form `X||Y`, `X` is first evaluated and interpreted as a boolean value. If this boolean value is `true`, then `true` (1) is returned and `Y` is not evaluated, since the "or" condition has already been satisfied. If this boolean value is "false", though, we still don't know if `X||Y` is true or false until we evaluate `Y`, and interpret it as a boolean value as well.

Accordingly, `0 || 1` evaluates to true (1), as does `1 || 2`.

The and (&&) operator. In an expression of the form `X&&Y` , `X` is first evaluated and interpreted as a boolean value. If this boolean value is `false` , then `false` (0) is returned and `Y` is not evaluated, since the "and" condition has already failed. If this boolean value is "true", though, we still don't know if `X&&Y` is true or false until we evaluate `Y` , and interpret it as a boolean value as well.

However, the interesting thing with the `&&` operator is that when an expression is evaluated as "true", then the expression itself is returned. This is fine, since it counts as "true" in logical expressions, but also can be used to return that value when you care to do so. This explains why, somewhat surprisingly, `1 && 2` returns 2 (whereas you might expect it to return `true` or 1).

[Comment](#)

What will be the output when the following code is executed? Explain.

```
CODE
console.log(false == '0')
console.log(false === '0')
```

[View the answer ↗](#)

The code will output:

```
CODE
true
false
```

In JavaScript, there are two sets of equality operators. The triple-equal operator `====` behaves like any traditional equality operator would: evaluates to true if the two expressions on either of its sides have the same type and the same value. The double-equal operator, however, tries to coerce the values before comparing them. It is therefore generally good practice to use the `====` rather than `==` . The same holds true for `!==` vs `!=` .

[Comment](#)

What is the output out of the following code? Explain your answer.

```
CODE
var a={},  
    b={key:'b'},  
    c={key:'c'};  
  
a[b]=123;  
a[c]=456;  
  
console.log(a[b]);
```

[View the answer ↗](#)

The output of this code will be 456 (not 123).

The reason for this is as follows: When setting an object property, JavaScript will implicitly **stringify** the parameter value. In this case, since `b` and `c` are both objects, they will *both* be converted to "[object Object]" . As a result, `a[b]` and `a[c]` are both equivalent to `a["[object Object]"]` and can be used interchangeably. Therefore, setting or referencing `a[c]` is precisely the same as setting or referencing `a[b]` .

[Comment](#)

What will the following code output to the console:

```
console.log((function f(n){return ((n > 1) ? n * f(n-1) : n)})(10));
```

CODE

Explain your answer.

[View the answer ↗](#)

The code will output the value of 10 factorial (i.e., 10!, or 3,628,800).

Here's why:

The named function `f()` calls itself recursively, until it gets down to calling `f(1)` which simply returns `1`. Here, therefore, is what this does:

```
f(1): returns n, which is 1
f(2): returns 2 * f(1), which is 2
f(3): returns 3 * f(2), which is 6
f(4): returns 4 * f(3), which is 24
f(5): returns 5 * f(4), which is 120
f(6): returns 6 * f(5), which is 720
f(7): returns 7 * f(6), which is 5040
f(8): returns 8 * f(7), which is 40320
f(9): returns 9 * f(8), which is 362880
f(10): returns 10 * f(9), which is 3628800
```

CODE

[Comment](#)

Consider the code snippet below. What will the console output be and why?

```
(function(x) {
    return (function(y) {
        console.log(x);
    })(2)
})(1);
```

CODE

[View the answer ↗](#)

The output will be `1`, even though the value of `x` is never set in the inner function. Here's why:

As explained in our [JavaScript Hiring Guide](#), a **closure** is a function, along with all variables or functions that were in-scope at the time that the closure was created. In JavaScript, a closure is implemented as an "inner function"; i.e., a function defined within the body of another function. An important feature of closures is that an inner function still has access to the outer function's variables.

Therefore, in this example, since `x` is not defined in the inner function, the scope of the outer function is searched for a defined variable `x`, which is found to have a value of `1`.

[Comment](#)

What will the following code output to the console and why:

```
var hero = {
  _name: 'John Doe',
  getSecretIdentity: function (){
    return this._name;
  }
};

var stoleSecretIdentity = hero.getSecretIdentity;

console.log(stoleSecretIdentity());
console.log(hero.getSecretIdentity());
```

CODE

What is the issue with this code and how can it be fixed.

[View the answer ↗](#)

The code will output:

```
undefined
John Doe
```

CODE

The first `console.log` prints `undefined` because we are extracting the method from the `hero` object, so `stoleSecretIdentity()` is being invoked in the global context (i.e., the `window` object) where the `_name` property does not exist.

One way to fix the `stoleSecretIdentity()` function is as follows:

```
var stoleSecretIdentity = hero.getSecretIdentity.bind(hero);
```

CODE

[Comment](#)

Create a function that, given a DOM Element on the page, will visit the element itself and *all* of its descendants (*not just its immediate children*). For each element visited, the function should pass that element to a provided callback function.

The arguments to the function should be:

- a DOM element
- a callback function (that takes a DOM element as its argument)

[View the answer ↗](#)

Visiting all elements in a tree (DOM) is a classic [Depth-First-Search algorithm](#) application. Here's an example solution:

CODE

```

function Traverse(p_element,p_callback) {
    p_callback(p_element);
    var list = p_element.children;
    for (var i = 0; i < list.length; i++) {
        Traverse(list[i],p_callback); // recursive call
    }
}

```

Comment

* There is more to interviewing than tricky technical questions, so these are intended merely as a guide. Not every "A" candidate worth hiring will be able to answer them all, nor does answering them all guarantee an "A" candidate. At the end of the day, [hiring remains an art, a science](#) and a lot of work.

63. JavaScript variables hoisting in details

Variables in a program are everywhere. They are small pieces of data and logic that always interact with each other: and this activity makes the application alive.

In JavaScript an important aspect of working with variables is hoisting, which defines when a variable is accessible. If you're looking for a detailed description of this aspect, then you're in the right place. Let's begin.

1. Introduction

Hoisting is the mechanism of moving the variables and functions declaration to the top of the function scope (or global scope if outside any function).

Hoisting influences the variable life-cycle, which consists of these 3 steps:

- **Declaration** - create a new variable. E.g. `var myValue`
- **Initialization** - initialize the variable with a value. E.g. `myValue = 150`
- **Usage** - access and use the variable value. E.g. `alert(myValue)`

The process usually goes this way: first a variable should be *declared*, then *initialized* with a value and finally *used*. Let's see an example:

[Try in JS Bin](#)

CODE

```

// Declare
var strNumber;
// Initialize
strNumber = '16';
// Use
parseInt(strNumber); // => 16

```

A function can be *declared* and later *used* (or invoked) in the application. The *initialization* is omitted. For instance:

[Try in JS Bin](#)

CODE

```
// Declare
function sum(a, b) {
  return a + b;
}
// Use
sum(5, 6); // => 11
```

Everything looks simple and natural when these steps are successive: *declare -> initialize -> use*. If possible, you should apply this pattern when coding in JavaScript.

JavaScript does not follow strictly this sequence and offers more flexibility.

For instance, functions can be used before the declaration: *use -> declare*.

The following code sample first calls the function `double(5)`, and only later declares it `function double(num) {...}`:

[Try in JS Bin](#)

CODE

```
// Use
double(5); // => 10
// Declare
function double(num) {
  return num * 2;
}
```

It happens because the [function declaration](#) in JavaScript is hoisted to the top of the scope.

Hoisting affects differently:

- variable declarations: using `var`, `let` or `const` keywords
- function declarations: using `function <name>() {...}` syntax
- class declarations: using `class` keyword

Let's examine these differences in more details.

2. Function scope variables: `var`

The [variable statement](#) creates and initializes variables inside the function scope: `var myVar, myVar2 = 'Init'`.

By default a declared yet not initialized variable has `undefined` value.

Plain and simple, developers use this statement from first JavaScript versions:

[Try in JS Bin](#)

CODE

```
// Declare num variable
var num;
console.log(num); // => undefined
// Declare and initialize str variable
var str = 'Hello World!';
console.log(str); // => 'Hello World!'
```

Hoisting and `var`

Variables declared with `var` are hoisted to the top of the enclosing function scope. If the variable is accessed before declaration, it evaluates to `undefined`.

Suppose `myVariable` is accessed before declaration with `var`. In this situation the declaration is **moved to the top** of `double()` function scope and the variable is assigned with `undefined`:

[Try in JS Bin](#)

```
function double(num) {
  console.log(myVariable); // => undefined
  var myVariable;
  return num * 2;
}
double(3); // => 6
```

CODE

JavaScript will move the declaration `var myVariable` to the top of `double()` scope and interpret the code this way:

[Try in JS Bin](#)

```
function double(num) {
  var myVariable;           // moved to the top
  console.log(myVariable); // => undefined
  return num * 2;
}
double(3); // => 6
```

CODE

The `var` syntax allows not only to declare, but right away to assign an initial value: `var str = 'initial value'`. When the variable is hoisted, the declaration is moved to the top, but the initial value assignment **remains** in place:

[Try in JS Bin](#)

```
function sum(a, b) {
  console.log(myString); // => undefined
  var myString = 'Hello World';
  console.log(myString); // => 'Hello World'
  return a + b;
}
sum(16, 10); // => 26
```

CODE

`var myString` is hoisted to the top of the scope, however the initial value assignment `myString = 'Hello World'` is not affected. The above code is equivalent to the following:

[Try in JS Bin](#)

CODE

```

function sum(a, b) {
    var myString;           // moved to the top
    console.log(myString); // => undefined
    myString = 'Hello World'; // remains
    console.log(myString); // => 'Hello World'
    return a + b;
}
sum(16, 10); // => 26

```

3. Block scope variables: let

The [let statement](#) creates and initializes variables inside the block scope: `let myVar, myVar2 = 'Init'`. By default a declared yet not initialized variable has `undefined` value.

`let` is a great addition introduced by [ECMAScript 6](#), which allows to keep the code modular and encapsulated on a [block statement](#) level:

[Try in JS Bin](#)

CODE

```

if (true) {
    // Declare name block variable
    let month;
    console.log(month); // => undefined
    // Declare and initialize year block variable
    let year = 1994;
    console.log(year); // => 1994
}
// name and year or not accessible here, outside the block
console.log(year); // ReferenceError: year is not defined

```

Hoisting and let

Variables declared with `let` are hoisted to the top of the block. But when the variable is accessed before declaration, JavaScript throws an error: `ReferenceError: <variable> is not defined`.

From the declaration statement up to the beginning of the block the variable is in a *temporal dead zone* and cannot be accessed.

Let's follow an example:

[Try in JS Bin](#)

CODE

```

function isTruthy(value) {
  var myVariable = 'Value 1';
  if (value) {
    /**
     * temporal dead zone for myVariable
     */
    // Throws ReferenceError: myVariable is not defined
    console.log(myVariable);
    let myVariable = 'Value 2';
    // end of temporary dead zone for myVariable
    console.log(myVariable); // => 'Value 2'
    return true;
  }
  return false;
}
isTruthy(1); // => true

```

`myVariable` is in a temporal dead zone from `let myVariable` line up to the top of the block `if (value) {...}`. If trying to access the variable in this zone, JavaScript throws a [ReferenceError](#).

An interesting question appears: is really `myVariable` **hoisted** up to the beginning of the block, or maybe is just **not defined** in the temporal dead zone (before declaration)? The exception `ReferenceError` is thrown also when a variable is not defined at all.

If you take a look at the beginning of the function block, `var myVariable = 'Value 1'` is declaring a variable for the entire function scope. In the block `if (value) {...}`, if `let` variables would not be hoisted, then in the temporal dead zone `myVariable` would have the value '`Value 1`', which does not happen. So block variables are truly hoisted.

`let` hoisting in the entire block protects variables from modification by outer scopes, even before declaration. Generate reference errors when accessing a `let` variables in temporary dead zone ensures better coding practice: first declare - then use.

Both these restrictions are an effective approach to write better JavaScript in terms of encapsulation and code flow. This is a result of lessons based on `var` usage, where accessing the variable before declaration is a source of misunderstanding.

4. Constants: `const`

The [constant statement](#) creates and initializes constants inside the block scope: `const MY_CONST = 'Value'`, `MY_CONST2 = 'Value 2'`. Take a look at this sample:

[Try in JS Bin](#)

CODE

```

const COLOR = 'red';
console.log(COLOR); // => 'red'
const ONE = 1, HALF = 0.5;
console.log(ONE); // => 1
console.log(HALF); // => 0.5

```

When a constant is defined, it must be initialized with a value in the same `const` statement. After declaration and initialization, the value of a constant cannot be modified:

[Try in JS Bin](#)

CODE

```
const PI = 3.14;
console.log(PI); // => 3.14
PI = 2.14; // TypeError: Assignment to constant variable
```

Hoisting and const

Constants declared with `const` are hoisted to the top of the block.

The constants cannot be accessed before declaration because of the *temporal dead zone*. When accessed before declaration, JavaScript throws an error: `ReferenceError: <constant> is not defined`.

`const` hoisting has the same behavior as the variables declared with `let` statement (see [hoisting and let](#)).

Let's define a constant in a function `double()`:

[Try in JS Bin](#)

CODE

```
function double(number) {
    // temporal dead zone for TWO constant
    console.log(TWO); // ReferenceError: TWO is not defined
    const TWO = 2;
    // end of temporal dead zone
    return number * TWO;
}
double(5); // => 10
```

If `TWO` is used before the declaration, JavaScript throws an error `ReferenceError: TWO is not defined`. So the constants should be first declared and initialized, and later accessed.

5. Function declarations

The [function declaration](#) defines a function with the provided name and parameters.

An example of function declaration:

[Try in JS Bin](#)

CODE

```
function isOdd(number) {
    return number % 2 === 1;
}
isOdd(5); // => true
```

The code `function isOdd(number) {...}` is a declaration that defines a function. `isOdd()` verifies if a number is odd.

Hoisting and function declaration

Hoisting in a function declaration allows to use the function anywhere in the enclosing scope, even before the declaration. In other words, the function can be called from any place of the current or inner scopes (no `undefined`

values, temporal dead zones or reference errors).

This hoisting behavior is flexible, because you can first *use* the function and only later *declare* it. Or apply the classic scenario: first *declare* and then *use*. As you wish.

The following code from the start invokes a function, and after defines it:

[Try in JS Bin](#)

```
// Call the hoisted function
equal(1, '1'); // => false
// Function declaration
function equal(value1, value2) {
    return value1 === value2;
}
```

CODE

The code works nice because `equal()` is created by a function declaration and hoisted to the top of the scope.

Notice the **difference** between a **function declaration** `function <name>() { ... }` and a **function expression**

`var <name> = function() { ... }.` Both are used to create functions, however have different hoisting mechanisms.

The following sample demonstrates the distinction:

[Try in JS Bin](#)

```
// Call the hoisted function
addition(4, 7); // => 11
// The variable is hoisted, but is undefined
subtraction(10, 7); // TypeError: subtraction is not a function
// Function declaration
function addition(num1, num2) {
    return num1 + num2;
}
// Function expression
var subtraction = function (num1, num2) {
    return num1 - num2;
};
```

CODE

`addition` is hoisted entirely and can be called before the declaration.

However `subtraction` is declared using a variable statement (see [2.](#)) and is hoisted too, but has an `undefined` value when invoked. This scenario throws an error: `TypeError: subtraction is not a function`.

6. Class declarations

The [class declaration](#) defines a constructor function with the provided name and methods. Classes are a great addition introduced by ECMAScript 6.

Classes are built on top of the JavaScript prototypal inheritance and have some additional goodies like `super` (to access the parent class), `static` (to define static methods), `extends` (to define a child class) and more.

Take a look how to declare a class and instantiate an object:

[Try in JS Bin](#)

CODE

```

class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
  move(dX, dY) {
    this.x += dX;
    this.y += dY;
  }
}
// Create an instance
var origin = new Point(0, 0);
// Call a method
origin.move(50, 100);

```

Hoisting and class

The class declaration is hoisted up to the beginning of the block scope. But if you try to access the class before the definition, JavaScript throws `ReferenceError: <name> is not defined`. So the correct approach is first to *declare* the class and later *use* it to instantiate objects.

Hoisting in class declarations is similar to variables declared with `let` statement (see [3](#)).

Let's see what happens if a class is instantiated before declaration:

[Try in JS Bin](#)

CODE

```

// Use the Company class
// Throws ReferenceError: Company is not defined
var apple = new Company('Apple');
// Class declaration
class Company {
  constructor(name) {
    this.name = name;
  }
}
// Use correctly the Company class after declaration
var microsoft = new Company('Microsoft');

```

As expected, executing `new Company('Apple')` before the class definition throws `ReferenceError`. This is nice, because JavaScript suggests to use a good approach to first declare something and then make use of it.

Classes can be created using a [class expression](#), which involves variable declaration statements (with `var`, `let` or `const`). Let's see the following scenario:

[Try in JS Bin](#)

```
// Use the Sqaure class
console.log(typeof Square); // => 'undefined'
//Throws TypeError: Square is not a constructor
var mySquare = new Square(10);
// Class declaration using variable statement
var Square = class {
  constructor(sideLength) {
    this.sideLength = sideLength;
  }
  getArea() {
    return Math.pow(this.sideLength, 2);
  }
};
// Use correctly the Square class after declaration
var otherSquare = new Square(5);
```

The class is declared with a variable statement `var Square = class {...}`. The variable `Square` is hoisted to the top of the scope, but has an `undefined` value until the class declaration line. So the execution of `var mySquare = new Square(10)` before class declaration tries to invoke an `undefined` as a constructor and JavaScript throws `TypeError: Square is not a constructor`.

7. Final thoughts

As seen in the explanations, hoisting in JavaScript has many forms. Even if you know exactly how it works, the general advice is to code variables in a sequence of *declare* > *initialize* > *use*. ECMAScript 6 certainly suggests this approach by the way hoisting is implemented for `let`, `const` and `class`. This will save you from *unexpected* variable appearances, `undefined` and `ReferenceError`.

As an exception, sometimes functions can be invoked before the definition: an effect of function declaration hoisting. It's useful in cases when developer needs to read quickly how functions are invoked at the top of the source file, without the necessity to scroll down and read the details about function implementation.

For example, [see here](#) how this approach increases the readability of Angular controllers.

I hope you enjoyed the reading, so do not hesitate to [share](#) it. See you in my next post :).

P.S. You might also want to check out:

[Gentle explanation of 'this' keyword in JavaScript](#)

[The legend of JavaScript equality operator](#)

[JavaScript addition operator in details](#)

64. Gentle explanation of 'this' keyword in JavaScript

1. The mystery of this

A lot of time `this` keyword was a mystery for me and many starting JavaScript developers. It is a powerful feature, but requires some efforts to be understood.

From a background like *Java*, *PHP* or other *standard* language, `this` is seen as an instance of current object in the class method: no more and no less. Mostly, it cannot be used outside the method and this simple approach does not

create confusion.

In JavaScript, `this` is the current execution context of a function. Because the language has 4 function invocation types:

- function invocation: `alert('Hello World!')`
- method invocation: `console.log('Hello World!')`
- constructor invocation: `new RegExp('\\d')`
- indirect invocation: `alert.call(undefined, 'Hello World!')`

and each one defines its own context, `this` behaves slightly different than developer expects.

Moreover [strict mode](#) also affects the execution context.

The key to understanding `this` keyword is having a clear view over functions invocation and how this impacts the context.

This article is focused on the invocation explanation, how the function call influences `this` and demonstrates the common pitfalls of identifying the context.

Before starting, let's familiarize with a couple of terms:

- **Invocation** is executing the code that makes the body of a function (simply calling the function). For example `parseInt` function **invocation** is `parseInt('15')`.
- **Context** of an invocation is the value of `this` within function body.
- **Scope** of a function is a set of variables, objects, functions accessible within a function body.

Table of contents:

1. [The mystery of `this`](#)
2. [Function invocation](#)
 - 2.1. [this in function invocation](#)
 - 2.2. [this in function invocation, strict mode](#)
 - 2.3. [Pitfall: this in an inner function](#)
3. [Method invocation](#)
 - 3.1. [this in method invocation](#)
 - 3.2. [Pitfall: separating method from its object](#)
4. [Constructor invocation](#)
 - 4.1. [this in constructor invocation](#)
 - 4.2. [Pitfall: forgetting about new](#)
5. [Indirect invocation](#)
 - 5.1. [this in indirect invocation](#)
6. [Bound function](#)
 - 6.1. [this in bound function](#)
7. [Arrow function](#)
 - 7.1. [this in arrow function](#)
 - 7.2. [Pitfall: defining method with arrow function](#)
8. [Conclusion](#)

2. Function invocation

Function invocation is performed when an expression that evaluates to a function object is followed by an open

parenthesis (, a comma separated list of arguments expressions and a close parenthesis) . For example `parseInt('18')` .

The expression cannot be a [property accessor](#) `myObject.myFunction` , which creates a method invocation. For example `[1,5].join(',')` is **not** a function invocation, but a method call.

A simple example of function invocation:

[Try in JS Bin](#)

CODE

```
function hello(name) {  
    return 'Hello ' + name + '!';  
}  
// Function invocation  
var message = hello('World');  
console.log(message); // => 'Hello World!'
```

`hello('World')` is the function invocation: `hello` expression evaluates to a function object, followed by a pair of parenthesis with 'Word' argument.

A more advanced example is the [IIFE](#) (immediately-invoked function expression):

[Try in JS Bin](#)

CODE

```
var message = (function(name) {  
    return 'Hello ' + name + '!';  
})('World');  
console.log(message) // => 'Hello World!'
```

IIFE is a function invocation too: first pair of parenthesis `(function(name) {...})` is an expression that evaluates to a function object, followed by a pair of parenthesis with 'World' argument: `('World')` .

2.1. this in function invocation

this is the global object in a function invocation

The global object is determined by the execution environment. It is the [window](#) object in a web browser and the [process](#) object in a NodeJS script.

In a function invocation the execution context is the global object.

Let's check the context in the following function:

[Try in JS Bin](#)

CODE

```

function sum(a, b) {
    console.log(this === window); // => true
    this.myNumber = 20; // add 'myNumber' property to global object
    return a + b;
}
// sum() is invoked as a function
// this in sum() is a global object (window)
sum(15, 16); // => 31
window.myNumber; // => 20

```

At the time `sum(15, 16)` is called, JavaScript automatically sets `this` as the global object, which in a browser is `window`.

When `this` is used outside any function scope (the top most scope: global execution context), it also refers to the global object:

[Try in JS Bin](#)

CODE

```

console.log(this === window); // => true
this.myString = 'Hello World!';
console.log(window.myString); // => 'Hello World!'

```

[Try in JS Bin](#)

CODE

```

<!-- In an html file -->
<script type="text/javascript">
    console.log(this === window); // => true
</script>

```

2.2. this in function invocation, strict mode

*this is **undefined** in a function invocation in strict mode*

The strict mode was introduced in [ECMAScript 5.1](#), which is a restricted variant of JavaScript and provides better security and stronger error checking.

To enable it, place the directive '`use strict`' at the top of a function body. This mode affects the execution context making `this` to be `undefined`. The execution context is not the global object anymore, contrary to above case [2.1](#).

An example of running a function in strict mode:

[Try in JS Bin](#)

CODE

```
function multiply(a, b) {  
    'use strict'; // enable the strict mode  
    console.log(this === undefined); // => true  
    return a * b;  
}  
// multiply() function invocation with strict mode enabled  
// this in multiply() is undefined  
multiply(2, 5); // => 10
```

When `multiply(2, 5)` is invoked as a function `this` is `undefined`.

The strict mode is active not only in the current scope, but also in the inner scopes (for all functions declared inside):

[Try in JS Bin](#)

CODE

```
function execute() {  
    'use strict'; // activate the strict mode  
    function concat(str1, str2) {  
        // the strict mode is enabled too  
        console.log(this === undefined); // => true  
        return str1 + str2;  
    }  
    // concat() is invoked as a function in strict mode  
    // this in concat() is undefined  
    concat('Hello', ' World!'); // => "Hello World!"  
}  
execute();
```

'use strict' is inserted at the top of `execute` body, which enables the strict mode within its scope. Because `concat` is declared within the `execute` scope, it inherits the strict mode. And the invocation `concat('Hello', ' World!')` makes `this` to be `undefined`.

A single JavaScript file may contain both strict and non-strict modes. So it is possible to have different context behavior in a single script for the same invocation type:

[Try in JS Bin](#)

CODE

```

function nonStrictSum(a, b) {
  // non-strict mode
  console.log(this === window); // => true
  return a + b;
}
function strictSum(a, b) {
  'use strict';
  // strict mode is enabled
  console.log(this === undefined); // => true
  return a + b;
}
// nonStrictSum() is invoked as a function in non-strict mode
// this in nonStrictSum() is the window object
nonStrictSum(5, 6); // => 11
// strictSum() is invoked as a function in strict mode
// this in strictSum() is undefined
strictSum(8, 12); // => 20

```

2.3. Pitfall: this in an inner function

A common trap with the function invocation is thinking that `this` is the same in an inner function as in the outer function.

Correctly the context of the inner function depends only on invocation, but not on the outer function's context.

To have the expected `this`, modify the inner function's context with indirect invocation (using [.call\(\)](#) or [.apply\(\)](#), see [5.](#)) or create a bound function (using [.bind\(\)](#), see [6.](#)).

The following example is calculating a sum of two numbers:

[Try in JS Bin](#)

CODE

```

var numbers = {
  numberA: 5,
  numberB: 10,
  sum: function() {
    console.log(this === numbers); // => true
    function calculate() {
      // this is window or undefined in strict mode
      console.log(this === numbers); // => false
      return this.numberA + this.numberB;
    }
    return calculate();
  }
};
numbers.sum(); // => NaN or throws TypeError in strict mode

```

`numbers.sum()` is a method invocation on an object (see [3.](#)), so the context in `sum` is `numbers` object.

`calculate` function is defined inside `sum`, so you might expect to have `this` as `numbers` object in `calculate()` too.

However `calculate()` is a function invocation (but **not** method invocation) and it has `this` as the global object `window` (case [2.1.](#)) or `undefined` in strict mode (case [2.2.](#)). Even if the outer function `sum` has the context as `numbers` object, it doesn't have influence here.

The invocation result of `numbers.sum()` is `Nan` or an error is thrown `TypeError: Cannot read property 'numberA' of undefined` in strict mode. Definitely not the expected result $5 + 10 = 15$, all because `calculate`

is not invoked correctly.

To solve the problem, `calculate` function should be executed with the same context as the `sum` method, in order to access `numberA` and `numberB` properties. One solution is to use `.call()` method (see section [5.](#)):

[Try in JS Bin](#)

```
var numbers = {
    numberA: 5,
    numberB: 10,
    sum: function() {
        console.log(this === numbers); // => true
        function calculate() {
            console.log(this === numbers); // => true
            return this.numberA + this.numberB;
        }
        // use .call() method to modify the context
        return calculate.call(this);
    }
};
numbers.sum(); // => 15
```

CODE

`calculate.call(this)` executes `calculate` function as usual, but additionally modifies the context to a value specified as the first parameter. Now `this.numberA + this.numberB` is equivalent to `numbers.numberA + numbers.numberB` and the function returns the expected result $5 + 10 = 15$.

3. Method invocation

A **method** is a function stored in a property of an object. For example:

[Try in JS Bin](#)

```
var myObject = {
    // helloFunction is a method
    helloFunction: function() {
        return 'Hello World!';
    }
};
var message = myObject.helloFunction();
```

CODE

`helloFunction` is a method in `myObject`. To get the method, use a property accessor:
`myObject.helloFunction`.

Method invocation is performed when an expression in a form of [property accessor](#) that evaluates to a function object is followed by an open parenthesis `(`, a comma separated list of arguments expressions and a close parenthesis `)`.

Using the previous example, `myObject.helloFunction()` is a method invocation of `helloFunction` on the object `myObject`. Also method calls are: `[1, 2].join(',')` or `/\s/.test('beautiful world')`.

It is important to distinguish **function invocation** (see section [2.](#)) from **method invocation**, because they are different types. The main difference is that method invocation requires a property accessor form to call the function (`<expression>.functionProperty()` or `<expression>['functionProperty']()`), while function invocation

does not (`<expression>()`).

[Try in JS Bin](#)

CODE

```
'Hello', 'World'].join(' ', ); // method invocation
({ ten: function() { return 10; } }).ten(); // method invocation
var obj = {};
obj.myFunction = function() {
  return new Date().toString();
};
obj.myFunction(); // method invocation

var otherFunction = obj.myFunction;
otherFunction(); // function invocation
parseFloat('16.60'); // function invocation
isNaN(0); // function invocation
```

3.1. `this` in method invocation

this is the object that owns the method in a method invocation

When invoking a method on an object, `this` becomes the object itself.

Let's create an object with a method that increments a number:

[Try in JS Bin](#)

CODE

```
var calc = {
  num: 0,
  increment: function() {
    console.log(this === calc); // => true
    this.num += 1;
    return this.num;
  }
};
// method invocation. this is calc
calc.increment(); // => 1
calc.increment(); // => 2
```

Calling `calc.increment()` will make the context of the `increment` function to be `calc` object. So using `this.num` to increment the number property is working well.

A JavaScript object inherits a method from its `prototype`. When the inherited method is invoked on the object, the context of the invocation is still the object itself:

[Try in JS Bin](#)

CODE

```

var myDog = Object.create({
  sayName: function() {
    console.log(this === myDog); // => true
    return this.name;
  }
});
myDog.name = 'Milo';
// method invocation. this is myDog
myDog.sayName(); // => 'Milo'

```

[Object.create\(\)](#) creates a new object `myDog` and sets the prototype. `myDog` object inherits `sayName` method. When `myDog.sayName()` is executed, `myDog` is the context of invocation.

In ECMAScript 6 [class](#) syntax, the method invocation context is also the instance itself:

[Try in JS Bin](#)

CODE

```

class Planet {
  constructor(name) {
    this.name = name;
  }
  getName() {
    console.log(this === earth); // => true
    return this.name;
  }
}
var earth = new Planet('Earth');
// method invocation. the context is earth
earth.getName(); // => 'Earth'

```

3.2. Pitfall: separating method from its object

A method from an object can be extracted into a separated variable. When calling the method using this variable, you might think that `this` is the object on which the method was defined.

Correctly if the method is called without an object, then a function invocation happens: where `this` is the global object `window` or `undefined` in strict mode (see [2.1](#) and [2.2](#)).

Creating a bound function (using [.bind\(\)](#), see [6.](#)) fixes the context, making it the object that owns the method.

The following example creates `Animal` constructor and makes an instance of it - `myCat`. Then `setTimeout()` after 1 second logs `myCat` object information:

[Try in JS Bin](#)

CODE

```

function Animal(name, legs) {
  this.type = type;
  this.legs = legs;
  this.logInfo = function() {
    console.log(this === myCat); // => false
    console.log('The ' + this.type + ' has ' + this.legs + ' legs');
  }
}
var myCat = new Animal('Cat', 4);
// logs "The undefined has undefined legs"
// or throws a TypeError in strict mode
setTimeout(myCat.logInfo);

```

You might think that `setTimeout` will call the `myCat.logInfo()`, which will log the information about `myCat` object. But the method is separated from its object when passed as a parameter: `setTimeout(myCat.logInfo)`, and after 1 second a function invocation happens. When `logInfo` is invoked as a function, `this` is global object or `undefined` in strict mode (but **not** `myCat` object), so the object information does not log correctly.

A function can be bound with an object using [.bind\(\)](#) method (see [6.](#)). If the separated method is bound with `myCat` object, the context problem is solved:

[Try in JS Bin](#)

CODE

```

function Animal(type, legs) {
  this.type = type;
  this.legs = legs;
  this.logInfo = function() {
    console.log(this === myCat); // => true
    console.log('The ' + this.type + ' has ' + this.legs + ' legs');
  };
}
var myCat = new Animal('Cat', 4);
// logs "The Cat has 4 legs"
setTimeout(myCat.logInfo.bind(myCat));

```

`myCat.logInfo.bind(myCat)` returns a new function that executes exactly like `logInfo`, but has `this` as `myCat` even in a function invocation.

4. Constructor invocation

Constructor invocation is performed when `new` keyword is followed by an expression that evaluates to a function object, an open parenthesis `(`, a comma separated list of arguments expressions and a close parenthesis `)`. For example: `new RegExp('\\d')`.

This example declares a function `Country`, then invokes it as a constructor:

[Try in JS Bin](#)

CODE

```

function Country(name, traveled) {
    this.name = this.name ? this.name : 'United Kingdom';
    this.traveled = Boolean(traveled); // transform to a boolean
}
Country.prototype.travel = function() {
    this.traveled = true;
};
// Constructor invocation
var france = new Country('France', false);
// Constructor invocation
var unitedKingdom = new Country;

france.travel(); // Travel to France

```

`new Country('France', false)` is a constructor invocation of the `Country` function. The result of execution is a new object, which `name` property is '`France`'.

If the constructor is called without arguments, then the parenthesis pair can be omitted: `new Country`.

Starting ECMAScript 6, JavaScript allows to define constructors using [class](#) keyword:

[Try in JS Bin](#)

CODE

```

class City {
    constructor(name, traveled) {
        this.name = name;
        this.traveled = false;
    }
    travel() {
        this.traveled = true;
    }
}
// Constructor invocation
var paris = new City('Paris', false);
paris.travel();

```

`new City('Paris')` is a constructor invocation. The object initialization is handled by a special method in the class: [constructor](#), which has `this` as the newly created object.

A constructor call creates an empty new object, which inherits properties from constructor's prototype. The role of constructor function is to initialize the object.

As you might know already, the context in this type of call is the created instance. This is a next chapter subject.

When a property accessor `myObject.myFunction` is preceded by `new` keyword, JavaScript will execute a **constructor invocation**, but **not a method invocation**.

For example `new myObject.myFunction()`: first the function is extracted using a property accessor `extractedFunction = myObject.myFunction`, then invoked as a constructor to create a new object: `new extractedFunction()`.

4.1. this in constructor invocation

this is the newly created object in a constructor invocation

The context of a constructor invocation is the newly created object. It is used to initialize the object with data that comes from constructor function arguments, setup initial value for properties, attach event handlers, etc.

Let's check the context in the following example:

[Try in JS Bin](#)

```
function Foo () {
  console.log(this instanceof Foo); // => true
  this.property = 'Default Value';
}
// Constructor invocation
var fooInstance = new Foo();
fooInstance.property; // => 'Default Value'
```

CODE

`new Foo()` is making a constructor call where the context is `fooInstance`. Inside `Foo` the object is initialized: `this.property` is assigned with a default value.

The same scenario happens when using [`class`](#) syntax (available in ES6), only the initialization happens in the `constructor` method:

[Try in JS Bin](#)

```
class Bar {
  constructor() {
    console.log(this instanceof Bar); // => true
    this.property = 'Default Value';
  }
}
// Constructor invocation
var barInstance = new Bar();
barInstance.property; // => 'Default Value'
```

CODE

At the time when `new Bar()` is executed, JavaScript creates an empty object and makes it the context of the `constructor` method. Now you can add properties to object using `this` keyword: `this.property = 'Default Value'`.

4.2. Pitfall: forgetting about `new`

Some JavaScript functions create instances not only when invoked as constructors, but also when invoked as functions. For example `RegExp`:

[Try in JS Bin](#)

```
var reg1 = new RegExp('\\w+');
var reg2 = RegExp('\\w+');

reg1 instanceof RegExp;      // => true
reg2 instanceof RegExp;      // => true
reg1.source === reg2.source; // => true
```

CODE

When executing `new RegExp('\\w+')` and `RegExp('\\w+')` JavaScript creates equivalent regular expression

objects.

Using a function invocation to create objects is a potential problem (excluding [factory pattern](#)), because some constructors may omit the logic to initialize the object when `new` keyword is missing.

The following example illustrates the problem:

[Try in JS Bin](#)

CODE

```
function Vehicle(type, wheelsCount) {
    this.type = type;
    this.wheelsCount = wheelsCount;
    return this;
}
// Function invocation
var car = Vehicle('Car', 4);
car.type;      // => 'Car'
car.wheelsCount // => 4
car === window // => true
```

`Vehicle` is a function that sets `type` and `wheelsCount` properties on the context object.

When executing `Vehicle('Car', 4)` an object `car` is returned, which has the correct properties: `car.type` is `'Car'` and `car.wheelsCount` is `4`. You might think it works well for creating and initializing new objects.

However this is `window` object in a function invocation (see [2.1](#)) and `Vehicle('Car', 4)` is setting properties on the `window` object - faulty scenario. A new object is not created.

Make sure to use `new` operator in cases when a constructor call is expected:

[Try in JS Bin](#)

CODE

```
function Vehicle(type, wheelsCount) {
    if (!(this instanceof Vehicle)) {
        throw Error('Error: Incorrect invocation');
    }
    this.type = type;
    this.wheelsCount = wheelsCount;
    return this;
}
// Constructor invocation
var car = new Vehicle('Car', 4);
car.type          // => 'Car'
car.wheelsCount   // => 4
car instanceof Vehicle // => true

// Function invocation. Generates an error.
var brokenCar = Vehicle('Broken Car', 3);
```

`new Vehicle('Car', 4)` works well: a new object is created and initialized, because `new` keyword is present in the constructor invocation.

A verification is added in the constructor function: `this instanceof Vehicle`, to make sure that execution context is a correct object type. If `this` is not a `Vehicle`, then an error is generated. This way if `Vehicle('Broken Car', 3)` is executed (without `new`) an exception is thrown: `Error: Incorrect invocation`.

5. Indirect invocation

Indirect invocation is performed when a function is called using `.call()` or `.apply()` methods.

Functions in JavaScript are first-class objects, which means that a function is an object. The type of this object is [Function](#).

From the [list of methods](#) that a function object has, `.call()` and `.apply()` are used to invoke the function with a configurable context.

The method `.call(thisArg[, arg1[, arg2[, ...]]])` accepts the first argument `thisArg` as the context of the invocation and a list of arguments `arg1, arg2, ...` that are passed as arguments to the called function.

And the method `.apply(thisArg, [args])` accepts the first argument `thisArg` as the context of the invocation and an [array-like object](#) of values `[args]` that are passed as arguments to the called function.

The following example shows the indirect invocation:

[Try in JS Bin](#)

CODE

```
function increment(number) {
  return ++number;
}
increment.call(undefined, 10);    // => 11
increment.apply(undefined, [10]); // => 11
```

`increment.call()` and `increment.apply()` both invoke the `increment` function with `10` argument.

The main difference between the two is that `.call()` accepts a list of arguments, for example `myFunction.call(thisValue, 'value1', 'value2')`. But `.apply()` accepts a list of values in an array-like object, e.g. `myFunction.apply(thisValue, ['value1', 'value2'])`.

5.1. this in indirect invocation

*this is the **first argument** of `.call()` or `.apply()` in an indirect invocation*

CODE

It's obvious that `this` in indirect invocation is the value passed as first argument to `.call()` or `.apply()`. The following example shows that:

[Try in JS Bin](#)

```
var rabbit = { name: 'White Rabbit' };
function concatName(string) {
  console.log(this === rabbit); // => true
  return string + this.name;
}
// Indirect invocations
concatName.call(rabbit, 'Hello '); // => 'Hello White Rabbit'
concatName.apply(rabbit, ['Bye ']); // => 'Bye White Rabbit'
```

The indirect invocation is useful when a function should be executed with a specific context. For example to solve the

context problems with function invocation, where `this` is always `window` or `undefined` in strict mode (see [2.3.](#)). It can be used to simulate a method call on an object (see the previous code sample).

Another practical example is creating hierarchies of classes in ES5 to call the parent constructor:

[Try in JS Bin](#)

CODE

```
function Runner(name) {
  console.log(this instanceof Rabbit); // => true
  this.name = name;
}
function Rabbit(name, countLegs) {
  console.log(this instanceof Rabbit); // => true
  // Indirect invocation. Call parent constructor.
  Runner.call(this, name);
  this.countLegs = countLegs;
}
var myRabbit = new Rabbit('White Rabbit', 4);
myRabbit; // { name: 'White Rabbit', countLegs: 4 }
```

`Runner.call(this, name)` inside `Rabbit` makes an indirect call of the parent function to initialize the object.

6. Bound function

A **bound function** is a function bind with an object. Usually it is created from the original function using [.bind\(\)](#) method. The original and bound functions share the same code and scope, but different contexts on execution.

The method `.bind(thisArg[, arg1[, arg2[, ...]]])` accepts the first argument `thisArg` as the context of the bound function on invocation and an optional list of arguments `arg1, arg2, ...` that are passed as arguments to the called function. It returns a new function bound with `thisArg`.

The following code creates a bound function and later invokes it:

[Try in JS Bin](#)

CODE

```
function multiply(number) {
  'use strict';
  return this * number;
}
// create a bound function with context
var double = multiply.bind(2);
// invoke the bound function
double(3); // => 6
double(10); // => 20
```

`multiply.bind(2)` returns a new function object `double`, which is bound with number `2`. `multiply` and `double` have the same code and scope.

Contrary to `.apply()` and `.call()` methods (see [5.](#)), which invokes the function right away, the `.bound()` method returns a new function that it supposed to be invoked later with a pre-configured `this`.

6.1. `this` in bound function

*this is the **first argument** of .bind() when invoking a bound function*

The role of `.bind()` is to create a new function, which invocation will have the context as the first argument passed to `.bind()`. It is a powerful technique that allows to create functions with a predefined `this` value.

Let's see how to configure `this` of a bound function:

[Try in JS Bin](#)

CODE

```
var numbers = {
  array: [3, 5, 10],
  getNumbers: function() {
    return this.array;
  }
};
// Create a bound function
var boundGetNumbers = numbers.getNumbers.bind(numbers);
boundGetNumbers(); // => [3, 5, 10]
// Extract method from object
var simpleGetNumbers = numbers.getNumbers;
simpleGetNumbers(); // => undefined or throws an error in strict mode
```

`numbers.countNumbers.bind(numbers)` returns a function `boundGetNumbers` that is bound with `numbers` object.

Then `boundGetNumbers()` is invoked with `this` as `numbers` and returns the correct array object.

The function `numbers.getNumbers` can be extracted into a variable `simpleGetNumbers` without binding. On later function invocation `simpleGetNumbers()` has `this` as `window` or `undefined` in strict mode, but not `numbers` object (see [3.2. Pitfall](#)). In this case `simpleGetNumbers()` will not return correctly the array.

`.bind()` makes a permanent context link and will always keep it. A bound function cannot change its linked context when using `.call()` or `.apply()` with a different context, or even a rebound doesn't have any effect.

Only the constructor invocation of a bound function can change that, however this is not a recommended approach (for constructor invocation use *normal*, not bound functions).

The following example declares a bound function, then tries to change its pre-defined context:

[Try in JS Bin](#)

CODE

```
function getThis() {
  'use strict';
  return this;
}
var one = getThis.bind(1);
// Bound function invocation
one(); // => 1
// Use bound function with .apply() and .call()
one.call(2); // => 1
one.apply(2); // => 1
// Bind again
one.bind(2)(); // => 1
// Call the bound function as a constructor
new one(); // => Object
```

Only `new one()` changes the context of the bound function, other types of invocation always have `this` equal to

1 .

7. Arrow function

Arrow function is designed to declare the function in a shorter form and [lexically](#) bind the context.

It can be used the following way:

[Try in JS Bin](#)

```
CODE
var hello = (name) => {
  return 'Hello ' + name;
};
hello('World'); // => 'Hello World'
// Keep only even numbers
[1, 2, 5, 6].filter(item => item % 2 === 0); // => [2, 6]
```

Arrow functions bring a lighter syntax, excluding the verbose keyword `function`. You could even omit the `return`, when the function has only 1 statement.

An arrow function is [anonymous](#), which means that `name` property is an empty string '' . This way it doesn't have a lexical function name (which would be useful for recursion, detaching event handlers).

Also it doesn't provide the [arguments](#) object, opposed to a regular function. However this is fixed using ES6 [rest parameters](#):

[Try in JS Bin](#)

```
CODE
var sumArguments = (...args) => {
  console.log(typeof arguments); // => 'undefined'
  return args.reduce((result, item) => result + item);
};
sumArguments.name      // => ''
sumArguments(5, 5, 6); // => 16
```

7.1. `this` in arrow function

this is the enclosing context where the arrow function is defined

The arrow function doesn't create its own execution context, but takes `this` from the outer function where it is defined.

The following example shows this context transparency property:

[Try in JS Bin](#)

CODE

```

class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
  log() {
    console.log(this === myPoint); // => true
    setTimeout(()=> {
      console.log(this === myPoint);      // => true
      console.log(this.x + ':' + this.y); // => '95:165'
    }, 1000);
  }
}
var myPoint = new Point(95, 165);
myPoint.log();

```

`setTimeout` is calling the arrow function with the same context (`myPoint` object) as the `log()` method. As seen, the arrow function "inherits" the context from the function where it is defined.

If trying to use a regular function in this example, it would create its own context (`window` or `undefined` in strict mode). So to make the same code work correctly with a function expression it's necessary to manually bind the context: `setTimeout(function() {...}.bind(this))`. This is verbose, and using an arrow function is a cleaner and shorter solution.

If the arrow function is defined in the top most scope (outside any function), the context is always the global object (`window` in a browser and `process` object in NodeJS):

[Try in JS Bin](#)

CODE

```

var getContext = () => {
  console.log(this === window); // => true
  return this;
};
console.log(getContext() === window); // => true

```

An arrow function is bound with the lexical context **once and forever**. `this` cannot be modified even if using the context modification methods:

[Try in JS Bin](#)

CODE

```

var numbers = [1, 2];
(function() {
  var get = () => {
    console.log(this === numbers); // => true
    return this;
  };
  console.log(this === numbers); // => true
  get(); // => [1, 2]
  // Use arrow function with .apply() and .call()
  get.call([0]); // => [1, 2]
  get.apply([0]); // => [1, 2]
  // Bind
  get.bind([0])(); // => [1, 2]
}).call(numbers);

```

A function expression is called indirectly using `.call(numbers)`, which makes `this` of the invocation as `numbers`. The `get` arrow function has `this` as `numbers` too, because it takes the context lexically.

No matter how `get` is called, it always keeps the initial context `numbers`. Indirect call with other context (using `.call()` or `.apply()`), rebinding (using `.bind()`) have no effect.

Arrow function cannot be used as a constructor. If invoking it as a constructor `new get()`, JavaScript throws an error: `TypeError: get is not a constructor`.

7.2. Pitfall: defining method with arrow function

You might want to use arrow functions to declare methods on an object. Fair enough: their declaration is quite short comparing to a [function expression](#): `(param) => {...}` instead of `function(param) {...}`.

This example defines a method `format()` on a class `Period` using an arrow function:

[Try in JS Bin](#)

CODE

```

function Period (hours, minutes) {
  this.hours = hours;
  this.minutes = minutes;
}
Period.prototype.format = () => {
  console.log(this === window); // => true
  return this.hours + ' hours and ' + this.minutes + ' minutes';
};
var walkPeriod = new Period(2, 30);
walkPeriod.format(); // => 'undefined hours and undefined minutes'

```

Since `format` is an arrow function and is defined in the global context (top most scope), it has `this` as `window` object.

Even if `format` is executed as a method on an object `walkPeriod.format()`, `window` is kept as the context of invocation. It happens because arrow function have a static context that doesn't change on different invocation types. `this` is `window`, so `this.hours` and `this.minutes` are `undefined`. The method returns the string: `'undefined hours and undefined minutes'`, which is not the expected result.

The function expression solves the problem, because a regular function does change its context depending on invocation:

[Try in JS Bin](#)

CODE

```

function Period (hours, minutes) {
  this.hours = hours;
  this.minutes = minutes;
}
Period.prototype.format = function() {
  console.log(this === walkPeriod); // => true
  return this.hours + ' hours and ' + this.minutes + ' minutes';
};
var walkPeriod = new Period(2, 30);
walkPeriod.format(); // => '2 hours and 30 minutes'

```

`walkPeriod.format()` is a method invocation on an object (see [3.1.](#)) with the context `walkPeriod` object. `this.hours` evaluates to 2 and `this.minutes` to 30 , so the method returns the correct result: '2 hours and 30 minutes' .

8. Conclusion

Because the function invocation has the biggest impact on `this` , from now on **do not** ask yourself:

Where is `this` taken from?

but **do** ask yourself:

How is the function invoked ?

For an arrow function ask yourself:

What is `this` where the arrow function is defined ?

This mindset is correct when dealing with `this` and will save you from headache.

If you have an interesting example of context pitfall or just experience difficulties with a case, write a comment bellow and let's discuss!

Spread the knowledge about JavaScript and [share](#) the post, your colleagues will appreciate it.

Don't lose your context ;)

See also the recent popular posts:

[JavaScript variables hoisting in details](#)

[The legend of JavaScript equality operator](#)

[JavaScript addition operator in details](#)

65. Javascript Factory Patterns

Javascript Factory Patterns

Former Hack Reactor Student Ryan Atkinson wrote a great piece on his blog about [Javascipt Instantiation Patterns](#). The highlight of it was the graphic that neatly breaks down the types. What it failed thoroughly explain is why Pseudoclassical is best and how to ensure subclasses maintain all of their features.

Functional Pattern

```
var Plane = function(color, name){
  var obj = {};
  obj.name = name;
  obj.color = color;
  obj.speed = 50;
  obj.barrelRoll = function(){
    console.log("you have barrel rolled");
  };
  obj.ludaSpeed = function(){
    obj.speed = 400;
  };
  return obj;
};
var plane = Plane('black', 'roxy');
```

Functional represents a basic level of understanding, but it does not allow us to take real advantage of constructor functions. You will see everything is recreated with each new object. We are having to reuse code and are going to take a hit to performance

Functional shared

```
var Plane = function(color, name){
  var obj = {};
  obj.name = name;
  obj.color = color;
  obj.speed = 50;
  obj.barrelRoll = planeMethods.barrelRoll;
  obj.ludaSpeed = planeMethods.ludaSpeed;
  return obj;
};
var planeMethods = {};
planeMethods.barrelRoll = function(){
  console.log(this.name +"did a barrel roll");
};
planeMethods.ludaSpeed = function(){
  this.speed = 400;
};
var plane = Plane('black', 'Star Fox');
```

Functional shared is slightly better, but still has many of the same issues that functional does. We have to generate a lot of code just to get our objects built. Despite moving the functions outside of the constructor function, we are still having to formally reference them.

Prototypal

```
var Plane = function(color,name){
  var obj = Object.create(Plane.prototype);
  obj.name = name;
  obj.color = color;
  obj.speed = 50;

  return obj;
};
Plane.prototype.barrelRoll = function(){
  console.log(this.name +"did a barrel roll");
};
Plane.prototype.ludaSpeed = function(){
  this.speed = 400;
};
var plane = Plane('black','Star Fox');
```

Prototypal serves almost no purpose. If you have made it this far you mind as well, make the jump to Pseudoclassical. You will notice in the example there are some benefits. We can use a prototype; thus, avoiding still having to reference the functions that exist outside of the constructor functions. Unfortunately, we are still reusing code that we should not be.

Pseudoclassical

```
var Plane = function(color,name){
  this.name = name;
  this.color = color;
  this.speed = 50;
};
Plane.prototype.barrelRoll = function(){
  console.log(this.name +"did a barrel roll");
};
Plane.prototype.ludaSpeed = function(){
  this.speed = 400;
};
var plane = new Plane('black','Star Fox');
```

There is a reason Pseudoclassical has taken prominence within the Javascript community. Its simple and its fast. It allows us to take advantage of prototypal inheritance meaning we can easily reuse code. One thing Ryan left off though is an example of how this becomes powerful.

Example

```

var PeppyPlane = function(color,name){
  Plane.call(this,color,name);
  this.speed = 60;
};

PeppyPlane.prototype = Object.create(Plane.prototype);
PeppyPlane.prototype.constructor = PeppyPlane; //Ensures PeppyPlane has the right constructor.
PeppyPlane.prototype.barrelRoll = function(name){
  console.log(name +"do a barrel roll");
};

var starFox = new Plane('black','Star Fox');
var peppy = new PeppyPlane('green','Peppy');
peppy.barrelRoll('Star Fox');
//Would log - "Star Fox do a barrel roll"
starFox.barrelRoll();
//Would log - "Star Fox did a barrelRoll"

```

Our Plane constructor starts with a function called do a barrelRoll. We could pass this down to the sub constructor of Plane called PeppyPlane. Within PeppyPlane we use Plane.call to inherit all of the Plane's properties. On PeppyPlane, we change the barrelRoll function for new PeppyPlanes, but all other Planes would retain the original barrelRoll function. This is powerful because different sub-constructors can inherit functions and properties while easily personalizing them if needed.

```

var Plane = function(color,name){
  var obj = {};
  obj.name = name;
  obj.color = color;
  obj.speed = 50;
  obj.barrelRoll = function(){
    console.log("you have barrel rolled");
  };
  obj.ludaSpeed = function(){
    obj.speed = 400;
  };
  return obj;
};
var plane = Plane('black','roxy');

```

```

var Plane = function(color,name){
  var obj = {};
  obj.name = name;
  obj.color = color;
  obj.speed = 50;
  obj.barrelRoll = planeMethods.barrelRoll;
  obj.ludaSpeed = planeMethods.ludaSpeed;
  return obj;
};

var planeMethods = {};
planeMethods.barrelRoll = function(){
  console.log(this.name +"did a barrel roll");
};

planeMethods.ludaSpeed = function(){
  this.speed = 400;
};

var plane = Plane('black','Star Fox');

var Plane = function(color,name){
  var obj = Object.create(Plane.prototype);
  obj.name = name;
  obj.color = color;
  obj.speed = 50;

  return obj;
};

Plane.prototype.barrelRoll = function(){
  console.log(this.name +"did a barrel roll");
};

Plane.prototype.ludaSpeed = function(){
  this.speed = 400;
};

var plane = Plane('black','Star Fox');

```

Functional, Functional-Shared, Prototypal

```

var Plane = function(color,name){
  this.name = name;
  this.color = color;
  this.speed = 50;
};

Plane.prototype.barrelRoll = function(){
  console.log(this.name +"did a barrel roll");
};

Plane.prototype.ludaSpeed = function(){
  this.speed = 400;
};

var plane = new Plane('black','Star Fox');

```

```

var PeppyPlane = function(color,name){
  Plane.call(this,color,name);
  this.speed = 60;
};

PeppyPlane.prototype = Object.create(Plane.prototype);
PeppyPlane.prototype.constructor = PeppyPlane; //Ensures PeppyPlane has the right constructor.
PeppyPlane.prototype.barrelRoll = function(name){
  console.log(name +"do a barrel roll");
};

var starFox = new Plane('black','Star Fox');
var peppy = new PeppyPlane('green','Peppy');
peppy.barrelRoll('Star Fox');
//Would log - "Star Fox do a barrel roll"
starFox.barrelRoll();
//Would log - "Star Fox did a barrelRoll"

```

Pseudoclassical, Example

Final Note

As programmers we are always tasked with writing less code that has more power. Functional may be a great way to dip your toes into Object Oriented Programming, but you only realize its true power with Pseudoclassical style.

66. JavaScript Classes and Instantiation Patterns

JavaScript's idiosyncrasies can be a bit frustrating, and functional classes are no exception. The language implements functional classes (aka constructor functions) as a critical technique for *generating multiple objects with similar properties and methods*. The four styles utilized to implement functional classes are **functional, functional-shared, prototypal, and pseudoclassical**. Mastering all four can prove to be an unnecessarily complicated endeavor, as each approach levies a unique interface with differing approaches to object construction. The fantastic news is that **under the hood all four constructor function styles are doing basically the exact same thing**. The quick reference below ([Download @2x](#)) demonstrates these strong similarities to hopefully eliminate confusion between styles.

	functional	functional - shared	prototypal	pseudoclassical
generate object	<pre>var House = function(color){ var obj = {}; obj.color = color; obj.door = 'open'; };</pre>	<pre>var House = function(color){ var obj = {}; obj.color = color; obj.door = 'open'; };</pre>	<pre>var House = function(color){ var obj = Object.create(House.prototype); obj.color = color; obj.door = 'open'; };</pre>	<pre>var House = function(color){ // (Automatically generated by interpreter) // var this = Object.create(House.prototype); this.color = color; this.door = 'open'; };</pre>
assign properties				
explicitly define or borrow methods	<pre>obj.openDoor = function(){ obj.door = 'open'; }; obj.closeDoor = function(){ obj.door = 'closed'; };</pre>	<pre>obj.open = houseMethods.open; obj.close = houseMethods.close;</pre>		
return object	<pre>return obj;</pre>	<pre>return obj;</pre>	<pre>return obj;</pre>	<pre>// (Automatically generated by interpreter) // return this;</pre>
Add methods to delegated fallback object /*not including functional-shared*/	<pre>}; var houseMethods = {}; houseMethods.openDoor = function(){ this.door = 'open'; }; houseMethods.closeDoor = function(){ this.door = 'closed'; };</pre>	<pre>}; // (Automatically generated by interpreter) // House.prototype = {}; House.prototype.openDoor = function(){ this.door = 'open'; }; House.prototype.closeDoor = function(){ this.door = 'closed'; };</pre>	<pre>}; // (Automatically generated by interpreter) // House.prototype = {}; House.prototype.openDoor = function(){ this.door = 'open'; }; House.prototype.closeDoor = function(){ this.door = 'closed'; };</pre>	<pre>}; // (Automatically generated by interpreter) // House.prototype = {}; House.prototype.openDoor = function(){ this.door = 'open'; }; House.prototype.closeDoor = function(){ this.door = 'closed'; };</pre>
Instantiation pattern	<pre>var house = House('red');</pre>	<pre>var house = House('red');</pre>	<pre>var house = House('red');</pre>	<pre>var house = new House('red');</pre>
Class pros	Super clear object construction	Methods shared between objects	All instances delegate fallback to House.prototype methods	Super concise code, clear 'this' binding, and delegates to House.prototype methods
Class cons	Results in duplicate methods	Setting method pointers is less efficient than delegating a fallback	Utilizes unnecessary lines of code compared to pseudo	Interpreter magic makes object construction unclear

In the example above, every house on the street needs to have a color, a door, and methods to open and close the door. **Functional classes are perfect here because every instance created by the function will have these basic desired attributes, so we can quickly create thousands of homes.** Each functional class style performs the same basic steps to help us generate all these new homes:

- generate an object
- assign some properties
- add some methods
- return that object

It's so simple! So, why even confuse the matter with four different styles? Each one has its pros and cons. It all boils down to readability and techniques to prevent unnecessary method duplication.

The clarity of the **functional** style is unmatched, with object generation, property and method assignment, and object return all right before your eyes. This is a great place to start when first diving into functional class styles. However, given the tremendous variation in utilization across the industry, it's wise to master each and understand why modern frameworks have gravitated toward pseudoclassical style. The biggest setback of the functional style is that our methods will be duplicated for every object created. That means a new function, stored in a new spot in memory, every time we generate an object; not perfect.

Functional-shared helps alleviate those concerns by utilizing a single repository for our methods, and then generating pointers each time we create an object. In the example above, every new object will point back to the houseMethods object where these shared functions are stored a single time in memory. Awesome, now we are saving space! Still, setting these method pointers can be less efficient than delegating through fallback.

So what are fallbacks? Fallbacks are a back-up plan for objects, essential to the prototypal style. If a method (or property) is called on an object, and that method doesn't exist, JavaScript will check if it's defined on this *fallback* object. Fortunately, every function has a property called "prototype" where these fallback methods are typically stored. In the **prototypal's** House function, we explicitly delegate the House function's "prototype" property (**House.prototype**) as the fallback location for EVERY house object created by House. Now every method in

`House.prototype` is available to every object created by `House`, without duplicating methods in memory! You can set anything as the fallback too, like our old `houseMethods` object, and this is specified via the input to `Object.create()`. The `prototype` property on the constructor function just happens to be the most organized location to delegate fallback.

Lastly, **pseudoclassical** attempts to wrap all this goodness in the most concise presentation possible. Rather than assign `Object.create(House.prototype)` to a new variable, the pseudoclassical style simply assigns it to "this" for simple property assignment and method creation. And to top it off, **the interpreter will do this automatically behind the scenes, along with returning the object, as long as the keyword `new` is utilized at instantiation time** `var house = new House('red')`. The remaining code is clean and concise, but can be tricky to reason with before developing a complete understanding of this [syntactic sugar](#).

So how does instantiation fit into this class debacle? Instantiation takes place when a functional class is utilized to create a new object, `var house = House('red')` in the examples above. Pseudoclassical is the only class that takes advantage of the `new` keyword (detailed above), while the rest of the classes are used like any other function call.

It absolutely makes sense to run with the style that works best for you. Still, if you come across any pattern and are still confused, check out the quick reference for a refresh on the subtle differences. Best wishes on your utilization of the four class styles. A quick dive into subclassing is soon to follow!

67. Inheritance in JavaScript: understanding the constructor property

JavaScript has an interesting inheritance mechanism: [prototypal](#). Most of the starting JavaScript developers have hard time understanding it, as well as I had.

All types in JavaScript (except the `null` and `undefined` values) have a `constructor` property, which is a part of the inheritance. For example:

```
CODE
var num = 150;
num.constructor === Number // => true
var obj = {};
str.constructor === Object // => true
var reg = /\d/g;
reg.constructor === RegExp; // => true
```

In this article we'll dive into the `constructor` property of an object. It serves as a public identity of the class, which means it can be used for:

- Identify to what class belongs an object (an alternative to `instanceof`)
- Reference from an object or a prototype the constructor function
- Get the class name

1. The constructor in primitive types

In JavaScript the [primitive types](#) are `number`, `boolean`, `string`, `symbol` (in ES6), `null` and `undefined`.

Any value except `null` and `undefined` has a `constructor` property, which refers to the corresponding type function:

- `Number()` for numbers: `(1).constructor === Number`
- `Boolean()` for booleans: `(true).constructor === Boolean`
- `String()` for strings: `('hello').constructor === String`
- `Symbol()` for symbols: `Symbol().constructor === Symbol`

The constructor property of a primitive can be used to determine its type by comparing it with the corresponding function. For example to verify if the value is a number:

```
if (myVariable.constructor === Number) {
    // code executed when myVariable is a number
    myVariable += 1;
}
```

CODE

Notice that this approach is generally not recommended and `typeof` is preferable (see 1.1). But it can be useful for a `switch` statement, to reduce the number of `if/else`:

```
// myVariable = ...
var type;
switch (myVariable.constructor) {
    case Number:
        type = 'number';
        break;
    case Boolean:
        type = 'boolean';
        break;
    case String:
        type = 'string';
        break;
    case Symbol:
        type = 'symbol';
        break;
    default:
        type = 'unknown';
        break;
}
```

CODE

1.1 The object wrapper for a primitive value

An object wrapper for a primitive is created when invoking the function with `new` operator. Wrappers can be created for `new String('str')`, `new Number(15)` and `new Boolean(false)`. It cannot be created for a `Symbol`, because invoked this way `new Symbol('symbol')` generates a `TypeError`.

The wrapper exists to allow developer to attach custom properties and methods to a primitive, because JavaScript doesn't allow for primitives to have own properties.

Existence of these objects may create confusion for determining the variable type based on the constructor, because the wrapper has the same constructor as the primitive:

CODE

```
var booleanObject = new Boolean(false);
booleanObject.constructor === Boolean // => true

var booleanPrimitive = false;
booleanPrimitive.constructor === Boolean // => true
```

2. The constructor in a prototype object

The `constructor` property in a prototype is automatically setup to reference the constructor function.

CODE

```
function Cat(name) {
  this.name = name;
}
Cat.prototype.getName = function() {
  return this.name;
}
Cat.prototype.clone = function() {
  return new this.constructor(this.name);
}
Cat.prototype.constructor === Cat // => true
```

Because properties are inherited from the prototype, the `constructor` is available on the instance object too.

CODE

```
var catInstance = new Cat('Mew');
catInstance.constructor === Cat // => true
```

Even if the object is created from a literal, it inherits the constructor from `Object.prototype`.

CODE

```
var simpleObject = {
  weekDay: 'Sunday'
};
simpleObject.prototype === Object // => true
```

2.1 Don't loose the constructor in the subclass

`constructor` is a regular non-enumerable property in the prototype object. It does not update automatically when a new object is created based on it. When creating a subclass, the correct constructor should be setup manually.

The following example creates a subclass `Tiger` of the `Cat` superclass. Notice that initially `Tiger.prototype` still points to `Cat` constructor.

CODE

```

function Tiger(name) {
  Cat.call(this, name);
}
Tiger.prototype = Object.create(Cat.prototype);
// The prototype has the wrong constructor
Tiger.prototype.constructor === Cat // => true
Tiger.prototype.constructor === Tiger // => false

```

Now if we clone a `Tiger` instance using `clone()` method defined on `Cat.prototype`, it will create a wrong `Cat` instance.

CODE

```

var tigerInstance = new Tiger('RrrMew');
var wrongTigerClone = tigerInstance.clone();
tigerInstance instanceof Tiger // => true
// Notice that wrongTigerClone is incorrectly a Cat instance
wrongTigerClone instanceof Tiger // => false
wrongTigerClone instanceof Cat // => true

```

It happens because `Cat.prototype.clone()` uses `new this.constructor()` to create a new clone. But the constructor still points to `Cat` function.

To fix this problem it's necessary to manually update the `Tiger.prototype` with the correct constructor function: `Tiger`. The `clone()` method will be fixed too.

CODE

```

//Fix the Tiger prototype constructor
Tiger.prototype.constructor = Tiger;
Tiger.prototype.constructor === Tiger // => true

var tigerInstance = new Tiger('RrrMew');
var correctTigerClone = tigerInstance.clone();

// Notice that correctTigerClone is correctly a Tiger instance
correctTigerClone instanceof Tiger // => true
correctTigerClone instanceof Cat // => false

```

[Check this demo](#) for a complete example.

3. An alternative to `instanceof`

`object instanceof Class` is used to determine if the `object` has the same prototype as the `Class`.

This operator searches in the prototype chain too, which sometimes makes difficult to identify the subclass instance from superclass instance. For example:

CODE

```

var tigerInstance = new Tiger('RrrMew');

tigerInstance instanceof Cat // => true
tigerInstance instanceof Tiger // => true

```

As seen in the example, it's not possible to check if `tigerInstance` is exactly a `Cat` or `Tiger`, because

`instanceof` returns true in both cases.

This is where the `constructor` property shines, allowing to strictly determine the instance class.

CODE

```
tigerInstance.constructor === Cat // => false
tigerInstance.constructor === Tiger // => true

// or using switch
var type;
switch (tigerInstance.constructor) {
  case Cat:
    type = 'Cat';
    break;
  case Tiger:
    type = 'Tiger';
    break;
  default:
    type = 'unknown';
}
type // => 'Tiger'
```

4. Get the class name

The function object in JavaScript has a property [name](#). It returns the name of the function or an empty string for anonymous one.

In addition with `constructor` property, this can be useful to determine the class name, as an alternative to `Object.prototype.toString.call(objectInstance)`.

CODE

```
var reg = /\d+/";
reg.constructor.name // => 'RegExp'
Object.prototype.toString.call(reg) // => '[object RegExp]'

var myCat = new Cat('Sweet');
myCat.constructor.name // => 'Cat'
Object.prototype.toString.call(myCat) // => '[object Object]'
```

Because `name` returns an empty string for an anonymous function (however in ES6 the name can be inferred), this approach should be used carefully.

Conclusion

The `constructor` property is a piece of the inheritance mechanism in JavaScript. Precautions should be taken when creating hierarchies of classes.

However it offers nice alternatives to determine the type of an instance.

See also

[Object.prototype.constructor](#)

[What's up with the constructor property in JavaScript?](#)

68. 16 Common JavaScript Gotchas

We all know that JavaScript can trip you up. Here are a 16 common traps that can trip you up when coding javascript. You likely know most of the code on the page, but if you keep these 16 gotchas in your mind, coding and debugging will be less of a headache:

1. [Case sensitivity](#): Variable names, properties and methods are all case sensitive
2. [Mismatching quotes, parenthesis or curly braces will throw an error](#)
3. [Conditional Statements](#): 3 common gotchas
4. [Line breaks](#): Always end statements in semi-colons to avoid common line break issues
5. [Punctuation](#): Trailing commas in object declarations will trip you up
6. [HTML id conflicts](#)
7. [Variable Scope](#): global versus local scope
8. [string replace function isn't global](#)
9. [parseInt should include two arguments](#)
10. ['this' and binding issues](#)
11. [Function overloading](#): Overwriting functions, as overloading doesn't exist
12. [Setting default values for parameters in case you omit them](#)
13. [For each loops are for objects, not arrays](#)
14. [switch statements are a little tricky](#)
15. [Always check for Undefined before checking for null](#)
16. [Function location may matter](#)

Case Sensitivity

Variables and function names are case sensitive. Like mismatched quotes, you already know this. But, since the error may be silent, here is the reminder. Pick a naming convention for yourself, and stick with it. And, remember that native javascript function and CSS properties in javascript are camelCase.

```
getElementsByID('myId') != getElementByID('myId'); // it should be "Id" not "ID"
getElementById('myId') != getElementById('myID'); // "Id" again does not equal "ID"
```

```
document.getElementById('myId').style.Color; // returns "undefined"
```

Mismatching quotes, parenthesis and curly brace

The best way to avoid falling into the trap of mismatched quotes, parentheses and curly brackets is to always code your opening and closing element at the same time, then step into the element to add your code. Start with:

```
var myString = ""; //code the quotes before entering your string value
function myFunction(){
    if(){ //close out every bracket when you open one.

    }
}

//count all the left parens and right parens and make sure they're equal
alert(parseInt(var1)*(parseInt(var2)+parseInt(var3))); //close every parenthesis when a parenthesis is open
```

Every time you open an element, close it. Put the arguments for a function into the parenthesis after you've

added the closing parenthesis. If you have a bunch of parenthesis, count the opening parenthesis and then the closing parenthesis, and make sure those two numbers are equal.

Conditional statements (3 gotchas)

1. All conditional comments must be within parentheses (duh!)

```
if(var1 == var2){}
```

CODE

2. Don't get tripped up by accidentally using the assignment operator: assigning your second argument's value to your first argument. Since it's a logic issue, it will always return true and won't throw an error.

```
if(var1 = var2){} // returns true. Assigns var2 to var1
```

CODE

3. Javascript is loosely typed, except in [switch statements](#). JavaScript is NOT loosely typed when it comes to case comparisons.

```
var myVar = 5;
if(myVar == '5'){ // returns true since Javascript is loosely typed
    alert("hi"); //this alert will show since JS doesn't usually care about data type.
}
switch(myVar){
    case '5':
        alert("hi"); // this alert will not show since the data types don't match
}
```

CODE

Line Breaks

- Beware of hard line breaks in JavaScript. Line breaks are interpreted as line-ending semicolons. Even in a string, if you include a hard line break in between quotes you'll get a parse error (unterminated string).

```
var bad = '<ul id="myId">
            <li>some text</li>
            <li>more text</li>
        </ul>'; // unterminated string error

var good = '<ul id="myId">' +
            '<li>some text</li>' +
            '<li>more text</li>' +
        '</ul>'; //correct
```

CODE

- The line break being interpreted as a semi-colon rule, discussed above, does not hold true in the case of control structures: line breaks after the closing parenthesis of a conditional statement is NOT given a semi-colon.

Always use semi-colons and parenthesis so you don't get tripped up by breaking lines, so your code is easier to read, and, less thought of but a source of quirks for those who don't use semicolons: so when you move code around and end up with two statements on one line, you don't have to worry that your first statement is correctly closed.

Extra commas

The last property in any JavaScript object definition must never end with a comma. Firefox won't barf on the trailing, unnecessary commas, but IE will.

HTML id conflicts

The [JavaScript DOM bindings](#) allow indexing by HTML id. Functions and properties share the same namespace in JavaScript. So, when an id in your HTML has the same name as one of your functions or properties, you can get logic errors that are hard to track down. While this is more of a [CSS best practice](#) issue, it's important to remember when you can't solve your javascript issue.

```
var listitems = document.getElementsByTagName('li');

var liCount = listitems.length; // if you have <li id="length">, returns that <li> instead of a count.
```

CODE

If you're marking up (X)HTML, never use a javascript method or property name as the value for an ID. And, when you're coding the javascript, avoid giving variables names that are ID values in the (X)HTML.

variable scope

Many problems in javascript come from variable scope: either thinking that a local variable is global, or overwriting a global variable unwittingly with what should be a local variable in a function. To avoid issues, it's best to basically not have any global variables. But, if you have a bunch, you should know the "gotchas".

Variables that are not declared with the `var` keyword are global. Remember to declare variables with the `var` keyterm to keep variables from having global scope. In this example, a variable that is declared within a function has global scope because the `var` ke

```
anonymousFunction1 = function(){
    globalvar = 'global scope'; // globally declared because "var" is missing.
    return localvar;
}();

alert(globalvar); // alerts 'global scope' because variable within the function is declared globally

anonymousFunction2 = function(){
    var localvar = 'local scope'; //locally declared with "var"
    return localvar;
}();

alert(localvar); // error "localvar is not defined". there is no globally defined localvar
```

CODE

Variable names that are introduced as parameter names are local to the function. There is no conflict if your parameter name is also the name of a global variable as the parameter variable has local scope. If you want to change the global variable from within a function that has a parameter duplicating the global variable's name, remember that global variables are properties of the `window` object.

CODE

```

var myscope = "global";

function showScope(myscope){
    return myscope; // local scope even though there is a global var with same name
}
alert(showScope('local'));

function globalScope(myscope){
    myscope = window.myscope; // global scope
    return myscope;
}
alert(globalScope('local'));

```

You should even declare variables within loops

CODE

```
for(var i = 0; i < myarray.length; i++){};
```

string replace

A common mistake is assuming the behavior of the string replace method will impact all possible matches. Infact, the javascript string replace method only changes the first occurrence. To replace all occurrences, you need to set the global modifier.

CODE

```

var myString = "this is my string";
myString = myString.replace(/ /,"%20"); // "this%20is my string"
myString = myString.replace(/ /g,"%20"); // "this%20is%20my%20string"

```

parseInt

The most common error with parding integers in javascript is the assumption that parseInt returns the integer to base 10. Don't forget the second argument, the radix or base, which can be anything from 2 to 36. To ensure you don't screw up, always include the second parameter.

CODE

```
parseInt('09/10/08'); //0parseInt('09/10/08',10); //9, which is most likely what you want from a date.
```

'this'

Another common mistake is forgetting to use ' this '. Functions defined on a JavaScript object accessing properties on that JavaScript object and failing to use the 'this' reference identifier. For example, the following is incorrect:

CODE

```

function myFunction() {
    var myObject = {
        objProperty: "some text",
        objMethod: function() {
            alert(objProperty);
        }
    };
    myObject.objMethod();
}

function myFunction() {
    var myObject = {
        objProperty: "some text",
        objMethod: function() {
            alert(this.objProperty);
        }
    };
    myObject.objMethod();
}

```

There's an [A List Apart](#) article that puts this binding issue into plain English

Overwriting functions / overloading functions

When you declare a function more than once, the last declaration of that function will overwrite all previous version of that function throwing no errors or warnings. This is different from other programming languages, like java, where you can have multiple functions with the same name as long as they take different arguments: called function overloading. There is no overloading in javascript. This makes it vitally important to not use the names of core javascript functions in your code. Also, beware of including multiple javascript files, as an included script may overwrite a function in another script. Use anonymous functions and namespaces.

CODE

```

(function(){
    // creation of my namespace
    // if namespace doesn't exist, create it.    if(!window.MYNAMESPACE) {           wind
ow['MYNAMESPACE'] = {};
}

// this function only accessible within the anonymous function
function myFunction(var1, var2){
    //local function code goes here
}

/* attaches the local function to the namespace
   making it accessible outside of the anonymous function with use of namespace */
window['MYNAMESPACE']['myFunction'] = myFunction;

})();// the parenthesis = immediate execution    // parenthesis encompassing all the code m
ake the function anonymous

```

Missing Parameters

A common error is forgetting to update all the function calls when you add a parameter to a function. If you need to add a parameter to handle a special case call in your function that you've already been calling, set a default value for that parameter in your function, just in case you missed updating one of the calls in one of your scripts.

CODE

```
function addressFunction(address, city, state, country){
    country = country || "US"; // if country is not passed, assume USA
    //rest of code
}
```

You can also get the length of the argument array. But we're focusing on "gotchas" in this post.

For each

The "for" loop in javascript will iterate it over all object attributes, both methods and properties, looping thru all of the property names in an object. The enumeration will include all of the properties--including functions and prototype properties that you might not be interested in--so filter out the values you don't want using `hasOwnProperty` method and `typeof` to exclude functions. Never use `for each` to iterate thru an array: only use `for each` when needing to iterate thru object properties and methods.

- `for each (var myVar in myObject)` iterates a specified variable over all values of object's properties.
- `for (var myVar in myObject)` iterates a specified variable over all the properties of an object, in arbitrary order. The `for...in` loop does not iterate over built-in properties. For each distinct property the code is executed
- `for (var i=0; i < myArray.length; i++)` iterates thru all the elements of an array.

To fix the problem, generally you'll want to opt for `for ... in` for objects and use the `for` loop for arrays:

CODE

```
listItems = document.getElementsByTagName('li');

for each (var listItem in listItems){
    // this goes thru all the properties and methods of the object,
    // including native methods and properties, but doesn't go thru the array: throws error!
}

//since you have an array of objects that you are looping thru, use the for loop
for ( var i = 0; i < listItems.length; i++) {
    // this is what you really wanted
}
```

Switch statements

I wrote a whole blog post on [switch statement quirks](#), but the gist is:

- there is no data type conversion
- once there is a match, all expressions will be executed until the next break or return statement is executed, and
- you can include multiple cases for a single block of code

Undefined ? null

Null is for an object, undefined is for a property, method or variable. To be null, your object has to be defined. If your object is not defined, and you test to see whether it's null, since it's not defined, it can't test, and will throw an error.

CODE

```

if(myObject !== null && typeof(myObject) !== 'undefined') {
    //if myObject is undefined, it can't test for null, and will throw an error
}

if(typeof(myObject) !== 'undefined' && myObject !== null) {
    //code handling myObject
}

```

[Harish Mallipeddi](#) has an explanation of undefined versus null

Function location can matter

There are two ways to declare function: When declaring a function in the form of a variable assignment, location does matter:

CODE

```

var myFunction = function(arg1, arg2){
    // do something here
};

```

A function declared with

CODE

```
function myFunction (arg1, arg2){}
```

can be used by any code in the file no matter its location (before or after the function declaration). A function declared with the variable assignment syntax can only be used by code that executes after the assignment statement that declares the function.

69. Common JavaScript "Gotchas"

70.

Introduction

JavaScript has a lot of weird behaviours that trip up noobs to the language - especially those acquainted with more traditional OOP languages. Hopefully this guide will provide a quickly scannable, easily understood list to save a lot of pain to those getting acquainted with the language.

The intended audience for this article are engineers from other programming languages who are working with JavaScript for the first time. This article will not focus on detailed explanations of why the language works the way it does. It's merely intended to get you past the early hurdles quickly.

JavaScript is a flexible language with many ways to achieve the same result. What I document below is not the single "best" way. Rather, it's a series of strategies I use to write code in such a way that it is less confusing to people new to the language.

If you take nothing else away from this article, I **highly** recommend you read [JavaScript: The Good Parts, by Douglas Crockford](#)

[Crockford](#). It's the single best resource I'm aware of for getting new JavaScript developers up to speed.

Note that I use [Allman indentation style](#) in this document, despite the fact that a large portion of the JavaScript community frowns upon it (and for reasons I don't completely disagree with). However, I find this indentation style to be much clearer and more readable than the alternatives. I like its gestalt. But please don't take my use of this indentation style as a suggestion that *all* JavaScript should be written using Allman style.

Author:

Steve Kwan

mail@stevekwan.com

<http://www.stevekwan.com/>

Originally from my GitHub:

<https://github.com/stevekwan/best-practices/>

Common Javascript "Gotchas"

Let's talk about some of the JavaScript 101 problems that noobs to the language encounter. If you're relatively unfamiliar with JavaScript, I highly recommend you read this section - it'll save you a lot of pain down the road.

The var keyword: what exactly does it do?

`var` declares a variable. But...you don't need it. Both these will work:

```
var myFunction = function()
{
    var foo = 'Hello'; // Declares foo, scoped to myFunction
    bar = 'Hello';    // Declares bar...in global scope?
};
```

CODE

But you should never, EVER use the latter. See the comment for the reason why.

If you ever forget the `var` keyword, your variable will be declared...but it will be scoped **globally** (to the `window` object), rather than to the function it was declared in.

This is extremely bizarre behaviour, and really an unfortunate design decision in the JavaScript language. There are no pragmatic situations where you would want local variables to be declared globally. So remember to **always** use `var` in your declarations.

Why are there so many different ways to define a function? What are the differences?

In the wild, you'll most commonly see three types of function definitions:

```
myFunction1 = function(arg1, arg2) {};      // NEVER do this!
function myFunction2(arg1, arg2) {};        // This is OK, but...
var myFunction3 = function(arg1, arg2) {}; // This is best!
```

CODE

Assigning the function to a variable via the `=` operator is called a *function expression*. This is what happens to `myFunction1` and `myFunction3`. The syntax used with `myFunction2` is called *function declaration*.

Of the three options, the first option is bad because it assigns the function to a variable without the `var` keyword. This creates a global variable!

The second option is better, and is properly scoped, but it leads to a slightly more complicated syntax once you get into closures. It can also cause your code to behave in ways you may not expect due to [JavaScript variable hoisting](#).

The third option is best, is syntactically consistent with the rest of your variables, and doesn't throw the function into global scope.

There are other ways to define a function which can be useful for debugging. See [Appendix A](#) for details.

The `this` keyword: how does it behave?

`this` in JavaScript does **not** behave the way you would expect. Its behaviour is very, very different from other languages.

If you are merely looking for a way to encapsulate your code or create a Singleton, you may be better off using the Module Pattern ([here's an example](#)) than something relying on `this`. But if you truly need to use `this` to write object-oriented JavaScript, here's an explanation...

In more sane languages, `this` gets you a pointer to the current object. But in JavaScript, it means something quite different: it gets you a pointer to the **calling context** of the given function.

This will make much more sense with an example:

```
CODE
var myFunction = function()
{
    console.log(this);
};

var someObject = {};           // Create an empty object. Same as: new Object();
someObject.myFunction = myFunction; // Give someObject a property
someObject.myFunction();        // Logs Object
myFunction();                  // Logs...Window?
```

That's bizarre, isn't it? Depending on how you call a function, its `this` pointer changes.

In the first example, because `myFunction` was a property of `someObject`, the `this` pointer referred to `someObject`.

But in the second example, because `myFunction` wasn't a property of anything, the `this` pointer defaults to the "global" object, which is `window`.

The important take-away here is that **this points to whatever is "left of the dot."**

Note that you can actually override what `this` points to by using the built-in `call()` and `apply()` functions.

As you can imagine, this causes a ton of confusion - particularly for new JavaScript developers. My recommendation is to avoid writing code in such a way that relies on the intricacies of `this`.

WTF are constructor and prototype ?

If you're asking this question, it means you're getting knee-deep into JavaScript OOP. Good for you!

The first thing you need to know is that JavaScript does NOT use classical OOP. It uses something called prototypal OOP. This is very, very different. If you really want to know how JavaScript OOP works, you need to read [Constructors considered mildly confusing, by Joost Diepenmaat](#). Joost does a better job of explaining it than I ever will.

But for the lazy, I'll summarize: JavaScript does not have any classes. You don't create a class and spawn new objects off of it like in other languages. Instead, you create a new object, and set its "prototype" as the old one.

When you refer to an object's property, if that property doesn't exist JavaScript will look at its prototype...and its prototype, and its prototype, all the way up until it hits the `Object` object.

So unlike the class/object model you see in other languages, JavaScript relies on a series of "parent pointers."

`constructor` is a pointer to the function that gets called when you use the `new` keyword.

This is best explained with an in-depth code example, so read this [constructor vs prototype experiment](#) if you want to learn specifically how `constructor` and `prototype` play together.

WTF is `__proto__`, and how is it different from `prototype` ?

If you've tried debugging your JavaScript in Chrome Inspector, you may have noticed a `__proto__` property on all your objects. Try copying and pasting this code Chrome Inspector's console, and dig into the `__proto__` property to see what I mean.

```
var ParentObject = function()
{
    // If ParentObject had constructor code, it would go here
};

ParentObject.prototype.parentProperty = 'foo';

// Again, note we're not using a classical OOP class/object relationship here.
// In JavaScript, objects just "point" to their parent via prototypes.
var ChildObject = new ParentObject();

ChildObject.childProperty = 'bar';

console.log("ChildObject:");
console.dir(ChildObject); // Outputs a viewable object

console.log("ParentObject:");
console.dir(ParentObject);
```

CODE

For me, that outputs a `ChildObject` that looks like this:

CODE

```

ChildObject:
ParentObject
  childProperty: "bar"
  __proto__: Object
  constructor: function ()
  parentProperty: "foo"
  __proto__: Object

```

And a `ParentObject` that looks like this:

CODE

```

ParentObject:
function () { // If ParentObject had constructor code, it would go here }
arguments: null
caller: null
length: 0
name: ""
prototype: Object
  constructor: function ()
  parentProperty: "foo"
  __proto__: Object
__proto__: function Empty() {}

```

I highly recommend checking this out yourself in Chrome Inspector, as it's a great debugging tool.

Let's look at `ChildObject` first. You'll notice it contains `childProperty`, which is what we would expect. It also has a property called `__proto__`, which itself is an object. Among its contents, it contains `parentProperty`.

Knowing this, you could reasonably assume that `__proto__` is the same as `prototype`. Unfortunately, that's incorrect. The difference between `prototype` and `__proto__` is subtle, but it's very important.

If you look at the output for `ParentObject`, you'll see that it contains both `__proto__` AND `prototype`, making the situation even more confusing. So what's going on?

Well, it turns out that every function gets an automatically-created property called `prototype`. That's right, **every** function gets this property, because any function can be used as a constructor. As we discussed earlier, JavaScript uses prototype properties like this to figure out your object inheritance chain.

When you create a new object using the syntax:

CODE

```
var ChildObject = new ParentObject();
```

JavaScript will look in `ParentObject`, find its `prototype` property, and copy it into `ChildObject`. But it will **not** copy it into `ChildObject` as `prototype`...it will copy it in as `__proto__`.

So why not copy it in as `prototype`? For a variety of reasons, the main one I am aware of being that it is totally possible for an object to have both a `prototype` **and** a `__proto__`. In fact, this is quite common, because all functions have a `prototype` and all objects have a `__proto__`. Heck, take a look at our `ParentObject` up above...it definitely does.

This can lead to an interesting scenario where you create a `ChildObject`, and then change the `prototype` property of the `ParentObject` later. If you do this, `ChildObject` will *continue to use the old prototype*.

Let's see an example:

CODE

```

var ParentObject = function()
{
    // If ParentObject had constructor code, it would go here
};

ParentObject.prototype.parentProperty = 'foo';

// Again, note we're not using a classical OOP class/object relationship here.
// In JavaScript, objects just "point" to their parent via the prototype
// property.
var ChildObject = new ParentObject();

ChildObject.childProperty = 'bar';

delete ParentObject.prototype;

console.log("ChildObject:");
console.dir(ChildObject);

```

Even though we have removed the `prototype` property from `ParentObject`, outputting `ChildObject` will continue to look like this:

CODE

```

ChildObject:
ParentObject
    childProperty: "bar"
    __proto__: Object
        constructor: function ()
        parentProperty: "foo"
        __proto__: Object

```

That's right, it will *still have its parent pointer*.

So to recap: **prototype goes on constructors that create objects, and `__proto__` goes on objects being created.**

One more thing: for the love of beer and pork tacos, please don't ever try to manipulate the `__proto__` pointer. JavaScript is not supposed to support editing of `__proto__` as it is an internal property. Some browsers will let you do it, but it's a bad idea to rely on this functionality. If you need to manipulate the prototype chain you're better off using `hasOwnProperty()` instead.

WTF is a closure?

Closures are a concept that appear in functional languages like JavaScript, but they have started to trickle their way into other languages like PHP and C#. For those unfamiliar, "closure" is a language feature that ensures variables never get destroyed if they are still required.

This explanation is not particularly meaningful in and of itself, so here's an example:

CODE

```
// When someone clicks a button, show a message.
var setup = function()
{
    var clickMessage = "Hi there!";
    $('button').click
    (
        function()
        {
            window.alert(clickMessage);
        }
    );
};
setup();
```

In a language without closures, you'd likely get an error when that click handler fires because `clickMessage` will be undefined. It'll have fallen out of scope long ago.

But in a language with closures (like JavaScript), the engine *knows* that `clickMessage` is still required, so it keeps it around. When a user clicks a button, they'll see, "Hi there!"

As you can imagine, closures are particularly useful when dealing with event handling, because an event handler often gets executed long after the calling function falls out of scope.

Because of closures, we can go one step further and do something cool like this!

CODE

```
(function()
{
    var clickMessage = "Hi there!";
    $('button').click
    (
        function()
        {
            window.alert(clickMessage);
        }
    );
})();
```

In the above example, we don't even need to give the function a name! Instead, we execute it once with the `()` at the end, and forget about it. Nobody can ever reference the function again, but *it still exists*. And if someone clicks that button, it will still work!

Why is `parseInt()` mucking up when I get into big numbers?

Despite looking like it, JavaScript doesn't actually have an integer data type - it only has a floating point type. This isn't an issue when you do:

CODE

```
parseInt("1000000000000000", 10) < parseInt("1000000000000001", 10); //true
```

but add one more zero:

```
parseInt("1000000000000000", 10) < parseInt("1000000000000001", 10); //false
```

And you'll see where the difference between integers and floating points manifests.

Why does JavaScript have so many different ways to do the same thing?

So you can choose which way is best for you. Some parts of JavaScript are not designed that well. Your best guide to muddle through it is to read [JavaScript: The Good Parts, by Douglas Crockford](#). He clearly outlines which pieces of the language you should ignore.

Appendix A: Other ways to define a function

As discussed earlier, there are many ways to define a function in JavaScript. In addition to the syntaxes described earlier, you can use:

```
var myFunction4 = function myFunction4(arg1, arg2) {};
```

Although more verbose, this option can be useful because it provides a little more context when debugging. This is a *named function expression*, which gives the function a `name` property. That property shows up when debugging.

But be aware of which name is used:

```
var myFunction5 = function aDifferentName(arg1, arg2) {};
console.log(myFunction5.name); // logs "aDifferentName"
```

Javascript minifiers such as YUI Compressor and UglifyJS often rename your functions, which can reduce the usefulness of this technique.

Acknowledgements

71. Short Guide to Javascript Gotchas

This is a blog post of our [Code Reading Wednesdays](#) from [Codacy](#) (<http://www.codacy.com>): we make code reviews easier and automatic.

Recently I have been working mostly in Javascript static code analysis and error detection. In this blog post I will to dive into Javascript scoping details, try to explain a few common errors that we, as developers, make and show how you can prevent those issues.

Javascript Scopes

Much has been written about Javascript scopes. It is a difficult concept to master and even experienced

programmers sometimes make incorrect assumptions about how scopes work in Javascript. This is not a comprehensive explanation, but I will try to cover a few important details.

Function-scope vs Block-scope

Unlike other languages, Javascript has function level scope. This means that the declared variables/functions are accessible from anywhere within the function they were declared in (or from the global scope, if that is the case). This contrasts with block-scope which restricts the variable scope to the block they were declared in. Consider the following code:

```
var a = 3;

function foo() {
    var a = 2
    console.log(a);
}

if(true)
{
    var a = 1;
    console.log(a)
}

foo();
console.log(a)
```

CODE

If you execute this, it will print 1 2 1. The `if` statement does not create a new scope, therefore, the variable declaration inside it belongs to the global scope and overrides the previously defined value of `a`. On the other hand, we can see that the function `foo` manipulates its own definition of `a`.

Hoisting

In Javascript, functions and variables are always declared at the beginning of their scope. If you didn't declare them at the beginning of the scope, the Javascript interpreter will move them there. This feature, usually referred to as *hoisting*, may cause unexpected results for those who are unaware of it.

Let's look at some examples:

```
console.log(a);
var a = 1;
```

CODE

Usually when you try to use an undeclared variable you get a `ReferenceError`, but in this case the output is `undefined` because `a` was *hoisted* to the beginning of the scope. Here is what was actually executed by the interpreter:

```
var a;

console.log(a); //undefined
a = 1;
```

CODE

Notice how the variable declaration is moved to the top, but the initialisation remains in the same line. *Hoisting* is also applied to function declarations, here is another example:

CODE

```
var a = 1;
function a() {
    return 2;
}

console.log(a) //1
console.log(a()) //TypeError: a is not a function
```

Here, the declarations are moved to the top and the assignment `a = 1` overrides the function body. Note that function expressions, that usually take the form of `var a = function () { }`, follow the same rules as variable declarations, which means that the variable name will be hoisted but not the function body. A more detailed explanation of the differences between function declarations and function expressions can be found [here](#).

As you can see, declaring variables and functions with the same name can cause a lot of problems. Ideally, all variables should be declared at the top of the scope to avoid confusion.

Implicit declarations

Implicitly declared variables do not throw an error in Javascript. Instead, they are added as a property to the global scope.

CODE

```
var a = 1;
function createGlobal(){
    b = 2;
}
createGlobal();

console.log(a); //1
console.log(b); //2
```

After invoking the function `createGlobal`, the variable `b` was added to the global scope and became accessible from anywhere. Using global variables is already a bad practice; silently declaring a global variable with an implicit statement only makes it worse.

One way to prevent this from happening is to enable Javascript [strict mode](#) because it forbids variable declaration without the `var` identifier, among other features.

Common Javascript errors

Besides scoping problems, there are many other language *gotchas* that you should be careful with, specially if you are new to Javascript. Here are a few common ones:

Use '===' instead of '=='

The difference between these operators is simple:

- `==` compares two values

- `==` compares two values after making the necessary type conversions

A comprehensive explanation of how these type conversions are performed can be found in the [ECMAScript 5.1 documentation](#), but I think that you will get the idea with a few examples:

CODE

```
1 == "1" //true because "1" is converted to 1
1 === "1" //false

1 == true //true because true is converted to 1
1 === true //false

undefined == null //true - specific case detailed in the documentation
undefined === null //false

"hello" == "hello" //true
"hello" === "hello" //true
```

The `==` comparison is dangerous because its behaviour depends on the type of data being compared. Most of the time, you want to make an explicit comparison without any surprises so you should use `===`.

Always specify radix parameter in parseInt

The second argument of the `parseInt` function is used to specify a radix for the number conversion. If no radix is specified, the function can return unexpected results. For example, if the string begins with a 0, the string may be interpreted as an octal number:

CODE

```
parseInt("032") //returns 26
parseInt("032", 10) //returns 32
```

Recent browsers already cast the string to a decimal number in this case, but, in order to prevent compatibility problems, the radix argument should always be used.

string.replace() only replaces the first occurrence

To replace all occurrences within a string you must specify the `/g` flag to the `string.replace` method, otherwise it will only replace the first occurrence of the pattern:

CODE

```
var zed = "Zed's dead, baby. Zed's dead."
zed.replace(/dead/, "alive") //Zed's alive, baby. Zed's dead.
zed.replace(/dead/g, "alive") //Zed's alive, baby. Zed's alive.
```

Missing parameters on function call

All function parameters in Javascript are optional. This means that you can always invoke a function with less parameters than it expects:

CODE

```
function concat(a, b) {
    return a + b
}

concat("hello"); // "helloundefined"
```

While sometimes intended, this is a frequent source of bugs because the function call does not throw any errors at all. The body is executed and the unspecified arguments have the value `undefined`. In order to prevent this, you could set a default value for each parameter or throw an error if any of them is `undefined`.

Conclusion

Writing bug free and maintainable code is a challenge that we face everyday. Despite being the assembly of the web, Javascript language leaves some room for ambiguity. A good code style reviewed by your team as whole is a great starting point to make sure some of these issues don't get into production.

Next time I will try to analyze open source projects and detect some of these issues in real world products.

Share your opinion on [/r/javascript](#)

(shameless plug)

[Codacy](#) already has analysis patterns that detect most of the problems mentioned in this article. For example, using **implicit global variables** or **redeclaring variables with the same name** in the same scope will automatically raise a red flag. If you have other complaints or issues that you want to automatically detect, feel free to contact me to talk about your code.

Nuno ([@nmat](#))

72. A beginner's list of JavaScript gotchas

09 Dec 2013

I consider myself a novice when it comes to coding in JavaScript but lately I played a bit more with this language and stumbled upon a few random things I thought were worth sharing. So, if you are new to JavaScript or just don't know it as well as you would like, I hope you find this article useful.



0 == "0", even though 0 is falsy and "0" is truthy

At first sight, this is very odd, but once you read more about the [comparison operators](#) this make sense. To avoid misunderstandings like this one, be sure to always use the strictly equal operator (`==`) to do this type of comparison. [Lous Lazaris](#) wrote an entire article on the unpredictable results that can occur when the equality comparison operator is not used the right way.

CODE

```
console.log(0 == "0"); // true
console.log(0 === "0"); // false
```

You may want to check this JavaScript [truth table](#) as well, but don't try to memorize it :)

1+2+"3" != "1"+"2"+3

JavaScript is quite tolerant when it's about strings and numbers, just think about the "+" operator, which both adds and concatenates. Unlike other programming languages that would shout at you when encountering the line above, JavaScript actually tries to solve this by assuming, at a time, that numbers are strings too.

```
1+2+"3" != "1"+"2"+3;
// 3+"3" != "12"+3
// 33 !=123
```

CODE

JavaScript hoists variable and function declarations

The following code piece will run without problems because the JavaScript allows you to use the function before the point at which it was declared.

```
sayHello(); //Call the function before its declaration
function sayHello() {
    console.log("Hello!");
}
```

CODE

However, watch out for functions expressions and variable declarations! Although their declarations are hoisted as well, the function's definition and respectively the variable's initialization are not. If you want to read more, [Jeffrey Way](#) and [James Allardice](#) wrote some great articles on variable and function hoisting in JavaScript.

Omitting the var keyword

The `var` keyword helps defining a variable and optionally initializing it to a value. But, when you omit it, the code still works and the JavaScript interpreter displays no error. That happens because the respective variable will be created in the global scope and not in the local scope you defined it in. This kind of behavior is often described as a JavaScript pitfall and it's highly recommended to avoid it as it can create [unexpected results](#).

```
var myFunction = function() {
    var foo = 'Hello'; // Declares and initialize foo, scoped to myFunction
    bar = 'World!';    // This works too, but the scope is global
};
```

CODE

The semicolon is optional, but...

You can go crazy and write JavaScript statements without caring about adding semicolons at the end. But there's a catch. Even though JavaScript automatically inserts semicolons at the end of each line, in some cases, especially if you're an [Allman style](#) indentation fan, problems may appear.

```
// OK
return {
  'ok': false
};

// not OK
return
{
  'ok': false
};
```

CODE

Douglas Crockford gave the best example on this matter, and why should always use semicolons in the JavaScript code. At first sight, the above code pieces look like they do the same: return an object. In reality, the second one will fail because the returned value is `undefined`, due the automatic semicolon injection.

```
return; // returns undefined

// code block that does nothing
{
  'ok': false
}
```

CODE

Also, if you didn't already, you should try to adopt Douglas Crockford's [JavaScript code convention](#).

The console is your friend

You already know the "*Google is your friend*" phrase, with that in mind I'd say "*The console is your friend*" applies best in this case. The `console` object has some interesting methods you can use to debug your JavaScript code, the `log()` being by far the most known.

I think the `console.log()` doesn't need any introduction anymore, its non-blocking behavior and nice formatting was a big plus comparing with `alert()`. The issue here is that, even though the `console.log` was implemented in IE8 and IE9, the `console` object somehow is not created until you toggle the DevTools.

So, found out on my own skin that if you let any `console.log` calls in your code, it will break your code on browsers like [IE8 and IE9](#). After browsing for a solution, I found the following as a good and simple solution. Basically, if the `window` object does not have access to browser's debugging console, just override the `console.log` with a dummy function that does nothing at all.

```
if (!window.console) {
  console.log = function(){};
}
```

CODE

Colorful log messages

But if logging methods like `log`, `info`, `error` & `warn` are not enough to distinguish your messages, you can try [adding your own styling](#) to the `log` messages.

```
CODE
var consoleStyling = 'background: #0f0; color: #fff; font-weight: bold;';
console.log('%c A colorful message', consoleStyling);
```

Debugging with console.table()

Again, if using `console.log()` feels too mainstream, you could try some advanced JavaScript debugging with [`console.table\(\)`](#). A very nice feature of `console.table()` is that it also works with objects.

```
CODE
var browsers = {
  chrome: { name: "Chrome", engine: "WebKit" },
  firefox: { name: "Firefox", engine: "Gecko" }
};

console.table(browsers);
```

Final words

When reading more about how JavaScript actually works, the following question will inevitably pop up in your head: **Why does JavaScript have so many different ways to do the same thing?** Maybe the answer is so we can choose our own way, with the best practices in mind.

To get back, the above are just some things I got in mind when I wrote this article, but there are so many more. Thank you for reading and I'm looking forward to read your comments as well on what JavaScript pieces of code did you found as being the most intriguing.

73. 19+ JavaScript Shorthand Coding Techniques

This really is a must read for any JavaScript based developer. I have made this post as a vital source of reference for learning shorthand JavaScript coding techniques that I have picked up over the years. To help you understand what is going on I have included the longhand versions to give some coding perspective on the shorts.

Update 05/05/2013 – There are now 19 strong keep em coming! :)

- Step 1 – Learn the JavaScript shorthand techniques.
- Step 2 – Save valuable coding time by keeping your code to a minimum.
- Step 3 – Impress your colleagues with your awesome new coding skillz!

It's as easy as steps 1,2,3. Here they are.

1. If true ... else Shorthand

This is a great code saver for when you want to do something if the test is true, else do something else by using the ternary operator.

Longhand:

CODE

```

var big;
if (x > 10) {
    big = true;
}
else {
    big = false;
}

```

Shorthand:

CODE

```
var big = (x > 10) ? true : false;
```

If you rely on some of the weak typing characteristics of JavaScript, this can also achieve more concise code. For example, you could reduce the preceding code fragment to this:

CODE

```

var big = (x > 10);

//further nested examples
var x = 3,
big = (x > 10) ? "greater 10" : (x < 5) ? "less 5" : "between 5 and 10";
console.log(big); // "less 5"

var x = 20,
big = {true: x>10, false : x<=10};
console.log(big); // "Object {true=true, false=false}"

```

2. Null, Undefined, Empty Checks Shorthand

When creating new variables sometimes you want to check if the variable your referencing for it's value isn't null or undefined. I would say this is a very common check for JavaScript coders.

Longhand:

CODE

```

if (variable1 !== null || variable1 !== undefined || variable1 !== '') {
    var variable2 = variable1;
}

```

Shorthand:

CODE

```
var variable2 = variable1 || '';
```

Don't believe me? Test it yourself (paste into Firebug and click run):

CODE

```
//null value example
var variable1 = null;
var variable2 = variable1 || '';
console.log(variable2);
//output: '' (an empty string)

//undefined value example
var variable1 = undefined;
var variable2 = variable1 || '';
console.log(variable2);
//output: '' (an empty string)

//normal value example
var variable1 = 'hi there';
var variable2 = variable1 || '';
console.log(variable2);
//output: 'hi there'
```

3. Object Array Notation Shorthand

Useful way of declaring small arrays on one line.

Longhand:

CODE

```
var a = new Array();
a[0] = "myString1";
a[1] = "myString2";
a[2] = "myString3";
```

Shorthand:

CODE

```
var a = ["myString1", "myString2", "myString3"];
```

4. Associative Array Notation Shorthand

The old school way of setting up an array was to create a named array and then add each named element one by one. A quicker and more readable way is to add the elements at the same time using the object literal notation.

Please note that Associative Array are essentially JavaScript Objects with properties.

Longhand:

CODE

```
var skillSet = new Array();
skillSet['Document language'] = 'HTML5';
skillSet['Styling language'] = 'CSS3';
skillSet['Javascript library'] = 'jQuery';
skillSet['Other'] = 'Usability and accessibility';
```

Shorthand:

CODE

```
var skillSet = {
  'Document language' : 'HTML5',
  'Styling language' : 'CSS3',
  'Javascript library' : 'jQuery',
  'Other' : 'Usability and accessibility'
};
```

Don't forget to omit the final comma otherwise certain browsers will complain (not naming any names, IE).

5. Declaring variables Shorthand

It is sometimes good practice to include variable assignments at the beginning of your functions. This shorthand method can save you lots of time and space when declaring multiple variables at the same time.

longhand:

```
var x;
var y;
var z = 3;
```

CODE

shorthand:

```
var x, y, z=3;
```

CODE

6. Assignment Operators Shorthand

Assignment operators are used to assign values to JavaScript variables and no doubt you use arithmetic everyday without thinking (no matter what programming language you use Java, PHP, C++ it's essentially the same principle).

Longhand:

```
x=x+1;
minusCount = minusCount - 1;
y=y*10;
```

CODE

Shorthand:

```
x++;
minusCount--;
y*=10;
```

CODE

Other shorthand operators, given that x=10 and y=5, the table below explains the assignment operators:

CODE

```
x += y //result x=15
x -= y //result x=5
x *= y //result x=50
x /= y //result x=2
x %= y //result x=0
```

7. RegExp Object Shorthand

Example to avoid using the RegExp object.

Longhand:

CODE

```
var re = new RegExp("\d+(.)+\d+","igm"),
result = re.exec("padding 01234 text text 56789 padding");
console.log(result); //"01234 text text 56789"
```

Shorthand:

CODE

```
var result = /d+(.)+d+/igm.exec("padding 01234 text text 56789 padding");
console.log(result); //"01234 text text 56789"
```

8. If Presence Shorthand

This might be trivial, but worth a mention. When doing “if checks” assignment operators can sometimes be omitted.

Longhand:

CODE

```
if (likeJavaScript == true)
```

Shorthand:

CODE

```
if (likeJavaScript)
```

Here is another example. If “a” is NOT equal to true, then do something.

Longhand:

CODE

```
var a;
if ( a != true ) {
// do something...
}
```

Shorthand:

CODE

```
var a;
if ( !a ) {
// do something...
}
```

9. Function Variable Arguments Shorthand

Object literal shorthand can take a little getting used to, but seasoned developers usually prefer it over a series of nested functions and variables. You can argue which technique is shorter, but I enjoy using object literal notation as a clean substitute to functions as constructors.

Longhand:

CODE

```
function myFunction( myString, myNumber, myObject, myArray, myBoolean ) {
    // do something...
}
myFunction( "String", 1, [], {}, true );
```

Shorthand (looks long but only because I have `console.log`'s in there!):

CODE

```
function myFunction() {
    console.log( arguments.length ); // Returns 5
    for ( i = 0; i < arguments.length; i++ ) {
        console.log( typeof arguments[i] ); // Returns string, number, object, object, boolean
    }
}
myFunction( "String", 1, [], {}, true );
```

10. JavaScript `foreach` Loop Shorthand

This little tip is really useful if you want plain JavaScript and hence can't use [jQuery.each](#) or `Array.forEach()`.

Longhand:

CODE

```
for (var i = 0; i < allImgs.length; i++)
```

Shorthand:

CODE

```
for(var i in allImgs)
```

Shorthand for `Array.forEach`:

CODE

```
function logArrayElements(element, index, array) {
    console.log("a[" + index + "] = " + element);
}
[2, 5, 9].forEach(logArrayElements);
// logs:
// a[0] = 2
// a[1] = 5
// a[2] = 9
```

11. charAt() Shorthand

You can use the eval() function to do this but this bracket notation shorthand technique is much cleaner than an evaluation, and you will win the praise of colleagues who once scoffed at your amateur coding abilities!

Longhand:

```
"myString".charAt(0);
```

CODE

Shorthand:

```
"myString"[0]; // Returns 'm'
```

CODE

12. Comparison returns

We're no longer relying on the less reliable == as !(ret == undefined) could be rewritten as !(ret) to take advantage of the fact that in an or expression, ret (if undefined or false) will skip to the next condition and use it instead. This allows us to trim down our 5 lines of code into fewer characters and it's once again, a lot more readable.

Longhand:

```
if (!(ret == undefined)) {
    return ret;
} else{
    return fum('g2g');
}
```

CODE

Shorthand:

```
return ret || fum('g2g');
```

CODE

13. Short function calling

Just like #1 you can use ternary operators to make function calling shorthand based on a conditional.

Longhand:

CODE

```
function x() {console.log('x');}function y() {console.log('y');}
var z = 3;
if (z == 3)
{
    x();
} else
{
    y();
}
```

Shorthand:

CODE

```
function x() {console.log('x');}function y() {console.log('y');}var z = 3;
(z==3?x:y)(); // Short version!
```

14. Switch Nightmare

Everyone loves switch statements, *cough*. Here is how you might avoid switch case syndrome.

Longhand:

CODE

```
switch (something) {

    case 1:
        doX();
        break;

    case 2:
        doY();
        break;

    case 3:
        doN();
        break;

    // And so on...

}
```

Shorthand:

CODE

```
var cases = {
    1: doX,
    2: doY,
    3: doN
};
if (cases[something]) {
    cases[something]();
}
```

15. Decimal base exponents

You may have seen this one around it's essentially a fancy way to write without the zeros. 1e7 essentially means 1 followed by 7 zeros – it represents a decimal base (JS interprets as a float type) equal to 10,000,000.

Longhand:

```
for (var i = 0; i < 10000; i++) {
```

CODE

Shorthand:

```
for (var i = 0; i < 1e7; i++) {
```

CODE

16. Decimal base exponents

You can use 1 and 0 to represent true and false. I've seen this used in JavaScript game development in shorthand while loops. Note that if you use the negative start your array may be in reverse. You can also use while(i++<10) and you don't have to add the i++ later on inside the while.

Longhand:

```
var i=0;
while (i<9)
{
    //do stuff
    i++; //say
}
```

CODE

Shorthand:

```
var i=9;
while(i--)
{
    //goes until i=0
}

or

var i=-9;
while(i++)
{
    //goes until i=0
}
```

CODE

17. Shorter IF'z

If you have multiple IF variable value comparisons you can simply ass them to an array and check for presence. You

could use `$.inArray` as an alternative.

Longhand:

```
if( myvar==1 || myvar==5 || myvar==7 || myvar==22 ) alert('yeah')
```

CODE

Shorthand:

```
if([1,5,7,22].indexOf(myvar)!=-1) alert('yeah baby!')
```

CODE

18. Lookup Tables Shorthand

If you have code that behaves differently based on the value of a property, it can often result in conditional statements with multiple else ifs or a switch cases. You may prefer to use a lookup table if there is more than two options (even a switch statement looks ugly!).

Longhand:

```
if (type === 'aligator')
{
    aligatorBehavior();
}
else if (type === 'parrot')
{
    parrotBehavior();
}
else if (type === 'dolphin')
{
    dolphinBehavior();
}
else if (type === 'bulldog')
{
    bulldogBehavior();
}
else
{
    throw new Error('Invalid animal ' + type);
}
```

CODE

Shorthand:

```
var types = {
    aligator: aligatorBehavior,
    parrot: parrotBehavior,
    dolphin: dolphinBehavior,
    bulldog: bulldogBehavior
};

var func = types[type];
if (!func) throw new Error('Invalid animal ' + type); func();
```

CODE

19. Double Bitwise

The double bitwise trick provides us with some pretty nifty shorthand tricks. Read more about it here: [Double bitwise NOT \(~\)](#).

Longhand:

```
Math.floor(4.9) === 4 //true
```

CODE

Shorthand:

```
~~4.9 === 4 //true
```

CODE

20. Suggest one?

I really do love these and would love to find more, please leave a comment!

74. Javascript shorthand for cleaner code

Translation: [Croatian](#)

A few ways to save on some bytes in your Javascript code, as well as making it more readable and quicker to write:

Variable increment/decrement/multiply/divide

When you want to increase or decrease a number variable by one; instead of this:

```
growCount = growCount + 1;
shrinkCount = shrinkCount - 1;
```

CODE

You can simply do the following:

```
growCount++;
shrinkCount--;
```

CODE

Or to add/subtract/multiply/divide a number to/from/by itself you can do:

```
growCout += 100;
shrinkCount -= 2;

moreSweets *= 5; // multiply moreSweets by 5
lessApple /= 2; // divide lessApple by 2
```

CODE

Ternary operator (conditional)

This is a great code saver for when you want to do something if the test is true, else do something else:

```
if(myAge > legalAge) {
    canDrink = true;
}
else {
    canDrink = false;
}
```

CODE

Instead, put the condition before the question mark then the if true statement and false statement after that separated by a colon:

```
var canDrink = (myAge > legalAge) ? true : false;
```

CODE

29-05-2010 As pointed out in the comments, the above example can be further simplified to `var canDrink = myAge > legalAge` because it's returning a boolean.

Associative array notation

The old school way of setting up an array was to create a named array and then add each named element one by one:

```
var skillSet = new Array();
skillSet['Document language'] = 'HTML5';
skillSet['Styling language'] = 'CSS3';
skillSet['Javascript library'] = 'jQuery';
skillSet['Other'] = 'Usability and accessibility';
```

CODE

A quicker and more readable way is to add the elements at the same time using the object literal notation to become:

```
var skillSet = {
    'Document language' : 'HTML5',
    'Styling language' : 'CSS3',
    'Javascript library' : 'jQuery',
    'Other' : 'Usability and accessibility'
};
```

CODE

Don't forget to omit the final comma otherwise certain browsers will complain.

Default assignments

The following is useful if you are testing if a variable has previously been set and if not to try something else:

CODE

```
function displayValues(limit) {
    var length;
    if(limit) {
        length = limit;
    } else {
        length = 10;
    }
    for(var i = 0; i++ <= length) {
        ...
    }
}
```

A shorter way is to use the double pipe. If `limit` has not been passed to the function then `length` will be set to the default of 10:

CODE

```
function displayValues(limit) {
    var length = limit || 10;
    for(var i = 0; i <= length; i++) {
        ...
    }
}
```

02-07-2010 The variable `length` will be set to the value of the left operand if it evaluates to true, therefore anything other than the following:

- `false`
- `0`
- `null`
- `undefined`
- empty string

Otherwise it will be set to the value of the right operand. So this isn't the right thing to use if you need to explicitly set the length to zero.

75. JavaScript Prototype Chains, Scope Chains, and Performance: What You Need to Know

JavaScript can seem like a very easy language to learn at first. Perhaps it's because of its flexible syntax. Or perhaps it's because of its similarity to other well known languages like Java. Or perhaps it's because it has so few data types in comparison to languages like Java, Ruby, or .NET.

But in truth, JavaScript is much less simplistic and more nuanced than most [developers](#) initially realize. Even for [developers with more experience](#), some of JavaScript's most salient features continue to be misunderstood and lead to confusion. One such feature is the way that data (property and variable) lookups are performed and the JavaScript performance ramifications to be aware of.

In JavaScript, data lookups are governed by two things: *prototypal inheritance* and *scope chain*. As a developer, clearly understanding these two mechanisms is essential, since doing so can improve the structure, and often the performance, of your code.

Property lookups through the prototype chain

When accessing a property in a prototype-based language like JavaScript, a dynamic lookup takes places that involves different layers within the object's prototypal tree.

In JavaScript, every function is an object. When a function is invoked with the `new` operator, a new object is created. For example:

```
function Person(firstName, lastName) {  
    this.firstName = firstName;  
    this.lastName = lastName;  
}  
  
var p1 = new Person('John', 'Doe');  
var p2 = new Person('Robert', 'Doe');
```

CODE

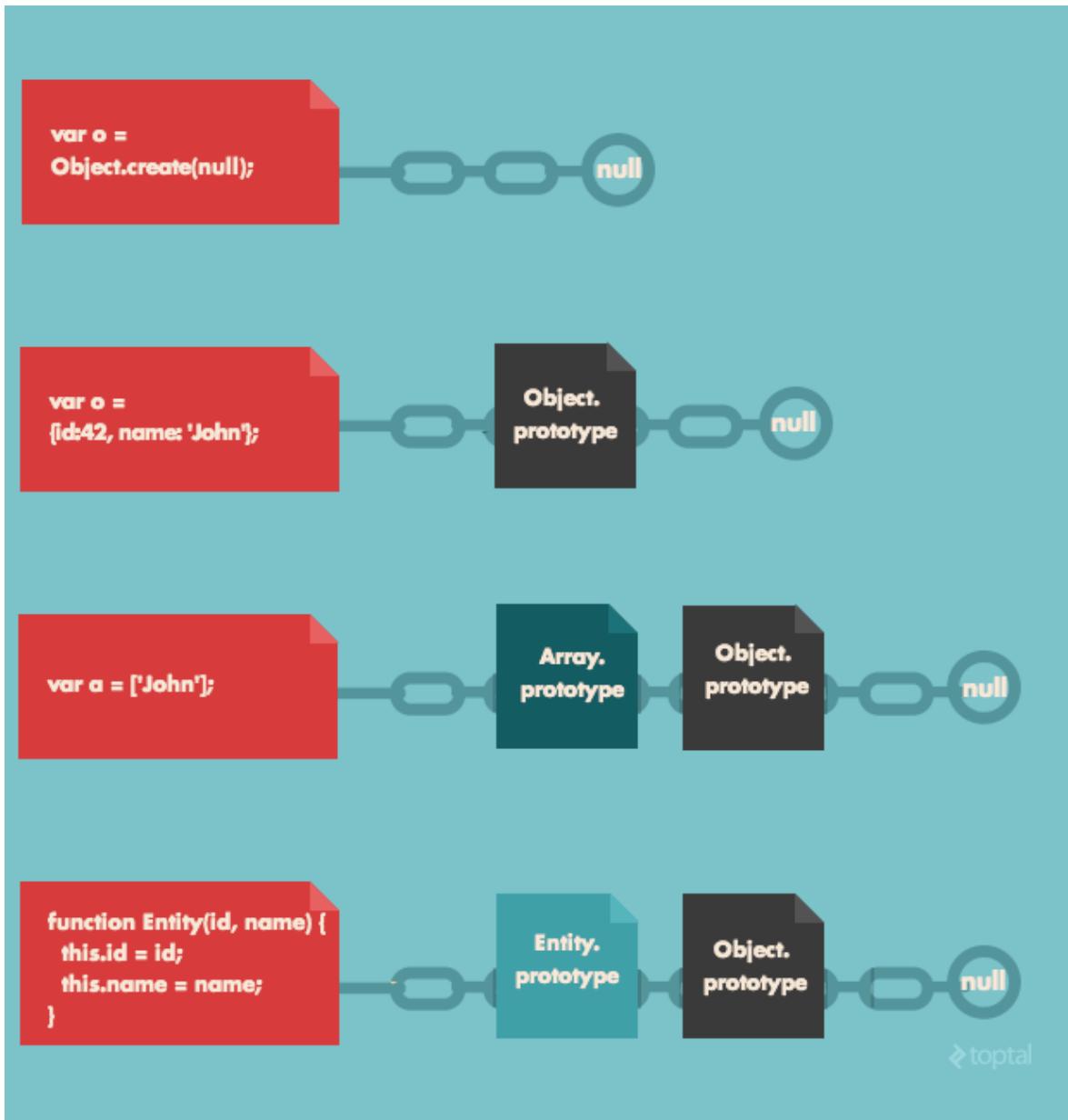
In the above example, `p1` and `p2` are two different objects, each created using the `Person` function as a constructor. They are independent instances of `Person`, as demonstrated by this code snippet:

```
console.log(p1 instanceof Person); // prints 'true'  
console.log(p2 instanceof Person); // prints 'true'  
console.log(p1 === p2);           // prints 'false'
```

CODE

Since JavaScript functions are objects, they can have properties. A particularly important property that each function has is called `prototype`.

`prototype`, which is itself an object, inherits from its parent's prototype, which inherits from its parent's prototype, and so on. This is often referred to as the *prototype chain*. `Object.prototype`, which is always at the end of the prototype chain (i.e., at the top of the prototypal inheritance tree), contains methods like `toString()`, `hasOwnProperty()`, `isPrototypeOf()`, and so on.



Each function's prototype can be extended to define its own custom methods and properties.

When you instantiate an object (by invoking the function using the `new` operator), it inherits all the properties in the prototype of that function. Keep in mind, though, that those instances will not have direct access to the `prototype` object but only to its properties. For example:

```
// Extending the Person prototype from our earlier example to
// also include a 'getFullName' method:
Person.prototype.getFullName = function() {
    return this.firstName + ' ' + this.lastName;
}

// Referencing the p1 object from our earlier example
console.log(p1.getFullName());           // prints 'John Doe'
// but p1 can't directly access the 'prototype' object...
console.log(p1.prototype);              // prints 'undefined'
console.log(p1.prototype.getFullName()); // generates an error
```

CODE

There's an important and somewhat subtle point here: Even if `p1` was created before the `getFullName` method was defined, it will still have access to it because its prototype is the `Person` prototype.

(It is worth noting that browsers also store a reference to the prototype of any object in a `__proto__` property, but it's **really bad practice** to directly access the prototype via the `__proto__` property, since it's not part of the standard [ECMAScript Language Specification](#), so *don't do it!*)

Since the `p1` instance of the `Person` object doesn't itself have direct access to the `prototype` object, if we want overwrite `getFullName` in `p1`, we would do so as follows:

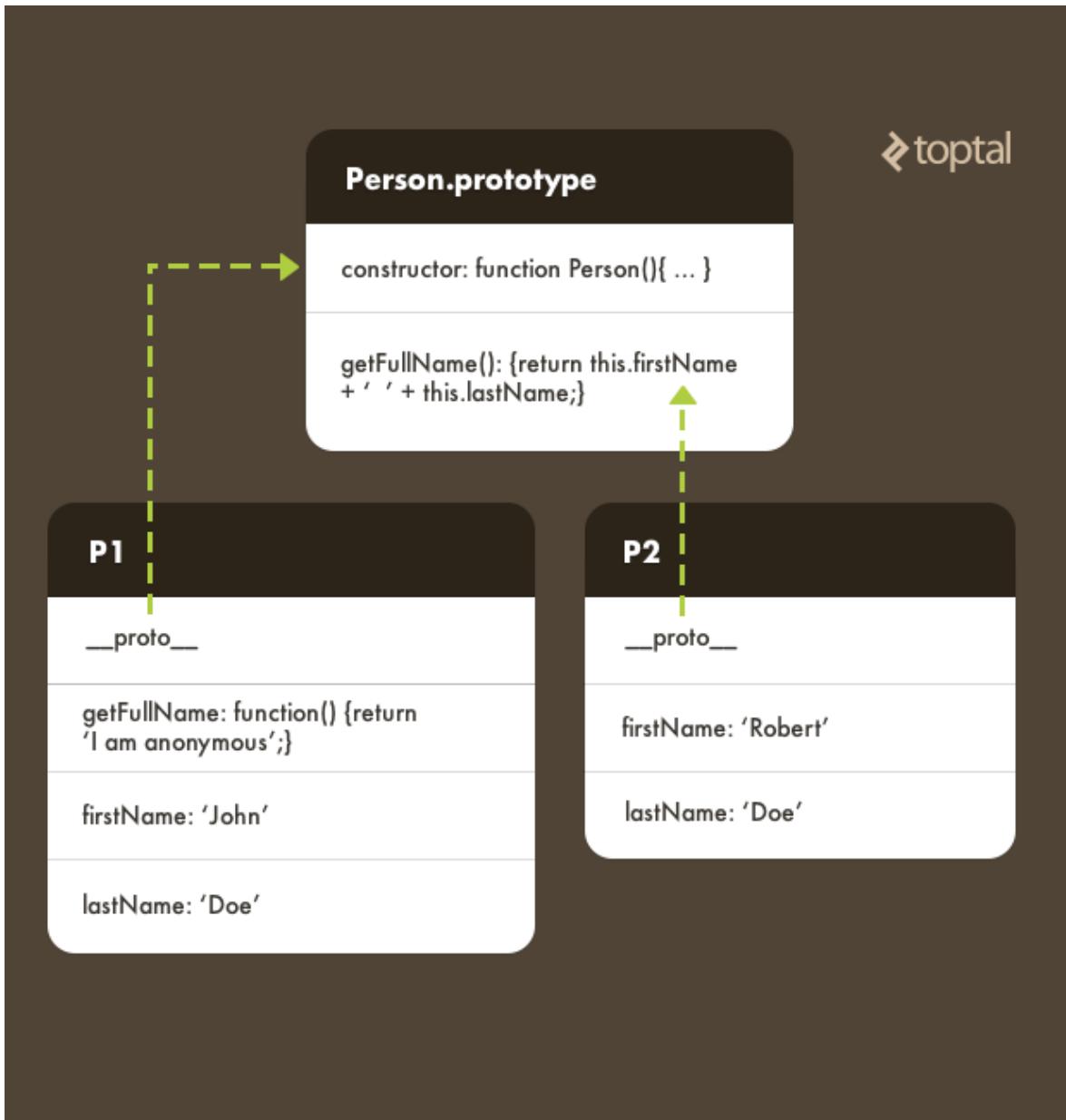
```
// We reference p1.getFullName, *NOT* p1.prototype.getFullName,  
// since p1.prototype does not exist:  
  
p1.getFullName = function(){  
    return 'I am anonymous';  
}
```

CODE

Now `p1` has its own `getFullName` property. But the `p2` instance (created in our earlier example) does *not* have any such property of its own. Therefore, invoking `p1.getFullName()` accesses the `getFullName` method of the `p1` instance itself, while invoking `p2.getFullName()` goes up the prototype chain to the `Person` prototype object to resolve `getFullName`:

```
console.log(p1.getFullName()); // prints 'I am anonymous'  
console.log(p2.getFullName()); // prints 'Robert Doe'
```

CODE



Another important thing to be aware of is that it's also possible to *dynamically* change an object's prototype. For example:

CODE

```
function Parent() {
    this.someVar = 'someValue';
};

// extend Parent's prototype to define a 'sayHello' method
Parent.prototype.sayHello = function(){
    console.log('Hello');
};

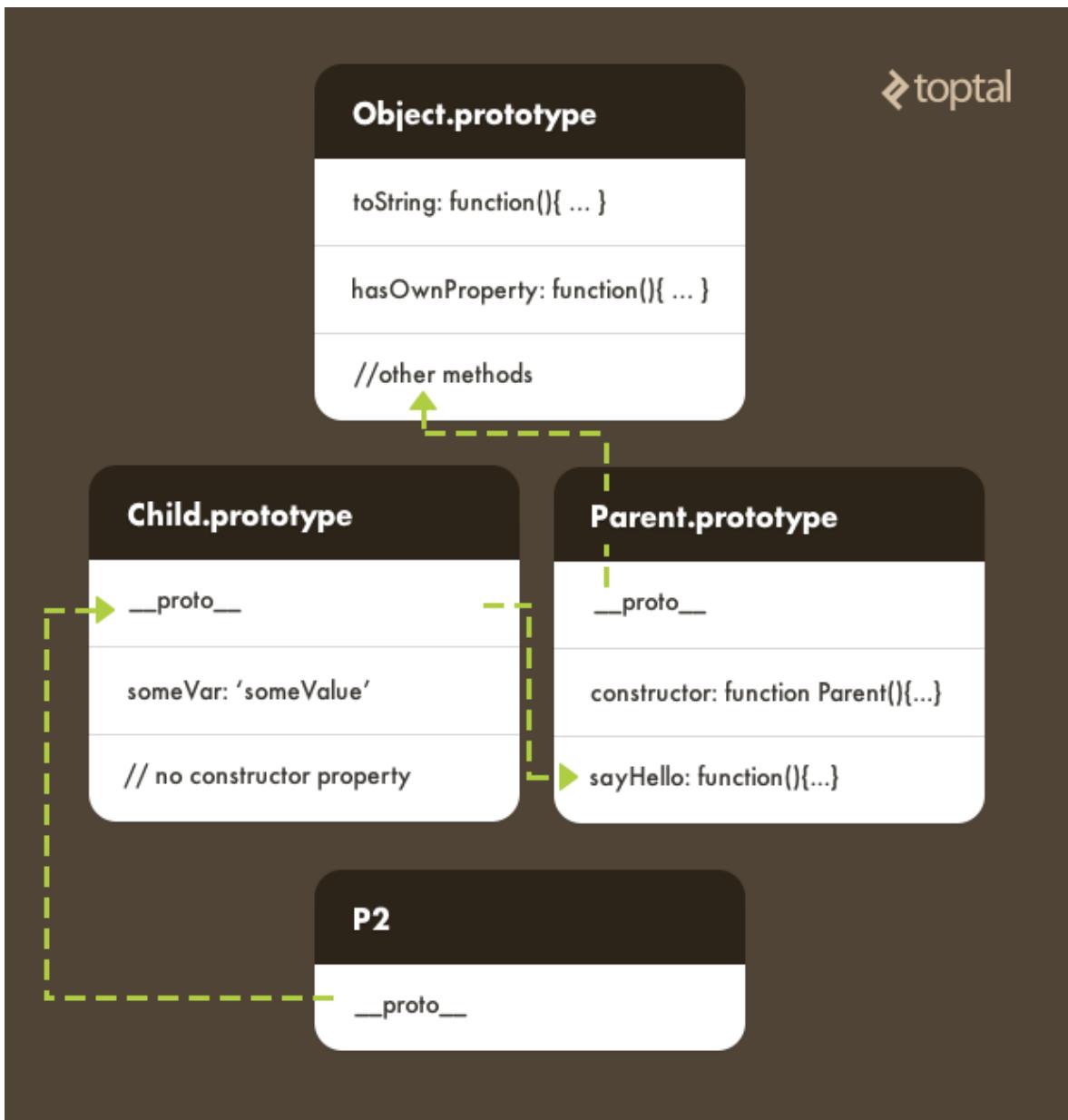
function Child(){
    // this makes sure that the parent's constructor is called and that
    // any state is initialized correctly.
    Parent.call(this);
};

// extend Child's prototype to define an 'otherVar' property...
Child.prototype.otherVar = 'otherValue';

// ... but then set the Child's prototype to the Parent prototype
// (whose prototype doesn't have any 'otherVar' property defined,
// so the Child prototype no longer has 'otherVar' defined!)
Child.prototype = Object.create(Parent.prototype);

var child = new Child();
child.sayHello();           // prints 'Hello'
console.log(child.someVar); // prints 'someValue'
console.log(child.otherVar); // prints 'undefined'
```

When using prototypal inheritance, remember to define properties in the prototype *after* having either inherited from the parent class or specified an alternate prototype.



To summarize, property lookups through the JavaScript prototype chain work as follows:

- If the object has a property with the given name, that value is returned. (The [hasOwnProperty](#) method can be used to check if an object has a particular named property.)
- If the object does not have the named property, the object's prototype is checked
- Since the prototype is an object as well, if it does not contain the property either, its parent's prototype is checked.
- This process continues up the prototype chain until the property is found.
- If `Object.prototype` is reached and it does not have the property either, the property is considered `undefined`.

Understanding how prototypal inheritance and property lookups work is important in general for developers but is also essential because of its (sometimes significant) JavaScript performance ramifications. As mentioned in the documentation for [V8](#) (Google's open source, high performance JavaScript engine), most JavaScript engines use a dictionary-like data structure to store object properties. Each property access therefore requires a dynamic look-up in that data structure to resolve the property. This approach makes accessing properties in JavaScript typically much slower than accessing instance variables in programming languages like Java and Smalltalk.

Variable lookups through the scope chain

Another lookup mechanism in JavaScript is based on scope.

To understand how this works, it's necessary to introduce the concept of [execution context](#).

In JavaScript, there are two types of execution contexts:

- Global context, created when a JavaScript process is launched
- Local context, created when a function is invoked

Execution contexts are organized into a stack. At the bottom of the stack, there is always the global context, that is unique for each JavaScript program. Each time a function is encountered, a new execution context is created and pushed onto the top of the stack. Once the function has finished executing, its context is popped off the stack.

Consider the following code:

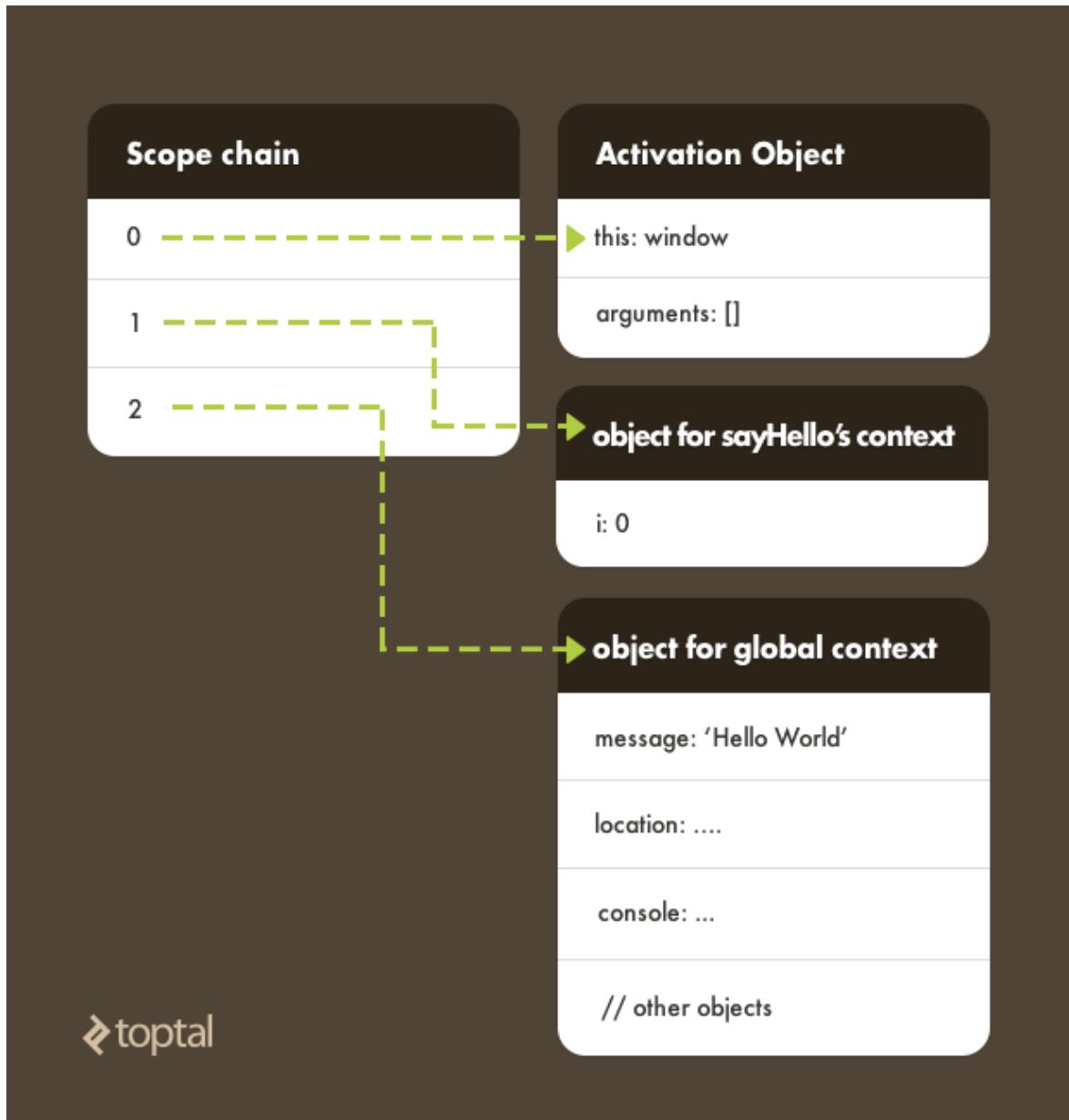
```
// global context
var message = 'Hello World';

var sayHello = function(n){
    // local context 1 created and pushed onto context stack
    var i = 0;
    var innerSayHello = function() {
        // local context 2 created and pushed onto context stack
        console.log((i + 1) + ': ' + message);
        // local context 2 popped off of context stack
    }
    for (i = 0; i < n; i++) {
        innerSayHello();
    }
    // local context 1 popped off of context stack
};

sayHello(3);
// Prints:
// 1: Hello World
// 2: Hello World
// 3: Hello World
```

CODE

Within each execution context is a special object called a *scope chain* which is used to resolve variables. A scope chain is essentially a stack of currently accessible scopes, from the most immediate context to the global context. (To be a bit more precise, the object at the top of the stack is called an *Activation Object* which contains references to the local variables for the function being executed, the named function arguments, and two "special" objects: `this` and `arguments`.) For example:



Note in the above diagram how `this` points to the `window` object by default and also how the global context contains examples of other objects such as `console` and `location`.

When attempting to resolve variables via the scope chain, the immediate context is first checked for a matching variable. If no match is found, the next context object in the scope chain is checked, and so on, until a match is found. If no match is found, a `ReferenceError` is thrown.

It is important to also note that a new scope is added to the scope chain when a `try-catch` block or a `with` block is encountered. In either of these cases, a new object is created and placed at top of the scope chain:

CODE

```

function Person(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
}

function persist(person) {
    with (person) {
        // The 'person' object was pushed onto the scope chain when we
        // entered this "with" block, so we can simply reference
        // 'firstName' and 'lastName', rather than person.firstName and
        // person.lastName
        if (!firstName) {
            throw new Error('FirstName is mandatory');
        }
        if (!lastName) {
            throw new Error('LastName is mandatory');
        }
    }
    try {
        person.save();
    } catch(error) {
        // A new scope containing the 'error' object is accessible here
        console.log('Impossible to store ' + person + ', Reason: ' + error);
    }
}

var p1 = new Person('John', 'Doe');
persist(p1);

```

To fully understand how scope-based variable lookups occur, it is important to keep in mind that in JavaScript there are currently no block-level scopes. For example:

CODE

```

for (var i = 0; i < 10; i++) {
    /* ... */
}
// 'i' is still in scope!
console.log(i); // prints '10'

```

In most other languages, the code above would lead to an error because the "life" (i.e., scope) of the variable `i` would be restricted to the `for` block. In JavaScript, though, this is not the case. Rather, `i` is added to the activation object at the top of the scope chain and it will stay there until that object is removed from the scope, which happens when the corresponding execution context is removed from the stack. This behavior is known as variable hoisting.

It is worth noting, though, that support for block-level scopes is making its way into JavaScript through the new [let](#) keyword. The `let` keyword is already available in JavaScript 1.7 and is slated to become an officially supported JavaScript keyword as of ECMAScript 6.

JavaScript Performance Ramifications

The way that property and variable lookups, using prototype chain and scope chain respectively, work in JavaScript is one of the language's key features, yet it is one of the trickiest and most subtle to understand.

The lookup operations we've described in this example, whether based on the prototype chain or the scope chain,

are repeated every time a property or variable is accessed. When this lookup occurs within loops or other intensive operations, it can have significant JavaScript performance ramifications, especially in light of the single-threaded nature of the language which prevents multiple operations from happening concurrently.

Consider the following example:

CODE

```
var start = new Date().getTime();
function Parent() { this.delta = 10; }

function ChildA(){}
ChildA.prototype = new Parent();
function ChildB(){}
ChildB.prototype = new ChildA();
function ChildC(){}
ChildC.prototype = new ChildB();
function ChildD(){}
ChildD.prototype = new ChildC();
function ChildE(){}
ChildE.prototype = new ChildD();

function nestedFn() {
    var child = new ChildE();
    var counter = 0;
    for(var i = 0; i < 1000; i++) {
        for(var j = 0; j < 1000; j++) {
            for(var k = 0; k < 1000; k++) {
                counter += child.delta;
            }
        }
    }
    console.log('Final result: ' + counter);
}

nestedFn();
var end = new Date().getTime();
var diff = end - start;
console.log('Total time: ' + diff + ' milliseconds');
```

In this example, we have a long inheritance tree and three nested loops. Inside the deepest loop, the counter variable is incremented with the value of `delta`. But `delta` is located almost at the top of the inheritance tree! *This means that each time `child.delta` is accessed, the full tree needs to be navigated from bottom to top.* This can have a really negative impact on performance.

Understanding this, we can easily improve performance of the above `nestedFn` function by using a local `delta` variable to cache the value in `child.delta` (and thereby avoid the need for repetitive traversal of the entire inheritance tree) as follows:

CODE

```

function nestedFn() {
  var child = new ChildE();
  var counter = 0;
  var delta = child.delta; // cache child.delta value in current scope
  for(var i = 0; i < 1000; i++) {
    for(var j = 0; j < 1000; j++) {
      for(var k = 0; k < 1000; k++) {
        counter += delta; // no inheritance tree traversal needed!
      }
    }
  }
  console.log('Final result: ' + counter);
}

nestedFn();
var end = new Date().getTime();
var diff = end - start;
console.log('Total time: ' + diff + ' milliseconds');

```

Of course, this particular technique is only viable in a scenario where it is known that the value of `child.delta` won't change while the for loops are executing; otherwise, the local copy would need to be updated with the current value.

OK, let's run both versions of the `nestedFn` method and see if there is any appreciable performance difference between the two.

We'll start by running the first example in a [node.js REPL](#):

CODE

```

diego@alkadia:~$ node test.js
Final result: 10000000000
Total time: 8270 milliseconds

```

So that takes about 8 seconds to run. That's a long time.

Now let's see what happens when we run the optimized version:

CODE

```

diego@alkadia:~$ node test2.js
Final result: 10000000000
Total time: 1143 milliseconds

```

This time it took just one second. Much faster!

Note that use of local variables to avoid expensive lookups is a technique that can be applied both for property lookup (via the prototype chain) and for variable lookups (via the scope chain).

Moreover, this type of "caching" of values (i.e., in variables in the local scope) can also be beneficial when using some of the most common JavaScript libraries. Take [jQuery](#), for example. jQuery supports the notion of "selectors", which are basically a mechanism for retrieving one or more matching elements in the [DOM](#). The ease with which one can specify selectors in jQuery can cause one to forget how costly (from a performance standpoint) each selector lookup can be. Accordingly, storing selector lookup results in a local variable can be extremely beneficial to performance. For example:

```
// this does the DOM search for $('.container') "n" times
for (var i = 0; i < n; i++) {
    $('.container').append("Line "+i+"  
");
}

// this accomplishes the same thing...
// but only does the DOM search for $('.container') once,
// although it does still modify the DOM "n" times
var $container = $('.container');
for (var i = 0; i < n; i++) {
    $container.append("Line "+i+"  
");
}

// or even better yet...
// this version only does the DOM search for $('.container') once
// AND only modifies the DOM once
var $html = '';
for (var i = 0; i < n; i++) {
    $html += 'Line ' + i + '<br />';
}
$('.container').append($html);
```

Especially on a web page with a large number of elements, the second approach in the code sample above can potentially result in significantly better performance than the first.

Wrap-up

Data lookup in JavaScript is quite different than it is in most other languages, and it is highly nuanced. It is therefore essential to fully and properly understand these concepts in order to truly master the language. Data lookup and [other common JavaScript mistakes](#) should be avoided whenever possible. This understanding is likely to yield cleaner, more robust code that achieves improved JavaScript performance.

76. JavaScript Object Creation: Patterns and Best Practices

JavaScript object creation is a tricky subject. The language has a multitude of styles for creating objects, and newcomers and veterans alike can feel overwhelmed by the choices and unsure which they should use. But despite the variety and how different the syntax for each may look, they're more similar than you probably realize. In this article, I'm going to take you on a tour of the various styles of object creation and how each builds on the others in incremental steps.

Object Literals

The first stop on our tour is the absolute simplest way to create objects, the object literal. JavaScript touts that objects can be created “ex nihilo”, out of nothing--no class, no template, no prototype--just poof, an object with methods and data.

CODE

```
var o = {
  x: 42,
  y: 3.14,
  f: function() {},
  g: function() {}
};
```

But there's a drawback. If we need to create the same type of object in other places, then we'll end up copy-pasting the object's methods, data, and initialization. We need a way to create not just the one object, but a family of objects.

Factory Functions

The next stop on our tour is the [factory function](#). This is the absolute simplest way to create a family of objects that share the same structure, interface, and implementation. Rather than creating an object literal directly, instead we return an object literal from a function. This way, if we need to create the same type of object multiple times or in multiple places, we only need to invoke a function.

CODE

```
function thing() {
  return {
    x: 42,
    y: 3.14,
    f: function() {},
    g: function() {}
  };
}

var o = thing();
```

But there's a drawback. This approach can cause memory bloat because each object contains its own unique copy of each function. Ideally we want every object to *share* just one copy of its functions.

Prototype Chains

JavaScript gives us a built-in mechanism to share data across objects, called the prototype chain. When we access a property on an object, it can fulfill that request by delegating to some other object. We can use that and change our factory function so that each object it creates contains only the data unique to that particular object, and delegate all other property requests to a single, shared object.

CODE

```
var thingPrototype = {  
  f: function() {},  
  g: function() {}  
};  
  
function thing() {  
  var o = Object.create(thingPrototype);  
  
  o.x = 42;  
  o.y = 3.14;  
  
  return o;  
}  
  
var o = thing();
```

In fact, this is such a common pattern that the language has built-in support for it. We don't need to create our own shared object (the prototype object). Instead, a prototype object is created for us automatically alongside every function, and we can put our shared data there.

CODE

```
thing.prototype.f = function() {};  
thing.prototype.g = function() {};  
  
function thing() {  
  var o = Object.create(thing.prototype);  
  
  o.x = 42;  
  o.y = 3.14;  
  
  return o;  
}  
  
var o = thing();
```

But there's a drawback. This is going to result in some repetition. The first and last lines of the "thing" function are going to be repeated almost verbatim in every such delegating-to-prototype-factory-function.

ES5 Classes

We can isolate the repetitive lines by moving them into their own function. This function would create an object that delegates to some other arbitrary function's prototype, then invoke that function with the newly created object as an argument, and finally return the object.

CODE

```
function create(fn) {
  var o = Object.create(fn.prototype);

  fn.call(o);

  return o;
}

// ...

Thing.prototype.f = function() {};
Thing.prototype.g = function() {};

function Thing() {
  this.x = 42;
  this.y = 3.14;
}

var o = create(Thing);
```

In fact, this too is such a common pattern that the language has some built-in support for it. The “create” function we defined is actually a rudimentary version of the “new” keyword, and we can drop-in replace “create” with “new”.

CODE

```
Thing.prototype.f = function() {};
Thing.prototype.g = function() {};

function Thing() {
  this.x = 42;
  this.y = 3.14;
}

var o = new Thing();
```

We've now arrived at what we commonly call ES5 classes. They are object creation functions that delegate shared data to a prototype object and rely on the “new” keyword to handle repetitive logic.

But there's a drawback. It's verbose and ugly, and implementing inheritance is even more verbose and ugly.

ES6 Classes

A relatively recent addition to JavaScript is [ES6 classes](#), which offer a significantly cleaner syntax for doing the same thing.

```
class Thing {  
    constructor() {  
        this.x = 42;  
        this.y = 3.14;  
    }  
  
    f() {}  
    g() {}  
}  
  
var o = new Thing();
```

Comparison

Over the years, we JavaScripters have had an on-and-off relationship with the prototype chain, and today the two most common styles you're likely to encounter are the class syntax, which relies heavily on the prototype chain, and the factory function syntax, which typically doesn't rely on the prototype chain at all. The two styles differ--but only slightly--in performance and features.

Performance

JavaScript engines are so heavily optimized today that it's nearly impossible to look at our code and reason about what will be faster. Measurement is crucial. Yet sometimes even measurement can fail us. Typically an updated JavaScript engine is released every six weeks, sometimes with significant changes in performance, and any measurements we had previously taken, and any decisions we made based on those measurements, go right out the window. So, my rule of thumb has been to favor the most official and most widely used syntax, under the presumption that it will receive the most scrutiny and be the most performant *most of the time*. Right now that's the class syntax, and as I write this, the class syntax is roughly 3x faster than a factory function returning a literal.

Features

What few feature differences there were between classes and factory functions evaporated with ES6. Today, both factory functions and classes can enforce truly private data--factory functions with [closures](#) and classes with [weak maps](#). Both can achieve multiple inheritance--factory functions by mixing other properties into its own object, and classes also by mixing other properties into its prototype, or with class factories, or with proxies. Both factory functions and classes can return any arbitrary object if need be. And both offer a simple syntax.

Conclusion

All things considered, my preference is for the class syntax. It's standard, it's simple and clean, it's fast, and it provides every feature that once upon a time only factories could deliver.

77. Master Closures by Reimplementing Them from Scratch

This article was peer reviewed by [Tim Severien](#) and [Michaela Lehr](#). Thanks to all of SitePoint's peer reviewers for making SitePoint content the best it can be!

To say there are a lot of articles about closures would be an understatement. Most will explain the definition of a closure, which usually boils down to a simple sentence: A closure is a function that remembers the environment in which it was created. But *how* does it remember? And why can a closure use local variables long after those variables have gone out of scope? To lift the veil of magic surrounding closures, I'm going to pretend that JavaScript *doesn't* have closures and *can't* nest functions, and then we're going to re-implement closures from scratch. In doing so, we'll discover what closures really are and how they work under the hood.

For this exercise, I'll also need to pretend that JavaScript has one feature it doesn't really have. I'll need to pretend that an ordinary object can be called as if it were a function. You may have already seen this feature in other languages. Python lets you define a `__call__` method, and PHP has a special `__invoke` method, and it's these methods that are executed when an object is called as if it were a function. If we pretend that JavaScript has this feature too, here's how that might look:

```
// An otherwise ordinary object with a "__call__" method
let o = {
  n: 42,
  __call__() {
    return this.n;
  }
};

// Call object as if it were a function
o(); // 42
```

CODE

Here we have an ordinary object that we're pretending we can call as if it were a function, and when we do, the special `__call__` method is executed, same as if we had written `o.__call__()`.

With that, let's now look at a simple closure example.

```
function f() {
  // This variable is local to "f"
  // Normally it would be destroyed when we leave "f"'s scope
  let n = 42;

  // An inner function that references "n"
  function g() {
    return n;
  }

  return g;
}

// Get the "g" function created by "f"
let g = f();

// The variable "n" should be destroyed by now, right?
// After all, "f" is done executing and we've left its scope
// So how can "g" still reference a freed variable?
g(); // 42
```

CODE

Here we have an outer function `f` with a local variable, and an inner function `g` that references `f`'s local variable. Then we return the inner function `g` and execute it from outside `f`'s scope. But if `f` is done executing, then how can `g` still use variables that have gone out of scope?

Here's the magic trick: A closure isn't merely a function. It's an *object*, with a constructor and private data, that we can call as *if* it were a function. If JavaScript didn't have closures and we had to implement them ourselves, here's how that would look.

CODE

```
class G {  
    // An instance of "G" will be constructed with a value "n",  
    // and it stores that value in its private data  
    constructor(n) {  
        this._n = n;  
    }  
  
    // When we call an instance of "G", it returns the value from its private data  
    __call__() {  
        return this._n;  
    }  
}  
  
function f() {  
    let n = 42;  
  
    // This is the closure  
    // Our inner function isn't really a function  
    // It's a callable object, and we pass "n" to its constructor  
    let g = new G(n);  
  
    return g;  
}  
  
// Get the "g" callable object created by "f"  
let g = f();  
  
// It's okay if the original variable "n" from "f"'s scope is destroyed now  
// The callable object "g" is actually referencing its own private data  
g(); // 42
```

Here we replaced the inner function `g` with an instance of the class `G`, and we captured `f`'s local variable by passing it to `G`'s constructor, which then stores that value in the new instance's private data. And that, ladies and gentlemen, is a closure. It really is that simple. A closure is a callable object that privately stores values passed through the constructor from the environment in which it was instantiated.

Taking It Further

The astute reader will notice there's some behavior we haven't yet accounted for. Let's look at another closure example.

CODE

```
function f() {
  let n = 42;

  // An inner function that references "n"
  function get() {
    return n;
  }

  // Another inner function that also references "n"
  function next() {
    n++;
  }

  return {get, next};
}

let o = f();

o.get(); // 42
o.next();
o.get(); // 43
```

In this example, we have two closures that both reference the same variable `n`. One function's manipulation of that variable affects the other function's value. But if JavaScript didn't have closures and we had to implement them ourselves, then we wouldn't get that same behavior.

```
class Get {
  constructor(n) {
    this._n = n;
  }

  __call__() {
    return this._n;
  }
}

class Next {
  constructor(n) {
    this._n = n;
  }

  __call__() {
    this._n++;
  }
}

function f() {
  let n = 42;

  // These are the closures
  // They're callable objects that privately store the values
  // passed through their constructors
  let get = new Get(n);
  let next = new Next(n);

  return {get, next};
}

let o = f();

o.get(); // 42
o.next();
o.get(); // 42
```

Like before, we replaced the inner functions `get` and `next` with instances of the classes `Get` and `Next`, and they capture `f`'s local variable by passing it to the constructors and storing that value in each instance's private data. But notice that one callable object's manipulation of `n` *didn't* affect the other callable object's value. This happened because they didn't capture a *reference* to `n`; they captured a *copy* of the value of `n`.

To explain why JavaScript's closures will reference the same `n`, we need to explain variables themselves. Under the hood, JavaScript's local variables aren't really local in the traditional sense. Instead, they're properties of a dynamically allocated and reference counted object, called a "LexicalEnvironment" object, and JavaScript's closures capture a reference to that whole environment rather than to any one particular variable.

Let's change our callable object implementation to capture a lexical environment rather than `n` specifically.

```

class Get {
  constructor(lexicalEnvironment) {
    this._lexicalEnvironment = lexicalEnvironment;
  }

  __call__() {
    return this._lexicalEnvironment.n;
  }
}

class Next {
  constructor(lexicalEnvironment) {
    this._lexicalEnvironment = lexicalEnvironment;
  }

  __call__() {
    this._lexicalEnvironment.n++;
  }
}

function f() {
  let lexicalEnvironment = {
    n: 42
  };

  // These callable objects capture a reference to the lexical environment,
  // so they will share a reference to the same "n"
  let get = new Get(lexicalEnvironment);
  let next = new Next(lexicalEnvironment);

  return {get, next};
}

let o = f();

// Now our callable objects exhibit the same behavior as JavaScript's functions
o.get(); // 42
o.next();
o.get(); // 43

```

Here we replaced the local variable `n` with a `lexicalEnvironment` object that has a property `n`. And the closures--the callable instances of classes `Get` and `Next`--capture a reference to the lexical environment object rather than the value of `n`. And because they now share a reference to the same `n`, one callable object's manipulation of `n` affects the other callable object's value.

Conclusion

Closures are objects that we can call as if they were functions. Every function in JavaScript is in fact a callable object, also called a “function object” or “functor”, that is instantiated with and privately stores a lexical environment object, even if it’s the outermost global lexical environment. In JavaScript, a function doesn’t create a closure; the function *is* the closure.

Has this post helped you to understand closures? I’d be glad to hear your thoughts or questions in the comments below.

78. Object-Oriented JavaScript -- A Deep Dive into ES6 Classes

Often we need to represent an idea or concept in our programs--maybe a car engine, a computer file, a router, or a temperature reading. Representing these concepts directly in code comes in two parts: data to represent the state and functions to represent the behavior. Classes give us a convenient syntax to define the state and behavior of objects that will represent our concepts. They make our code safer by guaranteeing an initialization function will be called, and they make it easier to define a fixed set of functions that operate on that data and maintain valid state. If you can think of something as a separate entity, it's likely you should define a class to represent that "thing" in your program.

Consider this non-class code. How many errors can you find? How would you fix them?

CODE

```
// set today to December 24
let today = {
  month: 12,
  day: 24,
};

let tomorrow = {
  year: today.year,
  month: today.month,
  day: today.day + 1,
};

let dayAfterTomorrow = {
  year: tomorrow.year,
  month: tomorrow.month,
  day: tomorrow.day + 1 <= 31 ? tomorrow.day + 1 : 1,
};
```

The date `today` isn't valid; there is no month 24. Also, `today` isn't fully initialized; it's missing the year. It would be better if we had an initialization function that couldn't be forgotten. Notice also, that when adding a day, we checked in one place if we went beyond 31 but missed that check in another place. It would be better if we interacted with the data only through a small and fixed set of functions that each maintain valid state.

Here's the corrected version that uses classes.

```
class SimpleDate {  
    constructor(year, month, day) {  
        // Check that (year, month, day) is a valid date  
        // ...  
  
        // If it is, use it to initialize "this" date  
        this._year = year;  
        this._month = month;  
        this._day = day;  
    }  
  
    addDays(nDays) {  
        // Increase "this" date by n days  
        // ...  
    }  
  
    getDay() {  
        return this._day;  
    }  
}  
  
// "today" is guaranteed to be valid and fully initialized  
let today = new SimpleDate(2000, 2, 28);  
  
// Manipulating data only through a fixed set of functions ensures we maintain valid state  
today.addDays(1);
```

JARGON TIP:

- When a function is associated with a class or object, we call it a “method”.
- When an object is created from a class, that object is said to be an “instance” of the class.

Constructors

The `constructor` method is special, and it solves the first problem. Its job is to initialize an instance to a valid state, and it will be called automatically so we can't forget to initialize our objects.

Keep Data Private

We try to design our classes so that their state is guaranteed to be valid. We provide a constructor that creates only valid values, and we design methods that also always leave behind only valid values. But as long as we leave the data of our classes accessible to everyone, someone *will* mess it up. We protect against this by keeping the data inaccessible except through the functions we supply.

JARGON TIP: Keeping data private to protect it is called “encapsulation”.

Privacy with Conventions

Unfortunately, private object properties don't exist in JavaScript. We have to fake them. The most common way to do that is to adhere to a simple convention: If a property name is prefixed with an underscore (or, less commonly, suffixed with an underscore), then it should be treated as non-public. We used this approach in the earlier code example. Generally this simple convention works, but the data is technically still accessible to everyone, so we have

to rely on our own discipline to do the right thing.

Privacy with Privileged Methods

The next most common way to fake private object properties is to use ordinary variables in the constructor, and capture them in closures. This trick gives us truly private data that is inaccessible to the outside. But to make it work, our class's methods would themselves need to be defined in the constructor and attached to the instance.

```
class SimpleDate {  
    constructor(year, month, day) {  
        // Check that (year, month, day) is a valid date  
        // ...  
  
        // If it is, use it to initialize "this" date's ordinary variables  
        let _year = year;  
        let _month = month;  
        let _day = day;  
  
        // Methods defined in the constructor capture variables in a closure  
        this.addDays = function(nDays) {  
            // Increase "this" date by n days  
            // ...  
        }  
  
        this.getDay = function() {  
            return _day;  
        }  
    }  
}
```

CODE

Privacy with Symbols

Symbols are a new feature to JavaScript, and they give us another way to fake private object properties. Instead of underscore property names, we could use unique symbol object keys, and our class can capture those keys in a closure. But there's a leak. Another new feature to JavaScript is `Object.getOwnPropertySymbols`, and it allows the outside to access the symbol keys we tried to keep private.

CODE

```
let SimpleDate = (function() {
  let _yearKey = Symbol();
  let _monthKey = Symbol();
  let _dayKey = Symbol();

  class SimpleDate {
    constructor(year, month, day) {
      // Check that (year, month, day) is a valid date
      // ...

      // If it is, use it to initialize "this" date
      this[_yearKey] = year;
      this[_monthKey] = month;
      this[_dayKey] = day;
    }

    addDays(nDays) {
      // Increase "this" date by n days
      // ...
    }

    getDay() {
      return this[_dayKey];
    }
  }

  return SimpleDate;
}());
```

Privacy with Weak Maps

[Weak maps](#) are also a new feature to JavaScript. We can store private object properties in key/value pairs using our instance as the key, and our class can capture those key/value maps in a closure.

CODE

```

let SimpleDate = (function() {
  let _years = new WeakMap();
  let _months = new WeakMap();
  let _days = new WeakMap();

  class SimpleDate {
    constructor(year, month, day) {
      // Check that (year, month, day) is a valid date
      // ...

      // If it is, use it to initialize "this" date
      _years.set(this, year);
      _months.set(this, month);
      _days.set(this, day);
    }

    addDays(nDays) {
      // Increase "this" date by n days
      // ...
    }

    getDay() {
      return _days.get(this);
    }
  }

  return SimpleDate;
}());

```

Other Access Modifiers

There are other levels of visibility besides “private” that you’ll find in other languages, such as “protected”, “internal”, “package private”, or “friend”. JavaScript still doesn’t give us a way to enforce those other levels of visibility. If you need them, you’ll have to rely on conventions and self discipline.

Referring to the Current Object

Look again at `getDay()`. It doesn’t specify any parameters, so how does it know the object for which it was called? When a function is called as a method using the `object.function` notation, there is an implicit argument that it uses to identify the object, and that implicit argument is assigned to an implicit parameter named `this`. To illustrate, here’s how we would send the object argument explicitly rather than implicitly.

CODE

```

// Get a reference to the "getDay" function
let getDay = SimpleDate.prototype.getDay;

getDay.call(today); // "this" will be "today"
getDay.call(tomorrow); // "this" will be "tomorrow"

tomorrow.getDay(); // same as last line, but "tomorrow" is passed implicitly

```

Static Properties and Methods

We have the option to define data and functions that are part of the class but not part of any instance of that class. We call these static properties and static methods, respectively. There will only be one copy of a static property rather than a new copy per instance.

CODE

```
class SimpleDate {  
    static setDateDefaultDate(year, month, day) {  
        // A static property can be referred to without mentioning an instance  
        // Instead, it's defined on the class  
        SimpleDate._defaultDate = new SimpleDate(year, month, day);  
    }  
  
    constructor(year, month, day) {  
        // If constructing without arguments,  
        // then initialize "this" date by copying the static default date  
        if (arguments.length === 0) {  
            this._year = SimpleDate._defaultDate._year;  
            this._month = SimpleDate._defaultDate._month;  
            this._day = SimpleDate._defaultDate._day;  
  
            return;  
        }  
  
        // Check that (year, month, day) is a valid date  
        // ...  
  
        // If it is, use it to initialize "this" date  
        this._year = year;  
        this._month = month;  
        this._day = day;  
    }  
  
    addDays(nDays) {  
        // Increase "this" date by n days  
        // ...  
    }  
  
    getDay() {  
        return this._day;  
    }  
}  
  
SimpleDate.setDateDefaultDate(1970, 1, 1);  
  
let defaultDate = new SimpleDate();
```

Subclasses

Often we find commonality between our classes--repeated code that we'd like to consolidate. Subclasses let us incorporate another class's state and behavior into our own. This process is often called "inheritance," and our subclass is said to "inherit" from a parent class, also called a superclass. Inheritance can avoid duplication and simplify the implementation of a class that needs the same data and functions as another class. Inheritance also allows us to substitute subclasses, relying only on the [interface](#) provided by a common superclass.

Inherit to Avoid Duplication

Consider this non-inheritance code.

CODE

```
class Employee {
  constructor(firstName, familyName) {
    this._firstName = firstName;
    this._familyName = familyName;
  }

  getFullName() {
    return `${this._firstName} ${this._familyName}`;
  }
}

class Manager {
  constructor(firstName, familyName) {
    this._firstName = firstName;
    this._familyName = familyName;
    this._managedEmployees = [];
  }

  getFullName() {
    return `${this._firstName} ${this._familyName}`;
  }

  addEmployee(employee) {
    this._managedEmployees.push(employee);
  }
}
```

The data properties `_firstName` and `_familyName`, and the method `getFullName`, are repeated between our classes. We could eliminate that repetition by having our `Manager` class inherit from the `Employee` class. When we do, the state and behavior of the `Employee` class--its data and functions--will be incorporated into our `Manager` class.

Here's a version that uses inheritance. Notice the use of [super](#).

CODE

```
// Manager still works same as before but without repeated code
class Manager extends Employee {
  constructor(firstName, familyName) {
    super(firstName, familyName);
    this._managedEmployees = [];
  }

  addEmployee(employee) {
    this._managedEmployees.push(employee);
  }
}
```

IS-A and WORKS-LIKE-A

There are design principles to help you decide when inheritance is appropriate. Inheritance should always model an IS-A and WORKS-LIKE-A relationship. That is, a manager “is a” and “works like a” specific kind of employee, such that anywhere we operate on a superclass instance, we should be able to substitute in a subclass instance, and

everything should still just work. The difference between violating and adhering to this principle can sometimes be subtle. A classic example of a subtle violation is a `Rectangle` superclass and a `Square` subclass.

CODE

```

class Rectangle {
    set width(w) {
        this._width = w;
    }

    get width() {
        return this._width;
    }

    set height(h) {
        this._height = h;
    }

    get height() {
        return this._height;
    }
}

// A function that operates on an instance of Rectangle
function f(rectangle) {
    rectangle.width = 5;
    rectangle.height = 4;

    // Verify expected result
    if (rectangle.width * rectangle.height !== 20) {
        throw new Error("Expected the rectangle's area (width * height) to be 20");
    }
}

// A square IS-A rectangle... right?
class Square extends Rectangle {
    set width(w) {
        super.width = w;

        // Maintain square-ness
        super.height = w;
    }

    set height(h) {
        super.height = h;

        // Maintain square-ness
        super.width = h;
    }
}

// But can a rectangle be substituted by a square?
f(new Square()); // error

```

A square may be a rectangle *mathematically*, but a square doesn't *work like* a rectangle behaviorally.

This rule that any use of a superclass instance should be substitutable by a subclass instance is called the [Liskov Substitution Principle](#), and it's an important part of object oriented class design.

Beware Overuse

It's easy to find commonality everywhere, and the prospect of having a class that offers complete functionality can be alluring, even for experienced developers. But there are disadvantages to inheritance too. Recall that we ensure valid state by manipulating data only through a small and fixed set of functions. But when we inherit, we increase the list of functions that can directly manipulate the data, and those additional functions are then also responsible for maintaining valid state. If too many functions can directly manipulate the data, then that data becomes nearly as bad as global variables. Too much inheritance creates monolithic classes that dilute encapsulation, are harder to make correct, and harder to reuse. Instead, prefer to design minimal classes that embody just one concept.

Let's revisit the code duplication problem. Could we solve it without inheritance? An alternative approach is to connect objects through references to represent a part-whole relationship. We call this "composition".

Here's a version of the manager-employee relationship using composition rather than inheritance.

CODE

```
class Employee {
  constructor(firstName, familyName) {
    this._firstName = firstName;
    this._familyName = familyName;
  }

  getFullName() {
    return `${this._firstName} ${this._familyName}`;
  }
}

class Group {
  constructor(manager /* : Employee */ ) {
    this._manager = manager;
    this._managedEmployees = [];
  }

  addEmployee(employee) {
    this._managedEmployees.push(employee);
  }
}
```

Here, a manager isn't a separate class. Instead, a manager is an ordinary `Employee` instance that a `Group` instance holds a reference to. If inheritance models the IS-A relationship, then composition models the HAS-A relationship. That is, a group "has a" manager.

If either inheritance or composition can reasonably express our program concepts and relationships, then prefer composition.

Inherit to Substitute Subclasses

Inheritance also allows different subclasses to be used interchangeably through the interface provided by a common superclass. A function that expects a superclass instance as an argument can also be passed a subclass instance without the function having to know about any of the subclasses. Substituting classes that have a common superclass is often called "polymorphism."

CODE

```
// This will be our common superclass
class Cache {
    get(key, defaultValue) {
        let value = this._doGet(key);
        if (value === undefined || value === null) {
            return defaultValue;
        }

        return value;
    }

    set(key, value) {
        if (key === undefined || key === null) {
            throw new Error('Invalid argument');
        }

        this._doSet(key, value);
    }

    // Must be overridden
    // _doGet()
    // _doSet()
}

// Subclasses define no new public methods
// The public interface is defined entirely in the superclass
class ArrayCache extends Cache {
    _doGet() {
        // ...
    }

    _doSet() {
        // ...
    }
}

class LocalStorageCache extends Cache {
    _doGet() {
        // ...
    }

    _doSet() {
        // ...
    }
}

// Functions can polymorphically operate on any cache by interacting through the superclass interface
function compute(cache) {
    let cached = cache.get('result');
    if (!cached) {
        let result = // ...
        cache.set('result', result);
    }

    // ...
}

compute(new ArrayCache()); // use array cache through superclass interface
compute(new LocalStorageCache()); // use local storage cache through superclass interface
```

More than Sugar

JavaScript's class syntax is often said to be syntactic sugar, and in a lot of ways it is, but there are also real differences--things we can do with ES6 classes that we couldn't with ES5.

Static Properties Are Inherited

ES5 didn't let us create true inheritance between constructor functions. `Object.create` could create an ordinary object but not a function object. We faked inheritance of static properties by manually copying them. Now with ES6 classes, we get a real prototype link between a subclass constructor function and the superclass constructor.

```
// ES5
function B() {}
B.f = function () {};

function D() {}
D.prototype = Object.create(B.prototype);

D.f(); // error
```

CODE

```
// ES6
class B {
  static f() {}
}

class D extends B {}

D.f(); // ok
```

CODE

Built-in Constructors Can Be Subclassed

Some objects are “exotic” and don't behave like ordinary objects. Arrays, for example, adjust their `length` property to be greater than the largest integer index. In ES5, when we tried to subclass `Array`, the `new` operator would allocate an ordinary object for our subclass, not the exotic object of our superclass.

```
// ES5
function D() {
  Array.apply(this, arguments);
}
D.prototype = Object.create(Array.prototype);

var d = new D();
d[0] = 42;

d.length; // 0 - bad, no array exotic behavior
```

CODE

ES6 classes fixed this by changing when and by whom objects are allocated. In ES5, objects were allocated before invoking the subclass constructor, and the subclass would pass that object to the superclass constructor. Now with ES6 classes, objects are allocated before invoking the *superclass* constructor, and the superclass makes that object available to the subclass constructor. This lets `Array` allocate an exotic object even when we invoke `new` on our subclass.

```
// ES6
class D extends Array {}

let d = new D();
d[0] = 42;

d.length; // 1 - good, array exotic behavior
```

CODE

Miscellaneous

There's a small assortment of other, probably less significant differences. Class constructors can't be function-called. This protects against forgetting to invoke constructors with `new`. Also, a class constructor's `prototype` property can't be reassigned. This may help JavaScript engines optimize class objects. And finally, class methods don't have a `prototype` property. This may save memory by eliminating unnecessary objects.

Using New Features in Imaginative Ways

Many of the features described here and in other SitePoint articles are new to JavaScript, and the community is experimenting right now to use those features in new and imaginative ways.

Multiple Inheritance with Proxies

One such experiment uses [proxies](#), a new feature to JavaScript, to implement multiple inheritance. JavaScript's prototype chain allows only single inheritance. Objects can delegate to only one other object. Proxies give us a way to delegate property accesses to multiple other objects.

CODE

```

let transmitter = {
  transmit() {}
};

let receiver = {
  receive() {}
};

// Create a proxy object that intercepts property accesses and forwards to each parent,
// returning the first defined value it finds
let inheritsFromMultiple = new Proxy([transmitter, receiver], {
  get: function(proxyTarget, propertyKey) {
    const foundParent = proxyTarget.find(parent => parent[propertyKey] !== undefined);
    return foundParent && foundParent[propertyKey];
  }
});

inheritsFromMultiple.transmit(); // works
inheritsFromMultiple.receive(); // works

```

Can we expand this to work with classes? A class's `prototype` could be a proxy that forwards property access to multiple other prototypes. The JavaScript community is working on this right now. Can you figure it out? Join the discussion and share your ideas.

Multiple Inheritance with Class Factories

Another approach the JavaScript community has been experimenting with is generating classes on demand that extend a variable superclass. Each class still has only a single parent, but we can chain those parents in interesting ways.

CODE

```

function makeTransmitterClass(Superclass = Object) {
  return class Transmitter extends Superclass {
    transmit() {}
  };
}

function makeReceiverClass(Superclass = Object) {
  return class Receiver extends Superclass {
    receive() {}
  };
}

class InheritsFromMultiple extends makeTransmitterClass(makeReceiverClass()) {}

let inheritsFromMultiple = new InheritsFromMultiple();

inheritsFromMultiple.transmit(); // works
inheritsFromMultiple.receive(); // works

```

Are there other imaginative ways to use these features? Now's the time to leave your footprint in the JavaScript world.

Conclusion

Hopefully this article has given you an insight into how classes work in ES6 and has demystified some of the jargon surrounding them. Unfortunately, at the time of writing, [support for classes isn't very good](#), so you'll need to use a transpiler such as Babel if you want to give them a try. Nonetheless, I'd be keen to hear your thoughts on classes and whether this is an ES6 feature you'll consider using in the comments below.

79. HTML5: Introduction to <canvas>

80. Canvas: Tutorial of basic canvas functionality, canvas properties and methods

The HTML5 Canvas specification is a JavaScript API for coding drawings. The canvas API allows the definition of a canvas context object as the <canvas> element on your HTML page inside which we can draw.

We can draw in both 2D and 3D (WebGL) context. 2D is available in all the modern Web browsers, IE9, and via excanvas.js in current versions of IE, and will be more thoroughly introduced below. 3D is still nascent, with only experimental implementations.

2D context provides a simple yet powerful API for performing quick drawing operation, on a 2D bitmap surface. There is no file format, and you can only draw using script. You do not have any DOM nodes for the shapes you draw -- you're drawing pixels, not vectors. OK, not true. You are drawing vectors, but once drawn, only the pixels are remembered.

Your first <canvas>

Being a very basic introduction to canvas, we are only going to cover basic shapes and lines. If you are unfamiliar with JavaScript, the syntax may at first seem a bit confusing. If you are familiar, it should make sense.

Step 1 is adding the <canvas> element to your document. In terms of HTML, the only step involved in adding a canvas to your document is adding the <canvas> element to your document:

```
<canvas id="flag" width="320" height="220">  
  You don't support Canvas. If you did, you would see a flag  
</canvas>
```

CODE

That is it for the HTML part of it. . We could simply have written <canvas></canvas>. However, you should include an id for ease of JavaScript targeting, but you could also target via placement within the DOM. You can also define the width and height of the canvas, though you can define that in the CSS as well. We've also included alternative content for users that don't support or otherwise can't see your <canvas> content

With that, we've created your blank drawing board, or canvas. Everything else takes place in our JavaScript files. Step 2 is drawing on our canvas. From now on, everything is in javascript. We target the canvas node with getElementById('flag') or getElementsByTagName('canvas')[0], initialize a 2D context and start drawing using 2D context API commands. We can draw the Japanese flag:

```
<script>

    var el= document.getElementById("flag");

    if (el && el.getContext) {

        var context = el.getContext('2d');

        if(context){

            context.fillStyle     = "#ffffff";
            context.strokeStyle = "#CCCCCC";
            context.lineWidth   = 1;
            context.shadowOffsetX = 5;
            context.shadowOffsetY = 5;
            context.shadowBlur    = 4;
            context.shadowColor   = 'rgba(0, 0, 0, 0.4)';
            context.strokeRect(10, 10, 300, 200);
            context.fillRect(10, 10, 300, 200);
            context.shadowColor='rgba(0,0,0,0)';
            context.fillStyle = "#d60818";
            context.arc(160, 107, 60, 0, Math.PI*2, false);
            context.fill();

        }

    }

</script>
```

The first line finds your `<canvas>` element by matching the element's id attribute. Before creating the 2D context, we check to make sure that the canvas element has been found AND that the browser supports canvas by checking for the existence of the `getContext` method.

We have to then create a reference to a context using the `getContext(contextId)` method of the canvas element –'2d' and '3d' are the `contextId` value choices. If context creation is successful, we are finally free to draw in our canvas.

Before drawing a shape, we must define the look and feel of the shape we want to draw by setting properties on the context object. We define the look of the border (stroke and linewidth) properties and the shadow of our first rectangle, which we draw with the `strokeRect()` method. We pass the same parameters as our SVG example: (10, 10, 300, 200). The four values are the x-offset, the y-offset, width and height respectively. Once the script executes a command, the script forgets about what it has done, and moves onto the next line of code. Unlike our SVG example,

the rectangle we've drawn on our canvas is not part of the DOM.

When we draw our second rectangle using the fillRect method, which paints rectangles using the previously set fillStyle property, we need to pass the coordinates again as the DOM does not remember our first rectangle, though it can access pixel information.

Both rectangle method calls have the same parameters -- 10, 10, 300, 200 -- we've drawn our fill rectangle directly on top of our dropshadow rectangle. We could have created an object with those coordinates and passed it to both methods, but we can't tell the canvas to access the first rectangle's coordinates and copy to the second after the method call

As mentioned above, once you paint onto the canvas, the DOM has no recollection of what you've painted. Yes, the JavaScript remembers the values of the properties you've set, but the pixels that are places on the canvas are just pixels of color. As we start the process of drawing the disc or sun on our flag, the DOM has no recollection of which pixels were painted with which colors, but it does remember some properties we set, like our shadowColor. As we don't want a shadow on the red circle, we can set the shadowColor to transparent.

Next we define our circle. We are not actually drawing the circle yet. context . arc(x-offset of center, y-offset of center, radius, startAngle, endAngle, anticlockwise) adds points to an arced path creating a virtual circumference of a circle described by the arguments, starting at the given start angle, in our case 0, which is on the right horizon, and ending at the given end angle, going in the given direction, which in our case is clockwise. Had our endAngle been less than 2?, our circle would have been flattened: the start and end points connected by a straight line. ? would have created a half circle. We also re-define the fill color, from white to red. We then paint the circle we created using the fill() method that fills the described arc in the fillStyle color.

We haven't even touched the surface of what <canvas> can do.

<http://ie.microsoft.com/testdrive/Graphics/CanvasPad/Default.html> is a fun page to learn simple shapes, colors, shadows, text, images, transformation, animation and mouse movement with <canvas>.

Canvas functions and properties

Styles

Set the fillStyle

```
context.fillStyle="color"
```

CODE

Set the strokeStyle

```
context.strokeStyle="color"
```

CODE

Line widths

```
context.lineWidth=number
```

CODE

Line join styles

```
context.lineJoin="bevel || round || miter"
```

CODE

Line end styles

```
context.lineCap="butt || round || square"
```

CODE

Rectangles

Draw a rectangle

```
context.strokeRect(left, top, width, height)
```

CODE

Fill a rectangle

```
context.fillRect(left, top, width, height)
```

CODE

Erase a rectangle

```
context.clearRect(left, top, width, height)
```

CODE

paths

Begin a path

```
context.beginPath
```

CODE

Complete a path

```
context.closePath
```

CODE

Move the pen to a location

```
context.moveTo(horizontal, vertical)
```

CODE

Draw a straight line from current point to a new location

```
context.lineTo(horizontal, vertical)
```

CODE

Stroke the current path

```
context.stroke()
```

CODE

Fill the current path

```
context.fill()
```

CODE

Shadows

Shadow color

```
context.shadowColor="color"
```

CODE

Shadow horizontal offset

```
context.shadowOffsetX=number
```

CODE

Shadow vertical offset

```
context.shadowOffsetY=number
```

CODE

Shadow blur

```
context.shadowBlur=number
```

CODE

Canvas versus SVG

HTML5 Canvas and SVG may seem similar, in that they are both web technologies that allow you to create rich graphics inside the browser, but they are fundamentally different. In SVG, you 'draw' with XML. For canvas, you draw with JavaScript. Canvas is the painting of pixels onto a canvas, once painted, each pixel is forgotten. SVG, on the other hand, creates DOM nodes, accessible until deleted or until navigation away from the page. They both have their advantages and disadvantages.

SVG is resolution independent, making SVG an excellent choice for user interfaces of all sizes as it allows scaling for all screen resolutions. SVG is an XML file format enabling easy accessibility. SVG can be animated using a declarative syntax, or via JavaScript. Each element becomes part of and is accessible via the SVG DOM API in JavaScript. However, anything that accesses the DOM repeatedly slows the page down.

Canvas is all drawn in pixels. Zooming can lead to pixilation. Canvas is inherently less accessible: accessibility is limited mainly to including fallback content should canvas not render. Interactivity requires redrawing of each pixel. There are no DOM nodes for anything you draw. There's no animation API, instead timers are generally used for updating the canvas at quick intervals. Canvas gives you a surface to draw onto with the API of the context you choose. Canvas, however, is very well suited for editing of images, generating raster graphics such as for games or

fractals, and operations requiring pixel-level manipulation. Canvas can also be exported to gif or jpeg.

81. JavaScript Variable Scope

There are two variable scopes in JavaScript: *local* and *global*. *Local variables* exist only inside the function in which they were declared. *Local scope* means a variable can only be referenced from within that function in which it was declared. *Global variables*, on the other hand, once declared, exist throughout the script and their values can be accessed (and changed) from within any function or outside of a function. In other words, *global scope* means a variable can be referenced from anywhere within your site's javascript.

There is a third type of scope called "static" or "closure". "Closures" are variables that are local to a function, but keep their values between function calls. Closures are an advanced javascript topic and will not be discussed in this article.

Let's show 5 variables being declared, and discuss their scope

CODE

```
1. <script type="text/javascript">
2.
3. var outsideFunction1 = "hello";
4. outsideFunction2 = 42;
5.
6. function myFunction(myParameter){
7.     var insideFunction1 = new Array();
8.     insideFunction2 = false;
9. }
10.
11. </script>
```

Listing 1: Example of local and global variable declaration

The first variable is on line 3: *outsideFunction1*. Even though this variable is declared with the *var* keyword, it is declared outside of any function, and therefore has global scope.

The second variable is on line 4: *outsideFunction2*. This variable is also declared outside of any function, and therefore has global scope.

The third variable is on line 6: *myParameter*. Function parameters always have local scope. Even if there were a global variable called *myParameter*, that global variable will maintain its value even if the value of *myParameter* was changed within the function.

The fourth variable is on line 7: *insideFunction1*. Because this variable is declared within a function with the *var* keyword, it also only has local scope. Similar to parameter variables, even if there were a global variable called *insideFunction1*, that global variable would maintain its value even if the value of *insideFunction1* were changed within the function.

The fifth variable is on line 8: *insideFunction2*. This is really the reason that this article is needed: this is a global variable declared within a function, which is one of the most common causes of logic errors. Because the *var* keyword has been omitted, the *insideFunction2* variable is global.

Local Variables

Variables initialized inside a function using the *var* keyword will have a local scope. If you initialize a variable inside a

function without the `var` keyword, your variable will have a global scope. Parameters are local variables, as if the keyword `var` was included before the parameter. Local variables, including a parameters, can have the same name as a global variable.

CODE

```
var myName = "estelle"; // global
alertThisName("jonathan");

function alertThisName(myName){ // local
    alert(myName); // alerts "jonathan"
}

alert(myName); // alerts "estelle";
```

Listing 2: The variable `myName` is declared both globally outside of the function, and locally as the function parameter. Variables declared as function parameters have local scope. The `myName` variable on the first line is a global variable. The variable `myName` declared as a parameter of the function is a local variable only visible within the `alertThisName` function.

In this case, declaring a local variable with the same name as a global variable masks the global variable: all global variables in your script are accessible to this function EXCEPT the global `myName` variable.*

Note: In browsers, global variables that are masked by a function's local variable with the same variable name are still accessible to the function by accessing the global variable thru it's parent. In Listing 2, if you are in a browser, you could access the global `myName` variable from within the `alertThisName` function by using `window.myName` or `top.myName`.

CODE

```
var myCount = 7; // global
doSomeMath();

function doSomeMath(){ //
    var myCount = 7 * 7;
    alert(myCount); // alerts "49"
}

alert(myCount); // alerts "7";
```

Listing 3: In this example, the global and local variables both have the same name. Since the variable within the function was declared using the `var` keyword, the variable has local scope. Manipulating the value of the local value has no impact on the value of the global variable declared in the first line.

Once we exit the function, the local variable `myCount` no longer exists and recognition of the global variable's existence is reinstated.

Similar to listing 2, in this case the local declaration of the `myCount` variable makes the global `myCount` variable inaccessible to this function. The function can access all global variables EXCEPT for the global `myCount` variable, unless you access the variable as noted in the tip above.

CODE

```

var myCount = 7; // global
doSomeMath();

function doSomeMath(){
    myCount = 7 * 7;
    alert(myCount); // alerts "49"
}

alert(myCount); // alerts "49";

```

Listing 4: Since the variable within the function was declared WITHOUT using the `var` keyword, the variable has GLOBAL scope. The global variable declared in the first line and the variable used in the function both have the same name and are, indeed, the same global variable. In this example, manipulating the value of the global value within the function changed the value of the variable throughout the program.

CODE

```

function doSomeMath(){
    myCount = 7 * 7;
    alert(myCount); // alerts "49"
}

alert(myCount); // throws an error: myCount is undefined

```

Listing 5: While the variable in the `doSomeMath` function would be a global variable, as the variable is declared in the function WITHOUT using the `var` keyword. Since the function is never called, the variable is never declared. Accessing the variable that doesn't exist in the last line throws an error.

Rules to remember:

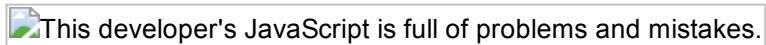
1. Local variables inside a function can only be referenced from within the function in which the local variable was declared.
2. All global variables can be referenced from within any function.
3. All global variables can be referenced from outside any function.
4. No local variables, declared with use of the `var` keyword, can be referenced from outside the function in which they were declared

82. Buggy JavaScript Code: The 10 Most Common Mistakes JavaScript Developers Make

Today, [JavaScript](#) is at the core of virtually all modern web applications. The past several years in particular have witnessed the proliferation of a wide array of powerful JavaScript-based libraries and frameworks for [single page application \(SPA\)](#) development, graphics and animation, and even server-side JavaScript platforms. JavaScript has truly become ubiquitous in the world of web app development and is therefore an increasingly important skill to master.

At first blush, JavaScript may seem quite simple. And indeed, to build basic JavaScript functionality into a web page is a fairly straightforward task for any experienced software developer, even if they're new to JavaScript. Yet the language is significantly more nuanced, powerful, and complex than one would initially believe. **Indeed,**

many of JavaScript's subtleties lead to a number of common problems that keep it from working – 10 of which we discuss here – that are important to be aware of and avoid in one's quest to become a master JavaScript developer.



Common Mistake #1: Incorrect references to this

I once heard a comedian say:

*"I'm not really **here**, because what's **here**, besides **there**, without the '**t**'?"*

That joke in many ways characterizes the type of confusion that often exists for developers regarding [JavaScript's this keyword](#). I mean, is `this` really this, or is it something else entirely? Or is it `undefined`?

As JavaScript coding techniques and design patterns have become increasingly sophisticated over the years, there's been a corresponding increase in the proliferation of self-referencing scopes within callbacks and closures, which are a fairly common source of "this/that confusion".

Consider this example code snippet:

```
Game.prototype.restart = function () {
  this.clearLocalStorage();
  this.timer = setTimeout(function() {
    this.clearBoard();    // what is "this"?
  }, 0);
};
```

CODE

Executing the above code results in the following error:

```
Uncaught TypeError: undefined is not a function
```

CODE

Why?

It's all about context. The reason you get the above error is because, when you invoke `setTimeout()`, you are actually invoking `window.setTimeout()`. As a result, the anonymous function being passed to `setTimeout()` is being defined in the context of the `window` object, which has no `clearBoard()` method.

A traditional, old-browser-compliant solution is to simply save your reference to `this` in a variable that can then be inherited by the closure; e.g.:

```
Game.prototype.restart = function () {
  this.clearLocalStorage();
  var self = this;    // save reference to 'this', while it's still this!
  this.timer = setTimeout(function(){
    self.clearBoard();    // oh OK, I do know who 'self' is!
  }, 0);
};
```

CODE

Alternatively, in newer browsers, you can use the `bind()` method to pass in the proper reference:

```
Game.prototype.restart = function () {
  this.clearLocalStorage();
  this.timer = setTimeout(this.reset.bind(this), 0); // bind to 'this'
};

Game.prototype.reset = function(){
  this.clearBoard(); // ahhh, back in the context of the right 'this'!
};
```

CODE

Common Mistake #2: Thinking there is block-level scope

As discussed in our [JavaScript Hiring Guide](#), a common source of confusion among JavaScript developers (and therefore a common source of bugs) is assuming that JavaScript creates a new scope for each code block. Although this is true in many other languages, it is *not* true in JavaScript. Consider, for example, the following code:

```
for (var i = 0; i < 10; i++) {
  /* ... */
}
console.log(i); // what will this output?
```

CODE

If you guess that the `console.log()` call would either output `undefined` or throw an error, you guessed incorrectly. Believe it or not, it will output `10`. Why?

In most other languages, the code above would lead to an error because the "life" (i.e., scope) of the variable `i` would be restricted to the `for` block. In JavaScript, though, this is not the case and the variable `i` remains in scope even after the `for` loop has completed, retaining its last value after exiting the loop. (This behavior is known, incidentally, as [variable hoisting](#)).

It is worth noting, though, that support for block-level scopes *is* making its way into JavaScript through the [new `let` keyword](#). The `let` keyword is already available in JavaScript 1.7 and is slated to become an officially supported JavaScript keyword as of [ECMAScript 6](#).

New to JavaScript? Read up on [scopes, prototypes, and more](#).

Common Mistake #3: Creating memory leaks

Memory leaks are almost inevitable JavaScript problems if you're not consciously coding to avoid them. There are numerous ways for them to occur, so we'll just highlight a couple of their more common occurrences.

Memory Leak Example 1: Dangling references to defunct objects

Consider the following code:

CODE

```

var theThing = null;
var replaceThing = function () {
    var priorThing = theThing; // hold on to the prior thing
    var unused = function () {
        // 'unused' is the only place where 'priorThing' is referenced,
        // but 'unused' never gets invoked
        if (priorThing) {
            console.log("hi");
        }
    };
    theThing = {
        longStr: new Array(1000000).join('*'), // create a 1MB object
        someMethod: function () {
            console.log(someMessage);
        }
    };
};
setInterval(replaceThing, 1000); // invoke `replaceThing` once every second

```

If you run the above code and monitor memory usage, you'll find that you've got a massive memory leak, leaking a full megabyte per second! And even a manual GC doesn't help. So it looks like we are leaking `longStr` every time `replaceThing` is called. But why?

Let's examine things in more detail:

Each `theThing` object contains its own 1MB `longStr` object. Every second, when we call `replaceThing`, it holds on to a reference to the prior `theThing` object in `priorThing`. But we still wouldn't think this would be a problem, since each time through, the previously referenced `priorThing` would be dereferenced (when `priorThing` is reset via `priorThing = theThing;`). And moreover, is only referenced in the main body of `replaceThing` and in the function `unused` which is, in fact, never used.

So again we're left wondering why there is a memory leak here!?

To understand what's going on, we need to better understand how things are working in JavaScript under the hood. The typical way that closures are implemented is that every function object has a link to a dictionary-style object representing its lexical scope. If both functions defined inside `replaceThing` actually used `priorThing`, it would be important that they both get the same object, even if `priorThing` gets assigned to over and over, so both functions share the same lexical environment. But as soon as a variable is used by any closure, it ends up in the lexical environment shared by all closures in that scope. And that little nuance is what leads to this gnarly memory leak. (More detail on this is available [here](#).)

Memory Leak Example 2: Circular references

Consider this code fragment:

CODE

```

function addClickHandler(element) {
    element.click = function onClick(e) {
        alert("Clicked the " + element.nodeName)
    }
}

```

Here, `onClick` has a closure which keeps a reference to `element` (via `element.nodeName`). By also assigning

`onClick` to `element.click`, the circular reference is created; i.e.: `element -> onClick -> element -> onClick -> element ...`

Interestingly, even if `element` is removed from the DOM, the circular self-reference above would prevent `element` and `onClick` from being collected, and hence, a memory leak.

Avoiding Memory Leaks: What you need to know

JavaScript's memory management (and, in particular, [garbage collection](#)) is largely based on the notion of object reachability.

The following objects are assumed to be *reachable* and are known as "roots":

- Objects referenced from anywhere in the current *call stack* (that is, all local variables and parameters in the functions currently being invoked, and all the variables in the closure scope)
- All *global* variables

Objects are kept in memory at least as long as they are accessible from any of the roots through a reference, or a chain of references.

There is a Garbage Collector (GC) in the browser which cleans memory occupied by unreachable objects; i.e., objects will be removed from memory *if and only if* the GC believes that they are unreachable. Unfortunately, it's fairly easy to end up with defunct "zombie" objects that are in fact no longer in use but that the GC still thinks are "reachable".

Related: [JavaScript Best Practices and Tips by Toptal Developers](#)

Common Mistake #4: Confusion about equality

One of the conveniences in JavaScript is that it will automatically coerce any value being referenced in a boolean context to a boolean value. But there are cases where this can be as confusing as it is convenient. Some of the following, for example, have been known to bite many a JavaScript developer:

```
// All of these evaluate to 'true'!
console.log(false == '0');
console.log(null == undefined);
console.log(" \t\r\n" == 0);
console.log('' == 0);

// And these do too!
if ({}) // ...
if ([]) // ...
```

CODE

With regard to the last two, despite being empty (which might lead one to believe that they would evaluate to `false`), both `{}` and `[]` are in fact objects and *any* object will be coerced to a boolean value of `true` in JavaScript, consistent with the [ECMA-262 specification](#).

As these examples demonstrate, the rules of type coercion can sometimes be clear as mud. Accordingly, unless type coercion is explicitly desired, it's typically best to use `==` and `!=` (rather than `=` and `!=`), so as to avoid any unintended side-effects of type coercion. (`=` and `!=` automatically perform type conversion when comparing two things, whereas `==` and `!=` do the same comparison without type conversion.)

And completely as a sidepoint – but since we're talking about type coercion and comparisons – it's worth mentioning that comparing `NaN` with *anything* (even `NaN` !) will *always* return `false`. You therefore cannot use the equality operators (`==`, `===`, `!=`, `!==`) to determine whether a value is `NaN` or not. Instead, use the built-in global `isNaN()` function:

```
console.log(NaN == NaN);      // false
console.log(NaN === NaN);     // false
console.log(isNaN(NaN));      // true
```

CODE

Common Mistake #5: Inefficient DOM manipulation

JavaScript makes it relatively easy to manipulate the DOM (i.e., add, modify, and remove elements), but does nothing to promote doing so efficiently.

A common example is code that adds a series of DOM Elements one at a time. Adding a DOM element is an expensive operation. Code that adds multiple DOM elements consecutively is inefficient and likely not to work well.

One effective alternative when multiple DOM elements need to be added is to use [document fragments](#) instead, thereby improving both efficiency and performance.

For example:

```
var div = document.getElementsByTagName("my_div");

var fragment = document.createDocumentFragment();

for (var e = 0; e < elems.length; e++) { // elems previously set to list of elements
    fragment.appendChild(elems[e]);
}
div.appendChild(fragment.cloneNode(true));
```

CODE

In addition to the inherently improved efficiency of this approach, creating attached DOM elements is expensive, whereas creating and modifying them while detached and then attaching them yields much better performance.

Common Mistake #6: Incorrect use of function definitions inside for loops

Consider this code:

```
var elements = document.getElementsByTagName('input');
var n = elements.length; // assume we have 10 elements for this example
for (var i = 0; i < n; i++) {
    elements[i].onclick = function() {
        console.log("This is element #" + i);
    };
}
```

CODE

Based on the above code, if there were 10 input elements, clicking *any* of them would display "This is element #10"! This is because, by the time `onclick` is invoked for *any* of the elements, the above for loop will have completed and the value of `i` will already be 10 (for *all* of them).

Here's how we can correct the above code problems, though, to achieve the desired behavior:

```
var elements = document.getElementsByTagName('input');
var n = elements.length;      // assume we have 10 elements for this example
var makeHandler = function(num) { // outer function
    return function() { // inner function
        console.log("This is element #" + num);
    };
};

for (var i = 0; i < n; i++) {
    elements[i].onclick = makeHandler(i+1);
}
```

CODE

In this revised version of the code, `makeHandler` is immediately executed each time we pass through the loop, each time receiving the then-current value of `i+1` and binding it to a scoped `num` variable. The outer function returns the inner function (which also uses this scoped `num` variable) and the element's `onclick` is set to that inner function. This ensures that each `onclick` receives and uses the proper `i` value (via the scoped `num` variable).

Common Mistake #7: Failure to properly leverage prototypal inheritance

A surprisingly high percentage of JavaScript developers fail to fully understand, and therefore to fully leverage, the features of prototypal inheritance.

Here's a simple example. Consider this code:

```
BaseObject = function(name) {
    if(typeof name !== "undefined") {
        this.name = name;
    } else {
        this.name = 'default'
    }
};
```

CODE

Seems fairly straightforward. If you provide a name, use it, otherwise set the name to 'default'; e.g.:

```
var firstObj = new BaseObject();
var secondObj = new BaseObject('unique');

console.log(firstObj.name); // -> Results in 'default'
console.log(secondObj.name); // -> Results in 'unique'
```

CODE

But what if we were to do this:

```
delete secondObj.name;
```

CODE

We'd then get:

CODE

```
console.log(secondObj.name); // -> Results in 'undefined'
```

But wouldn't it be nicer for this to revert to 'default'? This can easily be done, if we modify the original code to leverage prototypal inheritance, as follows:

CODE

```
BaseObject = function (name) {
    if(typeof name !== "undefined") {
        this.name = name;
    }
};

BaseObject.prototype.name = 'default';
```

With this version, `BaseObject` inherits the `name` property from its `prototype` object, where it is set (by default) to `'default'`. Thus, if the constructor is called without a name, the name will default to `default`. And similarly, if the `name` property is removed from an instance of `BaseObject`, the prototype chain will then be searched and the `name` property will be retrieved from the `prototype` object where its value is still `'default'`. So now we get:

CODE

```
var thirdObj = new BaseObject('unique');
console.log(thirdObj.name); // -> Results in 'unique'

delete thirdObj.name;
console.log(thirdObj.name); // -> Results in 'default'
```

Common Mistake #8: Creating incorrect references to instance methods

Let's define a simple object, and create an instance of it, as follows:

CODE

```
var MyObject = function() {}

MyObject.prototype.whoAmI = function() {
    console.log(this === window ? "window" : "MyObj");
};

var obj = new MyObject();
```

Now, for convenience, let's create a reference to the `whoAmI` method, presumably so we can access it merely by `whoAmI()` rather than the longer `obj.whoAmI()`:

CODE

```
var whoAmI = obj.whoAmI;
```

And just to be sure everything looks copacetic, let's print out the value of our new `whoAmI` variable:

CODE

```
console.log(whoAmI);
```

Outputs:

```
function () {
    console.log(this === window ? "window" : "MyObj");
}
```

CODE

OK, cool. Looks fine.

But now, look at the difference when we invoke `obj.whoAmI()` vs. our convenience reference `whoAmI()`:

```
obj.whoAmI(); // outputs "MyObj" (as expected)
whoAmI();     // outputs "window" (uh-oh!)
```

CODE

What went wrong?

The headfake here is that, when we did the assignment `var whoAmI = obj.whoAmI;`, the new variable `whoAmI` was being defined in the *global* namespace. As a result, its value of `this` is `window`, *not* the `obj` instance of `MyObject`!

Thus, if we really need to create a reference to an existing method of an object, we need to be sure to do it within that object's namespace, to preserve the value of `this`. One way of doing this would be, for example, as follows:

```
var MyObject = function() {}

MyObject.prototype.whoAmI = function() {
    console.log(this === window ? "window" : "MyObj");
};

var obj = new MyObject();
obj.w = obj.whoAmI; // still in the obj namespace

obj.whoAmI(); // outputs "MyObj" (as expected)
obj.w();      // outputs "MyObj" (as expected)
```

CODE

Common Mistake #9: Providing a string as the first argument to `setTimeout` or `setInterval`

For starters, let's be clear on something here: Providing a string as the first argument to `setTimeout` or `setInterval` is **not** itself a mistake per se. It is perfectly legitimate JavaScript code. The issue here is more one of performance and efficiency. What is rarely explained is that, under the hood, if you pass in a string as the first argument to `setTimeout` or `setInterval`, it will be passed to the *function constructor* to be converted into a new function. This process can be slow and inefficient, and is rarely necessary.

The alternative to passing a string as the first argument to these methods is to instead pass in a *function*. Let's take a look at an example.

Here, then, would be a fairly typical use of `setInterval` and `setTimeout`, passing a *string* as the first parameter:

CODE

```
setInterval("logTime()", 1000);
setTimeout("logMessage('" + msgValue + "')", 1000);
```

The better choice would be to pass in a *function* as the initial argument; e.g.:

```
setInterval(logTime, 1000); // passing the logTime function to setInterval

setTimeout(function() {
    logMessage(msgValue); // (msgValue is still accessible in this scope)
}, 1000);
```

CODE

Common Mistake #10: Failure to use "strict mode"

As explained in our [JavaScript Hiring Guide](#), "strict mode" (i.e., including `'use strict'`; at the beginning of your JavaScript source files) is a way to voluntarily enforce stricter parsing and error handling on your JavaScript code at runtime, as well as making it more secure.

While, admittedly, failing to use strict mode is not a "mistake" per se, its use is increasingly being encouraged and its omission is increasingly becoming considered bad form.

Here are some key benefits of strict mode:

- **Makes debugging easier.** Code errors that would otherwise have been ignored or would have failed silently will now generate errors or throw exceptions, alerting you sooner to problems in your code and directing you more quickly to their source.
- **Prevents accidental globals.** Without strict mode, assigning a value to an undeclared variable automatically creates a global variable with that name. This is one of the most common errors in JavaScript. In strict mode, attempting to do so throws an error.
- **Eliminates this coercion.** Without strict mode, a reference to a `this` value of null or undefined is automatically coerced to the global. This can cause many headfakes and pull-out-your-hair kind of bugs. In strict mode, referencing a `this` value of null or undefined throws an error.
- **Disallowss duplicate property names or parameter values.** Strict mode throws an error when it detects a duplicate named property in an object (e.g., `var object = {foo: "bar", foo: "baz"};`) or a duplicate named argument for a function (e.g., `function foo(val1, val2, val1){}`), thereby catching what is almost certainly a bug in your code that you might otherwise have wasted lots of time tracking down.
- **Makes eval() safer.** There are some differences in the way `eval()` behaves in strict mode and in non-strict mode. Most significantly, in strict mode, variables and functions declared inside of an `eval()` statement are *not* created in the containing scope (they *are* created in the containing scope in non-strict mode, which can also be a common source of problems).
- **Throws error on invalid usage of delete.** The `delete` operator (used to remove properties from objects) cannot be used on non-configurable properties of the object. Non-strict code will fail silently when an attempt is made to delete a non-configurable property, whereas strict mode will throw an error in such a case.

Wrap-up

As is true with any technology, the better you understand why and how JavaScript works and doesn't work, the more solid your code will be and the more you'll be able to effectively harness to true power of the language. Conversely,

lack of proper understanding of JavaScript paradigms and concepts is indeed where many JavaScript problems lie.

Thoroughly familiarizing yourself with the language's nuances and subtleties is the most effective strategy for improving your proficiency and increasing your [productivity](#). Avoiding many common JavaScript mistakes will help when your JavaScript is not working.

83. What You Should Already Know about JavaScript Scope

If you are a novice JavaScript programmer, or if you've been messing around with JQuery to pull off a few animations on your website, chances are you're missing a few vital chunks of knowledge about JavaScript.

One of the most important concepts is how scope binds to "this".

For this post, I'm going to assume you have a decent understanding of JavaScript's basic syntax/objects and general terminology when discussing scope (block vs. function scope, this keyword, lexical vs. dynamic scoping).

Lexical Scoping

First off, JavaScript has *lexical scoping* with *function scope*. In other words, even though JavaScript looks like it should have block scope because it uses curly braces { }, a new scope is created only when you create a new function.

```
var outerFunction = function(){

    if(true){
        var x = 5;
        //console.log(y); //line 1, ReferenceError: y not defined
    }

    var nestedFunction = function() {

        if(true){
            var y = 7;
            console.log(x); //line 2, x will still be known prints 5
        }

        if(true){
            console.log(y); //line 3, prints 7
        }
    }
    return nestedFunction;
}

var myFunction = outerFunction();
myFunction();
```

CODE

In this example, the variable x is available everywhere inside of outerFunction(). Also, the variable y is available everywhere within the nestedFunction(), but neither are available outside of the function where they were defined. The reason for this can be explained by lexical scoping. The scope of variables is defined by their position in source code. In order to resolve variables, JavaScript starts at the innermost scope and searches outwards until it finds the variable it was looking for. Lexical scoping is nice, because we can easily figure out what the value of a variable will be by looking at the code; whereas in dynamic scoping, the meaning of a variable can change at runtime,

making it more difficult.

Closures

The fact that we can access the variable `x` might still be confusing, because, normally, a local variable inside a function is gone after a function finishes executing. We called `outerFunction()` and assigned its result, `nestedFunction()`, into `myFunction()`. How does the variable `x` still exist if `outerFunction()` has already returned?

Merely accessing a variable outside of the immediate scope (no return statement is necessary) will create something called a *closure*. [Mozilla Development Network\(MDN\)](#) gives a great definition:

"A closure is a special kind of object that combines two things: a function, and the environment in which that function was created. The environment consists of any local variables that were in-scope at the time that the closure was created."

Since `x` is a member of the environment that created `nestedFunction()`, `nestedFunction()` will have access to it. Straightforward enough? It's about to get more interesting, because `this` doesn't behave like a regular variable. Try this example, which has nested functions inside of an object with some properties:

```
var cat = {
    name: "Gus",
    color: "gray",
    age: 15,
    printInfo: function() {
        console.log("Name:", this.name, "Color:", this.color, "Age:", this.age); //line 1, prints correctly
    },
    nestedFunction = function() {
        console.log("Name:", this.name, "Color:", this.color, "Age:", this.age); //line 2, loses cat scope
    }
}
cat.printInfo(); //prints Name: window Color: undefined Age: undefined
```

CODE

Why are `color` and `age` undefined in line 2? You might be thinking, "the `cat` object properties are clearly defined above and are in an outer more global scope aren't they?" More importantly, where did "window" come from?

JavaScript loses scope of `this` when used inside of a function that is contained inside of another function. When it is lost, by default, `this` will be bound to the global `window` object. In our example, it just so happens that the `window` object also has a "name" property with a value of "window".

Controlling Context

So what now?

We can't alter how lexical scoping in JavaScript works, but we can control the *context* in which we call our functions. Context is decided at runtime when the function is called, and it's always bound to the object the function was called within. The only exception to this rule is the nested function case above.

By saying, change the "context", I mean, we're changing what "*this*" actually is. In the example below, what do "line 1" and "line 2" print out?

```
var obj1 = {
  printThis: function() {
    console.log(this);
  }
};

var func1 = obj1.printThis;
obj1.printThis(); //line 1
func1(); //line 2
```

CODE

Line 1 prints out `obj1`. Line 2 prints out the window. The context of line 1 is `obj1`, because we called `printThis()` on it immediately. However, in `func1()`, we first store a reference to the `printThis()` function, then call it in context of the global object; as a result, it prints the window. If `func1()` had been nested within a different function, that would have been its context.

Call, Bind, and Apply

There are multiple ways to control the value of *this*, including the following:

- storing a reference to *this* in another variable
- `.call()`
- `.apply()`
- `.bind()`

The first one:

```
var cat = {
  name: "Gus",
  color: "gray",
  age: 15,

  printInfo: function() {
    var that = this;
    console.log("Name:", this.name, "Color:", this.color, "Age:", this.age); //prints correctly

    nestedFunction = function() {
      console.log("Name:", that.name, "Color:", that.color, "Age:", that.age); //prints correctly
    }
    nestedFunction();
  }
}

cat.printInfo();
```

CODE

Because we bound *this* to a variable `that`, it will be available just like any other variable. Let's take a look now at

`call()`, `apply()`, and `bind()`, which, I think, are a bit cleaner.

CODE

```
var cat = {
  name: "Gus",
  color: "gray",
  age: 15,

  printInfo: function() {
    console.log("Name:", this.name, "Color:", this.color, "Age:", this.age);
    nestedFunction = function() {
      console.log("Name:", this.name, "Color:", this.color, "Age:", this.age);
    }
    nestedFunction.call(this);
    nestedFunction.apply(this);

    var storeFunction = nestedFunction.bind(this);
    storeFunction();
  }
}
cat.printInfo();
```

Focus on the first argument of each function: `this`. It refers to the `cat` object. Now, `nestedFunction()`'s `this` refers to the `cat` object. If, instead of passing in `this` as an argument, we passed in a "dog" object, then `nestedFunction()`'s `this` refers to dog. Basically, whatever the first argument is becomes the `nestedFunction()`'s `this`.

So what's the difference between the three?

The main difference between `call()` and `apply()` lies within how to pass extra arguments. `Call()` takes in multiple arguments, separated by commas, which allows the `nestedFunction()` to utilize them. `Bind()` also takes extra arguments in this way. However, `apply()` takes in a single array of arguments, rather than multiple arguments.

It is important to remember that using `call()` and `apply()` actually invoke your function -- so this would be incorrect (notice the `()` on `nestedFunction`):

CODE

```
1----- nestedFunction().call(this);-----
```

CODE

`Bind()`, on the other hand, is nifty because it allows you to change what `this` references and then store a reference to the altered function in a variable to be used at a later time (see `storeFunction` in the code example above). Meanwhile, because `call()` and `apply()` immediately run the function, they return the result of calling that function.

Practical Applications

Thus far, we have seen closures, `call()`, `apply()`, and `bind()`, but we have not discussed any practical applications and when to use each one to get the correct binding of `this`.

1. Closures are best used when you have nested functions similar to the first `cat` example above, which used `var that = this`. This method is also nice because you don't have to worry about cross-platform issues. For instance, `bind()` is a recent addition to ECMAScript5 and isn't always supported.

2. `Call()` and `apply()` are useful for when you want to borrow a method from one object and use it in a completely separate object. For example, using our cat example above, we could reuse its `printInfo()` function to print information about a dog.
3. `Bind()` is useful for maintaining context in asynchronous callbacks and events.

Great! I hope you now have a little more insight regarding how scope in JavaScript is handled. We have covered the basics of lexical scoping with function scope, closures, ways to control the context through closures, `call()`, `apply()`, `bind()`, and lastly, some practical applications.

84. Understanding "Prototypes" in JavaScript

For the purposes of this post, I will be talking about JavaScript objects using syntax defined in ECMAScript 5.1. The basic semantics existed in Edition 3, but they were not well exposed.

A Whole New Object

In JavaScript, objects are pairs of keys and values (in Ruby, this structure is called a Hash; in Python, it's called a dictionary). For example, if I wanted to describe my name, I could have an object with two keys: `firstName` would point to "Yehuda" and `lastName` would point to "Katz". Keys in a JavaScript object are Strings.

To create the simplest new object in JavaScript, you can use `Object.create`:

```
var person = Object.create(null); // this creates an empty object
```

CODE

Why didn't we just use `var person = {};`? Stick with me! To look up a value in the object by key, use bracket notation. If there is no value for the key in question, JavaScript will return `undefined`.

```
person['name'] // undefined
```

CODE

If the String is a valid identifier[1], you can use the dot form:

```
person.name // undefined
```

CODE

[1] in general, an [identifier](#) starts with a unicode letter, \$, _, followed by any of the starting characters or numbers. A valid identifier must also not be a [reserved word](#). There are other allowed characters, such as unicode combining marks, unicode connecting punctuation, and unicode escape sequences. Check out the spec for the full details

Adding values

So now you have an empty object. Not that useful, eh? Before we can add some properties, we need to understand what a property (what the spec calls a "named data property") looks like in JavaScript.

Obviously, a property has a name and a value. In addition, a property can be **enumerable**, **configurable** and **writable**. If a value is enumerable, it will show up when enumerating over an object using a `for(prop in obj)` loop. If a property is writable, you can replace it. If a property is configurable, you can delete it or change its other

attributes.

In general, when we create a new property, we will want it to be enumerable, configurable, and writable. In fact, prior to ECMAScript 5, that was the only kind of property a user could create directly.

We can add a property to an object using `Object.defineProperty`. Let's add a first name and last name to our empty object:

```
var person = Object.create(null);
Object.defineProperty(person, 'firstName', {
  value: "Yehuda",
  writable: true,
  enumerable: true,
  configurable: true
});

Object.defineProperty(person, 'lastName', {
  value: "Katz",
  writable: true,
  enumerable: true,
  configurable: true
});
```

CODE

Obviously, this is extremely verbose. We can make it a bit less verbose by eliminating the common defaults:

```
var config = {
  writable: true,
  enumerable: true,
  configurable: true
};

var defineProperty = function(obj, name, value) {
  config.value = value;
  Object.defineProperty(obj, name, config);
}

var person = Object.create(null);
defineProperty(person, 'firstName', "Yehuda");
defineProperty(person, 'lastName', "Katz");
```

CODE

Still, this is pretty ugly to create a simple property list. Before we can get to a prettier solution, we will need to add another weapon to our JavaScript object arsenal.

Prototypes

So far, we've talked about objects as simple pairs of keys and values. In fact, JavaScript objects also have one additional attribute: a pointer to *another* object. We call this pointer the object's *prototype*. If you try to look up a key on an object and it is not found, JavaScript will look for it in the prototype. It will follow the "prototype chain" until it sees a `null` value. In that case, it returns `undefined`.

You'll recall that we created a new object by invoking `Object.create(null)`. The parameter tells JavaScript what it should set as the Object's *prototype*. You can look up an object's prototype by using `Object.getPrototypeOf`:

CODE

```

var man = Object.create(null);
defineProperty(man, 'sex', "male");

var yehuda = Object.create(man);
defineProperty(yehuda, 'firstName', "Yehuda");
defineProperty(yehuda, 'lastName', "Katz");

yehuda.sex      // "male"
yehuda.firstName // "Yehuda"
yehuda.lastName // "Katz"

Object.getPrototypeOf(yehuda) // returns the man object

```

We can also add functions that we share across many objects this way:

CODE

```

var person = Object.create(null);
defineProperty(person, 'fullName', function() {
  return this.firstName + ' ' + this.lastName;
});

// this time, let's make man's prototype person, so all
// men share the fullName function
var man = Object.create(person);
defineProperty(man, 'sex', "male");

var yehuda = Object.create(man);
defineProperty(yehuda, 'firstName', "Yehuda");
defineProperty(yehuda, 'lastName', "Katz");

yehuda.sex      // "male"
yehuda.fullName() // "Yehuda Katz"

```

Setting Properties

Since creating a new writable, configurable, enumerable property is pretty common, JavaScript makes it easy to do so using assignment syntax. Let's update the previous example using assignment instead of `defineProperty`:

CODE

```

var person = Object.create(null);

// instead of using defineProperty and specifying writable,
// configurable, and enumerable, we can just assign the
// value directly and JavaScript will take care of the rest
person['fullName'] = function() {
    return this.firstName + ' ' + this.lastName;
};

// this time, let's make man's prototype person, so all
// men share the fullName function
var man = Object.create(person);
man['sex'] = "male";

var yehuda = Object.create(man);
yehuda['firstName'] = "Yehuda";
yehuda['lastName'] = "Katz";

yehuda.sex      // "male"
yehuda.fullName() // "Yehuda Katz"

```

Just like when looking up properties, if the property you are defining is an *identifier*, you can use dot syntax instead of bracket syntax. For instance, you could say `man.sex = "male"` in the example above.

Object Literals

Still, having to set a number of properties every time can get annoying. JavaScript provides a literal syntax for creating an object and assigning properties to it at one time.

CODE

```
var person = { firstName: "Paul", lastName: "Irish" }
```

This syntax is approximately sugar for:

CODE

```

var person = Object.create(Object.prototype);
person.firstName = "Paul";
person.lastName = "Irish";

```

The most important thing about the expanded form is that object literals *always* set the newly created object's prototype to an object located at `Object.prototype`. Internally, the object literal looks like this:

The default `Object.prototype` dictionary comes with a number of the methods we have come to expect objects to contain, and through the magic of the prototype chain, all new objects created as object literal will contain these properties. Of course, objects can happily override them by defining the properties directly. Most commonly, developers will override the `toString` method:

CODE

```

var alex = { firstName: "Alex", lastName: "Russell" };

alex.toString() // "[object Object]"

var brendan = {
  firstName: "Brendan",
  lastName: "Eich",
  toString: function() { return "Brendan Eich"; }
};

brendan.toString() // "Brendan Eich"

```

This is especially useful because a number of internal coercion operations use a supplied `toString` method.

Unfortunately, this literal syntax only works if we are willing to make the new object's prototype `Object.prototype`. This eliminates the benefits we saw earlier of sharing properties using the prototype. In many cases, the convenience of the simple object literal outweighs this loss. In other cases, you will want a simple way to create a new object with a particular prototype. I'll explain it right afterward:

CODE

```

var fromPrototype = function(prototype, object) {
  var newObject = Object.create(prototype);

  for (var prop in object) {
    if (object.hasOwnProperty(prop)) {
      newObject[prop] = object[prop];
    }
  }

  return newObject;
};

var person = {
  toString: function() {
    return this.firstName + ' ' + this.lastName;
  }
};

var man = fromPrototype(person, {
  sex: "male"
});

var jeremy = fromPrototype(man, {
  firstName: "Jeremy",
  lastName: "Ashkenas"
});

jeremy.sex      // "male"
jeremy.toString() // "Jeremy Ashkenas"

```

Let's deconstruct the `fromPrototype` method. The goal of this method is to create a new object with a set of properties, but with a particular prototype. First, we will use `Object.create()` to create a new empty object, and assign the prototype we specify. Next, we will enumerate all of the properties in the object that we supplied, and copy them over to the new object.

Remember that when you create an object literal, like the ones we are passing in to `fromPrototype`, it will always have `Object.prototype` as its prototype. By default, the properties that JavaScript includes on `Object.prototype` are *not enumerable*, so we don't have to worry about `valueOf` showing up in our loop. However, since `Object.prototype` is an object like any other object, anyone can define a new property on it, which may (and probably would) be marked enumerable.

As a result, while we are looping through the properties on the object we passed in, we want to restrict our copying to properties that were defined on the object itself, and not found on the prototype. JavaScript includes a method called `hasOwnProperty` on `Object.prototype` to check whether a property was defined on the object itself. Since object literals will always have `Object.prototype` as their prototype, you can use it in this manner.

The object we created in the example above looks like this:

Native Object Orientation

At this point, it should be obvious that prototypes can be used to inherit functionality, much like traditional object oriented languages. To facilitate using it in this manner, JavaScript provides a `new` operator.

In order to facilitate object oriented programming, JavaScript allows you to use a Function object as a combination of a prototype to use for the new object and a constructor function to invoke:

```
var Person = function(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

Person.prototype = {
  toString: function() { return this.firstName + ' ' + this.lastName; }
}
```

CODE

Here, we have a single Function object that is both a constructor function and an object to use as the prototype of new objects. Let's implement a function that will create new instances from this `Person` object:

```
function newObject(func) {
  // get an Array of all the arguments except the first one
  var args = Array.prototype.slice.call(arguments, 1);

  // create a new object with its prototype assigned to func.prototype
  var object = Object.create(func.prototype);

  // invoke the constructor, passing the new object as 'this'
  // and the rest of the arguments as the arguments
  func.apply(object, args);

  // return the new object
  return object;
}

var brendan = newObject(Person, "Brendan", "Eich");
brendan.toString() // "Brendan Eich"
```

CODE

The `new` operator in JavaScript essentially does this work, providing a syntax familiar to those comfortable with

traditional object oriented languages:

```
var mark = new Person("Mark", "Miller");
mark.toString() // "Mark Miller"
```

CODE

In essence, a JavaScript "class" is just a Function object that serves as a constructor plus an attached prototype object. I mentioned before that earlier versions of JavaScript did not have `Object.create`. Since it is so useful, people often created something like it using the `new` operator:

```
var createObject = function (o) {
  // we only want the prototype part of the `new`
  // behavior, so make an empty constructor
  function F() {}

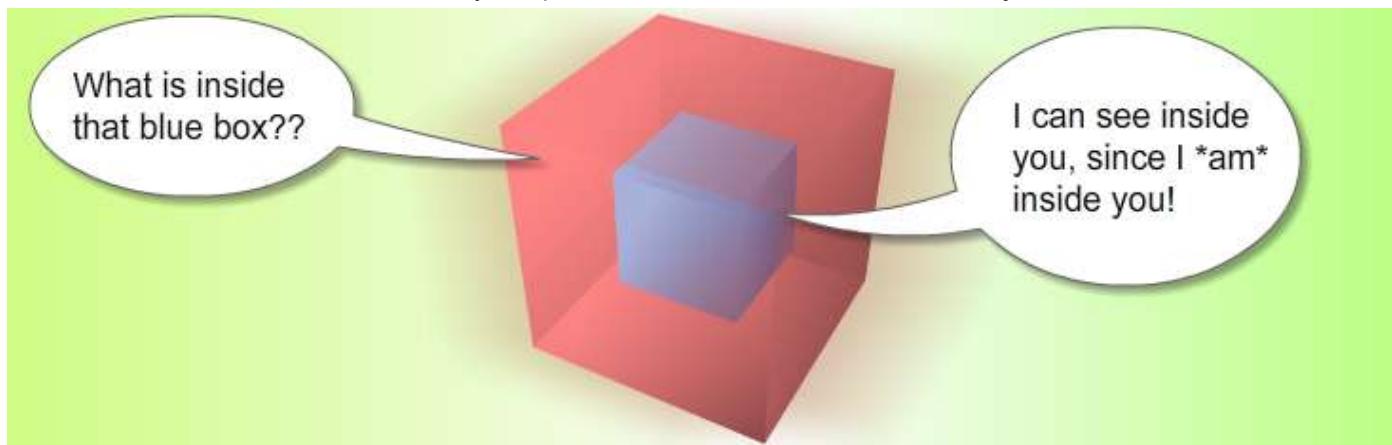
  // set the function's `prototype` property to the
  // object that we want the new object's prototype
  // to be.
  F.prototype = o;

  // use the `new` operator. We will get a new
  // object whose prototype is o, and we will
  // invoke the empty function, which does nothing.
  return new F();
};
```

CODE

85. JavaScript Scope and Context – Not the Same Thing!

In forums and other places I often see people incorrectly use the term scope when they should be using the term context. Scope applies to the variable and functional access of a function, whereas context is the property and method access of a function. Essentially, scope is function-based, and context is object-based.



Scope

The following shows a very simple example of the use of scope. `funcA` has access to `var a`, and `funcB` has access to `var b` and `var a`:

CODE

```

var funcA = function(){
  var a = 1;
  var funcB = function(){
    var b = 2;
    console.log(a, b); // outputs: 1, 2
  }
  console.log(a, b); // Error! b is not defined
}
funcA();
//Example #1 - Simple Scope

```

The second log in Example #1 fails because a function captures its scope, and keeps it all to itself. Nobody else has access to `funcB`'s scope, not even `funcA`. However, when `funcA` captured its scope, it included `funcB` and therefore `funcB` has access to all of `funcA`'s goodies.

Confused? Try looking at scope as a box made with one-way mirrors. Nothing can see inside the box, but the contents can see out.

Closures

A closure is when you then pass `funcB` to another function somewhere. The scope of `funcB` is then captured, or "enclosed" and passed along with it. An in-depth look at closures is beyond the "scope" (heh heh) of this article, but you can learn more than you ever wanted to know about them from [Jibbering](#).

Context

When learning the JavaScript language, scope gets all of the attention, but the concept that is most difficult to grasp is *context*. Not knowing what context is and how it works is like trying to learn to paint and not being aware of the color red. The term context can loosely be related to usage of the keyword "this". Hopefully the following example is obvious:

CODE

```

var o = {
  x:10,
  m: function(){
    var x = 1;
    console.log(x, this.x); // outputs 1, 10
  }
}
o.m();
//Example #2 - Simple Context

```

Both `x`'s are able to be accessed because `this.x` is a property within the `m()` method's context and `var x` is a variable within its scope. They do not clobber each other.

That was simple enough, but things start to get tricky when functions are used within methods. Take the following modification to the example:

CODE

```

var o = {
  x:10,
  m: function(){
    var x = 1;
    var f = function(){
      console.log(x, this.x); // outputs 1, undefined
    }
    f();
  }
}
o.m();
//Example #3 - Broken Context

```

Refresher: Methods are associated to objects like: `object.myMethod = function(){};` while functions are not: `var myFunc = function(){}.` Properties are associated to objects like: `object.myProperty = 96;` while variables are not: `var myVariable = 96.`

Example #3 logs `1, undefined`. What happened is that the `f()` function was treated as a method. In the previous example, this worked because `m()` was a method and had the context of `o`. Note the difference between writing `o.m = function()` and `var f = function():` `m()` has a receiving object while `f()` has no such object.

One solution to work around the previous nuance is by accessing `x` through the global object. Instead of `this.x`, `o.x` could be used and that would work.

But if the solution is so simple, then what is the need for the keyword `this`? The reason is you don't always know what object you're in, especially with the use of object instances.

CODE

```

var C = function(){}
C.prototype = {
  x:10,
  m: function(){
    var x = 1;
    var f = function(){
      console.log(x, this.x); // outputs 1, undefined
    }
    f();
  }
}
var instance1 = new C();
instance1.m();
//Example #4 - Broken Context Instance

```

Now `this.x` is part of an instance object, and there can be several, or even hundreds or thousands of object instances. So it becomes impossible or at least unrealistic to try and access the properties via the object name.

So perhaps we should just make `f()` a method, and then it will have access to `this.x`?

CODE

```

var C = function(){}
C.prototype = {
  x:10,
  m: function(){
    var x = 1;
    this.f();
  },
  f: function(){
    console.log(x, this.x); // Reference ERROR!!
  }
}
var instance1 = new C();
instance1.m();
//Example #5 - Context with Broken Scope

```

Oops. That was definitely not the right thing if you need access to both `var x` and `this.x`.

Using Context to fix Scope

So how to fix... this... example? One possible solution is to create `f()` with `this` context, **within** the `m()` method:

CODE

```

var C = function(){}
C.prototype = {
  x:10,
  m: function(){
    var x = 1;
    this.f = function(){
      console.log(x, this.x); // outputs 1, 10
    }
    this.f();
  }
}
var instance1 = new C();
instance1.m();

//Example #6 - Context Fixing Scope

```

Now calling `this.f()`, works pretty well because now not only does `f()` has the context of the `instance1`, but it was created within `m()`, and so it captures that scope, which includes `var x`.

Using Scope to Fix Context

We've probably all run into the following scenario at one time or another.

CODE

```

var o = {
  x:10,
  onTimeout: function(){
    console.log("x:", this.x);
  },
  m: function(){
    setTimeout(function(){
      this.onTimeout(); // ERROR: this.onTimeout is not a function
    }, 1);
  }
}
o.m();

//Example #7 - setTimeout with Broken Context

```

When I first encountered this situation several years ago, I was perplexed for days. It's a hard problem to resolve because you don't really know what's wrong and what keywords to use to search for solutions.

The problem is the context is broken. The way JavaScript currently works is if a function doesn't have context, or is not a method associated with an object, it executes in the global space. That should help identify the issue:

`setTimeout` contains an anonymous function that is not associated with `this`, therefore, `this.onTimeout` does not resolve correctly.

We can fix this, literally, by assigning it to a variable:

CODE

```

var o = {
  x:10,
  onTimeout: function(){
    console.log("x:", this.x); // outputs 10
  },
  m: function(){
    var self = this;
    setTimeout(function(){
      self.onTimeout();
    }, 1);
  }
}
o.m();

//Example #8 - Scope to Fix setTimeout Context

```

`this` is nothing more than a keyword that is used to refer to an object. In this case the object is `o`, and previously it was the `C` object instance. Since it is just an object, we can assign it to a variable. By convention, the most popular variable name is `self` since we are referring to our "self" in a sense. Now when `onTimeout` is invoked, it is done so with the proper context and the code works correctly.

Conclusion

Knowing the difference between scope and context will not only help you understand the JavaScript language better, but you'll be better able to communicate with other developers and ask the proper questions. Likewise, knowing the difference between methods and functions, and properties and variables is important to communicate exactly what it is you're referring to. Remember, `var x` and `this.x` are not the same thing!

86. What is the Difference Between Scope and Context in JavaScript?

In JavaScript, scope and context are not the same thing, and you should understand the difference between them. Fortunately, the answer is short and simple.

When interviewing front-end developers, I usually try to include a question about Scope and Context in JavaScript. I always ask what the difference is. The answers I get often surprise me. It seems that even those with some experience have difficulty answering this question.

The answer is short and simple: Scope pertains to the visibility of variables, and context refers to the object within which a function is executed.

Scope

Scope has to do with the visibility of variables. In JavaScript, scope is achieved through the use of functions. When you use the keyword "var" inside of a function, the variable that you are initializing is private, and cannot be seen outside of that function. But, if there are functions inside of this function, then those "inner" functions can "see" that variable; that variable is said to be "in-scope". Functions can "see" variables that are declared inside of them, and any that are declared outside of them, but never those declared inside of functions that are nested in that function. This is scope in JavaScript.

Context

Context is related to objects. It refers to the object within which a function is executed. When you use the JavaScript keyword "this", that word refers to the object that the function is executing in.

For example, inside
of a function, when

Scope refers to the visibility of variables, and scope refers to the context within which a function is executed.

you say: "this.accountNumber", you are referring to the property "accountNumber", that belongs to the object within which the function is executing. If the object "foo" has a method called "bar", when the JavaScript keyword "this" is used inside of "bar", it refers to "foo". If the function "bar" were executed in the global scope, then "this" refers to the window object. It is important to keep in mind that by using the JavaScript call() or apply() methods, you can alter the context within which a function is executed, which in-turn changes the meaning of "this" inside of that function when it is executed.

Summary

In my opinion, it is critical to understand the difference between scope and context in JavaScript. As soon as you have to write or edit even intermediate-level JavaScript, these two topics become very important. If you don't understand the difference between scope and context, your ability to comfortably and confidently write / edit JavaScript will never improve. This is an important topic. The good news is, it's not that difficult to understand and once you get it, you've got it.

87. Understanding Scope in JavaScript

Although the 'with' statement creates a block-level scope effect, and recent implementations of the "let" statement have the same effect, these are fodder for another conversation. I'm not a big fan of the "with" statement, and at the time of this writing, you can't be 100% sure that the "let" statement is fully supported by the user's browser. The end result of this is the fact that there is no block-level scope in JavaScript. Scope is created by functions.

Example # 1

CODE

```
month = 'july'; //global variable

function foo(){
    var month = "august"; //private
    console.log(month);

};

console.log(month); //in the global scope, month = july

foo(); //in the local scope of the foo function, month = august
```

Here is the jsFiddle.net link for Example # 1 : <http://jsfiddle.net/2ZkkR/>

In Example # 1, we set the global variable "month" equal to "july". But inside of the function "foo()", we also create a variable named "month", and give it a value of "august". The second statement in foo() is a call to the console, telling it to output the value of the variable "month".

So, why is it that in the global scope, "console.log(month)" outputs "july", but executing "foo()" outputs "august" ?

The reason is that inside of the function "foo()", we used the "var" keyword to create the variable "month". When you use the "var" keyword, the variable you create becomes local to the function that you create it in. So, inside of "foo()", the variable "month" is local, and equal to "july". As a result, on the very next line, the statement:

"console.log(month)" outputs "july". Inside of "foo()", we have no knowledge of the global variable "month", because in the scope chain, the variable "month" is found locally, so the search for "month" ends within the local scope of "foo()".

Example # 2

CODE

```
month = 'july'; //global variable

function foo(){
    var month = "august"; //private
    console.log(month);

};

function bar(){
    console.log(month); //no local "month", so it looks to the global "month"
};

console.log(month); //in the global scope, month = july

foo(); //in the local scope of the foo function, month = august

bar(); //in the local scope of the bar function, month = july
```

Here is the jsFiddle.net link for Example # 2 : <http://jsfiddle.net/6TBEM/>

In Example # 2, we have added a function named "bar()". Bar does not have a local variable named "month". So, when "bar()" executes, the statement "console.log(month)" kicks off a search for the variable "month". In the scope chain, there is no local variable named "month", so the JavaScript engine looks to the next level of the scope chain, which happens to be the global scope. In the global scope, there is a variable named "month", so the value of that variable is returned, and it is "july".

So, "foo()" outputs: "august" and "bar()" outputs: "july", because, although they both look for a variable named "month", they find completely different values; in their respective scope chains, the value of a variable named "month" differs.

Example # 3

```
month = 'july'; //global variable

function foo(){
    var month = "august"; //private
    window.season = 'summer'; //global
    console.log(month);
};

console.log(month); //in the global scope, month = july

foo(); //in the local scope of the foo function, month = august

console.log(season); //in the global scope, season = summer
```

CODE

Here is the jsFiddle.net link for Example # 3 : <http://jsfiddle.net/ZaXxk/>

In Example # 3, notice a new statement in the "foo()" function: "window.season".

In this statement, we are creating a global variable named "season". Although we are within the context of the "foo()" function, we can reference the global scope by mutating the "window" object. "window" is essentially the global object. Creating a global variable while inside of the local scope of "foo()" is easily done by adding a property to the "window" object; in our case we name it "season" and give it a value of "summer".

So once again, although we are in the local scope of "foo()", we have just created a global variable named "season" and given it the value of "summer", by adding a new property to the "window" object (e.g., window.season = 'summer').

Example # 4

CODE

```

month = 'july'; //global variable

function foo(){
    var month = "august"; //private
    window.season = 'summer'; //global
    weather = 'hot'; //global
    console.log(month);
};

console.log(month); //in the global scope, month = july

foo(); //in the local scope of the foo function, month = august

console.log(season); //in the global scope, season = summer

console.log(weather); //in the global scope, weather = hot

```

Here is the jsFiddle.net link for Example # 4 : <http://jsfiddle.net/XYu4x/>

In Example # 4, we create another global variable while within the local scope of "foo()", named "weather". This approach is different because when we say: weather = "hot", the absence of the "var" keyword automatically makes the variable global. This is important to remember and make note of: If you omit the "var" keyword when creating a variable, no matter where you do it, that variable becomes global.

In general, this is something that you want to avoid, as it can lead to code that is hard to understand and even harder to maintain. But for the purpose of this discussion, it illustrates an important behavior in JavaScript: omitting the "var" keyword when creating a variable makes that variable global, no matter where you do it. I'm repeating this because it is an important detail to remember.

Example # 5

CODE

```

month = 'july'; //global variable

function foo(){
    var month = "august"; //private
    window.season = 'summer'; //global
    weather = 'hot'; //global
    this.activity = 'swimming' //global
    console.log(month);
};

console.log(month); //in the global scope, month = july

foo(); //in the local scope of the foo function, month = august

console.log(season); //in the global scope, season = summer

console.log(weather); //in the global scope, weather = hot

console.log(activity); //in the global scope, activity = swimming

```

Here is the jsFiddle.net link for Example # 5 : <http://jsfiddle.net/NTjYe/>

In Example # 5, we yet again create a new global variable from within the local scope of "foo()". This variable is

named "activity". We demonstrate yet another way in which you can do this by saying: `this.activity = 'swimming'`. This introduces another concept: the meaning of "this" inside a function (no pun intended).

Inside a function, the "this" keyword refers to the context in which the function is executed. In our example, `"foo()"` is executed in the context of the global object, so "this" refers to the global object, which means that `"this.activity"` adds a property to the global object named "activity".

Make note: While `"foo()"` has its own "local" scope, the keyword "this" refers to the context in which `"foo()"` is executed. This is another important detail to remember. It will come up a lot when writing more advanced JavaScript.

Another (very) important note: An "implied global" is what occurs when you omit the "var" keyword when declaring a variable. There is a subtle, yet important difference between that and a variable created using the "var" keyword in the global scope: an implied global is actually not a variable; it is a property of the "window" object. For the most part, it behaves very much like a global variable, but there are differences: for example, you cannot delete a global variable, but you can delete a property of the window object. This is a topic worth looking into when you have time, and at minimum, good to be aware of.

Summary

At first, the concept of scope in JavaScript can be a challenge to fully understand. But it is very much worth the effort. Once you understand scope, the JavaScript language becomes a sharp knife that can be used to sculpt elegant and expressive code. This tutorial discussed only the most basic concept of scope in JavaScript. I highly recommend exploring it in-depth.

88. Understanding JavaScript Function Invocation and "this"

Over the years, I've seen a lot of confusion about JavaScript function invocation. In particular, a lot of people have complained that the semantics of `this` in function invocations is confusing.

In my opinion, a lot of this confusion is cleared up by understanding the core function invocation primitive, and then looking at all other ways of invoking a function as sugar on top of that primitive. In fact, this is exactly how the ECMAScript spec thinks about it. In some areas, this post is a simplification of the spec, but the basic idea is the same.

The Core Primitive

First, let's look at the core function invocation primitive, a Function's `call` method[1]. The `call` method is relatively straight forward.

1. Make an argument list (`argList`) out of parameters 1 through the end
2. The first parameter is `thisValue`
3. Invoke the function with `this` set to `thisValue` and the `argList` as its argument list

For example:

```
function hello(thing) {
  console.log(this + " says hello " + thing);
}

hello.call("Yehuda", "world") //=> Yehuda says hello world
```

CODE

As you can see, we invoked the `hello` method with `this` set to "Yehuda" and a single argument "world". This is the core primitive of JavaScript function invocation. You can think of all other function calls as desugaring to this primitive. (to "desugar" is to take a convenient syntax and describe it in terms of a more basic core primitive).

[1] In [the ES5 spec](#), the `call` method is described in terms of another, more low level primitive, but it's a very thin wrapper on top of that primitive, so I'm simplifying a bit here. See the end of this post for more information.

Simple Function Invocation

Obviously, invoking functions with `call` all the time would be pretty annoying. JavaScript allows us to invoke functions directly using the parens syntax (`hello("world")`). When we do that, the invocation desugars:

```
function hello(thing) {
  console.log("Hello " + thing);
}

// this:
hello("world")

// desugars to:
hello.call(window, "world");
```

CODE

This behavior has changed in ECMAScript 5 **only when using strict mode**[2]:

```
// this:
hello("world")

// desugars to:
hello.call(undefined, "world");
```

CODE

The short version is: **a function invocation like `fn(...args)` is the same as `fn.call(window [ES5-strict: undefined], ...args)`**.

Note that this is also true about functions declared inline: `(function() {})()` is the same as `(function() {}).call(window [ES5-strict: undefined])`.

[2] Actually, I lied a bit. The ECMAScript 5 spec says that `undefined` is (almost) always passed, but that the function being called should change its `thisValue` to the global object when not in strict mode. This allows strict mode callers to avoid breaking existing non-strict-mode libraries.

Member Functions

The next very common way to invoke a method is as a member of an object (`person.hello()`). In this case, the invocation desugars:

CODE

```

var person = {
  name: "Brendan Eich",
  hello: function(thing) {
    console.log(this + " says hello " + thing);
  }
}

// this:
person.hello("world")

// desugars to this:
person.hello.call(person, "world");

```

Note that it doesn't matter how the `hello` method becomes attached to the object in this form. Remember that we previously defined `hello` as a standalone function. Let's see what happens if we attach it to the object dynamically:

CODE

```

function hello(thing) {
  console.log(this + " says hello " + thing);
}

person = { name: "Brendan Eich" }
person.hello = hello;

person.hello("world") // still desugars to person.hello.call(person, "world")

hello("world") // "[object DOMWindow]world"

```

Notice that the function doesn't have a persistent notion of its 'this'. It is always set at call time based upon the way it was invoked by its caller.

Using `Function.prototype.bind`

Because it can sometimes be convenient to have a reference to a function with a persistent `this` value, people have historically used a simple closure trick to convert a function into one with an unchanging `this`:

CODE

```

var person = {
  name: "Brendan Eich",
  hello: function(thing) {
    console.log(this.name + " says hello " + thing);
  }
}

var boundHello = function(thing) { return person.hello.call(person, thing); }

boundHello("world");

```

Even though our `boundHello` call still desugars to `boundHello.call(window, "world")`, we turn right around and use our primitive `call` method to change the `this` value back to what we want it to be.

We can make this trick general-purpose with a few tweaks:

CODE

```

var bind = function(func, thisValue) {
  return function() {
    return func.apply(thisValue, arguments);
  }
}

var boundHello = bind(person.hello, person);
boundHello("world") // "Brendan Eich says hello world"

```

In order to understand this, you just need two more pieces of information. First, `arguments` is an Array-like object that represents all of the arguments passed into a function. Second, the `apply` method works exactly like the `call` primitive, except that it takes an Array-like object instead of listing the arguments out one at a time.

Our `bind` method simply returns a new function. When it is invoked, our new function simply invokes the original function that was passed in, setting the original value as `this`. It also passes through the arguments.

Because this was a somewhat common idiom, ES5 introduced a new method `bind` on all `Function` objects that implements this behavior:

CODE

```

var boundHello = person.hello.bind(person);
boundHello("world") // "Brendan Eich says hello world"

```

This is most useful when you need a raw function to pass as a callback:

CODE

```

var person = {
  name: "Alex Russell",
  hello: function() { console.log(this.name + " says hello world"); }
}

$("#some-div").click(person.hello.bind(person));

// when the div is clicked, "Alex Russell says hello world" is printed

```

This is, of course, somewhat clunky, and TC39 (the committee that works on the next version(s) of ECMAScript) continues to work on a more elegant, still-backwards-compatible solution.

On jQuery

Because jQuery makes such heavy use of anonymous callback functions, it uses the `call` method internally to set the `this` value of those callbacks to a more useful value. For instance, instead of receiving `window` as `this` in all event handlers (as you would without special intervention), jQuery invokes `call` on the callback with the element that set up the event handler as its first parameter.

This is extremely useful, because the default value of `this` in anonymous callbacks is not particularly useful, but it can give beginners to JavaScript the impression that `this` is, **in general** a strange, often mutated concept that is hard to reason about.

If you understand the basic rules for converting a sugary function call into a desugared `func.call(thisValue, ...args)`, you should be able to navigate the not so treacherous waters of the JavaScript `this` value.

PS: I Cheated

In several places, I simplified the reality a bit from the exact wording of the specification. Probably the most important cheat is the way I called `func.call` a "primitive". In reality, the spec has a primitive (internally referred to as `[[Call]]`) that both `func.call` and `[obj.]func()` use.

However, take a look at the definition of `func.call`:

1. If `IsCallable(func)` is false, then throw a `TypeError` exception.
2. Let `argList` be an empty List.
3. If this method was called with more than one argument then in left to right order starting with `arg1` append each argument as the last element of `argList`
4. Return the result of calling the `[[Call]]` internal method of `func`, providing `thisArg` as the `this` value and `argList` as the list of arguments.

As you can see, this definition is essentially a very simple JavaScript language binding to the primitive `[[Call]]` operation.

If you look at the definition of invoking a function, the first seven steps set up `thisValue` and `argList`, and the last step is: "Return the result of calling the `[[Call]]` internal method on `func`, providing `thisValue` as the `this` value and providing the list `argList` as the argument values."

It's essentially identical wording, once the `argList` and `thisValue` have been determined.

I cheated a bit in calling `call` a primitive, but the meaning is essentially the same as had I pulled out the spec at the beginning of this article and quoted chapter and verse.

There are also some additional cases (most notably involving `with`) that I didn't cover here.

89. Everything you wanted to know about JavaScript scope

The JavaScript language has a few concepts of "scope", none of which are straightforward or easy to understand as a new JavaScript developer (and even some experienced JavaScript developers). This post is aimed at those wanting to learn about the many depths of JavaScript after hearing words such as `scope`, `closure`, `this`, `namespace`, `function scope`, `global scope`, `lexical scope` and `public/private scope`. Hopefully by reading this post you'll know the answers to:

- What is Scope?
- What is Global/Local Scope?
- What is a Namespace and how does it differ to Scope?
- What is the `this` keyword and how does Scope affect it?
- What is Function/Lexical Scope?
- What are Closures?
- What is Public/Private Scope?
- How can I understand/create/do all of the above?

What is Scope?

In JavaScript, scope refers to the current context of your code. Scopes can be *globally* or *locally* defined.

Understanding JavaScript scope is key to writing bulletproof code and being a better developer. You'll understand where variables/functions are accessible, be able to change the scope of your code's context and be able to write faster and more maintainable code, as well as debug much faster.

Thinking about scope is easy, are we inside Scope A or Scope B ?

What is Global Scope?

Before you write a line of JavaScript, you're in what we call the `Global Scope`. If we declare a variable, it's defined globally:

```
// global scope
var name = 'Todd';
```

CODE

Global scope is your best friend and your worst nightmare, learning to control your scopes is easy and in doing so, you'll run into no issues with global scope problems (usually namespace clashes). You'll often hear people saying "Global Scope is *bad*", but never really justifying as to *why*. Global scope isn't bad, you need it to create Modules/APIs that are accessible across scopes, you must use it to your advantage and not cause issues.

Everyone's used jQuery before, as soon as you do this...

```
jQuery('.myClass');
```

CODE

... we're accessing jQuery in *global* scope, we can refer to this access as the `namespace`. The namespace is sometimes an interchangeable word for scope, but usually the refers to the highest level scope. In this case, `jQuery` is in the global scope, and is also our namespace. The `jQuery` namespace is defined in the global scope, which acts as a namespace for the jQuery library as everything inside it becomes a descendent of that namespace.

What is Local Scope?

A local scope refers to any scope defined past the global scope. There is typically one global scope, and each function defined has its own (nested) local scope. Any function defined within another function has a local scope which is linked to the outer function.

If I define a function and create variables inside it, those variables becomes locally scoped. Take this example:

```
// Scope A: Global scope out here
var myFunction = function () {
    // Scope B: Local scope in here
};
```

CODE

Any locally scoped items are not visible in the global scope - *unless* exposed, meaning if I define functions or variables within a new scope, it's inaccessible *outside* of that current scope. A simple example of this is the following:

```

var myFunction = function () {
    var name = 'Todd';
    console.log(name); // Todd
};

// Uncaught ReferenceError: name is not defined
console.log(name);

```

CODE

The variable `name` is scoped locally, it isn't exposed to the parent scope and therefore undefined.

Function scope

All scopes in JavaScript are created with `Function Scope only`, they aren't created by `for` or `while` loops or expression statements like `if` or `switch`. New functions = new scope - that's the rule. A simple example to demonstrate this scope creation:

```

// Scope A
var myFunction = function () {
    // Scope B
    var myOtherFunction = function () {
        // Scope C
    };
};

```

CODE

It's easy to create new scope and create local variables/functions/objects.

Lexical Scope

Whenever you see a function within another function, the inner function has access to the scope in the outer function, this is called Lexical Scope or Closure - also referred to as Static Scope. The easiest way to demonstrate that again:

```

// Scope A
var myFunction = function () {
    // Scope B
    var name = 'Todd'; // defined in Scope B
    var myOtherFunction = function () {
        // Scope C: `name` is accessible here!
    };
};

```

CODE

You'll notice that `myOtherFunction` *isn't* being called here, it's simply defined. It's order of call also has effect on how the scoped variables react, here I've defined my function and called it *under* another `console` statement:

```

var myFunction = function () {
    var name = 'Todd';
    var myOtherFunction = function () {
        console.log('My name is ' + name);
    };
    console.log(name);
    myOtherFunction(); // call function
};

// Will then log out:
// `Todd`
// `My name is Todd`
```

CODE

Lexical scope is easy to work with, *any* variables/objects/functions defined in *its* parent scope, are available in the scope chain. For example:

```

var name = 'Todd';
var scope1 = function () {
    // name is available here
    var scope2 = function () {
        // name is available here too
        var scope3 = function () {
            // name is also available here!
        };
    };
};
};
```

CODE

The only important thing to remember is that Lexical scope does *not* work backwards. Here we can see how Lexical scope *doesn't* work:

```

// name = undefined
var scope1 = function () {
    // name = undefined
    var scope2 = function () {
        // name = undefined
        var scope3 = function () {
            var name = 'Todd'; // locally scoped
        };
    };
};
};
```

CODE

I can always return a reference to `name`, but never the variable itself.

Scope Chain

Scope chains establish the scope for a given function. Each function defined has its own nested scope as we know, and any function defined within another function has a local scope which is linked to the outer function - this link is called the chain. It's always the *position* in the code that defines the scope. When resolving a variable, JavaScript starts at the innermost scope and searches outwards until it finds the variable/object/function it was looking for.

Closures

Closures ties in very closely with Lexical Scope. A better example of how the *closure* side of things works, can be seen when returning a *function reference* - a more practical usage. Inside our scope, we can return things so that they're available in the parent scope:

```
CODE
var sayHello = function (name) {
  var text = 'Hello, ' + name;
  return function () {
    console.log(text);
  };
};
```

The `closure` concept we've used here makes our scope inside `sayHello` inaccessible to the public scope. Calling the function alone will do nothing as it *returns* a function:

```
CODE
sayHello('Todd'); // nothing happens, no errors, just silence...
```

The function returns a function, which means it needs assignment, and *then* calling:

```
CODE
var helloTodd = sayHello('Todd');
helloTodd(); // will call the closure and log 'Hello, Todd'
```

Okay, I lied, you *can* call it, and you may have seen functions like this, but this will call your closure:

```
CODE
sayHello('Bob')(); // calls the returned function without assignment
```

AngularJS uses the above technique for its `$compile` method, where you pass the current scope reference into the closure:

```
CODE
$compile(template)(scope);
```

Meaning we could guess that their code would (over-simplified) look like this:

```
CODE
var $compile = function (template) {
  // some magic stuff here
  // scope is out of scope, though...
  return function (scope) {
    // access to `template` and `scope` to do magic with too
  };
};
```

A function doesn't *have* to return in order to be called a closure though. Simply accessing variables outside of the immediate lexical scope creates a closure.

Scope and 'this'

Each scope binds a different value of `this` depending on how the function is invoked. We've all used the `this` keyword, but not all of us understand it and how it differs when invoked. By default `this` refers to the outer most global object, the `window`. We can easily show how invoking functions in different ways binds the `this` value differently:

```
CODE
var myFunction = function () {
    console.log(this); // this = global, [object Window]
};

myFunction();

var myObject = {};
myObject.myMethod = function () {
    console.log(this); // this = Object { myObject }
};

var nav = document.querySelector('.nav'); // <nav class="nav">
var toggleNav = function () {
    console.log(this); // this = <nav> element
};

nav.addEventListener('click', toggleNav, false);
```

There are also problems that we run into when dealing with the `this` value, for instance if I do this, even inside the same function the scope can be changed and the `this` value can be changed:

```
CODE
var nav = document.querySelector('.nav'); // <nav class="nav">
var toggleNav = function () {
    console.log(this); // <nav> element
    setTimeout(function () {
        console.log(this); // [object Window]
    }, 1000);
};

nav.addEventListener('click', toggleNav, false);
```

So what's happened here? We've created new scope which is not invoked from our event handler, so it defaults to the `window` Object as expected. There are several things we can do if we want to access the proper `this` value which isn't affected by the new scope. You might have seen this before, where we can cache a reference to the `this` value using a `that` variable and refer to the lexical binding:

```
CODE
var nav = document.querySelector('.nav'); // <nav class="nav">
var toggleNav = function () {
    var that = this;
    console.log(that); // <nav> element
    setTimeout(function () {
        console.log(that); // <nav> element
    }, 1000);
};

nav.addEventListener('click', toggleNav, false);
```

This is a neat little trick to be able to use the proper `this` value and resolve problems with newly created scope.

Changing scope with .call(), .apply() and .bind()

Sometimes you need to manipulate the scopes of your JavaScript depending on what you're looking to do. A simple demonstration of how to change the scope when looping:

```
CODE
var links = document.querySelectorAll('nav li');
for (var i = 0; i < links.length; i++) {
    console.log(this); // [object Window]
}
```

The `this` value here doesn't refer to our elements, we're not invoking anything or changing the scope. Let's look at how we can change scope (well, it looks like we change scope, but what we're really doing is changing the `context` of how the function is called).

.call() and .apply()

The `.call()` and `.apply()` methods are really sweet, they allows you to pass in a scope to a function, which binds the correct `this` value. Let's manipulate the above function to make it so that our `this` value is each element in the array:

```
CODE
var links = document.querySelectorAll('nav li');
for (var i = 0; i < links.length; i++) {
    (function () {
        console.log(this);
    }).call(links[i]);
}
```

You can see I'm passing in the current element in the Array iteration, `links[i]`, which changes the scope of the function so that the `this` value becomes that iterated element. We can then use the `this` binding if we wanted. We can use either `.call()` or `.apply()` to change the scope, but any further arguments are where the two differ: `.call(scope, arg1, arg2, arg3)` takes individual arguments, comma separated, whereas `.apply(scope, [arg1, arg2])` takes an Array of arguments.

It's important to remember that using `.call()` or `.apply()` actually invokes your function, so instead of doing this:

```
CODE
myFunction(); // invoke myFunction
```

You'll let `.call()` handle it and chain the method:

```
CODE
myFunction.call(scope); // invoke myFunction using .call()
```

.bind()

Unlike the above, using `.bind()` does not *invoke* a function, it merely binds the values before the function is invoked. It's a real shame this was introduced in ECMAScript 5 and not earlier as this method is fantastic. As you know we can't pass parameters into function references, something like this:

```
// works
nav.addEventListener('click', toggleNav, false);

// will invoke the function immediately
nav.addEventListener('click', toggleNav(arg1, arg2), false);
```

CODE

We can fix this, by creating a new function inside it:

```
nav.addEventListener('click', function () {
    toggleNav(arg1, arg2);
}, false);
```

CODE

But again this changes scope and we're creating a needless function again, which will be costly on performance if we were inside a loop and binding event listeners. This is where `.bind()` shines through, as we can pass in arguments but the functions are not called:

```
nav.addEventListener('click', toggleNav.bind(scope, arg1, arg2), false);
```

CODE

The function isn't invoked, and the scope can be changed if needed, but arguments are sat waiting to be passed in.

Private and Public Scope

In many programming languages, you'll hear about `public` and `private` scope, in JavaScript there is no such thing. We can, however, emulate public and private scope through things like Closures.

By using JavaScript design patterns, such as the `Module` pattern for example, we can create `public` and `private` scope. A simple way to create private scope, is by wrapping our functions inside a function. As we've learned, functions create scope, which keeps things out of the global scope:

```
(function () {
    // private scope inside here
})();
```

CODE

We might then add a few functions for use in our app:

```
(function () {
    var myFunction = function () {
        // do some stuff here
    };
})();
```

CODE

But when we come to calling our function, it would be out of scope:

```
(function () {
    var myFunction = function () {
        // do some stuff here
    };
})();

myFunction(); // Uncaught ReferenceError: myFunction is not defined
```

CODE

Success! We've created private scope. But what if I want the function to be public? There's a great pattern (called the Module Pattern [and Revealing Module Pattern]) which allows us to scope our functions correctly, using private and public scope and an `Object`. Here I grab my global namespace, called `Module`, which contains all of my relevant code for that module:

```
// define module
var Module = (function () {
    return {
        myMethod: function () {
            console.log('myMethod has been called.');
        }
    };
})();

// call module + methods
Module.myMethod();
```

CODE

The `return` statement here is what returns our `public` methods, which are accessible in the global scope - *but* are namespaced. This means our `Module` takes care of our namespace, and can contain as many methods as we want. We can extend the `Module` as we wish:

```
// define module
var Module = (function () {
    return {
        myMethod: function () {

        },
        someOtherMethod: function () {

        }
    };
})();

// call module + methods
Module.myMethod();
Module.someOtherMethod();
```

CODE

So what about private methods? This is where a lot of developers go wrong and pollute the global namespace by dumping all their functions in the global scope. Functions that help our code *work* do not need to be in the global scope, only the API calls do - things that *need* to be accessed globally in order to work. Here's how we can create private scope, by *not* returning functions:

```
var Module = (function () {
    var privateMethod = function () {

    };
    return {
        publicMethod: function () {

        }
    };
})();
```

CODE

This means that `publicMethod` can be called, but `privateMethod` cannot, as it's privately scoped! These privately scoped functions are things like helpers, `addClass`, `removeClass`, Ajax/XHR calls, Arrays, Objects, anything you can think of.

Here's an interesting twist though, anything in the same scope has access to anything in the same scope, even *after* the function has been returned. Which means, our `public` methods have *access* to our `private` ones, so they can still interact but are unaccessible in the global scope.

```
var Module = (function () {
    var privateMethod = function () {

    };
    return {
        publicMethod: function () {
            // has access to `privateMethod`, we can call it:
            // privateMethod();
        }
    };
})();
```

CODE

This allows a very powerful level of interactivity, as well as code security. A very important part of JavaScript is ensuring security, which is exactly *why* we can't afford to put all functions in the global scope as they'll be publicly available, which makes them open to vulnerable attacks.

Here's an example of returning an Object, making use of `public` and `private` methods:

```

var Module = (function () {
    var myModule = {};
    var privateMethod = function () {

    };
    myModule.publicMethod = function () {

    };
    myModule.anotherPublicMethod = function () {

    };
    return myModule; // returns the Object with public methods
})();

// usage
Module.publicMethod();

```

CODE

One neat naming convention is to begin `private` methods with an underscore, which visually helps you differentiate between public and private:

```

var Module = (function () {
    var _privateMethod = function () {

    };
    var publicMethod = function () {

    };
})();

```

CODE

This helps us when returning an anonymous `Object`, which the Module can use in Object fashion as we can simply assign the function references:

```

var Module = (function () {
    var _privateMethod = function () {

    };
    var publicMethod = function () {

    };
    return {
        publicMethod: publicMethod,
        anotherPublicMethod: anotherPublicMethod
    }
})();

```

CODE

Happy scoping!

90. JavaScript Context, Call and Bind – Ninja Level

In my [previous article](#) I showed the differences between scope and context, basic problems that arise and how to fix them. If you are just using some JavaScript and maybe jQuery, an understanding of scope is all that is needed to get

you by. Once you start using objects or namespaces however, you'll start to run into issues with context and will need to use the keyword `this`. But when you get into object oriented JavaScript, you'll need an advanced understanding of context and how to make it work for you. To do this, we'll use the `call()` and `apply()` methods, and then a backwards compatible version of the new feature Mozilla recently released in JavaScript 1.8.5 called [bind\(\)](#).

Who the Hell Would Ever Use this F@#ing Language Anyway?

The title of this section is dedicated to a one-time Club AJAX attendee who looked at the next example and was so frustrated at how little he understood it, he started blurting out obscenities. Needless to say, he is now sentenced to life, doing Java.

So, to paraphrase Jack Nicholson... wait until you get a load of this:

```
var o = {
  x:10,
  onTimeout: function(){
    console.log("x:", this.x); // outputs undefined
  },
  m: function(){
    setTimeout(this.onTimeout, 1); // not working?
  }
}
o.m();

//Example #1 - Advanced Broken Context
```

CODE

"Wait a minute!" I hear you say. "I applied the context to the function, and `onTimeout` fired -- why didn't `this.x` resolve? And who the hell would ever use this language, anyway?"

I empathize, really. But remember, you don't know how to paint without knowing about the color red, and you can't really say you know JavaScript unless you know about context and all of its pitfalls. When you are able to understand this concept, it will elevate you from just a guy who writes some JavaScript to outright NINJA!

Before we can decipher why the previous example doesn't work, let's change it up so we can better break down the issues.

```
var o = {
  x:10,
  onTimeout: function(){
    console.log("x:", this.x); // outputs undefined
  },
  otherMethod: function(f){
    f();
  },
  m: function(){
    this.otherMethod(this.onTimeout);
  }
}
o.m();

//Example #2 - Advanced Broken Context Variation
```

CODE

Example #2 basically substitutes `otherMethod` for `setTimeout`, so we can side step that particular nuance for now.

This example shows a similar situation: we are passing a method to another method and trying to invoke it. However, `this.x` is still undefined. The reason for this is kind of tricky.

Yes, we used `this` with `onTimeout`. But we only used `this` in order to *find* `onTimeout`, we didn't use it to *invoke* `onTimeout`. If we didn't use `this`, we would get an error saying the function `onTimeout` is not found, which makes sense, because that would indicate we want a function (as in, `var onTimeout();`), but what we really want is a method. So we use `this`, to get the method, and pass it to `this.otherMethod`.

Refresher: Methods are associated to objects like: `object.myMethod = function(){}`; while functions are not: `var myFunc = function(){}`.

The Reason

Ironically, if `m()` or `otherMethod()` invoked `this.onTimeout`, it would work. But when we retrieved the method we sort of "extracted it" from the object, and passed it as a **variable**, of a **function** if you will. It's no longer a *method*, it's just a function with no call object bound to it. So we need to re-bind `this`, which we can do with the `call()` method:

```
var o = {
  x:10,
  onTimeout: function(){
    console.log("x:", this.x); // outputs 10! Fixed!
  },
  otherMethod: function(f){
    f.call(this); // bound with call()
  },
  m: function(){
    this.otherMethod(this.onTimeout);
  }
}
o.m();

//Example #3 - Advanced Broken Context Variation Fixed
```

CODE

NOTE: The ECMAScript documentation refers to a method's bound object as its *activation* object. David Flanagan, author of [JavaScript: The Definitive Guide](#) refers to it as a *call* object, which I like, because you can assign context with a function's `call` method.

Is "this" Broken?

We fixed it, but I sense dissatisfaction in you, young Skywalker. "*Why should I jump through these hoops? This language is broken!*" I agree that this aspect is at the least confusing and seems just plain wrong. But there is a reason for this behavior, believe it or not. Remember, JavaScript functions are just data and can be passed around as such.

Okay, if you don't believe me, perhaps you'll believe Brendan Eich from [this Talk](#):

[The reason for the keyword "this"] is I wanted functions, which are the main idea in JavaScript, to also be methods; and for them to be methods, they had to have a receiving object who's property was the value of that function.

That explanation was a bit... recursive. Perhaps a simple exercise will help. In Example #4, one object copies a method from another. What should be logged, 10 or 20?

```
var o = {
  x:10,
  m: function(){
    console.log(this.x); // what is this.x?
  }
};
var o2 = {
  x:20
};
o2.m = o.m; // copying methods
o2.m();
```

//Example #4 - Copied Method

CODE

If you answered 20, you'd be correct. Even though `o.m` was copied as data, it was still invoked on the last line with the `o2` object, which set its context. But you may see what the issue would be if a copied method carried its context around. Then the answer would be 10, derived from the original object. That would make JavaScript look even more broken.

The following is another example of the use of `call()`. You can borrow a method from one object and apply it to another object temporarily:

```
var o = {
  x:1,
  y:2,
  changeProps:function(n){
    this.x += n;
    this.y += n;
    console.log(this.x, this.y);
  }
};
var o2 = {
  x:10,
  y:20
};
o.changeProps(1); // outputs 2, 3 (as normal)
o.changeProps.call(o2, 1); // outputs 11, 21 (utilizing object o2)
```

//Example #5 - Borrowing a Method from another Object

CODE

In Example #5, `o.changeProps(1);` invokes as normal, but the last line, `o.changeProps.call(o2, 1);` calls that same method, yet invokes it within a different object. Admittedly, this example is a bit contrived and is probably making you squirm from all the best practices it violates. But it does show the versatility of the language and exactly how `call` works.

Note: The first argument of `call()` should be an activation object and all arguments after that will be passed to the method. `apply()` works just like `call()` except the second argument should be an array and is often used to pass the arguments object.

Now that we have a better understanding of context and how to fix it, we can readdress the problem in Example #1.

However, we can't use `call()` to fix the context, because that would have to happen within the `setTimeout` function and we can't change native code. Well, we *could*, but that would be like trying to waterproof a screen with a toothpick. Because the same problem will arise again and again, in callbacks, connections, and myriad other places. We have a better way.

Bind

The problem can be fixed with the new `bind` method in JavaScript 1.8.5. And in spite of the complexity of the problem, the solution is really quite simple:

```
var o = {
  x:10,
  onTimeout: function(){
    console.log("x:", this.x); // outputs 10! yay!
  },
  m: function(){
    setTimeout(this.onTimeout.bind(this), 1); // bind applied
  }
}
o.m();

//Example #6 - Advanced Broken Context Fixed
```

CODE

Now that `this` is bound `this.setTimeout`, we can pass it around and it won't lose its context. And that's it! Fixed! Good night nurse!

What's that? Does this work in **what** other browser? Hm, I promised some sort of ninja status, didn't I? That would necessitate some sort of explanation.

Bind Under the Hood

First of all, no, this does not work in *that* browser, but we can home brew a utility method that does, which we will add it to the prototype object of the Function constructor. Ever added the `forEach` method to Arrays? We're doing the same thing here except with the Function's prototype.

What we need to do is take our existing function and the context passed, and create a newly bound function. We are not going to return the bound function directly because if we did, it would fire immediately, which we don't want. We are passing the bound function as data to be invoked later. Therefore we need to return *another* function that has the bound function inside it.

```
Function.prototype.bind = function (context) {
  var fn = this; // correlates to this.onTimeout
  return function(){ // what gets passed: an anonymous function
    // when invoked, our original function, this.onTimeout,
    // will have the proper context applied
    return fn.apply(context, arguments);
  }
}

//Example #7 - Simple Bind Prototype
```

CODE

You'll notice that the function is bound with `apply()` instead of `call()`. This is because our utility code has no idea how many arguments may be passed, and `apply()` handles a variable amount... of variables.

Include the *Simple Bind Prototype* early within your code, and Example #6 will work -- in all browsers.

But Extending Native Objects is Bad

Oh, so you're one of those purists who believes you shouldn't mess with native objects eh? That's okay, I'm concerned about that too. The tide is shifting a bit in the JavaScript library developers community that [it's okay to extend native objects](#) (not host objects) -- as long as those extensions are to fix gaps in weaker browsers with standardized functionality. And of course if it's your code, you can do whatever you want!

And... what you want is to **not** extend the native object? That's actually a good thing here, because the wrapper-implementation is a little more clear in how it works as there isn't the confusion of what "this" is, as in Example #7.

```
var bind = function(method, context){  
    return function(){  
        method.apply(context, arguments);  
    }  
}  
  
//Example #8 - Simple Bind
```

CODE

If using the wrapper implementation, you would change the code from `setTimeout(this.onTimeout.bind(this), 1);` to:
`setTimeout(bind(this.onTimeout, this), 1);`

The `bind` method passes the argument object, which is quite handy on callbacks. The following example is a recreation of a callback mechanism, which demonstrates how variables are passed:

```

var Loader = function(args){
  // simulate load with setTimeout
  setTimeout(function(){
    // fire the callback on "load"
    args.callback(2, 3);
  },1);
}

var o2 = {
  x:10,
  onLoad: function(y, z){
    // verify context with this.x, and passed vars, x and y
    console.log('onLoad', this.x, y, z); // ---> outputs 10, 2, 3
  },
  load: function(){
    // create our Loader object
    new Loader({
      // our args object only contains a callback - bound to "this".
      callback: this.onLoad.bind(this)
    });
  }
}
o2.load();

//Example #9 - Passing Variables

```

The Mozilla Implementation

You may have taken a peak at the [Mozilla bind page](#), which would be only natural since I linked to it. I didn't recreate their functionality because it's a bit complex. Our solution handles 95% of what you would ever do with `bind()`, while theirs handles new concepts like binding to instance constructors and Arrays. But feel free to experiment!

Conclusion

Congratulations. If you made it this far into the blog and haven't reached for a bottle of aspirin, bottle of alcohol, or a noose, that means you've managed to absorb one of the hardest concepts in JavaScript. Context is the key differentiator between JavaScript and all the other major languages.

91. Object Oriented Programming in JavaScript (Extending / Inherit classes)

As you probably know JavaScript is not exactly OOP based language. Of course there are some ways to handle with this and you can still create classes and inherit them. It is much much better to use classes. Your application will be well structured and split to modules.

The first thing that we have to do is to create a class. We will start with the constructor:

CODE

```
var BaseClass = function() {
    this.name = "I'm BaseClass";
};
```

Nothing special, just declaring of a function. The body of the class is constructed by using `.prototype` property of our variable.

CODE

```
var BaseClass = function() {
    this.name = "I'm BaseClass";
};
BaseClass.prototype = {
    getName:function() {
        return this.name;
    },
    setName:function(str) {
        this.name = str;
    }
};
```

So far we have:

- a) a class that is called `BaseClass`
 - b) a public property called "name"
 - c) two public methods called "getName" and "setName" that will get/set the "name" property
- And the usage of the class:

CODE

```
var base = new BaseClass();
alert(base.getName());
```

The interesting part is to make possible the inheritance of the class. We will create a small function that will do that for us every time when we need to extend the methods and the properties of some class.

CODE

```
function extend(ChildClass, ParentClass) {
    ChildClass.prototype = new ParentClass();
    ChildClass.prototype.constructor = ChildClass;
}
```

It's just two lines, but it does its job. The first one copies all the properties and methods including the constructor. Which means that if we create an object by "ChildClass" we will use the constructor of "ParentClass". In the second line we are restoring the constructor.

Let's see some examples of inheritance:

```

/* extending */
function extend(ChildClass, ParentClass) {
    ChildClass.prototype = new ParentClass();
    ChildClass.prototype.constructor = ChildClass;
}
/* base class */
var BaseClass = function() {
    this.name = "I'm BaseClass";
};
BaseClass.prototype = {
    getName:function() {
        return this.name;
    },
    setName:function(str) {
        this.name = str;
    }
};
/* sub class 1 */
var SubClass1 = function() {
    this.setName("I'm SubClass1");
}
/* sub class 2 */
var SubClass2 = function() {
    this.setName("I'm SubClass2");
}
extend(SubClass1, BaseClass);
extend(SubClass2, BaseClass);
var base = new BaseClass();
var sub1 = new SubClass1();
var sub2 = new SubClass2();
alert(base.getName());
alert(sub1.getName());
alert(sub2.getName());

```

The result of this script is "I'm base class", "I'm SubClass1" and at the end "I'm SubClass2". As you can see we are creating objects of "SubClass1" and "SubClass2" that extend the BaseClass. That's why they have public methods "setName" and "getName".

92. JavaScript Function Invocation

93. JavaScript Function Invocation

[« Previous](#)

[Next Chapter »](#)

JavaScript functions can be invoked in 4 different ways.

Each method differs in how **this** is initialized.

The **this** Keyword

In JavaScript, the thing called **this**, is the object that "owns" the current code.

The value of this, when used in a function, is the object that "owns" the function.



Note that **this** is not a variable. It is a keyword. You cannot change the value of **this**.

Invoking a JavaScript Function

You have already learned that the code inside a JavaScript function will execute when "something" invokes it.

The code in a function is not executed when the function is **defined**. It is executed when the function is **invoked**.

Some people use the term "**call a function**" instead of "**invoke a function**".

It is also quite common to say "call upon a function", "start a function", or "execute a function".

In this tutorial, we will use **invoke**, because a JavaScript function can be invoked without being called.

Invoking a Function as a Function

Example

```
function myFunction(a, b) {  
    return a * b;  
}  
myFunction(10, 2);      // myFunction(10, 2) will return 20  
Try it Yourself »
```

The function above does not belong to any object. But in JavaScript there is always a default global object.

In HTML the default global object is the HTML page itself, so the function above "belongs" to the HTML page.

In a browser the page object is the browser window. The function above automatically becomes a window function.

`myFunction()` and `window.myFunction()` is the same function:

Example

```
function myFunction(a, b) {  
    return a * b;  
}  
window.myFunction(10, 2); // window.myFunction(10, 2) will also return 20  
Try it Yourself »
```

This is a common way to invoke a JavaScript function, but not a good practice in computer programming.



Global variables, methods, or functions can easily create name conflicts and bugs in the global object.

The Global Object

When a function is called without an owner object, the value of **this** becomes the global object.

In a web browser the global object is the browser window.

This example returns the window object as the value of **this**:

Example

```
function myFunction() {  
    return this;  
}  
myFunction();           // Will return the window object  
Try it Yourself »
```



Invoking a function as a global function, causes the value of **this** to be the global object. Using the window object as a variable can easily crash your program.

Invoking a Function as a Method

In JavaScript you can define function as object methods.

The following example creates an object (**myObject**), with two properties (**firstName** and **lastName**), and a method (**fullName**):

Example

```
var myObject = {  
    firstName:"John",  
    lastName: "Doe",  
    fullName: function () {  
        return this.firstName + " " + this.lastName;  
    }  
}  
myObject.fullName();      // Will return "John Doe"  
Try it Yourself »
```

The **fullName** method is a function. The function belongs to the object. **myObject** is the owner of the function.

The thing called **this**, is the object that "owns" the JavaScript code. In this case the value of **this** is **myObject**.

Test it! Change the **fullName** method to return the value of **this**:

Example

```
var myObject = {
  firstName:"John",
  lastName: "Doe",
  fullName: function () {
    return this;
  }
}
myObject.fullName();      // Will return [object Object] (the owner object)
```

[Try it Yourself »](#)



Invoking a function as an object method, causes the value of **this** to be the object itself.

Invoking a Function with a Function Constructor

If a function invocation is preceded with the **new** keyword, it is a constructor invocation.

It looks like you create a new function, but since JavaScript functions are objects you actually create a new object:

Example

// This is a function constructor:

```
function myFunction(arg1, arg2) {
  this.firstName = arg1;
  this.lastName = arg2;
}
```

// This creates a new object

```
var x = new myFunction("John","Doe");
x.firstName;           // Will return "John"
```

[Try it Yourself »](#)

A constructor invocation creates a new object. The new object inherits the properties and methods from its constructor.



The **this** keyword in the constructor does not have a value.
The value of **this** will be the new object created when the function is invoked.

Invoking a Function with a Function Method

In JavaScript, functions are objects. JavaScript functions have properties and methods.

call() and **apply()** are predefined JavaScript function methods. Both methods can be used to invoke a function, and both methods must have the owner object as first parameter.

Example

```
function myFunction(a, b) {
    return a * b;
}
myObject = myFunction.call(myObject, 10, 2); // Will return 20
Try it Yourself »
```

Example

```
function myFunction(a, b) {
    return a * b;
}
myArray = [10, 2];
myObject = myFunction.apply(myObject, myArray); // Will also return 20
Try it Yourself »
```

Both methods take an owner object as the first argument. The only difference is that call() takes the function arguments separately, and apply() takes the function arguments in an array.

In JavaScript strict mode, the first argument becomes the value of **this** in the invoked function, even if the argument is not an object.

In "non-strict" mode, if the value of the first argument is null or undefined, it is replaced with the global object.



With call() or apply() you can set the value of **this**, and invoke a function as a new method of an existing object.

94. Javascript call and apply

Javascript call and apply

09

2011.11

Category

- [Javascript](#)

Tags

[apply](#) [call](#) [javascript](#)

For a long time javascript call and apply have been really confusing me. I neither know when to use it nor what are the differences between these two.

Basically call and apply invoke the function and switch the function context with the first argument. The main difference between these two is with call the rest arguments are passed to the calling function and those

arguments have to be listed explicitly. However with `apply` the amount of arguments don't have to be predefined. we can pass in an array.

CODE

```
function fn( arg1, arg2,... ){
    // do something
}

fn( arg1, arg2,... );

fn.call( context, arg1, arg2,... );

fn.apply( context, [ arg1, arg2,... ]);
```

It's not necessary to have the second argument

A real world use case

Ajax is a good example to use `call` and `apply`.

CODE

```
function Album( id, title, owner_id ){
    this.id      = id;
    this.name    = title;
    this.owner_id = owner_id;
};

Album.prototype.get_owner = function( callback ){
    var self = this;

    $.get( '/owners/' + this.owner_id , function( data ){
        callback && callback.call( self, data.name );
    });
};

var album = new Album( 1, 'node.js conf', 2 );

album.get_owner( function( owner ){
    alert( 'The album' + this.name + ' belongs to ' + owner );
});
```

Speed

Function invoke with `call` is slightly faster than with `apply`. But don't worry, you really can't tell the difference, use whatever suits you.

95. The Difference Between Call and Apply in Javascript

One very common thing that trips me up when writing Javascript is knowing when to use `call` and when to use `apply`. If you're wondering what these methods are, or don't know how scope works in JavaScript, then it might

make sense to read the [Javascript Guide](#) first.

Let's look at some ways we might want to use them:

CODE

```
var person1 = {name: 'Marvin', age: 42, size: '2xM'};
var person2 = {name: 'Zaphod', age: 4200000000, size: '1xS'};

var sayHello = function(){
    alert('Hello, ' + this.name);
};

var sayGoodbye = function(){
    alert('Goodbye, ' + this.name);
};
```

Now if you've read the [guide](#), this example will look really familiar. You'd already know that writing the following code:

CODE

```
sayHello();
sayGoodbye();
```

will give errors (if you're lucky), or just unexpected results (if you aren't). This is because both functions rely on their scope for the `this.name` data, and calling them without explicit scope will just run them in the scope of the current window.

So how do we scope them? Try this:

CODE

```
sayHello.call(person1);
sayGoodbye.call(person2);

sayHello.apply(person1);
sayGoodbye.apply(person2);
```

All four of these lines do exactly the same thing. They run `sayHello` or `sayGoodbye` in the scope of either `person1` or `person2`.

Both `call` and `apply` perform very similar functions: they execute a function in the context, or scope, of the first argument that you pass to them. Also, they're both functions that can only be called on other functions. You're not going to be able to run `person1.call()`, nor does it make any sense to do so.

The difference is when you want to *seed* this call with a set of arguments. Say you want to make a `say()` method that's a little more dynamic:

CODE

```
var say = function(greeting){
    alert(greeting + ', ' + this.name);
};

say.call(person1, 'Hello');
say.call(person2, 'Goodbye');
```

So that's `call` for you. It runs the function in the context of the first argument, and subsequent arguments are

passed in to the function to work with. So how does it work with more than one argument?

```
var update = function(name, age, size){
  this.name = name;
  this.age = age;
  this.size = size;
};

update.call(person1, 'Slarty', 200, '1xM');
```

CODE

No big deal. They're simply passed to the function if it takes more than one parameter.

The limitations of `call` quickly become apparent when you want to write code that doesn't (or shouldn't) know the number of arguments that the functions need... like a dispatcher.

```
var dispatch = function(person, method, args){
  method.apply(person, args);
};

dispatch(person1, say, ['Hello']);
dispatch(person2, update, ['Slarty', 200, '1xM']);
```

CODE

So that's where `apply` comes in - the second argument needs to be an array, which is unpacked into arguments that are passed to the called function.

So that's the difference between `call` and `apply`. Both can be called on functions, which they run in the context of the first argument. In `call` the subsequent arguments are passed in to the function as they are, while `apply` expects the second argument to be an array that it unpacks as arguments for the called function.

96. Javascript callbacks

What is a callback? A callback is a function to be executed after another function is executed. Sounds tongue-twisted? Normally if you want to call function `do_b` after function `do_a` the code looks something like

```
function do_a(){
  console.log(`do_a: this comes out first`);
}

function do_b(){
  console.log(`do_b: this comes out later` );
}

do_a();
do_b();
```

CODE

Result

CODE

```
`do_a`: this comes out first
`do_b`: this comes out later
```

However javascript is an event driven language. If `do_a` takes longer than `do_b`, the result of `do_b` comes out first than `do_a`;

CODE

```
function do_a(){
  // simulate a time consuming function
  setTimeout( function(){
    console.log( `do_a`: this takes longer than `do_b` );
  }, 1000 );
}

function do_b(){
  console.log( `do_b`: this is supposed to come out after `do_a` but it comes out before `do_a` );
}

do_a();
do_b();
```

Result

CODE

```
`do_b`: this is supposed to come out after `do_a` but it comes out before `do_a`
`do_a`: this takes longer than `do_b`
```

So how do we make sure `do_b` comes out after `do_a` in that situation? This is where callbacks comes in handy.

CODE

```
function do_a( callback ){
  setTimeout( function(){
    // simulate a time consuming function
    console.log( `do_a`: this takes longer than `do_b` );

    // if callback exist execute it
    callback && callback();
  }, 3000 );
}

function do_b(){
  console.log( `do_b`: now we can make sure `do_b` comes out after `do_a` );
}

do_a( function(){
  do_b();
});
```

Result

```
`do_a`: this takes longer than `do_b`  
`do_b`: now we can make sure `do_b` comes out after `do_a`
```

Different ways of applying a callback

CODE

```
function basic( callback ){
    console.log( 'do something here' );

    var result = 'i am the result of `do something` to be past to the callback';

    // if callback exist execute it
    callback && callback( result );
}

function callbacks_with_call( arg1, arg2, callback ){
    console.log( 'do something here' );

    var result1 = arg1.replace( 'argument', 'result' ),
        result2 = arg2.replace( 'argument', 'result' );

    this.data = 'i am some data that can be use for the callback function with `this` key word';

    // if callback exist execute it
    callback && callback.call( this, result1, result2 );
}

// this is similar to `callbacks_with_call`
// the only difference is we use `apply` instead of `call`
// so we need to pass arguments as an array
function callbacks_with_apply( arg1, arg2, callback ){
    console.log( 'do something here' );

    var result1 = arg1.replace( 'argument', 'result' ),
        result2 = arg2.replace( 'argument', 'result' );

    this.data = 'i am some data that can be use for the callback function with `this` key word';

    // if callback exist execute it
    callback && callback.apply( this, [ result1, result2 ] );
}

basic( function( result ){
    console.log( 'this callback is going to print out the result from the function `basic`' );
    console.log( result );
} );

console.log( '-----');
---');

( function(){
    var arg1 = 'i am argument1',
        arg2 = 'i am argument2';

    callbacks_with_call( arg1, arg2, function( result1, result2 ){
        console.log( 'this callback is going to print out the results from the function `callbacks_with_call`' );
        console.log( 'result1: ' + result1 );
        console.log( 'result2: ' + result2 );
        console.log( 'data from `callbacks_with_call`: ' + this.data );
    });
})();

console.log( '-----');
---');
```

```
( function(){
  var arg1 = 'i am argument1',
      arg2 = 'i am argument2';

  callbacks_with_apply( arg1, arg2, function( result1, result2 ){
    console.log( 'this callback is going to print out the result from the function `callbacks_with_apply`' );
    console.log( 'result1: ' + result1 );
    console.log( 'result2: ' + result2 );
    console.log( 'data from `callbacks_with_apply`: ' + this.data );
  });
})();
```

Result

CODE

```
do something here
this callback is going to print out the result from the function `basic`
i am the result of `do something` to be past to the callback
-----
do something here
this callback is going to print out the results from the function `callbacks_with_call`
result1: i am result1
result2: i am result2
data from `callbacks_with_call`: i am some data that can be use for the callback function with `this` key word
-----
do something here
this callback is going to print out the result from the function `callbacks_with_apply`
result1: i am result1
result2: i am result2
data from `callbacks_with_apply`: i am some data that can be use for the callback function with `this` key word
```

Now we know how to [execute node.js code, using require and exports](#), the importance of [function scopes and closures](#) and callbacks. Next we are going to take a look at the most important thing in node.js – [Events](#).

97. Demystifying JavaScript Closures, Callbacks and IIFEs

We've already taken a close look at [variable scope and hoisting](#), so today we'll finish our exploration by examining three of the most important and heavily-used concepts in modern JavaScript development — closures, callbacks and IIFEs.

Closures

In JavaScript, a [closure](#) is any function that keeps reference to variables from its parent's scope *even after the parent has returned*.

This means practically any function can be considered a closure, because, as we learned in the [variable scope](#) section from the first part of this tutorial, a function can refer to, or have access to –

- any variables and parameters in its own function scope
- any variables and parameters of outer (parent) functions
- any variables from the global scope.

So, chances are you've already used closures without even knowing it. But our aim is not just to use them – it is to understand them. If we don't understand how they work, we can't use them *properly*. For that reason, we are going to split the above closure definition into three easy-to-comprehend points.

Point 1: You can refer to variables defined outside of the current function.

CODE

```
function setLocation(city) {
  var country = "France";

  function printLocation() {
    console.log("You are in " + city + ", " + country);
  }

  printLocation();
}

setLocation ("Paris"); // output: You are in Paris, France
```

[Try out the example in JS Bin](#)

In this code example, the `printLocation()` function refers to the `country` variable and the `city` parameter of the enclosing (parent) `setLocation()` function. And the result is that, when `setLocation()` is called, `printLocation()` successfully uses the variables and parameters of the former to output "You are in Paris, France".

Point 2: Inner functions can refer to variables defined in outer functions even after the latter have returned.

CODE

```
function setLocation(city) {
  var country = "France";

  function printLocation() {
    console.log("You are in " + city + ", " + country);
  }

  return printLocation;
}

var currentLocation = setLocation ("Paris");

currentLocation(); // output: You are in Paris, France
```

[Try out the example in JS Bin](#)

This is almost identical to the first example, except that this time `printLocation()` is *returned* inside the outer `setLocation()` function, instead of being immediately called. So, the value of `currentLocation` is the inner `printLocation()` function.

If we alert `currentLocation` like this – `alert(currentLocation);` – we'll get the following output:

CODE

```
function printLocation () {
  console.log("You are in " + city + ", " + country);
}
```

As we can see, `printLocation()` is executed outside its lexical scope. It seems that `setLocation()` is gone, but `printLocation()` still has access to, and "remembers", its variable (`country`) and parameter (`city`).

A closure (inner function) is able to remember its surrounding scope (outer functions) even when it's executed outside its lexical scope. Therefore, you can call it at any time later in your program.

Point 3: *Inner functions store their outer function's variables by reference, not by value.*

CODE

```
function cityLocation() {
  var city = "Paris";

  return {
    get: function() { console.log(city); },
    set: function(newCity) { city = newCity; }
  };
}

var myLocation = cityLocation();

myLocation.get();          // output: Paris
myLocation.set('Sydney');
myLocation.get();          // output: Sydney
```

[Try out the example in JS Bin](#)

Here `cityLocation()` returns an object containing two closures – `get()` and `set()` – and they both refer to the outer variable `city`. `get()` obtains the current value of `city`, while `set()` updates it. When `myLocation.get()` is called for the second time, it outputs the updated (current) value of `city` – "Sydney" – rather than the default "Paris".

So, closures can both read and update their stored variables, and the updates are visible to any closures that have access to them. This means that closures store *references* to their outer variables, rather than copying their values. This is a very important point to remember, because not knowing it can lead to some hard-to-spot logic errors – as we'll see in the "Immediately-Invoked Function Expressions (IIFEs)" section.

One interesting feature of closures is that the variables in a closure are automatically hidden. Closures store data in their enclosed variables without providing direct access to them. The only way to alter those variables is by providing access to them indirectly. For example, in the last piece of code we saw that we can modify the variable `city` only obliquely by using the `get()` and `set()` closures.

We can take advantage of this behavior to store private data in an object. Instead of storing the data as an object's properties, we can store it as variables in the constructor, and then use closures as methods that refer to those variables.

As you can see, there is nothing mystical or esoteric around the closures – only three simple points to remember.

Callbacks

In JavaScript, functions are first-class objects. One of the consequences of this fact is that functions can be passed as arguments to other functions and can also be returned by other functions.

A function that takes other functions as arguments or returns functions as its result is called a [higher-order function](#), and the function that is passed as an argument is called a [callback function](#). It's named "callback" because at some point in time it is "called back" by the higher-order function.

Callbacks have many everyday usages. One of them is when we use the `setTimeout()` and `setInterval()` methods of the browser's `window` object – methods that accept and execute callbacks:

```
function showMessage(message){
  setTimeout(function(){
    alert(message);
  }, 3000);
}

showMessage('Function called 3 seconds ago');
```

CODE

[Try out the example in JS Bin](#)

Another example is when we attach an event listener to an element on a page. By doing that we're actually providing a pointer to a callback function that will be called when the event occurs.

```
// HTML

<button id='btn'>Click me</button>

// JavaScript

function showMessage(){
  alert('Woohoo!');
}

var el = document.getElementById("btn");
el.addEventListener("click", showMessage);
```

CODE

[Try out the example in JS Bin](#)

The easiest way to understand how higher-order functions and callbacks work is to create your own. So, let's create one now:

```
function fullName(firstName, lastName, callback){
  console.log("My name is " + firstName + " " + lastName);
  callback(lastName);
}

var greeting = function(ln){
  console.log('Welcome Mr. ' + ln);
};

fullName("Jackie", "Chan", greeting);
```

CODE

[Try out the example in JS Bin](#)

Here we create a function `fullName()` that takes three arguments – two for the first and last name, and one for the callback function. Then, after the `console.log()` statement, we put a function call that will trigger the actual callback function – the `greeting()` function defined below the `fullName()`. And finally, we call `fullName()`, where `greeting()` is passed as a variable – *without parentheses* – because we don't want it executed right away, but simply want to point to it for later use by `fullName()`.

We are passing the function definition, not the function call. This prevents the callback from being executed immediately, which is not the idea behind the callbacks. Passed as function definitions, they can be executed at any time and at any point in the containing function. Also, because callbacks behave as if they are actually placed inside that function, they are in practice closures: they can access the containing function's variables and parameters, and even the variables from the global scope.

The callback can be an existing function as shown in the preceding example, or it can be an anonymous function, which we create when we call the higher-order function, as shown in the following example:

CODE

```
function fullName(firstName, lastName, callback){  
    console.log("My name is " + firstName + " " + lastName);  
    callback(lastName);  
}  
  
fullName("Jackie", "Chan", function(ln){console.log('Welcome Mr. ' + ln)});
```

[Try out the example in JS Bin](#)

Callbacks are heavily used in JavaScript libraries to provide generalization and reusability. They allow the library methods to be easily customized and/or extended. Also, the code is easier to maintain, and much more concise and readable. Every time you need to transform your unnecessary repeated code pattern into more abstract/generic function, callbacks come to the rescue.

Let's say we need two functions – one that prints information about published articles and another that prints information about sent messages. We create them, but we notice that some part of our logic is repeated in both of the functions. We know that having one and the same piece of code in different places is unnecessary and hard to maintain. So, what is the solution? Let's illustrate it in the next example:

CODE

```

function publish(item, author, callback){ // Generic function with common data
    console.log(item);
    var date = new Date();

    callback(author, date);
}

function messages(author, time){ // Callback function with specific data
    var sendTime = time.toLocaleTimeString();
    console.log("Sent from " + author + " at " + sendTime);
}

function articles(author, date){ // Callback function with specific data
    var pubDate = date.toDateString();
    console.log("Written by " + author);
    console.log("Published " + pubDate);
}

publish("How are you?", "Monique", messages);

publish("10 Tips for JavaScript Developers", "Jane Doe", articles);

```

[Try out the example in JS Bin](#)

What we've done here is to put the repeated code pattern (`console.log(item)` and `var date = new Date()`) into a separate, generic function (`publish()`), and leave only the specific data inside other functions – which are now callbacks. That way, with one and the same function we can print information for all sorts of related things – messages, articles, books, magazines and so on. The only thing you need to do is to create a specialized callback function for each type, and pass it as an argument to the `publish()` function.

Immediately-Invoked Function Expressions (IIFEs)

An [Immediately-invoked function expression](#), or IIFE (pronounced "iffy"), is a function expression (named or anonymous) that is executed right away after its creation.

There are two slightly different syntax variations of this pattern:

CODE

```

// variant 1

(function () {
    alert('Woohoo!');
})();

// variant 2

(function () {
    alert('Woohoo!');
}());

```

To turn a regular function into an IIFE you need to perform two steps:

1. You need to wrap the whole function in parentheses. As the name suggests, an IIFE must be a function expression, not a function definition. So, the purpose of the enclosing parentheses is to transform a function

definition into an expression. This is because, in JavaScript, everything in parentheses is treated as an expression.

2. You need to add a pair of parentheses at the very end (variant 1), or right after the closing curly brace (variant 2), which causes the function to be executed immediately.

There are also three more things to bear in mind:

First, if you assign the function to a variable, you don't need to enclose the whole function in parentheses, because it is already an expression:

```
var sayWoohoo = function () {
  alert('Woohoo!');
}();
```

CODE

Second, a semicolon is required at the end of an IIFE, as otherwise your code may not work properly.

And third, you can pass arguments to an IIFE (it's a function, after all), as the following example demonstrates:

```
(function (name, profession) {
  console.log("My name is " + name + ". I'm an " + profession + ".");
})(("Jackie Chan", "actor")); // output: My name is Jackie Chan. I'm an actor.
```

CODE

[Try out the example in JS Bin](#)

It's a common pattern to pass the global object as an argument to the IIFE so that it's accessible inside of the function without having to use the `window` object, which makes the code independent of the browser environment.

The following code creates a variable `global` that will refer to the global object no matter what platform you are working on:

```
<function (global) {
  // access the global object via 'global'
}(this);
</code></pre>
```

CODE

`<p>This code will work both in the browser (where the global object is <code>window</code>), or in a Node.js environment (where we refer to the global object with the special variable <code>global</code>). </p>`

`<p>One of the great benefits of an IIFE is that, when using it, you don't have to worry about polluting the global space with temporary variables. All the variables you define inside an IIFE will be local. Let's check this out:</p>`

```
[code language="javascript"](function(){

  var today = new Date();
  var currentTime = today.toLocaleTimeString();
  console.log(currentTime); // output: the current local time (e.g. 7:08:52 PM)

})();

console.log(currentTime); // output: undefined
```

[Try out the example in JS Bin](#)

In this example, the first `console.log()` statement works fine, but the second fails, because the variables `today` and `currentTime` are made local thanks to the IIFE.

We know already that closures keep references to outer variables, and thus, they return the most recent/updated values. So, what do you think is going to be the output of the following example?

```
function printFruits(fruits){
  for (var i = 0; i < fruits.length; i++) {
    setTimeout( function(){
      console.log( fruits[i] );
    }, i * 1000 );
  }
}

printFruits(["Lemon", "Orange", "Mango", "Banana"]);
```

CODE

[Try out the example in JS Bin](#)

You may have expected that the names of the fruits would be printed one after another at one-second intervals. But, in practice, the output is four times "undefined". So, where is the catch?

The catch is that the value of `i`, inside the `console.log()` statement, is equal to 4 for each iteration of the loop. And, since we have nothing at index 4 in our `fruits` array, the output is "undefined". (Remember that, in JavaScript, an array's index starts at 0.) The loop terminates when `i < fruits.length` returns `false`. So, at the end of the loop the value of `i` is 4. That most recent version of the variable is used in all the functions produced by the loop. All this happens because closures are linked to the variables themselves, not to their values.

To fix the problem, we need to provide a new scope – for each function created by the loop – that will capture the current state of the `i` variable. We do that by closing the `setTimeout()` method in an IIFE, and defining a private variable to hold the current copy of `i`.

```
function printFruits(fruits){
  for (var i = 0; i < fruits.length; i++) {
    (function(){
      var current = i; // define new variable that will hold the current value
      // of "i"
      setTimeout( function(){
        console.log( fruits[current] ); // this time the value of "current" will be different fo
        // r each iteration
      }, current * 1000 );
    })();
  }
}

printFruits(["Lemon", "Orange", "Mango", "Banana"]);
```

CODE

[Try out the example in JS Bin](#)

We can also use the following variant, which does the same job:

```

function printFruits(fruits){
  for (var i = 0; i < fruits.length; i++) {
    (function(current){
      setTimeout( function(){
        console.log( fruits[current] );
      }, current * 1000 );
    })( i );
  }
}

printFruits(["Lemon", "Orange", "Mango", "Banana"]);

```

[Try out the example in JS Bin](#)

An IIFE is often used to create scope to encapsulate modules. Within the module there is a private scope that is self-contained and safe from unwanted or accidental modification. This technique, called the [module pattern](#), is a powerful example of using closures to manage scope, and it's heavily used in many of the modern JavaScript libraries (jQuery and Underscore, for example).

Conclusion

The aim of this tutorial has been to present these fundamental concepts as clearly and concisely as possible – as a set of simple principles or rules. Understanding them well is key to being a successful and productive JavaScript developer.

For a more detailed and in-depth explanation of the topics presented here, I recommend you take a look at Kyle Simpson's [You Don't Know JS: Scope & Closures](#).

98. Understand JavaScript Callback Functions and Use Them

(Learn JavaScript Higher-order Functions, aka Callback Functions)

In JavaScript, functions are first-class objects; that is, functions are of the type *Object* and they can be used in a first-class manner like any other object (String, Array, Number, etc.) since they are in fact objects themselves. They can be "stored in variables, passed as arguments to functions, created within functions, and returned from functions"¹.

Because functions are first-class objects, we can pass a function as an argument in another function and later execute that passed-in function or even return it to be executed later. This is the essence of using callback functions in JavaScript. In the rest of this article we will learn everything about JavaScript callback functions. Callback functions are probably the most widely used functional programming technique in JavaScript, and you can find them in just about every piece of JavaScript and jQuery code, yet they remain mysterious to many JavaScript developers. The mystery will be no more, by the time you finish reading this article.

Callback functions are derived from a programming paradigm known as **functional programming**. At a fundamental level, functional programming specifies the use of functions as arguments. Functional programming was —and still is, though to a much lesser extent today—seen as an esoteric technique of specially trained, master programmers.

Fortunately, the techniques of functional programming have been elucidated so that mere mortals like you and me

can understand and use them with ease. One of the chief techniques in functional programming happens to be *callback functions*. As you will read shortly, implementing callback functions is as easy as passing regular variables as arguments. This technique is so simple that I wonder why it is mostly covered in advanced JavaScript topics.

What is a Callback or Higher-order Function?

A callback function, also known as a higher-order function, is a function that is passed to another function (let's call this other function "otherFunction") as a parameter, and the callback function is called (or executed) inside the otherFunction. A callback function is essentially a pattern (an established solution to a common problem), and therefore, the use of a callback function is also known as a callback pattern.

Consider this common use of a callback function in jQuery:

```
//Note that the item in the click method's parameter is a function, not a variable.
//The item is a callback function
$("#btn_1").click(function() {
    alert("Btn 1 Clicked");
});
```

CODE

As you see in the preceding example, we pass a function as a parameter to the *click* method. And the *click* method will call (or execute) the callback function we passed to it. This example illustrates a typical use of callback functions in JavaScript, and one widely used in jQuery.

Ruminate on this other classic example of callback functions in basic JavaScript:

```
var friends = ["Mike", "Stacy", "Andy", "Rick"];

friends.forEach(function (eachName, index){
    console.log(index + 1 + ". " + eachName); // 1. Mike, 2. Stacy, 3. Andy, 4. Rick
});
```

CODE

Again, note the way we pass an anonymous function (a function without a name) to the *forEach* method as a parameter.

So far we have passed anonymous functions as a parameter to other functions or methods. Lets now understand how callbacks work before we look at more concrete examples and start making our own callback functions.

How Callback Functions Work?

We can pass functions around like variables and return them in functions and use them in other functions. When we pass a callback function as an argument to another function, we are only passing the function definition. We are not executing the function in the parameter. In other words, we aren't passing the function with the trailing pair of executing parenthesis () like we do when we are executing a function.

And since the containing function has the callback function in its parameter as a function definition, it can execute the callback anytime.

Note that the callback function is not executed immediately. It is “called back” (hence the name) at some specified point inside the containing function’s body. So, even though the first jQuery example looked like this:

```
//The anonymous function is not being executed there in the parameter.  
//The item is a callback function  
$("#btn_1").click(function() {  
    alert("Btn 1 Clicked");  
});
```

the anonymous function will be called later inside the function body. Even without a name, it can still be accessed later via the *arguments* object by the containing function.

Callback Functions Are Closures

When we pass a callback function as an argument to another function, the callback is executed at some point inside the containing function's body just as if the callback were defined in the containing function. This means the callback is a closure. Read my post, [Understand JavaScript Closures With Ease](#) for more on closures. As we know, closures have access to the containing function's scope, so the callback function can access the containing functions' variables, and even the variables from the global scope.

Basic Principles when Implementing Callback Functions

While uncomplicated, callback functions have a few noteworthy principles we should be familiar with when implementing them.

Use Named OR Anonymous Functions as Callbacks

In the earlier jQuery and forEach examples, we used anonymous functions that were defined in the parameter of the containing function. That is one of the common patterns for using callback functions. Another popular pattern is to declare a named function and pass the name of that function to the parameter. Consider this:

CODE

```

// global variable
var allUserData = [];

// generic logStuff function that prints to console
function logStuff (userData) {
    if ( typeof userData === "string")
    {
        console.log(userData);
    }
    else if ( typeof userData === "object")
    {
        for (var item in userData) {
            console.log(item + ": " + userData[item]);
        }
    }
}

// A function that takes two parameters, the last one a callback function
function getInput (options, callback) {
    allUserData.push (options);
    callback (options);

}

// When we call the getInput function, we pass logStuff as a parameter.
// So logStuff will be the function that will be called back (or executed) inside the getInput function
getInput ({name:"Rich", speciality:"JavaScript"}, logStuff);
// name: Rich
// speciality: JavaScript

```

Pass Parameters to Callback Functions

Since the callback function is just a normal function when it is executed, we can pass parameters to it. We can pass any of the containing function's properties (or global properties) as parameters to the callback function. In the preceding example, we pass *options* as a parameter to the callback function. Let's pass a global variable and a local variable:

CODE

```

//Global variable
var generalLastName = "Clinton";

function getInput (options, callback) {
    allUserData.push (options);
    // Pass the global variable generalLastName to the callback function
    callback (generalLastName, options);
}

```

Make Sure Callback is a Function Before Executing It

It is always wise to check that the callback function passed in the parameter is indeed a function before calling it. Also, it is good practice to make the callback function optional.

Let's refactor the getInput function from the previous example to ensure these checks are in place.

CODE

```

function getInput(options, callback) {
    allUserData.push(options);

    // Make sure the callback is a function
    if (typeof callback === "function") {
        // Call it, since we have confirmed it is callable
        callback(options);
    }
}

```

Without the check in place, if the `getInput` function is called either without the `callback` function as a parameter or in place of a function a non-function is passed, our code will result in a runtime error.

Problem When Using Methods With The `this` Object as Callbacks

When the callback function is a method that uses the `this` object, we have to modify how we execute the callback function to preserve the `this` object context. Or else the `this` object will either point to the global window object (in the browser), if `callback` was passed to a global function. Or it will point to the object of the containing method.

Let's explore this in code:?

CODE

```

// Define an object with some properties and a method
// We will later pass the method as a callback function to another function
var clientData = {
    id: 094545,
    fullName: "Not Set",
    // setUserName is a method on the clientData object
    setUserName: function (firstName, lastName) {
        // this refers to the fullName property in this object
        this.fullName = firstName + " " + lastName;
    }
}

function getUserInput(firstName, lastName, callback) {
    // Do other stuff to validate firstName/lastName here

    // Now save the names
    callback (firstName, lastName);
}

```

In the following code example, when `clientData.setUserName` is executed, `this.fullName` will not set the `fullName` property on the `clientData` object. Instead, it will set `fullName` on the `window` object, since `getUserInput` is a global function. This happens because the `this` object in the global function points to the `window` object.

CODE

```

getUserInput ("Barack", "Obama", clientData.setUserName);

console.log (clientData.fullName); // Not Set

// The fullName property was initialized on the window object
console.log (window.fullName); // Barack Obama

```

Use the Call or Apply Function To Preserve `this`

We can fix the preceding problem by using the `Call` or `Apply` function (we will discuss these in a full blog post later).

For now, know that every function in JavaScript has two methods: `Call` and `Apply`. And these methods are used to set

the *this* object inside the function and to pass arguments to the functions.

Call takes the value to be used as the *this* object inside the function as the first parameter, and the remaining arguments to be passed to the function are passed individually (separated by commas of course). The **Apply** function's first parameter is also the value to be used as the *this* object inside the function, while the last parameter is an array of values (or the *arguments* object) to pass to the function.

This sounds complex, but lets see how easy it is to use Apply or Call. To fix the problem in the previous example, we will use the Apply function thus:

```
//Note that we have added an extra parameter for the callback object, called "callbackObj"
function getUserInput(firstName, lastName, callback, callbackObj) {
    // Do other stuff to validate name here

    // The use of the Apply function below will set the this object to be callbackObj
    callback.apply(callbackObj, [firstName, lastName]);
}
```

CODE

With the *Apply* function setting the *this* object correctly, we can now correctly execute the callback and have it set the *fullName* property correctly on the *clientData* object:

```
// We pass the clientData.setUserName method and the clientData object as parameters. The clientD-
ata object will be used by the Apply function to set the this object
getUserInput ("Barack", "Obama", clientData.setUserName, clientData);

// the fullName property on the clientData was correctly set
console.log (clientData.fullName); // Barack Obama
```

CODE

We would have also used the *Call* function, but in this case we used the *Apply* function.

Multiple Callback Functions Allowed

We can pass more than one callback functions into the parameter of a function, just like we can pass more than one variable. Here is a classic example with jQuery's AJAX function:

CODE

```

function successCallback() {
    // Do stuff before send
}

function successCallback() {
    // Do stuff if success message received
}

function completeCallback() {
    // Do stuff upon completion
}

function errorCallback() {
    // Do stuff if error received
}

$.ajax({
    url:"http://fiddle.jshell.net/favicon.png",
    success:successCallback,
    complete:completeCallback,
    error:errorCallback
});

});
```

"Callback Hell" Problem And Solution

In asynchronous code execution, which is simply execution of code in any order, sometimes it is common to have numerous levels of callback functions to the extent that you have code that looks like the following. The messy code below is called callback hell because of the difficulty of following the code due to the many callbacks. I took this example from the node-mongodb-native, a MongoDB driver for Node.js. [2]. The **example code below is just for demonstration:**

CODE

```

var p_client = new Db('integration_tests_20', new Server("127.0.0.1", 27017, {}), {'pk':CustomPKFactory});
p_client.open(function(err, p_client) {
    p_client.dropDatabase(function(err, done) {
        p_client.createCollection('test_custom_key', function(err, collection) {
            collection.insert({'a':1}, function(err, docs) {
                collection.find({'_id':new ObjectId("aaaaaaaaaaaa")}, function(err, cursor) {
                    cursor.toArray(function(err, items) {
                        test.assertEquals(1, items.length);

                        // Let's close the db
                        p_client.close();
                    });
                });
            });
        });
    });
});
```

You are not likely to encounter this problem often in your code, but when you do—and you will from time to time—here are two solutions to this problem. [3]

1. Name your functions and declare them and pass just the name of the function as the callback, instead of defining an anonymous function in the parameter of the main function.
2. Modularity: Separate your code into modules, so you can export a section of code that does a particular job. Then you can import that module into your larger application.

??

Make Your Own Callback Functions

Now that you completely (I think you do; if not it is a quick reread :)) understand everything about JavaScript callback functions and you have seen that using callback functions are rather simple yet powerful, you should look at your own code for opportunities to use callback functions, for they will allow you to:

- Do not repeat code (DRY—Do Not Repeat Yourself)
- Implement better abstraction where you can have more generic functions that are versatile (can handle all sorts of functionalities)
- Have better maintainability
- Have more readable code
- Have more specialized functions.

It is rather easy to make your own callback functions. In the following example, I could have created one function to do all the work: retrieve the user data, create a generic poem with the data, and greet the user. This would have been a messy function with much if/else statements and, even still, it would have been very limited and incapable of carrying out other functionalities the application might need with the user data.

Instead, I left the implementation for added functionality up to the callback functions, so that the main function that retrieves the user data can perform virtually any task with the user data by simply passing the user's full name and gender as parameters to the callback function and then executing the callback function.

In short, the `getUserInput` function is versatile: it can execute all sorts of callback functions with myriad of functionalities.

```
CODE
// First, setup the generic poem creator function; it will be the callback function in the getUserInput function below.
function genericPoemMaker(name, gender) {
    console.log(name + " is finer than fine wine.");
    console.log("Altruistic and noble for the modern time.");
    console.log("Always admirably adorned with the latest style.");
    console.log("A " + gender + " of unfortunate tragedies who still manages a perpetual smile");
}

//The callback, which is the last item in the parameter, will be our genericPoemMaker function we defined above.
function getUserInput(firstName, lastName, gender, callback) {
    var fullName = firstName + " " + lastName;

    // Make sure the callback is a function
    if (typeof callback === "function") {
        // Execute the callback function and pass the parameters to it
        callback(fullName, gender);
    }
}
```

Call the getUserInput function and pass the genericPoemMaker function as a callback:

CODE

```
getUserInput("Michael", "Fassbender", "Man", genericPoemMaker);
// Output
/* Michael Fassbender is finer than fine wine.
Altruistic and noble for the modern time.
Always admirably adorned with the latest style.
A Man of unfortunate tragedies who still manages a perpetual smile.
*/
```

Because the getUserInput function is only handling the retrieving of data, we can pass any callback to it. For example, we can pass a greetUser function like this:

CODE

```
function greetUser(customerName, sex) {
  var salutation = sex && sex === "Man" ? "Mr." : "Ms.";
  console.log("Hello, " + salutation + " " + customerName);
}

// Pass the greetUser function as a callback to getUserInput
getUserInput("Bill", "Gates", "Man", greetUser);

// And this is the output
Hello, Mr. Bill Gates
```

We called the same getUserInput function as we did before, but this time it performed a completely different task.

As you see, callback functions afford much versatility. And even though the preceding example is relatively simple, imagine how much work you can save yourself and how well abstracted your code will be if you start using callback functions. Go for it. Do it in the mornings; do it in the evenings; do it when you are down; do it when you are k

Note the following ways we frequently use callback functions in JavaScript, especially in modern web application development, in libraries, and in frameworks:?

- For asynchronous execution (such as reading files, and making HTTP requests)
- In Event Listeners/Handlers
- In setTimeout and setInterval methods
- For Generalization: code conciseness

Final Words

JavaScript callback functions are wonderful and powerful to use and they provide great benefits to your web applications and code. You should use them when the need arises; look for ways to refactor your code for Abstraction, Maintainability, and Readability with callback functions.

99. Quick and Easy Javascript Namespacing

One of the ways to stop polluting the global environment in Javascript is namespacing - holding all your objects and data in a hierarchy inside of one variable.

Let's say you want to write a method in `Foods.Grains.Wheat`, without worrying about `Foods` and `Foods.Grains`.

Maybe they're in different files, being worked on by different teams, or whatever. While there are many ways to do this, here's one common way:

```
CODE
window.Foods = window.Foods || {};
Foods.Grains = Foods.Grains || {};
Foods.Grains.Wheat = Foods.Grains.Wheat || {};

Foods.Grains.Wheat.harvest = function(){
    // Do something
}
```

While this method gets the work done - you get to use the namespace without overwriting code from other files - it results in a lot of unnecessary code at the top of each file. You're essentially ensuring the existence of each namespace that you want to use, and doing it again at the top of each file.

There's an easier way, though:

```
CODE
var ns = function(namespace){
    return namespace.split('.').reduce(function(holder, name){
        holder[name] = holder[name] || {};
        return holder[name];
    }, window);
};
```

Make sure that's included first, then do

```
CODE
ns('Foods.Grains.Wheat');
```

at the top of each file that you use it in.

The code itself is fairly simple if you're into functional programming and have used map / reduce before. The `namespace.split('.')` breaks the namespace into its constituent parts. `reduce` then steps through each part from left to right, ensuring that it exists as an object on `holder` , and returning the object that does. The process starts with `window` .

If you're not familiar with functional programming, hopefully there'll be a post soon with some examples (at least for `map` and `reduce`). [Subscribe](#) if you don't want to miss it.

It normally helps to add the `ns` to jQuery (or to any other library) as a plugin. Also, since `reduce` as a built-in function is fairly new, not all browsers support it. Feel free to replace it with a [library-provided method](#), or add [this implementation](#) in. If you're using NodeJS, you'll definitely want to replace `window` , probably in [an environment-agnostic way](#).

100. The Javascript Guide to Objects, Functions, Scope, Prototypes and Closures

Throughout my time writing Javascript code, I've come to realize that while I love the language to bits, it is a little difficult to understand. A lot of people have attributed this to its (admittedly not so great) design, or its obvious

deviations from the paths well worn by other languages. Either way, understanding a few simple truths can go a long way with JS. What follows is the condensed and written version of the introductory class on JavaScript that I give at training programs and user groups.

Let's dive in with objects and variables.

```
var name = 'zaphod';
var age = 42;
```

CODE

That was easy. The `var` keyword restricts the scope of the variables and prevents them from going global without us knowing, but we'll talk about that a little later.

Now how about

```
var person = {name: 'zaphod', age: 42};
```

CODE

That's a little more interesting. What we're doing here is creating one variable, `person`. That variable contains the `name` and the `age` in a simple key value store... a map, if you will. Interestingly, it turns out that all Javascript objects can be thought of this way; they're all essentially key-value maps, or dictionaries.

Accessing the data in these objects is easy:

```
alert(person.name);
alert(person.age);
```

CODE

What's more interesting is that this is equivalent to

```
alert(person['name']);
alert(person['age']);
```

CODE

This is the first 'aha' moment for most people, because it starts to give you a hazy idea of power that this approach gives you. More on that soon, though. Let's move on.

```
var sayHello = function(){
  alert('Hello.');
};
```

CODE

There's a load of things to learn from the three lines above. First, this looks suspiciously like the way we normally define objects. *That's because it is*. Functions in JS are just glorified objects. Truth be told, they aren't even very glorified - they're just objects which can be 'called'.

```
var sayHello = function(name, age) {
  return "Hello, I'm " + name + " and I'm " + age + "years old.";
}
```

CODE

Simple enough. They take arguments and they return values. Better still, we could write:

```
var sayHello = function(person) {
    return "Hello, I'm " + person.name + " and I'm " + person.age + "years old.";
}
```

CODE

So far so good. Call the function with your person, and you'd get what you expect.

```
sayHello(person);
```

CODE

So way to *call* a function *object* is to use `()`, possibly with arguments inside.

But if functions are just objects themselves, then what's to stop us from doing this?

```
person.doSomething = sayHello;
```

CODE

So `person.doSomething(person)` should give the same result. That's good. But the `person` seems written there too many times, though. If we're calling the function off a person, why pass the person in again? Let's try something else:

```
var sayHello = function() {
    return "Hello, I'm " + this.name + " and I'm " + this.age + "years old.";
}
```

CODE

So here's our first look at **scope**. Scope is the environment, or world, that the function executes in. In JS the default scope for executing functions is the `Window` object of that particular page, which is a browser level object that represents the window (or rather tab, these days) that the page is on.

So that means running `sayHello()` (by itself) now will give you an error - `this.name` will refer to the current tab object's name, but I don't think your tab will have an `age` property. That's something to watch out for; if the function only referred to `this.name`, it would have always run, but probably not with the results you expect.

This should give us what we want:

```
person.doSomething = sayHello;
person.doSomething();
```

CODE

Now when `doSomething` runs, it executes in the scope of `person`. So `this.name` and `this.age` will use the data in the `person` object. To be absolutely explicit, you can always write `sayHello.call(person)`; which will run `sayHello` in the context of `person`. The `call` method is very useful in scope related code.

We still haven't modified `sayHello` itself, though. So calling `sayHello()` will still give the error. `doSomething` only has a *reference* to the same function that `sayHello` has. You can confirm this by subsequently setting `sayHello` to something else. `person.doSomething` will continue to work as expected.

Here's another interesting piece of code:

CODE

```

var zaphod = {name: 'Zaphod', age: 42};
var marvin = {name: 'Marvin', age: 420000000000};

zaphod.sayHello = function(){
  return "Hi, I'm " + this.name;
}
marvin.sayHello = zaphod.sayHello;

```

So `zaphod.sayHello()` is going to give you "Hi, I'm Zaphod". Trick question: *What is `marvin.sayHello()` going to return?*

That's the next thing to remember about scope. It's always *runtime*. This means that the scope of a function is always the context in which it *executes*, not the context in which it was *defined*. So `marvin.sayHello()` is going to return "Hi, I'm Marvin."

We already saw this in the last example: the function was actually defined in the context of `Window` when there was no obvious or explicit scope. This one makes it a lot more clear.

Let's use what we've looked at so far to build a simple calculator.

CODE

```

var plus = function(x,y){ return x + y };
var minus = function(x,y){ return x - y };

var operations = {
  '+': plus,
  '-': minus
};

```

The functional way to use this would probably be something like this:

CODE

```

var calculate = function(x, y, operation){
  return operations[operation](x, y);
}

calculate(38, 4, '+');
calculate(47, 3, '-');

```

This example also shows how absurdly easy it is to implement some patterns like the [strategy pattern](#) in JavaScript. While I've never liked implementing patterns for their own sake, this is still something to remember.

On the other hand, let's try something a little more object-oriented. While most 'object oriented' languages today use class based inheritance, JavaScript uses prototypes. There is no class construct here, but it's still a very powerful language. Let's see how things work:

CODE

```

var Problem = function(x, y){
  this.x = x;
  this.y = y;
}

var problem1 = new Problem(4, 5);

```

So here we have a function that takes two parameters, `x` and `y` and stores them. The `new` keyword is essential here because it tells the interpreter that you're using `Problem` not as a function, but as a constructor instead. Leaving it out will simply execute the `Problem` function, which returns nothing, so `problem1` will have no value. The `new` keyword gives us a copy of the function object after running it, while preserving the original intact (so we can use it to make more `problem` objects).

Remember that `Problem` itself is a perfectly normal function - we only capitalize the 'P' as a convention to remind ourselves to use the `new` keyword, and all the constructor behavior comes from the use of the `new` keyword.

Also keep in mind that since functions are simply objects, we can easily do this:

```
alert(problem1.x);
alert(problem1.y);
```

CODE

Now we'd like to have our `Problem` capable of solving itself, so let's do this:

```
Problem.prototype.operations = {
  '+': function(x,y){ return x + y; },
  '-': function(x,y){ return x - y; }
};

Problem.prototype.calculate = function(operation){
  return this.operations[operation](this.x, this.y);
};

problem1.calculate('+');
problem1.calculate('-');
```

CODE

So here's our first look at the JavaScript **prototype**. Although prototypes are one of the cornerstones of the language, it's possible to write JS for years and not come across any reason to use them. Like most things, though, understanding them is essential to mastery.

In JS, each and every object has one built-in property called the `__proto__`. This property refers to another object which is considered the prototype for this object, a sort of *parent* or *mould* or *original* from which this object was created. This is a very useful link because it means that all objects have the properties and methods of their prototype.

Conversely, modifying a prototype to add new features to it will result in that new feature being available to all the objects which have that prototype.

The prototype model in JS works similar to the class model in other languages, in the sense that when a method is not found in the class definition of an object, its superclass is then checked, then the superclass' superclass, and so on. The same process takes place here - first the object is checked. If the method or property is not found, its `__proto__` is checked, then its prototype's `__proto__` and so on, until the interpreter hits the last object on the chain (`__proto__` is `null`).

You'll notice though, that we haven't touched `__proto__` anywhere in our code. That's because it doesn't make sense to and is a *very bad idea* to do so. Instead JS lets you use the `prototype` object on the function that you're using as a constructor (`Problem`, in this case) to specify the behavior of the prototypes of the constructed objects. It automatically sets the `__proto__` of the constructed objects to `Problem.prototype` for you.

Now it's easy to see that calling `problem1.calculate()` actually looks at `problem1` itself first. Finding no `calculate` method, it then looks at its `__proto__`, which is the same as `Problem.prototype`, on which it finds the method. It then runs `calculate` in the scope of `problem1`. This is how the method can be defined only in the prototype and still work correctly in all the objects which have that prototype.

We also notice that this prototype resolution (like most things in JavaScript) is *runtime*. We're adding methods to `Problem.prototype` *after* creating the `problem1` and `problem2` objects, but we still have no issues using these methods on `problem1` and `problem2`. The interpreter resolves the function calls at the times at which you actually call them.

Let's add one final method to our `Problem`:

```
Problem.prototype.newMessageMaker = function(){
    var self = this;
    var formatter = function(){
        return 'Values: ' + self.x + ' and ' + self.y;
    };

    return function(start, end){
        return '' + start + formatter() + end;
    };
}

var messageMaker = problem1.newMessageMaker();
// Do lots of things.
var message = messageMaker('This was the problem: ', 'Thanks for solving it!');
alert(message);
```

CODE

Now this is a contrived example, but it serves to illustrate a very important part of JavaScript - the concept of **closures**. Here, the `problem1.newMessageMaker()` actually returns *another function*, which you can then call at your leisure.

The interesting bit here is the `formatter` function. We know it's being defined in the `newMessageMaker` function, but `newMessageMaker` executes and finishes long before `messageMaker` is actually used. What then happens to `formatter`? Isn't it supposed to have vanished?

The answer is that the function returned by `newMessageMaker` *closes over* the `formatter` variable. It holds on the value of `formatter` long after everything else lets go, including the function that originally defined it.

This is also how the trick with the `self` variable works. We can't really use `this` in this case because we have no idea what scope the `messageMaker` is going to run in. So we assign `this` to some variable (here we use `self`) and use it in the function that we're going to give away. That way, the closure will ensure that the right data is always used, irrespective of which scope the function eventually runs in.

Closures are especially useful in AJAX code. More often than not, you'll find yourself passing a function to the AJAX library to run after a particular server call is finished. If you need that function to have access to any special data that you don't want to give the library, closures are a great way to do it.

This is a fairly small start down the road to JavaScript mastery and there's still a long way to go, but, as always, actually trying things out and reading other people's good code is a great way to move forward.

Edits & Credits:

- Thanks to Matthew for pointing out that `formatter(x, y)` should have been `formatter()` in the closures example.
- Thanks to Marc Prud'hommeaux for taking the time to point out my typos.
- Thanks to Gavin M for pointing out a misplaced semicolon next to Marvin's age.

101. Callback Functions in JavaScript

If you're fairly inexperienced with JavaScript but you've used jQuery, then its likely you've used [callback functions](#). But maybe you don't fully understand how they work or how they're implemented.

In this post, which is based on what I've learned about callback functions, I'll try to enlighten you on this fairly common JavaScript technique. And maybe some of our JavaScript experts can chime in and let me know what I've omitted or oversimplified.

What is a Callback Function?

The above-linked Wikipedia article defines it nicely:

A reference to executable code, or a piece of executable code, that is passed as an argument to other code.

Here's a simple example that's probably quite familiar to everyone, taken from jQuery:

```
$('#element').fadeIn('slow', function() {  
    // callback function  
});
```

CODE

This is a call to jQuery's [fadeIn\(\)](#) method. This method accepts two arguments: The speed of the fade-in and an optional callback function. In that function you can put whatever you want.

When the `fadeIn()` method is completed, then the callback function (if present) will be executed. So, depending on the speed chosen, there could be a noticeable delay before the callback function code is executed. You can read more about jQuery's callback functions [here](#).

How to Write a Callback Function

If you're writing your own functions or methods, then you might come across a need for a callback function. Here's a very simple example of a custom callback function:

CODE

```

function mySandwich(param1, param2, callback) {
    alert('Started eating my sandwich.\n\nIt has: ' + param1 + ', ' + param2);
    callback();
}

mySandwich('ham', 'cheese', function() {
    alert('Finished eating my sandwich.');
});
```

Here we have a function called `mySandwich` and it accepts three parameters. The third parameter is the callback function. When the function executes, it spits out an alert message with the passed values displayed. Then it executes the callback function.

Notice that the actual parameter is just “callback” (without parentheses), but then when the callback is executed, it’s done using parentheses. You can call this parameter whatever you want, I just used “callback” so it’s obvious what’s going on.

The callback function itself is defined in the third argument passed to the function call. That code has another alert message to tell you that the callback code has now executed. You can see in this simple example that an argument passed into a function can be a function itself, and this is what makes callbacks possible in JavaScript.

Here's [a JSBin](#) that uses the simple example above.

Make the Callback Optional

One thing you'll notice about jQuery callbacks is that they're optional. This means if a method accepts a callback, it won't return an error if a callback is not included. In our simple example, the page will return an error if we call the function without a callback, like this:

CODE

```

function mySandwich(param1, param2, callback) {
    alert('Started eating my sandwich.\n\nIt has: ' + param1 + ', ' + param2);
    callback();
}

mySandwich('ham', 'cheese');
```

You can see this in action [here](#). If you open your developer tools, you'll see an error that says “undefined is not a function” (or something similar) that appears after the initial alert message.

To make the callback optional, we can just do this:

CODE

```

function mySandwich(param1, param2, callback) {
    alert('Started eating my sandwich.\n\nIt has: ' + param1 + ', ' + param2);
    if (callback) {
        callback();
    }
}

mySandwich('ham', 'cheese');
```

Now, since we're checking to ensure the existence of `callback`, the function call won't cause an error without it. You can test this example [here](#).

Make Sure the Callback is a Function

Finally, you can ensure that whatever value is passed as the third argument is in fact a proper function, by doing this:

```
function mySandwich(param1, param2, callback) {
    alert('Started eating my sandwich.\n\nIt has: ' + param1 + ', ' + param2);
    if (callback && typeof(callback) === "function") {
        callback();
    }
}

mySandwich('ham', 'cheese', 'vegetables');
```

CODE

Notice that the function now includes a test using the `typeof` operator, to ensure that whatever is passed is actually a function. The function call has a third argument passed, but it's not a function, it's a string. So the test using `typeof` ensures no error occurs.

[Here's a JSBin](#) with a nonfunction argument passed as the callback.

A Note About Timing

Although it is true that a callback function will execute last if it is placed last in the function, this will not always appear to happen. For example, if the function included some kind of asynchronous execution (like an Ajax call or an animation), then the callback would execute after the asynchronous action begins, but possibly before it finishes.

Here's an example that uses jQuery's `animate` method:

```
function mySandwich(param1, param2, callback) {
    alert('Started eating my sandwich.\n\nIt has: ' + param1 + ', ' + param2);

    $('#sandwich').animate({
        opacity: 0
    }, 5000, function() {
        // Animation complete.
    });

    if (callback && typeof(callback) === "function") {
        callback();
    }
}

mySandwich('ham', 'cheese', function() {
    alert('Finished eating my sandwich.');
});
```

CODE

And again [here's that code live](#).

Notice that although the callback appears later in source order than the animation, the callback will actually execute long before the animation completes. In this case, solving this problem is easy: You just put the callback execution

inside the `animate` method's callback function (where it says "Animation complete").

This doesn't cover all the details regarding asynchronous functions, but it should serve as a basic warning that callback functions will only execute last as long as all the code in the function is synchronous.

Anything to Add?

For most JavaScript junkies, this is probably pretty easy stuff. So forgive me if you were expecting something deeper — this is the best I can do. :) But if you have anything else to add or want to correct anything I've said, please do so.

102. JavaScript goes to Asynchronous city

JavaScript has come a long way since its early versions and thanks to all efforts done by [TC39](#) (The organization in charge of standardizing JavaScript (or **ECMAScript** to be exact)) we now have a modern language that is used widely.

One area within **ECMAScript** that received vast improvements is **asynchronous code**. You can learn more about [asynchronous programming here](#) if you're a new developer. Fortunately we've included these changes in Windows 10's new Edge browser. Check out the change log below:

<https://dev.modern.ie/platform/changelog/desktop/10547/?compareWith=10532>

Among all these new features, let's specifically focus on "ES2016 Async Functions" behind the **Experimental Javascript** features flag and take a journey through the updates and see how ECMAScript can improve your currently workflow.

First stop: ECMAScript 5 – Callbacks city

ECMAScript 5 (and previous versions as well) are all about callbacks. To better picture this, let's have a simple example that you certainly use more than once a day: executing a XHR request.

```

var displayDiv = document.getElementById("displayDiv");

// Part 1 - Defining what do we want to do with the result
var processJSON = function (json) {
    var result = JSON.parse(json);

    result.collection.forEach(function(card) {
        var div = document.createElement("div");
        div.innerHTML = card.name + " cost is " + card.price;

        displayDiv.appendChild(div);
    });
}

// Part 2 - Providing a function to display errors
var displayError = function(error) {
    displayDiv.innerHTML = error;
}

// Part 3 - Creating and setting up the XHR object
var xhr = new XMLHttpRequest();

xhr.open('GET', "cards.json");

// Part 4 - Defining callbacks that XHR object will call for us
xhr.onload = function(){
    if (xhr.status === 200) {
        processJSON(xhr.response);
    }
}

xhr.onerror = function() {
    displayError("Unable to load RSS");
}

// Part 5 - Starting the process
xhr.send();

```

Established JavaScript developers will note how familiar this looks since XHR callbacks are used all the time! It's simple and fairly straight forward: the developer creates an XHR request and then provides the callback for the specified XHR object.

In contrast, callback complexity comes from the execution order which is not linear due to the inner nature of asynchronous code:

The "[callbacks hell](#)" can even be worse when using another asynchronous call inside of your own callback.

Second stop: ECMAScript 6 – Promises city

ECMAScript 6 is gaining momentum and Edge is [has leading support](#) with 88% coverage so far.

Among a lot of great improvements, **ECMAScript 6** standardizes the usage of *promises* (formerly known as *futures*).

According to [MDN](#), a *promise* is an object which is used for deferred and asynchronous computations. A *promise* represents an operation that hasn't completed yet, but is expected in the future. Promises are a way of organizing asynchronous operations in such a way that they appear synchronous. Exactly what we need for our XHR example.

Promises have been around for a while but the good news is that now you don't need any library anymore as they are provided by the browser.

Let's update our example a bit to support *promises* and see how it could improve the readability and maintainability of our code:

CODE

```

var displayDiv = document.getElementById("displayDiv");

// Part 1 - Create a function that returns a promise
function getJsonAsync(url) {
    // Promises require two functions: one for success, one for failure
    return new Promise(function (resolve, reject) {
        var xhr = new XMLHttpRequest();

        xhr.open('GET', url);

        xhr.onload = () => {
            if (xhr.status === 200) {
                // We can resolve the promise
                resolve(xhr.response);
            } else {
                // It's a failure, so let's reject the promise
                reject("Unable to load RSS");
            }
        }

        xhr.onerror = () => {
            // It's a failure, so let's reject the promise
            reject("Unable to load RSS");
        };

        xhr.send();
    });
}

// Part 2 - The function returns a promise
// so we can chain with a .then and a .catch
getJsonAsync("cards.json").then(json => {
    var result = JSON.parse(json);

    result.collection.forEach(card => {
        var div = document.createElement("div");
        div.innerHTML = `${card.name} cost is ${card.price}`;

        displayDiv.appendChild(div);
    });
}).catch(error => {
    displayDiv.innerHTML = error;
});

```

You may have noticed a lot of improvements here. Let's have a closer look.

Creating the promise

In order to "promisify" (sorry but I'm French so I'm allowed to invent new words) the old XHR object, you need to create a *Promise* object:

```
function getJsonAsync(url) {
    // Promises require two functions: one for success, one for failure
    return new Promise(function (resolve, reject) {
        var xhr = new XMLHttpRequest();

        xhr.open('GET', url);

        xhr.onload = () => {
            if (xhr.status === 200) {
                // We can resolve the promise
                resolve(xhr.response);
            } else {
                // It's a failure, so let's reject the promise
                reject("Unable to load RSS");
            }
        }

        xhr.onerror = () => {
            // It's a failure, so let's reject the promise
            reject("Unable to load RSS");
        };

        xhr.send();
    });
}
```

The Promise object is now an object provided by the browser

It provides two functions that developers has to call

Call resolve to fulfill the promise

Call reject to reject the promise

Using the promise

Once created, the *promise* can be used to chain asynchronous calls in a more elegant way:

```

1   getJsonAsync("cards.json").then(json => {
2     var result = JSON.parse(json);
3
4     result.collection.forEach(card => {
      var div = document.createElement("div");
      div.innerHTML = `${card.name} cost is ${card.price}`;
      displayDiv.appendChild(div);
    });
  }).catch(error => {
    displayDiv.innerHTML = error;
  });

```

So now we have (from the user standpoint):

- Get the promise **(1)**
- Chain with the success code **(2 and 3)**
- Chain with the error code **(4)** like in a try/catch block

What's interesting is that chaining *promises* are easily called using `.then().then()`, etc.

Side note: Since JavaScript is a modern language, you may notice that I've also used [syntax sugar](#) from **ECMAScript 6** like [template strings](#) or [arrow functions](#).

Terminus: ECMAScript 7 – Asynchronous city

Finally, we've reached our destination! We are almost in the [future](#), but thanks to Edge's rapid development cycle, the team is able to introduce a bit of **ECMAScript 7** with **async functions** in the latest build!

Async functions are a syntax sugar to improve the language-level model for writing asynchronous code.

Async functions are built on top of ECMAScript 6 features like generators. Indeed, generators can be used jointly with promises to produce the same results but with much more user code

We do not need to change the function which generates the promise as async functions work directly with promise.

We only need to change the calling function:

```
// Let's create an async anonymous function
(async function() {
  try {
    // Just have to await the promise!
    var json = await getJsonAsync("cards.json");
    var result = JSON.parse(json);

    result.collection.forEach(card => {
      var div = document.createElement("div");
      div.innerHTML = `${card.name} cost is ${card.price}`;

      displayDiv.appendChild(div);
    });
  } catch (e) {
    displayDiv.innerHTML = e;
  }
})();
```

This is where magic happens. This code looks like a regular synchronous code with a perfectly linear execution path:

The function has to be marked as `async`

Instead of chaining code with `then`, you can just "await" the promise

No need to chain a `.catch` function to handle errors. A regular `try/catch` block is enough

```
// Let's create an async anonymous function
(async function() {
  try {
    // Just have to await the promise!
    var json = await getJsonAsync("cards.json");
    var result = JSON.parse(json);

    result.collection.forEach(card => {
      var div = document.createElement("div");
      div.innerHTML = `${card.name} cost is ${card.price}`;

      displayDiv.appendChild(div);
    });
  } catch (e) {
    displayDiv.innerHTML = e;
  }
})();
```

Quite impressive, right?

And the good news is that you can even use `async` functions with arrow functions or class methods.

Going further

If you want more detail on how we implemented it in Chakra, please check the official post on Edge blog:

<http://blogs.windows.com/msedgedev/2015/09/30/asynchronous-code-gets-easier-with-es2016-async-function-support-in-chakra-and-microsoft-edge/>

You can also track the progress of various browsers implementation of **ECMAScript 6 and 7** using [Kangax's website](#):

Feel free also to check [our JavaScript roadmap](#) as well!

Please, do not hesitate to give us your feedback and support your favorite features by using the vote button:

Thanks for reading and we're eager to hear your feedback and ideas!

David Catuhe

Principal Program Manager

[@deltakosh](#)

103. The Callback Syndrome In Node.js

When starting out in event-driven programming, you often find yourself amazed at how it works. It is pretty much what it feels like when you discover loops or conditional statements. As you progress, it becomes obvious that the overuse of the new paradigm will most likely prevent anyone else from successfully going through your code. This article dissects the problem of asynchronous control flow in JavaScript and suggests a simple solution using closures.

What It Is

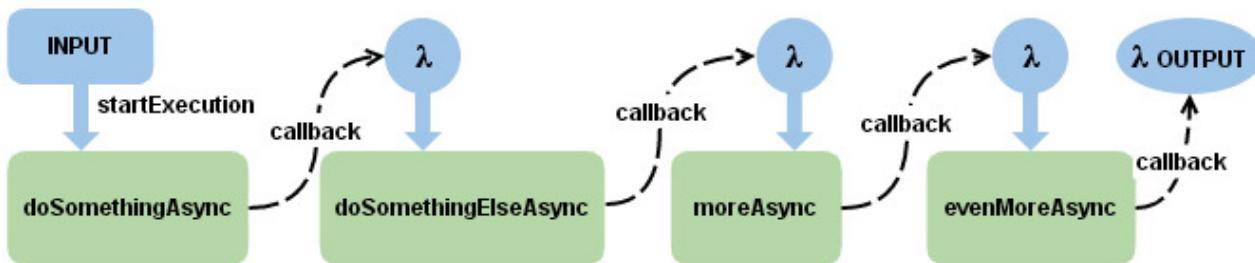
The callback syndrome describes a readability and maintainability anti-pattern found in the JavaScript code. On the whole, it means nesting callbacks, which is similar to nesting conditional statements or loops in sequential code. Still, the trouble starts when the level of nesting complicates the code to such an extent that understanding its functionality gets difficult and bugs start to creep in from unaddressed error conditions.

```
function startExecution() {
    doSomethingAsync(function(error, result) {
        doSomethingElseAsync(function(error, result) {
            moreAsync(function(error, result) {
                evenMoreAsync(function(error, result) {
                    // ... I think you got it
                });
            });
        });
    });
};
```

CODE

Why It Happens

Nesting callbacks in JavaScript is essential for long-running code like fetching data from a web API or for database queries. The core of Node.js uses callbacks for non-blocking operations. Most of the really bad nesting occurs when a lot of callbacks need to be executed in sequence, like in the diagram below.



How It Can Be Stopped

My early experience with integrating APIs in Node.js pushed me to search for a solution that improved overall readability. The first thing I came across was the `Async` module. It relies on simple asynchronous control flow patterns used by the JavaScript community for quite some time. This article deals with understanding those patterns and the ability to implement them on your own.

In all these examples, I referenced a couple of simple functions that emulate common asynchronous behavior:

```

// prints text and waits one second
function doSomethingAsync(callback) {
    console.log('doSomethingAsync: Wait for one second.');
    setTimeout(function() { callback(); }, 1000);
}

// prints text and waits half a second
function doSomethingElseAsync(callback) {
    console.log('doSomethingElseAsync: Wait for half a sec.');
    setTimeout(function() { callback(); }, 500);
}

// prints text and waits two seconds
function moreAsync(callback) {
    console.log('moreAsync: Wait for two seconds.');
    setTimeout(function() { callback(); }, 2000);
}

// prints text and waits a second and a half
function evenMoreAsync(callback) {
    console.log('evenMoreAsync: Wait for a second and a half.');
    setTimeout(function() { callback(); }, 1500);
}

// prints text
function finish() { console.log('Finished.'); }
  
```

CODE

The first thing to do is identify the common patterns used when dealing with `Async` code in Node.js. Mainly, you have two types of control flow (series and parallel) and each of them comes with its own quirks and flavors.

A series control flow requires that a set of callbacks are executed, one after another. The main reason for which you want to mimic sequential code is because some steps in a workflow must be executed in sequence. Normally, you would most likely implement something like this:

CODE

```
// executes the callbacks one after another
function series() {
    var callbackSeries = [doSomethingAsync, doSomethingElseAsync,
        moreAsync, evenMoreAsync];

    function next() {
        var callback = callbackSeries.shift();
        if (callback) {
            callback(next);
        }
        else {
            finish();
        }
    }
    next();
};

// run the example
series();
```

Parallel control flow should be used when steps in a workflow are to be executed in parallel.

CODE

```
// execute callbacks in parallel
function parallel() {
    var callbacksParallel = [doSomethingAsync, doSomethingElseAsync,
        moreAsync, evenMoreAsync];
    var executionCounter = 0;
    callbacksParallel.forEach(function(callback, index) {
        callback(function() {
            executionCounter++;
            if(executionCounter == callbacksParallel.length) {
                finish();
            }
        });
    });
};

// run the example
parallel();
```

Both of the control structures above can be generalized for any set of callbacks. I've parametrized the calls to accept an array of callbacks and a final callback. I've also stored the results inside the closure, so that they are sent as parameters for the final callback.

The last step requires that you put the code into our own Node.js module. This is fairly simple.

```
// *** index.js in casync module folder

// execute callbacks in parallel// executes the callbacks one after another
function series(callbacks, last) {
    var results = [];
    function next() {
        var callback = callbacks.shift();
        if(callback) {
            callback(function() {
                results.push(Array.prototype.slice.call(arguments));
                next();
            });
        } else {
            last(results);
        }
    }
    finish();
}

// executes the callbacks one after the other
function series(callbacks, last) {
    var results = [];
    var result_count = 0;
    callbacks.forEach(function(callback, index) {
        callback(function() {
            results[index] = Array.prototype.slice.call(arguments);
            result_count++;
            if(result_count == callbacks.length) {
                last(results);
            }
        });
    });
}

module.exports = {
    series: series,
    parallel: parallel
};

// *** test-module.js in main folder

var casync = require('./casync');

// ... declare example async functions

// run the example
casync.parallel([doSomethingAsync, doSomethingElseAsync,
    moreAsync, evenMoreAsync], finish);
```

Alternatively, you could use Deferred Objects or promises which are implemented in a couple of modules or libraries in the JavaScript community like [node-promise](#) for Node.js. Also, for nesting a maximum of three callbacks, you might get away with leaving the callbacks nested.

In Short

Because it's relatively new, Node.js has a lot of tricks and quirks that can be exploited to obtain better results. While

this improvement may not seem like such a big deal, imagine a huge code-base written entirely with nested callbacks. In the long term, this approach will drastically improve the readability and maintainability of your code.

For more cool articles on Node.js, tune in regularly. The next one will help you increase overall performance.

104. Closures in JavaScript

Vladut, [once our intern](#) now a permanent member of our software engineering team, wanted to share his experience on closures in JavaScript in the hope that it would help other developers understand them better. This article is the first in a series dedicated to use cases of various programming problems, concepts, and methods.

Closures are powerful JavaScript tools that many developers tend to ignore. Understanding closures may seem difficult at first, but it's all about the principles behind them.

Function properties

In JavaScript, just like any other programming language, a function can access any variable stated in its body.

```
function myFunction() {
    var foo = 1;
    return foo * 2;
}
```

CODE

The result after calling the function is “2”.

In the function detailed above, the lifespan of the *foo* variable is the same as the execution time of *myFunction()*. The value of the *foo* variable is “1” at the start of every function call and can only be used in the function body.

Functions can access not only the variables inside their body, but also global variables:

```
var bar = 2;
function myFunction() {
    var foo = 1;
    return foo * bar;
}
```

CODE

The result returned by the function is once again “2”.

In the function above, the lifespan of the *foo* variable is limited by the execution time of *myFunction()*, whereas the lifespan of the *bar* variable is the same as the lifespan of the window.

Although you can set your own preferences, the use of global variables is generally considered to be a disadvantage because they can be modified at any time by any other function, and this can compromise the good functioning of the application.

Helpful concepts

The concept of **lexical scoping** refers to the way in which certain variables and methods can be used in accordance

with the tree of the written code. The two properties described above are an introduction to the concept of **local scoping**:

```
function myFunction() {  
    var name = "Hubgets"; //local variable created by myFunction()  
    function displayText() {  
        alert(name); //use of variables declared in the parent function  
    }  
    displayText();  
}
```

CODE

Calling *myFunction()* will return an alert pop-up with the message "Hubgets".

In the example above, we're dealing with two nested functions:

- *myFunction()* that creates the variable name;
- *displayText()* that can access the local variable name created by the enclosing function.

Each variable created by any of the two functions has a limited lifespan and is reinitialized by every function call. The inner function *displayText()* has a major flaw, i.e. it cannot be used outside *myFunction()*.

Finally, what is a “closure”?

A **closure** is an inner function that has access to the variables of the enclosing function – the scope chain. A closure has three scope chains:

- Has access to its own scope, i.e. can access the variables stated in the function body.
- Has access to the variables created by the enclosing function.
- Has access to the variables created globally.

A **closure** has the following **major properties**:

- Most important, the inner function has access not only to its enclosing functions, but also to their parameters.
- The moment a closure is created, its entire environment is preserved *as is*. This means that the value of local variables is preserved as well. They don't get "destroyed" anymore.
- A closure can help you access variables outside a function.

Therefore, you can create a closure by adding a function inside another function, keeping in mind the properties described above. To better understand how that happens, let's see an example and some use cases.

Example

In the example below, a closure is being created with the help of the functions *addFirstName* and *showEntireName*. These functions follow the conditions and the properties of a closure as described above.

CODE

```

function myName(my_last_name) {
    var last_name = my_last_name;
    var first_name = null;

    return {
        addFirstName: function(my_first_name){
            first_name = my_first_name;
        },
        showEntireName: function() {
            console.log(first_name + " " + last_name);
        }
    }
}

var name = myName('Doe');
name.addFirstName('John');
name.showEntireName();

```

The program will display my name, i.e. “John Doe”.

When the function `myName('Doe')` is called, an environment is being created. The local variable `last_name` is assigned a value that lasts in time. In this case, the variable no longer gets “destroyed” at the end of the function call. That is why, once the function `addFirstName` is called, the function `showEntireName` can display the concatenation of the 2 variables. This is a very important, highly used characteristic of the **closure**.

Also, please note that the functions creating the closure are public, which means they can be accessed globally.

Use case

Let's assume we need to monitor something. Here are a few things you could do about it, but none of them is a solution. This is where you could use closures instead.

CODE

```

var counter = 0; //global
function add() {
    counter += 1;
}
add();add();add();

```

After calling the function for 3 times, the value of the counter will be “3”, but with the compromise of using a global variable – which is not recommended.

CODE

```

function add() {
    var counter = 0; //local
    counter += 1;
}
add();add();add();

```

After calling the `add` function, the value of the counter will be “1” because each call starts with 0 and basically only the increment of the last call will be taken into consideration.

CODE

```
function add() {  
    var counter = 0;  
    function plus() {counter += 1;} //nested function  
    plus();  
    return counter;  
}
```

This could solve the problem under the condition that the function *plus* can be accessed globally.

Conclusions

You can use closures in Node.js to treat callbacks, in the asynchronous style. You can also rely on closures if you want to create something that can be manipulated outside a function without having to reinitialize it on each and every occasion. Closures are much like an abstraction mechanism that allows you to separate problems very clearly. The code is more compact, readable, organized and suitable for functional re-use.

I hope this article has shed more light on what closures are and how they work. Certainly, there's more to closures – this is just an introduction. However, I'll leave you with a complete example that sums up all the explanations above, an application that creates a "dynamic list" in JavaScript.

```

function node (value) {
    var val = value;
    var next = null;

    return {
        getValue: function(){
            return val;
        },
        setValue: function(value){
            val = value;
        },
        setNext: function(following){
            next = following;
        },
        getNext: function() {
            return next;
        }
    }
}

function list(value) {
    var start = new node(value);
    var end = start;

    return {
        add: function(value){
            var foo = new node(value);
            end.setNext(foo);
            end = foo;
        },
        getFirst: function() {
            return start;
        }
    }
}
.....
var dynamicList=list(5)
dynamicList.getFirst().getValue()
5
dynamicList.add(7)
dynamicList.getFirst().getNext().getValue()
7

```

I'll be happy to answer any questions you have, so feel free to ask me anything in the comment fields below ??

105. Create Javascript MVC Framework in less than 200 lines (part-1)

Why ?

It is required to write javascript code in a robust and secure way. If you've a large scale application then you can use popular frameworks like Angular JS, Knockout JS, Backbone JS, etc.

But if you've a small application then you need to write your code in a robust and secure manner. In order to achieve this you can use MVC pattern.

If we want to have a MVC Pattern framework then we need to have following concepts.

What ?

- Routes and Controllers
- Factory (for code reusability)
- Constants
- Template loading and binding
- Flexible integration of modules

So, let's achieve this step by step.

1. Know about 'return' keyword in javascript functions.
2. Concept of class's and objects in javascript
3. Know about Javascript design patterns.
4. Architect your framework.
5. Concept of factories.
6. Concept of constants.
7. Concept of Dependency Injection
8. Concept of routes and controllers.
9. Sample app

1. About "return" keyword

In javascript functions, if you return something from function then it acts as a public item.

Examples:

Returning a value:

```
function getValue(){
  var a = 10;
  return a;
}
```

CODE

In function "getValue" **a** is a private variable, but you can access it through return that variable.

```
var new_a = getValue();
```

CODE

CODE

```
console.log(new_a);
```

Output:

CODE

```
10
```

Returning an object with values:

CODE

```
function getAnObject(){

var Obj = {
'key1':'val1',
'key2':'val2'
}

return Obj;
}
```

CODE

```
var new_obj = getAnObject();
```

CODE

```
console.log(new_obj);
```

Output:

CODE

```
{'key1':'val1', 'key2':'val2'}
```

Returning a function

CODE

```
function getAFunction(){

var func = function(){
console.log("im a private function");
}

return func;
}
```

CODE

```
var new_func = getAFunction();
```

CODE

```
console.log(new_func);
```

CODE

Output:

```
function(){
  console.log("im a private function");
}
```

CODE

Returning an object with functions

```
function getFunctions(){
  var funcs = {
    'func1': function(){
      console.log("im func1");
    },
  },
}
```

CODE

```
  'func2': function(){
    console.log("im func2");
  },
}
```

CODE

```
return funcs;
```

CODE

```
}
```

CODE

```
var new_funcs = getFunctions();
```

CODE

```
console.log(new_funcs);
```

CODE

Output:

```
{func1: function(){ console.log("im func1")}, func2: function(){ console.log("im func2") }}
```

CODE

2. Concept of Objects and Privacy

In JavaScript, there are different ways to create an object. The function is a first-class object. That affords us numerous ways in which to use them that other languages simply cannot. For example we can pass functions as

arguments to other functions and/or return them. Functions also play a highly specialized role in Object creation, because objects require a constructor function to create them - either explicitly or implicitly. Another salient feature of the JavaScript language is that it is interpreted at runtime. This is relevant because code can be written and evaluated at runtime.

Objects

JavaScript is fundamentally about objects. Arrays are objects. Functions are objects. Objects are objects. So what are objects? Objects are collections of name-value pairs. The names are strings, and the values are strings, numbers, booleans, and objects (including arrays and functions). Objects are usually implemented as hash tables so values can be retrieved quickly.

If a value is a function, we can consider it a method. When a method of an object is invoked, the this variable is set to the object. The method can then access the instance variables through the this variable.

Objects can be produced by constructors, which are functions which initialize objects. Constructors provide the features that classes provide in other languages, including static variables and methods.

Public

The members of an object are all public members. Any function can access, modify, or delete those members, or add new members. There are two main ways of putting members in a new object:

In the constructor

This technique is usually used to initialize public instance variables. The constructor's this variable is used to add members to the object.

```
function Container(param) {
    this.member = param;
}
```

CODE

So, if we construct a new object

```
var myContainer = new Container('abc');
```

CODE

```
then myContainer.member contains 'abc'.
```

CODE

In the prototype

This technique is usually used to add public methods. When a member is sought and it isn't found in the object itself, then it is taken from the object's constructor's prototype member. The prototype mechanism is used for inheritance. It also conserves memory. To add a method to all objects made by a constructor, add a function to the constructor's

prototype:

```
Container.prototype.stamp = function (string) {
    return this.member + string;
}
So, we can invoke the method
```

CODE

```
myContainer.stamp('def');
```

CODE

which produces 'abcdef'.

CODE

Private

Private members are made by the constructor. Ordinary vars and parameters of the constructor becomes the private members.

```
function Container(param) {
    this.member = param;
    var secret = 3;
    var that = this;
}
```

CODE

This constructor makes three private instance variables: param, secret, and that. They are attached to the object, but they are not accessible to the outside, nor are they accessible to the object's own public methods. They are accessible to private methods. Private methods are inner functions of the constructor.

```
function Container(param) {
```

CODE

```
    function dec() {
        if (secret > 0) {
            secret -= 1;
            return true;
        } else {
            return false;
        }
    }
```

CODE

```
this.member = param;
```

CODE

CODE

```

var secret = 3;
var that = this;
}

```

The private method dec examines the secret instance variable. If it is greater than zero, it decrements secret and returns true. Otherwise it returns false. It can be used to make this object limited to three uses.

By convention, we make a private that variable. This is used to make the object available to the private methods. This is a workaround for an error in the ECMAScript Language Specification which causes this to be set incorrectly for inner functions.

Private methods cannot be called by public methods. To make private methods useful, we need to introduce a privileged method.

Privileged

A privileged method is able to access the private variables and methods, and is itself accessible to the public methods and the outside. It is possible to delete or replace a privileged method, but it is not possible to alter it, or to force it to give up its secrets.

Privileged methods are assigned with this within the constructor.

CODE

```
function Container(param) {
```

CODE

```

    function dec() {
        if (secret > 0) {
            secret -= 1;
            return true;
        } else {
            return false;
        }
    }
}
```

CODE

```

    this.member = param;
    var secret = 3;
    var that = this;
}
```

CODE

```

    this.service = function () {
        return dec() ? that.member : null;
    };
}
}
```

service is a privileged method. Calling myContainer.service() will return 'abc' the first three times it is called. After that, it will return null. service calls the private dec method which accesses the private secret variable. service is available to other objects and methods, but it does not allow direct access to the private members.



In next part we'll see about design patterns and architecting our framework.

sources: [here](#)

106. Create MVC Framework (Architect the Structure) part2

In this article we're going to learn about design patterns and Architeting the framework

Design patterns

First of all what is a pattern ? and Why ?

A pattern is a reusable solution that can be applied to commonly occurring problems in software design - in our case - in writing JavaScript web applications. Another way of looking at patterns are as templates for how we solve problems - ones which can be used in quite a few different situations.

Why ?

Patterns are proven solutions: They provide solid approaches to solving issues in software development using proven techniques that reflect the experience and insights the developers that helped define them bring to the pattern.

Patterns can be easily reused: A pattern usually reflects an out of the box solution that can be adapted to suit our own needs. This feature makes them quite robust.

Patterns can be expressive: When we look at a pattern there's generally a set structure and vocabulary to the solution presented that can help express rather large solutions quite elegantly.

Categories of design patterns

1. Creational Design Patterns

2. Structural Design Patterns

3. Behavioral Design Patterns

Do you know about "class" in Javascript ?

Keep in mind that there will be patterns in this table that reference the concept of "classes". JavaScript is a class-less language, however classes can be simulated using functions.

CODE

```
// A car "class"
function Car( model ) {
    this.model = model;
    this.color = "silver";
    this.year = "2012";

    this.getInfo = function () {
        return this.model + " " + this.year;
    };
}
```

CODE

```
var myCar = new Car("ford");
myCar.year = "2010";
console.log( myCar.getInfo() );
```

Patterns

- Constructor Pattern
- Module Pattern
- Revealing Module Pattern
- Singleton Pattern
- Observer Pattern
- Mediator Pattern
- Prototype Pattern
- Command Pattern
- Facade Pattern
- Factory Pattern
- Mixin Pattern
- Decorator Pattern
- Flyweight Pattern

In this section i'm not going to explain all of those. Please refer [this](#) to get more understanding on design patterns.

About module pattern

The Module pattern was originally defined as a way to provide both private and public encapsulation for classes in conventional software engineering.

In JavaScript, the Module pattern is used to further emulate the concept of classes in such a way that we're able to include both public/private methods and variables inside a single object, thus shielding particular parts from the global scope. What this results in is a reduction in the likelihood of our function names conflicting with other functions defined in additional scripts on the page.

Privacy

The Module pattern encapsulates "privacy", state and organization using closures. It provides a way of wrapping a mix of public and private methods and variables, protecting pieces from leaking into the global scope and accidentally colliding with another developer's interface. With this pattern, only a public API is returned, keeping everything else within the closure private.

Example:

CODE

```
var testModule = (function () {
    var counter = 0;
    return {
        incrementCounter: function () {
            return counter++;
        },
        resetCounter: function () {
            console.log( "counter value prior to reset: " + counter );
            counter = 0;
        }
    };
})();

// Usage:

// Increment our counter
testModule.incrementCounter();

// Check the counter value and reset
// Outputs: counter value prior to reset: 1
testModule.resetCounter();
```

Architecting the framework

In the above picture we can observe the flow. We've constants, factory, routes and controllers.

Let's create some rules.

1. You should write your code which is repeatable in "Factory" and if a factory is dependant on other factory then you should be able to import it into your factory.
2. These factories should be instantiated when the project gets loaded.
3. Constants should be able to modified in any component like controller or factory.
4. We should be able to define routes that means when particular route comes then the specified controller should get executed.
5. The controllers are the functions which should be able to import factories and constants which are mapped for urls.
6. These controller should get executed when the particular url comes.

Let's prepare some Syntactical structure

Creating App

```
var app = MiApp();
```

CODE

If we've a structure like this then we can create multiple apps.

Creating factory

```
app.factory('factory name', ['dependancy1', 'dependancy2', function(dependency1, dependency2){  
//you should access dependency1
```

CODE

```
return {  
'publicAccess': function(){  
    return "something"  
}  
}  
}]);
```

CODE

Creating constants

```
app.constants('constant name', function(){  
return {  
'item1':'val1',  
'item2':'val2'  
}  
});
```

CODE

Wait,

You don't find any difference between factory and constants ?

I'll explain later.

Creating routes

```
app.routes('routeurlwithregularexpression', 'contrllername');
```

CODE

example:

```
app.routes('test/:id/', 'TestController');
```

CODE

Creating Controllers

```
app.Controller('TestController', ['dependency1', 'dependency2', function(dependency1, dependency2) {
{
  //your stuff that runs when the page gets loaded
}]);
```

CODE

This assumed MVC structure needs to get developed and then needs to get implemented.

So let's develop the structure.

Observe the public items and Private items

If MiApp is a module then the public items should be

1. Factory
2. Routes
3. Controller
4. Constants

So, Create a module pattern which makes these items as public

```
var MiApp = (function () {
  'use strict';
```

CODE

```
    function constants() {
```

```
}
```

```
    function routes(){
```

CODE

```
}
```

```
    function controller(){
```

CODE

```
}
```

```
    function factory(){
```

CODE

```
}
```

CODE

```

return {
  'factory': factory,
  'routes': routes,
  'controller': controller,
  'constants': constants
}
});

```

Now, observe our assumed structure and read the arguments as per our assumed structure. Let's be specific, in any component(factory, routes, controller or constant) the first argument as key and second argument as value.

But for the privacy purpose read the arguments through dynamic argument format instead of formal parameters.

So,

CODE

```

var MiApp = (function () {
  'use strict';

  function constants() {
    var key = arguments[0], val = arguments[1];
  }
}

```

CODE

```

function routes(){
  var key = arguments[0], val = arguments[1];
}

```

CODE

```

function controller(){
  var key = arguments[0], val = arguments[1];
}

```

CODE

```

function factory(){
  var key = arguments[0], val = arguments[1];
}

```

CODE

```

return {
  'factory': factory,
  'routes': routes,
  'controller': controller,
  'constants': constants
}
});

```

now you can create an app like as we assumed before

CODE

```

var app = MiApp()

```

```
app.factory('factory_name', ['$dependency1', function(dependency1){
}]);
```

CODE

```
app.routes('url', 'controller_name');
```

CODE

```
app.controller('controller_name', ['$dependency1', function(dependency1){
}]);
```

CODE

```
app.constants('name', function(){
  return {
});
```

CODE

Upto now we've just created our required structure only. We didn't implement any feature.

In next article we'll discuss about how to develop these components individually and also concept of dependency injection.

source [here](#)

107. Create MVC framework(Implement the structure)part3

In this article we're going to discuss about implementing the structure of MVC framework.

Upto now we've got basic knowledge to construct the MVC framework and also we've setup structure of MVC. In the previous part we've discussed about creation of structure with some assumed basic rules.

let's recall the rules

1. You should write your code which is repeatable in "Factory" and if a factory is dependant on other factory then you should be able to import it into your factory.
2. These factories should be instantiated when the project gets loaded.
3. Constants should be able to modified in any component like controller or factory.
4. We should be able to define routes that means when particular route comes then the specified controller should get executed.
5. The controllers are the functions which should be able to import factories and constants which are mapped for urls.
6. These controller should get executed when the particular url comes.

Step-1:

Maintain a private JSON object to hold all the factories, constants, routers and controllers.

For example:

```
var MiApp = (function () {
  'use strict';

  //Private Data
  var Resource = {
    'constants' : { },
    'factory' : { },
    'mode' : null,
    'root' : '/',
    'routes' : [],
    'controller' : { },
    'controller_dependency':{ },
  };
});
```

CODE

```
function constants() {
  var key = arguments[0],val = arguments[1];
}
function routes(){
  var key = arguments[0],val = arguments[1];
}
function controller(){
  var key = arguments[0],val = arguments[1];
}
function factory(){
  var key = arguments[0],val = arguments[1];
}
return {
  'factory': factory,
  'routes': routes,
  'controller': controller,
  'constants': constants
});
});
```

CODE

In above we've an object called "Resource" which is used to store the user defined data like factories, constants, routes and controllers.

Step-2:

Maintain an adapter to serve private functionality to publicly accessable items. This is adapter is the actual engine we need to implement to serve the stored data in Resource object. This will act as an adapter between and private and public items.

CODE

```

var api = {
  'factory': function (key, arrayArg) {

  },
  'routes' : function(route, controller){

  },
  'controller' : function(controller, handler){

  },
  'constants': function (key, val) {

  }
  'loadDependancies' : function(arrayArg){

  },
};


```

Now we've a structure of storing data and serve the data to public items.

Concept of constants:

Assumed structure:

CODE

```

app.constants('name', function(){
  return {
  }
});
```

Assumed rules:

Constants should be able to modified in any component like controller or factory. Constants can be functions, objects or hybrid. So, whenever you create a constant then the instantiated object need to get stored in private object "Resource" inside the constant with the given key through adapter. Then the constant function will be

CODE

```

'constants': function (key, val) {
  resources.constants[key] = val();
},
```

In above we've stored the constants as private so that these are not accessible outside.

Concept of Factory:

Assumed structure:

CODE

```
app.factory('factory_name', ['dependency1', 'dependency2', function(dependency1, dependency2){
  //you should access dependency1
  return {
    'publicAccess': function(){
      return "something"
    }
  }
}]);
```

Assumed rules:

1. You should write your code which is repeatable in "Factory" and if a factory is dependant on other factory then you should be able to import it into your factory.
2. These factories should be instantiated when the project gets loaded.

Factories are the functions which need to be reused in another factory or controllers. And like constants we need to instantiate the factories whenever they defined and store the instantiated factory functions in Private resource because these factories should not be able to access by outside and we need to implement the adapter to instantiate, store and to serve the factories over the controllers.

But, we've discussed that factories can be used inside another factory. So before creating and storing a factory plan for injection of factory in another factory.

Concept of dependency injection:

In order to create a factory, a factory or constant can be used inside another factory. Here we need to inject those dependencies to the factory.

How to inject constants into a factory ?

We're maintaining a Private object called Resource to hold constants with their key name. If we've already created a constant with the name "constant1" with value 1.

Now if we want to inject this value into factories you can access through Resource.constants[constant1] So,

CODE

```
app.factory('factory1', ['constant1', function(constant1){
  //you should access dependency1
  return {
    'publicAccess': function(){
      console.log(constant1);
    }
  }
}]);
```

Now create an adapter function to serve the dependencies from constants and factories based on keys.

CODE

```
'loadDependencies' : function(arrayArg){
  var dependancy = [], iter;
  for (iter = 0; iter < arrayArg.length; iter += 1) {
    if (typeof arrayArg[iter] === "string") {
      //look in modules
      if (resources.hasOwnProperty(arrayArg[iter])){
        dependancy.push(api.loadModule(arrayArg[iter]));
      } else {
        //look in factory
        if (resources.factory.hasOwnProperty(arrayArg[iter])) {
          dependancy.push(api.loadDependency(arrayArg[iter]));
        } else {
          //look in constants
          if (resources.constants.hasOwnProperty(arrayArg[iter])) {
            dependancy.push(api.loadConstant(arrayArg[iter]));
          } else {
            //if it is $me scope
            if (arrayArg[iter] === "$mi") {
              dependancy.push({}); 
            } else {
              console.log("Error: " + arrayArg[iter] + " is not Found in constants and Factories");
            }
          }
        }
      }
    }
  }
  return dependancy;
},
```

The above code will receive list of names or keys. It will first look in modules, factories, constants respectively. we'll discuss later about modules.

Simply, it will take array of strings and return array of Functions which are stored in Private object called Resource.

So the factory adapter implementation will be,

CODE

```
'factory': function (key, arrayArg) {
  var last_index = arrayArg.length-1;
  var dependancies = arrayArg.slice(0, -1);
  if (typeof arrayArg[last_index] === "function") {
    console.log("-"+api.loadDependencies(dependancies));
    resources.factory[key] = arrayArg[last_index].apply(this, api.loadDependencies(dependancies)); //arrayArg[last_index];
  } else {
    console.log("Nan");
  }
},
```

In the above we're storing the instantiated object in private object called Resource inside the factory.

Concept of Routes and Controllers

Assumed structure

```
//routes
app.routes('routeurlwithregularexpression', 'contrllername');
```

CODE

```
//controllers
app.Controller('TestController', ['dependancy1', 'dependancyn', function(dependancy1, dependancyn)
{
    //your stuff that runs when the page gets loaded
}]);
```

CODE

Assumed rules

4. We should be able to define routes that means when particular route comes then the specified controller should get executed.
5. The controllers are the functions which should be able to import factories and constants which are mapped for urls.
6. These controller should get executed when the particular url comes.

Concept of routes

Routes are the urls those can be partial urls or normal urls (/url or #url). When particular route come then we need to execute the specified controller which is collection of dependencies and your logic. But when comes to the structure routes able to specify the route name and controller name. So, route adapter will take the route and controller and store it in the private object called Resource.

```
'routes' : function(route, controller){
    var temp = {'path':route, 'handler':controller };
    resources.routes.push(temp);
},
```

CODE

Concept of Controllers:

Controller is a module which is capable of loading dependencies and should be loaded when particular url come. So we should not instantiate it whenever it get created. But here the problem is we need to remember its dependencies. So,

```
'controller' : function(controller, handler){
  var last_index = handler.length-1;
  var dependancies = handler.slice(0, -1);
  if (typeof handler[last_index] === "function") {
    resources.controller[controller] = handler[last_index];
    resources.controller_dependency[controller] = dependancies;
  } else {
    console.log("Nan");
  }
},
```

Loading controllers when route come:

I've found a resource to load some functionality when a particular url come from

here <http://krasimirtsonev.com/blog/article/A-modern-JavaScript-router-in-100-lines-history-api-pushState-hash-url> by Krasimir.

I've integrated those code with this. Please refer to Krasimir article.

Concept of template binding.

We've made this code as OpenSource. So anybody can make changes in order to improve or to add some features. Right now this is not supporting template binding.

There are libraries which are available purely for template binding.

We're trying to integrate those libraries to this project.