Cargo-Culting in JavaScript

James Padolsey on May 24th 2013 with 62 Comments

Tutorial Details

-
- **Difficulty:** Intermediate
- **Completion Time:** 30 Minutes

View post on Tuts+ Beta**Tuts+ Beta** is an optimized, mobile-friendly and easy-to-read version of the Tuts+ network.

Cargo-cult programming is what a programmer does when he or she doesn't know a particular language or paradigm well enough, and so ends up writing redundant and possibly harmful code. It rears its head quite often in the land of JavaScript. In this article, I explore the concept of cargo-cult programming and places to watch out for it in JavaScript.

Dogmatic rules surface and spread, until they are considered the norm.

Cargo-culting is sometimes defined as "the extreme adherence to the form instead of content." The form, in programming, being the syntax, paradigms, styles and patterns that we employ. The content being the abstract thing that you are seeking to represent through your code — the very substance of your program. A person with lacking understanding in an area is likely to copy the form of others without truly understanding, and thus their content — their program — can suffer.

Cargo–culting is curiously common in JavaScript, probably because of the general low barrier to entry in the front–end development world. You can whip up an HTML page with a bit of JavaScript in seconds. As a result, there are many people who become sufficiently proficient in these technologies to feel comfortable creating and imposing rules on themselves and others. Eventually, other newcomers copy these rules. Dogmatic rules surface and spread, until they are considered the norm:

- Always use strict equality operators
- Never use eval
- Always use a single var declaration per scope
- Always use an IIFE – it "protects" you

A rule continues to spread until a programmer is only using a given technique because of its popularity, instead of considering each specific use–case independently.

JavaScript Abuzz with Semicolons

If you've had the opportunity to witness the witty banter and rhetoric of the software developer over the years, you will have spotted a tendency to discuss seemingly tiny things at great lengths. Things like the semicolon, the comma, white–space or the curly brace.

Syntax like semicolons or white–space may seem to purely be elements of form, not of content. But many of these subtle syntax rules can have significant effects in JavaScript. If you don't understand the 'form' then you cannot begin to understand the 'content'.

So in this article, we will identify what areas of form in JavaScript are frequently cargo-culted off of — that is, copied without understanding.

How JavaScript can seem… an image from Angus Croll's "The Politics Of JavaScript" presentation

---

Undefined

Angus Croll, in a recent presentation, titled "The Politics Of JavaScript", highlighted one of the most common pieces of JS dogma that people cargo-cult off of:

```
if (typeof myObject.foo === 'undefined') {...}
```

Most of the time, doing such a long-winded check for `undefined` is pointless. The technique became common because people were copying other people, not because of it's actual value.

Of course, there are times when:

```
typeof x === 'undefined'
```

… is preferable to:

```
x === undefined
```

But, equally, there are times when the latter is preferred. A quick overview of the options:

```
// Determine if `x` is undefined:
x === undefined
typeof x == 'undefined'
typeof x === 'undefined'
x === void 0

// Determine if `x` is undefined OR null:
x == null
x == undefined
```

People started using the `typeof` approach because they were protecting themselves against:

- A potentially undeclared variable (*non-typeof approaches would throw TypeErrors*)
- Someone overwrote undefined globally or in a parent scope. Some environments allow you to overwrite `undefined` to something like `true`. You have to ask yourself: "*Is it likely that someone overwrote undefined, and should my script have to pander to such silliness?*"

But most of the time they're protecting themselves from having to worry. It's a catch-all avoidance of having to know the details. Knowing the details can help you though. Every character of your code should exist with a purpose in mind.

The only time that you should need to use a `typeof` check for `undefined` is when you are checking for a variable that may not have been declared, e.g. checking for jQuery in the global scope:

```
if (typeof jQuery != 'undefined') {
    // ... Use jQuery
}
```

The thing is, if jQuery *does* exist, then we can be sure that it's an object – a "truthy" thing. So this would be sufficient:

```
// or:
if (window.jQuery) {

}
```

The Great Strict/non-strict Debate

Let's take something very common and generally considered good advice, solely using strict-equality:

```
a === b
```

Strict-equality is said to be good because it avoids ambiguity. It checks both the value and the type, meaning that we don't have to worry about implicit coercion. With non-strict equality, we do have to worry about it though:

```
1 == 1    // true &mdash; okay, that's good
```

```
1 == "1"  // true &mdash; hmm
1 == [1]  // true &mdash; wat!?
```

So it would seem sensible advice to entirely avoid non-strict equality, right? Actually, no. There are many situations where strict-equality creates large amounts of redundancy, and non-strict equality is preferable.

When you know, with 100% certainty, that the types of both operands are the same, you can avoid the need for strict-equality. For example, I always know that the `typeof` operator returns a string, and my right-hand operand is also a string (e.g. `"number"`):

```
// With strict-equals
typeof x === 'number'

// With non-strict-equals:
typeof x == 'number'
```

They're both effectively identical. I am not necessarily suggesting that we abandon strict-equals in this case — I am suggesting that we remain aware of what we're doing so that we can make the best choices given each situation.

Another quite useful example is when you want to know if a value is either `null` or `undefined`. With strict equality, you might do this:

```
if (value === undefined || value === null) {
    // ...
}
```

With non-strict equality, it's far simpler:

```
if (value == null) {
    // ...
}
```

There is no catch here — it is doing exactly what we want, only, arguably, less visibly. But, if we know the language, then what's the problem? It's right there in the spec:

The comparison `x == y`, where `x` and `y` are values, produces `true` or `false`. Such a comparison is performed as follows:

- If x is null and y is undefined, return true.
- If x is undefined and y is null, return true.

If you're writing JavaScript with the intention of it being read, if at all, by people that know JavaScript, then I would argue that you shouldn't feel bad taking advantage of implicit language rules, like this.

---

hasOwnProperty

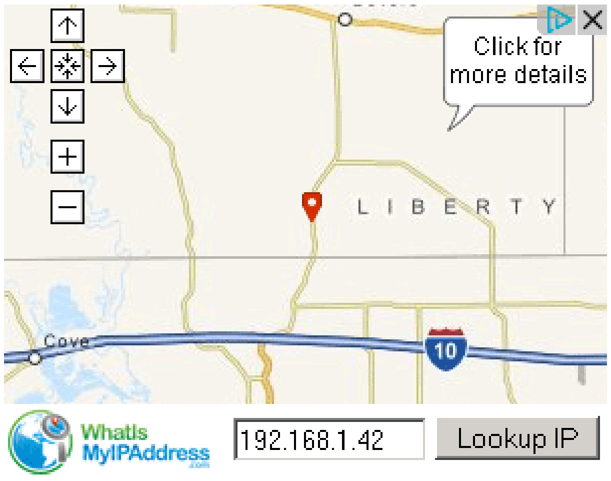
The `hasOwnProperty` method is used to determine whether a property is directly owned by an object. Is it commonly found in `for..in` loops to ensure that you only mess with direct properties and not inherited properties.

```
for (var i in object) {
    if (object.hasOwnProperty(i)) {
        // We can do stuff with `object[i]`
    }
}
```

It's important to note that the `for-in` statement will only loop through enumarable properties. Native inherited methods, for example, are not enumerable and so you don't need to worry about them anyway.

The `hasOwnProperty` check is specifically preventing you from touching properties that you or some third-party script has defined, i.e. when your object's prototype has enumerable properties.

If you know that your object's prototype (*or its prototype's prototype* etc.) doesn't have any enumerable properties, then *you don't have to worry* about using `hasOwnProperty` in your `for-in` loops. And, if your object is initialized, via ES5's `Object.create(null)`, then you won't even be able to call `hasOwnProperty` directly on the object (*no prototype means no inherited native methods*). This means that using `hasOwnProperty` by default in all of your `for-in` loops may actually break sometimes.



One potential solution for objects with `null` prototypes is to use a saved reference to `hasOwnProperty`, like so:

```
var hasOwnProperty = Object.prototype.hasOwnProperty;

// Later in your code:
for (var i in someObject) {
    if (hasOwnProperty.call(someObject, i)) {
        // ...
    }
}
```

That will work even if the object has no prototype (in the case of `Object.create(null)`). But, of course, we should only do this in the first place if we know we need it. If you're writing a third-party script for a "hostile" environment, then yes, definitely check for enumerable inherited properties. Otherwise, it may not be necessary all the time.

Note: IE9 and Safari 2.0 complicate the matter further when you're trying to identify enumerable properties that are already defined as non-enumerable. It's worth checking out a truly cross-browser forOwn loop implementation.

To conclude: your use of `hasOwnProperty` should depend on the object being looped over. It depends on what assumptions you can safely make. Blindly protecting yourself using the `hasOwnProperty` will not suffice in all cases. Be wary of cross-browser differences too.

Over-Parenthesising

Another common redundancy that creeps into JS code is the parenthesis. Within expressions, it is used to force specific grouping of sub–expressions. Without them, you are at the mercy of operator precedences and associativities. For example:

```
A && B || C
A && (B || C)
(A && B) || C
```

One of those is not like the other. The parentheses force a specific grouping, and many people prefer the extra clarity. In this case, the logical AND operator has a higher precedence than the logical OR operator, meaning that it is the first and last lines that are equivelant. The second line is an entirely different logical operation.

> Higher precedence means that it will occur before other operations in a series of operations.

To avoid this complexity, developers frequently opt for a "parentheses policy" — where you keep adding parentheses until it is abundantly clear which operations are occurring, both for you and potential readers of the code. It can be argued that this verbosity ends up making things less clear though.

It’s tricky for a reader sometimes. One must consider that any given parentheses may have been added because:

- It was needed to override default precedence/associativity
- For no functional reason at all, just for "protection" or "clarity"

Take this example:

```
A && B ? doFoo() : doBaz()
```

Without knowledge of operator precedence rules, we can see two possible operations here:

```
(A && B) ? doFoo() : doBaz()
A && (B ? doFoo() : doBaz())
```

In this case, it's the logical AND that has the higher precedence, meaning that the equivalent parenthesised expression is:

```
(A && B) ? doFoo() : doBaz()
```

We should feel no obligation to add these parentheses in our code, though. It happens implicitly. Once we recognize that it happens implicitly, we are free to ignore it and focus on the program itself.

There are, of course, valid arguments to retain the parentheses where implicit grouping is unclear. This really comes down to you and what you're comfortable with. I would, however, implore you to learn the precedences and then you can be fully empowered to take the best route, dependent on the specific code you're dealing with.

Object Keys

It's not rare to see redundant quotes in object literals:

```
var data = {
  'date': '2011-01-01',
  'id': 3243,
  'action': 'UPDATE',
  'related': { '1253': 2, '3411': 3 }
};
```

In addition to strings, JavaScript allows you to use valid identifier names and numbers as object literal keys, so the above could be re-written to:

```
var data = {
  date: '2011-01-01',
  id: 3243,
  action: 'UPDATE',
  related: { 1253: 2, 3411: 3 }
};
```

Sometimes, you may prefer the added consistency of being able to use quotes, especially if a field-name happens to be a reserved word in JavaScript (like 'class' or 'instanceof'). And that's fine.

Using quotes is not a bad thing. But it is redundant. Knowing that you don't have to use them is half the battle won. It is now your choice to do what you want.

Comma Placement

There is a huge amount of subjective preference, when it comes to punctuation placement in programming. Most recently, the JavaScript world has been abuzz with rhetoric and discontent over the comma.

Initialising an object in traditionally idiomatic JavaScript looks like this:

```
var obj = {
    a: 1,
    b: 2,
    c: 3
};
```

There is an alternative approach, which has been gaining momentum though:

```
var obj = {
      a: 1
    , b: 2
    , c: 3
};
```

The supposed benefit of placing the commas before each key-value pair (apart from the first) is that it means you only have to touch one line in order to remove a property. Using the traditional approach, you would need to remove "`c: 3`" and then the trailing comma on the line above. But with the comma-first approach you're able to just remove ", `c: 3`". Proponents claim this makes trailing commas less likely and also cleans up source-control diffs.

Opponents, however, say that this approach only achieves getting rid of the trailing-comma "problem" by introducing a new leading-comma problem. Try removing the first line and you're left with a leading comma on the next line. This is actually considered a good thing by comma-first proponents, because a leading comma would immediately throw a SyntaxError. A trailing comma, however, throws nothing, except in IE6 and 7. So if the developer fails to test their JS in those versions of IE, then the trailing commas can often creep into production code, which is never good. A leading comma throws in all environments, so is less likely to be missed.

Of course, you might argue that this entire thing is moot. We should probably be using linters like JSLint or the kinder JSHint. Then we're free to use the punctuation and whitespace placement that makes the most sense to us and our coworkers.

Let's not even get started on the comma-first style in variable declarations..

```
var a = 1
  , b = 2
  , c = 3
  ;
```

Thou Shalt Code for Psychopaths?

We should endeavour to learn the languages we use to a good enough level that we're able to avoid cargo-culting and over-protective catch-all coding techniques. And we should trust our coworkers and other developers to do the same.

We've also discussed the abandonement of cruft in favor of taking advantage of a language's idiosyncracies and implicit rules. To some, this creates maintainability issues, especially if someone more junior in their acquisition of a given language approaches the code. For example, what if they don't know about JavaScript's weak vs. strict equality?

On the topic of maintainability, we're reminded by this famous quote:

> Always code as if the person who ends up maintaining your code is a violent psychopath who knows where you live.

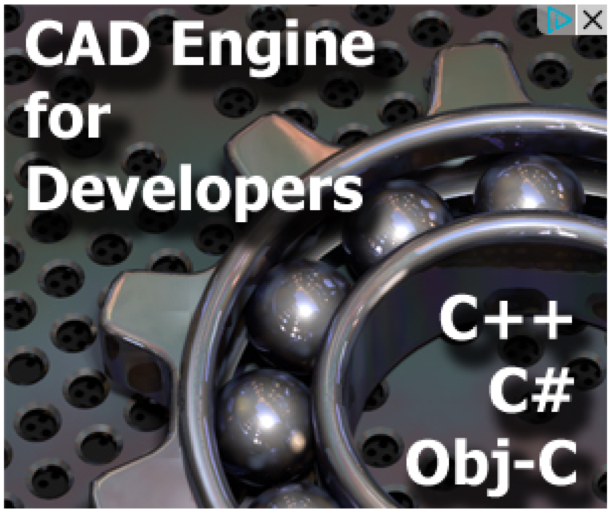
I don't know if that is truly good advice. Even taken metaphorically, it suggests a distrust of the fictional maintainer's competency — and the need to worry about their understanding above everything else. I would rather write code in the knowledge that it will be taken care of by people that know their stuff. So as a possible contradiction or even an addendum to that quote, I offer:

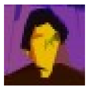
> Always code as if the person who ends up maintaing your code is knowledgeable about the language and its constructs and is seeking to gain understanding of the problem domain through reading your code.

While this may not always be true, we should seek for it be so. We should endeavour to ensure that people working on a specific technology have the sufficient understanding to do so. The learned cargo–culter says:

> If I forever pander to a lower level of understanding in my code — treading softly — strictly abiding to conventions and style guides and things I see the "experts" do, then I am never able to advance my own understanding, nor take advantage of a language in all its weirdness and beauty. I am happily and blissfully settled in this world of rules and absolutes, but to move forward, I must exit this world and embrace higher understanding.

Like 167 people like this. Be the first of your friends.



 James Padolsey is JimmyP on Codecanyon

-

-

-

-
-
-
-

By James Padolsey
I'm a freelance web developer based in Hampton, UK. I write about and enjoy front-end web development. Most of what I write focuses on my favorite topic, JavaScript! To read my blog or find out more about me please visit my site!

Note: Want to add some source code? Type <pre><code> before it and </code>

</pre> after it. Find out more