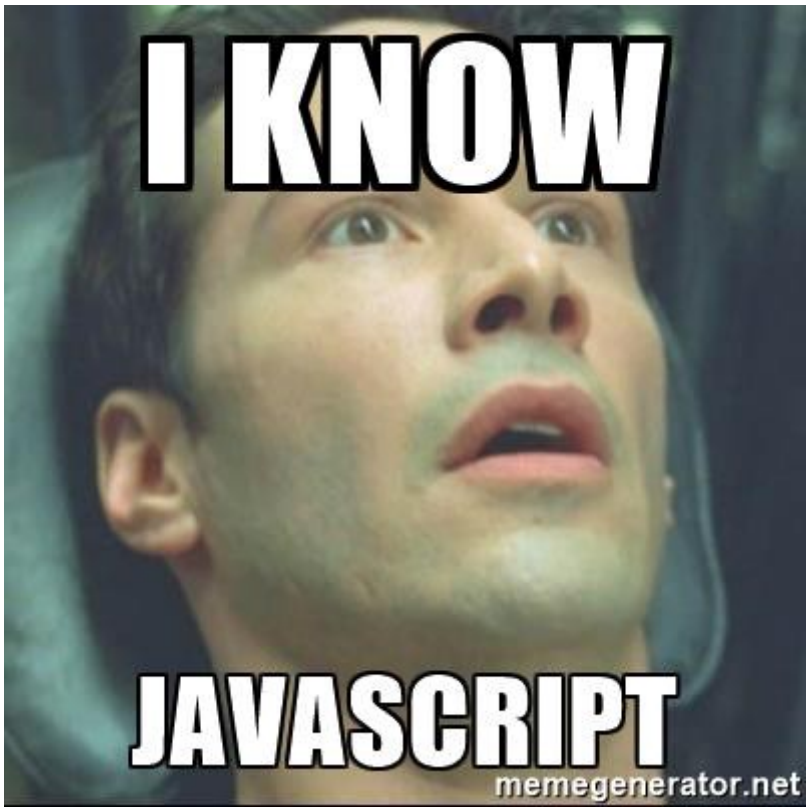
Thomas Karatzas Follow
Software Engineering at @McGillU. Interned @Microsoft. VP Internal @AiMcgill
Jul 22 · 2 min read

Avoid These Common JavaScript Mistakes



JavaScript is one of the most popular programming languages in today's industry. If you wish to dive into this language, here are a few mistakes to avoid.

1. Using "==" instead of "==="

This is probably the most common error when people start with JS. The difference between the two is that "==" will convert the type and "===" will not.

```
// examples with ==

1 == "1" // true
"/t" == 0 // true
"34" == 2 // false
new Number(10) == 10 // true
Number(10) === 10 //true

// examples with ===

1 === "1" // false
"/t" === 0 // false
"34" === 2 // false
10 === 10 //true
```

In general, you should always use the “===” also known as the Strict Equality Operator. It allows for more predictable behavior and fewer hard to track bugs.

2. Using typeof

In short, you should only use typeof to check if a variable is defined. Otherwise, you will have very inconsistent behavior.

```
console.log(typeof "foo" === "string"); //true
console.log(typeof String("foo") === "string"); // true
console.log(typeof new String("foo") === "string"); // fals
console.log(typeof 1.2 === "number"); //true
console.log(typeof new Date() === "Date"); //false Date is
console.log(typeof [1,2,3] === "array"); //false, an array

/** Proper way to test type of object **/

function is(type, obj) {
    var clas = Object.prototype.toString.call(obj).slice(8,
    return obj !== undefined && obj !== null && clas === ty
```

As you can see, the alternative can handle more general cases and allows you to be more flexible (ie: this will work with you own classes).

3. Using “this” incorrectly in a class

This is probably the most common blocking point for people getting started with JavaScript when they come from a language like Java. In Java, “this” always refers to the current object. This is not the case in JavaScript. In fact, “this” has 5 different ways of being bound.

```
// 1
console.log(this); // refers to the Global Object aka windo
// its methods include prompt alert confirm etc...

// 2
function test() {
  console.log(this);
  this.method = function inner() {console.log(this)};
}; // Here again "this" will refer to the global object
test();

//3
var obj = new test(); // Since we are using the constructor

//4
obj.method(); //infereing methods forced this to be set to

// 5: You can also force this to be a certain value using c
function foo(a, b, c) {
  this.a = a;
  this.b = b;
  ...
```

Theses 5 cases make sense and are easy to reason about. However, the second case is considered a design flaw because it leads to the following problem.

```
function test() {
  this.arr = [1,2,4];
  this.message = "I am here";
  this.fun = function() {
        this.arr.forEach(function(item) {
        console.log(this.message); // will print undefined
    });
  }
}

var t = new test();
t.fun();

// In the above code, "this" WILL NOT refer to the test obj

//to get around this, you can use a variable to store "this

// Since "this" still refers to the object, it will also ge

function test2() {
  var self = this;
  self.arr = [1,2,4];
```

4. Not using anonymous wrappers

JavaScript only has function scope and everything is shared in one global namespace. With large projects, this can introduce hard to track bugs.

```
var foo = 12;

function changeFoo() {
  foo = 34; // changes global scope and not local scope!
}

changeFoo();

console.log(foo);

// This becomes a more apparent issue in next example

// Out here is the global scope

for(var i = 0; i < 10; i++) {
    innerLoop();
```

To avoid this, you can use what are called anonymous wrappers. Essentially, they are functions that are called immediately. Since anonymous functions are considered values, they need to be evaluated before they can be callable.

Not only do they protect you from name clashes, but they can help you better organize your code.

```
(
  // evaluate the function in the parenthesis
  function(){}
  // return the evaluated function object
)() // call it immediately

// The same functionality can be done as follows;
!function(){}()
+function(){}()
(function(){}());

// PRO TIP:
// if some jerk on your team does this. AKA overwrites the

console.log(typeof undefined === "undefined") // true
```

5. Incorrectly Iterating Through Keys of an Object

There are several ways to iterate through the properties of an object. You can use Object.keys, Object.entries or a for in loop.

```
// Adding a property to the global Object => all objects in
Object.prototype.WTF = "this should not be in your object";

function a() {
  this.unwanted = 10;
  this.crap = "dhjbjbfdjbf";
}

function child() {
  this.a = 1;
  this.b = 2;
}

//child inherits from a
child.prototype = new a();

var obj = new child();

for (var property in obj) { // Not only is this slower than
        console.log(property + ": " + obj[property]);
```

6. Omitting Semi-Colons

JavaScript does add semi-colons if they are omitted but it can actually mess up your code and cause strange errors. Here are the two most popular cases:

```
/**
  Here, the parser will add a semi colon after return causi
**/
function test(){
  var name = "Hello";
  return // it will add a ; here
  {
    name: name
  }
}

/**
  This one is even weirder.....
  It does't add it in the case of a leading parenthesis
  You end up with a type error since is assumes "console.lo
**/
```

You should use a linter to make sure semi-colons are not forgotten. In addition, always put the braces on the same line of its corresponding statement to prevent unwanted behavior.

Sources:

- http://bonsaiden.github.io/JavaScript-Garden/