Javascript collection

Asked 12 years, 2 months ago Modified 6 years, 6 months ago Viewed 94k times

61

sorry for noobie question. Can you explain please, what is the difference between:

```
1. var a = [];
   a['b'] = 1;

2. var a = {};
   a['b'] = 1;
```

I could not find article in the internet so wrote here.

javascript collections

Share Improve this question Follow

asked Oct 19, 2012 at 11:52

karaxuna

26.9k ● 13 ● 84 ● 118

1 What kind of collection? – st3inn Oct 19, 2012 at 11:55

2 There are no collections in JS, just objects and arrays. I'd read up on those two types a bit more. – sync Oct 19, 2012 at 11:56

Can you tell me difference between these two? – karaxuna Oct 19, 2012 at 11:56

1 See JavaScript Object Literals & Array Literals and When to use an object or an array in javascript? – Bergi Oct 19, 2012 at 11:59

1 Check out "Associative Arrays" Considered Harmful – Bergi Oct 19, 2012 at 12:02

5 Answers

Sorted by: Highest score (default)

206

Literals

The `[]` and `{}` are called the array and object literals respectively.

`var x = []` is short for `var x = new Array();`

and `var y = {}` is short for `var y = new Object();`

Arrays

Arrays are structures with a length property. You can access values via their numeric index.

```
var x = [] or var x = new Array();
x[0] = 'b';
x[1] = 'c';
```

And if you want to list all the properties you do:

```
for(var i = 0; i < x.length; i++)
console.log(x[i]);// numeric index based access.
```

Performance tricks and gotchas

1. Inner-caching the length property

The standard array iteration:

```
for (var i = 0; i < arr.length; i++) {
    // do stuff
};
```

Little known fact: In the above scenario, the `arr.length` property is read at every step of the for loop. Just like any function you call there:

```
for (var i = 0; i < getStopIndex(); i++) {
     // do stuff
};
```

This decreases performance for no reason. Inner caching to the rescue:

```
for (var i = 0, len = arr.length; i < len; i++) {
     // enter code here
};
```

Here's proof of the above.

2. Don't specify the Array length in the constructor.

```
// doing this:
var a = new Array(100);
// is very pointless in JS. It will result in an array with 100 undefined
values.

// not even this:
var a = new Array();
// is the best way.
```

```
var a = [];
// using the array literal is the fastest and easiest way to do things.
```

Test cases for array definition are available **here**.

3. Avoid using Array.prototype.push(arr.push)

If you are dealing with large collections, direct assignment is faster than using the `Array.prototype.push();` method.

`myArray[i] = 0;` is faster than `myArray.push(0);`, according to jsPerf.com test cases.

4. It is wrong to use arrays for associative assignments.

The only reason why it works is because `Array` extends the `Object` class inside the core of the JS language. You can just as well use a `Date();` or `RegEx();` object for instance. It won't make a difference. `x['property'] = someValue` **MUST** always be used with **Objects**.

Arrays should only have numeric indexes. SEE THIS, the Google JS development guidelines! Avoid `for (x in arr)` loops or `arr['key'] = 5;`.

This can be easily backed up, look **HERE** for an example.

```
var x = [];
console.log(x.prototype.toString.call);
```

will output: `[object Array]`

This reveals the core language's 'class' inheritance pattern.

```
var x = new String();
console.log(x.prototype.toString.call);
```

will output `[object String]`.

5. Getting the minimum and maximum from an array.

A little known, but really powerful trick:

```
function arrayMax(arr) {
    return Math.max.apply(Math, arr);
};
```

, respectively:

```
function arrayMin(arr) {
    return Math.min.apply(Math, arr);
};
```

Objects

With an object you can only do:

`var y = {}` or `var y = new Object();`

`y['first'] = 'firstValue'` is the same as `y.first = 'firstValue'`, which you can't do with an array. Objects are designed for associative access with `String` keys.

And the iteration is something like this:

```
for (var property in y) {
    if (y.hasOwnProperty(property)) {
        console.log(y.property);
    };
};
```

Performance tricks and gotchas

1. Checking if an object has a property.

Most people use `Object.prototype.hasOwnProperty`. Unfortunately that often gives erroneous results leading to unexpected bugs.

Here's a good way to do it:

```
function containsKey(obj, key) {
    return typeof obj[key] !== 'undefined';
};
```

2. Replacing switch statements.

One of the simple but efficient JS tricks is `switch` replacement.

```
switch (someVar) {
    case 'a':
        doSomething();
        break;
    case 'b':
        doSomethingElse();
        break;
    default:
        doMagic();
```

```
        break;
};
```

In most JS engines the above is painfully slow. When you are looking at three possible outcomes, it doesn't make a difference, but what if you had tens or hundreds?

The above can easily be replaced with an object. Don't add the trailing `()`, this is not executing the functions, but simply storing references to them:

```
var cases = {
    'a': doSomething,
    'b': doSomethingElse,
    'c': doMagic
};
```

Instead of the `switch`:

```
var x = ???;
if (containsKey(cases, x)) {
    cases[x]();
} else {
    console.log("I don't know what to do!");
};
```

3. Deep-cloning made easy.

```
function cloneObject(obj) {
   var tmp = {};
   for (var key in obj) {
       tmp[key] = fastDeepClone(obj[key];
   };
   return tmp;
}

function cloneArr(arr) {
   var tmp = [];
   for (var i = 0, len = arr.length; i < len; i++) {
     tmp[i] = fastDeepClone(arr[i]);
   }
   return tmp;
}


function deepClone(obj) {
   return JSON.parse(JSON.stringify(obj));
};

function isArray(obj) {
   return obj instanceof Array;
}

function isObject(obj) {
```

```
  var type = typeof obj;
  return type === 'object' && obj !== null || type === 'function';
}

function fastDeepClone(obj) {
  if (isArray(obj)) {
    return cloneArr(obj);
  } else if (isObject(obj)) {
    return cloneObject(obj);
  } else {
    return obj;
  };
};
```

HERE is the deep clone function in action.

Auto-boxing

As a dynamically typed language, JavaScript is limited in terms of native object types:

- Object
- Array
- Number
- Boolean
- Date
- RegEx
- Error

> Null is not a type, `typeof null` is object.

What's the catch? There is a strong distinction between primitive and non-primitive objects.

```
var s = "str";
var s2 = new String("str");
```

They do the same thing, you can call all string methods on `s` and `s2`. Yet:

```
type of s == "string"; // raw data type
type of s2  == "object" // auto-boxed to non-primitive wrapper type
s2.prototype.toString.call == "[object String]";
```

You may hear in JS everything is an `object`. That's not exactly true, although it's a really easy mistake to make.

In reality there are 2 types, primitives and objects, and when you call `s.indexOf("c")`, the JS engine will automatically convert `s` to its non-primitive wrapper type, in this case `object String`, where all the methods are defined on the `String.prototype`.

This is called `auto-boxing`. The `Object.prototype.valueOf(obj)` method is a way to force the cast from primitive to non-primitive. It's the same behaviour a language like Java introduces for many of it's own primitives, specifically the pairs: `int` - Integer, `double` - Double, `float` - Float, etc.

Why should you care?

Simple:

```
function isString(obj) {
   return typeof obj === "string";
}
isString(s); // true
isString(s2); // false
```

So if `s2` was created with `var s2 = new String("test")` you are getting a false negative, even for an otherwise conceivably simple type check. More complex objects also bring with themselves a heavy performance penalty.

A micro-optimization as some would say, but the results are truly remarkable, even for extremely simple things such as string initialisation. Let's compare the following two in terms of performance:

```
var s1 = "this_is_a_test"
```

and

```
var s2 = new String("this_is_a_test")
```

You will probably expected matching performance across the board, but rather surprisingly the latter statement using `new String` is 92% slower than the first one, as proven here.

Functions

1. Default parameters

The `||` operator is the simplest possible way of defaulting. Why does it work? Because of truthy and falsy values.

When evaluated in a logical condition, `undefined` and `null` values will autocast to `false`.

A simple example(code **HERE**):

```
function test(x) {
   var param = x || 5;
   // do stuff with x
};
```

2. OO JS

The most important thing to understand is that the JavaScript `this` object is not immutable. It is simply a reference that can be changed with great ease.

In OO JS, we rely on the `new` keyword to guarantee implicit scope in all members of a JS Class. Even so, you can easily change the scope, via `Function.prototype.call` and `Function.prototype.apply`.

Another very important thing is the `Object.prototype`. Non-primitive values nested on an objects prototype are shared, while primitive ones are not.

Code with examples **HERE**.

A simple class definition:

```
function Size(width, height) {
    this.width = width;
    this.height = height;
};
```

A simple size class, with two members, `this.width` and `this.height`.

In a class definition, whatever has `this` in front of it, will create a new reference for every instance of Size.

Adding methods to classes and why the "closure" pattern and other "fancy name pattern" are pure fiction

This is perhaps where the most malicious JavaScript anti-patterns are found.

We can add a method to our `Size` class in two ways.

```
Size.prototype.area = function() {
   return this.width * this.height;
};
```

Or:

```
function Size2(width, height) {
   this.width = width;
   this.height = height;
   this.area = function() {
      return this.width * this.height;
   }
}

var s = new Size(5, 10);
var s2 = new Size2(5, 10);



var s3 = new Size2(5, 10);
var s4 = new Size(5, 10);
 // Looks identical, but lets use the reference equality operator to test
things:
s2.area === s3.area // false
s.area === s4.area // true
```

The `area` method of `Size2` is created for every instance. This is completely useless and slow, A LOT slower. 89% to be exact. Look **HERE**.

The above statement is valid for about 99% of all known "fancy name pattern". Remember the single most important thing in JS, all those are nothing more than fiction.

There are strong architectural arguments that can be made, mostly revolved around data encapsulation and the usage of closures.

Such things are unfortunately absolutely worthless in JavaScript, the performance loss simply isn't worth it. We are talking about 90% and above, it's anything but negligible.

3. Limitations

Because `prototype` definitions are shared among all instances of a class, you won't be able to put a non-primitive settings object there.

```
Size.prototype.settings = {};
```

Why? `size.settings` will be the same for every single instance. So what's with the primitives?

```
Size.prototype.x = 5; // won't be shared, because it's a primitive.
// see auto-boxing above for primitive vs non-primitive
// if you come from the Java world, it's the same as int and Integer.
```

The point:

The average JS guy will write JS in the following way:

```
var x = {
    doStuff: function(x) {
    },
    doMoreStuff: 5,
    someConstant: 10
}
```

Which is **fine** (fine = poor quality, hard to maintain code), as long as you understand that is a `Singleton` object, and those functions should only be used in global scope without referencing `this` inside them.

But then it gets to absolutely terrible code:

```
var x = {
   width: 10,
   height: 5
}
var y = {
   width: 15,
   height: 10
}
```

You could have gotten away with: `var x = new Size(10, 5); var y = new Size(15, 5);`.

Takes longer to type, you need to type the same thing every time. And again, it's A LOT SLOWER. Look **HERE**.

Poor standards throughout

This can be seen almost anywhere:

```
   function() {
      // some computation
      var x = 10 / 2;
      var y = 5;
      return {
         width: x,
         height: y
      }
    }
```

Again with the alternative:

```
function() {
    var x = 10 / 2;
```

```
    var y = 5;
    return new Size(10, 5);
};
```

The point: USE CLASSES WHEREVER APPROPRIATE!!

Why? Example 1 is **93% Slower**. Look **HERE**. The examples here are trivial, but they illustrate something being ignored in JS, OO.

It's a solid rule of thumb not to employ people who think JS doesn't have classes and to get jobs from recruiters talking about "Object Orientated" JS.

Closures

A lot of people prefer them to the above because it gives them a sense of data encapsulation. Besides the drastic 90% performance drop, here's something equally easy to overlook. Memory leaks.

```
function Thing(someParam) {
   this.someFn = function() {
     return someParam;
   }
}
```

You've just created a closure for `someParam`. Why is this bad? First, it forces you to define class methods as instance properties, resulting in the big performance drop.

Second, it eats up memory, because a closure will never get dereferenced. Look **here** for proof. Sure, you do get some fake data encapsulation, but you use three times the memory with a 90% performance drop.

Or you can add `@private` and get a way with an underscore function name.

Other very common ways of generating closures:

```
function bindSomething(param) {
   someDomElement.addEventListener("click", function() {
     if (param) //do something
     else // do something else
   }, false);
}
```

`param` is now a closure! How do you get rid of it? There are various tricks, some found **here**. The best possible approach, albeit more rigorous is to avoid using anonymous functions all-together, but this would require a way to specify scopes for event callbacks.

Such a mechanism is only available in Google Closure, as far as I know.

The singleton pattern

Ok, so what do I do for singletons? I don't want to store random references. Here's a wonderful idea shamelessly stolen from Google Closure's base.js

```
/**
 * Adds a {@code getInstance} static method that always return the same
instance
 * object.
 * @param {!Function} ctor The constructor for the class to add the static
 *     method to.
 */
function addSingletonGetter(ctor) {
  ctor.getInstance = function() {
    if (ctor.instance_) {
      return ctor.instance_;
    }
    return ctor.instance_ = new ctor;
  };
};
```

It's Java-esque, but it's a simple and powerful trick. You can now do:

```
project.some.namespace.StateManager = function() {
   this.x_ = 5;
};
project.some.namespace.prototype.getX = function() { return x; }
addSingletonGetter(project.some.namespace.StateManager);
```

How is this useful? Simple. In all other files, every time you need to reference `project.some.namespace.StateManager`, you can write: `project.some.namespace.StateManager.getInstance()`. This is more awesome than it looks.

You can have global state with the benefits of a class definition (inheritance, stateful members, etc.) and **without** polluting the global namespace.

The single instance pattern

You may now be tempted to do this:

```
function Thing() {
   this.someMethod = function() {..}
}
// and then use it like this:
Thing.someMethod();
```

That is another big no-no in JavaScript. Remember, the `this` object is only guaranteed to be immutable when the `new` keyword is used. The magic behind the above code is interesting. `this` is actually the global scope, so without meaning to

you are adding methods to the global object. And you guessed it, those things never get garbage collected.

There is nothing telling JavaScript to use something else. A `function` on it's own doesn't have a scope. Be really careful what you do with `static` properties. To reproduce a quote I once read, the JavaScript global object is like a public toilet. Sometimes you have no choice but to go there, yet try and minimise contact with the surfaces as much as possible.

Either stick to the above `Singleton` pattern or use a settings object nested under a namespace.

Garbage collection in JavaScript

JavaScript is a garbage collected language, but JavaScript GC is often rather poorly understood. The point is again speed. This is perhaps all too familiar.

```
// This is the top of a JavaScript file.
var a = 5;
var b = 20;
var x = {};//blabla

// more code
function someFn() {..}
```

That is bad, poor performance code. The reason is simple. JS will garbage collect a variable and free up the heap memory it holds only when that variable gets de-scoped, e.g. there are no references to it anywhere in the memory.

For example:

```
function test(someArgs) {
   var someMoreStuf = // a very big complex object;
}
test();
```

Three things:

- Function arguments are transformed into local definitions
- Inner declarations are hoisted.
- All the heap memory allocated for inner variables is freed up when the function finishes execution.

Why? Because they no longer belong to the "current" scope. They are created, used, and destroyed. There are no closures either, so all the memory you've used is freed up through garbage collection.

For that reason, you should never, your JS files should never look like this, as global scope will just keep polluting memory.

```
var x = 5;
var y = {..}; //etc;
```

Alright, now what?

Namespaces.

JS doesn't have namespaces per say, so this isn't exactly a Java equivalent, yet from a codebase administration perspective you get what you want.

```
var myProject = {};
myProject.settings = {};
myProject.controllers = {};
myProject.controlls.MainController = function() {
    // some class definition here
}
```

Beautiful. One global variable. Proper project structure. With a build phase, you can split your project across files, and get a proper dev environment.

There's no limit to what you can achieve from here.

Count your libraries

Having had the pleasure of working on countless codebases, the last and most important argument is to be very mindful of your code dependencies. I've seen programmers casually adding jQuery into the mix of the stack for a simple animation effect and so forth.

Dependency and package management is something the JavaScript world hadn't addresses for the longest time, until the creation of tools like Bower. Browsers are still somewhat slow, and even when they're fast, internet connections are slow.

In the world of Google for instance, they go through the lengths of writing entire compilers **just to save bytes**, and that approach is in many ways the right mentality to have in web programming. And I uphold Google in very high regard as their JS library powers apps like Google Maps, which are not only insanely complex, but also work everywhere.

Arguably JavaScript has an immense variety of tools available, given its popularity, accessibility, and to some extent very low quality bar the ecosystem as a whole is willing to accept.

For *Hacker News* subscribers, a day doesn't go by without a new JS library out there, and they are certainly useful but one cannot ignore the fact that many of them re-implement the exact same concerns without any strong notion of novelty or any killer ideas and improvements.

It's a strong rule of thumb to resist the urge of mixing in all the new toys before they have the time to prove their novelty and usefulness to the entire ecosystem and to strongly distinguish between Sunday coding fun and production deployments.

If your `<head></head>` tag is longer than this post, you're doing it all wrong.

Testing your knowledge of JavaScript

A few "perfectionist" level tests:

- http://perfectionkills.com/javascript-quiz/, thanks to Kangax.
- http://javascript-puzzlers.herokuapp.com/

Share

Improve this answer

Follow

edited Jun 8, 2018 at 4:35

community wiki
49 revs, 8 users 95%
flavian

`1. Inner-caching the length property` According to the tests you link to the inner-caching version is 1% slower on Safari than the non-optimised version. – Craig Oct 3, 2013 at 0:53

1 @Flavian - If you look at the results below the tests you'll find that many browsers (The latest mobile Safari, Opera, IE, Chrome) are actually slower with the "optimised" version, and it made no difference in Firefox. So the recommended optimisation is not a good thing. – Craig Oct 3, 2013 at 18:00

@Flavian Ah, I see my mistake. I knew the longer line was better but I assumed the first (blue) line next to each browser related to the first test in the list above (no-caching), and the last line (orange) was the last test (intern caching). – Craig Oct 3, 2013 at 19:35

@flavian there are several mistakes in the examples and some of the benchmarks are wrongly implemented thus leading to incorrect output, e.g. the example to create new Object, using object literal is actually faster than using new Size(...), see stackoverflow.com/questions/12973706/javascript-collection/…. – WispyCloud Jan 19, 2014 at 8:19

The fastDeepClone function has a false premise. ECMAScript doesn't limit references to just the first level of an object. A simple testing will show that this fastDeepClone will just fail to clone an object with two additional objects nested within. – Rene Saarsoo Jan 19, 2014 at 15:18

A collection of objects? Use this notation (JavaScript arrays):

8

```
var collection = [ {name:"object 1"} , {name:"object 2"} , {name:"object 3"} ];
```

To put a new element into your collection:

```
collection.push( {name:"object 4"} );
```

Share

Improve this answer

Follow

edited Oct 19, 2012 at 12:10

answered Oct 19, 2012 at 11:56

Juvanis

25.9k ● 5 ● 72 ● 88

1

In JavaScript all objects are associative arrays. In first case you create an array in the second case you created an empty object which is array too :).

So in JS you can work with any object as with array:

```
var a = {};
a["temp"] = "test";
```

And as object:

```
var a = {};
a.temp = "test";
```

Share Improve this answer Follow

answered Oct 19, 2012 at 11:57

Artem Vyshniakov

16.5k ● 3 ● 45 ● 47

1

I would use an array of objects:

```
collection = [
    { "key":"first key", "value":"first value" },
    { "key":"second key", "value":"second value" }
];
```

etc

Share Improve this answer Follow

answered Oct 19, 2012 at 11:59

st3inn

1,596 ● 1 ● 9 ● 18

Hello Please help me var visitorsData = [{"Key":"DE","Value":"3"},{"Key":"HK","Value":"2"}, {"Key":"IN","Value":"4"},{"Key":"RO","Value":"2"},{"Key":"US","Value":"2"}]; this data is get liike

this but i want like var visitorsData = {DE:3,HK:2,IN:4,RO:2,US:2}; Please help me
– user7918630 Oct 31, 2017 at 13:06

1

1) Is an Array 2) Is an Object

With Array all is usual as in other languages

With Object also. - You can get value a.b == 1 - But in JS you can also get value with such syntax a["b"] == 1

- This could be usefull when key look like something this "some key", in this case you can't use "chaining"
- also this usefull if key is the variable

you can write like this

```
    function some(f){
var Object = {name: "Boo", age: "foo"}, key;
if(f == true){
   key = "name";
}else{
   key = "age";
}
 return Object[key];
}
```

> but I want to use it as collection, which I have to choose?

This depends of what data you want to store

Share Improve this answer Follow

answered Oct 19, 2012 at 12:03

VitVad
336 ● 2 ● 8