vasco3 / es6-cheatsheet.markdown

ES6 cheatsheet

es6-cheatsheet.markdown

Table of Contents *generated with DocToc*

FrontEnd Masters - ES6 notes

Slides

- ECMAScript is now EcmaScript. Which is a standard for the API JavaScript and other languages use.
- TC39 stands for Technical Committee which regulate the EcmaScript API.
- ES.Next is a pointer to the next version of ES
- ES Harmony is the backlog of the new stuff coming to ES and the versions in development.

Proper Tail Call (PTC)

David Herman

Proper Tail Call (PTC) allows recursive calls without flooding the memory usage with garbage. The current limit of recursive calls is around 10k in Chrome and 49k in FF.

ES6 brings proper tail calls.

Tail position = the last instruction to fire before the return statement Tail call = calling another function from the tail position Close call = when the last instruction has to return to the method to do something. eg. return 1 + bar()

Only works on Strict Mode

Function Hoisting

```
// Function Declaration
function foo() {
  // code here
}
// Function Expression
var bar = function() {
  // code here
}
```

Function declaration gets hoisted to the top, while Function Expression does not.

Variables

- var: gets hoisted
- let: lives within block (curly braces)
- const: constant.. also lives within blocks

Temporal Dead Zone

```
function doSomething() {
  console.log(a); // should cause an error
  let a = 1;
  console.log(a);
}
```

Rest Parameters

Treats arguments as an array

```
function foo(...bar) {
  console.log(bar.join(' ')); // Logs 'I can haz teh arguments'
}
foo('I', 'can', 'haz', 'teh', 'arguments');
```

rules

1. It is similar to `arguments` but the rest params are a real array.
2. You just can have one rest param per function and has to be in the last position.
3. You can't use arguments

Spread Operator

Spreads an array into its individual values.

```
var a = [1, 2];
var b = returnTwo(a[0], a[1]); // [2, 1]
var c = returnTwo(...a); // [2, 1]
```

concat arrays with spread

```
let nums = [1, 2, 3];
let abcs = ['a', 'b', 'c'];
```

```
let alphanum = [ ...nums, ...abs ]; // [1, 2, 3, 'a', 'b', 'c']
```

Object short-hand

```
const x = 4;
const y = 2;

const o = { x, y, z: x * y }; // { x: 4, y: 2, z: 8 }
```

Descructuring

"Destructuring allows you to bind a set of variables to a corresponding set of values anywhere that you can normally bind a value to a single variable."

It helps pull incoming objects apart.

```
var address = {
  city: "Costa Mesa",
  state: "CA",
  zip: 92444
};
let {city, state, zip} = address;

log(city); // 'Costa Mesa'
log(state); // 'CA'
log(zip); // 92442
```

Alias

or we can use alias

```
var address = {
  city: "Costa Mesa",
  state: "CA",
  zip: 92444
};
let {city: c, state: s, zip: z} = address;

log(c, s, z); // 'Costa Mesa CA 92444'
```

Simpler way

You can also use it like

```
var person = {name: 'Aaron', age: 35};
displayPerson(person);

function displayPerson({name, age}) {
  // do something with name and age to display them
}
```

Default values

You can pass default values

```
var person = {name: 'Aaron', age: 35};
displayPerson(person);

function displayPerson({name = "No Name provided", age = 0}) {
  // do something with name and age to display them
}
```

Irrefutable pattern

The destructuring must match the object or else it will throw an error.

```
var person = {name: 'Aaron', age: 35};
let {name, age, address} = person; // throws! (irrefutable)
let {name, age, ?address} = person; // is ok because we specified address as undefin
let ?{name, age, address} = person; // Forgives the whole pattern
```

All patterns

```
let {a: x} = {}   // throw
let ?{a: x} = {}  // x = undefined
let ?{a: x} = 0   // x = undefined
let {?a: x} = {}  // x = undefined
let {?a: x} = 0   // throw
```

Patterns w/ Default Values

```
let ?{a: x = 1} = undefined   // x = 1
let {?a: x = 1} = undefined   // throw
let {?a: x = 1} = {}          // x = 1
```

Patterns - Nested

```
let person = {
  name: "Aaron",
  age: "35",
  address: {
    city: "Salt Lake City",
    state: "UT",
    zip: 84115
  }
};

let {name, age, address: {city, state, zip}} = person; // this won't create address,
```

Destructuring Arrays

```
var nums = [1, 2, 3, 4, 5];

var [first, second,,,,fifth] = nums;
log(first, second, fifth); // 1, 2, 5
```

Swapping variables

how to swap variables without using a temp var

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

// The Old Way
var temp = a, a = b, b = tmep;

// The New Way
[b, a] = [a, b];
```

Method signature

```
var nums = [1, 2, 3, 4];
doSomething(nums);

function doSomething([first, second, ...others]){
  log(first);   //logs 1
  log(second);  //logs 2
  log(others);  //logs [3, 4]
}
```

Nested Destructuring Array

```
var nums = [1, 2, [30, 40, [500, 600]]];

var [one,,[thirty,,[,sixhundert]]] = nums;
```

Pattern Errors

```
let [x] = [2, 3]    // x = 2
let [x] = {'0': 4}  // x = 4
let [x, y, z] = [1, 2]  // throw
```

Refutable

```
// Entire Pattern is Refutable
let ?[x, y, z] = [1, 2]  // x = 1, y = 2, z = undefined

// Only 'z' is Refutable
let [x, y, ?z] = [1, 2]  // z = 1, y = 2, z = undefined
```

Arrow Functions

They can't be use with `new` because of how they bind `this`.

```
var fn1 = function() {return 2;};
var fn2 = () => 2; // Here you can omit curly braces. It means return 2. If you add
```

Parenthesis-Parameter Rules

```
var x;
x = () => {};       // No parameters, MUST HAVE PARENS
x = (val) => {};    // One parameter w/ parens, OPTIONAL
x = val => {};      // One parameter w/o parens, OPTIONAL
x = (y, z) => {};   // Two or more parameters, MUST HAVE PARENS
x = y, z => {};     // Syntax Error: must wrap with parens when using multiple param
```

REAL benefit: lexical binding of 'this'

You don't need to bind(this) or var _this = this.

```
var widget = {
  init: function() {
    document.addEventListener("click", (event) => {
      this.doSomething(event.type);
    }, false);
  },
  doSomething: function(type) {
    console.log("Handling " + type + " event");
  }
};
Widget.init();
```

You can't replace all functions with Arrow functions because it will mess up `this`.

Classes

```
var monsterHealth = Symbol(); // Symbol() is a JS method that acts like a GUID gener
var monsterSpeed = Symbol();

class Monster {
  constructor(name, health, speed) {
    this.name = name;
    this[monsterHealth] = health;
    this[monsterSpeed] = speed;
  }
  // getter
  get isAlive() {
    return this[monsterHealth] > 0;
  }
  // setter
  set isAlive(alive) {
    if(!alive) this[monsterHealth] = 0;
  }
```

```
  // method
  attack(target) {
    console.log(this.name + ' attacks ' + target.name);
  }
}

var Jorge = new Monster('Jorge', 3);
Jorge.isAlive; // true

jorge.isAlive = false;
console.log(jorge.isAlive); // false
```

Classes gotchas

The following will fall in a cyclical death trap because the setter for name is already in the constructor.

```
class Monster {
  constructor(name) {
    this.name = name;
  }
  // setter
  set name (name) {
    this.name = name;
  }
}

var Jorge = new Monster('Jorge', 3);

jorge.name = 'kevin';
```

Classes don't hoist.

Extend classes

```
class Godzilla extends Monster {
    constructor() {
        super('Godzilla', 10000);
    }

    attack(target) {
      super(target); // will call the Monster attack method
    }
}
```

Collections

SET

SETs are similar to Arrays. The difference is they force unique values. No typecasting in keys.

```
var set = new Set();
set.add(1);
set.add(2);
set.add(3);
set.size; // logs 3. It is like Array.prototype.length
set.has(2); // true
set.clear(); // deletes all values
set.delete(2); // deletes value 2
```

Another way to create a Set

```
var set = new Set([1, 2, 3, 5]);
```

A new loop

```
var set = new Set([1, 2, 3, 5]);

for (let num of set) {
  console.log(num); // logs 1, 2, 3, 5
}
```

MAP

No typecasting in keys.

```
var map = new Map();
map.set('name', 'Jorge');
map.get('name'); // Jorge
map.has('name'); // true
```

Objects as keys

The key can be a function, a primitive, an object.. But it has to be exactly the same. If it is a copy or it is mutated, then it will stop working.

```
var user = { name: 'Jorge', id: 1234 };
var userHobbyMap = new Map();
userHobbyMap.set(user, ['Ice Fishing', 'Family Outting']);
```

WEAKMAP

Like a map but it doesn't has a size and no primitive keys.

It will not hold to a key that is not used by any other element. This is useful to prevent unlimited garbage. eg. when using a DOM element as a key in a map, then the DOM element gets deleted, the weakmap will delete that key-value as well.

A weakmap holds only a weak reference to a key, which means the reference inside of the weakmap doesn't prevent garbage collection of that object.

Modules

Like CommonJS

Default export

The default means will import the default export.

```
// MyClass.js
class MyClass{
  constructor() {}
}
export default MyClass;

// Main.js
import MyClass from 'MyClass';
```

Multiple exports.

You can call just the exports you need from a specific module.

```
// lib.js
export const sqrt = Math.sqrt;
export function square(x) {
  return x * x;
}
export function diag(x, y) {
  return sqrt(square(x) + square(y));
```

```
}

// main.js
import { square, diag } from 'lib';
console.log(square(11)); // 121
console.log(diag(4, 3)); // 5

// second.js
// or you can call them with '*'
// but then you have to prefix the exports with
// the module name

import * as lib from 'lib';
console.log(lib.square(11)); // 121
console.log(lib.diag(4, 3)); // 5
```

Export as

```
// lib.js
class MyClass {
  //...
}

// main.js
import { Dude as Bro } from 'lib';
var bro = new Bro(); // instanceof MyClass
```

Cyclical Dependencies

The following would be allowed

```
// lib.js
import Main from 'main';
var lib = {message: "This Is A Lib"};
export { lib as Lib };

// main.js
import { Lib } from 'lib';
export default class Main {
  // ....
}
```

More importing

```
// lib.js
// Default exports and named exports
import theDefault, { named1, named2 } from 'src/mylib';
import theDefault from 'src/mylib';
import { named1, named2 } from 'src/mylib';

// Renaming: import named1 as myNamed1
import { named1 as myNamed1, named2 } from 'src/mylib';

// Importing the module as an object
// (with one property per named export)
import * as mylib from 'src/mylib';

// Only load the module, don't import anything
import 'src/mylib';
```

More Exporting

```
export var myVar = ...;
export let myVar = ...;
export const MY_CONST = ...;

export function myFunc() {
  ...
}
export function* myGeneratorFunc() {
  ...
}
export class MyClass {
  ...
}
```

Re-exporting

This is for exporting something you are importing.

```
export * from 'src/other_module';
export { foo, bar } form 'src/other_module';

// Export other_module's foo as myFoo
export { foo as myFoo, bar } from 'src/other_module';
```

Modules - Programatic Loading API

System.import API

This method will return a promise

```
System.import('some_module')
.then(some_module => {
  ...
})
.catch(error => {
  ...
});
```

Load All

```
  Promise.all(
    ['module1', 'module2', 'module3']
    .map(x => System.import(x)))
  .then(function ([module1, module2, module3]) {
    // my code...
  });
```

System "Module" functions

```
System.import(source);
// Returns module via Promise

System.module(source, options);
// Returns module via Promise

System.set(name, module);
// Inline register a new module

System.define(name, source, options?);
// Eval code and register module
```

Module HTML Tag

To load module in the html

```
<head>
  <module import="my-module.js"></module>
</head>
```

```
<head>
  <module>
    import $ from 'lib/jquery';
    console.log('$' in this); // false becaue it won't attach the import to the wind
    // globals trapped in module

    // Other JS here
    console.log(window); // Still can call window

    // let x = 1;
    Module Tag is force strict mode
  </module>
</head>
```

Promises

Like using `Q`

Promise Constructor

```
var promise = new Promise(function(resolve, reject) {
  // do a thing, possibly async, then...

  if (/* everything turned out fine */) {
    resolve("Stuff worked!");
  } else {
    reject(Error("It broke"));
  }
});
return promise;
```

Promise Instance

A `promise` can be in 1 of 4 states

- fulfilled: successfully resolved (1)
- rejected: rejected (2)
- pending: hasn't resolved or rejected yet (undefined)
- settled: fulfilled or rejected (1 or 2)

Catch

You can use .catch instead of second handler in .then

```
get('users.all')
  .then(function(users) {
    myController.users = users;
  })
  .catch(function() {
    delete myController.users;
  });
```

All

```
var usersPromise = get('users.all');
var postsPromise = get('ports.everyone');

// Wait until BOTH are settled
Promise.all([usersPromise, postsPromise])
.then(function(results) {
  myController.users = results[0];
  myController.posts = results[1];
}, function() {
  delete myController.users;
  delete myController.posts;
});
```

Static Promise Methods

- Promise.all(iterable); // Wait until all settle
- Promise.race(iterable); // Wait until 1 settles
- Promise.reject(reason); // Create a promise that is already rejected
- Promise.resolve(value); // Create a promise that is already resolved

Generators

Generators are functions which can be exited and later re-entered. Useful for long iteration functions, so they can be paused to prevent blocking other functions for too long.

Basic Syntax

```
function* myGen() { }
// or
function *myGen() { }
```

Yield

```
function *three() {
  yield 1;
  yield 2;
  return 3;
}

var geni = three(); // starts the generator but doesn't run it
geni.next(); // runs the function for one iteration. Returns { value: 1, done: false
geni.next(); // Returns { value: 2, done: false }
geni.next(); // Returns { value: 3, done: true }. This ends the generator.
geni.next(); // Returns { value: undefined, done: true }
```

Iterating on Generators

It iterates while done = false.

```
function *foo() {
  yield 1;
  yield 2;
  yield 3;
  yield 4;
  yield 5;
  return 6;
}

for (var v of foo()) {
  console.log(v);
}
// Logs 1, 2, 3, 4, 5
```

Generator with arguments

```
function *foo(x) {
  var y = 2 * (yield (x + 1));
  var z = yield (y / 3);
  return (x + y + z); // 5 + 24 + 13
}

var genit = foo(5);

console.log(genit.next()); // { value: 6, done: false }
```

```
console.log(genit.next(12)); // { value: 8, done: false }
console.log(genit.next(13)); // { value: 42, done: true }
```

HTML Templates

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/template_strings

More info

iursevla commented on Jun 13, 2016

Thanks. Really simple and helpful cheat sheet.

nchathu2014 commented on Jun 17, 2016

cool....very useful.

jankarres commented on Jan 16, 2017

Thanks a lot for this overview!

jamesg1 commented on Mar 8, 2017

Great list!

gitexec commented on Apr 13, 2017

Thanks! Very concise and insightful cheatsheet!

devstojko commented on Jun 22, 2017

Like!

 **maldonadod** commented on Jun 29, 2017 • edited

Awesome list!
I think that in the spread operator is better say that it needs a iterator as parameter, instead of an array... is very fun play with it, there is a cool way to get an array from an iterator with this operator.

```
function *generator() {
yield 6;
yield 6;
yield 6;
};
const it = generator();
const [...nums] = it;
console.log(nums) /// [6, 6, 6]
```

 **drhenner** commented on Sep 29, 2017

This code had one too many commas:

Destructuring Arrays

```
var [first, second,,,,fifth] = nums;
```

vs

```
var [first, second,,,fifth] = nums;
```

 **DamianFekete** commented on Jul 9, 2018

Information about "Refutable/Irrefutable pattern" isn't valid (anymore).
http://2ality.com/2014/01/tc39-march-november-2013.html

> November 2013
> ES6 status: Cut from ES6 are
>
> - refutable matching. Matching is mostly irrefutable (does not throw an exception if a property is missing), as demonstrated above.
> - ...

mbrings commented on Dec 19, 2020

Thank very simple and easy