DrkSephy / es6-cheatsheet

ES2015 [ES6] cheatsheet containing tips, tricks, best practices and code snippets http://slides.com/drksephy/ecmascript...

#javascript #es6-javascript #cheatsheet

154 commits | 1 branch | 0 releases | 41 contributors

Branch: master | New pull request | Find file | Clone or download

DrkSephy Fixing internal license anchor — Latest commit a03267c on Feb 8

src	Converting repository to JavaScript	2 years ago
README.md	Fixing internal license anchor	4 months ago
README_ko.md	Fixing internal license anchor	4 months ago
README_zhCn.md	Fixing internal license anchor	4 months ago

README.md

es6-cheatsheet

A cheatsheet containing ES2015 [ES6] tips, tricks, best practices and code snippet examples for your day to day workflow. Contributions are welcome!

Table of Contents

- var versus let / const
- Replacing IIFEs with Blocks
- Arrow Functions
- Strings
- Destructuring
- Modules
- Parameters
- Classes
- Symbols
- Maps
- WeakMaps
- Promises
- Generators
- Async Await
- Getter/Setter functions
- License

var versus let / const

> Besides `var`, we now have access to two new identifiers for storing values — `let` and `const`. Unlike `var`, `let` and `const` statements are not hoisted to the top of their enclosing scope.

An example of using `var`:

```
var snack = 'Meow Mix';

function getFood(food) {
    if (food) {
        var snack = 'Friskies';
        return snack;
    }
    return snack;
}

getFood(false); // undefined
```

However, observe what happens when we replace `var` using `let`:

```
let snack = 'Meow Mix';

function getFood(food) {
    if (food) {
        let snack = 'Friskies';
        return snack;
    }
    return snack;
}

getFood(false); // 'Meow Mix'
```

This change in behavior highlights that we need to be careful when refactoring legacy code which uses `var`. Blindly replacing instances of `var` with `let` may lead to unexpected behavior.

> **Note**: `let` and `const` are block scoped. Therefore, referencing block-scoped identifiers before they are defined will produce a `ReferenceError`.

```
console.log(x); // ReferenceError: x is not defined

let x = 'hi';
```

> **Best Practice**: Leave `var` declarations inside of legacy code to denote that it needs to be carefully refactored. When working on a new codebase, use `let` for variables that will change their value over time, and `const` for variables which cannot be reassigned.

(back to table of contents)

Replacing IIFEs with Blocks

> A common use of **Immediately Invoked Function Expressions** is to enclose values within its scope. In ES6, we now have the ability to create block-based scopes and therefore are not limited purely to function-based scope.

```
(function () {
    var food = 'Meow Mix';
}());

console.log(food); // Reference Error
```

Using ES6 Blocks:

```
{
    let food = 'Meow Mix';
};

console.log(food); // Reference Error
```

(back to table of contents)

Arrow Functions

Often times we have nested functions in which we would like to preserve the context of `this` from its lexical scope. An example is shown below:

```
function Person(name) {
    this.name = name;
}

Person.prototype.prefixName = function (arr) {
    return arr.map(function (character) {
        return this.name + character; // Cannot read property 'name' of undefined
    });
};
```

One common solution to this problem is to store the context of `this` using a variable:

```
function Person(name) {
    this.name = name;
}

Person.prototype.prefixName = function (arr) {
    var that = this; // Store the context of this
    return arr.map(function (character) {
        return that.name + character;
    });
};
```

We can also pass in the proper context of `this`:

```
function Person(name) {
    this.name = name;
}

Person.prototype.prefixName = function (arr) {
    return arr.map(function (character) {
        return this.name + character;
    }, this);
};
```

As well as bind the context:

```
function Person(name) {
    this.name = name;
}

Person.prototype.prefixName = function (arr) {
    return arr.map(function (character) {
        return this.name + character;
    }.bind(this));
};
```

Using **Arrow Functions**, the lexical value of `this` isn't shadowed and we can re-write the above as shown:

```
function Person(name) {
    this.name = name;
}

Person.prototype.prefixName = function (arr) {
    return arr.map(character => this.name + character);
};
```

> **Best Practice**: Use **Arrow Functions** whenever you need to preserve the lexical value of `this`.

Arrow Functions are also more concise when used in function expressions which simply return a value:

```
var squares = arr.map(function (x) { return x * x }); // Function Expression


const arr = [1, 2, 3, 4, 5];
const squares = arr.map(x => x * x); // Arrow Function for terser implementation
```

> **Best Practice**: Use **Arrow Functions** in place of function expressions when possible.

(back to table of contents)

Strings

With ES6, the standard library has grown immensely. Along with these changes are new methods which can be used on strings, such as `.includes()` and `.repeat()`.

.includes( )

```
var string = 'food';
var substring = 'foo';

console.log(string.indexOf(substring) > -1);
```

Instead of checking for a return value `> -1` to denote string containment, we can simply use `.includes()` which will return a boolean:

```
const string = 'food';
const substring = 'foo';

console.log(string.includes(substring)); // true
```

.repeat( )

```
function repeat(string, count) {
    var strings = [];
    while(strings.length < count) {
        strings.push(string);
    }
    return strings.join('');
}
```

In ES6, we now have access to a terser implementation:

```
// String.repeat(numberOfRepetitions)
'meow'.repeat(3); // 'meowmeowmeow'
```

Template Literals

Using **Template Literals**, we can now construct strings that have special characters in them without needing to escape them explicitly.

```
var text = "This string contains \"double quotes\" which are escaped.";
```

```
let text = `This string contains "double quotes" which don't need to be escaped anymore.`;
```

Template Literals also support interpolation, which makes the task of concatenating strings and values:

```
var name = 'Tiger';
var age = 13;

console.log('My cat is named ' + name + ' and is ' + age + ' years old.');
```

Much simpler:

```
const name = 'Tiger';
const age = 13;

console.log(`My cat is named ${name} and is ${age} years old.`);
```

In ES5, we handled new lines as follows:

```
var text = (
    'cat\n' +
    'dog\n' +
    'nickelodeon'
);
```

Or:

```
var text = [
    'cat',
    'dog',
    'nickelodeon'
].join('\n');
```

Template Literals will preserve new lines for us without having to explicitly place them in:

```
let text = ( `cat
dog
nickelodeon`
);
```

Template Literals can accept expressions, as well:

```
let today = new Date();
let text = `The time and date is ${today.toLocaleString()}`;
```

(back to table of contents)

Destructuring

Destructuring allows us to extract values from arrays and objects (even deeply nested) and store them in variables with a more convenient syntax.

Destructuring Arrays

```
var arr = [1, 2, 3, 4];
var a = arr[0];
var b = arr[1];
var c = arr[2];
var d = arr[3];
```

```
let [a, b, c, d] = [1, 2, 3, 4];

console.log(a); // 1
console.log(b); // 2
```

Destructuring Objects

```
var luke = { occupation: 'jedi', father: 'anakin' };
var occupation = luke.occupation; // 'jedi'
var father = luke.father; // 'anakin'
```

```
let luke = { occupation: 'jedi', father: 'anakin' };
let {occupation, father} = luke;

console.log(occupation); // 'jedi'
console.log(father); // 'anakin'
```

(back to table of contents)

Modules

Prior to ES6, we used libraries such as Browserify to create modules on the client-side, and require in **Node.js**. With ES6, we can now directly use modules of all types (AMD and CommonJS).

Exporting in CommonJS

```
module.exports = 1;
module.exports = { foo: 'bar' };
module.exports = ['foo', 'bar'];
module.exports = function bar () {};
```

Exporting in ES6

With ES6, we have various flavors of exporting. We can perform **Named Exports**:

```
export let name = 'David';
export let age  = 25;
```

As well as **exporting a list** of objects:

```
function sumTwo(a, b) {
    return a + b;
}

function sumThree(a, b, c) {
```

```
    return a + b + c;
}

export { sumTwo, sumThree };
```

We can also export functions, objects and values (etc.) simply by using the `export` keyword:

```
export function sumTwo(a, b) {
    return a + b;
}

export function sumThree(a, b, c) {
    return a + b + c;
}
```

And lastly, we can **export default bindings**:

```
function sumTwo(a, b) {
    return a + b;
}

function sumThree(a, b, c) {
    return a + b + c;
}

let api = {
    sumTwo,
    sumThree
};

export default api;

/* Which is the same as
 * export { api as default };
 */
```

> **Best Practices**: Always use the `export default` method at **the end** of the module. It makes it clear what is being exported, and saves time by having to figure out what name a value was exported as. More so, the common practice in CommonJS modules is to export a single value or object. By sticking to this paradigm, we make our code easily readable and allow ourselves to interpolate between CommonJS and ES6 modules.

Importing in ES6

ES6 provides us with various flavors of importing. We can import an entire file:

```
import 'underscore';
```

> It is important to note that simply **importing an entire file will execute all code at the top level of that file**.

Similar to Python, we have named imports:

```
import { sumTwo, sumThree } from 'math/addition';
```

We can also rename the named imports:

```
import {
    sumTwo as addTwoNumbers,
    sumThree as sumThreeNumbers
} from 'math/addition';
```

In addition, we can **import all the things** (also called namespace import):

```
import * as util from 'math/addition';
```

Lastly, we can import a list of values from a module:

```
import * as additionUtil from 'math/addition';
const { sumTwo, sumThree } = additionUtil;
```

Importing from the default binding like this:

```
import api from 'math/addition';
// Same as: import { default as api } from 'math/addition';
```

While it is better to keep the exports simple, but we can sometimes mix default import and mixed import if needed. When we are exporting like this:

```
// foos.js
export { foo as default, foo1, foo2 };
```

We can import them like the following:

```
import foo, { foo1, foo2 } from 'foos';
```

When importing a module exported using commonjs syntax (such as React) we can do:

```
import React from 'react';
const { Component, PropTypes } = React;
```

This can also be simplified further, using:

```
import React, { Component, PropTypes } from 'react';
```

> **Note**: Values that are exported are **bindings**, not references. Therefore, changing the binding of a variable in one module will affect the value within the exported module. Avoid changing the public interface of these exported values.

(back to table of contents)

Parameters

In ES5, we had varying ways to handle functions which needed **default values**, **indefinite arguments**, and **named parameters**. With ES6, we can accomplish all of this and more using more concise syntax.

Default Parameters

```
function addTwoNumbers(x, y) {
    x = x || 0;
    y = y || 0;
    return x + y;
}
```

In ES6, we can simply supply default values for parameters in a function:

```
function addTwoNumbers(x=0, y=0) {
    return x + y;
}


addTwoNumbers(2, 4); // 6
addTwoNumbers(2); // 2
addTwoNumbers(); // 0
```

Rest Parameters

In ES5, we handled an indefinite number of arguments like so:

```
function logArguments() {
    for (var i=0; i < arguments.length; i++) {
        console.log(arguments[i]);
    }
}
```

Using the **rest** operator, we can pass in an indefinite amount of arguments:

```
function logArguments(...args) {
    for (let arg of args) {
        console.log(arg);
    }
}
```

Named Parameters

One of the patterns in ES5 to handle named parameters was to use the **options object** pattern, adopted from jQuery.

```
function initializeCanvas(options) {
    var height = options.height || 600;
    var width  = options.width  || 400;
    var lineStroke = options.lineStroke || 'black';
}
```

We can achieve the same functionality using destructuring as a formal parameter to a function:

```
function initializeCanvas(
    { height=600, width=400, lineStroke='black'}) {
        // Use variables height, width, lineStroke here
    }
```

If we want to make the entire value optional, we can do so by destructuring an empty object:

```
function initializeCanvas(
    { height=600, width=400, lineStroke='black'} = {}) {
        // ...
    }
```

Spread Operator

In ES5, we could find the max of values in an array by using the `apply` method on `Math.max` like this:

```
Math.max.apply(null, [-1, 100, 9001, -32]); // 9001
```

In ES6, we can now use the spread operator to pass an array of values to be used as parameters to a function:

```
Math.max(...[-1, 100, 9001, -32]); // 9001
```

We can concat array literals easily with this intuitive syntax:

```
let cities = ['San Francisco', 'Los Angeles'];
let places = ['Miami', ...cities, 'Chicago']; // ['Miami', 'San Francisco', 'Los Angeles', 'Chicago']
```

(back to table of contents)

Classes

Prior to ES6, we implemented Classes by creating a constructor function and adding properties by extending the prototype:

```
function Person(name, age, gender) {
    this.name   = name;
    this.age    = age;
    this.gender = gender;
}

Person.prototype.incrementAge = function () {
    return this.age += 1;
};
```

And created extended classes by the following:

```
function Personal(name, age, gender, occupation, hobby) {
    Person.call(this, name, age, gender);
    this.occupation = occupation;
    this.hobby = hobby;
}

Personal.prototype = Object.create(Person.prototype);
Personal.prototype.constructor = Personal;
Personal.prototype.incrementAge = function () {
    Person.prototype.incrementAge.call(this);
    this.age += 20;
    console.log(this.age);
};
```

ES6 provides much needed syntactic sugar for doing this under the hood. We can create Classes directly:

```
class Person {
    constructor(name, age, gender) {
        this.name   = name;
        this.age    = age;
        this.gender = gender;
    }

    incrementAge() {
      this.age += 1;
    }
}
```

And extend them using the `extends` keyword:

```
class Personal extends Person {
    constructor(name, age, gender, occupation, hobby) {
        super(name, age, gender);
```

```
        this.occupation = occupation;
        this.hobby = hobby;
    }

    incrementAge() {
        super.incrementAge();
        this.age += 20;
        console.log(this.age);
    }
}
```

> **Best Practice**: While the syntax for creating classes in ES6 obscures how implementation and prototypes work under the hood, it is a good feature for beginners and allows us to write cleaner code.

(back to table of contents)

Symbols

Symbols have existed prior to ES6, but now we have a public interface to using them directly. Symbols are immutable and unique and can be used as keys in any hash.

Symbol( )

Calling `Symbol()` or `Symbol(description)` will create a unique symbol that cannot be looked up globally. A Use case for `Symbol()` is to patch objects or namespaces from third parties with your own logic, but be confident that you won't collide with updates to that library. For example, if you wanted to add a method `refreshComponent` to the `React.Component` class, and be certain that you didn't trample a method they add in a later update:

```
const refreshComponent = Symbol();

React.Component.prototype[refreshComponent] = () => {
    // do something
}
```

Symbol.for(key)

`Symbol.for(key)` will create a Symbol that is still immutable and unique, but can be looked up globally. Two identical calls to `Symbol.for(key)` will return the same Symbol instance. NOTE: This is not true for `Symbol(description)` :

```
Symbol('foo') === Symbol('foo') // false
Symbol.for('foo') === Symbol('foo') // false
Symbol.for('foo') === Symbol.for('foo') // true
```

A common use case for Symbols, and in particular with `Symbol.for(key)` is for interoperability. This can be achieved by having your code look for a Symbol member on object arguments from third parties that contain some known interface. For example:

```
function reader(obj) {
    const specialRead = Symbol.for('specialRead');
    if (obj[specialRead]) {
        const reader = obj[specialRead]();
        // do something with reader
    } else {
        throw new TypeError('object cannot be read');
    }
}
```

And then in another library:

```
const specialRead = Symbol.for('specialRead');

class SomeReadableType {
    [specialRead]() {
        const reader = createSomeReaderFrom(this);
        return reader;
    }
}
```

> A notable example of Symbol use for interoperability is `Symbol.iterator` which exists on all iterable types in ES6: Arrays, strings, generators, etc. When called as a method it returns an object with an Iterator interface.

(back to table of contents)

Maps

Maps is a much needed data structure in JavaScript. Prior to ES6, we created **hash** maps through objects:

```
var map = new Object();
map[key1] = 'value1';
map[key2] = 'value2';
```

However, this does not protect us from accidentally overriding functions with specific property names:

```
> getOwnProperty({ hasOwnProperty: 'Hah, overwritten'}, 'Pwned');
> TypeError: Property 'hasOwnProperty' is not a function
```

Actual **Maps** allow us to `set`, `get` and `search` for values (and much more).

```
let map = new Map();
> map.set('name', 'david');
> map.get('name'); // david
> map.has('name'); // true
```

The most amazing part of Maps is that we are no longer limited to just using strings. We can now use any type as a key, and it will not be type-cast to a string.

```
let map = new Map([
    ['name', 'david'],
    [true, 'false'],
    [1, 'one'],
    [{}, 'object'],
    [function () {}, 'function']
]);

for (let key of map.keys()) {
    console.log(typeof key);
    // > string, boolean, number, object, function
}
```

> **Note**: Using non-primitive values such as functions or objects won't work when testing equality using methods such as `map.get()`. As such, stick to primitive values such as Strings, Booleans and Numbers.

We can also iterate over maps using `.entries()`:

```
for (let [key, value] of map.entries()) {
    console.log(key, value);
}
```

(back to table of contents)

WeakMaps

In order to store private data versions < ES6, we had various ways of doing this. One such method was using naming conventions:

```
class Person {
    constructor(age) {
        this._age = age;
    }

    _incrementAge() {
        this._age += 1;
    }
}
```

But naming conventions can cause confusion in a codebase and are not always going to be upheld. Instead, we can use WeakMaps to store our values:

```
let _age = new WeakMap();
class Person {
    constructor(age) {
        _age.set(this, age);
    }

    incrementAge() {
        let age = _age.get(this) + 1;
        _age.set(this, age);
        if (age > 50) {
            console.log('Midlife crisis');
        }
    }
}
```

The cool thing about using WeakMaps to store our private data is that their keys do not give away the property names, which can be seen by using `Reflect.ownKeys()`:

```
> const person = new Person(50);
> person.incrementAge(); // 'Midlife crisis'
> Reflect.ownKeys(person); // []
```

A more practical example of using WeakMaps is to store data which is associated to a DOM element without having to pollute the DOM itself:

```
let map = new WeakMap();
let el  = document.getElementById('someElement');

// Store a weak reference to the element with a key
map.set(el, 'reference');

// Access the value of the element
let value = map.get(el); // 'reference'

// Remove the reference
el.parentNode.removeChild(el);
el = null;

// map is empty, since the element is destroyed
```

As shown above, once the object is destroyed by the garbage collector, the WeakMap will automatically remove the key-value pair which was identified by that object.

> **Note**: To further illustrate the usefulness of this example, consider how jQuery stores a cache of objects corresponding to DOM elements which have references. Using WeakMaps, jQuery can automatically free up any memory that was associated with a particular DOM element once it has been removed from the document. In general, WeakMaps are very useful for any library that wraps DOM elements.

(back to table of contents)

Promises

Promises allow us to turn our horizontal code (callback hell):

```
func1(function (value1) {
    func2(value1, function (value2) {
        func3(value2, function (value3) {
            func4(value3, function (value4) {
                func5(value4, function (value5) {
                    // Do something with value 5
                });
            });
        });
    });
});
```

Into vertical code:

```
func1(value1)
    .then(func2)
    .then(func3)
    .then(func4)
    .then(func5, value5 => {
        // Do something with value 5
    });
```

Prior to ES6, we used bluebird or Q. Now we have Promises natively:

```
new Promise((resolve, reject) =>
    reject(new Error('Failed to fulfill Promise')))
        .catch(reason => console.log(reason));
```

Where we have two handlers, **resolve** (a function called when the Promise is **fulfilled**) and **reject** (a function called when the Promise is **rejected**).

> **Benefits of Promises**: Error Handling using a bunch of nested callbacks can get chaotic. Using Promises, we have a clear path to bubbling errors up and handling them appropriately. Moreover, the value of a Promise after it has been resolved/rejected is immutable - it will never change.

Here is a practical example of using Promises:

```
var request = require('request');

return new Promise((resolve, reject) => {
  request.get(url, (error, response, body) => {
    if (body) {
        resolve(JSON.parse(body));
      } else {
        resolve({});
      }
```

```
  });
});
```

We can also **parallelize** Promises to handle an array of asynchronous operations by using `Promise.all()`:

```
let urls = [
  '/api/commits',
  '/api/issues/opened',
  '/api/issues/assigned',
  '/api/issues/completed',
  '/api/issues/comments',
  '/api/pullrequests'
];

let promises = urls.map((url) => {
  return new Promise((resolve, reject) => {
    $.ajax({ url: url })
      .done((data) => {
        resolve(data);
      });
  });
});

Promise.all(promises)
  .then((results) => {
    // Do something with results of all our promises
 });
```

(back to table of contents)

Generators

Similar to how Promises allow us to avoid callback hell, Generators allow us to flatten our code - giving our asynchronous code a synchronous feel. Generators are essentially functions which we can pause their execution and subsequently return the value of an expression.

A simple example of using generators is shown below:

```
function* sillyGenerator() {
    yield 1;
    yield 2;
    yield 3;
    yield 4;
}

var generator = sillyGenerator();
> console.log(generator.next()); // { value: 1, done: false }
> console.log(generator.next()); // { value: 2, done: false }
> console.log(generator.next()); // { value: 3, done: false }
> console.log(generator.next()); // { value: 4, done: false }
```

Where next will allow us to push our generator forward and evaluate a new expression. While the above example is extremely contrived, we can utilize Generators to write asynchronous code in a synchronous manner:

```
// Hiding asynchronousity with Generators

function request(url) {
    getJSON(url, function(response) {
        generator.next(response);
    });
}
```

And here we write a generator function that will return our data:

```
function* getData() {
    var entry1 = yield request('http://some_api/item1');
    var data1  = JSON.parse(entry1);
    var entry2 = yield request('http://some_api/item2');
    var data2  = JSON.parse(entry2);
}
```

By the power of `yield`, we are guaranteed that `entry1` will have the data needed to be parsed and stored in `data1`.

While generators allow us to write asynchronous code in a synchronous manner, there is no clear and easy path for error propagation. As such, as we can augment our generator with Promises:

```
function request(url) {
    return new Promise((resolve, reject) => {
        getJSON(url, resolve);
    });
}
```

And we write a function which will step through our generator using `next` which in turn will utilize our `request` method above to yield a Promise:

```
function iterateGenerator(gen) {
    var generator = gen();
    (function iterate(val) {
        var ret = generator.next();
        if(!ret.done) {
            ret.value.then(iterate);
        }
    })();
}
```

By augmenting our Generator with Promises, we have a clear way of propagating errors through the use of our Promise `.catch` and `reject`. To use our newly augmented Generator, it is as simple as before:

```
iterateGenerator(function* getData() {
    var entry1 = yield request('http://some_api/item1');
    var data1  = JSON.parse(entry1);
    var entry2 = yield request('http://some_api/item2');
    var data2  = JSON.parse(entry2);
});
```

We were able to reuse our implementation to use our Generator as before, which shows their power. While Generators and Promises allow us to write asynchronous code in a synchronous manner while retaining the ability to propagate errors in a nice way, we can actually begin to utilize a simpler construction that provides the same benefits: async-await.

(back to table of contents)

Async Await

While this is actually an upcoming ES2016 feature, `async await` allows us to perform the same thing we accomplished using Generators and Promises with less effort:

```
var request = require('request');

function getJSON(url) {
  return new Promise(function(resolve, reject) {
    request(url, function(error, response, body) {
      resolve(body);
```

```
    });
  });
}

async function main() {
  var data = await getJSON();
  console.log(data); // NOT undefined!
}

main();
```

Under the hood, it performs similarly to Generators. I highly recommend using them over Generators + Promises. A great resource for getting up and running with ES7 and Babel can be found here.

(back to table of contents)

Getter and setter functions

ES6 has started supporting getter and setter functions within classes. Using the following example:

```
class Employee {

    constructor(name) {
        this._name = name;
    }

    get name() {
      if(this._name) {
        return 'Mr. ' + this._name.toUpperCase();
      } else {
        return undefined;
      }
    }

    set name(newName) {
      if (newName == this._name) {
        console.log('I already have this name.');
      } else if (newName) {
        this._name = newName;
      } else {
        return false;
      }
    }
}

var emp = new Employee("James Bond");

// uses the get method in the background
if (emp.name) {
  console.log(emp.name);  // Mr. JAMES BOND
}

// uses the setter in the background
emp.name = "Bond 007";
console.log(emp.name);  // Mr. BOND 007
```

Latest browsers are also supporting getter/setter functions in Objects and we can use them for computed properties, adding listeners and preprocessing before setting/getting:

```
var person = {
  firstName: 'James',
  lastName: 'Bond',
  get fullName() {
      console.log('Getting FullName');
      return this.firstName + ' ' + this.lastName;
```

```
  },
  set fullName (name) {
      console.log('Setting FullName');
      var words = name.toString().split(' ');
      this.firstName = words[0] || '';
      this.lastName = words[1] || '';
  }
}

person.fullName; // James Bond
person.fullName = 'Bond 007';
person.fullName; // Bond 007
```

(back to table of contents)

License

(back to table of contents)