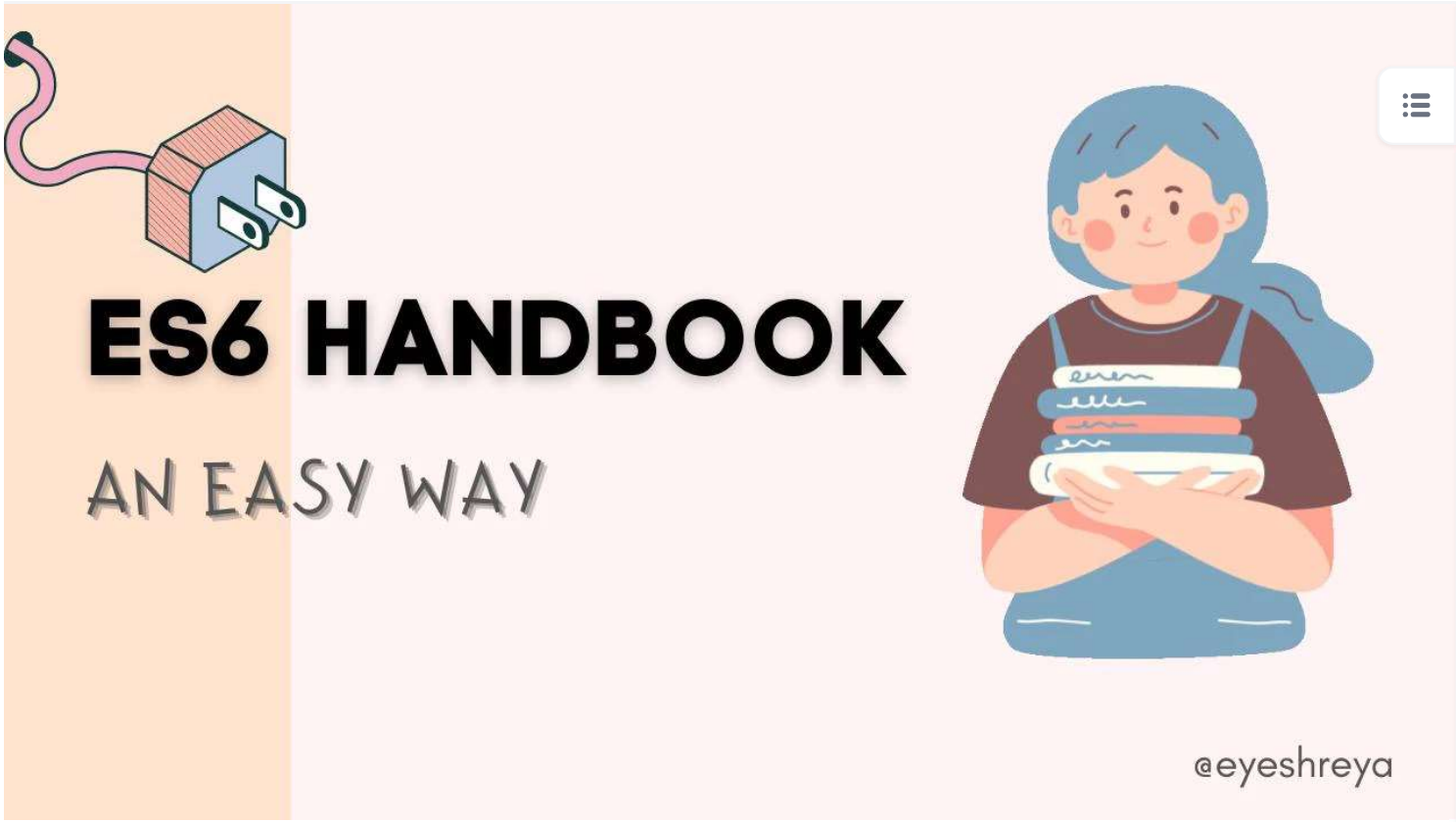

ES6 Handbook: The Complete Guide

Shreya · September 20, 2021
7 minute read · Updated September 27, 2021

#javascript #Es6 #guide #es6 guide

Hey readers! This blog is all about ES6. It includes all the topics related with examples. Before reading further, I want to specify that this was not a blog post initially, these are just my personal notes that I use as a reference guide, so I apologize for any misspells here :)

Table of Contents

Notes

let/const

Before moving to the point, let us understand two concepts here:

1. **Global Scope** - Variable is declared outside the function. This variable is accessible inside every function present in the code.
2. **Function Scope** - Variable is declared inside (within) a function, outside that it is not accessible anywhere.
3. **Block Scope** - In short, block scope means variables which are declared in a { } block are not accessible outside it. This block can be an `if` statement, `for` / `while` loop, etc.

`var` : function/ global scoped. Eg:

```
var name = 'Jack';              // global scope

function message() {
    var msg = 'Hey Jack!'       //function scope
}

console.log(msg);               // ERROR
```

The above line of code will throw an error as there's no variable **msg** outside the function **message** (where we have logged the variable). So it will show as **undefined** .

let : block scoped. Eg:

→ **let** keyword can't be redeclared:

```
let x = 1;
let x = 3;
```

> *"result: SyntaxError - redeclaration of let x"*

```
let size = "big";

function box() {
    for (let x = 0; x < 7; x++) {
        console.log(size);

        //Output: ReferenceError - `size` is not defined

        let size = "small";
        console.log(size);
    }
}

box();                          // small
console.log(size);              //big
```

Inside the function **`box()`** when we log the value of size, it shows a reference error. That is because, **`let`** is block scoped.

Anything inside curly braces { } is block scoped. In the above scenario, the function **`box()`** is a block.

`const` : block scoped. Eg:

`const` are very similar to **`let`** except that they can't be changed and redeclared.

```
const m = 8;
console.log(m);                 //m = 8

m = 5;                          // 🚫 this will throw an error
console.log(m);

// Uncaught TypeError: invalid assignment to const 'm'.
}
```

→ therefore **`let`** and **`const`** are preferred over **`var`** keyword for declaring variables.

Objects

objects are written within curly braces **`{ }`** as collection of key:value pairs.

value : value of that property

Creating an empty object:

```
const car = {
    model: 'Tesla',
    color: 'black',
    price: 800
}
```

Talking specifically about ES6, before ES6 we had to specify both (key, value) even if both are of same names.

```
function Boy(name, age) {
    return(
        name: name,
        age: age
    );
}
```

ES6 help us to get rid of duplication when we have same key:value names. So now our code will look like this:

```
    return(name, age);
}
```

this

this is a keyword. It basically returns a reference to the object it is placed within

💡 **NOTE**:

> When we call a function as a method in an object, the **this** keyword returns a reference to that object. 👇

```
const user = {
    name: 'Mike';
    call() {
        console.log(this);
    }
}
user.call();

// ⚙️ Output: {name: 'Mike, call: f}
```

but when we call the function alone, outside the object "**this**" returns the global object (browser window) and hence we get the result as ***undefined*** 👇

```
const user = {
    name: 'Mike';
    call() {
        console.log(this);
    }
}

const myCall = user.call;

myCall();

// ⚙️ Output: undefined
```

Arrow Functions

Normally, before ES6:

```
    return num * num;
}
```

In ES6:

```
const square = num => num * num;
```

array.map()

If we have an array -

```
const colors = ["red", "green", "blue"];
```

We want to map the objects. Now there are two methods, `es6` one is shorter and easier.

normal case:

```
const items1 = colors.map(function (color) {
    return "<li>" + color + "</li>";
});
```

```
const items2 = colors.map((color) => `<li> ${color}  </li>`);
```

Object Destructuring

Let's say we have an object called `girl` such that it has 3 keys as follows:

```
const girl = {
    name: "",
    age: "",
    country: "",
};
```

Normally, we would do something like this to get the values:

```
const name = girl.name;
const age = girl.age;
const country = girl.country;
```

destructuring:

```
const { name, age, country } = girl;
```

> *"this one line code works same as the previous code. So destructuring made our code shorter and easier to understand."*

In case you want to use an alias (a different variable name) for your work:

```
const {country: ctry} = girl;
```

This above line of code means we've defined a new variable called `ctry` and set that equals to `country` .

Spread Operator

> *"CASE I - COMBINING ARRAYS"*

If we want to combine these two arrays:

```
const one = [1, 2, 3];
const two = [4, 5, 6];
```

```
const combined = one.concat(two);
```

With ES6:

```
const combined = [...one, ...two];
```

If we want to add things in-between:

```
const combined = [...one, '9', '7', ...two ];
```

If we want to clone an array:

```
const myDupli = [...two];
```

> *"CASE II - COMBINING OBJECTS"*

If we want to combine these two objects:

```
const alpha = { name: 'Shreya' };
const beta = { age: 19 };
```

In ES6:

```
const combined = {...alpha, ...beta};
```

If we want to add more properties in b/w:

cloning an object:

```
const betaV2 = {...beta};
```

Classes

Let us take an example of an object `boy` here. We have a function called `run` inside it. Now if we've some bug in the future or we've to modify our function for a different object, it would be a long way.

```
const boy = {
    name: "Sam",
    run() {
        console.log("running...");
    },
};
```

To overcome this and make our work easier, we use classes:

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

    run() {
        console.log("running...");
    }
}
```

Now that we've created a class, let's try building our object again -

```
const boy = new Boy("Samridh");
```

with this above class, we've implemented the run method in a single line of code. If someday we find a bug here, we've to modify it in just a single place {inside class Boy}. So this is the advantage of using classes in JS.

Inheritance

If we have a class Boy such that -

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

    run() {
        console.log("running");
    }
}
```

and we want to create another class (having similar properties + some specific properties of its own). We can do this using the keyword **extends**

```
class Girl extends Boy {
    eat() {
        console.log("eating");
    }
}
```

we just created the class **Girl** here. Let us now create a const using this -

```
const myGirl = new Girl("Shreya");
```

and we're done. This code basically means that now the const **myGirl** will have the functions **eat** + **run** + **constructor** property of **Boy** class.

```
myGirl.eat();
myGirl.run();
```

Now let's say we want to create another constructor inside the `Girl` class {which is extended from `Boy` class, So the constructor inside this `Girl` class is called **derived class constructor.**}.

We MUST HAVE TO call `super()` constructor inside the new constructor, otherwise we'll get an error (as using `this` in derived class constructor requires `super()` class). Now this must be looking confusing, let's look at the example below -

```
class Girl extends Boy {
    constructor(age) {
        this.age = age;
    }
    eat() {
        console.log("eating");
    }
}

// *result - Uncaught ReferenceError: must call super constructor be
```

```
class Girl extends Boy {
    constructor(name, age) {
        super(name);
        this.age = age;
    }
    eat() {
        console.log("eating");
    }
}
const myGirl = new Girl("Shreya");
```

In a child class constructor, `this` cannot be used until `super` is called.

Modules

Sometimes we have many no. of classes declared in a single file. This makes the code long, confusing and messy. To avoid this, we separate these classes into different files and import them as a `module` into the main file. This is called modularity.

Let's look it in action. Here's what our folder `src` will look like:

```
// src/boy.js

export class Boy {
    constructor(name) {
        this.name = name;
    }

    run() {
        console.log("running");
    }
}


// src/girl.js

import { Boy } from './src/boy';

export class Girl extends Boy {
    constructor(name, age) {
        super(name);
        this.age = age;
    }
    eat() {
        console.log("eating");
    }
}
```

> *"We use `import` keyword in line 1 of girl.js as it is an extended version of the `Boy` class."*

Now half of the work is done. For now, these classes are not accessible in our main **`app.js`** file. For that we've to import them in our **`app.js`** file. We can do that by using -

```
import { Boy } from './src/boy';

import { Girl } from './src/girl';
```

Default and Named Exports

Named Exports

We can export more than one objects from a specific module. This is called named export. Eg:

```
    constructor(model) {
        this.model = model;
    }
}

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

Here we exported a class `Car` and a function `add`.

Default Exports

It is basically the main object that is exported from the module. It is generally used in case we've only a single object to export. Let's see how it is -

```
export default class Car {
    constructor(model) {
        this.model = model;
    }
}
```

Instead, we use **`import Car from "./car";`** in case of default exports.

```
Default exports -> import Car from "./car";
Named exports -> import { Car } from "./car";
```

👋 Woosh! You've made it to the end. Hope I've helped you somehow. I write articles like this whenever I've some spare time. Besides this, I share content related to web development daily on Twitter. Let's connect there! @eyeshreya

---

Comments (2)