ECMAScript(ES)

Krishna Prasad Timilsina

bikranshu.t@gmail.com

ECMAScript is the standard upon which JavaScript is based, and it's often abbreviated to **ES**.

ES6

1. Block-Scoped Constructs Let and Const

1.1 let

The let statement declares a block scope local variable, optionally initializing it to a value.

Syntax:

let variableName;

Eg.

```
let x = 1;
```

1.2 const

Constants are block-scoped, much like variables defined using the ***let*** statement. The value of a constant cannot change through reassignment, and it can't be redeclared.

Syntax:

const variableName;

Eg.

```
const CONST_IDENTIFIER = 0; // constants are uppercase by convention
```

2. Default Parameters

In ES5,

```
var link = function (height, color, url) {
        var height = height || 50
        var color = color || 'red'
        var url = url || 'http://azat.co'
}
```

In ES6,

```
var link = function(height = 50, color = 'red', url = 'http://azat.co') {
        ….
}
```

3. String interpolation

In ES5,

```
function printCoord(x, y) {
        console.log('('+x+', '+y+')');
}
```

In ES6,

```
function printCoord(x, y) {
        console.log(`(${x}, ${y})`);
}
```

4. Template Literals

Template String or Template literals are enclosed with **` *(back-tick)*** . You can embed variables in template string just by surrounding variable by ***${}***

In ES5,

```
var name = 'Your name is ' + first + ' ' + last + '.'
var url = 'http://localhost:3000/api/messages/' + id
```

In ES6,

```
var name = `Your name is ${first} ${last}.`
var url = `http://localhost:3000/api/messages/${id}`
```

5. Multi-line Strings

In ES5,

```
var roadPoem = 'Then took the other, as just as fair,\n\t'
+ 'And having perhaps the better claim\n\t'
+ 'Because it was grassy and I wanted wear,\n\t'
+ 'Though as for that the passing there\n\t'
+ 'Had worn them really about the same,\n\t'

var agreements = 'You have the right to be you.\n\
You can only be you when you do your best.'
```

In ES6,

```
var roadPoem = `Then took the other, as just as fair,
And having perhaps the better claim
Because it was grassy and wanted wear,
Though as for that the passing there
Had worn them really about the same,`

var agreements = `You have the right to be you.
You can only be you when you do your best.
```

6. Handling multiple return values

6.1 Multiple return values via arrays

exec() returns captured groups via an Array-like object.

In ES5,

```
var matchObj = /^(\d\d\d\d)-(\d\d)-(\d\d)$/.exec('2999-12-31');
var year = matchObj[1];
var month = matchObj[2];
var day = matchObj[3];
```

In ES6,

```
const [, year, month, day] =/^(\d\d\d\d)-(\d\d)-(\d\d)$/.exec('2999-12-31');
```

The empty slot at the beginning of the Array pattern skips the Array element at index zero.

6.2 Multiple return values via objects

The method Object.getOwnPropertyDescriptor() returns a property descriptor, an object that holds multiple values in its properties.

In ES5

```
var obj = { foo: 123 };
var propDesc = Object.getOwnPropertyDescriptor(obj, 'foo');
var writable = propDesc.writable;
var configurable = propDesc.configurable;
console.log(writable, configurable); // true true
```

In ES6,

```
const obj = { foo: 123 };
const {writable, configurable} = Object.getOwnPropertyDescriptor(obj, 'foo');
console.log(writable, configurable); // true true
```

7. Destructuring Assignment

In ES5,

```
var data = $('body').data(),  // data has properties house and mouse
house = data.house,
mouse = data.mouse
```

In ES6,

```
var { house, mouse} = $('body').data()  // we'll get house and mouse variables
```

8. Parameter Optional

In ES5,

```
function selectEntries(options) {
    options = options || {}; // (A)
    var start = options.start || 0;
    var end = options.end || -1;
    var step = options.step || 1;
}
```

In ES6 you can specify {} as a parameter default value:

```
function selectEntries({ start=0, end=-1, step=1 } = {}) {
    ….
}
```

9. Parameter default values

In ES5,

```
function foo(x, y) {
    x = x || 0;
    y = y || 0;
    …
}
```

In ES6,

```
function foo(x=0, y=0) {
    …
}
```

10. Named parameters

In ES5,

```
function selectEntries(options) {
    var start = options.start || 0;
    var end = options.end || -1;
    var step = options.step || 1;
    …
}
```

In ES6,

```
function selectEntries({ start=0, end=-1, step=1 }) {
    …
}
```

11. Arrow Functions

A shorthand notation for ***function()***, but it does not bind ***this*** in the same way.

Specifying parameters:

```
        () => { ... }  // no parameter
        x => { ... }  // one parameter, an identifier
   (x, y) => { ... }  // several parameters
```

In ES5, such callbacks are relatively verbose:

```
var arr = [1, 2, 3];
var squares = arr.map(function (x) { return x * x });
```

In ES6, arrow functions are much more concise:

```
const arr = [1, 2, 3];
const squares = arr.map(x => x * x);
```

Implicit return

Arrow functions allow you to have an implicit return: values are returned without having to use the return keyword.

It works when there is a one-line statement in the function body.
Eg.

```
const myFunction = () => 'test'
myFunction()  // 'test'
```

When returning an object, remember to wrap the curly brackets in parentheses to avoid it being considered the wrapping function body brackets:
Eg.

```
const myFunction = () => ({ value: 'test' })
myFunction()  // {value: 'test'}
```

12. Classes

In ES5, you implement constructor functions directly:

```
function Person(name) {
   this.name = name;
 }
 Person.prototype.describe = function () {
         return 'Person called '+this.name;
 };
```

In ES6, classes provide slightly more convenient syntax for constructor functions:

```
class Person {
   constructor(name) {
     this.name = name;
   }
   describe() {
     return 'Person called '+this.name;
   }
}
```

When the object is initialized, the ***constructor*** method is called, with any parameters passed.

```
const flavio = new Person('Flavio')
flavio.hello()
```

12.1 Class inheritance

A class can extend another class, and objects initialized using that class inherit all the methods of both classes.

If the inherited class has a method with the same name as one of the classes higher in the hierarchy, the closest method takes precedence:

Eg.

```
class Programmer extends Person {
  hello() {
        return super.hello() + ' I am a programmer.'
  }
}
const flavio = new Programmer('Flavio')
flavio.hello()  // "Hello, I am Flavio. I am a programmer."
```

Classes do not have explicit class variable declarations, but you must initialize any variables in the constructor.

Inside a class, you can reference the parent class calling ***super()***.

12.2 Static methods

Normally methods are defined on the instance, not on the class.
Static methods are executed in the class instead:

Eg.

```
class Person {
  static genericHello() {
        return 'Hello'
  }
}
```

```
    Person.genericHello()  // Hello
```

13. Rest and spread operators

13.1 Spread operator (…)

It enables extraction of array or object content as single elements.
It provides you with the ability to expand iterable objects into multiple elements.
The spread operator allows you to expand an array into its individual elements.
Eg.

```
    const movies = ["Leon", "Love Actually", "Lord of the Rings"];
    console.log(...movies); // Leon Love Actually Lord of the Rings
```

Before the spread operator
Eg.

```
    const shapes = ["triangle", "square", "circle"];
    const objects = ["pencil", "notebook", "eraser"];
    const chaos = shapes.concat(objects);
    console.log(chaos); // ["triangle", "square", "circle", "pencil", "notebook", "eraser"]
```

After the spread operator

```
    const chaos = [...shapes, ...objects];
    console.log(chaos); // ["triangle", "square", "circle", "pencil", "notebook", "eraser"]
```

Without the spread operator
Eg.

```
    const chaos = [shapes, objects];
    console.log(chaos); // [Array(3), Array(3)]
```

13.2 Rest operator (…)

The rest parameter lets you bundle elements back into an array.
Eg.

```
    const movie = ["Life of Brian", 8.1, 1979, "Graham Chapman", "John Cleese", "Michael
    Palin"];
    const [title, rating, year, ...actors] = movie;
    console.log(title, rating, year, actors); // "Life of Brian", 8.1, 1979, ["Graham Chapman",
    "John Cleese", "Michael Palin"]
```

14. Destructuring

14.1 Of array

Enables extraction of requested elements from the array and assigning them to variables.
Eg.

```
    const firstAndSecondElement = ([first, second]) => {
        console.log('First element is ' + first + ', second is ' + second)
```

```
}

const secondAndFourthElement = ([, second, , fourth]) => {
        console.log('Second element is ' + second + ', fourth is ' + fourth)
}

let array = [1, 2, 3, 4, 5]

firstAndSecondElement(array) // First element is 1, the second is 2
secondAndFourthElement(array) // Second element is 2, fourth is 4
```

14.2 Of object

Enables extraction of requested properties from the object and assigning them to variables of the same name as properties.

Eg.

```
const basicInfo = ({firstName, lastName, profession}) =>ss {
        console.log(firstName + ' ' + lastName + ' - ' + profession)
}

let person = {
    firstName: 'John',
    lastName: 'Smith',
    age: 33,
    children: 3,
    profession: 'teacher'
}

basicInfo(person) // John Smith - teacher
```

15. Modules

ES6 module.js

```
export var port = 3000;
export function getAccounts(url) {
     ...
}
```

ES6 file main.js

```
import {port, getAccounts} from 'module';
console.log(port) // 3000
```

Or we can import everything as a variable service in main.js

```
import * as service from 'module';
console.log(service.port) // 3000
```

16. IIFEs to blocks(Avoid IIFEs)

In ES5, you had to use a pattern called IIFE (Immediately-Invoked Function Expression) if you wanted to restrict the scope of a variable tmp to a block:

```
(function () {  // open IIFE
        var tmp = ···;
        …
}());  // close IIFE

console.log(tmp); // ReferenceError
```

In ES6, you can simply use a block and a let declaration (or a const declaration):

```
{  // open block
        let tmp = ···;
        …
}  // close block

console.log(tmp); // ReferenceError
```

17. From function expressions in object literals to method definitions

In ES5 object literals, methods are created like other properties. The property values are provided via function expressions.

```
var obj = {
        foo: function () {
                …
        },
        bar: function () {
                this.foo();
        }, // trailing comma is legal in ES5
}
```

In ES6,

```
const obj = {
        foo() {
                …
        },
        bar() {
                this.foo();
        },
}
```

18. Array helper functions

18.1 forEach

Executes the provided function for each element of the array, passing the array element as an argument.

Eg.

```
let arr = [1, 2, 3, 4, 5];
arr.forEach((element) => {
        console.log(element);
});
```

18.2 map

Creates a new array containing the same number of elements, but output elements are created by the provided function. It just converts each array element to something else.

Eg.

```
let array1 = [1, 4, 9, 16];
const map1 = array1.map(x => x * 2);
console.log(map1); // Array [2, 8, 18, 32]
```

18.3 filter

Creates a new array containing a subset of the original array. The result has these elements that pass the test implemented by the provided function, which should return true or false.

Eg.

```
let words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];
const result = words.filter(word => word.length > 6);
console.log(result); // Array ["exuberant", "destruction", "present"]
```

18.4 find

The ***find()*** method returns the value of the first element in the array that satisfies the provided testing function. Otherwise ***undefined*** is returned.

Eg.

```
let items = [5, 12, 8, 130, 44];
const found = items.find((item) => item > 10);
console.log(found); // 12
```

18.5 every

Checks if every element of the array passes the test implemented by the provided function, which should return ***true*** or ***false***.

Eg.

```
let items = [1, 30, 39, 29, 10, 13];
const output = items.every((item) => item < 40);
console.log(output); // true
```

18.6 some
Checks if any element of the array passes the test implemented by the provided function, which should return ***true*** or ***false***.
Eg.

```
let items = [1, 2, 3, 4, 5];
const output = items.some((item) => item  % 2 === 0); // checks whether an element is even
console.log(output); // true
```

18.7 reduce
Applies a function passed as the first parameter against an accumulator and each element in the array (from left to right), thus reducing it to a single value. The initial value of the accumulator should be provided as the second parameter of the reduce function.
Eg.

```
const array1 = [1, 2, 3, 4];
const sum = (accumulator, currentValue) => accumulator + currentValue;
const mul = (accumulator, currentValue) => accumulator * currentValue;

// 1 + 2 + 3 + 4
console.log(array1.reduce(sum)); // 10

// 5 + 1 + 2 + 3 + 4
console.log(array1.reduce(sum, 5)); // 15

// 1 * 2 * 3 * 4
console.log(array1.reduce(mul)); // 24
```

16. Key/property shorthand

In ES5,

```
var obj = {
    firstName: firstName,
    lastName: lastName
}
```

In ES6,

```
let obj = {
    firstName,
    lastName
}
```

17. Single export

In ES6, the same thing is done via a so-called default export (declared via export default):

```
 //------ myFunc.js ------
 export default function () { ··· } // no semicolon!

  //------ main1.js ------
  import myFunc from 'myFunc';
   myFunc();
```

18. for-of *loop*

New type of iterator, an alternative to ***for..in***. It returns the values instead of the ***keys***.
Eg.

```
        const arr = ['a', 'b', 'c'];
        for (const elem of arr) {
             console.log(elem);
         }
```

Note:

- for...of ***iterates over the property values***
- for...in ***iterates the property names***

19. New Object methods

19.1 Object.is()

This method aims to help comparing values.
Eg.

```
        Object.is(a, b)
```

The result is always ***false*** unless:

- ***a*** and ***b*** are the same exact object
- ***a*** and ***b*** are equal strings (strings are equal when composed by the same characters)
- ***a*** and ***b*** are equal numbers (numbers are equal when their value is equal)
- ***a*** and ***b*** are both ***undefined***, both ***null***, both ***NaN***, both ***true*** or both ***false***

0 and ***-0*** are different values in JavaScript, so pay attention in this special case (convert all to ***+0*** using the **+** unary operator before comparing).

19.2 Object.assign()

This method copies all the ***enumerable own properties*** of one or more objects into another.

Its primary use case is to create a shallow copy of an object.
Eg.

```
        const copied = Object.assign({}, original)
```

Being a shallow copy, values are cloned, and objects references are copied (not the objects themselves), so if you edit an object property in the original object, that’s modified also in the copied object, since the referenced inner object is the same:

Eg.
```
const original = {
  name: 'Fiesta',
  car: {
        color: 'blue'
  }
}
const copied = Object.assign({}, original)
original.name = 'Focus'
original.car.color = 'yellow'
copied.name  // Fiesta
copied.car.color  // yellow
```

19.3 Object.setPrototypeOf()
Set the prototype of an object. Accepts two arguments: the object and the prototype.
Syntax:
```
Object.setPrototypeOf(object, prototype)
```
Eg.
```
const animal = {
  isAnimal: true
}
const mammal = {
  isMammal: true
}
mammal.__proto__ = animal
mammal.isAnimal  // true
const dog = Object.create(animal)
dog.isAnimal  // true
console.log(dog.isMammal)  // undefined
Object.setPrototypeOf(dog, mammal)
dog.isAnimal  // true
dog.isMammal  // true
```

ES7

1. Array.prototype.includes()
With ES6 and lower, to check if an array contains an element you had to use ***indexOf***, checks the index in the array, and returns ***-1*** if the element is not there.
Since ***-1*** is evaluated as a true value, you could ***not*** do for example

```
if (![1,2].indexOf(3)) {
        console.log('Not found')
    }
```

In ES7,

```
    if (![1,2].includes(3)) {
        console.log('Not found')
    }
```

2. Exponentiation Operator

The exponentiation operator ** is the equivalent of ***Math.pow()***, but brought into the language instead of being a library function.

Syntax:

```
    Math.pow(base, exponent);
```

Eg.

```
    Math.pow(4, 2) == 4 ** 2  // base ** exponent
```

ES8

1. String padding

The purpose of ***padStart()*** and ***padEnd()*** is to pad the start or the end of the string, so that the resulting string reaches the given length. You can pad it with a specific character or string or just pad with spaces by default.

Syntax:

```
    str.padStart(targetLength [, padString])
    str.padEnd(targetLength [, padString])
```

Note: The default value padString is ***space***.

Eg.

```
    let str = ‘es8’;
    str.padStart(2);          // 'es'
    str.padStart(5);          // '  es8'
    str.padStart(6, 'woof');  // 'wooes8'
    str.padStart(14, 'wow');  // 'wowwowwowwoes8'
    str.padStart(7, '0');     // '0000es8'
    str.padEnd(2);            // 'es8'
    str.padEnd(5);            // 'es8  '
    str.padEnd(6, 'woof’);  // 'es8woo'
    str.padEnd(14, 'wow’);  // 'es8wowwowwowwo'
    str.padEnd(7, '6’);       // 'es86666
```

2. Object.values()

This method returns an array containing all the object's own property values.
Eg.

```
const person = { name: 'Fred', age: 87 }
Object.values(person)   // ['Fred', 87]
```

Object.values() also works with arrays
Eg.

```
const people = ['Fred', 'Tony']
Object.values(people)  // ['Fred', 'Tony']
```

3. Object.entries()
This method returns an array containing all the object own properties, as an array of ***[key, Value]*** pairs.
Eg.

```
const person = { name: 'Fred', age: 87 }
Object.entries(person)  // [['name', 'Fred'], ['age', 87]]
```

Object.entries() also works with arrays.
Eg.

```
const people = ['Fred', 'Tony']
Object.entries(people)  // [['0', 'Fred'], ['1', 'Tony']]
```

4. Trailing commas
Before ES8, trailing commas could be used only in ***array*** and ***objects***. This feature allows to have trailing commas in ***function declarations***, and in ***functions calls***.
Eg.

```
const func = (var1, var2,) => {
    //...
}
func('test2', 'test2',)
```

5. Async functions (async/await)
An asynchronous function is a function which operates asynchronously via the ***event loop***, using an implicit ***Promise*** to return its result. But the syntax and structure of your code using ***async*** functions is much more like using standard synchronous functions.

Its non blocking (just like promises and callbacks).
Async functions always returns a Promise.

- If the function throws an error, promise will be ***rejected***.
- If the function returns a value, promise will be ***resolved***.

Fetch JSON resource, and parse it, using promises:

```
const getUserData = () => {
        return fetch('/users.json')  // get users list
        .then(response => response.json())  // parse JSON
        .then(users => users[0])  // pick first user
        .then(user => fetch(`/users/${user.name}`))  // get user data
        .then(userResponse => response.json())  // parse JSON
}
getUserData()
```

And same functionality provided using await/async:

```
const getUserData = async () => {
        const response = await fetch('/users.json')  // get users list
        const users = await response.json()  // parse JSON
        const user = users[0]  // pick first user
        const userResponse = await fetch(`/users/${user.name}`)  // get user data
        const userData = await user.json()  // parse JSON
        return userData
}
getUserData()
```

Pay attention that an ***async function*** always returns a ***promise*** and an ***await*** keyword may only be used in functions marked with the ***async*** keyword.

Async/await makes easier for debugging because to the compiler it's just like synchronous code

6. Object.getOwnPropertyDescriptors

The ***getOwnPropertyDescriptors*** method returns all of the own properties descriptors of the specified object. An own property descriptor is one that is defined directly on the object and is not inherited from the object's prototype.

Eg.

```
        const obj = {
          get es7() { return 777; },
          get es8() { return 888; }
        };
        Object.getOwnPropertyDescriptors(obj);

        // {
        //   es7: {
        //         configurable: true,
        //         enumerable: true,
        //         get: function es7(){}, //the getter function
        //         set: undefined
        //   },
        //   es8: {
        //         configurable: true,
        //         enumerable: true,
        //         get: function es8(){}, //the getter function
        //         set: undefined
        //   }
        // }
```

The ***obj*** is the source object. The possible keys for the returned descriptor objects result are ***configurable***, ***enumerable***, ***writable***, ***get***, ***set*** and ***value***.

1. ***value*** —The value associated with the property (data descriptors only).
2. ***writable*** **—true** if and only if the value associated with the property may be changed (data descriptors only).
3. ***get*** **—A** function which serves as a getter for the property, or undefined if there is no getter (accessor descriptors only).
4. ***set*** —A function which serves as a setter for the property, or undefined if there is no setter (accessor descriptors only).
5. ***configurable*** —true if and only if the type of this property descriptor may be changed and if the property may be deleted from the corresponding object.
6. ***enumerable*** —true if and only if this property shows up during enumeration of the properties on the corresponding object.

Use Case:
JavaScript has a method to copy properties ***Object.assign()***. It copies the property whose key is key.
Eg.

```
const value = source[key];  // get
target[key] = value;  // set
```

And in some cases it fails because it doesn't properly copy the properties with non-default attributes such as ***getters***, ***setters*** and ***non-writable*** properties.

Eg.

```
const objTarget = {};
const objSource = {
        set greet(name) { console.log('hey, ' + name); }
};
Object.assign(objTarget, objSource);
objTarget.greet = 'love';   // trying to set fails, sets greet = 'love'
```

Solution:
Eg.
```
        const objTarget = {};
        const objSource = {
                set greet(name) {
                        console.log('hey, ' + name);
                         }
        };
        Object.defineProperties(objTarget, Object.getOwnPropertyDescriptors(objSource));
        objTarget.greet = 'love'; // prints 'hey, love'
```

7. Shared Memory and Atomics
WebWorkers are used to create multithreaded programs in the browser.

They offer a messaging protocol via events. Since ES8, you can create a shared memory array between web workers and their creator, using a ***SharedArrayBuffer***.

Since it's unknown how much time writing to a shared memory portion takes to propagate, Atomics are a way to enforce that when reading a value, any kind of writing operation is completed.

ES9

1. Rest/Spread Properties
The three-dot operator ***(…)*** was introduced with ES6 and has been used for array literals. ES9 introduces rest/spread feature for object destructuring as well as arrays.
Eg.
```
        const {a, ...n} = {a: 5, b: 6, c: 7, d: 8};
        console.log(a);  // 5
        console.log(n);  // {b: 6, c: 7, d: 8}
```

Spread properties allow to create a new object by combining the properties of the object passed after the spread operator:
Eg.
```
        const items = { a, b, ...n };
        console.log(items);  //{ a: 5, b: 6, c: 7, d: 8}
```

2. Asynchronous Iteration

The new construct ***for-await-of*** allows you to use an async iterable object as the loop iteration:
Eg.

```
for await (const line of readLines(filePath)) {
  console.log(line)
}
```

Since this uses ***await***, you can use it only inside ***async*** functions, like a normal ***await.***

3. Promise.prototype finally

- When a promise is fulfilled, successfully it calls the then() methods, one after another.
- If something fails during this, the then() methods are jumped and the catch() method is executed.
- finally() allow you to run some code regardless of the successful or not successful execution of the promise.

This new callback will always be executed, if catch was called or not.
Eg.

```
fetch('http://website.com/files')
.then(data => data.json())
.catch(err => console.error(err))
.finally(() => console.log('processed!'))
```

4. Regex changes

I. s (dotAll) flag for regular expressions

While using regular expressions, you expect that the dot . matches a single character, but it's not always true. One exception is with line terminator characters:
Eg.

```
/hello.bye/.test('hello\nbye') // false
```

The solution is the new flag ***/s*** (from singleline):
Eg.

```
/hello.bye/s.test('hello\nbye')  // true
```

II. RegExp named capture groups

This is the old way of getting the year, month, and day from a date:
Eg.

```
const REGEX = /([0-9]{4})-([0-9]{2})-([0-9]{2});
const results = REGEX.exec('2018-07-12');
console.log(results[1]); // 2018
console.log(results[2]); // 07
console.log(results[3]); // 12
```

Using capture groups,
Eg.

```
    const REGEX = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2});
    const results = REGEX.exec('2018-07-12');
    console.log(results.groups.year);  // 2018
    console.log(results.groups.month); // 07
    console.log(results.groups.day);   // 12
```

III. RegExp Unicode Property Escapes
Now we can search for characters by mentioning their Unicode character property inside of ***\p{}***
Eg.

```
    /\p{Script=Greek}/u.test('μ')  // true
```

ES10

1. Dynamic import
Imports can now be assigned to a variable:
Eg.

```
    element.addEventListener('click', async () => {
        const module = await import(`./api-scripts/button-click.js`);
        module.clickEvent();
    });
```

2. Array.flat()
Flattening of a multidimensional array:
Eg.

```
    let multi = [1,2,3,[4,5,6,[7,8,9,[10,11,12]]]];

    multi.flat();               // [1,2,3,4,5,6,Array(4)]
    multi.flat().flat();        // [1,2,3,4,5,6,7,8,9,Array(3)]
    multi.flat().flat().flat(); // [1,2,3,4,5,6,7,8,9,10,11,12]
    multi.flat(Infinity);       // [1,2,3,4,5,6,7,8,9,10,11,12]
```

BONUS TIP :
You can also remove any empty slots in the array using the flat() method.
Eg.

```
    var arr = [1, 2, , 4, 5];
    arr.flat();  // [1, 2, 4, 5]
```

3. Array.flatMap()

The ***flatMap()*** method first maps each element using a mapping function, then flattens the result into a new array. It is identical to a ***map*** followed by a flat of depth 1, but ***flatMap*** is often quite useful, as merging both into one method is slightly more efficient.
Eg.

```
let array = [1, 2, 3, 4, 5];
array.map(x => [x, x * 2]);
```

Becomes:

```
[Array(2), Array(2), Array(2)]
0: (2)[1, 2]
1: (2)[2, 4]
2: (2)[3, 6]
3: (2)[4, 8]
4: (2)[5, 10]
```

Flatten the map again:

```
array.flatMap(v => [v, v * 2]);    // [1, 2, 2, 4, 3, 6, 4, 8, 5, 10]
```

4. Object.fromEntries()

Transform a list of key & value pairs into an object.
Note: Object.fromEntries only accept iterable (i.e) Object.fromEntries(iterable). It will accept only ***Map*** or ***Array***.
Eg.

```
let obj = { apple : 10, orange : 20, banana : 30 };

let entries = Object.entries(obj);
entries;
(3) [Array(2), Array(2), Array(2)]
 0: (2) ["apple", 10]
 1: (2) ["orange", 20]
 2: (2) ["banana", 30]

let fromEntries = Object.fromEntries(entries);    // { apple: 10, orange: 20, banana: 30 }
```

Eg.

```
let entries = new Map([
        ['name', 'ben'],
        ['age', 25]
]);

Object.fromEntries(entries);  // {name: "ben", age: 25}
```

5. String.trimStart() & String.trimEnd()

The ***trimStart()*** method removes whitespace from the beginning of a string.

The ***trimEnd()*** method removes whitespace from the end of a string.

Note: You can think why another new method already there are two method trimRight()and trimLeft() but it will be ***alias*** of above new methods.

Eg.

```
let greeting = "           Space around ";
greeting.trimEnd();     // "        Space around";
greeting.trimStart();   // "Space around          ";
```

6. Optional Catch Binding

In the past catch clause in a try / catch statement required a variable. You are free to go ahead and make use of catch block without a ***param***.

Eg.

```
try {
   throw new Error("Some error");
} catch {
   console.log("no param for catch");
}
```

7. Function.toString()

Functions are objects. And every object has a ***.toString()*** method because it originally exists on ***Object.prototype.toString()***. All objects (including functions) are inherited from it via prototype-based class inheritance.

The ***toString()*** method returns a string representing the source code of the function.Earlier *white spaces*,***new lines*** and ***comments*** will be removed when you do now they are retained with original source code.

Eg.

```
const person =() => {
        // This is a comment....
}
console.log(person.toString());
// Output
// () => {
//      // This is a comment....
// }
```

8. Symbol.description

The read-only ***description*** property is a string returning the optional description of ***Symbol*** objects.

When you create a Symbol you can specify a description for the symbol for debugging purposes. In order to ***console.log()*** out the symbol description, you may have to convert it to string.
Eg.

```
const desc = 'my symbol';
const sym = Symbol(desc);
console.log(String(sym) == `Symbol(${desc})`);  // true
```

But now with the new proposal, you can simply access the description by using ***Symbol.description***.
Eg.

```
const desc = 'my symbol';
const sym = Symbol(desc);
console.log(sym.description == desc);  // true
```

9. Stable Array.prototype.sort()

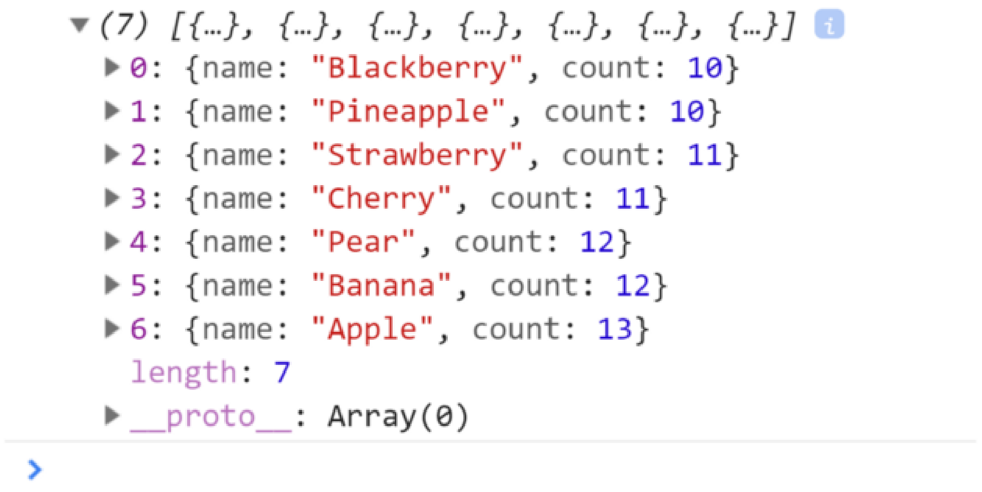
Previous implementation of **V8** used an unstable quick sort algorithm for arrays containing more than 10 items.

A stable sorting algorithm is when two objects with equal keys appear in the same order in the sorted output as they appear in the unsorted input.

Eg.

```
var fruit = [
    { name: "Apple",      count: 13, },
    { name: "Pear",       count: 12, },
    { name: "Banana",     count: 12, },
    { name: "Strawberry", count: 11, },
    { name: "Cherry",     count: 11, },
    { name: "Blackberry", count: 10, },
    { name: "Pineapple",  count: 10, }
];
// Create our own sort criteria function:
let my_sort = (a, b) => a.count - b.count;

// Perform stable ES10 sort:
let sorted = fruit.sort(my_sort);
console.log(sorted);
```



```
▼(7) [{…}, {…}, {…}, {…}, {…}, {…}, {…}]
  ▶0: {name: "Blackberry", count: 10}
  ▶1: {name: "Pineapple", count: 10}
  ▶2: {name: "Strawberry", count: 11}
  ▶3: {name: "Cherry", count: 11}
  ▶4: {name: "Pear", count: 12}
  ▶5: {name: "Banana", count: 12}
  ▶6: {name: "Apple", count: 13}
   length: 7
  ▶__proto__: Array(0)
>
```

10. Well Formed JSON.Stringify()
To prevent ***JSON.stringify*** from returning ill-formed Unicode strings.

The proposed solution is to represent unpaired surrogate code points as JSON escape sequences rather than returning them as single UTF-16 code units.
Eg.

```
JSON.stringify('\uDF06\uD834');   // → '"\\udf06\\ud834"'
JSON.stringify('\uDEAD');          // → '"\\udead"'
```

11. Standardized globalThis object
The ***globalThis*** object which should be used from now on to access global scope on any platform.
Eg.

```
// Access global array constructor
globalThis.Array(0, 1, 2);  //[0, 1, 2]

// Similar to window.v = { flag: true } in <= ES5
globalThis.v = { flag: true };

console.log(globalThis.v);  // {flag: true }
```

Resource Link:
https://github.com/airbnb/javascript

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array
https://codetower.github.io/es6-features/