# ECMAscript 8/9

## Background

## WTF is ECMAscript?

1. JavaScript was originally named JavaScript in hopes of capitalizing on the success of Java.
2. Netscape then submitted JavaScript to ECMA International for Standardization. (ECMA is an organization that standardizes information)
3. This results in a new language standard, known as ECMAScript.

## WTF is ECMAscript?

Key Takeaways:

1. An update to ECMAscript can be expected annually.
2. Initial Editions of ECMAScript are named numerically, increasing by 1: ES1, ES2, ES3, ES4, ES5
3. New editions (starting with 2015) will be named ES followed by the year of release: ES2015, ES2016, ES2017
4. ECMAScript is a standard. JavaScript is the most popular implementation of that standard. Other implementations include: SpiderMonkey, V8, and ActionScript.

## ES8 Features

## String Padding

Adds two functions to the String object: padStart & padEnd. As their names, the purpose of those functions is to pad the start or the end of the string, so that the resulting string reaches the given length.

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

```
'es8'.padStart(2);          // 'es8'
'es8'.padStart(5);          // '  es8'
'es8'.padStart(6, 'woof');  // 'wooes8'
'es8'.padStart(14, 'wow');  // 'wowwowwowwoes8'
'es8'.padStart(7, '0');     // '0000es8'
```

## Object.values

The Object.values method returns an array of a given object’s own enumerable property values, in the same order as that provided by a for in loop.

```
Object.values(obj)
```

## Object.values (cont’d)

```
const obj = { x: 'xxx', y: 1 };
Object.values(obj); // ['xxx', 1]

const obj = ['e', 's', '8']; // same as { 0: 'e', 1: 's', 2: '8' };
Object.values(obj); // ['e', 's', '8']

// when we use numeric keys, the values returned in a numerical
// order according to the keys
const obj = { 10: 'xxx', 1: 'yyy', 3: 'zzz' };
Object.values(obj); // ['yyy', 'zzz', 'xxx']
Object.values('es8'); // ['e', 's', '8']
```

## Object.entries

The Object.entries method returns an array of a given object's own enumerable property [key, value] pairs, in the same order as Object.values. The declaration of the function is trivial

## Object.entries (cont’d)

```
const obj = { x: 'xxx’, y: 1 };
Object.entries(obj); // [[’x’, 'xxx’], [’y’, 1]]

const obj = [’e’, 's’, '8’];
Object.entries(obj); // [[’0’, 'e’], [’1’, 's’], [’2’, '8’]]

const obj = { 10: 'xxx’, 1: 'yyy’, 3: 'zzz' };
Object.entries(obj); // [[’1’, 'yyy’], [’3’, 'zzz’], [’10’, 'xxx’]]
Object.entries('es8'); // [['0', 'e'], ['1', 's'], ['2', '8']]
```

## 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. The declaration of the function is:

```
Object.getOwnPropertyDescriptors(obj)
```

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

## 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. The declaration of the function is:

```
Object.getOwnPropertyDescriptors(obj)
```

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

```
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
//   }
// }
```

## Trailing Commas

Trailing commas in function parameter is the ability of the compiler not to raise an error (SyntaxError) when we add an unnecessary comma in the end of the list:

```
function es8(var1, var2, var3,) {
  // ...
}
```

As the function declaration, this can be applied on function’s calls:

```
es8(10, 20, 30,);
```

This feature inspired from the trailing of commas in object literals and Array literals [10, 20, 30,] and { x: 1, }.

## Async FUnctions

## Async Function

The async function declaration defines an asynchronous function, which returns an AsyncFunction object. Internally, async functions work much like generators, but they are not translated to generator functions.

```
function fetchTextByPromise() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve("es8");
    }, 2000);
  });
}
async function sayHello() {
  const externalFetchedText = await fetchTextByPromise();
  console.log(`Hello, ${externalFetchedText}`); // Hello, es8
}
sayHello();
```

## Async Functions (cont’d)

The call of sayHello will log Hello, es8 after 2 seconds.

```
console.log(1);
sayHello();
console.log(2);
```

And now the prints are:

```
1 // immediately
2 // immediately
Hello, es8 // after about 2 seconds
```

This is because the function call doesn’t block the flow.
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.

## ES9

## ES9 Key Features

Lifting template literal restriction

Asynchronous iterators

Promise.prototype.finally library

Unicode property escapes in regular expressions

Object Rest/spread properties

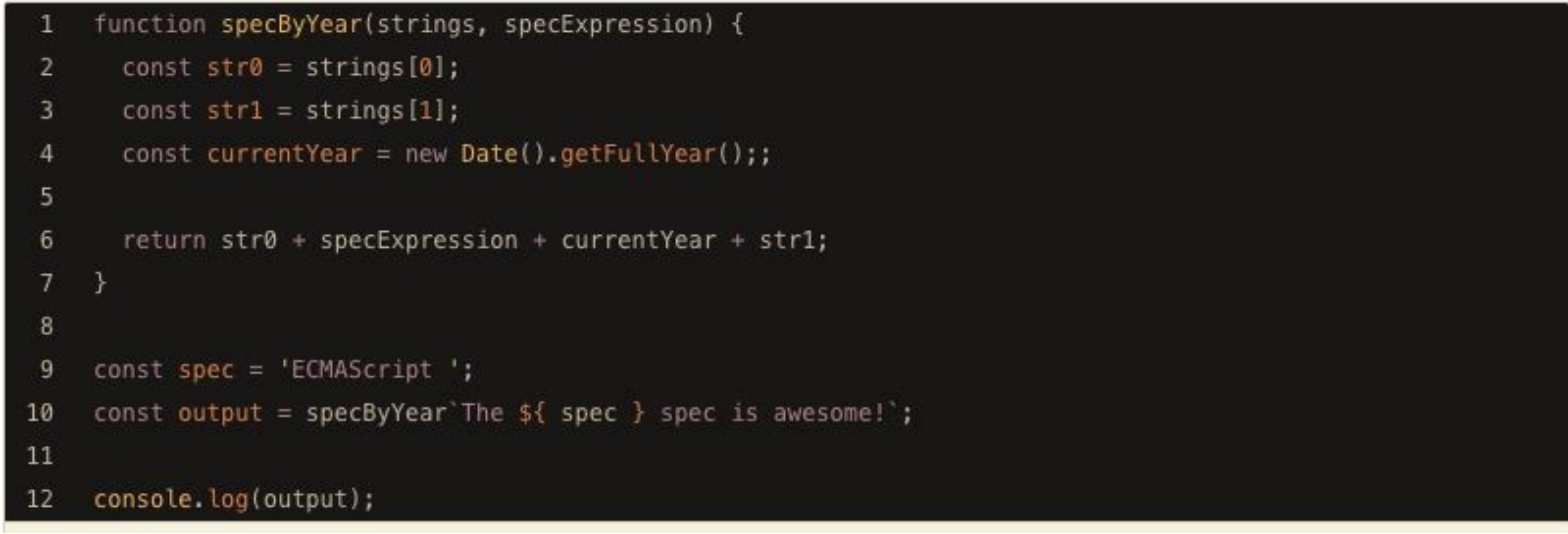
s ‘dotAll’ flag for regular expressions

RegExp lookbehind assertions

## Template Literals Revision

Template Literals were defined officially in ES2015 (ES6) specification - so it allows using multi-line strings and performing an expression interpolations easily.

On top of that, template literals could be attached with “tags” that are just an ordinary functions for manipulating the string.



```javascript
function specByYear(strings, specExpression) {
  const str0 = strings[0];
  const str1 = strings[1];
  const currentYear = new Date().getFullYear();;

  return str0 + specExpression + currentYear + str1;
}

const spec = 'ECMAScript ';
const output = specByYear`The ${ spec } spec is awesome!`;

console.log(output);
```

## So Wut?

## What it means

Well, when a string has substrings which include escape sequences inadvertently, for instance, a string that starts with \x, \u, \u{} (and so on) - it will be interpreted as escape sequences due to the restriction on escape sequences. In other words, each string which starts with escape sequences is illegal and an error will be emitted.

The proposed solution, which has been finally approved, is removing the restriction on escape sequences. So, it’s possible to use a string with illegal escape sequences as we wish.

## Rest/Spread Properties

ES6 introduced the concept of a rest element when working with array destructuring:

```
const numbers = [1, 2, 3, 4, 5]
[first, second, ...others] = numbers
```

And spread:

```
const numbers = [1, 2, 3, 4, 5]
const sum = (a, b, c, d, e) => a + b + c + d + e
const sumOfNumbers = sum(...numbers)
```

## Rest/Spread (cont’d)

ES2018 introduces the same but for objects.
Rest:

```
const { first, second, ...others } = { first: 1, second: 2, third: 3, fourth: 4,
fifth: 5 }
first // 1
second // 2
others // { third: 3, fourth: 4, fifth: 5 }
```

Spread properties allow to create a new object by combining the properties of the object passed after the spread operator:

```
const items = { first, second, ...others }
items //{ first: 1, second: 2, third: 3, fourth: 4, fifth: 5 }
```

## Asynchronous iteration

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

```
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 (see async/await)

The for await...of statement creates a loop iterating over async iterable objects as well as on sync iterables, including: built-in String, Array, Array-like objects (e.g., arguments or NodeList), TypedArray, Map, Set, and user-defined async/sync iterables. It invokes a custom iteration hook with statements to be executed for the value of each distinct property of the object.

## 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:

```
fetch('file.json')
  .then(data => data.json())
  .catch(error => console.error(error))
  .finally(() => console.log('finished'))
```

## ES9 RegEx Improvements

## A New “s” Flag for Regex

In general, regular expressions provide the ability to match any single character using the dot - ..

However, in ECMAScript specification there are two exceptions:

The dot doesn’t match line terminator characters, such as \n, \r and more.
The dot doesn’t match astral characters, which means, non-BMP characters. An excellent example of non-BMP character is the Emoji.
As it seems, the dot is supposed to match any character - but in fact, it omits some.

## The s flag for regular expressions

The s flag, short for single line, causes the . to match new line characters as well. Without it, the dot matches regular characters but not the new line:

```
/hi.welcome/.test('hi\nwelcome') // false

/hi.welcome/s.test('hi\nwelcome') // true
```

## RegExp lookbehind assertions: match a string depending on what precedes it

This is a lookahead (already in JS) : you use ?= to match a string that’s followed by a specific substring:

```
/Roger(?=Waters)/

/Roger(?= Waters)/.test('Roger is my dog') //false
/Roger(?= Waters)/.test('Roger is my dog and Roger Waters is a famous
musician') //true
```

?! performs the inverse operation, matching if a string is not followed by a specific substring:

```
/Roger(?!Waters)/

/Roger(?! Waters)/.test('Roger is my dog') //true
/Roger(?! Waters)/.test('Roger Waters is a famous musician') //false
```

## RegExp lookbehind assertions: match a string depending on what precedes it

Lookbehinds, a new feature, uses ?<=.

```
/(?<=Roger) Waters/

/(?<=Roger) Waters/.test('Pink Waters is my dog') //false
/(?<=Roger) Waters/.test('Roger is my dog and Roger Waters is a famous musician')
//true
```

A lookbehind is negated using ?<!:

```
/(?<!Roger) Waters/

/(?<!Roger) Waters/.test('Pink Waters is my dog') //true
/(?<!Roger) Waters/.test('Roger is my dog and Roger Waters is a famous musician') //false
```

## Unicode property escapes \p{…} and \P{…}

In a regular expression pattern you can use \d to match any digit, \s to match any character that's not a white space, \w to match any alphanumeric character, and so on.

This new feature extends this concept to all Unicode characters introducing \p{} and is negation \P{}.

Any unicode character has a set of properties. For example Script determines the language family, ASCII is a boolean that's true for ASCII characters, and so on.

## Unicode property escapes \p{...} and \P{...}

You can put this property in the graph parentheses, and the regex will check for that to be true:

```
/^\p{ASCII}+$/u.test('abc')   //✔️

/^\p{ASCII}+$/u.test('ABC@')  //✔️

/^\p{ASCII}+$/u.test('ABC🙃') //❌
```

## ASCII_Hex_Digit

ASCII_Hex_Digit is another boolean property, that checks if the string only contains valid hexadecimal digits:

```
/^\p{ASCII_Hex_Digit}+$/u.test('0123456789ABCDEF') //✔️
/^\p{ASCII_Hex_Digit}+$/u.test('h')                //❌
```

## Lower/Uppercase & Emoji

There are many other boolean properties, which you just check by adding their name in the graph parentheses, including Uppercase, Lowercase, White_Space, Alphabetic, Emoji and more:

```
/^\p{Lowercase}$/u.test('h') //✅
/^\p{Uppercase}$/u.test('H') //✅

/^\p{Emoji}+$/u.test('H')   //❌
/^\p{Emoji}+$/u.test('🙃🙃') //✅
```

## "Romani ite domum"

In addition to those binary properties, you can check any of the unicode character properties to match a specific value. In this example, I check if the string is written in the greek or latin alphabet:

```
/^\p{Script=Greek}+$/u.test('ελληνικά') //✅

/^\p{Script=Latin}+$/u.test('hey') //✅
```

## Named capturing groups

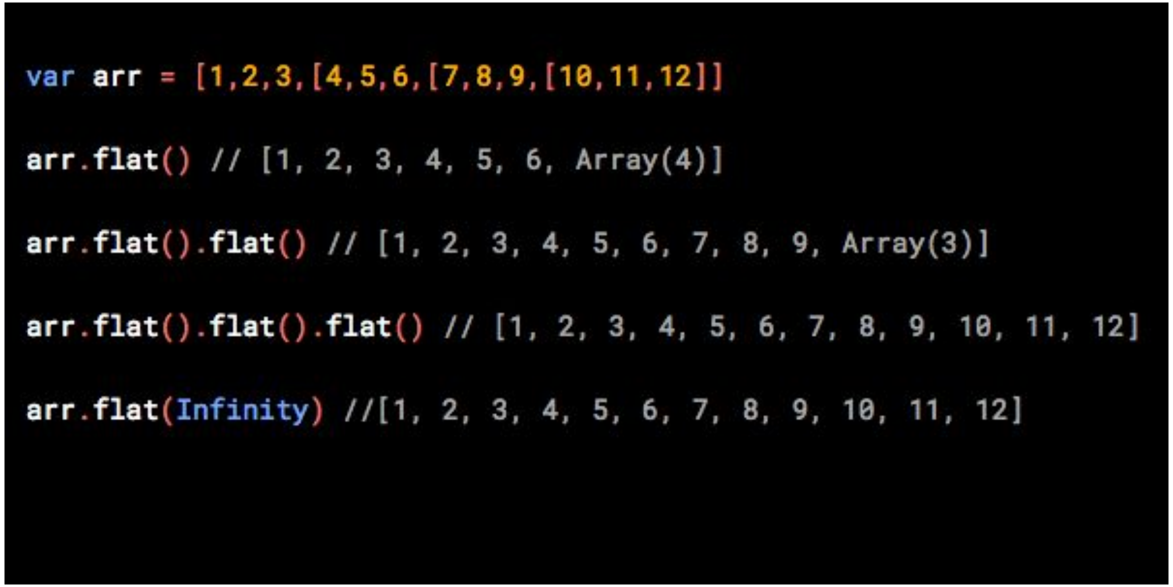
In ES2018 a capturing group can be assigned to a name, rather than just being assigned a slot in the result array:

```
const re = /(?<year>\d{4})-(?<month>\d{2})-(?<day>\d{2})/
const result = re.exec('2015-01-02')

// result.groups.year === '2015';
// result.groups.month === '01';
// result.groups.day === '02';
```

## ES10 - Whats Coming

## Array.Flat()

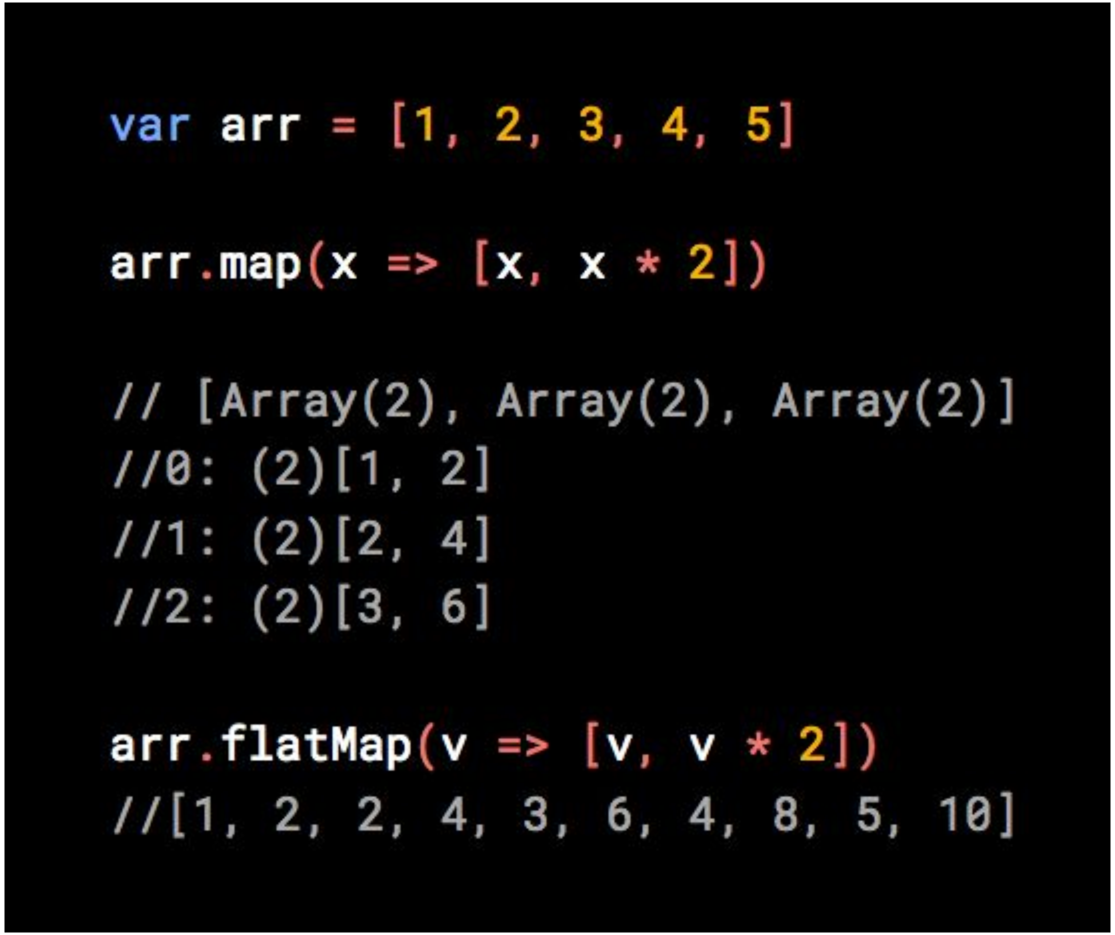
The flat() method creates a new array with all sub-array elements concatenated into it recursively up to the specified depth.



```
var arr = [1,2,3,[4,5,6,[7,8,9,[10,11,12]]]

arr.flat() // [1, 2, 3, 4, 5, 6, Array(4)]

arr.flat().flat() // [1, 2, 3, 4, 5, 6, 7, 8, 9, Array(3)]

arr.flat().flat().flat() // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

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

## 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.



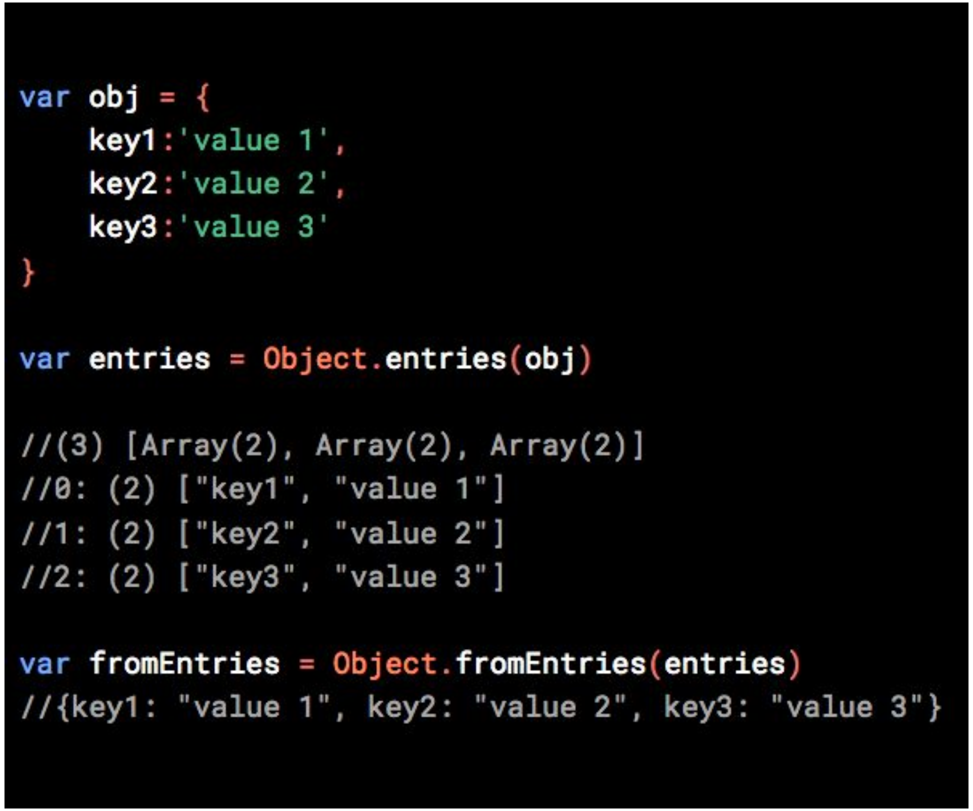
```
var arr = [1, 2, 3, 4, 5]

arr.map(x => [x, x * 2])

// [Array(2), Array(2), Array(2)]
//0: (2)[1, 2]
//1: (2)[2, 4]
//2: (2)[3, 6]

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

## Object.fromEntries()

The Object.fromEntries() method transforms a list of key-value pairs into an object.

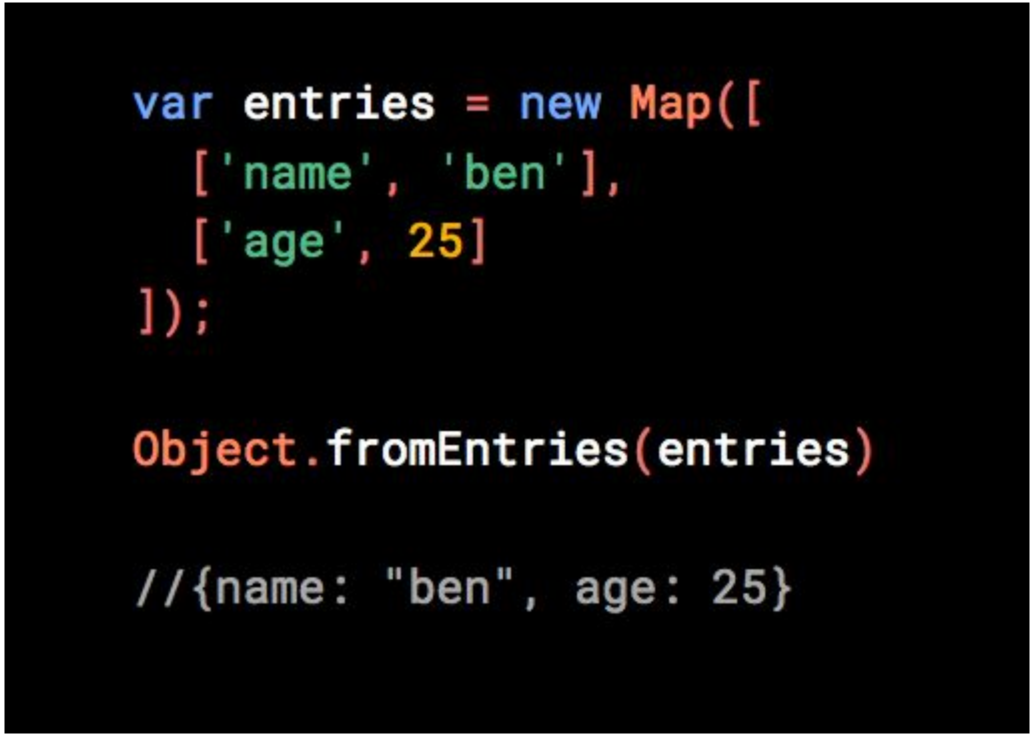
Note: Object.fromEntries only accept iterable (i.e) Object.fromEntries(iterable)

It will accept only Map or Array



```
var obj = {
    key1:'value 1',
    key2:'value 2',
    key3:'value 3'
}

var entries = Object.entries(obj)

//(3) [Array(2), Array(2), Array(2)]
//0: (2) ["key1", "value 1"]
//1: (2) ["key2", "value 2"]
//2: (2) ["key3", "value 3"]

var fromEntries = Object.fromEntries(entries)
//{key1: "value 1", key2: "value 2", key3: "value 3"}
```

Ex 1



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

Object.fromEntries(entries)

//{name: "ben", age: 25}
```

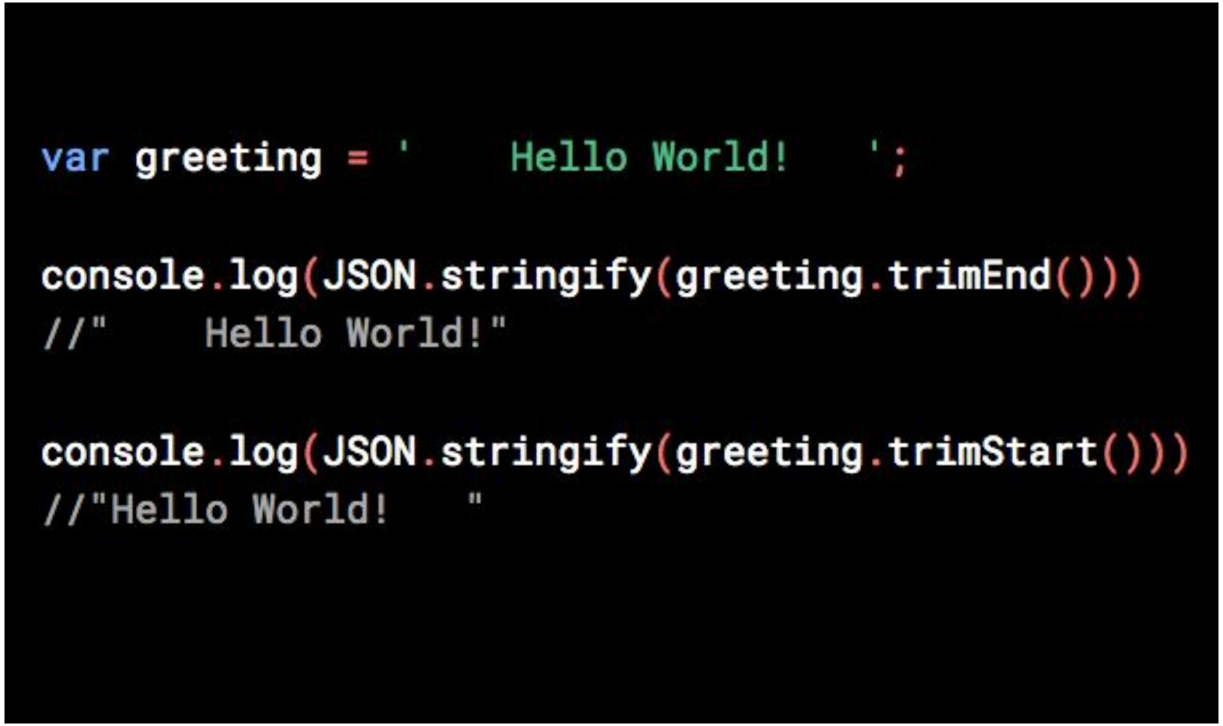
Ex2

## 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.

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

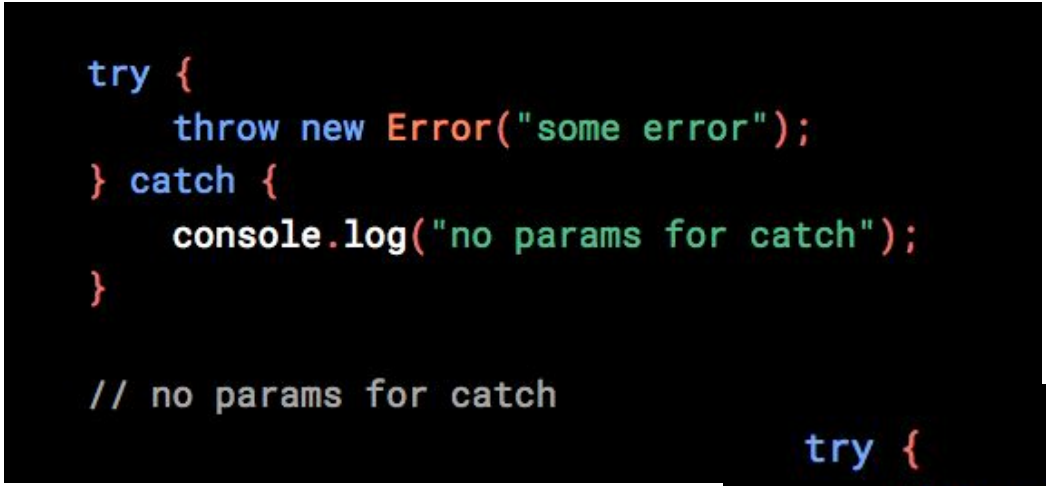


```
var greeting = '    Hello World!   ';

console.log(JSON.stringify(greeting.trimEnd()))
//"    Hello World!"

console.log(JSON.stringify(greeting.trimStart()))
//"Hello World!    "
```

Trim ex

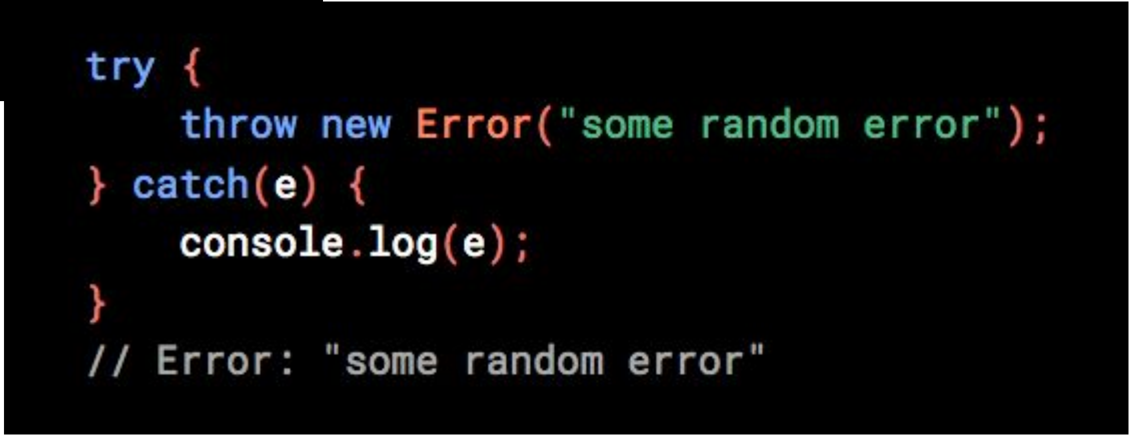
## Optional Catch Binding

Allow developers to use try/catch without creating an unused binding. You are free to go ahead make use of catch block without a param.



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

// no params for catch
```

<= NEW



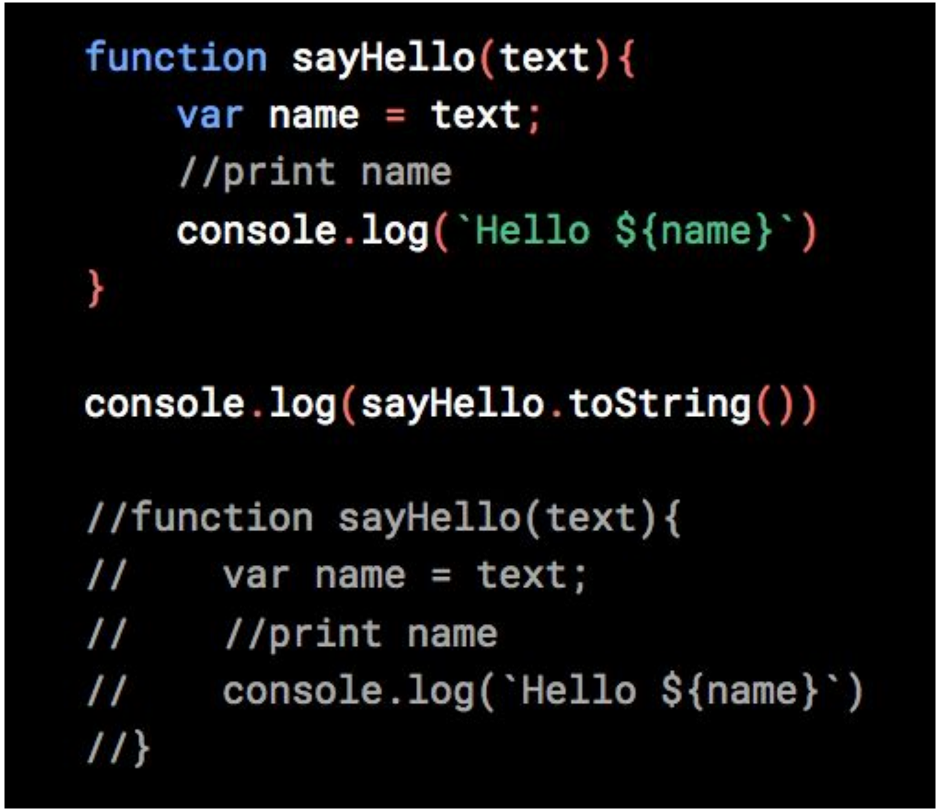
```
try {
    throw new Error("some random error");
} catch(e) {
    console.log(e);
}
// Error: "some random error"
```

OLD =>

Earlier it is mandatory to use param in catch block

## Function.toString()

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

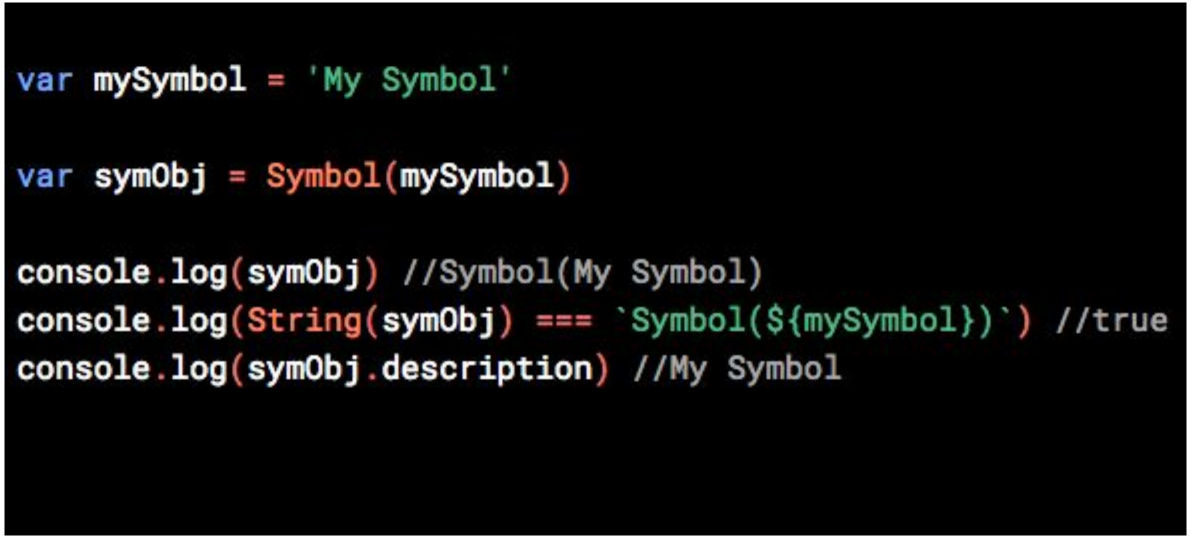


```
function sayHello(text){
    var name = text;
    //print name
    console.log(`Hello ${name}`)
}

console.log(sayHello.toString())

//function sayHello(text){
//    var name = text;
//    //print name
//    console.log(`Hello ${name}`)
//}
```

toString()

## Symbol.description

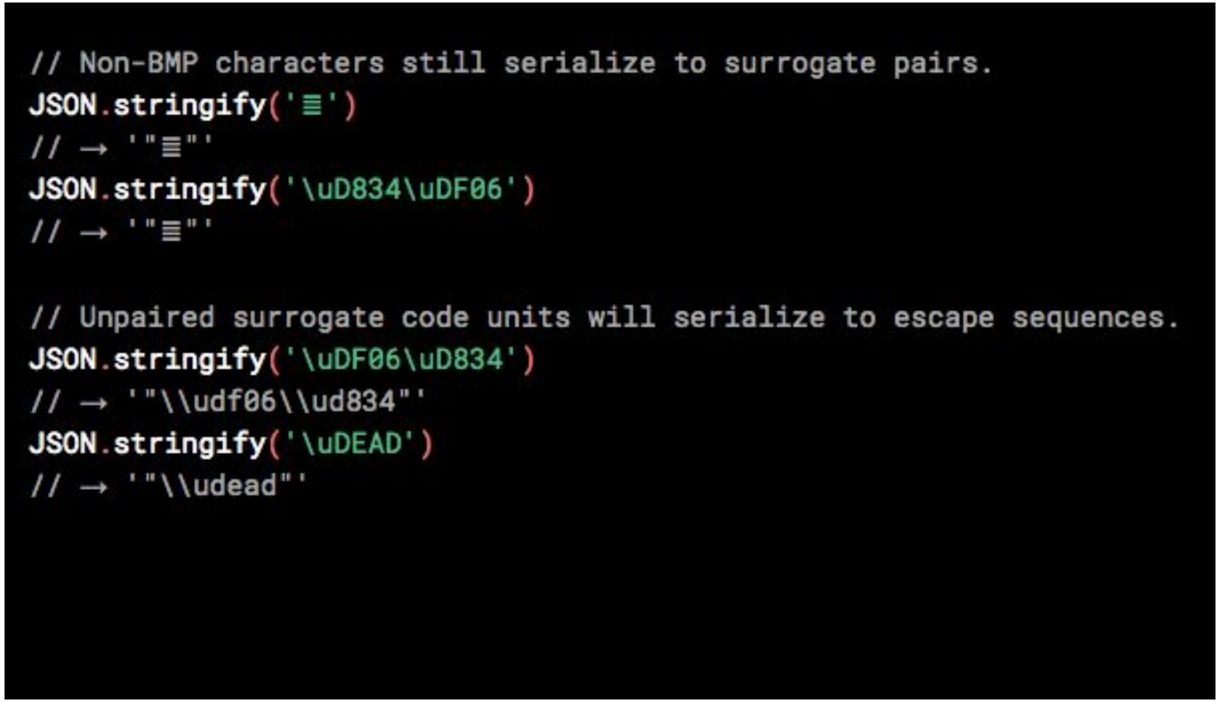
The read-only description property is a string returning the optional description of Symbol objects.



```
var mySymbol = 'My Symbol'

var symObj = Symbol(mySymbol)

console.log(symObj) //Symbol(My Symbol)
console.log(String(symObj) === `Symbol(${mySymbol})`) //true
console.log(symObj.description) //My Symbol
```

## Well Formed JSON.Stringify()

To prevent JSON.stringify from returning ill-formed Unicode strings.

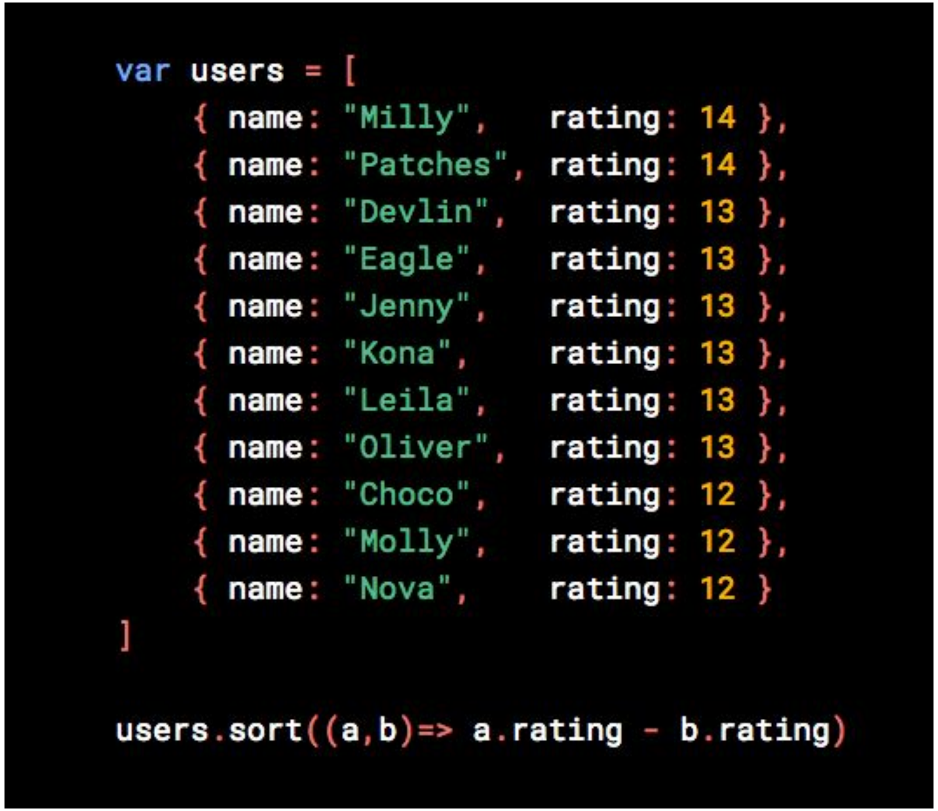


```
// Non-BMP characters still serialize to surrogate pairs.
JSON.stringify('𝌆')
// → '"𝌆"'
JSON.stringify('\uD834\uDF06')
// → '"𝌆"'

// Unpaired surrogate code units will serialize to escape sequences.
JSON.stringify('\uDF06\uD834')
// → '"\\udf06\\ud834"'
JSON.stringify('\uDEAD')
// → '"\\udead"'
```

## Array.Sort Stability

Previously, V8 used an unstable QuickSort for arrays with more than 10 elements. As of V8 v7.0 / Chrome 70, V8 uses the stable TimSort algorithm. 🎉

The only major engine JavaScript engine that still has an unstable Array#sort implementation is Chakra, which uses QuickSort for arrays with more than 512 elements (and stable insertion sort for anything else).



```
var users = [
    { name: "Milly",   rating: 14 },
    { name: "Patches", rating: 14 },
    { name: "Devlin",  rating: 13 },
    { name: "Eagle",   rating: 13 },
    { name: "Jenny",   rating: 13 },
    { name: "Kona",    rating: 13 },
    { name: "Leila",   rating: 13 },
    { name: "Oliver",  rating: 13 },
    { name: "Choco",   rating: 12 },
    { name: "Molly",   rating: 12 },
    { name: "Nova",    rating: 12 }
]

users.sort((a,b)=> a.rating - b.rating)
```

## Array.Sort Stability

## JSON ⊂ ECMAScript (JSON Superset)

Extend ECMA-262 syntax into a superset of JSON.

JSON syntax is defined by ECMA-404 and permanently fixed by RFC 7159, but the DoubleStringCharacter and SingleStringCharacter productions of ECMA-262 can be extended to allow unescaped U+2028 LINE SEPARATOR and U+2029 PARAGRAPH SEPARATOR characters.