ES6

POWERED BY:

nsoft sportradar

ES6 === ES2015

- EcmaScript is the "official" name for JavaScript
- ECMAScript is a subset of JavaScript (ActionScript, JScript, ...)
- short-hand for EcmaScript 6, which is a deprecated name for the EcmaScript 2015 language specification (ES2015 is the 6th version of EcmaScript)
- released in 2015 (previous major version release was in 2009)
- most browsers support ES5, so you need compiler for ES6 like babel
- new version of ECMAScript every year starting in 2015 (ES7, ES8, ES9)
- features
- caniuse.com

const and let

const

- once used, the variable can't be reassigned

let

- can be reassigned and take new value

let is the same as **const** in that both are blocked-scope

```
const a = 'hello'
a = 'world!' // this will throw an error
const a = { greet: 'hello' }
a.greet = 'goodbye'  // this is ok

let b = 'hello'
b = 'world!' // this will work
```

block scope

- block is defined by { }
- ES5 had function scope, not block scope
- **const**, **let** and **function** are block scoped

```
{
 const foo = 1;
 console.log(foo === 1);
 {
   const foo = 2;
   console.log(foo === 2);
 }
 console.log(foo === 1);
}
console.log(foo); // this will throw an error
```

arrow functions =>

- **function** shorthand using the **=>** syntax
- unlike **functions**, arrows share the same lexical **this** as their surrounding code

```
// expression bodies
const odds  = evens.map(n => n + 1)
const nums  = evens.map((n, i) => n + i)
const pairs = evens.map(n => ({ even: n, odd: n + 1 }))

// statement bodies
nums.forEach(n => {
if (n % 5 === 0)
  fives.push(n)
})
```

default parameter

- simple and intuitive default values for **function** parameters

```
function f(x, y = 7, z = 42) {
 return x + y + z;
}

f(1) === 50;
```

rest parameter

- aggregation of remaining arguments into single parameter of variadic **functions**

```
function f(x, y, ...z) {
 // z is an Array
 return (x + y) * z.length;
}

f(1, 2, 'hello', true, 7) === 9;
```

spread operator

- spreading of elements of an iterable collection (like an array or even a string) into both literal elements and individual function parameters

```
const odds = [1, 3, 5]
const even = [0, 2, 4, ...odds] // [0, 2, 4, 1, 3, 5]

const str = 'hello'
const chars = [...str] // ["h", "e", "l", "l", "o"]
```

destructuring

- destructuring allows binding using pattern matching
- support for matching **arrays** and **objects**
- fail-soft, producing **undefined** values when not found

```
const [x, , y] = [1, 2, 3] // x = 1, y = 3
const {name} = { name: 'hello' } // name = 'hello'

// you can rename them as needed
const {name: a} = { name: 'hello' } // a = 'hello'
```

template strings

- syntactic sugar for constructing strings
- Everything inside ${} is JS land
- tag can be added to allow the string construction to be customized
- raw string

```
const subject = 'class'
const time = 'class'
console.log(`hello ${subject}, how are you ${time}`)

const plus2 = ((str, num) => {
return num + 2
})

const four = plus2`str line 1 ${2} str line 2`
// four === 4
String.raw`foo\n${ 42 }bar` // foo\n42bar
```

classes

- sugarcoted prototype-based OO pattern
- more intuitive and boilerplate-free classes
- classes support prototype-based **inheritance**, **super** calls, **instance** and **static** methods and **constructors**

```
class Person {
 constructor(name) {
   this.name = name;
 }
 get yellName() {
   return this
     .name
     .toUpperCase();
 }
 set rename(name) {
   this.name = name;
 }
}
```

enhanced object literals

- support setting the **prototype** at construction
- shorthand for *foo: foo* assignments
- defining **methods**
- making **super** calls
- computing property names with expressions

```
const name = 'Joe';
const obj = {
 __proto__: thePrototype,
 name,
 sayHello() {
   return 'hello';
 },
 [`prop_${ 3}`]: 53,
};
```

modules

- **exporting/importing** values from/to modules without global namespace pollution
- two different types of export, **named** and **default**. you can have multiple named exports per module but only one default export
- no code executes until requested **modules** are available and processed

```
// lib/myLib.js
export const PI = 3.1415
export default n => n + 1

// main.js
import plusOne, {PI} from 'lib/myLib.js'

import * as utils from 'lib/myLib.js' // import all named exports as utils
import 'lib/myLib.js' // import myLib for its side effects only
import{PI as pii} from 'lib/myLib.js' // rename exports during import
import('lib/myLib.js').then() // dynamic import
```

promises

- library for **asynchronous** programming
- representation of a value that may be made available in the future
- no more **callback hell**
- used mostly for **API** calls

```
const p1 = new Promise(function(resolve, reject) {
 setTimeout(function() {
   resolve('hello');
 }, 300);
});

p1.then(res => {
 console.log(res); // prints hello after 300ms
});
```

Array methods

- **Array** got a few new methods

```
Array.from(document.querySelectorAll('h1'))
Array.of(3, 5, 7) // [3, 5, 7]

[0, 0, 0].fill(7, 1) // [0, 7, 7]
[1, 2, 3].find(n => n == 2) // 2
[1, 2, 3].findIndex(n => n == 2) // 1

['a', 'b', 'c'].copyWithin(0, 1) // ['b', 'c', 'c']
['a', 'b', 'c'].entries() // iterator [0, "a"], [1,"b"], [2,"c"]
['a', 'b', 'c'].keys(0, 1) // iterator 0, 1, 2
['a', 'b', 'c'].values(0, 1) // iterator "a", "b", "c"
```

Math & Number methods

- **Math** & **Number** got a few new methods

```
Math.acosh(3) // 1.762747174039086
Math.hypot(3, 4) // 5
Math.trunc(42.7) // 42
Math.imul(Math.pow(2, 32) - 1, Math.pow(2, 32) - 2) // 2

Number.isNaN(3) // false  ES5 version (isNaN) was bad
Number.isNaN(NaN) // true
Number.isFinite(3) // true
Number.isFinite(Infinity) // false
Number.isSafeInteger(42) // true
Number.isSafeInteger(9007199254740992) // false
Number.imul(Math.pow(2, 32) - 1, Math.pow(2, 32) - 2) // 2
```

Object.assign

- merge multiple objects into one
- make copy of an **object**, one level deep

```
const dest = { foo: 1 }
const src1 = { bar: 2 }
const src2 = { baz: 3 }

Object.assign(dest, src1, src2) // {foo: 1, bar: 2, baz: 3}
```

Symbols

- new primitive type
- unique but not private
- **Object.getOwnPropertySymbols**
- this won't work **new Symbol()**

```
const sym1 = Symbol()
const sym2 = Symbol(1)

typeof sym1 // symbol
sym1 === Symbol() // false

const obj = {}
obj[sym2] = 3
Object.getOwnPropertySymbols(obj) // [Symbol(1)]
```

Set

- like **Array** but with unique values
- Can contain any value eg. **primitive**, **object**, **function**
- you can iterate over values with **for ... of** loop

```
const mySet = new Set()
mySet.add(1)
mySet.add(5)
mySet.add(5) // Set [ 1, 5 ]
mySet.add('some text')
mySet.add({ foo: 1 })
mySet.has(5) // true
mySet.delete(5)
mySet.size // 3

const obj = { foo: 1 }
mySet.add(obj) // why is this ok?
```

WeakSet

- can contain only **objects**
- references to objects in the collection are held weakly. if there is no other reference to an object stored in the **WeakSet**, they can be **garbage collected**
- not iterable

```
const mySet = new WeakSet()
const obj = { foo: 1 }
mySet.add(obj)
mySet.has(obj) // true
```

Map

- holds key-value pairs and remembers the original insertion order of the keys
- like regular **Object** + key can be anything, not just primitive value
- you can iterate over values with for … of loop
- may perform better in scenarios involving frequent addition and removal of key pairs

```
const myMap = new Map()
myMap.set(2, 'two')
myMap.set(NaN, 'not a number')
myMap.get(NaN) // "not a number"
```

WeakMap

- key must be **Object**
- keys are weakly referenced. if there is no other reference to an object that's used as key it can be garbage collected
- not iterable

```
const myMap = new WeakMap()
const o1 = {}
const o2 = window

mySet.set(o1, 'this can be anything')
mySet.set(o2, () => {console.log(1)})
```

generator

- **function*** declaration (**function** keyword followed by an *) defines a **generator function**, which returns a **Generator** object
- **Generator** object is returned by a **generator function** and it conforms to both the **iterable** protocol and the **iterator** protocol.
- easy way to create iterators

```
// function* generator == function *generator == function * generator
function* generator(i) {
 yield i;
 yield i + 10;
 const x = yield i + 20;
 yield x;
}
var gen = generator(10);
console.log(gen.next()); // {value: 10, done: false}
console.log(gen.next()); // {value: 20, done: false}
console.log(gen.next()); // {value: 30, done: false}
console.log(gen.next('foo')); // {value: 'foo', done: false}
console.log(gen.next()); // {value: undefined, done: true}
```

iterable and iterator

- iterable - an object must implement the **@@iterator** method (provided by Symbol.iterator)
- iterator - an object that implements a **next()** method that returns **{done**: Boolean, **value**: any JS value**}**
- **String**, **Array**, **TypedArray**, **Map** and **Set** are all built-in **iterables**

```
function idMaker() {
 var index = 0;
 return {
    next: function(){
        return {value: index++, done: false};
    }
 };
}
var it = idMaker();
console.log(it.next().value); // '0'
console.log(it.next().value); // '1'
console.log(it.next().value); // '2'
```

for. . . of

- loop iterating over **iterable** objects

```
function* foo(){
 yield 1;
 yield 2;
}
for (let o of foo()) {
 console.log(o);  // 1, 2
}
```

Proxy

- **Proxy** object is used to define custom behavior for fundamental operations (e.g. property lookup, assignment, enumeration, function invocation, etc)

```
const handler = {
 get: function(obj, prop) {
   return prop in obj ?
     obj[prop] :
     37;
 },
};

const p = new Proxy({}, handler);
p.a = 1;
p.b = undefined;

console.log(p.a, p.b); // 1, undefined
console.log('c' in p, p.c); // false, 37
```

POWERED BY: nsoft sportradar

SPARK. SCHOOL

THANK YOU

for attention