
ES6


WHAT IS
ES6?


JavaScript is born as LiveScript
1995
ECMAScript standard is established
1997
ES3 comes out and IE5 is all the rage
1999
XMLHttpRequest, a.k.a. AJAX, gains popularity
2000
ES5 comes out and standard JSON
2009
ES6/ECMAScript2015 comes out
2015
ES7/ECMAScript2016 comes out
2016
ES.Next
2017

What is ES6

- Ecma stands for (European Computers Manufacturers Association) Specification is called **E**cma**S**cript a scripting language which has Many Implementors.

What is ES6

- Ecma stands for (European Computers Manufacturers Association) Specification is called **E**cma**S**cript a scripting language which has Many Implementors.
- Javascript is Implementation of Ecma script

What is ES6

- Ecma stands for (European Computers Manufacturers Association) Specification is called **E**cma**S**cript a scripting language which has Many Implementors.
- Javascript is Implementation of Ecma script
- Standard for all Browsers to Interpret Javascript

What is ES6

- Ecma stands for (European Computers Manufacturers Association) Specification is called **E**cma**S**cript a scripting language which has Many Implementors.
- Javascript is Implementation of Ecma script
- Standard for all Browsers to Interpret Javascript
- Latest version of EcmaScript Specification is ES6 also Called **Harmony**

Compatibility Table

- Refer

  https://kangax.github.io/compat-table/es6/

How to use Es6 Today

- Although Today's browser support 97% of features some are still pending
- Use Babel Transpiler to transpile Es6 to Es5

  https://babeljs.io
- Webpack 2 for bundling and Transpiling

  https://webpack.js.org/

  https://blog.madewithenvy.com/getting-started-with-webpack-2-ed2b86c68783
- Other notable transpilers Grunt/Traceur/Gulp

What is Covered under Ecma Script

- Language Syntax - parsing, keywords, operators

What is Covered under Ecma Script

- Language Syntax - parsing, keywords, operators
- Types - undefined, null, boolean, number, string, and object

What is Covered under Ecma Script

- Language Syntax - parsing, keywords, operators
- Types - undefined, null, boolean, number, string, and object
- Prototypes and inheritance

What is Covered under Ecma Script

- Language Syntax - parsing, keywords, operators
- Types - undefined, null, boolean, number, string, and object
- Prototypes and inheritance
- Built-in object and functions including Math, JSON, Array, object methods, etc

What is Covered under Ecma Script

- Language Syntax - parsing, keywords, operators
- Types - undefined, null, boolean, number, string, and object
- Prototypes and inheritance
- Built-in object and functions including Math, JSON, Array, object methods, etc
- SpiderMonkey, Trident, and V8 also implement this specification. They are used in Firefox, IE and Chrome/Node.js/Opera respectively.

What are benefits of ES6

- Most of ES6 new features provides syntactical sugar over ES5 and older versions
- Makes Code cleaner
- Improves Readability
- Improves Coding Standards
- Makes code better Maintainable,Re usable
- Easy to learn and Implement
- Can be used with any Modern JS Framework like ReactJS,Angular 4,Ember ,Vue, etc ..... (Never ending Frameworks)

ES6 Features

- Classes
- Modules
- Arrow Functions
- Block Scoping with Let,Const
- Object and Array Destructuting
- Rest and Spread Operators
- Template Literals
- Generators and Interators
- Promises
- Symbols
- Collections - (Maps/Sets/Weak Maps)

ES6 Classes

- ES6 classes are a simple sugar over the prototype-based OO pattern. Having a single convenient declarative form makes class patterns easier to use, and encourages interoperability. Classes support prototype-based inheritance, super calls, instance and static methods and constructors.

Class

ES5

```
function Car(data) {
  this.model = data.model;
  this.year = data.year;
  this.wheels = data.wheels || 4;
}

Car.prototype.drive = function() {
  console.log(
    'Driving ' + this.model);
}

var yugo = new Car({
  model: 'Yugo 55', year: 1985 });

yugo.drive();
```

CLASSES

ES5

```
function Car(data) {
  this.model = data.model;
  this.year = data.year;
  this.wheels = data.wheels || 4;
}

Car.prototype.drive = function() {
  console.log(
    'Driving ' + this.model);
}

var yugo = new Car({
  model: 'Yugo 55', year: 1985 });

yugo.drive();
```

ES6

```
class Car {
  constructor (data) {
    this.model = data.model;
    this.year = data.year;
    this.wheels = data.wheels || 4;
  }

  drive() {
    console.log(
      'Driving ' + this.model);
  }
}

let yugo = new Car({
  model: 'Yugo 55', year: 1985 });
yugo.drive();
```

PROPERTIES

```
class Car {
  constructor ({model, year, wheels=4}) {
    this.model = model; this.year = year; this.wheels = wheels;
  }

  get isNew() { return this._isNew; }
  set isNew(value) { this._isNew = value; }

  drive() {
    console.log(`Driving ${this.model}`); this.isNew = false;
  }
}
let yugo = new Car({ model: 'Yugo 55', year: 1985 });

yugo.drive();
console.log(yugo.isNew);
```

INHERITANCE

```
class Car { ... }

class Truck extends Car {
  constructor ({model, year, wheels=6}) {
    super({model, year, wheels});
  }

  drive () {
    super.drive();
    console.log(`Driving like a boss with ${this.wheels} wheels`);
  }
}

let actros = new Truck({ model: 'Actros', year: 2005 });
actros.drive();
```

- Note: favor composition over inheritance.

STATIC METHODS

ES5

```
function Car(data) {
  // ...
}

Car.drive = function(data) {
  console.log('Driving...');
}

//var yugo = new Car({
// model: 'Yugo 55', year: 1985 });
//yugo.drive();

Car.drive({ road: 'autobahn' });
```

ES6

```
class Car {
  // ...
  static drive(data) {
    console.log('Driving...');
  }
}

//let yugo = new Car({
// model: 'Yugo 55', year: 1985 });
//yugo.drive();

Car.drive({ road: 'autobahn' });
```

Modules

- Language-level support for modules for component definition. Codifies patterns from popular JavaScript module loaders (AMD, CommonJS). Runtime behaviour defined by a host-defined default loader. Implicitly async model – no code executes until requested modules are available and processed.

Modules

MODULES

```
// lib/math.js
export function sum(x, y) {
  return x + y;
}
export var pi = 3.141593;

// app.js
import * as math from "lib/math";
alert("2π = " + math.sum(math.pi, math.pi));

// otherApp.js
import {sum, pi} from "lib/math";
alert("2π = " + sum(pi, pi));
```

Arrow Functions

- Anonymous can be denoted with => to make the code neater
- Arrow functions can be useful to maintain scopes

ARROWS

ES5

```
var square = function(x) {
  return x * x;
};
var add = function(x, y) {
  return x + y;
};
var total = function() {
  return square(add(5, 3));
};

console.log('5 * 5 = ', square(5));
console.log('2 + 3 = ', add(2, 3));
console.log(
  '(5 + 3)*(5 + 3) = ', total());
```

ARROWS

ES5

```
var square = function(x) {
  return x * x;
};
var add = function(x, y) {
  return x + y;
};
var total = function() {
  return square(add(5, 3));
};

console.log('5 * 5 = ', square(5));
console.log('2 + 3 = ', add(2, 3));
console.log(
  '(5 + 3)*(5 + 3) = ', total());
```

ES6

```
let square = x => x * x;
let add = (x, y) => x + y;
let total = () => square(add(5,3));

console.log('5 * 5 = ', square(5));
console.log('2 + 3 = ', add(2, 3));
console.log(
  '(5 + 3)*(5 + 3) = ', total());
```

MULTILINE ARROWS

ES5

```
var add = function(x, y) {
  var result = x + y;
  return result;
};

console.log('2 + 3 = ', add(2, 3));
```

MULTILINE ARROWS

ES5

```
var add = function(x, y) {
  var result = x + y;
  return result;
};

console.log('2 + 3 = ', add(2, 3));
```

ES6

```
let add = (x, y) => {
  let result = x + y;
  return result;
};

console.log('2 + 3 = ', add(2, 3));
```

THIS

ES5

```
var obj = {
  index: 1,

  loadData: function() {
    var self = this;

   $.get('http://ip.jsontest.com')
      .then(function(data) {
        console.log(
          'IP: ' + data.ip,
          'Index: ' + self.index)
      });
  }
};

obj.loadData();
```

ES6

```
let obj = {
  index: 1,

  loadData: function() {
   $.get('http://ip.jsontest.com')
      .then((data) => {
        console.log(
          'IP: ' + data.ip,
          'Index: ' + this.index)
      });
  }
};

obj.loadData();
```

Let,Const,Var

- With ES6, we went from declaring variables with var to use let/const.
- The issue with var is the variable leaks into other code block such as for loops or if blocks.

ES5

```
1  var x = 'outer';
2  function test(inner) {
3    if (inner) {
4      var x = 'inner'; // scope whole function
5      return x;
6    }
7    return x; // gets redefined because line 4 declaration is hoiste
8  }
9
10 test(false); // undefined
11 test(true); // inner
```

> var **hoisting**:
>
> - `var` is function scoped. It is availble in the whole function even before being declared.
> - Declarations are Hoisted. So you can use a variable before it has been declared.
> - Initializations are NOT hoisted. If you are using `var` ALWAYS declare your variables at the top.
> - After applying the rules of hoisting we can understand better what's happening:

ES5

```
var x = 'outer';
function test(inner) {
  var x; // HOISTED DECLARATION
  if (inner) {
    x = 'inner'; // INITIALIZATION NOT HOISTED
    return x;
  }
  return x;
}
```

ES6

```
1  let x = 'outer';
2  function test(inner) {
3    if (inner) {
4      let x = 'inner';
5      return x;
6    }
7    return x; // gets result from line 1 as expected
8  }
9
10 test(false); // outer
11 test(true); // inner
```

```
function test() {
  const PI = 3.141569;

  // PI = 6; -> ERROR

  console.log(PI);
}
```

```
var list = document.getElementById('list');

for (let i = 1; i <= 5; i++) {
  let item = document.createElement('li');
  item.appendChild(document.createTextNode('Item ' + i));

  item.onclick = function(ev) {
    console.log('Item ' + i + ' is clicked.');
  };
  list.appendChild(item);
}

// to achieve the same effect with 'var'
// you have to create a different context
// using a closure to preserve the value
for (var i = 1; i <= 5; i++) {
  var item = document.createElement('li');
  item.appendChild(document.createTextNode('Item ' + i));

    (function(i){
        item.onclick = function(ev) {
            console.log('Item ' + i + ' is clicked.');
        };
    })(i);
  list.appendChild(item);
}
```

Object and Array Destructuring

- ES6 desctructing is very useful and consise. Follow this examples:

DESTRUCTURING

```
let numbers = [10, 20];
let [a, b] = numbers;

console.log(a, b);

let position = { lat: 42.34455, lng: 17.34235 };
let {lat, lng} = position;

console.log(lat, lng);
```

DESTRUCTURING

COMPLEX OBJECTS

```
let book = {
  title: 'Lord of the Rings',
  author: 'J.R.R. Tolkien',
  pages: 550,
  tags: ['fantasy', 'fiction'],
  price: {
    hardcover: 34.5,
    softcover: 22.5
  }
};

let {author, tags: [,tag], price: {softcover}} = book;

console.log(author, tag, softcover);
```

DESTRUCTURING

FUNCTION PARAMETERS

```
let httpPost = function(url, {
    cache = true,
    contentType = 'application/json',
    timeout = 2500,
    headers = {}
  }) {
  console.log(url, cache, contentType, timeout, headers);
};

httpPost('http://google.com', {
  cache: false,
  headers: {
    Authorization: 'Bearer SOMETOKEN'
  }
});
```

DESTRUCTURING ASSIGNMENT

destructuring of Arrays into individual variables

ES6

```
var list = [ 1, 2, 3 ];
var [ a, , b ] = list;
[ b, a ] = [ a, b ];
```

ES5

```
var list = [ 1, 2, 3 ];
var a = list[0], b = list[2];
var tmp = a;
a = b;
b = tmp;
```

DESTRUCTURING ASSIGNMENT

Fail-soft destructuring, optionally with defaults.

ES6

```
var list = [ 7 ];
var [ a = 1, b = 3, c ] = list;
a === 7
b === 3
c === undefined
```

ES5

```
var list = [ 7 ];
var a = typeof list[0] !== "undefined" ? list[0] : 1;
var b = typeof list[1] !== "undefined" ? list[1] : 3;
var c = typeof list[2] !== "undefined" ? gslist[2] : undefined;
a === 7;
b === 3;
c === undefined;
```

Rest Paramters

- Provide Arguments Efficiently and neatly
-

ES5

```
1  function printf(format) {
2    var params = [].slice.call(arguments, 1);
3    console.log('params: ', params);
4    console.log('format: ', format);
5  }
6
7  printf('%s %d %.2f', 'adrian', 321, Math.PI);
```

We can do the same using the rest operator `...`.

ES6

```
1  function printf(format, ...params) {
2    console.log('params: ', params);
3    console.log('format: ', format);
4  }
5
6  printf('%s %d %.2f', 'adrian', 321, Math.PI);
```

Rest Parameters

FUNCTIONS: REST PARAMETERS

ES5

```
function add(x) {
  var result = x;

  // arguments.forEach(...)
  for (var i = 1;
      i < arguments.length; i++) {
    result += arguments[i];
  }

  return result;
}

console.log(add(1, 2, 3, 4, 5));
```

ES6

```
function add(x, ...numbers) {
  var result = x;

  numbers.forEach(function(y) {
    result += y;
  });

  return result;
}

console.log(add(1, 2, 3, 4, 5));
```

Spread Operator ...

- Spread operator changed the way coding was done traditionally in Javascript, Can be Applied only on iterables

ES5

```
1  var array1 = [2,100,1,6,43];
2  var array2 = ['a', 'b', 'c', 'd'];
3  var array3 = [false, true, null, undefined];
4
5  console.log(array1.concat(array2, array3));
```

In ES6, you can flatten nested arrays using the spread operator:

ES6

```
1  const array1 = [2,100,1,6,43];
2  const array2 = ['a', 'b', 'c', 'd'];
3  const array3 = [false, true, null, undefined];
4
5  console.log([...array1, ...array2, ...array3]);
```

Spread Operator

SPREAD OPERATOR

ES5

```
function add(x, y, z) {
  return x + y + z;
}

var numbers = [2, 4, 6];

console.log(
  add.apply(null, numbers));
```

Spread Operator

SPREAD OPERATOR

ES5

```
function add(x, y, z) {
  return x + y + z;
}

var numbers = [2, 4, 6];

console.log(
  add.apply(null, numbers));
```

ES6

```
function add(x, y, z) {
  return x + y + z;
}

let numbers = [2, 4, 6];

console.log(add(...numbers));

// Example 2
let n2 =
  [1, 2, ...numbers, 7, ...[2, 3]];
console.log(n2);
// [1, 2, 2, 4, 6, 7, 2, 3]
```

Spread Operator

ES5

```
1  Math.max.apply(Math, [2,100,1,6,43]) // 100
```

In ES6, you can use the spread operator:

ES6

```
1  Math.max(...[2,100,1,6,43]) // 100
```

Spread Operator

- Spread Operator on non iterables like object is underproposal

Rest Properties

Rest properties collect the remaining own enumerable property keys that are not already picked off by the destructuring pattern. Those keys and their values are copied onto a new object.

```
let { x, y, ...z } = { x: 1, y: 2, a: 3, b: 4 };
x; // 1
y; // 2
z; // { a: 3, b: 4 }
```

Spread Properties

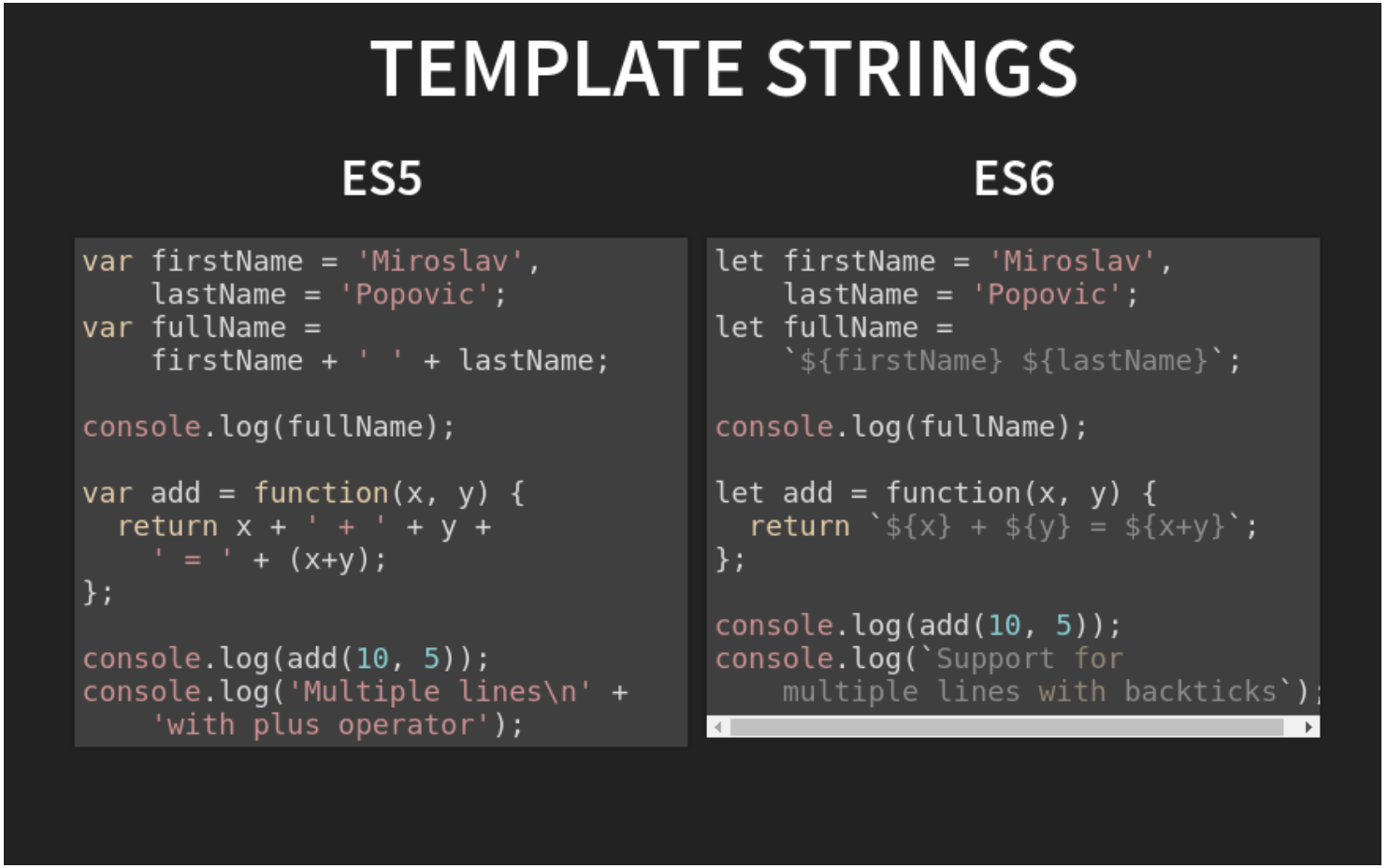
Spread properties in object initializers copies own enumerable properties from a provided object onto the newly created object.

```
let n = { x, y, ...z };
n; // { x: 1, y: 2, a: 3, b: 4 }
```

Literals

- Template Literals (Interpolation)
- Object Literals
- Unicode Literals
- Binary and Octal Literals

Template Literals



Object Literals

EXTENDED OBJECT LITERALS

```
var obj = {
    // defining prototype on object creation
    __proto__: theProtoObj,

    // shortcut for ‘handler: handler’
    handler,

    // shortcut for methods - 'toString: function() {}'
    toString() {
     // base class method call with 'super'
     return 'd ' + super.toString();
    },

    // dynamic property names
    [ 'prop_' + (() => 42)() ]: 42
};
```

Symbols

- A new primitive type of course! The seventh type of value in JavaScript to be exact. They are similar to Symbols in Ruby, but are not the same exact thing.

  ```
  var computerName = Symbol('awesome desktop');
  ```

Symbols

- Can be used for UUID’s

```
var camsComputer = Symbol('awesome desktop');
var camsOtherComputer = Symbol('awesome desktop');
// these will not equal each other!
console.assert(
    camsComputer === camsOtherComputer,
    'these are not equal!'
);
> these are not equal!
```

Symbols

USING SYMBOLS AND CONSTANTS

ES6

```
// create a unique symbol
const isComplete = Symbol("isComplete");
...
var codeFromJustin = document.getElementById('code-block-justin')

if(codeFromJustin[isComplete]) {
    deployToProduction();
} else {
    sendBackToDev();
}
codeFromJustin[isComplete] != codeFromJustin.isComplete
codeFromJustin[isComplete] != codeFromJustin['isComplete']
```

`codeFromJustin[isComplete]` can only be obtained by having a reference to `isComplete`

Maps,Sets

- ES6 introduces two new data structures: Maps and Sets

Maps,Sets

- ES6 introduces two new data structures: Maps and Sets
- Maps − enables mapping a key to a value.

Maps,Sets

- ES6 introduces two new data structures: Maps and Sets
- Maps − enables mapping a key to a value.
- Sets − Sets are similar to arrays. However, sets do not encourage duplicates.

Maps

- The Map object is a simple key/value pair. What makes it different from Object is Keys and values in a map may be primitive or objects.

```
var map = new Map()
map.set(new Date(), function today () {})
map.set(() => 'key', { pony: 'foo' })
map.set(Symbol('items'), [1, 2])
```

Maps

You can also provide `Map` objects with any object that follows the *iterable* protocol and produces a collection such as `[['key', 'value'], ['key', 'value']]`.

```
var map = new Map([
  [new Date(), function today () {}],
  [() => 'key', { pony: 'foo' }],
  [Symbol('items'), [1, 2]]
])
```

Sets

- Set is similar to an array with an exception that it cannot contain duplicates. In other words, it lets you store unique values. Sets support both primitive values and object references.

Sets

```
var mySet = new Set();

mySet.add(1); // Set { 1 }
mySet.add(5); // Set { 1, 5 }
mySet.add(5); // Set { 1, 5 }
mySet.add('some text'); // Set { 1, 5, 'some text' }
var o = {a: 1, b: 2};
mySet.add(o);

mySet.add({a: 1, b: 2}); // o is referencing a different object so this is okay

mySet.has(1); // true
mySet.has(3); // false, 3 has not been added to the set
mySet.has(5);              // true
mySet.has(Math.sqrt(25));  // true
mySet.has('Some Text'.toLowerCase()); // true
mySet.has(o); // true

mySet.size; // 5

mySet.delete(5); // removes 5 from the set
mySet.has(5);    // false, 5 has been removed

mySet.size; // 4, we just removed one value
console.log(mySet);// Set {1, "some text", Object {a: 1, b: 2}, Object {a: 1, b: 2
```

- For more details of Maps and sets refer http://2ality.com/2015/01/es6-maps-sets.html
- MDN References of Maps and sets
- https://ponyfoo.com/articles/es6-maps-in-depth

Weak Maps and Weak Sets

- Both Weak Maps and Weak Sets are used for Garbage Collections
- For more details refer

  https://ponyfoo.com/articles/es6-weakmaps-sets-and-weaksets-in-depth