Something about ES6 Javascript Standard

I am a Virus

ECMAScript® 2015 Language Specification

In other word, ES6.

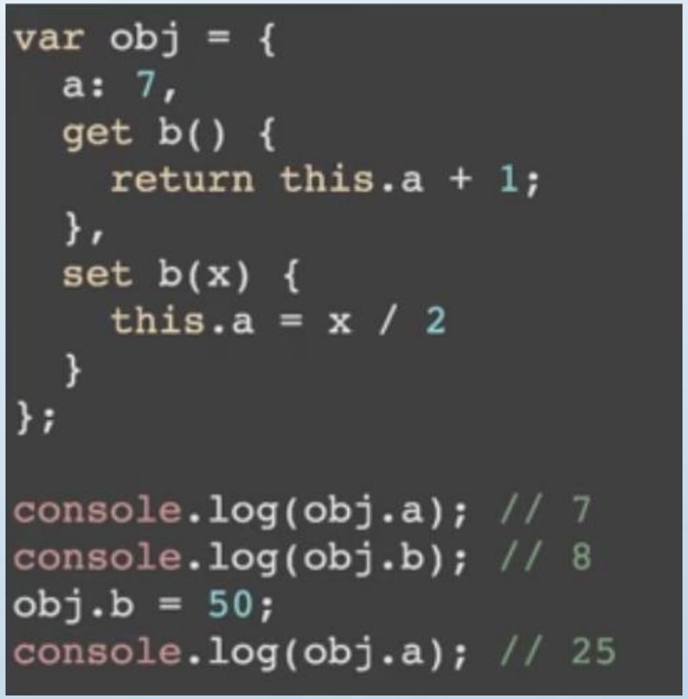
If you don't like ES6, you can still type code in ES5, ES4, ... or native Javascript.

Reference:

http://www.ecma-international.org/ecma-262/6.0/
https://en.wikipedia.org/wiki/ECMAScript
https://developer.mozilla.org/en-US/docs/Web/JavaScript

ES5 – Getters/Setters

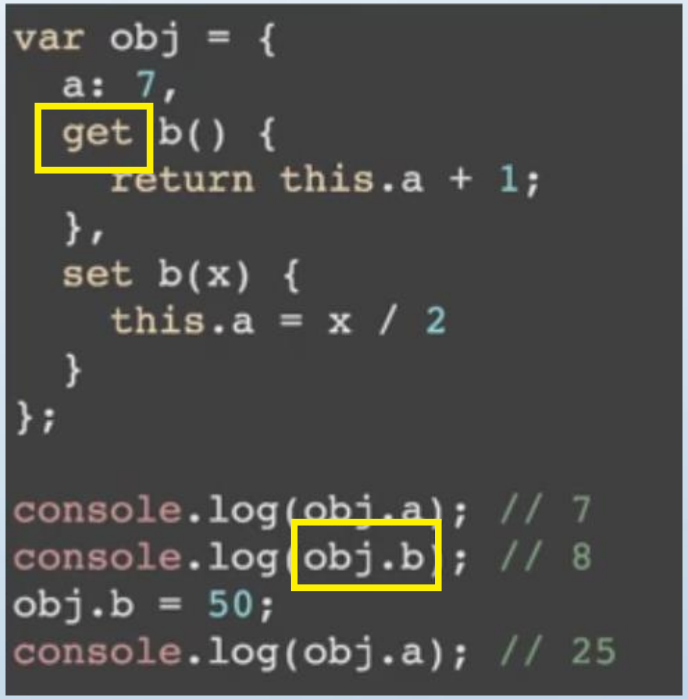
Getters and Setter are psudo-properties that return or set a dynamically computed value.



```
var obj = {
  a: 7,
  get b() {
    return this.a + 1;
  },
  set b(x) {
    this.a = x / 2
  }
};

console.log(obj.a); // 7
console.log(obj.b); // 8
obj.b = 50;
console.log(obj.a); // 25
```

ES5 – Getters/Setters

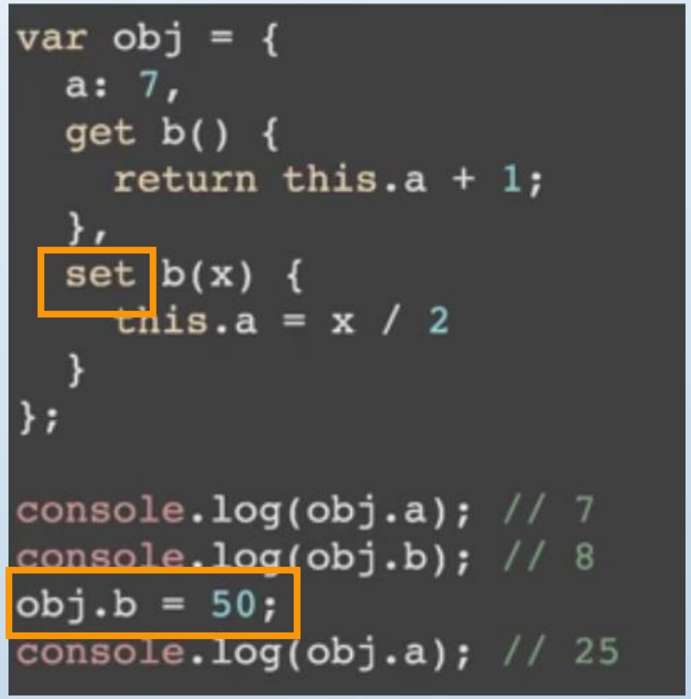
Getters and Setter are psudo-properties that return or set a dynamically computed value.



```
var obj = {
  a: 7,
  get b() {
    return this.a + 1;
  },
  set b(x) {
    this.a = x / 2
  }
};

console.log(obj.a); // 7
console.log(obj.b); // 8
obj.b = 50;
console.log(obj.a); // 25
```

ES5 – Getters/Setters

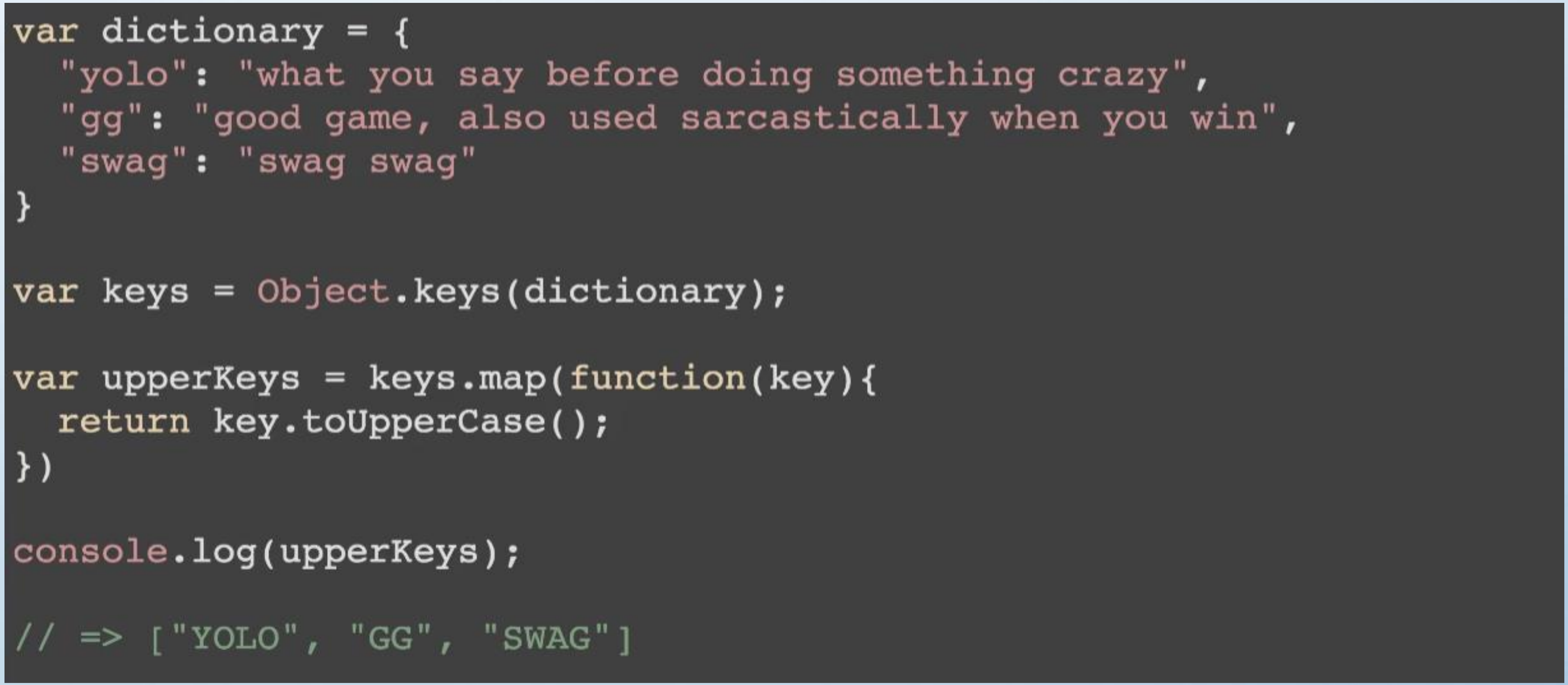
Getters and Setter are psudo-properties that return or set a dynamically computed value.



```
var obj = {
  a: 7,
  get b() {
    return this.a + 1;
  },
  set b(x) {
    this.a = x / 2
  }
};

console.log(obj.a); // 7
console.log(obj.b); // 8
obj.b = 50;
console.log(obj.a); // 25
```

ES5 – Object.keys

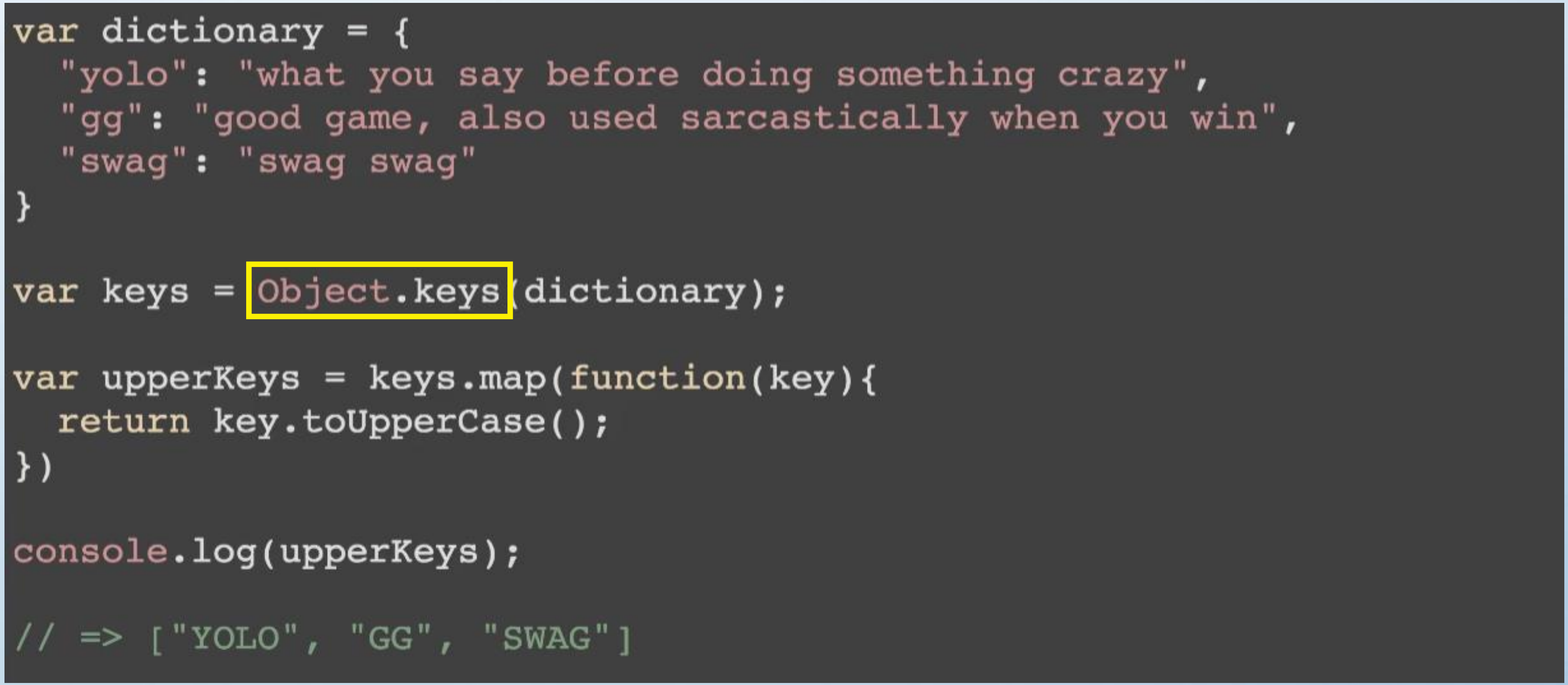
Converting the keys of an object to array.



```
var dictionary = {
  "yolo": "what you say before doing something crazy",
  "gg": "good game, also used sarcastically when you win",
  "swag": "swag swag"
}

var keys = Object.keys(dictionary);

var upperKeys = keys.map(function(key){
  return key.toUpperCase();
})

console.log(upperKeys);

// => ["YOLO", "GG", "SWAG"]
```

ES5 – Object.keys

Converting the keys of an object to array.



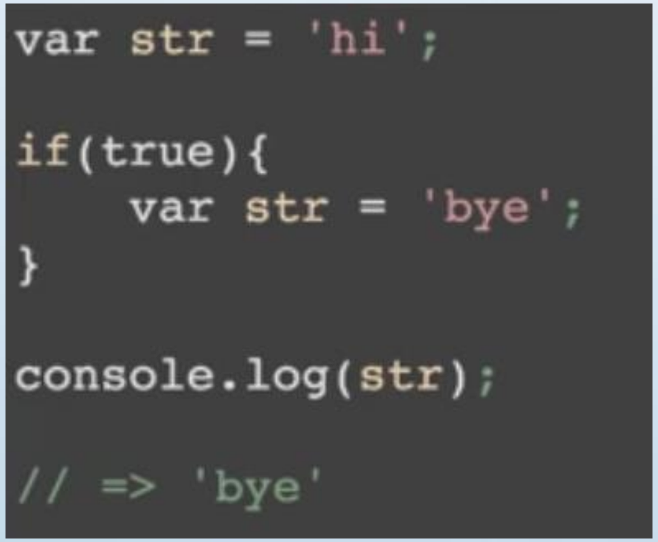
```
var dictionary = {
  "yolo": "what you say before doing something crazy",
  "gg": "good game, also used sarcastically when you win",
  "swag": "swag swag"
}

var keys = Object.keys(dictionary);

var upperKeys = keys.map(function(key){
  return key.toUpperCase();
})

console.log(upperKeys);

// => ["YOLO", "GG", "SWAG"]
```

ES2015 – Var, Let & Const

ES5 Var is not block scoped can have unexpected behavior.

Let & Const are block scoped to fix this.

ES5 var



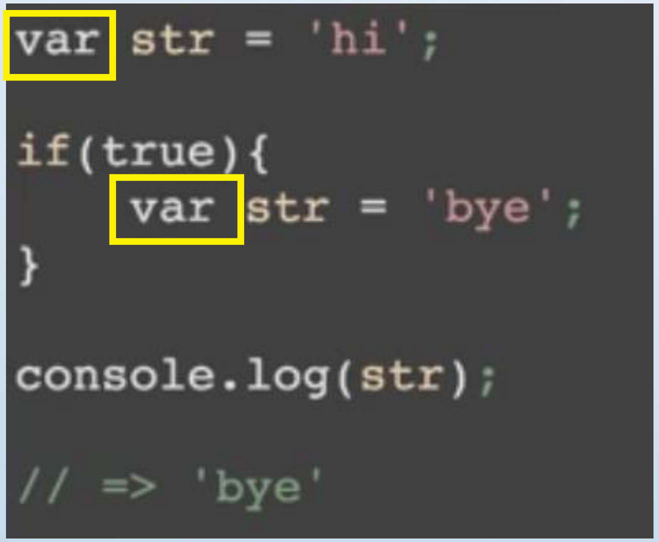
```
var str = 'hi';

if(true){
    var str = 'bye';
}

console.log(str);

// => 'bye'
```

ES2015 – Var, Let & Const

ES5 Var is not block scoped can have unexpected behavior.

Let & Const are block scoped to fix this.

ES5 var



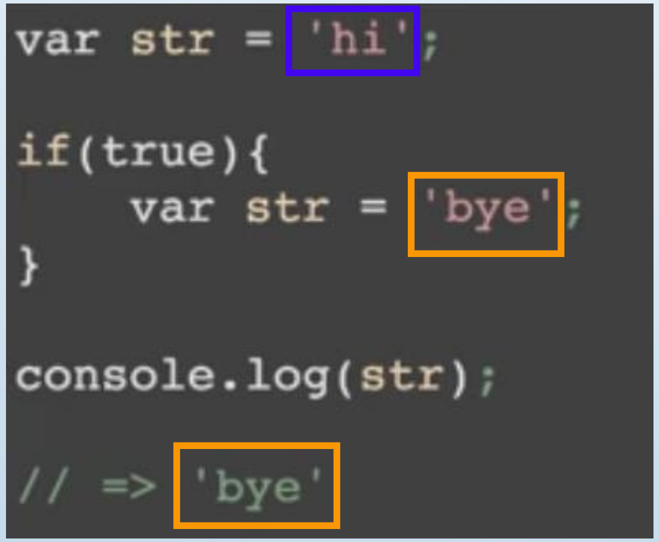
```
var str = 'hi';

if(true){
    var str = 'bye';
}

console.log(str);

// => 'bye'
```

ES2015 – Var, Let & Const

ES5 Var is not block scoped can have unexpected behavior.

Let & Const are block scoped to fix this.

ES5 var



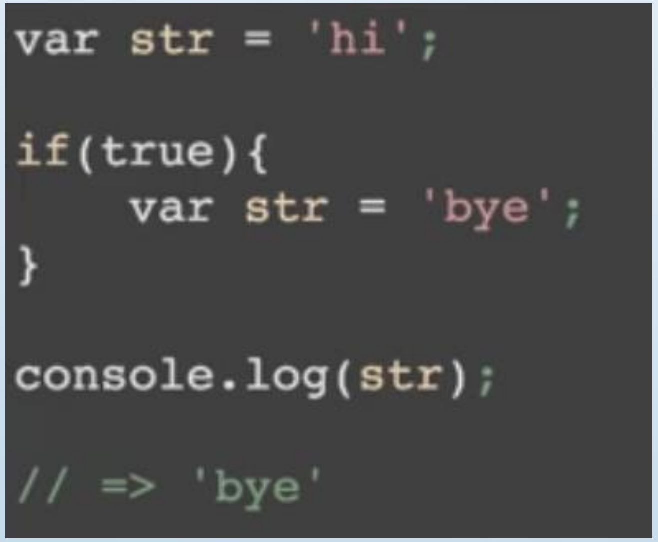
```
var str = 'hi';

if(true){
    var str = 'bye';
}

console.log(str);

// => 'bye'
```

ES2015 – Var, Let & Const

ES5 Var is not block scoped can have unexpected behavior.

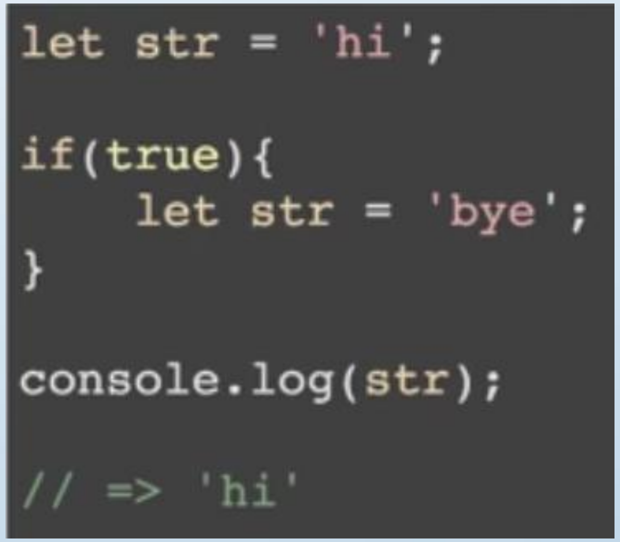
Let & Const are block scoped to fix this.

ES5 var



```
var str = 'hi';

if(true){
    var str = 'bye';
}

console.log(str);

// => 'bye'
```

ES2015 let



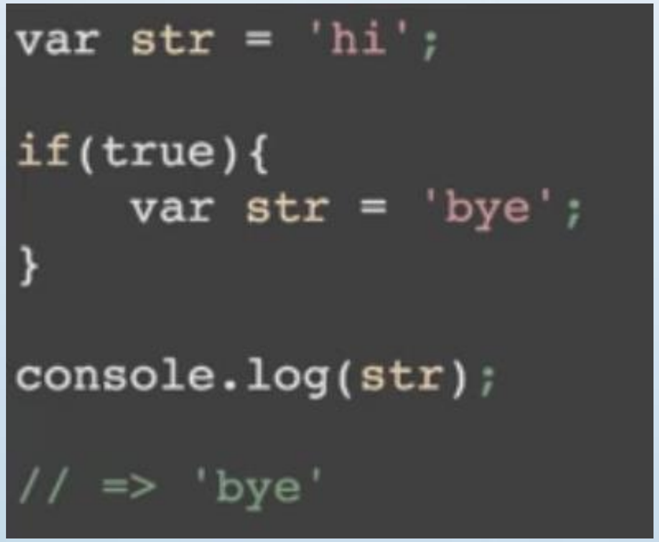
```
let str = 'hi';

if(true){
    let str = 'bye';
}

console.log(str);

// => 'hi'
```

ES2015 – Var, Let & Const

ES5 Var is not block scoped can have unexpected behavior.

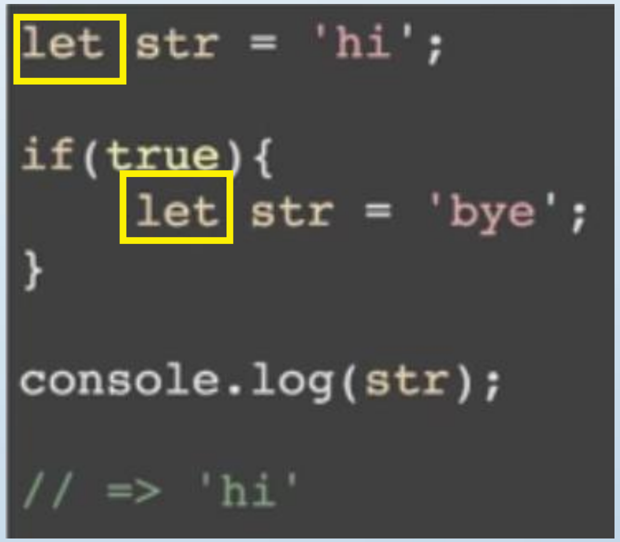
Let & Const are block scoped to fix this.

ES5 var



```
var str = 'hi';

if(true){
    var str = 'bye';
}

console.log(str);

// => 'bye'
```

ES2015 let



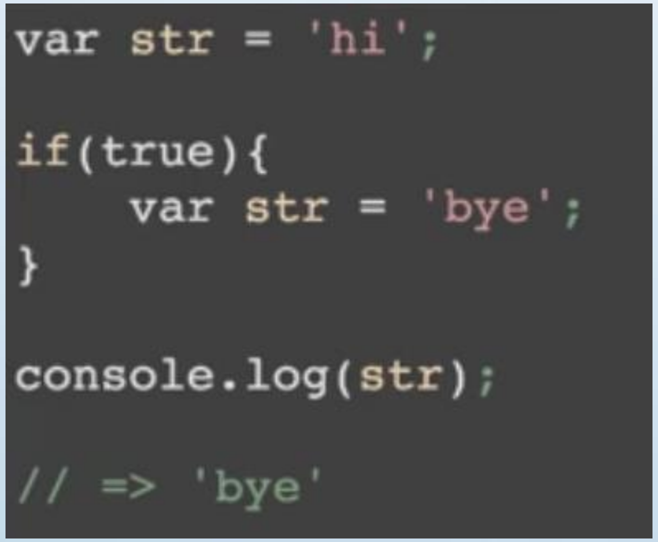
```
let str = 'hi';

if(true){
    let str = 'bye';
}

console.log(str);

// => 'hi'
```

ES2015 – Var, Let & Const

ES5 Var is not block scoped can have unexpected behavior.

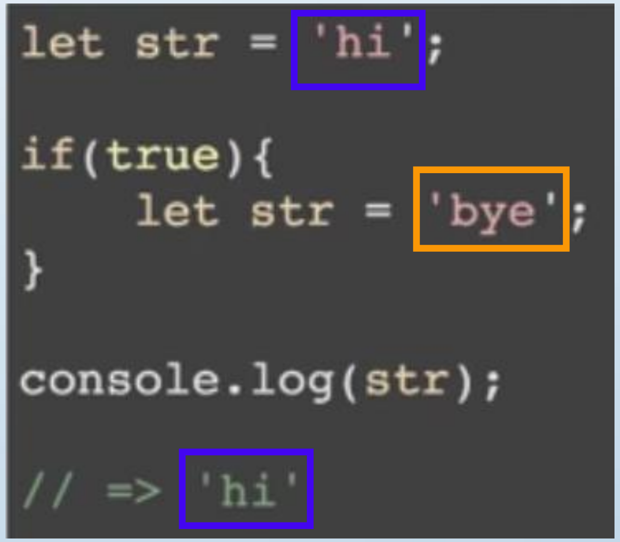
Let & Const are block scoped to fix this.

ES5 var



```
var str = 'hi';

if(true){
    var str = 'bye';
}

console.log(str);

// => 'bye'
```

ES2015 let



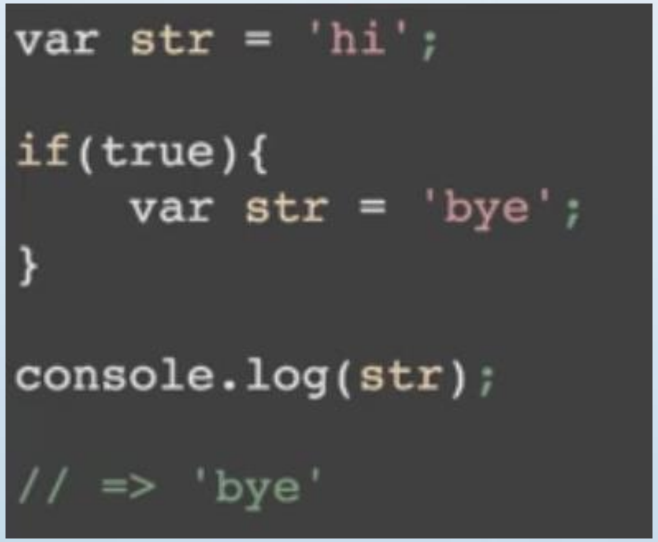
```
let str = 'hi';

if(true){
    let str = 'bye';
}

console.log(str);

// => 'hi'
```

ES2015 – Var, Let & Const

ES5 Var is not block scoped can have unexpected behavior.

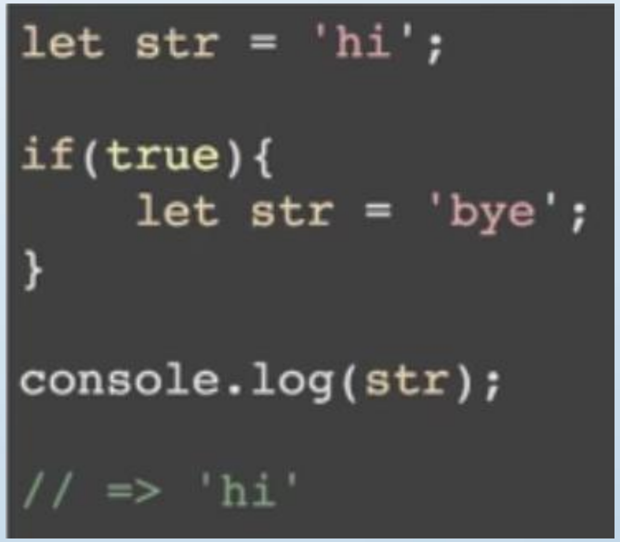
Let & Const are block scoped to fix this.

ES5 var



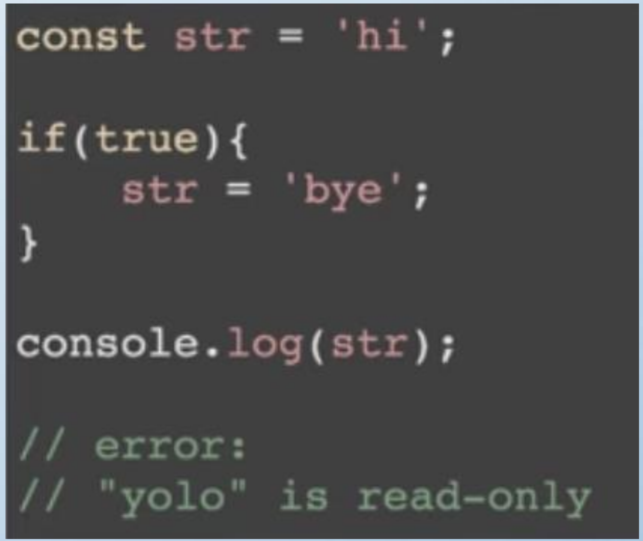
```
var str = 'hi';

if(true){
    var str = 'bye';
}

console.log(str);

// => 'bye'
```

ES2015 let



```
let str = 'hi';

if(true){
    let str = 'bye';
}

console.log(str);

// => 'hi'
```

ES2015 const



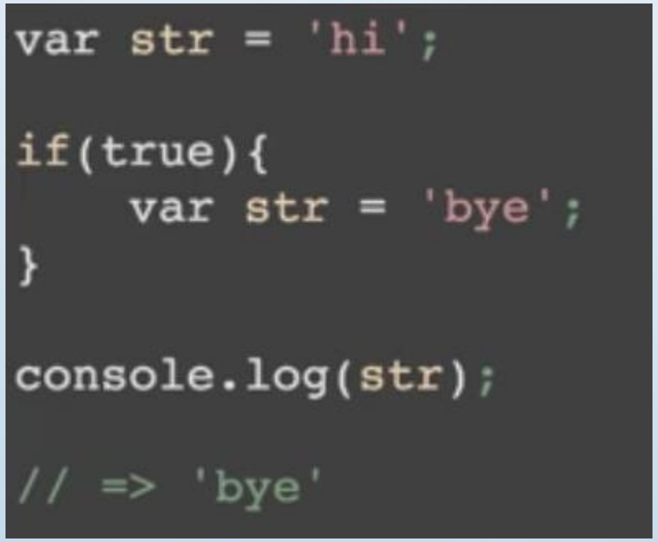
```
const str = 'hi';

if(true){
    str = 'bye';
}

console.log(str);

// error:
// "yolo" is read-only
```

ES2015 – Var, Let & Const

ES5 Var is not block scoped can have unexpected behavior.

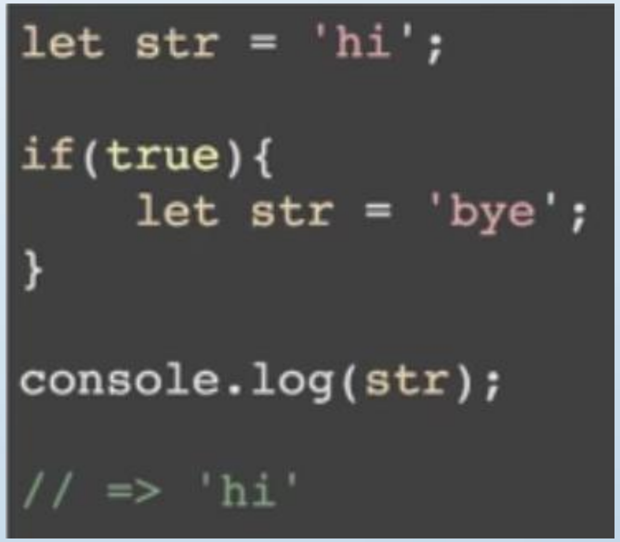
Let & Const are block scoped to fix this.

ES5 var



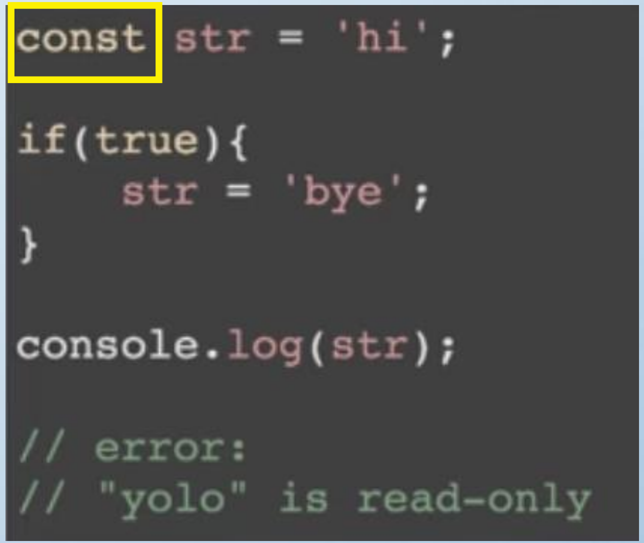
```
var str = 'hi';

if(true){
    var str = 'bye';
}

console.log(str);

// => 'bye'
```

ES2015 let



```
let str = 'hi';

if(true){
    let str = 'bye';
}

console.log(str);

// => 'hi'
```

ES2015 const

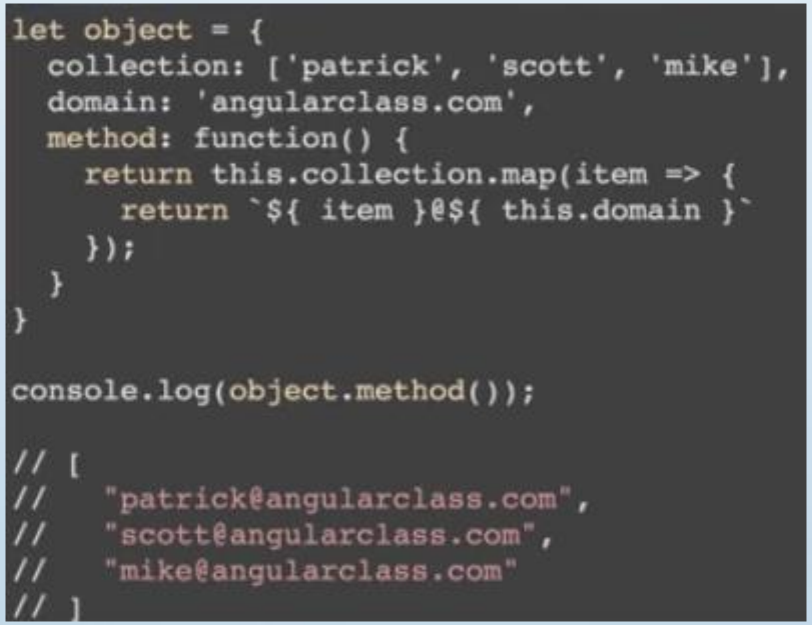


```
const str = 'hi';

if(true){
    str = 'bye';
}

console.log(str);

// error:
// "yolo" is read-only
```

ES2015 – Arrow Functions

New syntax for maintaining the parent object scope in callback functions.

ES2015

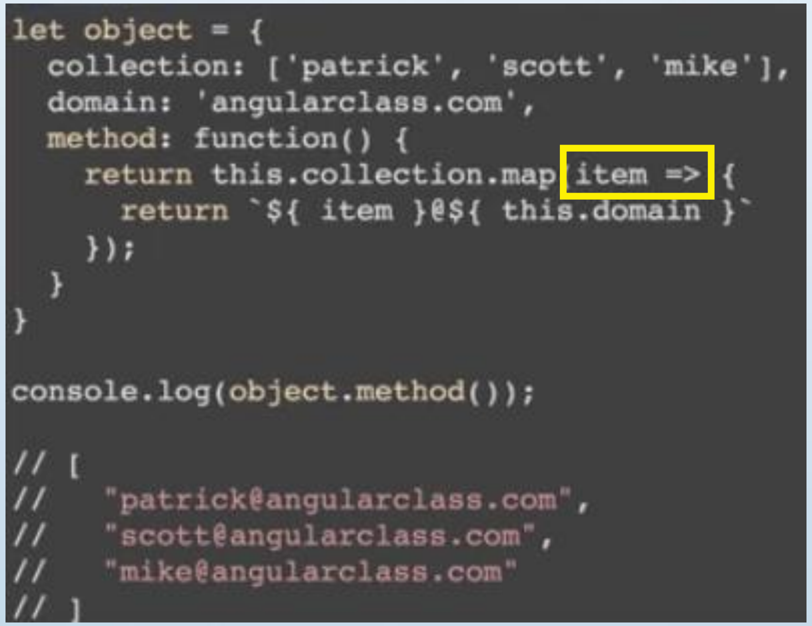


```
let object = {
  collection: ['patrick', 'scott', 'mike'],
  domain: 'angularclass.com',
  method: function() {
    return this.collection.map(item => {
      return `${ item }@${ this.domain }`
    });
  }
}

console.log(object.method());

// [
//   "patrick@angularclass.com",
//   "scott@angularclass.com",
//   "mike@angularclass.com"
// ]
```

ES2015 – Arrow Functions

New syntax for maintaining the parent object scope in callback functions.

ES2015

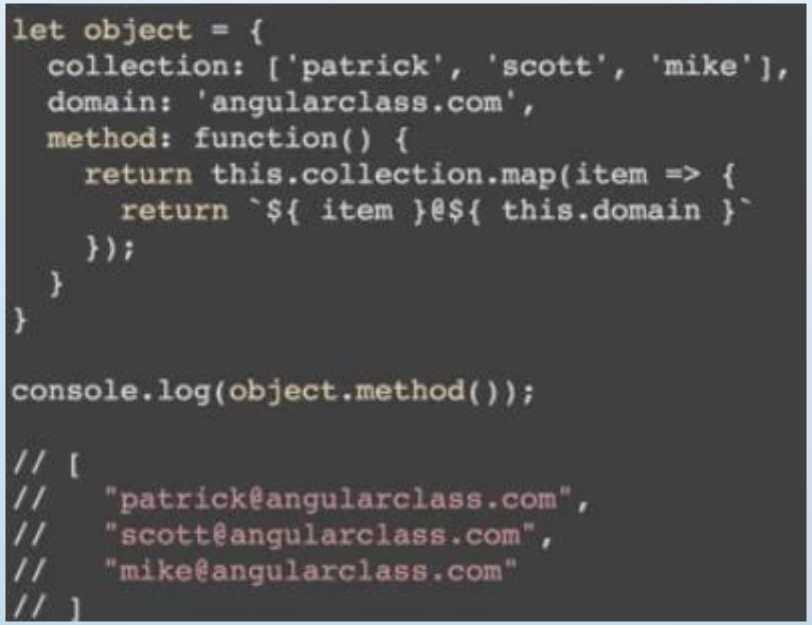


```
let object = {
  collection: ['patrick', 'scott', 'mike'],
  domain: 'angularclass.com',
  method: function() {
    return this.collection.map(item => {
      return `${ item }@${ this.domain }`
    });
  }
}

console.log(object.method());

// [
//   "patrick@angularclass.com",
//   "scott@angularclass.com",
//   "mike@angularclass.com"
// ]
```

ES2015 – Arrow Functions

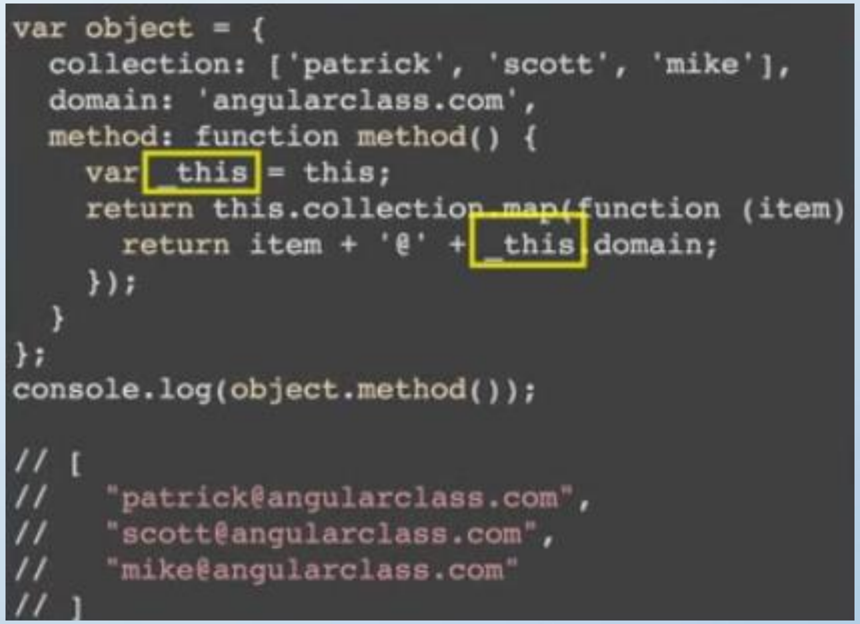
New syntax for maintaining the parent object scope in callback functions.

ES2015



```
let object = {
  collection: ['patrick', 'scott', 'mike'],
  domain: 'angularclass.com',
  method: function() {
    return this.collection.map(item => {
      return `${ item }@${ this.domain }`
    });
  }
}

console.log(object.method());

// [
//    "patrick@angularclass.com",
//    "scott@angularclass.com",
//    "mike@angularclass.com"
// ]
```

ES5 Output

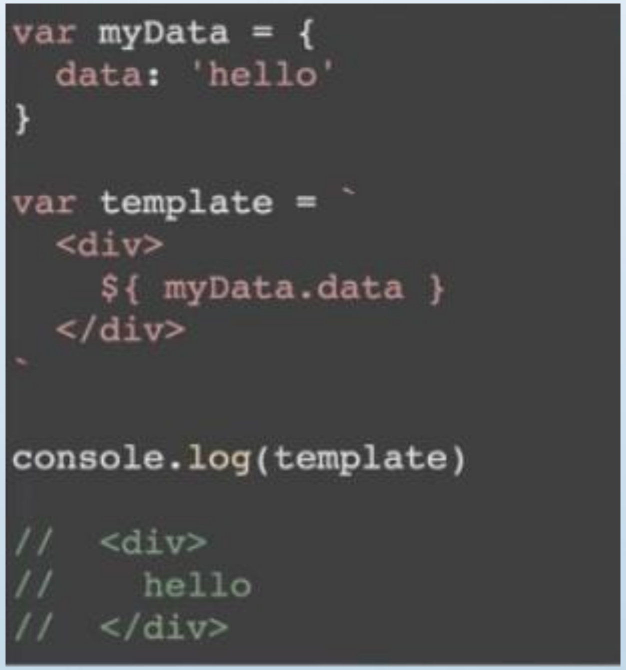


```
var object = {
  collection: ['patrick', 'scott', 'mike'],
  domain: 'angularclass.com',
  method: function method() {
    var _this = this;
    return this.collection.map(function (item)
      return item + '@' + _this.domain;
    });
  }
};
console.log(object.method());

// [
//    "patrick@angularclass.com",
//    "scott@angularclass.com",
//    "mike@angularclass.com"
// ]
```

ES2015 – Template Strings

Template Strings are using the back tick symbol for multi-line strings and string interpolation.

ES2015

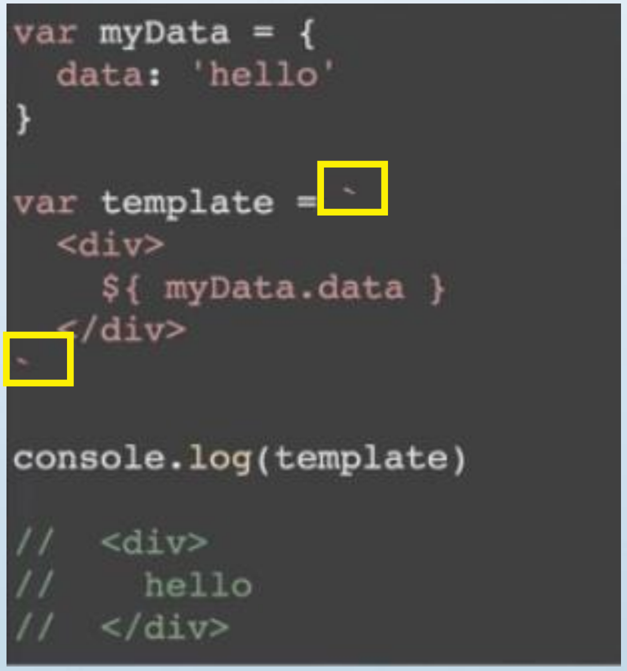


```
var myData = {
  data: 'hello'
}

var template = `
  <div>
    ${ myData.data }
  </div>
`

console.log(template)

//   <div>
//     hello
//   </div>
```

ES2015 – Template Strings

Template Strings are using the back tick symbol for multi-line strings and string interpolation.

ES2015

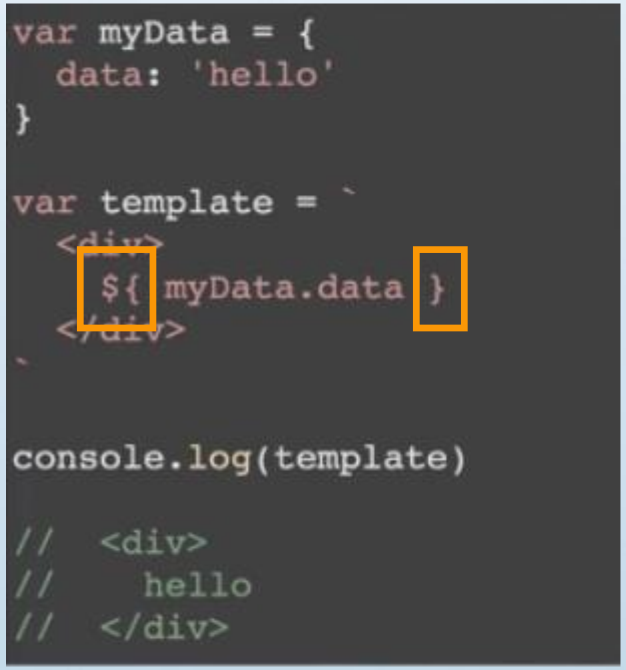


```
var myData = {
  data: 'hello'
}

var template = `
  <div>
    ${ myData.data }
  </div>
`

console.log(template)

//  <div>
//    hello
//  </div>
```

ES2015 – Template Strings

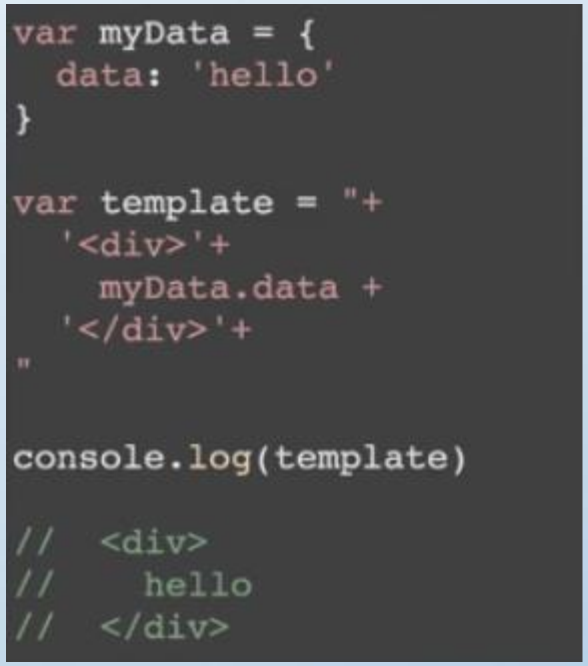
Template Strings are using the back tick symbol for multi-line strings and string interpolation.

ES2015



```
var myData = {
  data: 'hello'
}

var template = `
  <div>
    ${ myData.data }
  </div>
`

console.log(template)

//  <div>
//    hello
//  </div>
```

ES5 Output

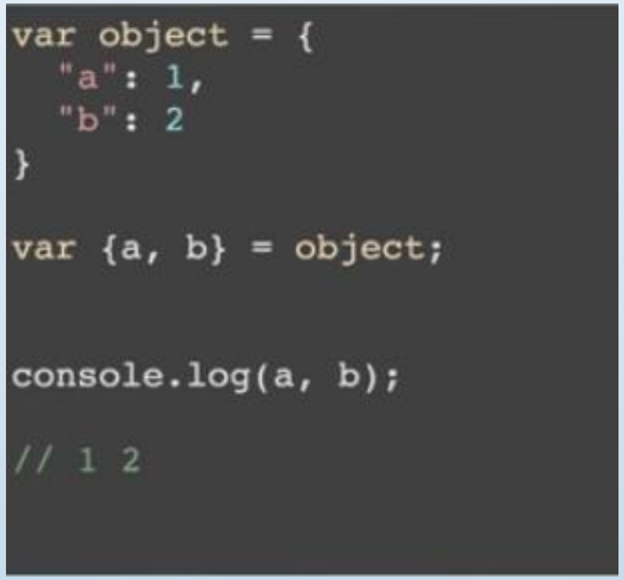


```
var myData = {
  data: 'hello'
}

var template = ''+
   '<div>'+
     myData.data +
   '</div>'+
''

console.log(template)

//  <div>
//    hello
//  </div>
```

ES2015 – Destructuring

Destructuring is a way to pluck properties off of a data structure and assign them to distinct variables.

ES6

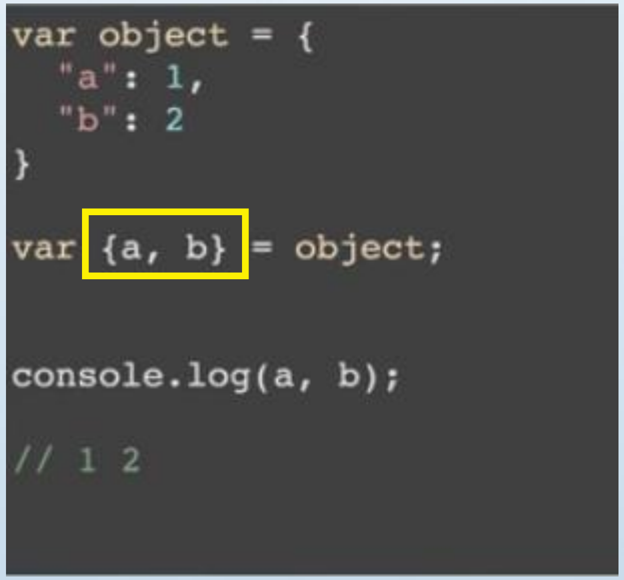


```
var object = {
  "a": 1,
  "b": 2
}

var {a, b} = object;


console.log(a, b);

// 1 2
```

ES2015 – Destructuring

Destructuring is a way to pluck properties off of a data structure and assign them to distinct variables.

ES6

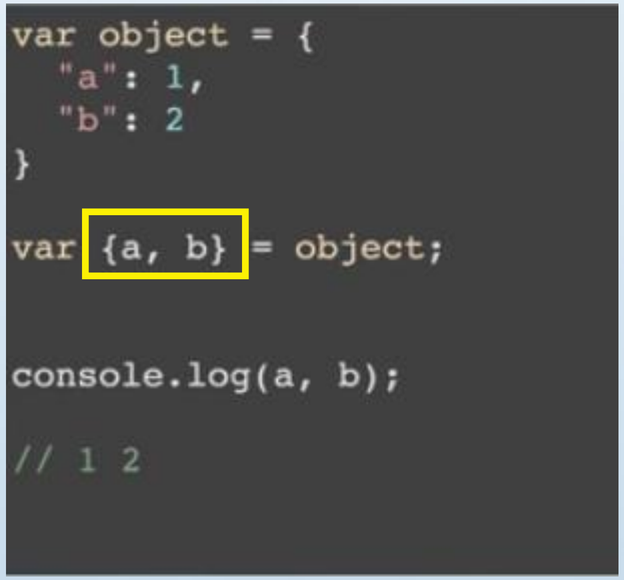


```
var object = {
  "a": 1,
  "b": 2
}

var {a, b} = object;

console.log(a, b);

// 1 2
```

ES2015 – Destructuring

Destructuring is a way to pluck properties off of a data structure and assign them to distinct variables

ES6

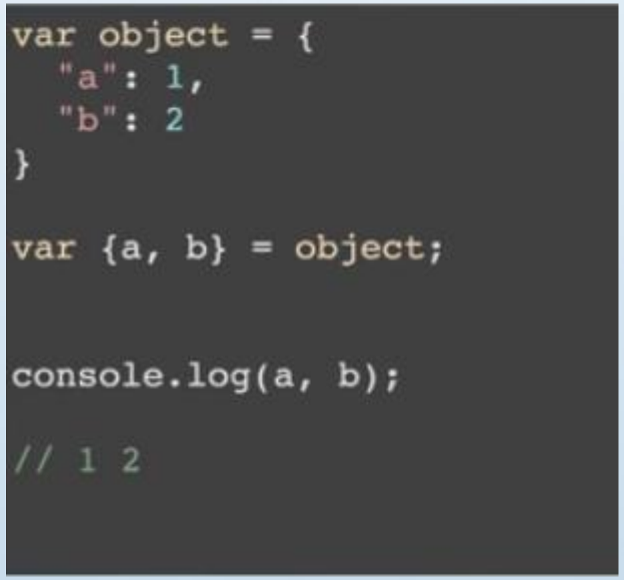


```
var object = {
  "a": 1,
  "b": 2
}

var {a, b} = object;


console.log(a, b);

// 1 2
```

ES2015 – Destructuring

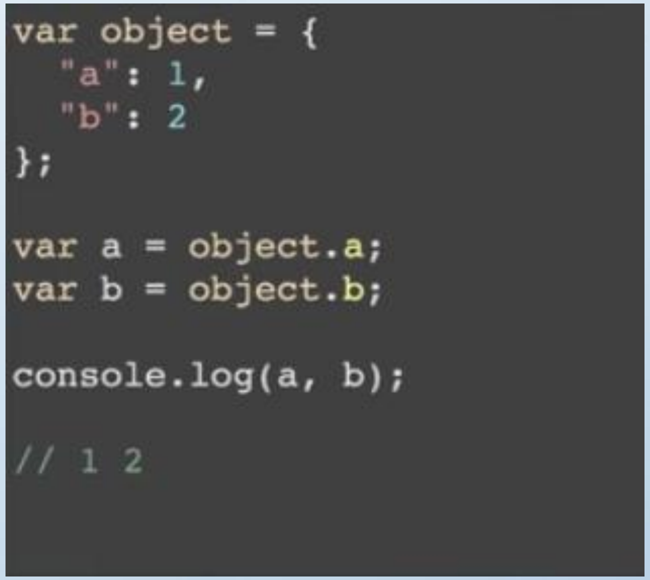
Destructuring is a way to pluck properties off of a data structure and assign them to distinct variables.

ES6



```
var object = {
  "a": 1,
  "b": 2
}

var {a, b} = object;


console.log(a, b);

// 1 2
```

ES5 Output

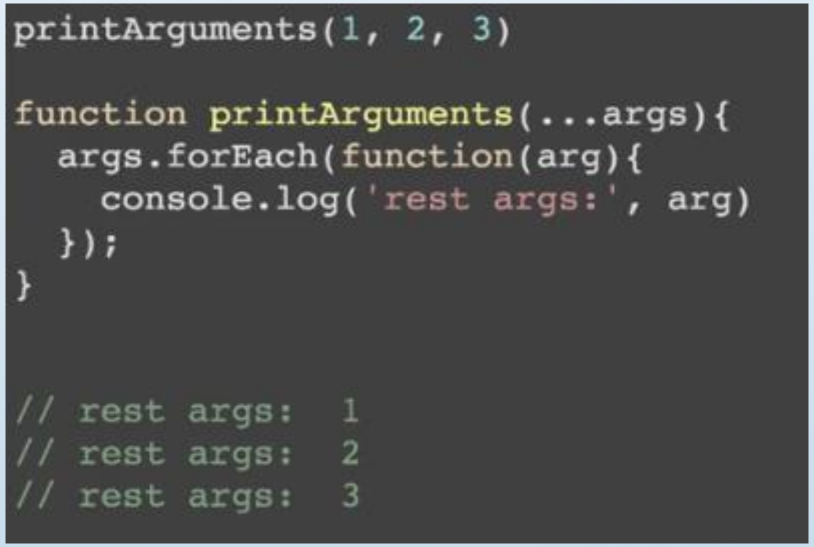


```
var object = {
  "a": 1,
  "b": 2
};

var a = object.a;
var b = object.b;

console.log(a, b);

// 1 2
```

ES2015 – Rest Parameters

If the last named argument is prefix with ... the argument collects itself and all consecutive arguments.

ES2015

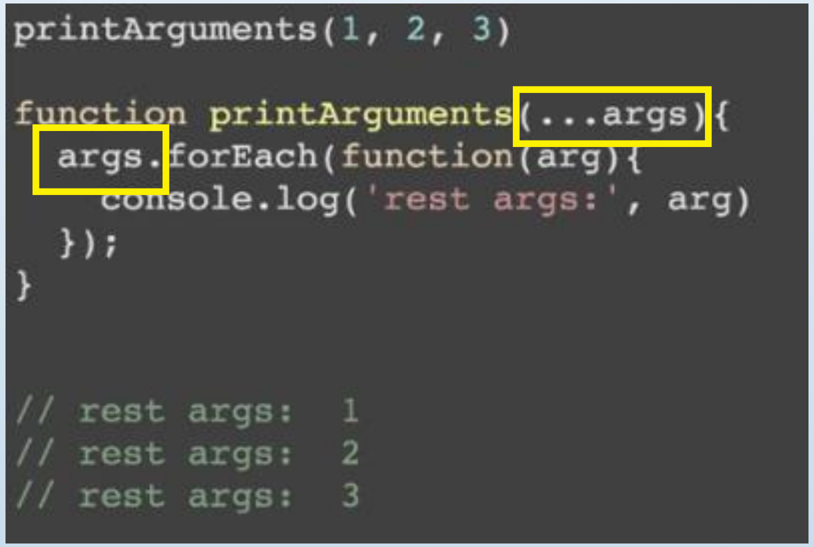


```
printArguments(1, 2, 3)

function printArguments(...args){
  args.forEach(function(arg){
    console.log('rest args:', arg)
  });
}


// rest args:  1
// rest args:  2
// rest args:  3
```

ES2015 – Rest Parameters

If the last named argument is prefix with ... the argument collects itself and all consecutive arguments.

ES2015

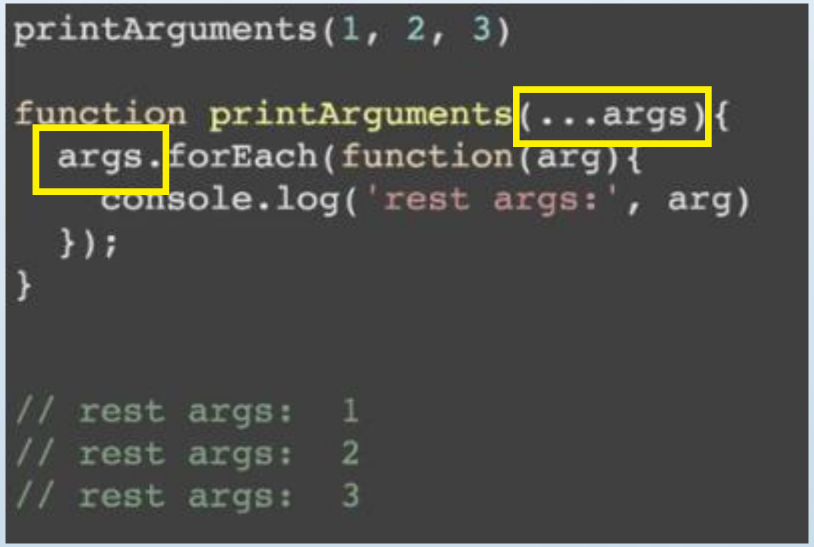


```
printArguments(1, 2, 3)

function printArguments(...args){
  args.forEach(function(arg){
    console.log('rest args:', arg)
  });
}


// rest args:  1
// rest args:  2
// rest args:  3
```

ES2015 – Rest Parameters

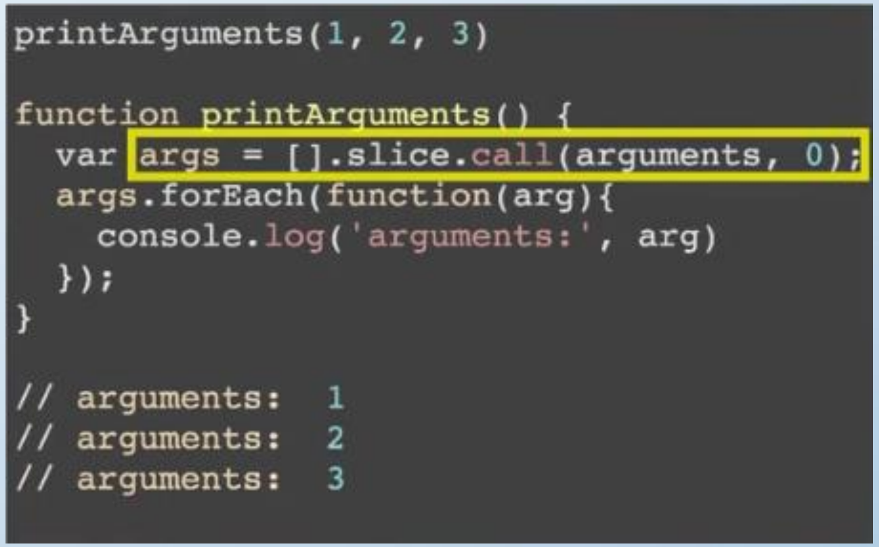
If the last named argument is prefix with … the argument collects itself and all consecutive arguments.

ES2015



```
printArguments(1, 2, 3)

function printArguments(...args){
  args.forEach(function(arg){
    console.log('rest args:', arg)
  });
}


// rest args:  1
// rest args:  2
// rest args:  3
```

ES5 Output

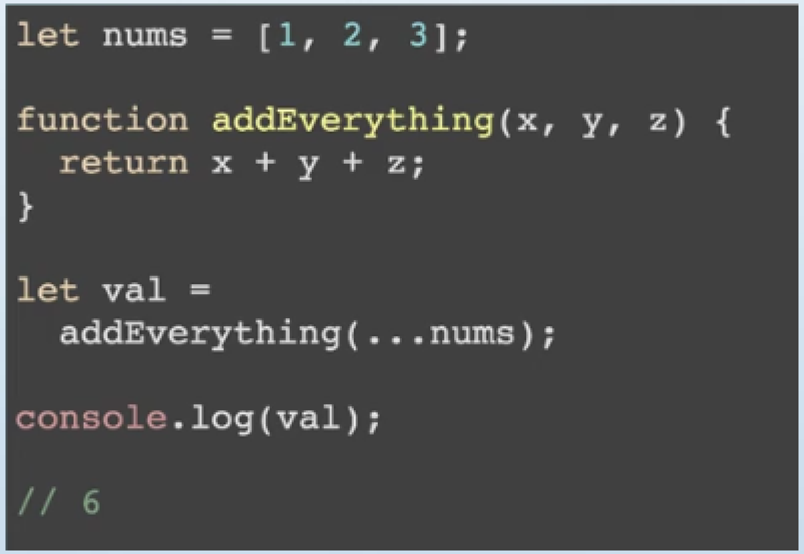


```
printArguments(1, 2, 3)

function printArguments() {
  var args = [].slice.call(arguments, 0);
  args.forEach(function(arg){
    console.log('arguments:', arg)
  });
}

// arguments:  1
// arguments:  2
// arguments:  3
```

ES2015 – Spread Operator

Spread Operators are conceptually the opposite of rest parameters. Enables dynamic expansion of an expression.

ES6

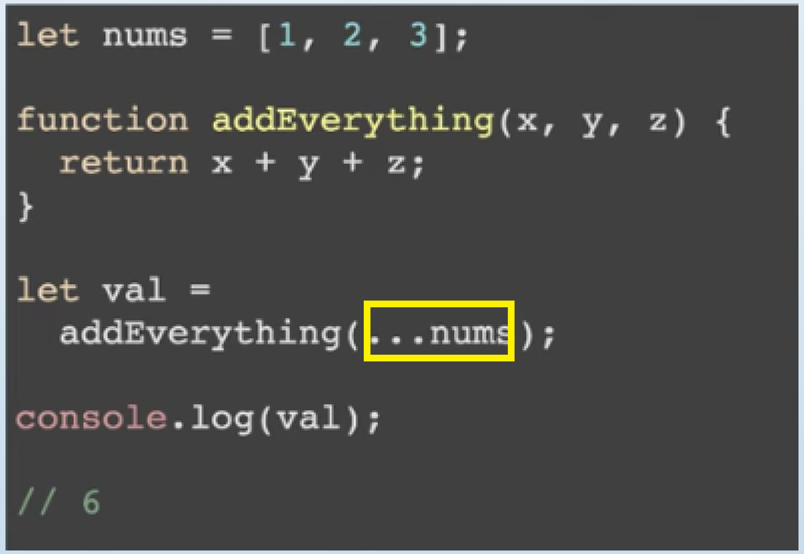


```
let nums = [1, 2, 3];

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

let val =
  addEverything(...nums);

console.log(val);

// 6
```

ES2015 – Spread Operator

Spread Operators are conceptually the opposite of rest parameters. Enables dynamic expansion of an expression.

ES6

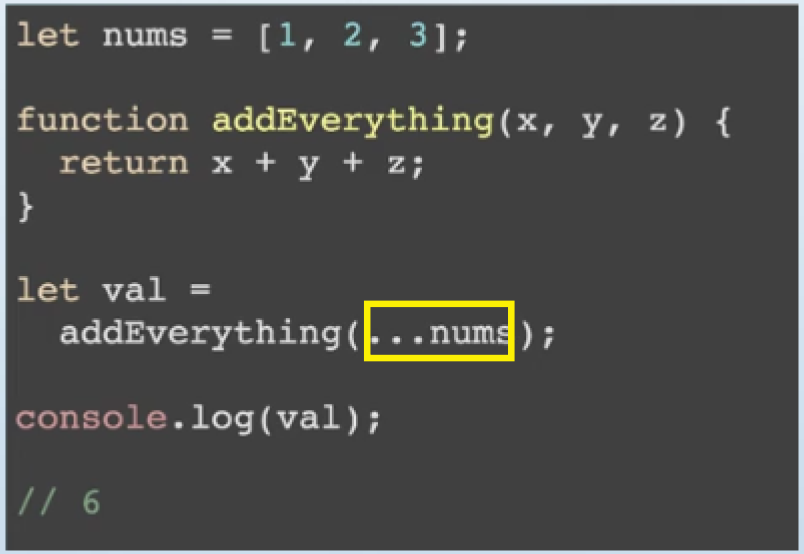


```
let nums = [1, 2, 3];

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

let val =
  addEverything(...nums);

console.log(val);

// 6
```

ES2015 – Spread Operator

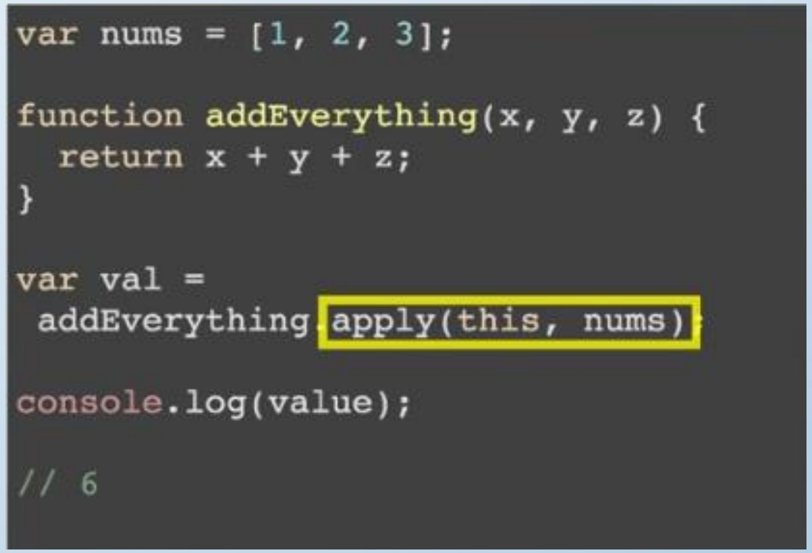
Spread Operators are conceptually the opposite of rest parameters. Enables dynamic expansion of an expression.

ES6



```
let nums = [1, 2, 3];

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

let val =
  addEverything(...nums);

console.log(val);

// 6
```

ES5 Output

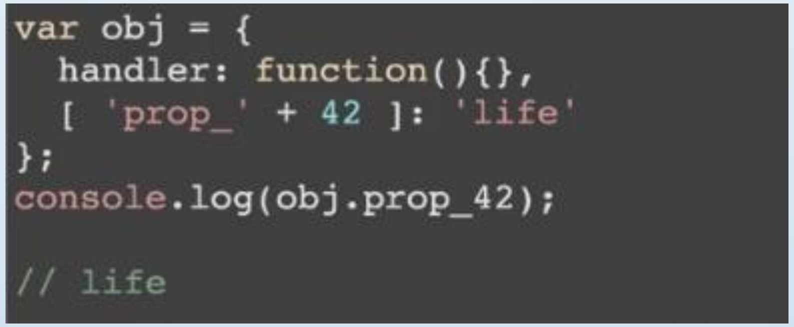


```
var nums = [1, 2, 3];

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

var val =
  addEverything.apply(this, nums);

console.log(value);

// 6
```

ES2015 – Enhanced Object Literals

Syntactial sugar for dynamic property generation in object literals.

ES2015

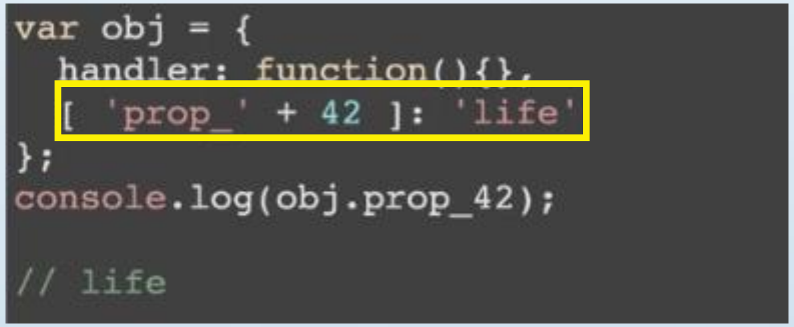


```
var obj = {
  handler: function(){},
  [ 'prop_' + 42 ]: 'life'
};
console.log(obj.prop_42);

// life
```

ES2015 – Enhanced Object Literals

Syntactial sugar for dynamic property generation in object literals.

ES2015

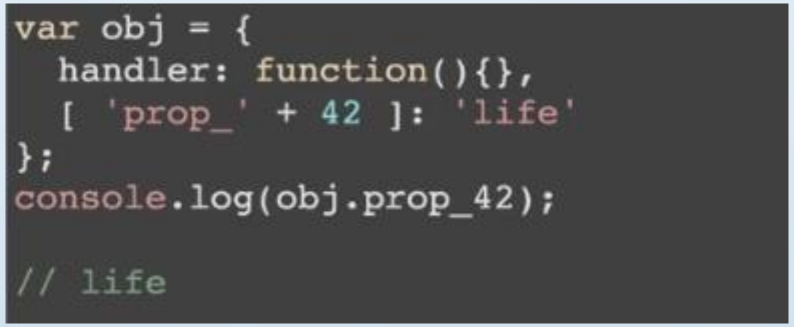


```
var obj = {
  handler: function(){},
  [ 'prop_' + 42 ]: 'life'
};
console.log(obj.prop_42);

// life
```

ES2015 – Enhanced Object Literals

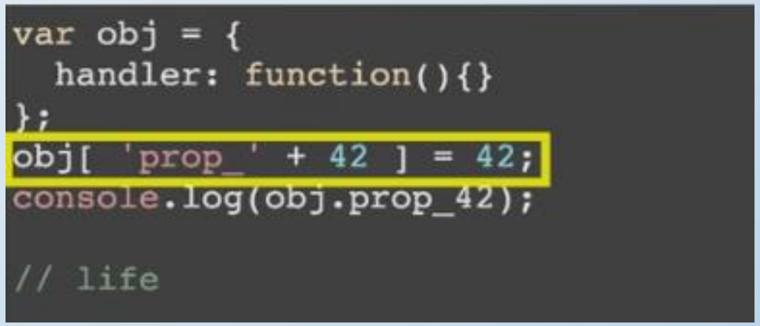
Syntactial sugar for dynamic property generation in object literals.

ES2015



```
var obj = {
  handler: function(){},
  [ 'prop_' + 42 ]: 'life'
};
console.log(obj.prop_42);

// life
```

ES5 Output

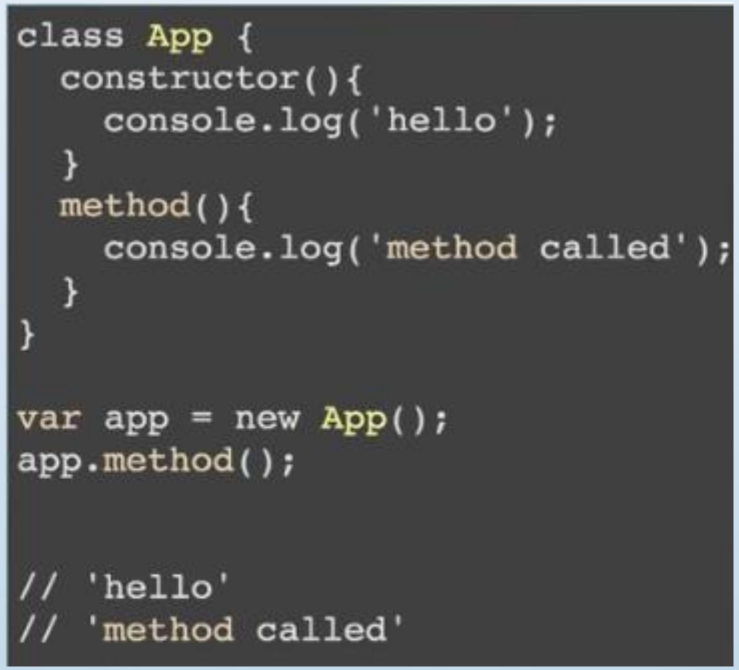


```
var obj = {
  handler: function(){}
};
obj[ 'prop_' + 42 ] = 42;
console.log(obj.prop_42);

// life
```

ES2015 – Classes

Syntactial sugar over Javascript's existing prototype-based inheritance.

ES2015

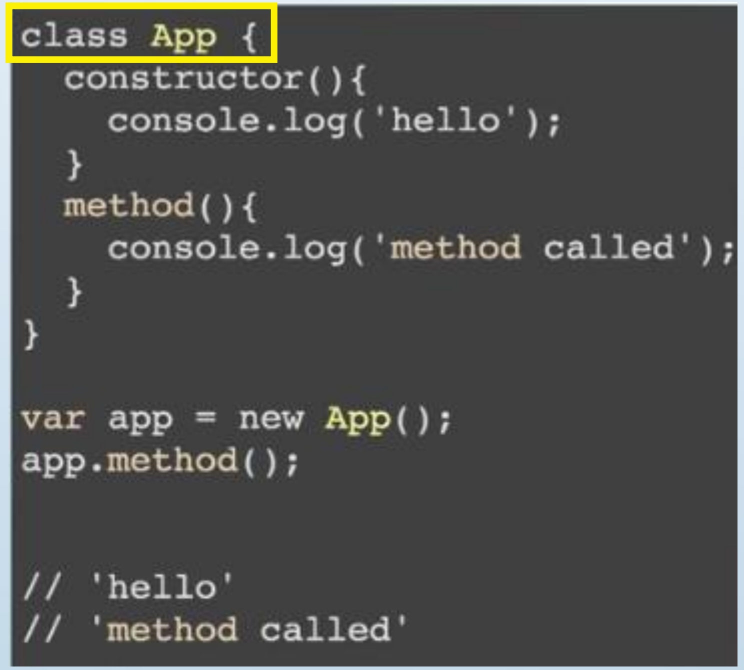


```
class App {
  constructor(){
    console.log('hello');
  }
  method(){
    console.log('method called');
  }
}

var app = new App();
app.method();


// 'hello'
// 'method called'
```

ES2015 – Classes

Syntactial sugar over Javascript’s existing prototype-based inheritance.

ES2015

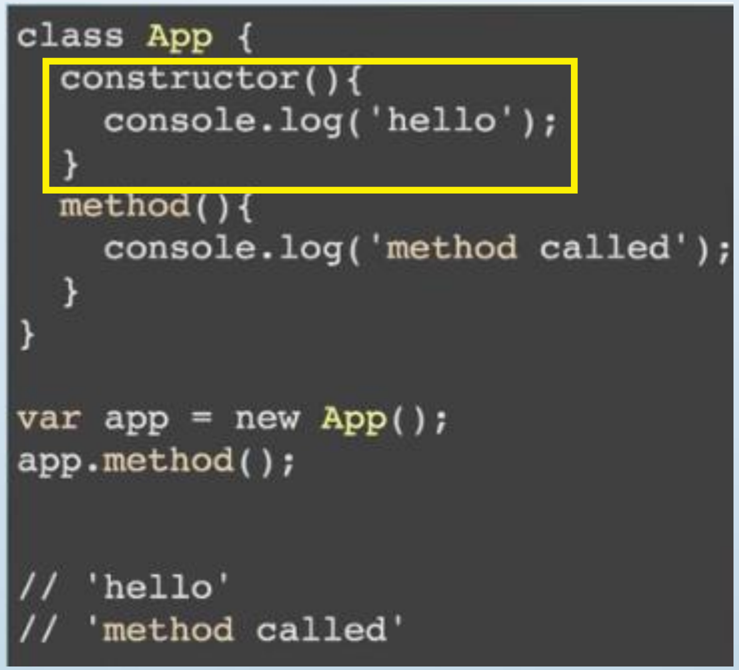


```
class App {
  constructor(){
    console.log('hello');
  }
  method(){
    console.log('method called');
  }
}

var app = new App();
app.method();


// 'hello'
// 'method called'
```

ES2015 – Classes

Syntactial sugar over Javascript's existing prototype-based inheritance.

ES2015

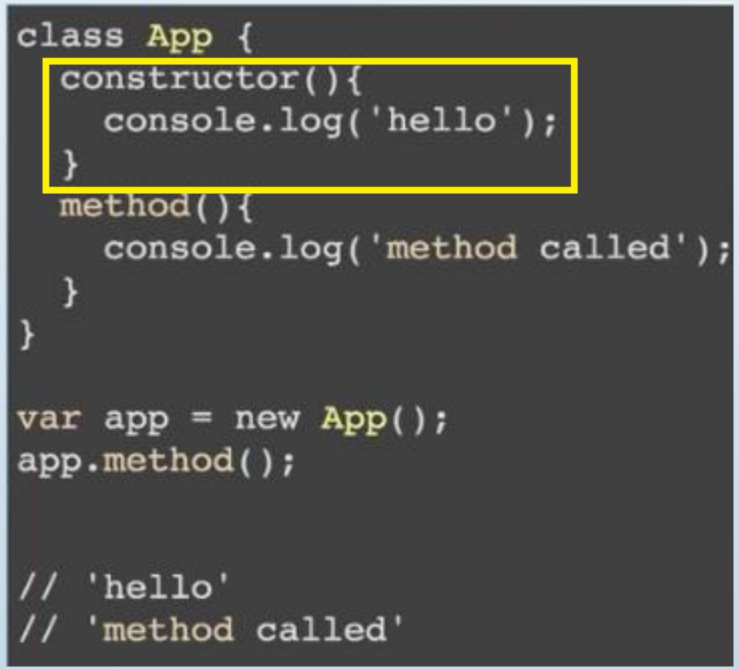


```
class App {
  constructor(){
    console.log('hello');
  }
  method(){
    console.log('method called');
  }
}

var app = new App();
app.method();


// 'hello'
// 'method called'
```

ES2015 – Classes

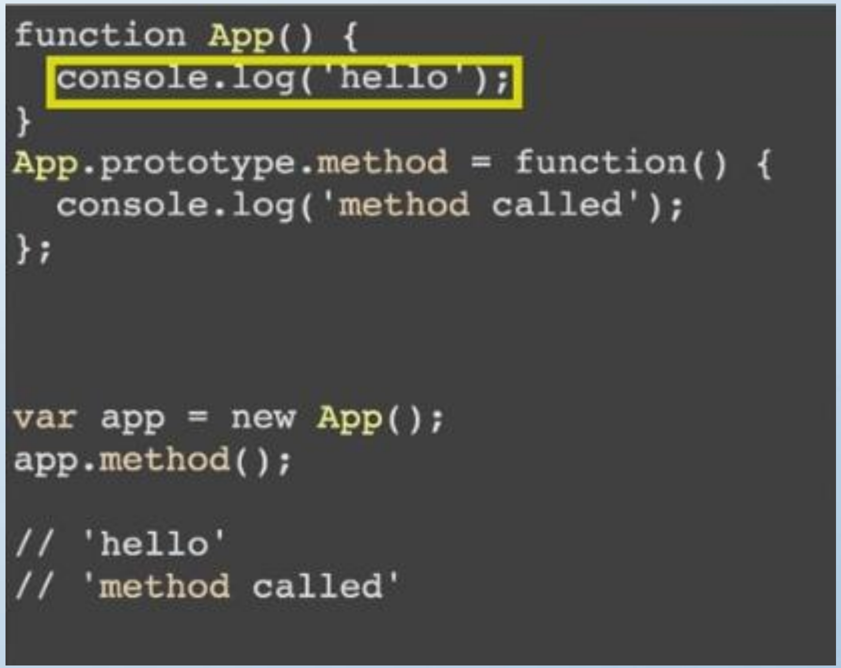
Syntactial sugar over Javascript's existing prototype-based inheritance.

ES2015



```
class App {
  constructor(){
    console.log('hello');
  }
  method(){
    console.log('method called');
  }
}

var app = new App();
app.method();


// 'hello'
// 'method called'
```

ES5 Output

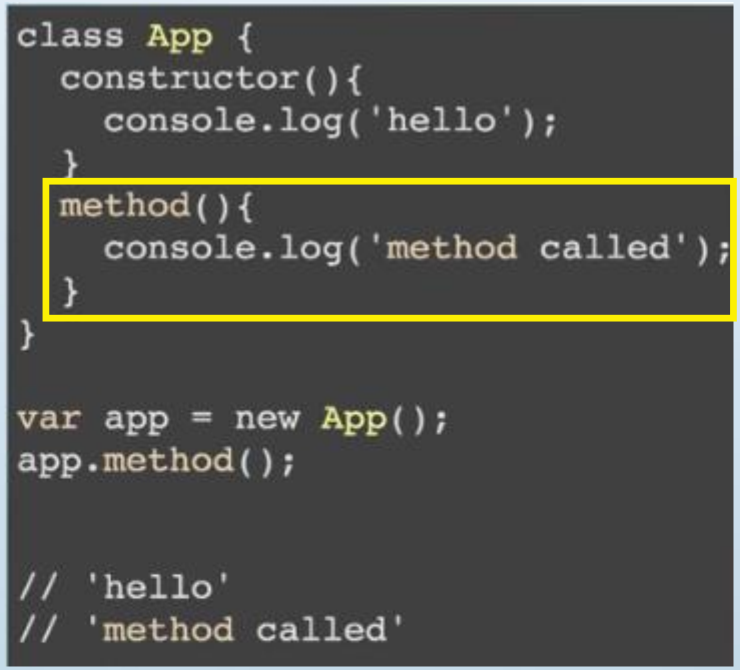


```
function App() {
  console.log('hello');
}
App.prototype.method = function() {
  console.log('method called');
};



var app = new App();
app.method();

// 'hello'
// 'method called'
```

ES2015 – Classes

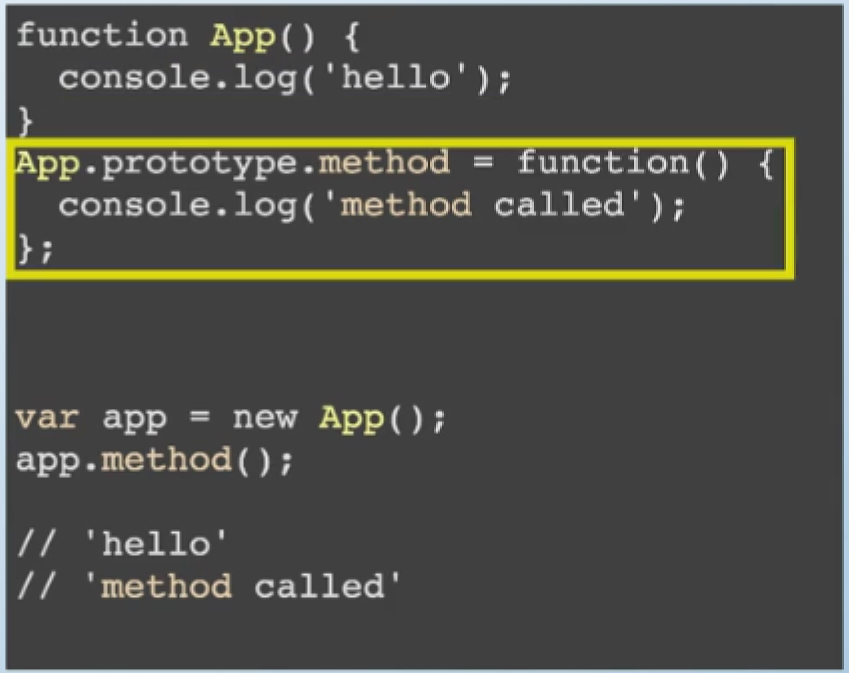
Syntactial sugar over Javascript's existing prototype-based inheritance.

ES2015



```
class App {
  constructor(){
    console.log('hello');
  }
  method(){
    console.log('method called');
  }
}

var app = new App();
app.method();


// 'hello'
// 'method called'
```

ES5 Output



```
function App() {
  console.log('hello');
}
App.prototype.method = function() {
  console.log('method called');
};


var app = new App();
app.method();

// 'hello'
// 'method called'
```

ES2015 – Modules

Modules allow code sharing between javascript files.

// require files

```
import something from 'framework';
```

```
import * as something from 'framework';
```

```
import {matchedProp} from 'framework';
```

// expose values

```
export default function something {}
```

```
export var value = 'value';
export var another = 'value2';
```

```
export var matchedProp = 'someValue';
```

THE END