JS

ES6

ECMAScript 6 (official name is ECMAScript 2015)

ts thruskills

https://www.thruskills.com

+91 831 737 5392

What you will learn

- ✓ Introduction to ES6
- ✓ Core Features of ES6

Goals for ES6

1. Be a better language

Be a better language for writing:

- complex applications
- libraries (possibly including the DOM) shared by those applications
- code generators targeting the new edition

Goals for ES6...

2. Improve interoperation

Improve interoperation, adopting de facto standards where possible.

- Classes: are based on how constructor functions are currently used.
- Modules: picked up design ideas from the CommonJS module format.
- Arrow functions: have syntax that is borrowed from CoffeeScript.

Goals for ES6...

3. Versioning

Keep versioning as simple and linear as possible.

- ES6 avoids versioning via "One JavaScript": In an ES6 code base, everything is ES6, there are no parts that are ES5-specific.

```
<script type="application/ecmascript;version=6">
</script>
```

Categories of ES6 features

There are three major categories of features:

1. **Better syntax for features** that already exist (e.g. via libraries). For example: Classes, Modules

2. **New functionality** in the standard library. For example: Promises, Maps, Sets

3. **Completely new features**. For example: Generators, Proxies, WeakMaps

Core ES6 features

1. `var` to `const/let`

In ES5, you declare variables via var. Such variables are function-scoped, their scopes are the innermost enclosing functions. The behavior of var is occasionally confusing. This is an example:

Core ES6 features: `var` to `const`/`let`

```
var x = 3;
function func(randomize) {
    if (randomize) {
        var x = Math.random(); // (A) scope: whole function
        return x;
    }
    return x; // accesses the x from line A
}
func(false); // undefined
```

```
let x = 3;
function func(randomize) {
    if (randomize) {
        let x = Math.random();
        return x;
    }
    return x;
}
func(false); // 3
```

Core ES6 features: `var` to `const/let`

That means that you can't blindly replace **var** with **let** or **const** in existing code; you have to be careful during refactoring.

- **Prefer const.** You can use it for all variables whose values never change.
- **Otherwise, use let** – for variables whose values do change.
- Avoid **var.**

Core ES6 features

2. `IIFEs to blocks`

From 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 = 'This is IIFE Block';
    console.log(tmp);
}());  // close IIFE
console.log(tmp); // ReferenceError
```

Core ES6 features: `IIFEs to blocks`

In ECMAScript 6, you can simply use a block and a let declaration (or a const declaration):

```
{  // open block
    var tmp = 'This is IIFE Block';
    console.log(tmp);
}  // close block
console.log(tmp); // ReferenceError
```

Core ES6 features

3. `concatenating strings to template literals`

With ES6, JavaScript finally gets literals for string interpolation and multi-line strings.

```
(function () {  // open IIFE
    var tmp = 'This is IIFE Block';
    console.log(tmp);
}());  // close IIFE
console.log(tmp); // ReferenceError
```

Core ES6 features: string interpolation

In ES6 you can use string interpolation via template literals:

```
function printCoord(x, y) {
    console.log(`(${x}, ${y})`);
}
printCoord(10,20);
```

Core ES6 features: Multi-line strings

Template literals also help with representing multi-line strings. For example, this is what you have to do to represent one in ES5:

```
var HTML5_SKELETON =
    '<!doctype html>\n' +
    '<html>\n' +
    '<head>\n' +
    '    <meta charset="UTF-8">\n' +
    '    <title></title>\n' +
    '</head>\n' +
    '<body>\n' +
    '</body>\n' +
    '</html>\n';

console.log(HTML5_SKELETON);
```

Core ES6 features: Template literals

ES6 template literals can span multiple lines:

(The examples differ in how much whitespace is included, but that doesn’t matter in this case.)

```
const HTML5_SKELETON = `
    <!doctype html>
    <html>
    <head>
        <meta charset="UTF-8">
        <title></title>
    </head>
    <body>
    </body>
    </html>`;

console.log(HTML5_SKELETON);
```

Core ES6 features

4. `function expressions to arrow functions`

In current ES5 code, you have to be careful with this whenever you are using function expressions. In the following example, I create the helper variable _this (line A) so that the this of UiComponent can be accessed in line B.

Core ES6 features: `arrow functions`

```
function UiComponent() {
    var _this = this; // (A)
    var button = document.getElementById('myButton');
    button.addEventListener('click', function () {
        console.log('CLICK');
        _this.handleClick(); // (B)
    });
}
UiComponent.prototype.handleClick = function () {
    ...
};
```

```
function UiComponent() {
    var button = document.getElementById('myButton');
    button.addEventListener('click', () => {
        console.log('CLICK');
        this.handleClick(); // (A)
    });
}
```

Core ES6 features: `arrow functions`

Arrow functions are especially handy for short callbacks that only return results of expressions.

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);
```

When defining parameters, you can even omit parentheses if the parameters are just a single identifier. Thus: (x) => x * x and x => x * x are both allowed.

Core ES6 features

5. Handling multiple return values

Some functions or methods return multiple values via arrays or objects. In ES5, you always need to create intermediate variables if you want to access those values.

```
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];
```

Core ES6 features: `multiple return`

In ES6, destructuring makes this code simpler:

```
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.

Core ES6 features: `multiple return`

`Multiple return values via objects`

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

In ES5, even if you are only interested in the properties of an object, you still need an intermediate variable (propDesc in the example below):

Core ES6 features: `multiple return`

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, you can use destructuring:

```
const obj = { foo: 123 };

const {writable, configurable} =
    Object.getOwnPropertyDescriptor(obj, 'foo');

console.log(writable, configurable); // true true
```

Core ES6 features

6. `for to forEach() to for-of`

Prior to ES5, you iterated over Arrays as follows:

```
var arr = ['a', 'b', 'c'];
for (var i=0; i<arr.length; i++) {
    var elem = arr[i];
    console.log(elem);
}
```

In ES5, you have the option of using the Array method **forEach**():

```
var arr = ['a', 'b', 'c'];
arr.forEach(function (elem) {
    console.log(elem);
});
```

Core ES6 features: `for-of loop`

A **for** loop has the advantage that you can break from it, **forEach**() has the advantage of conciseness.

In ES6, the for-of loop combines both advantages:

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

```
for (const [index, elem] of array.entries()) {
    console.log(index+'. '+elem);
}
```

If you want both index and value of each array element, for-of has got you covered, too, via the new Array method entries() and destructuring:

Core ES6 features: `default values`

7. **`Handling parameter default values`**

In ES5, you specify default values for parameters like this:

```
function foo(x, y) {
    x = x || 0; y = y || 0;
    console.log(x,y);
}
foo();
```

ES6 has nicer syntax:

```
function foo(x=0, y=0) {
    console.log(x,y);
}
foo();
```

Core ES6 features: `named parameters`

8. **`Handling named parameters`**

A common way of naming parameters in JavaScript is via object literals (the so-called options object pattern):

```
selectEntries({ start: 0, end: -1 });
```

Two advantages of this approach are: Code becomes more self-descriptive and it is easier to omit arbitrary parameters.

Core ES6 features: `named parameters`

In ES5, you can implement selectEntries() as follows:

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

In ES6, you can use destructuring in parameter definitions and the code becomes simpler:

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

Core ES6 features: `optional params`

9. `Making the parameter optional`

To make the parameter options optional in ES5, you’d add line A to the code:

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

Core ES6 features: `optional params`

In ES6, you can declare a rest parameter (args in the example below) via the ... operator:

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

Core ES6 features

10. `arguments to rest parameters`

In ES5, if you want a function (or method) to accept an arbitrary number of arguments, you must use the special variable arguments:

```
function logAllArguments() {
    for (var i=0; i < arguments.length; i++) {
        console.log(arguments[i]);
    }
}
logAllArguments('abcd', 123);
```

Core ES6 features: `rest parameter`

In ES6, you can declare a rest parameter (args in the example below) via the ... operator:

```
function logAllArguments(...args) {
    for (const arg of args) {
        console.log(arg);
    }
}
logAllArguments('abcd', 123);
```

Core ES6 features

12. `Array.prototype.push()`

ES5 – apply():

```
var arr1 = ['a', 'b'];
var arr2 = ['c', 'd'];

arr1.push.apply(arr1, arr2);
    // arr1 is now ['a', 'b', 'c', 'd']
```

ES6 – spread operator:

```
const arr1 = ['a', 'b'];
const arr2 = ['c', 'd'];

arr1.push(...arr2);
    // arr1 is now ['a', 'b', 'c', 'd']
```

Core ES6 features: `rest parameter`

In ES6, you can declare a rest parameter (args in the example below) via the ... operator:

ES5 – apply():

```
Math.max.apply(Math, [-1, 5, 11, 3])
```

ES6 – spread operator:

```
Math.max(...[-1, 5, 11, 3])
```

Core ES6 features

13. `function expressions to method definitions`

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

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

Core ES6 features

ES6 has method definitions, special syntax for creating methods:

```
const obj = {
    foo() {
        console.log('foo')
    },
    bar() {
        this.foo();
    },
}
```

Core ES6 features

14. `constructors to classes`

ES6 classes are mostly just more convenient syntax for constructor functions.

Core ES6 features: base 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;
    }
}
```

Core ES6 features: derived classes

Subclassing is complicated in ES5, especially referring to super-constructors and super-properties. This is the canonical way of creating a sub-constructor Employee of Person:

```
function Employee(name, title) {
    Person.call(this, name); // super(name)
    this.title = title;
}
Employee.prototype = Object.create(Person.prototype);
Employee.prototype.constructor = Employee;
Employee.prototype.describe = function () {
    return Person.prototype.describe.call(this) //
super.describe()
           + ' (' + this.title + ')';
};
```

Core ES6 features: derived classes

ES6 has built-in support for subclassing, via the extends clause:

```
class Employee extends Person {
    constructor(name, title) {
        super(name);
        this.title = title;
    }
    describe() {
        return super.describe() + ' (' + this.title + ')';
    }
}
```

Core ES6 features

15. `custom error constructors to subclasses of Error`

In ES5, it is impossible to subclass the built-in constructor for exceptions, **Error**. The following code shows a work-around that gives the constructor **MyError** important features such as a stack trace:

```
class MyError extends Error {

}
```

Core ES6 features: Extending Error

This is how it would look like in ES5

```
function MyError() {
    // Use Error as a function
    var superInstance = Error.apply(null, arguments);
    copyOwnPropertiesFrom(this, superInstance);
}
MyError.prototype = Object.create(Error.prototype);
MyError.prototype.constructor = MyError;

function copyOwnPropertiesFrom(target, source) {
    Object.getOwnPropertyNames(source)
    .forEach(function(propKey) {
        var desc = Object.getOwnPropertyDescriptor(source,
propKey);
        Object.defineProperty(target, propKey, desc);
    });
    return target;
};
```

Core ES6 features

16. `objects to Maps`

Using the language construct object as a map from strings to arbitrary values (a data structure) has always been a makeshift solution in JavaScript. The safest way to do so is by creating an object whose prototype is null. Then you still have to ensure that no key is ever the string '__proto__', because that property key triggers special functionality in many JavaScript engines.

Core ES6 features

The following ES5 code contains the function countWords that uses the object dict as a map:

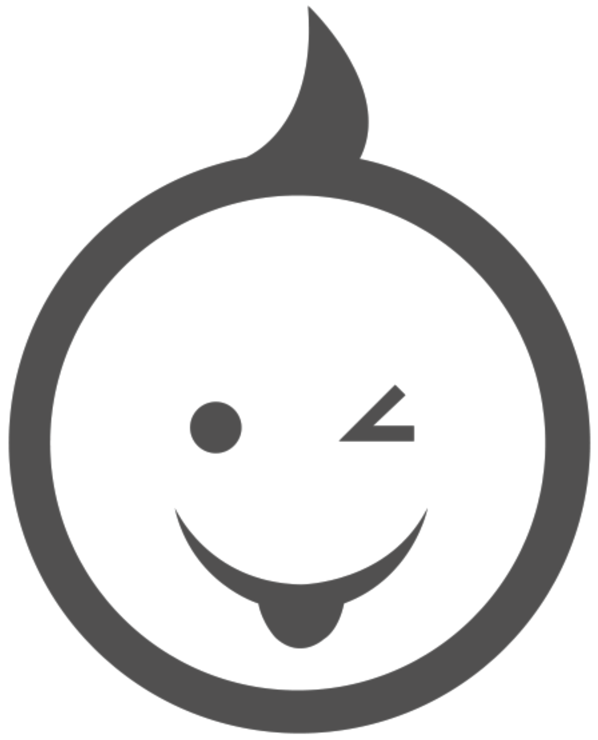
```
var dict = Object.create(null);
function countWords(word) {
    var escapedWord = escapeKey(word);
    if (escapedWord in dict) {
        dict[escapedWord]++;
    } else {
        dict[escapedWord] = 1;
    }
}
function escapeKey(key) {
    if (key.indexOf('__proto__') === 0) {
        return key+'%';
    } else {
        return key;
    }
}
```

Core ES6 features

In ES6, you can use the built-in data structure Map and don't have to escape keys. As a downside, incrementing values inside Maps is less convenient.

```
const map = new Map();
function countWords(word) {
    const count = map.get(word) || 0;
    map.set(word, count + 1);
}
```

Any Questions



Thank You

https://www.thruskills.com

+91 831 737 5392

ts thruskills