ES6

by

Anand Kulkarni

anand.kulkarni1@zensar.com

Table of Content

Module	Topic
Module 1:	Introduction to ES6
Module 2:	First ES6 application
Module 3:	ES6 constants & scoping
Module 4:	Enhanced object properties
Module 5:	Arrow Functions
Module 6:	Extended Parameter Handling & Template Literals
Module 7:	De-structuring assignment
Module 8:	Classes & Modules
Module 9:	Promises

Introduction to ES6

- ECMAScript is a specification for writing scripting language defined by European Computer Manufacturers Association (ECMA).
- Various scripting languages like JavaScript, ActionScript, Jscript etc. implement ECMAScript specifications. Thus, ECMAScript is a superset of JavaScript.
- ECMAScript's specification version 5 is called as ES5 & similarly specification version 6 is called as ES6 or ECMAScript 2015.

ECMAScript release history

Release	Year
ECMAScript 1	June 1997
ECMAScript 2	June 1998
ECMAScript 3	December 1999
ECMAScript 4	July 2008
ECMAScript 5	December 2009
ECMAScript 5.1	June 2011
ECMAScript 6	June 2015

ES6 features

- Added ‘const’ keyword to declare a constant & ‘let’ keyword to determine variable scope.
- Added several utility methods inside Math, Number, Array & String.
- Added ‘arrow functions’ similar to lambda expressions.
- Added ‘extended parameter handling’ similar to variable method arguments.
- Added module importing & exporting features.

ES6 features continue…

- Added object oriented concepts so that we can write a class, we can have inheritance, static methods, getter/setter methods etc.
- Added collection classes like Map & Set along with iteration facility.

Setup Environment

Install Node.js (https://nodejs.org/en/download/)

Install ‘Visual Studio Code’ (https://code.visualstudio.com/download)

Developing first ES6 application

Steps to create ES6 application

1.mkdir hello_app

2.cd hello_app

3.Create app.js

```
document.write('Hello from ES6!!');
console.log('ES6 app loaded');
```

4.Create index.html

```
<html>
  <body>
    <script src="bundle.js"></script>
  </body>
</html>
```

Steps to create ES6 application

5. Create package.json file using “npm init” command.
6. Add following dependencies into package.json

```
"devDependencies": {
 "webpack":"1.14.0",
 "babel-core":"6.21.0",
 "babel-loader":"6.2.10",
 "babel-preset-es2015":"6.18.0",
 "webpack-dev-server":"1.16.2“,
 "babel-polyfill": "",
  "typescript": "",
  "ts-loader": ""
}
```

7. Run *‘npm install’.* It will install all dependencies required to run ES6 application.

Steps to create ES6 application

8. Create webpack.config.js file. The webpack.config.js is a standard configuration file provided by webpack to put all of your configuration, loaders and other specific information relating to your build.

```
module.exports = {
 entry: "./app.js",
 output: {  filename: "bundle.js"  },
devServer: {  inline:true,  port: 8080  },
module: {  loaders: [ { … }]},
watch: true
}
```

Steps to create ES6 application

entry - name of the top level file or set of files that we want to include in our build, can be a single file or an array of files. In our build, we only pass in our main file (app.js).

output - an object containing your output configuration. In our build, we only specify the filename key (bundle.js) for the name of the file we want Webpack to build.

devServer - Server configuration like port number & auto-refresh the browser if code modified.

watch - It will auto build the ES6 code to ES5 if modified at runtime.

Steps to create ES6 application

9. Set the path for ‘webpack’ command.

   set PATH=%PATH%;./node_modules/.bin

10. Run the command *'webpack'* on console. It will convert your ES6 code into ES5 in the form of bundle.js.

11. Start webpack-dev-server:

    webpack-dev-server --inline

12. Find out on which port webpack-dev-server is running. Suppose it is 8080.

13. Finally, Run index.html inside browser:

    http://localhost:8080/index.html

ES6 features

Constants

ES6 allows to declare a constant whose value cannot be changed. For example:

```
const PI = 3.141593;

console.log(PI);

PI = 4.45; //Error
```

Scoping

In JavaScript, any variable that is declared in the program is raised up to the top execution context. For example:

```
var submit = function() {
        var x = "foo";
        if (x == "foo") {
                var y = "bar";
        }
        console.log(x);
        console.log(y);
}
submit();
```

Output:
foo
bar

Scoping continue…

ES6 introduces ‘let’ keyword that respects the scope of a variable. For example:

```
var submit = function() {
        var x = "foo";
        if (x == "foo") {
            let y = "bar";
        }
        console.log(x);
        console.log(y);
}
submit();
```

Output:
foo
Uncaught ReferenceError: y is not defined

Enhanced object properties

Creating object literals is made much easy in ES6 as compared to traditional JavaScript(ES5)

1.Computed Property Names:

ES6 provides support to create object literals where property name itself is a computed value.

```
var prop = "foo";
var o = { [prop]: "hey", ["b" + "ar"]: "there", };
console.log(o.foo);
console.log(o.bar);
```

Enhanced object properties

1.Method Properties:

A javascript object can have method as a value of any attribute & it is called as ‘method properties’.

ES5 code:

```
let myMath = {       add: function(a, b) { return a + b; },    subtract: function(a, b)
{ return a - b; }     }
```

ES6 code:

```
let myMath = {
      add(a, b) { return a + b; } ,
      subtract(a, b) { return a - b; } }
```

Object.assign()

The **object.assign()** method is used to copy property values from one or more source objects to a given target object. It will return the target object. Here is the syntax:

```
var copyObj = Object.assign(targetObj, sourceObj1, sourceObj2….)
var obj = { firstname: “John", lastname: “Doe" };
var copy = Object.assign({}, obj);
console.log(copy); //Object {firstname: “John", lastname: “Doe"}
```

Arrow Functions

- Arrows are a function shorthand using the **=>** syntax.
- They are syntactically similar to the fat arrow syntax in C#, Java, and CoffeeScript.
- Arrow functions support both expression bodies and statement block bodies that return the value of the expression.
- Unlike functions, arrows share the same lexical this as their surrounding code.

Arrow Functions as expression body

Expression bodies are a single line expression with the **=>** token and an implied return value.

```
let nos = [2, 4, 6, 8, 10];
JavaScript (ES5) code:
Let square_nos = nos.map(function(num) { return num * num; });
ES6 code:
let square_nos = nos.map(num => num * num); //Arrow function
console.log(square_nos); //[4, 16, 36, 64, 100]
```

Arrow Functions as statement body

Statement bodies are multiline statements that allow for more complex logic.

```
let fives = [];

let nums = [1, 2, 5, 15, 25, 32];

nums.forEach(v => {

if (v % 5 === 0)

fives.push(v);

});

console.log(fives); //[5, 15, 25]
```

Using '*this*' inside arrow function

ES6 allows to access 'this' inside arrow functions.

```
let matt = {
        name: "Matt",
        friends: [“Tom", “Jerry", “Ivan"],
        printFriends() {
                this.friends.forEach(f =>
                console.log(this.name + " knows " + f));
        }
}
matt.printFriends();
```

Output:

Matt knows Tom

Matt knows Jerry

Matt knows Ivan

Extended Parameter Handling

Extended parameter handling mechanism in ES6 provides us three major functionalities:

- Default parameter values and optional parameters
- Rest parameter
- Spread operator

Default parameter values and optional parameters

Default parameters allow your functions to have optional arguments.

```
let greet = (msg = 'hello', name = 'world') => {
                console.log(msg,name);
}
greet();
greet('hey');
```

Output:

hello world

hey world

Rest parameter

Rest parameter, indicated by three consecutive dot characters(…), allow your functions to have a variable number of arguments.

The rest parameter is an instance of Array, so all array methods work.

```
function foo(x, ...y) {
        console.log(y);
        // y is an Array
        return x * y.length;
}
console.log(foo(3, 'hello', true) === 6);
```

Output:

["hello", true]

true

Spread operator

The spread operator is like the reverse of rest parameters. It allows you to expand an array into multiple formal parameters.

```
function add(a, b) {
        return a + b;
}
let nums = [5, 4];
console.log(add(...nums));
```

Output: 9

```
let a = [2, 3, 4];
let b = [1, ...a, 5];
console.log(b);
```

Output: [1, 2, 3, 4, 5]

Template Literals

- Template literals are indicated by enclosing strings in backtick characters (``)

- Template literals are used to construct single line or multi-line strings.

```
`In JavaScript '\n' is a line-feed.`
`Now I can do multi-lines
with template literals.`
```

- Template literals provide ‘String interpolation’ facility which can be used to compose very powerful strings in a clean.

```
var fname = ‘Tom’;
var salary = 10000
var incentive = 2000
let message = `My name is ‘${fname}’ and I am having total salary ${salary + incentive}`;
console.log(message); //My name is ‘Tom’ & I am having total salary 12000
```

De-structuring Assignment

- The de-structuring assignment syntax is a JavaScript expression that makes it possible to extract data from arrays or objects.
- De-structuring can be applied at following places:
    1) Array matching
    2) Object matching
        I. Shorthand notation
        II. Deep matching
        III. Parameter context
    3) Fail-soft de-structuring

Array Matching using de-structuring assignment

Array matching is used to pull the required values from an array into stand-alone variables.

```
let [a, , b] = [ 11, 24, 92 ];  //Array de-structuring
console.log("a:", a, "b:", b);
```

Output:

a: 11 b: 92

Object Matching using de-structuring assignment

- Like array matching, object matching allows us to pull the required properties of an object into stand-along variables.
- There are three ways to apply object matching-
    I. Shorthand notation
    II. Deep matching
    III. Parameter context

Object Matching using Shorthand notation

Shorthand notation allows us to grab properties from an object & create new variables out of it.

```
let {id, title} = {id: 546, title: 'Fruit Delivery', price: 5200.85};
    //Note, stand-alone variable name & object property name should match.
console.log("Id:", id, "Title:", title);
```

Output:

Id: 546 Title: Fruit Delivery

Object Deep Matching

Sometimes our object is more complex & contains nested properties. Data from such complex objects can be retrieved using deep matching.

```
let cust = {
    name: “Microsoft Corp.",
    address: {
        street: “J. M. Road",
        city: “Pune",
        state: “Maharashtra",
        zip: “411002"
    } };
let {address: {city, state}} = cust; //Deep matching
console.log("City:", city, "State:", state);
```

Output:

City: Pune State: Maharashtra

Object matching using Parameter Context

Array matching & object matching can be applied towards function parameters.

```
function processArray([ name, val ]) {
        console.log(name, val);
}
function processObject({ name: n, val: v }) {
        console.log(n, v);
}
function processObject_2({ name, val }) {
        console.log(name, val);
}
processArray([ "bar", 42 ]);
processObject({ name: "foo", val: 7 });
processObject_2({ name: "bar", val: 42 });
```

Output: bar 42 foo 7 bar 42

Fail-soft de-structuring

Fail soft de-structuring allows us to retrieve required values from array or object.

However, if value is not present then we can provide default value of a variable.

```
let list = [ 7, 42 ];
let [a = 1, b = 2, c = 3, d] = list; //Fail-soft de-structuring with default values.
console.log("a:", a, "\nb:", b, "\nc:", c, "\nd:", d);
```

Output:

```
a: 7
b: 42
c: 3
d: undefined
```

Modules

Modules provide support for exporting and importing values without polluting the global namespace.

Exporting a module (arith.js)

```
export function sum(x, y) {
return x + y;
}
export var pi = 3.141593;
```

Importing a module (app.js)

```
import {sum, pi} from './arith';
console.log('2 pi = ' + sum(pi, pi));
```

Module export/import with alias

Export with alias:

```
//arith.js
function sum(x, y) {
                return x + y;
}
let pi = 3.141593;
export {sum as add, pi}
```

```
//app.js
import {add, pi} from './arith';
console.log('2 pi = ' + add(pi, pi));
```

Import with alias:

```
//app.js
import {add as plus, pi} from './arith';
console.log('2 pi = ' + plus(pi, pi));
```

Default export

Modules exporting single values are sometimes used in ES6. Such modules can be exported with default option. For example:

```
//arith.js
export default function sum(x, y) {   return x + y;   }
export function divide(x, y) {         return x / y;   }
//app.js
import sum from './arith'; //Note that default modules are imported without curly brackets.
import { divide } from './arith';
```

Module import with wildcard (*)

You can import all exported components into one line using wildcard (*). Suppose arith.js exports sum() & divide() functions then you can import them using wildcard as follows:

```
//app.js
import * as arithOpr from './arith';
document.write('sum = ' + arithOpr.sum(20, 50));
document.write('divide = ' + arithOpr.divide(20, 5));
```

Classes

ES6 provides support for writing classes.

```
class Animal {
        constructor(name) {
                this.name = name;
        }
        greeting(sound) {
                return `A ${this.name} ${sound}`;
        }
        static echo(msg) {
                console.log(msg);
        }
}
let  animal = new Animal("Dog");
console.log(animal.greeting("barks")); //A Dog barks
Animal.echo("roof, roof"); //roof, roof
```

Class Inheritance

```
class Dog extends Animal {
    constructor() {
                super(“Dog");
    }
    static echo() {
                super.echo("bow wow"); //super can be used for static methods as well
    }
}
```

Class with getters & setters

```
export class Animal {
        constructor(name) {
                this.name = name;
        }
        get name() {
                return this._name;
        }
        set name(value) {
                this._name = value;
        }
}
```

Promises

- Promises provide a standard implementation of handling asynchronous programming in JavaScript without using callbacks.
- A promise represents a value that we can handle at some point in the future.
- A promise contract is immutable.

Working with Promises

```
var p2 = Promise.resolve("foo");
p2.then((res) => console.log(res));  //Output: foo

var p = new Promise(function(resolve, reject) {
setTimeout(() => resolve(4), 2000);
});

p.then((res) => {
res += 2;
console.log(res);
}); //Output: 6

p.then((res) => console.log(res)); //Output: 4
```

Promises are immutable.

Rejecting a Promise

Any promise throwing an error is considered as rejecting a promise.

```
var p = new Promise(function(resolve, reject) {
        setTimeout(() => reject("Timed out!"), 2000);
});

p.then((res) => console.log(res),
(err) => console.log(err)); //Output: Timed out!
```

Rejecting a Promise

Any promise throwing an error is considered as rejecting a promise.

```
var p = new Promise(function(resolve, reject) {
        setTimeout(() => reject("Timed out!"), 2000);
});
p.then((res) => console.log(res),
(err) => console.log(err)); //Output: Timed out!
```

Promise.catch() method:

```
var p = new Promise(function(resolve, reject) {
setTimeout(() => {throw new Error("Error encountered!");}, 2000); });
p.then((res) => console.log("Response:", res))
.catch((err) => console.log("Error:", err));//Throwing an Error is the same as calling reject().
```

Promise.all

The Promise.all() method returns a single Promise that resolves when all of the promises in the iterable argument have resolved, or rejects.

```
var p = new Promise(function(resolve, reject) {
   resolve("bar");
});
var p2 = new Promise(function(resolve, reject) {
   setTimeout(() => resolve("foo"), 2000);
});

Promise.all([p, p2]).then(function (promises) {

      promises.forEach(function (text) { console.log(text); }); //Output: bar foo after 2 secs.

});
```

Promise.race

Sometimes we don't want to wait until all of the promises have completed; rather, we want to get the results of the first promise to fulfill.

```
function delay(ms) {
    return new Promise((resolve, reject) => {
    setTimeout(resolve, ms);
}); }
Promise.race([ delay(3000).then(() => "I finished second."),  delay(2000).then(() => "I finished first.")
])
.then(function(txt) {
            console.log(txt);
})
.catch(function(err) {        console.log("error:", err); });
//Output: I finished first.
```

Thank you!!