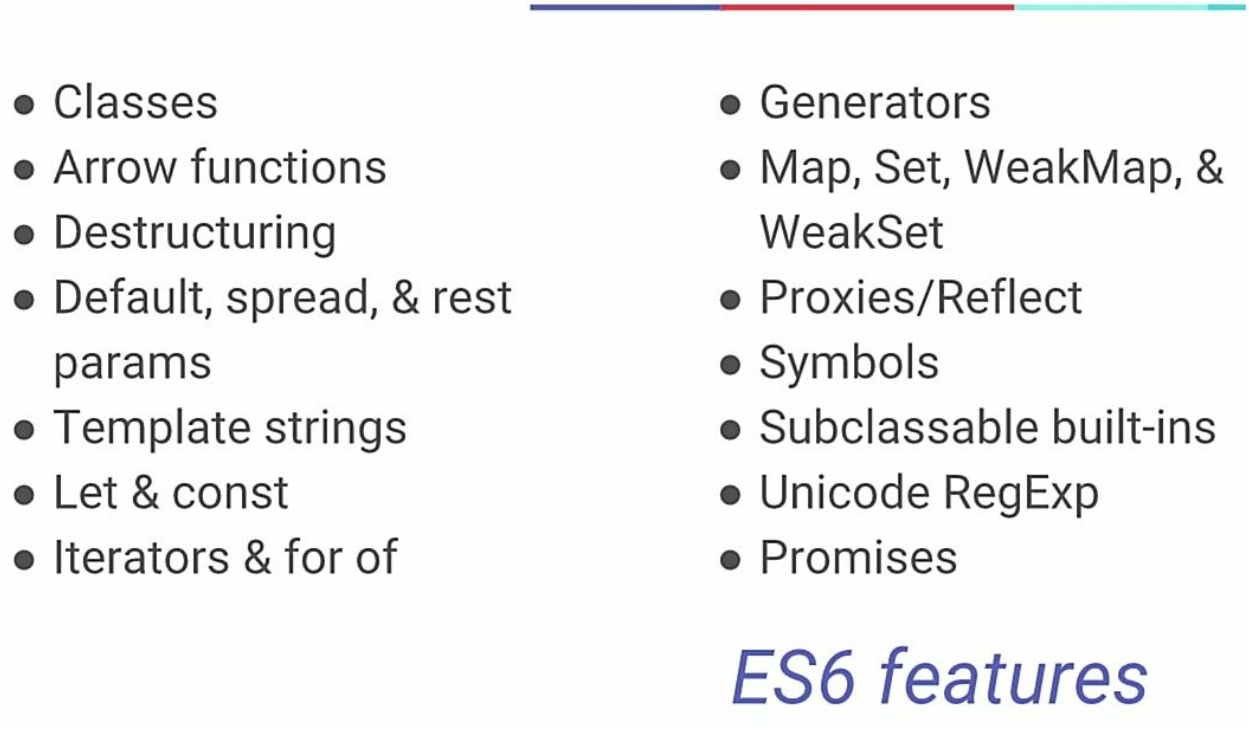
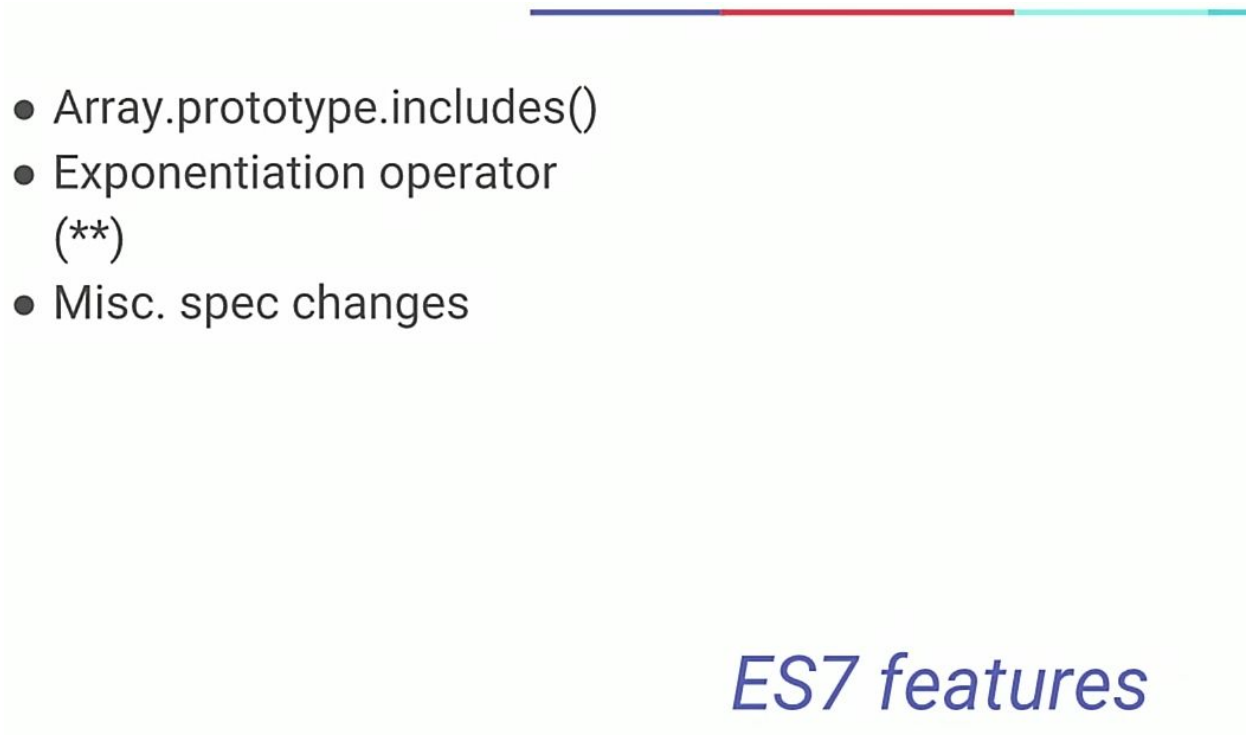
Javascript ES6

ES6

ES6 Features



ES7 Features



Run ES6 code on node 6 or older

If you are running an older version of node and want to run es6 code then run following commands.

$ npm install babel-cli -g
$ npm install babel-preset-es2015 --save-dev

E.g. if your ES6 script file name is app.js. Then run this command:
$ babel-node --presets es2015 app.js

let

```
var var1es5 = 100;
let var2es6 = 200;

if(true) {
  var1es5 = var1es5 + 10;
  var2es6 = var2es6 + 20;

  var var3es5 = 300;
  let var4es6 = 400;
```

```
}

console.log(var1es5);
console.log(var2es6);
console.log(var3es5);
// console.log(var4es6); // Can't access. Its blocked scoped
=========
110
220
300
=========
```

const

```
var var1es5 = 100;
const var2es6 = 200;

if(true) {
   var1es5 = var1es5 + 10;
   // var2es6 = var2es6 + 20; // Can't change const's value. It's read only

   var var3es5 = 300;
   const var4es6 = 400;
}

console.log(var1es5);
console.log(var2es6);
console.log(var3es5);
// console.log(var4es6); // Can't access. Its blocked scoped
============
110
200
300
============
```

Template literals

```
// Three methods to initialize Strings
var firstName = "Sheraz";
let middleName = 'Tariq';
const lastName = `Chaudhry`;

// Template literals can be multi-lines and
// can inject variables just like JSP expression language
```

```
let name = `My name
is ${firstName} ${middleName.substr(0, 1)} ${lastName}`;

console.log(name);
```
```
============
My name
is Sheraz T Chaudhry
============
```

Short object functions

These object functions do not need function keyword.

```
let person = {
  firstName: `Sheraz`,
  lastName: 'Chaudhry',
  sayName1: function() {
     return `sayName1() My name is ${this.firstName} ${this.lastName}.`;
  },
  // No function keyword used
  sayName2() {
     return `sayName2() My name is ${this.firstName} ${this.lastName}.`;
  }
};

console.log(person.sayName1());
console.log(person.sayName2());
```

Fat Arrow Function

```
let add = (a, b) => {
  return a + b;
};
console.log(add(5, 10));

// For single line function we can omit {} and return keyword.
let subtract = (a, b) => a - b;
console.log(subtract(50, 20));

// For single input parameter function we can even omit () around parameters
let double = a => a * 2;
console.log(double(3));

// But for no parameter arrow function we need to give ().
```

```
let myName = () => "My name is Sheraz";
console.log(myName());

============
15
30
6
My name is Sheraz
============
```

Fat Arrow Function & Array Map

```
let numbersArray = [1,2,3,4,5];
// map() array function runs a function on each element of array.
// map() returns array of returns forEach() do not return anything.
// And its
// ES5 method
let doubleArray1 = numbersArray.map(function (number) {
  return number * 2;
});

// ES6 arrow function method
let doubleArray2 = numbersArray.map(number => number * 2);

console.log(doubleArray1);
console.log(doubleArray2);
============
[ 2, 4, 6, 8, 10 ]
[ 2, 4, 6, 8, 10 ]
============
```

this regular vs short vs arrow functions

```
(function() {
  this.name = "Chaudhry";

  let person = {
    name: "Sheraz",
    regularFunction: function () {
      return `regularFunction(). Hi I am ${this.name}`;
    },
    // Technically short function are exactly same as regular function
    shortFunction() {
```

```
        return `shortFunction(). Hi I am ${this.name}`;
    },
    arrowFunction: () => {
        //console.log(this);
        // "this" in arrow function is lexical scope
        // Which means this in arrow function belong to parent scope
        // Below line will produce (arrowFunction(). Hi I am Chaudhry)
        return `arrowFunction(). Hi I am ${this.name}`;
    }
  };
  console.log(person.regularFunction());
  console.log(person.shortFunction());
  console.log(person.arrowFunction());
})();
============
regularFunction(). Hi I am Sheraz
shortFunction(). Hi I am Sheraz
arrowFunction(). Hi I am Chaudhry
============
```

Classes, Inheritance and Encapsulation

```
class MyClassA {
  constructor(arg1) {
    let privateVar1 = 10;
    this.publicVar1 = arg1 * 2;
    this.getPrivateVar1 = () => privateVar1;
    this.setPrivateVar1 = newPrivateVar1 => {privateVar1 = newPrivateVar1;}
  }

  processA() {
    return this.getPrivateVar1() + this.publicVar1;
  }
}

class MyClassB extends MyClassA {
  constructor() {
    super(20);
  }

  processB() {
    return this.processA() + 5;
  }
}
```

```
(function() {
  let myClassB = new MyClassB();
  console.log(myClassB.processB());
})();

============
55
============
```

Rest Function Parameter

```
// "...arg" rest parameter turns function arguments object to real array
console.log("sum1() ========");
let sum1 = function (...argumentsRest) {
  console.log(argumentsRest);
  return argumentsRest.reduce((prevNum, currNum) => prevNum + currNum);
};
console.log(sum1(2,4,6,8));

console.log("multiply1() ========");
// Rest parameter should always be the last function argument

// This is error because rest parameter should be the last parameter.
// let multiply = (...numbers, multiplier) => {
// Also there can only be one rest parameter in function
let multiply1 = (multiplier, ...numbers) => {
  // NOTE: Look at the output of "console.log(arguments);" when combination
  // of regular parameter and rest parameter are specified.
  //console.log(arguments);
  console.log(multiplier, numbers);
};
multiply1(2,3,4,5);

console.log("multiply2() ========");
let multiply2 = (multiplier, ...numbers) => {
  return numbers.map((number) => multiplier * number);
};
console.log(multiply2(2,3,4,5));

============
sum1() ========
[ 2, 4, 6, 8 ]
20
```

```
multiply1() ========
2 [ 3, 4, 5 ]
multiply2() ========
[ 6, 8, 10 ]
============
```

Spread Operator

```
let numbers = [7,2,4,6];
console.log(...numbers);

// When "..." spread operator used pass array values then it could be used
// in methods that do not expect an array. e.g. Math.max()
console.log(Math.max(...numbers));

// or function that expects less or equal number of parameters.
let mySum = (a, b, c) => a + b + c;
// Note: numbers array have 4 elements and mySum() expects 3 elements.
console.log(mySum(...numbers));
```

```
============
7 2 4 6
7
13
============
```

Object destructuring

```
let person = {
  name: "Sheraz",
  age: 20,
  // named it "personLocation" instead of "location" so that it will
  // not get confused by browser location object
  personLocation: "Atlanta"
};

// In ES5 to create a new variable from the object's property, we do:
let name1 = person.name;
let age1VarName = "age";
let age1 = person[age1VarName];
let personLocation1 = person["personLocation"];
console.log(name1, age1, personLocation1);

// In ES6 we can use destructuring
```

```
// Below line will create a new variable "name2" and put the value of person.name in it.
let {name: name2} = person;
// Or to get multiple object properties in new variable
let {age: age2, personLocation: personLocation2} = person;
console.log(name2, age2, personLocation2);

// We can also use shorthand version.
// This line will create same name variables as they are in person object.
let {name, age, personLocation} = person;
console.log(name, age, personLocation);

let {["name"]:name3, ["age"]: age3, ["personLocation"]: personLocation3} = person;
console.log(name3, age3, personLocation3);

// This syntax can be used to dynamically set property name
let name4VarName = "name";
let age4VarName = "age";
let personLocation4VarName = "personLocation";
let {[name4VarName]: name4, [age4VarName]: age4, [personLocation4VarName]:
personLocation4} = person;
console.log(name4, age4, personLocation4);
```

```
============
Sheraz 20 Atlanta
Sheraz 20 Atlanta
Sheraz 20 Atlanta
Sheraz 20 Atlanta
Sheraz 20 Atlanta
============
```

Modules - Export - Import

Module A - app_12_module_a.js

```
let var1 = "module_a var1";

let myObject01 = {
  var2: "module_a var2",
  func1: function () {
     console.log(var1 + " " + this.var2);
  }
};
```

```
// There can be one variable as default export
export default class MyClassA {
  processMyClassA() {
    console.log("Processing MyClassA");
  }
}

// There can be multiple non default export
export {myObject01};
```

Module B - app_12_module_b.js

```
let myObj2 = {
  func1: function () {
    console.log("I am module_b func1");
  }
};

let func2 = function () {
  console.log("I am module_b func2")
};

let func3 = function () {
  console.log("I am module_b func3")
};

export {myObj2 as default, func2, func3}
```

Module C - app_12_module_c.js

```
import MyClassA, {myObject01} from "./app_12_module_a";
import myObj2, {func2, func3} from "./app_12_module_b";

let myClassAInstance = new MyClassA();
myClassAInstance.processMyClassA();

console.log(myObject01.var2);

myObject01.func1();

console.log("myObj2 =",  myObj2);
func2();
func3();
```

Promises

Resolve, Reject and Then

```
/*
Promises are based on Promises/A+ and now it's part of ES6:
https://promisesaplus.com/

jQuery 3 also supports Promises/A+
https://api.jquery.com/category/deferred-object/
https://api.jquery.com/promise/

Promises are used for Asynchronous Events.

Once the promise is resolved or rejected, it can not change
it's state.


In the example below we are creating are call that will take
some time to run. In 2 seconds it will be successful and
in 1 second it will fail.
*/
let myPromise = new Promise((resolve, reject) => {
  console.log("Promise process started...");
  // We are doing setTimeout to mimic Asynchronous call
  setTimeout(() => {
     // We can pass any type of data to resolve, and
     // reject functions. This data will be passed to
     // the caller of the function.
     resolve("success");
  }, 2000);

  setTimeout(() => {
     reject("fail")
  }, 3000)

});

/*
Caller of the promise will perform separate logic on success
or failure.
```

```
/*
.then() takes in 2 function arguments. One for success and
other for failure.
*/
myPromise.then(
   (resolveData) => {
      console.log("Successful Logic:", resolveData)
   },
   (rejectData) => {
      console.log("Fail Logic:", rejectData)
   }
);
```

Catch

```
let myPromise = new Promise((resolve, reject) => {
   setTimeout(() => {
      resolve("success");
   }, 2000);

   setTimeout(() => {
      reject("fail")
   }, 1000)

});

/*
Another method to handle reject/failure is to handle
it in catch function
*/
myPromise.then(
   (resolveData) => {
      console.log("Successful Logic:", resolveData)
   }
).catch((rejectData) => {
   console.log("Fail Logic:", rejectData)
});
```

Multiple Promises - all

```
/*
If we have a situation when need to take an action
when multiple promises (maybe multiple Asynchronous tasks
or multiple animation routines) SUCCESSFULLY finish executing.

To handle above use case we use static method
```

```
Promises.all(). It takes an array of promises and return another
promise.
*/
let promiseA = new Promise((resolve, reject) => {
  setTimeout(() => {
     resolve("Promise A success");
  }, 2000);
});

let promiseB = new Promise((resolve, reject) => {
  setTimeout(() => {
     resolve("Promise B success");
  }, 1000);
});

/*
Below "resolveData" will receive array of all the
success/resolve data.
*/
Promise.all([promiseA, promiseB]).then((resolveData) => {
  console.log(resolveData);
});
```

Multiple promises fail

```
/*
If we are running multiple promises using Promise.all().then()
and a promise amoung them fails then all other promises that are
still executing will be IGNORED (but still finish their execution) and reject
executor of Promise.all().then() will run and receive that
failure data of the one that failed.

In the example below:
promiseA = takes 5 seconds to fail
promiseB = takes 1 second to succeed
promiseC = takes 1.5 seconds to fail

*/
let promiseA = new Promise((resolve, reject) => {
  console.log("Promise A started", new Date());
  setTimeout(() => {
     reject("Promise A fail");
     console.log("Promise A finished", new Date());
  }, 5000);
});
```

```
let promiseB = new Promise((resolve, reject) => {
  setTimeout(() => {
     resolve("Promise B success");
  }, 1000);
});

let promiseC = new Promise((resolve, reject) => {
  setTimeout(() => {
     reject("Promise C fail");
  }, 1500);
});

Promise.all([promiseA, promiseB, promiseC]).then((resolveData) => {
  console.log(resolveData);
}, (rejectData) => {
  console.log(rejectData);
});
```

async & await - Promise

```
/*
async and await are used to handle Promise.
By using async and await we don't have to use callback methods
*/

let makePromise = (successType) => {
    return new Promise((resolve, reject) => {
        console.log("=============\nPromise Started...");
        setTimeout(() => {
            if (successType) {
                resolve("Promise Successful!");
            } else {
                reject("Promise Failed!");
            }
        }, 2000);
    });
}

let callPromise = async () => {
    // Handling successful/resolve scenario
    let result = await makePromise(true);
    console.log(result);
```

```
    // Handling fail/reject scenario
    try {
        result = await makePromise(false);
        // Below console.log() will not be called
        console.log(result);
    } catch (error) {
        // Reject data will be returned in catch
        console.log(error);
    }
}

callPromise();


====================
$ node app_18_async_await_promise.js
=============
Promise Started...
Promise Successful!
=============
Promise Started...
Promise Failed!
====================
```