ES6

- ES6 offers a number of new features that extend the power of the language

- ES6 is not widely supported in today's browsers, so it needs to be transpiled to ES5

ES6 Features

- Constants and Block Scoped Variables
- Spread and Rest operators
- Destructuring
- Arrow Functions
- Classes
- Template Strings
- Inheritance
- Promise

Constants and Block Scoped Variables

```
var i;
for (i = 0; i < 10; i += 1) {
  var j = i;
  let k = i;
}
console.log(j); // 9
console.log(k); // undefined
```

```
const myName = 'pat';

let yourName = 'jo';

yourName = 'sam';

myName = 'jan'; // error
```

Spread Parameter

- Allows in-place expansion of an expression

```
let cde = ['c', 'd', 'e'];
let scale = ['a', 'b', ...cde, 'f', 'g']; // ['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

Rest parameter

- Rest parameters are used to access indefinite number of arguments passed to a function.

```
function add(...numbers) {
return numbers[0] + numbers[1];
}

add(3, 2); // 5
```

Array Destructuring

- Quickly extracts data out of an {} or [] without having to write much code.
  - let foo = ['one', 'two', 'three'];
  - let [one, two, three] = foo;
  - console.log(one); // 'one'

Object Destructuring

```
let options = {
  title: "Menu",
  width: 100,
  height: 200
};

let {title, width, height} = options;

alert(title);  // Menu
alert(width);  // 100
alert(height); // 200
```

Arrow Functions

- The new "fat arrow" notation can be used to define anonymous functions in a simpler way.

```
var hello = function() {
      return "Hello World!";
    }

var hello = () => {
      return "Hello World!";
    }
```

this

```
class Employee {
public name:string;
public address:string;
display1():void
{
setTimeout(function()
{
console.log(this.name); },1000);
}}
display2():void
{
setTimeout(() =>
{ console.log(this.name); } ,1000);
}}
```

```
let emp:Employee = new Employee();
emp.name="Sunny";
emp.address="Pune"
emp.display1();
emp.display2();
```

Classes

- Classes are a new feature in ES6, used to describe the blueprint of an object

```
class Hamburger {
constructor() {
    // This is the constructor.
}
listToppings() {
    // This is a method.
}
}
```

Object

- An object is an instance of a class which is created using the new operator.

```
let burger = new Hamburger();
burger.listToppings();
```

Template Strings

```
var name = 'Sam';
var age = 42;
console.log('hello my name is ' + name + ' I am ' + age + ' years old');

var name = 'Sam';
var age = 42;
console.log(`hello my name is ${name}, and I am ${age} years old`);
```

Inheritance

```
// Base Class : ES6
class Bird {
  constructor(weight, height) {
    this.weight = weight;
    this.height = height;
  }
  walk() {
    console.log('walk!');
  }
}
```

```
// Subclass
class Penguin extends Bird {
  constructor(weight, height) {
    super(weight, height);
  }
  swim() {
    console.log('swim!');
  }
}
// Penguin object
let penguin = new Penguin(...);
penguin.walk(); //walk!
penguin.swim(); //swim!
```

Promise

- JavaScript Promises are a new addition to ES6
- The promise constructor takes one argument, a callback with two parameters, resolve and reject.
- Do something within the callback, then call resolve if everything worked, otherwise call reject.

Syntax

```
var mypromise = new Promise(function(resolve, reject) {

        // asynchronous code to run here
         // call resolve() to indicate task successfully completed
         // call reject() to indicate task has failed
})
```

- resolve(value) — if the job finished successfully, with result value.
- reject(error) — if an error occurred, error is the error object.

Creating Promise

```
function add(a, b) {
    let pr = new Promise((resolve, reject) => {
        if (a < 0 || b < 0) {
            reject('Invalid Nos.')
        }
        else {
            resolve(a + b);
        }
    });
    return pr;
}
```

Using Promise

```
add(-10, 20).then((res) => {
   console.log(res);
}, (err) => {
   console.log(err);
});
console.log("_______________");
console.log("_______________");
```

async/await

- Special syntax to work with promises in a more comfortable fashion

```
async function f() {
  return 1;
}

  f().then(alert); // 1
```

- “async” before a function always returns a promise

await

- await makes JavaScript wait until that promise settles and returns its result.

```
async function f() {

  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("done!"), 1000)
  });

  let result = await promise; // wait until the promise resolves (*)

  alert(result); // "done!"
}

f();
```

Error handling

```
async function thisThrows() {
    throw new Error("Thrown from thisThrows()");
}

async function run() {
    try {
        await thisThrows();
    } catch (e) {
        console.error(e);
    } finally {
        console.log('We do cleanup here');
    }
}

run();
```

Type Script

- An open source programming language developed and maintained by Microsoft.
- TypeScript is a typed superset of JavaScript that compiles to plain JavaScript.
- Brings OOP constructs to JavaScript.
- >npm install –g typescript
- https://www.typescriptlang.org/play/

Static Typing

- A very distinctive feature of TypeScript is the support of static typing.
- Code written in TypeScript is more predictable, and is generally easier to debug.
- Declare the types of variables, and the compiler will make sure that they aren't assigned the wrong types of values.

Basic Types

- Boolean
  - let isDone: boolean = false;
- Number
  - let num: number = 6;
- String
  - let color: string = "blue";
- Array
  - let list: number[] = [1, 2, 3];
  - let list: Array<number> = [1, 2, 3];

Basic Types

- enum
  - enum Color {Red, Green, Blue}
  - let c:Color = Color.Green;
- any
  - let notSure: any = 4;
  - notSure = "maybe a string instead";
- void

  function warnUser(): void

  { alert("This is my warning message"); }

Interfaces

- Interfaces are used to type-check whether an object fits a certain structure.

```
interface interface_name {
        // variables' declaration
        // methods' declaration
}
```

- When translated to JavaScript, interfaces disappear - their only purpose is to help in the development stage.

```
interface LabeledValue {
    label: string;
}

function printLabel(labeledObj: LabeledValue) {
    console.log(labeledObj.label);
}

let myObj = {size: 10, label: "Size 10 Object"};
printLabel(myObj);
```

Implementing an interface

```
interface ClockInterface {
    currentTime: Date;
    setTime(d: Date): void;
}

class Clock implements ClockInterface {
    currentTime: Date = new Date();
    setTime(d: Date) {
        this.currentTime = d;
    }
    constructor(h: number, m: number) { }
}
```

RxJS

- The term, “reactive,” refers to programming models that are built around reacting to changes.

- Built around publisher-subscriber pattern

- In reactive style of programming, we make a request for resource and start performing other things. When the data is available, we get the notification along with data inform of call back function. In callback function, we handle the response as per application/user needs.

Stream

- A stream can emit 3 different things:
  - Value
  - Error
  - Completed signal

```
class ContactService {
contacts = [
   new Contact("John","John@gmail.com","983344562"),
   new Contact("Charles","charles@gmail.com","983344123")
];

public findAll(): Observable<Array<Contact>> {
      var obs = Observable.create((o) => {
         o.next(this.contacts);
         o.complete();
      });
            return obs;
}
```

```
var contactService<ContactService> = new ContactService();


this.contactService.findAll().subscribe((data) => {
     console.log(data);
   });
```

```
const rxjs = require('rxjs');
const products = require("./product.store");
class RProductService {
    getAllProducts() {
        let ob = new rxjs.Observable((subs) => {
            setTimeout(() => {
                subs.next(products[0]);

            }, 1000);
            setTimeout(() => {
                subs.next(products[1]);
            }, 2000);
            setTimeout(() => {
                subs.next(products[2]);
                subs.complete();
            }, 3000);
        });
        return ob;
    }
}
module.exports = RProductService;
```

```
const RProductService = require("./rproduct.service");

let ps = new RProductService();

let s = ps.getAllProducts().subscribe((p) => {
    console.log(p);
}, (err) => {
    console.log('Error.....')
}, () => {
    console.log('completed.....')
    s.unsubscribe();
    console.log('released...')
});
console.log("_________________");
```