Introduction to ES6

- Part 1

Objective

- Learn why we should use ES6
- Have a basic understanding of the major features of ES6
- Understand how to use arrow functions, classes, enhanced object literals, template strings and destructuring.
- Learn the difference between ES6 and the older version.

Content

- What is ES6?
- ES6 features
- Part 1:
  - Arrows
  - Classes
  - Enhanced object literals
  - Template strings
  - Destructuring

What is ES6?

- ECMAScript is a Standard for a scripting languages. Languages like Javascript are based on the ECMAScript standard.
- ECMAScript 6, also known as ECMAScript 2015, is the latest version of the ECMAScript standard.

ES6 features

- Part 1:
  - Arrows
  - Classes
  - Enhanced object literals
  - Template strings
  - Destructuring

ES6 features

- Part 2:
  - Default + rest + spread
  - Let + const
  - Iterators + for .. of
  - Generators

ES6 features

- Part 3:
  - Modules (NodeJS)
  - Map + Set + WeakMap + WeakSet
  - Promises
  - Math + Number + String + Array + Object APIs

Part 1

Part 1: Arrows

- **Syntax**

```
(param1, param2, …, paramN) => { statements }
(param1, param2, …, paramN) => expression
// equivalent to: (param1, param2, …, paramN) => { return expression; }

// Parentheses are optional when there's only one parameter name:
(singleParam) => { statements }
singleParam => { statements }

// A function with no parameters could be written with a couple of parentheses or with an underscore:
() => { statements }
_ => { statements }
```

Part 1: Arrows

- **Advantages**
  - Shorter functions
  - Non-binding of *this*

Part 1: Arrows

- Shorter functions

Example:

```
var materials = [
        'Hydrogen',
        'Helium',
        'Lithium',
        'Beryllium'
];
var materialsLength1 = materials.map(function(material) {
        return material.length;

}); // [8,6,7,9]
```

Part 1: Arrows

- Shorter functions

  Example: (cont.)

```
var materialsLength2 = materials.map((material) => {
        return material.length;
}); // [8,6,7,9]

var materialsLength3 = materials.map(material => material.length); // [8,6,7,9]
```

Part 1: Arrows

- Non-binding of *this*

Until arrow functions, every new function defined its own *this* value (**a new object** in the case of a constructor, **undefined** in strict mode function calls, **the context object** if the function is called as an "object method", etc.). This proved to be annoying with an object-oriented style of programming.

Part 1: Arrows

- Non-binding of *this*

Example:

```
function Person() {
  // The Person() constructor defines `this` as an instance of itself.
  this.age = 0;
  [1, 2, 3].forEach(function growUp() {
    // In non-strict mode, the growUp function defines `this`
    // as the global object, which is different from the `this`
    // defined by the Person() constructor.
    this.age++;
  });
}
```

Part 1: Arrows

- Non-binding of *this*

Example:

⇒ Fix by assign this into a variable (that)

```
function Person() {
  var that = this;
  that.age = 0;
  [1, 2, 3].forEach(function growUp() {
    // The callback refers to the `that` variable of which
    // the value is the expected object.
    that.age++;
  });
}
```

Part 1: Arrows

- Non-binding of *this*

  Example:

⇒ Fix by using bind() method

```
function Person() {
  this.age = 0;
  [1, 2, 3].forEach((function growUp() {
    this.age++;
  }).bind(this));
}
```

Part 1: Arrows

- Non-binding of *this*

Example:

⇒ Fix by using ES6 arrow function

```
function Person() {
  this.age = 0;
  [1, 2, 3].forEach(() => {
    this.age++;
  });
}
```

Part 1: Classes

a. Defining classes

Class declarations

Class expressions

Part 1: Classes

- Class declarations

```
class Rectangle {
        constructor(height, width) {
        this.height = height;
        this.width = width;
        }
}
```

Note:
Class declarations are not **hoisted**, unlike function declarations.

Part 1: Classes

- Class expressions

```
// unnamed
var Rectangle = class {
        constructor(height, width) {
        this.height = height;
        this.width = width;
        }
};
```

```
// named
var Rectangle = class Rectangle {
        constructor(height, width) {
        this.height = height;
        this.width = width;
        }
};
```

Note:
Class expressions are not **hoisted**, unlike function expressions.

b. Class body and method definitions

- **Strict mode**

The bodies of class declarations and class expressions are executed in strict mode.

- **Constructor**

The constructor method is a special method for creating and initializing an object created with a class. There can **only be one** special method with the name "constructor" in a class. A *SyntaxError* will be thrown if the class contains more than one occurrence of a constructor method.

b. Class body and method definitions

- **Prototype methods**

```
class Rectangle {
        constructor(height, width) {
                this.height = height;
                this.width = width;
        }
        get area() {
                return this.calcArea();
        }
        calcArea() {
                return this.height * this.width;
        }
}
```

b. Class body and method definitions

- **Static methods**

  - The *static* keyword defines a static method for a class.

  - Static methods are called without instantiating their class and cannot be called through a class instance.

b. Class body and method definitions

- **Static methods**

```
class Point {
        constructor(x, y) {
                this.x = x;
                this.y = y;
        }
        static distance(a, b) {
                const dx = a.x - b.x;
                const dy = a.y - b.y;
                return Math.sqrt(dx*dx + dy*dy);
        }
}
```

b. Class body and method definitions

- **Static methods**

```
...
const p1 = new Point(5, 5);
const p2 = new Point(10, 10);

console.log(Point.distance(p1, p2));
```

Part 1: Classes

c. Sub classing with extends

```
class Animal {
        constructor(name) {
                this.name = name;
        }

        speak() {
                console.log(this.name + ' makes a noise.');
        }
}
```

c. Sub classing with extends

```
...
class Dog extends Animal {
        speak() {
                console.log(this.name + ' barks.');
        }
}
var d = new Dog('Mitzie');
d.speak();
```

Note:

If there is a constructor present in sub-class, it needs to first call **super()** before using "this".

d. Super class calls with super

The **super** keyword is used to call functions on an object's parent.

```
class Cat {
        constructor(name) {
                this.name = name;
        }
        speak() {
                console.log(this.name + ' makes a noise.');
        }
}
```

d. Super class calls with super

...

```
class Lion extends Cat {
        speak() {
                super.speak();
                console.log(this.name + ' roars.');
        }
}
```

Part 1: Enhanced Object Literals

- In ES2015, object literals are extended to support:
  - setting the prototype at construction
  - **shorthand** for foo: foo assignments
  - defining methods
  - making super calls
  - computing property names with expressions.

Part 1: Template string

- Template literals are enclosed by the **back-tick** (` `) (grave accent) character instead of double or single quotes.
- Template literals can contain place holders. These are indicated by the **Dollar sign** and **curly braces** (${expression}). The expressions in the place holders and the text between them get passed to a function.

Part 1: Template string

a. Multi-line strings

Using normal strings, you would have to use the following syntax in order to get multi-line strings:

```
console.log('string text line 1\n' +
'string text line 2');
// "string text line 1
// string text line 2"
```

Part 1: Template string

a. Multi-line strings

To get the same effect with multi-line strings, **you can now write:**

```
console.log(`string text line 1
string text line 2`);
// "string text line 1
// string text line 2"
```

b. Expression interpolation

In order to embed expressions within normal strings, you would use the following syntax:

```
var a = 5;
var b = 10;
console.log('Fifteen is ' + (a + b) + ' and\nnot ' + (2 * a + b) + '.');
// "Fifteen is 15 and
// not 20."
```

b. Expression interpolation

Now, with template literals, you are able to make use of the syntactic sugar making substitutions like this more readable:

```
var a = 5;
var b = 10;
console.log(`Fifteen is ${a + b} and
not ${2 * a + b}.`);
// "Fifteen is 15 and
// not 20."
```

Part 1: Destructuring

a. Array destructuring

- Basic variable assignment

```
var foo = ['one', 'two', 'three'];

var [one, two, three] = foo;
console.log(one); // "one"
console.log(two); // "two"
console.log(three); // "three"
```

a. Array destructuring

- Assignment separate from declaration

A variable can be assigned its value via destructuring separate from the variable's declaration.

```
var a, b;

[a, b] = [1, 2];
console.log(a); // 1
console.log(b); // 2
```

Part 1: Destructuring

a. Array destructuring

- Basic variable assignment

```
var foo = ['one', 'two', 'three'];

var [one, two, three] = foo;
console.log(one); // "one"
console.log(two); // "two"
console.log(three); // "three"
```

Part 1: Destructuring

a. Array destructuring

- Assignment separate from declaration

A variable can be assigned its value via destructuring separate from the variable's declaration.

```
var a, b;

[a, b] = [1, 2];
console.log(a); // 1
console.log(b); // 2
```

a. Array destructuring

- Default values

A variable can be assigned a default, in the case that the value pulled from the array is undefined.

```
var a, b;

[a=5, b=7] = [1];
console.log(a); // 1
console.log(b); // 7
```

a. Array destructuring

- Swapping variables

Two variables values can be swapped in one destructuring expression.

```
var a = 1;
var b = 3;

[a, b] = [b, a];
console.log(a); // 3
console.log(b); // 1
```

Part 1: Destructuring

a. Array destructuring

- Parsing an array returned from a function

It's always been possible to return an array from a function. Destructuring can make working with an array return value more concise.

```
function f() {
        return [1, 2];
}
var a, b;
[a, b] = f(); //a = 1, b = 2
```

a. Array destructuring

- Ignoring some returned values

You can ignore return values that you're not interested in:

```
function f() {
        return [1, 2, 3];
}
var [a, , b] = f();
console.log(a); // 1
console.log(b); // 3
```

Part 1: Destructuring

a. Array destructuring

- Assigning the rest of an array to a variable

  When destructuring an array, you can assign the remaining part of it to a variable using the **rest** pattern:

```
var [a, ...b] = [1, 2, 3];
console.log(a); // 1
console.log(b); // [2, 3]
```

b. Object destructuring

- Basic assignment

```
var o = {p: 42, q: true};
var {p, q} = o;

console.log(p); // 42
console.log(q); // true
```

b. Object destructuring

- Assignment without declaration

A variable can be assigned its value with destructuring separate from its declaration.

```
var a, b;
({a, b} = {a: 1, b: 2});
```

b. Object destructuring

- Assigning to new variable names

  A property can be unpacked from an object and assigned to a variable with a different name than the object property.

```
var o = {p: 42, q: true};
var {p: foo, q: bar} = o;
console.log(foo); // 42
console.log(bar); // true
```

b. Object destructuring

- Default values

  A variable can be assigned a default, in the case that the value pulled from the object is undefined.

```
var {a = 10, b = 5} = {a: 3};

console.log(a); // 3
console.log(b); // 5
```

b. Object destructuring

- Setting a function parameter's default value

```
function drawES2015Chart({size = 'big', cords = {x: 0, y: 0}, radius =
25} = {}) {
        console.log(size, cords, radius);
        // do some chart drawing
}
drawES2015Chart({
        cords: {x: 18, y: 30},
        radius: 30
});
```

b. Object destructuring

- Nested object and array destructuring

```
var metadata = {
title: 'Scratchpad',
translations: [
                {
                url: '/de/docs/Tools/Scratchpad',
                title: 'JavaScript-Umgebung'
                }
],
url: '/en-US/docs/Tools/Scratchpad'
};

…
```

b. Object destructuring

- Nested object and array destructuring

```
...
var {title: englishTitle, translations: [{title: localeTitle}]} =          metadata;

console.log(englishTitle); // "Scratchpad"
console.log(localeTitle);  // "JavaScript-Umgebung"
```

b. Object destructuring

- Pulling fields from objects passed as function parameter

```
function userId({id}) {
        return id;
}

function whois({displayName, fullName: {firstName: name}}) {
        console.log(displayName + ' is ' + name);
}
...
```

b. Object destructuring

- Pulling fields from objects passed as function parameter

```
...
var user = {
                id: 42,
                displayName: 'jdoe',
                fullName: {
                firstName: 'John',
                lastName: 'Doe'
                }
};

console.log('userId: ' + userId(user)); // "userId: 42"
whois(user); // "jdoe is John"
```

Thank you