Introduction to ES6

- Part 3

Objective

- Have a basic understanding of the major features of ES6
- Learn the basic syntax of ES6 Modules
- Understand how to use Map, Set, WeakMap, WeakSet and the differences among them.
- Learn how to use Promise.
- Acknowledge of some new methods of Math, Number, String, Array and Object.

Content

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

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 3

- **Syntax**

export	import

export **default** expression;
 	
export **default** function (…) { … } // also class, function*
 	
export **default** function name1(…) { … } // also class, function*
 	
export { name1 as **default**, … };
	import **defaultMember** from "module-name";

- **Syntax**

export	import

export { name1, name2, …, nameN };
 	
export { variable1 as name1, variable2 as name2, …, nameN };
 	
export let name1, name2, …, nameN; // also var
 	
export let name1 = …, name2 = …, …, nameN; // also var, const
	import * as name from "module-name";
	
	import { member } from "module-name";
	
	import { member1 , member2 } from "module-name";

- **Alias**

```
import * as name from "module-name";
import { member as alias } from "module-name";
export { variable1 as name1, variable2 as name2, …, nameN }
```

a. Map

- **Description**

The **Map** object holds key-value pairs. Any value (both objects and primitive values) may be used as either a key or a value.

A **Map** object iterates its elements in insertion order — a for...of loop returns an array of [key, value] for each iteration.

a. Map

- **Syntax**

new Map([iterable])

Parameters

iterable

- An **Array** or other **iterable** object whose elements are key-value pairs. Each key-value pair added to the new Map. null is treated as undefined.

a. Map

- **Example**

```
var myMap = new Map();

var keyString = 'a string',
keyObj = {},
keyFunc = function() {};

// setting the values
myMap.set(keyString, "value associated with 'a string'");
myMap.set(keyObj, 'value associated with keyObj');
myMap.set(keyFunc, 'value associated with keyFunc');

myMap.size; // 3
...
```

a. Map

- **Example**

```
...
// getting the values
myMap.get(keyString);    // "value associated with 'a string'"
myMap.get(keyObj);       // "value associated with keyObj"
myMap.get(keyFunc);      // "value associated with keyFunc"

myMap.get('a string');   // "value associated with 'a string'"
      // because keyString === 'a string'
myMap.get({});           // undefined, because keyObj !== {}
myMap.get(function() {}) // undefined, because keyFunc !== function () {}
```

b. Set

- **Description**

The **Set** object lets you store unique values of any type, whether primitive values or object references.

Set objects are collections of values. You can iterate through the elements of a set in insertion order. A value in the Set may only occur once; it is unique in the **Set**'s collection.

b. Set

- **Syntax**

new Set([iterable])

Parameters

iterable

- If an **iterable** object is passed, all of its elements will be added to the new Set.

If null is passed instead of iterable, it is treated as not passing iterable at all.

b. Set

- **Example**

```
var mySet = new Set();

mySet.add(1); // Set { 1 }
mySet.add(5); // Set { 1, 5 }
mySet.add(5); // Set { 1, 5 }
mySet.add('some text'); // Set { 1, 5, 'some text' }
var o = {a: 1, b: 2};
mySet.add(o);

mySet.add({a: 1, b: 2}); // o is referencing a different object so      this is okay
...
```

b. Set

- **Example**

```
...
mySet.has(1); // true
mySet.has(3); // false, 3 has not been added to the set
mySet.has(5);              // true
mySet.has(Math.sqrt(25));  // true
mySet.has('Some Text'.toLowerCase()); // true
mySet.has(o); // true

mySet.size; // 5

mySet.delete(5); // removes 5 from the set
mySet.has(5);    // false, 5 has been removed
...
```

b. Set

- **Example**

```
...
mySet.size; // 4, we just removed one value
console.log(mySet); // Set {1, "some text", Object {a: 1, b: 2},       Object {a: 1, b: 2}}
```

c. WeakMap

- **Description**

The **WeakMap** object is a collection of key/value pairs in which the keys are *weakly referenced*.

The keys must be objects and the values can be arbitrary values.

c. WeakMap

- **Syntax**

new WeakMap([iterable])

Parameters

iterable

- If an **iterable** object is passed, all of its elements will be added to the new WeakSet. null is treated as undefined.

c. WeakMap

- **Why use it**

With manually written maps, the array of keys would keep references to key objects, preventing them from being garbage collected. In native **WeakMaps**, references to key objects are held "*weakly*", which means that they do not prevent garbage collection in case there would be no other reference to the object.

⇒ Prevent memory leak issue

d. WeakSet

- **Description**

The **WeakSet** object lets you store weakly held objects in a collection.

WeakSet objects are collections of objects. An object in the **WeakSet** may only occur once; it is unique in the **WeakSet**'s collection.

d. WeakSet

- **Description**

The main differences to the Set object are:

- In contrast to Sets, **WeakSets** are collections of objects only and not of arbitrary values of any type.
- The **WeakSet** is *weak*: References to objects in the collection are held *weakly*. If there is no other reference to an object stored in the **WeakSet**, they can be garbage collected. That also means that there is no list of current objects stored in the collection. **WeakSets** are not enumerable.

d. WeakSet

- **Syntax**

```
new WeakSet([iterable])
```

Parameters

iterable

- If an **iterable** object is passed, all of its elements will be added to the new WeakSet. null is treated as undefined.

Definitions

Promises are a library for asynchronous programming. The **Promise** object represents the eventual completion (or failure) of an asynchronous operation, and its resulting value.

Part 3: Promises

- **Syntax**

```
new Promise( /* executor */ function(resolve, reject) { ... } );
```

Description

Instead of immediately returning the final value, the asynchronous method returns a **promise** to supply the value at some point *in the future*.

A **Promise** is in one of these states:

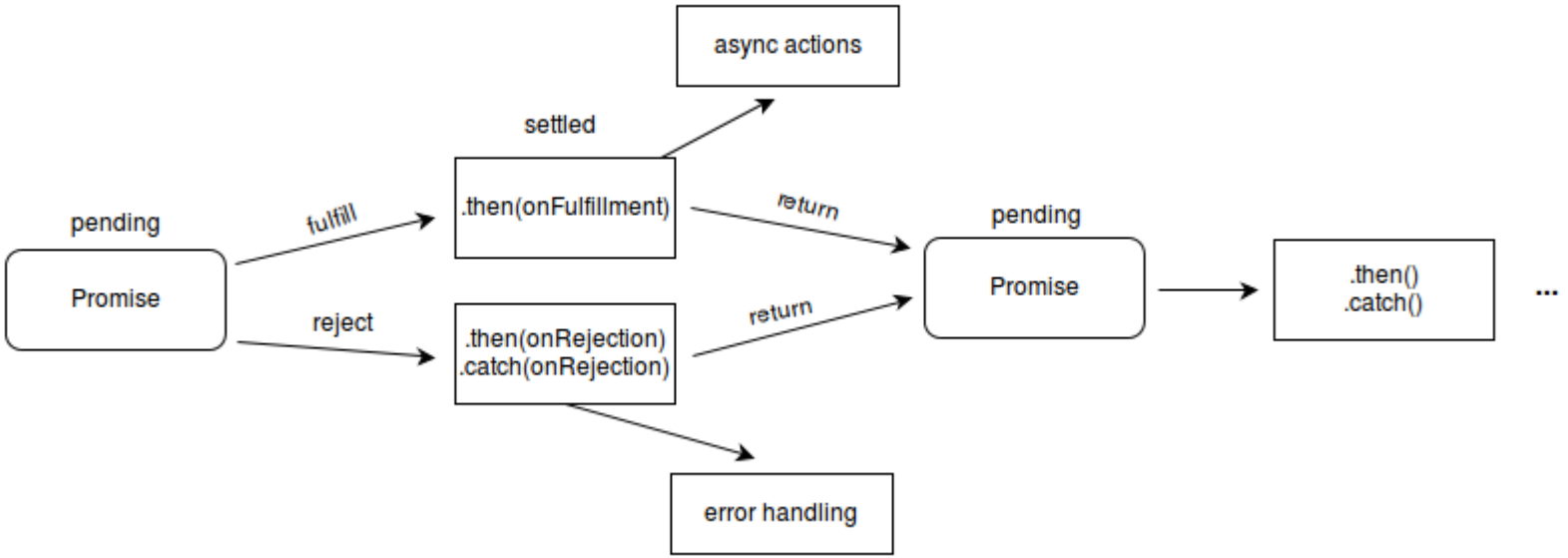
- *pending*: initial state, not fulfilled or rejected.
- *fulfilled*: meaning that the operation completed successfully.
- *rejected*: meaning that the operation failed.

Description

A *pending* promise can either be *fulfilled* with a value, or *rejected* with a reason (error). When either of these options happen, the associated handlers queued up by a **promise**'s *then* method are called.

As the Promise.prototype.then() and Promise.prototype.catch() methods return **promises**, they can be chained.

- **Description**



- **Examples**

```
function a(x) {
  return new Promise((resolve, reject) => {
if (x > 0) {
  resolve('Greater than 0');
}
reject('Less than 0');
  });
}
...
```

- **Examples**

```
...
a(-1).then((value) => {
  console.log(`1st then: ${value}`);
  return value;
}).catch((err) => {
  console.log(`1st catch: ${err}`);
  throw new Error(err);
}).then((value) => {
  console.log(`2nd then: ${value}`);
}).catch((err) => {
  console.log(`2nd catch: ${err.message}`);
});
```

- **Promise.all**

  - Definition

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

- Syntax

```
Promise.all(iterable);
```

- **Promise.all**

  - Examples

  Promise.all waits for all fulfillments

```
var p1 = Promise.resolve(3);
var p2 = 1337;
var p3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, 'foo');
});

Promise.all([p1, p2, p3]).then(values => {
  console.log(values); // [3, 1337, "foo"]
});
```

- **Promise.all**

  - Examples

  Promise.all is immediately rejected if any of the elements are rejected.

  For example, if you pass in four promises that resolve after a timeout and one promise that rejects immediately, then Promise.all will reject immediately.

- **Promise.all**

  - Examples

```
var p1 = new Promise((resolve, reject) => {
  setTimeout(resolve, 1000, 'one');
});

var p2 = new Promise((resolve, reject) => {
  setTimeout(resolve, 2000, 'two');
});

var p3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 3000, 'three');
});
...
```

- **Promise.all**

  - Examples

```
...
var p4 = new Promise((resolve, reject) => {
  setTimeout(resolve, 4000, 'four');
});
var p5 = new Promise((resolve, reject) => {
  reject('reject');
});

Promise.all([p1, p2, p3, p4, p5]).then(values => {
  console.log(values);
}).catch(reason => {
  console.log(reason)
});
```

- **Math**

  - Math.sign(x)

  Returns the sign of x as -1 or +1. Unless x is either NaN or zero; then x is returned.

```
> Math.sign(-8)
-1
> Math.sign(3)
1

> Math.sign(0)
0
> Math.sign(NaN)
NaN
```

```
> Math.sign(-Infinity)
-1
> Math.sign(Infinity)
1
```

- **Math**

  - Math.trunc(x)

  Removes the decimal fraction of x.

```
> Math.trunc(3.1)
3
> Math.trunc(3.9)
3
> Math.trunc(-3.1)
-3
> Math.trunc(-3.9)
-3
```

- **Math**

  - Math.cbrt(x)

  Returns the cube root of x ($\sqrt[3]{x}$).

  > Math.cbrt(8)
  2

- **Number**

  - Number.isNaN(x)

  Is number the value NaN? Making this check via === is hacky. NaN is the **only value that is not equal to itself**:

  > let x = NaN;
  > x === NaN
  False

  Therefore, this expression is used to check for it
  > x !== x
  True

  Using Number.isNaN() is more self-descriptive:
  > Number.isNaN(x)
  true

- **Number**

  - Number.isInteger(x)

```
> Number.isInteger(-17)
true
> Number.isInteger(33.0)
true
> Number.isInteger(33.1)
false
> Number.isInteger('33')
false
> Number.isInteger(NaN)
false
> Number.isInteger(Infinity)
false
```

- **String**

```
"abcde".includes("cd") // true
"abc".repeat(3) // "abcabcabc"
```

- **Array**

  - Array.from()

  The Array.from() method creates a new Array instance from an array-like or iterable object.

  Examples:

  - Create a new array from an iterable object

```
const bar = ["a", "b", "c"];
Array.from(bar);
// ["a", "b", "c"]

Array.from('foo');
// ["f", "o", "o"]
```

- **Array**

  - Array.from()

  Examples:

  - Create a new array from an array-like object

```
const obj = {
  0: 1,
  1: 2,
  2: 3,
  length: 3,
};

Array.from(obj);
// [1, 2, 3]
```

- **Array**

  - Array.of()

  The **Array.of()** method creates a new Array instance with a variable number of arguments, regardless of number or type of the arguments.

  The difference between **Array.of()** and the **Array** constructor is in the handling of integer arguments: **Array.of(7)** creates an array with a single element, 7, whereas **Array(7)** creates an empty array with a length property of 7

- **Array**

  - Array.of()

  Example:

```
Array.of(7);       // [7]
Array.of(1, 2, 3); // [1, 2, 3]

Array(7);          // [ , , , , , , ]
Array(1, 2, 3);    // [1, 2, 3]
```

- **Array**

  - Array.copyWithin()

  The **copyWithin()** method shallow copies part of an array to another location in the same array and returns it, without modifying its size.

Syntax:

```
arr.copyWithin(target)
arr.copyWithin(target, start)
arr.copyWithin(target, start, end)
```

- **Array**

  - Array.copyWithin()

  Example:

```
[1, 2, 3, 4, 5].copyWithin(-2);
// [1, 2, 3, 1, 2]

[1, 2, 3, 4, 5].copyWithin(0, 3);
// [4, 5, 3, 4, 5]

[1, 2, 3, 4, 5].copyWithin(0, 3, 4);
// [4, 2, 3, 4, 5]

[1, 2, 3, 4, 5].copyWithin(-2, -3, -1);
// [1, 2, 3, 3, 4]
```

- **Array**

  - Array.find()

  Example:

```
const students = [
  {
name: 'A',
age: 20,
  },
  {
name: 'B',
age: 19,
  },
];
const u20 = students.find(student => student.age < 20);
// { name: 'B', age: 19 }
```

- **Object**

  - Object.assign()

The Object.assign() method is used to copy the values of all enumerable own properties from one or more source objects to a target object. It will return the target object.

Syntax:

```
Object.assign(target, ...sources)
```

- **Object**

  - Object.assign()

  Example:

```
var obj = { a: 1 };
var copy = Object.assign({}, obj);
console.log(copy); // { a: 1 }
```

Thank you