Assembler
School of Software Engineering

ES6 and Beyond

Let, const, var and IIFE

Var:

1. Global and function scoped.
2. Flexibility at the time playing with the scopes.
3. Hoisted to the top.
4. Can pollute the scope if is not declared (No on strict mode).
5. Misleading redeclaration. Can overwrite outer scopes.

Let, const, var and IIFE

```
var greeting = "Hi";
    var cont = 4;

    if (cont > 3) {
        var greeting = "Say Hi instead";
    }

    console.log(greeting) //"Say Hi
instead"
```

```
var role = "Engineer";

console.log(role);

function displayRole(){

    role = "developer";

    console.log(role);

}

displayRole(); console.log(role);
```

Let, const, var and IIFE

Let:

1. It’s block scoped. Loves curly braces.
2. Cannot be redeclared
3. Hoisted to the top.
4. Consisting flow. declaration -> assignment.
5. Bit less of flexibility in occasions where is useful a variable across multiples scopes.

Let, const, var and IIFE

```
let greeting = "Hi";
   let cont = 4;

   if (cont > 3) {
        let hi = "say Hello instead";
        console.log(hi);//"say Hello instead"
    }
  console.log(hi) // hi is not defined
```

Let, const, var and IIFE

const:

1. Can maintain values.
2. Block scoped.
3. Bring coherence to our applications.
4. Cannot be updated o re declared.
5. Cannot be declared without being initialized.

Shadowing

There are several ways you might shadow something:

1. With a variable declaration (var, let, const)
2. With a parameter declaration:

```
var foo; // The outer one function
example(foo) { // This parameter
shadows the outer `foo` // ... }
```

3. With a function declaration:

```
var foo; // The outer one
function example() { function
foo() { // This shadows the
outer `foo` } // ... }
```

Modules

Why:

1. Maintainability
2. Namespacing
3. Reusability

```
const a = 1
const b = 2
const c = 3

export { a, b, c }
```

```
import * from 'module'

import { a } from 'module'

import { a, b } from 'module'

import { a, b as two } from 'module'
```

Promises

Signing a contract...the resolution will happen later...defining if the contract is accepted or rejected

Most important feature of modern JS. Asynchronous JavaScript is the backbone of the modern web

Why?

1. Basically, no way of dealing with async code.
2. Asynchronicity - the feature that makes dynamic web applications possible
3. Nice error handling process
4. Cleaner readable style
5. Non-blocking applications
6. They will always fall on Resolve or Reject

```
// Callback Hell


a(function (resultsFromA) {
    b(resultsFromA, function (resultsFromB) {
        c(resultsFromB, function (resultsFromC) {
            d(resultsFromC, function (resultsFromD) {
                e(resultsFromD, function (resultsFromE) {
                    f(resultsFromE, function (resultsFromF) {
                        console.log(resultsFromF);
                    })
                })
            })
        })
    })
});

```

```
// In our imaginary world it takes three Promises to build a boat and conquer the ocean
let getWood = () => { return new Promise((resolve, reject) => { resolve('Got wood.'); });
let buildBoat = () => { return new Promise((resolve, reject) => { resolve('Built a boat.'); });
let sailTheOcean = () => { return new Promise((resolve, reject) => { resolve('Sailed the ocean.');


// Resolve Promises
getWood()
    .then(() => { return buildBoat(); })
    .then(() => { return sailTheOcean(); })
    .then(() => { console.log('The sailor returns!');
});
```

Fetch

1. Generic definition of Request and Response objects (and other things involved with network requests)
2. It returns a Promise that resolves to the Response to that request
3. Browser API -> Not available out of it
4. Default GET

```
ProjectService.updateProjectFunctions(this.projectId, this.project)
    .then(resp => {
        this.updateProject(resp.data)
    })
    .catch(this.httpError)
    .finally(() => {
        console.log(`It's over`)
    })
```

Async and Await

```
async fetchBooks(volumns) {
    const googleapis = `https://www.googleapis.com/books/v1/volumes?q=volumns:${volumns}&maxResults=15`;
    const res = await fetch(googleapis);
    const data = await res.json();
    this.books = data.items.map(item => {
      item.volumeInfo.imageLinks.thumbnail = item.volumeInfo.imageLinks.thumbnail.replace(
        "http://",
        "https://"
      );
      return item;
    });
  }
```

```
function printHello(){
console.log("Hello world");
}
setTimeout(printHello,0);
console.log("Executed first!");
```

Arrays

1. From
2. Flat.
3. Flat Map
4. Include
5. Find
6. FindIndex

Arrays - From

Creates a new, shallow-copied Array instance from an array-like or iterable object

arrayLike
An array-like or iterable object to convert to an array.

mapFn Optional
Map function to call on every element of the array.

thisArg Optional
Value to use as this when executing mapFn.

```
const set = new Set(['foo', 'bar', 'baz', 'foo']);
Array.from(set); // [ "foo", "bar", "baz" ]
```

```
const mapper = new Map([['1', 'a'], ['2', 'b']]);
Array.from(mapper.values()); // ['a', 'b'];
```

Arrays - Flat

1. Allow to flat multidimensional array structures.
2. It's not recursive by default
3. Receive the level of deepness as a parameter

```
const arr1 = [1, 2, [3, 4]]; arr1.flat(); // [1, 2, 3, 4]
```

```
const arr2 = [1, 2, [3, 4, [5, 6]]]; arr2.flat(); // [1, 2, 3, 4, [5, 6]]
```

```
const arr3 = [1, 2, [3, 4, [5, 6]]]; arr3.flat(2); // [1, 2, 3, 4, 5, 6]
```

```
const arr4 = [1, 2, [3, 4, [5, 6, [7, 8, [9, 10]]]]];
arr4.flat(Infinity); // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Arrays - FlatMap

Maps each element using a mapping function, then flattens the result into a new array

callback

Function that produces an element of the new Array, taking three arguments:

currentValue

The current element being processed in the array.

index(Optional)

The index of the current element being processed in the array.

array(Optional)

The array map was called upon.

thisArg(Optional)

Value to use *as* this when executing callback.

```
totalAllocationDayCount() {
 return this.project.projectFunctions
   .flatMap(pFunction => pFunction.userAllocations)
   .reduce((acc, allocation) => acc + Number(allocation.dayCount), 0)
}
```

Arrays - Includes

Determines whether an array includes a certain value among its entries, returning true or false as appropriate.

```
const pets = ['cat', 'dog', 'bat'];

console.log(pets.includes('cat'));
// expected output: true
```

```
if (this.filesView.map((file) => file.name).includes(file.name)) {
 return
}
```

Arrays - Find

Returns the value of the first element in the provided array that satisfies the provided testing function.

```
const array1 = [5, 12, 8, 130, 44];

const found = array1.find(element => element > 10);

console.log(found);
// expected output: 12
```

```
findProduct(productNumber) {
 if (!this.products.length) {
   return []
 }
 return this.products.find((product) => product.productNumber === productNumber)
}
```

Arrays - FindIndex

Returns the index of the first element in the array that satisfies the provided testing function. Otherwise, it returns `-1`, indicating that no element passed the test.

```
const array1 = [5, 12, 8, 130, 44];

const isLargeNumber = (element) => element > 13;

console.log(array1.findIndex(isLargeNumber));
// expected output: 3
```

Arrow Functions

1. Shorter Syntax
2. No binding of this.
3. The this scope with arrow functions is inherited from the execution context.
4. Arrow Functions are Always Anonymous Functions

Shall we replace all our call with arrows?

```
document.addEventListener('click', function () {
    console.log(this);
});

document.addEventListener('click', () => console.log(this));
var person = {
    name: 'Anna',
    lastname: 'Jones',
    fullname: () => {
        console.log(`My name is: ${this.name} + ${this.lastname}`);
    }
};
person.fullname();
```

Templates Literals

They are string literals allowing embedded expressions. You can use multi-line strings and string interpolation features with them.

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

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

Tagged Templates

```
function doSomething(strings, ...rest) {
    return "Hello";
}

const name = "Kelvin";
const food = "Rice";

const sentence = doSomething`My name is ${name} and I love eating ${food}`;
```

Rest Parameter

Allows us to represent an indefinite number of arguments as an array.

Spread syntax??

```
function f(...[a, b, c]) { return a + b + c; }
```

```
function myFun(a, b, ...manyMoreArgs) { console.log("a", a)
console.log("b", b) console.log("manyMoreArgs", manyMoreArgs) }
myFun("one", "two", "three", "four", "five", "six") // a, one //
b, two // manyMoreArgs, [three, four, five, six]
```

Destructuring

Decomposing structures into their individual parts allowing you to assign the properties of an array or object to variables

```
var first = someArray[0];
var second = someArray[1];
var third = someArray[2];
```

```
var [first, second, third] = someArray;
```

```
var [, , third] = ["foo", "bar", "baz"];
console.log(third);
// "baz"
```

Destructuring Objects

```
var robotA = { name: "Bender" };
var robotB = { name: "Flexo" };

var { name: nameA } = robotA;
var { name: nameB } = robotB;

console.log(nameA);
// "Bender"
console.log(nameB);
// "Flexo"
```

```
var { foo, bar } = { foo: "lorem", bar:
"ipsum" };
console.log(foo);
// "lorem"
console.log(bar);
// "ipsum"
```

Practical applications of destructuring

Function parameter definitions

```
function removeBreakpoint({ url, line, column }) {
  // ...
}
```

Multiple return values

```
function returnMultipleValues() {
  return {
    foo: 1,
    bar: 2
  };
}
var { foo, bar } = returnMultipleValues();
```

Import names from modules

```
import { DatePicker, setupCalendar } from 'v-calendar'
```