Module Introduction

This document will essentially talk about the recent and state of the art developments in the way javascript applications are developed. Modern React applications are built using the latest ES6 syntax and is amazing in many ways.

Let & Const

The **let** statement declares a block scope local variable, optionally initializing it to a value.

The **let** keyword allows you to declare variables that are limited in scope to the **block, statement** or **expression** on which it is used. This is unlike the **var** keyword, which defines a variable globally, or locally to an entire function regardless of the block scope.

```
function varTest() {
  var x = 1;
  if (true) {
   // over-rides the previous var variable
    var x = 2;  // same variable!
    console.log(x);  // 2
  }
  console.log(x);  // 2
}

function letTest() {
  let x = 1;
  if (true) {
    let x = 2;  // different variable
    console.log(x);  // 2
  }
  // as let makes a different variable
  // the following console.log emits
  // a different value of the variable
  console.log(x);  // 1
}
```

At the top level of programs and functions, **let,** unlike **var**, does not create a property on the global object. For example:

```
var x = 'global';
let y = 'global';
console.log(this.x); // "global"
console.log(this.y); // undefined
```

Another good use of the **let** keyword is to emulate private variables.

In dealing with constructors, it is possible to use the **let** binding to share one or more private members without using closures.

```
var Thing;
{
  let privateScope = new WeakMap();
  let counter = 0;

  Thing = function() {
    this.someProperty = 'foo';

    privateScope.set(this, {
      hidden: ++counter,
    });
  };

  Thing.prototype.showPublic = function() {
    return this.someProperty;
  };

  Thing.prototype.showPrivate = function() {
    return privateScope.get(this).hidden;
  };
}
```

```
console.log(typeof privateScope);
// "undefined"

var thing = new Thing();

console.log(thing);
// Thing {someProperty: "foo"}

thing.showPublic();
// "foo"

thing.showPrivate();
// 1
```

The concept of **WeakMap**()

A WeakMap is a map (dictionary) where the **keys** are weak - that is, if all references to the *key* are lost and there are no more references to the value - the *value* can be garbage collected. Let's show this first through examples, then explain it a bit and finally finish with real use.

Let's say I'm using an API that gives me a certain object:

```
var obj = getObjectFromLibrary();
```

Now, I have a method that uses the object:

```
function useObj(obj){
  doSomethingWith(obj);
}
```

I want to keep track of how many times the method was called with a certain object and report if it happens more than N times. Naively one would think to use a Map:

```
var map = new Map(); // maps can have object keys
function useObj(obj){
   doSomethingWith(obj);
   var called = map.get(obj) || 0;
   called++; // called one more time
   if(called > 10) report(); // Report called more than 10 times
   map.set(obj, called);
}
```

This works, but it has a memory leak - we now keep track of every single library object passed to the function which keeps the library objects from ever being garbage collected. Instead - we can use a `WeakMap`:

```
var map = new WeakMap(); // create a weak map
function useObj(obj){
   doSomethingWith(obj);
   var called = map.get(obj) || 0;
   called++; // called one more time
   if(called > 10) report(); // Report called more than 10 times
   map.set(obj, called);
}
```

And the memory leak is gone.

Const

Constants are block-scoped, much like variables defined using the let statement. The value of a constant cannot change through reassignment, and it can’t be redeclared.

This declaration creates a constant whose scope can be either global or local to the block in which it is declared. Global constants do not become properties of the window object, unlike var variables. An initializer for a constant is required; that is, you must specify its value in the same statement in which it's declared (which makes sense, given that it can't be changed later).

The const declaration creates a read-only reference to a value. It does not mean the value it holds is immutable, just that the variable identifier cannot be reassigned. For instance, in the case where the content is an object, this means the object's contents (e.g., its properties) can be altered.

All the considerations about the "temporal dead zone"(https://stackoverflow.com/questions/33198849/what-is-the-temporal-dead-zone) apply to both let and const.

A constant cannot share its name with a function or a variable in the same scope.

```
// NOTE: Constants can be declared with uppercase or lowercase, but a common
// convention is to use all-uppercase letters.

// define MY_FAV as a constant and give it the value 7
const MY_FAV = 7;

// this will throw an error - Uncaught TypeError: Assignment to constant variable.
MY_FAV = 20;

// MY_FAV is 7
console.log('my favorite number is: ' + MY_FAV);
```

```
// trying to redeclare a constant throws an error -  Uncaught SyntaxError: Identifier 'MY_FAV' has already been declared
const MY_FAV = 20;

// the name MY_FAV is reserved for constant above, so this will fail too
var MY_FAV = 20;

// this throws an error too
let MY_FAV = 20;

// it's important to note the nature of block scoping
if (MY_FAV === 7) {
    // this is fine and creates a block scoped MY_FAV variable
    // (works equally well with let to declare a block scoped non const variable)
    let MY_FAV = 20;

    // MY_FAV is now 20
    console.log('my favorite number is ' + MY_FAV);

    // this gets hoisted into the global context and throws an error
    var MY_FAV = 20;
}

// MY_FAV is still 7
console.log('my favorite number is ' + MY_FAV);

// throws an error - Uncaught SyntaxError: Missing initializer in const declaration
const FOO;

// const also works on objects
const MY_OBJECT = {'key': 'value'};
```

```
// Attempting to overwrite the object throws an error - Uncaught TypeError: Assignment to constant variable.
MY_OBJECT = {'OTHER_KEY': 'value'};

// However, object keys are not protected,
// so the following statement is executed without problem
MY_OBJECT.key = 'otherValue'; // Use Object.freeze() to make object immutable

// The same applies to arrays
const MY_ARRAY = [];
// It's possible to push items into the array
MY_ARRAY.push('A'); // ["A"]
// However, assigning a new array to the variable throws an error - Uncaught TypeError: Assignment to constant variable.
MY_ARRAY = ['B'];
```

Summary for let and const

let and const basically replace var . You use let instead of var and const instead of var if you plan never re-assigning this "variable" (effectively turning it into a constant therefore).

ES6 Arrow Functions

Arrow functions are a different way of creating functions in javascript. Besides a shorter syntax, they offer advantages when it comes to keeping the scope of **this** keyword.

Basic Syntax

```
(param1, param2, …, paramN) => { statements }

(param1, param2, …, paramN) => expression

// equivalent to: => { return expression; }



// Parentheses are optional when there's only one parameter name:

(singleParam) => { statements }

singleParam => { statements }



// The parameter list for a function with no parameters should be
written with a pair of parentheses.

() => { statements }
```

Advanced Syntax

```


// Parenthesize the body of function to return an object literal
expression:

params => ({foo: bar})


```

```
// Rest parameters and default parameters are supported
(param1, param2, ...rest) => { statements }

(param1 = defaultValue1, param2, …, paramN = defaultValueN) => {

statements }



// Destructuring within the parameter list is also supported
var f = ([a, b] = [1, 2], {x: c} = {x: a + b}) => a + b + c;

f(); // 6
```

Two factors influenced the introduction of arrow functions: shorter functions and no existence of this keyword.

Shorter Functions

```
var elements = [
  'Hydrogen',
  'Helium',
  'Lithium',
  'Beryllium'
];

elements.map(function(element) {
  return element.length;
}); // this statement returns the array: [8, 6, 7, 9]

// The regular function above can be written as the arrow function
below
elements.map((element) => {
  return element.length;
```

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

// When there is only one parameter, we can remove the surrounding
parenthesies:
elements.map(element => {
  return element.length;
}); // [8, 6, 7, 9]

// When the only statement in an arrow function is `return`, we
can remove `return` and remove
// the surrounding curly brackets
elements.map(element => element.length); // [8, 6, 7, 9]

// In this case, because we only need the length property, we can
use destructuring parameter:
// Notice that the `length` corresponds to the property we want to
get whereas the
// obviously non-special `lengthFooBArX` is just the name of a
variable which can be changed
// to any valid variable name you want
elements.map(({ length :lengthFooBArX }) => lengthFooBArX); // [8,
6, 7, 9]

// This destructuring parameter assignment can also be written as
seen below. However, note that in
// this example we are not assigning `length` value to the made up
property. Instead, the literal name
// itself of the variable `length` is used as the property we want
to retrieve from the object.
elements.map(({ length }) => length); // [8, 6, 7, 9]
```

Arrow functions are a shorter way of creating functions in Javascript. Besides shorter syntax, they offer advantages when it comes to keeping the scope of the this keyword.

Arrow functions syntax may look strange but its actually simple

```
function callMe(name) {
     console.log(name);
}
```

can also be written as :

```
const callMe = function(name) {
     console.log(name);
}
```

becomes :

```
const callMe = (name) => {
     console.log(name);
}
```

Important

1. When having no arguments, you have to use empty paranthesis in the function declaration:

```
     const callMe = () => {
          console.log(name);
     }
```

2. When having exactly one argument, you may omit the parantheses:

```
     const callMe = name => {
          console.log(name);
     }
```

3. When just returning the value, you can use the following shortcut:

```
     const returnMe = name => name
```

4. The above syntax in the third point is equal to :

```
     const returnMe = name => {
          return name;
     }
```

Exports & Imports

In react projects, and actually in all modern javascript projects, you split your code across multiple javascript files – so called modules. You do this, to keep each file / module focused and manageable.

To access the functionality in anothe file, you first need to **export** (to make it available) and **import** (to get access) statements.

You got two different types of exports :

1. **Default (unnamed)**

2. **Named**

ES6 provides us to import a module and use it in other files. Strictly speaking in React terms, one can use stateless components in other components by exporting the components from their respective modules and using it in other files.

1. **Named Export**

Named exports are useful to export several values. During the import, one will be able to use the same name to refer to the corresponding value.

```
// imports
// ex. Importing a single statement named export
import { MyComponent } from “./MyComponent”;

// example : importing multiple named exports
import { MyComponent, MyComponent2 } from “.MyComponent”;

// exports from ./MyComponent.js file
export const MyComponent = () => {}
export const MyComponent2 = () => {}

// import all named components into one object
import * as MainComponents from “./MyComponent”;
// use MainComponents.MyComponent to use the component
```

2. **Default Export**

Concerning the default export, there is only a single default export per module. A default export can be a function, a class, an object or anything else.

```
// import
import MyDefaultComponent from “./MyDefaultExport”;

// export
const MyComponent = () = {}

// exporting default component
export default MyComponent;
```

A file can contain one default and an unlimited amount of named exports. You can also mix the one default export with any amount of named exports in the same file.

Classes

Javascript classes are primarily syntactical sugar over Javascript's existing prototype-based inheritance. The class syntax *does not* introduce a new object-oriented inheritance model to Javascript.

Classes are a feature which basically replace constructor functions and prototypes. You can define blueprints for javascript objects with them.

```
// defining a sample class in javascript
// using 'this' is the old way of doing it.
class Person {
     constructor() {
          this.name = "Mohit!";
     }
}
const person = new Person();
console.log(person.name); // prints 'Mohit!'

// In modern javascript projects, you can use the following,
class Person {
     name = 'Mohit';
}
const person = new Person();
console.log(person.name);

// define methods like this
class Person {
     name = 'Mohit';
     printMyName() {
          console.log(this.name);
     }
}
```

```
// or use arrow functions within classes
class Person {
     name = ‘Mohit’;
     const printMyName = () =>  {
          console.log(this.name);
     }
}
// you can also use inheritance when using classes:
class Human {
     species = “Human”;
}
class Person extends Human {
     name = “Mohit”;
     printMyName = () => {
          console.log(this.name);
     }
}
const person = new Person();
person.printMyName();
console.log(person.species);
```

https://stackoverflow.com/questions/34517581/access-modifiers-private-protected-in-es6

Spread & Rest Operator

1. As a pre-requisite – read about shallow copy and deep copy of objects in javascript:

https://we-are.bookmyshow.com/understanding-deep-and-shallow-copy-in-javascript-13438bad941c

The spread and the rest operators actually use the same syntax : **...**

Yes, it uses just 3 dots. Its usage determines whether you're using it as the spread or rest operator.

Using the spread operator

The spread operator allows you to pull elements out of an array (=> split the array into a list of its elements) or pull the properties out of an object. Here are 2 examples:

```
// using the spread operators with array
const oldArray = [1,2,3];
const newArray = [...oldArray,4,5]; // newArray = 1,2,3,4,5

// simple javascript object notation
const oldObject = {
     name : 'Mohit'
};
// we will now use the old object
// in creating the new object
const newObject = {
     ...oldObject,
     age : '25'
}
```

The spread operator is extremely useful for cloning arrays and objects. Since both are reference types (array and objects), copying them safely can be tricky. With the spread operator we get a shallow copy rather than a deep one.

Using the rest operator

With rest parameters, we can gather any number of arguments into an array and do what we want to do with them. Example:

```
function add(x,y) {
     return x+y;
}
// returns 3 – rest of the arguments are ignored.
add(1,2,3,4,5);

// using the rest operator
function add(...args) {
     let result = 0;
     for ( let arg of args ) result += arg;
     return result;
}
```

Note: Rest parameters have to be at the last argument. This is because it collects all remaining/ excess arguments into an array. So having a function definition like this does not make sense and it errors out.

Destructuring (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment)

Destructuring allows you to easily access the values of arrays or objects and assign them to variables.

```
Const array = [1,2,3];
const [a,b] = array;
console.log(a); // prints 1
console.log(b); // prints 2
console.log(c); // prints [1,2,3]
```

Destructuring for an object

```
const myObj = {
     name : ‘Max’,
     age : 28
};

// destructuring uses object keys to access info
const {name} = myObj;
console.log(name); // prints ‘Max’
console.log(age);  // prints undefined
```

Destructuring is very useful while working with function arguments :

```
// we only want to print the name in the object
// so we will pass the whole object
// but we will just extract the name in the object
// by destructuring the object

const printName = ({name}) => {
     console.log(name);
}
printName({name : ‘rafeeq’, age : 39});
```

Refreshing javascript array functions

Array definition and functions can be found here : https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

`map()` => https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map

```
const numbers = [1,2,3];
const doubleNumArray = numbers.map((number)=> {
          return 2*number;
});
```

`find()` => https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/find

The **find()** function method returns the value of the first element in the array that satisfies the provided testing function. Otherwise, undefined is returned.

```
var array1 = [5,12,8,130,44];
var found = array1.find((number) => {
     // find all numbers greater than 10
     return number > 10;
});
console.log(found);
```

`findIndex()` =>https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/findIndex

The findIndex() method 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.

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

function isLargeNumber(element) {
  return element > 13;
}

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

`filter()` =>https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter

The filter() method creates a new array with all elements that pass the test implemented by the provided function.

```
var words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];

const result = words.filter(word => word.length > 6);

console.log(result);
// expected output: Array ["exuberant", "destruction", "present"]
```

`reduce()` =>https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce?v=b

The reduce() method executes a reducer function (that you provide) on each element of the array, resulting in a single output value.

```
const array1 = [1, 2, 3, 4];

const reducer = (accumulator, currentValue) => accumulator + currentValue;

// 1 + 2 + 3 + 4
console.log(array1.reduce(reducer));
// expected output: 10

// 5 + 1 + 2 + 3 + 4
console.log(array1.reduce(reducer, 5));
// expected output: 15
```

`concat()` =>https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/concat?v=b

`slice()` =>https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/slice

`splice()` =>https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/splice

The splice() method changes the contents of an array by removing or replacing existing elements and/or adding new elements in place.

```
var months = ['Jan', 'March', 'April', 'June'];
months.splice(1, 0, 'Feb');
// inserts at index 1
console.log(months);
```

```
// expected output: Array ['Jan', 'Feb', 'March', 'April', 'June']

months.splice(4, 1, 'May');
// replaces 1 element at index 4
console.log(months);
// expected output: Array ['Jan', 'Feb', 'March', 'April', 'May']
```