ES6 Features of JavaScript:

1.let and const:

There are mainly three types of scope:

- Block Scope
- Functional Scope
- Global Scope

SCOPE:

- variables declared inside a `{ }` block to be accessed outside of the block, we need to declare them using the `let` or `const` keywords. Variables declared with the `var` keyword inside the `{ }` block **are** accessible outside of the block too.
- Do not use the `var` keyword inside a block (block scope). Always use `let` and `const` instead.
- The variable declared with `var` inside a function is not accessible outside of it. The keyword `var` has function-scope.
- `var`: The functional scope level
- `let`: The block scope level
- `const`: The block scope level

Reassign a New Value

- Once you've declared a variable with `var` or `let`, you **can** reassign a new value to the variable in your programming flow.
- But with `const`, you **can't** reassign a new value at all.
- When an object is declared and assigned a value with `const`, you can still change the value of its `properties`. But you can not reassign another object value to the same variable.

What Happens When You Access a Variable Before Declaring?

Var :Undefined

Let, const : Reference error

```
// Traditional Anonymous Function
function (a){
  return a + 100;
}
```

2.Arrow Functions:

// Arrow Function Break Down

//1. Remove the word "function" and place arrow between the argument and opening body bracket

```
(a) => {
  return a + 100;
}
```

// 2. Remove the body braces and word "return" -- the return is implied.

(a) => a + 100;

// 3. Remove the argument parentheses

a => a + 100;

The { braces } and (parentheses) and "return" are required in some cases. For example, if you have multiple arguments or no arguments, you'll need to re-introduce parentheses around the arguments:

```
(a, b) => {
  let chuck = 42;
  return a + b + chuck;
}
```

3. Template Literals:

- In ES6, we can use a new syntax `${PARAMETER}` inside of the back-ticked string.

```
var name = `Your name is ${firstName} ${lastName}.`
```

In ES5, we have to break strings like below.

```
var name = 'Your name is ' + firstName + ' ' + lastName + '.'
```

Multi-line strings:

Template literals allows multiline strings:

Example(In ES6)

```
let poemData = `Johny Johny Yes Papa,
                Eating sugar?  No, papa!
                Telling lies? No, papa!
                Open your mouth Ah, ah, ah!`

In ES5
     var poemData = 'Johny Johny Yes Papa,\n'
                  + 'Eating sugar?  No, papa!\n'
                  + 'Telling lies? No, papa!\n'
                  + 'Open your mouth Ah, ah, ah!'
```

Interpolation

Template literals provide an easy way to interpolate variables and expressions into strings.

The method is called string interpolation.

The syntax is:

```
     ${...}
```

4. Object Literals:

Object literals make it easy to quickly create objects with properties inside the curly braces.
In ES6:

```
function getLaptop(make, model, year) {
  return {
      make,
      model,
      year
  }
}

getLaptop("Apple", "MacBook", "2015");
```

In ES5:

```
function getLaptop(make, model, year) {
  return {
    make: make,
    model: model,
    year: year
  }
}
getLaptop("Apple", "MacBook", "2015");
```

5. Default Parameters:

In ES6, we can put the default values right in the signature of the

functions.

```
var calculateArea = function(height = 50, width = 80) {
```

```
   // write logic

   ...

}
```

In ES5, we were using logic OR operator.

```
var calculateArea = function(height, width) {

  height =  height || 50;

  width = width || 80;

  // write logic

   ...

}
```

Default parameters also came along with ES6. It allows you to set default values for your function parameters if no value is passed or if undefined is passed.

Example:1

```
function add(a=3, b=5) {
        return a + b;
    }

    add(4,2) // 6
    add(4) // 9
    add() // 8
```

Example:2

```
function createArray(a = 10, b) {
        return [a,b];
    }

    createArray() // [10, undefined]
    createArray(5) //[5, undefined]
```

Example:3

You can also set a function as the default parameter.

```
function createA() {
        return 10;
    }

    function add(a = createA(), b=5) {
        return a + b;
    }

    add() // 15
```

Note that when doing this, the function cannot be an internal function because the default arguments are evaluated when the function is called. Therefore the following will not work.

```
function add(a = createA(), b=5) {

        function createA() {
        return 10;
        }
        return a + b;
    }

    add() // createA is not defined
```

```
function createA() {
      return 5;
   }

   function add(a = createA(), b = a*2, c = b+3) {
      return a + b + c;
   }

   add() // 28 because 5 + (5*2) + ((5*2) + 3) = 5 + 10 + 13 = 28
   add(2)// 13 because 2 + (2*2) + ((2*2) + 3) = 2 + 4 + 7 = 13
   add(2,3)// 11 because 2 + 3 + (3+3) = 11
   add(2,3,1)//6
```

6. New Methods(forEach, map, filter, etc.):

Map, reduce, and filter are all array methods in JavaScript. Each one will iterate over an array and perform a transformation or computation. Each will return a new array based on the result of the function.

Map

The `map()` method is used for creating a new array from an existing one, applying a function to each one of the elements of the first array.

Syntax

```
var new_array = arr.map(function callback(element, index, array) {

    // Return value for new_array

}[, thisArg])
```

In the callback, only the array `element` is required. Usually some action is performed on the value and then a new value is returned.

Example

In the following example, each number in an array is doubled.

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

const doubled = numbers.map(item => item * 2);

console.log(doubled); // [2, 4, 6, 8]
```

Filter

The `filter()` method takes each element in an array and it applies a conditional statement against it. If this conditional returns true, the element gets pushed to the output array. If the condition returns false, the element does not get pushed to the output array.

Syntax

```
var new_array = arr.filter(function callback(element, index, array) {

    // Return true or false

}[, thisArg])
```

The syntax for `filter` is similar to `map`, except the callback function should return `true` to keep the element, or `false` otherwise. In the callback, only the `element` is required.

Examples

In the following example, odd numbers are "filtered" out, leaving only even numbers.

```
const numbers = [1, 2, 3, 4];
const evens = numbers.filter(item => item % 2 === 0);
console.log(evens); // [2, 4]
```

In the next example, `filter()` is used to get all the students whose grades are greater than or equal to 90.

```
const students = [
  { name: 'Quincy', grade: 96 },
  { name: 'Jason', grade: 84 },
  { name: 'Alexis', grade: 100 },
  { name: 'Sam', grade: 65 },
  { name: 'Katie', grade: 90 }
];
const studentGrades = students.filter(student => student.grade >= 90);
return studentGrades; // [ { name: 'Quincy', grade: 96 }, { name: 'Alexis', grade: 100 }, { name:
'Katie', grade: 90 } ]
```

Reduce

The `reduce()` method reduces an array of values down to just one value. To get the output value, it runs a reducer function on each element of the array.

Syntax

```
arr.reduce(callback[, initialValue])
```

The `callback` argument is a function that will be called once for every item in the array. This function takes four arguments, but often only the first two are used.

- *accumulator* - the returned value of the previous iteration
- *currentValue* - the current item in the array
- *index* - the index of the current item
- *array* - the original array on which reduce was called
- The `initialValue` argument is optional. If provided, it will be used as the initial accumulator value in the first call to the callback function.

Examples

The following example adds every number together in an array of numbers.

```
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce(function (result, item) {
  return result + item;
}, 0);
console.log(sum); // 10
```

In the next example, `reduce()` is used to transform an array of strings into a single object that shows how many times each string appears in the array. Notice this call to reduce passes an empty object `{}` as the `initialValue` parameter. This will be used as the initial value of the accumulator (the first argument) passed to the callback function.

```
var pets = ['dog', 'chicken', 'cat', 'dog', 'chicken', 'chicken', 'rabbit'];

var petCounts = pets.reduce(function(obj, pet){
    if (!obj[pet]) {
        obj[pet] = 1;
    } else {
        obj[pet]++;
    }
    return obj;
}, {});

console.log(petCounts);

/*
Output:
 {
    dog: 2,
    chicken: 3,
    cat: 1,
    rabbit: 1
 }
 */
```

forEach

.forEach(), is used to execute the same code on every element in an array but does not change the array and it returns undefined.

Example:

In the example below we would use .forEach() to iterate over an array of food and log that we would want to eat each of them.

```
let food = ['mango','rice','pepper','pear'];

food.forEach(function(foodItem){

   console.log('I want to eat '+foodItem);

});
```

Running this on your console;

```
SW registered
> let food = ['mango','rice','pepper','pear'];
    food.forEach(function(foodItem){
        console.log('I want to eat '+foodItem);
    });
I want to eat mango
I want to eat rice
I want to eat pepper
I want to eat pear
< undefined
>
```

7.Promise:

Promises are a way to implement async programming in JavaScript(ES6). A Promise will become a container for future value. Like if you order any food on any site to deliver it to your place that order record will be the promise and the food will be the value of that promise. So the order details are the container of the food you ordered.
Let's explain it with another example. You order an awesome camera online. After your order is placed you receive a receipt of the order. That receipt is a Promise that your order will be delivered to you. The receipt is a placeholder for the future value namely the camera.

Need of Promises: The Callbacks are great when dealing with basic cases. But while developing a web application where you have a lot of code. Callbacks can be great trouble. In complex cases, every callback adds a level of nesting which can make your code really messy and hard to understand. This excessive nesting of callbacks is often termed as Callback Hell.

Example: Callback Hell

```
f1(function(x){
    f2(x, function(y){
        f3(y, function(z){
            ...
        });
    });
});
```

To deal with this problem, we use Promises instead of callbacks.

Making Promises: A Promise is basically created when we are unsure of whether or not the assigned task will be completed. The Promise object represents the eventual completion (or failure) of an async(asynchronous) operation and its resulting value. As the name suggests a Promise is either kept or broken.

A Promise is always in one of the following states:

fulfilled: Action related to the promise succeeded.

rejected: Action related to the promise failed.

pending: Promise is still pending i.e not fulfilled or rejected yet.

settled: Promise has fulfilled or rejected

Syntax:

```
const promise = new Promise((resolve,reject) => {....});
```

Example:

```
const myPromise = new Promise((resolve, reject) => {
   if (Math.random() > 0) {
      resolve('Hello, I am positive number!');
   }
   reject(new Error('I failed some times'));
})
```

Callbacks to Promises: There are two types of callbacks which are used for handling promises .then() and .catch(). It can be used for handling promises in case of fulfillment (promise is kept) or rejection (promise is broken).

.then(): Invoked when a promise is kept or broken. It can be chained to handle the fulfillment or rejection of a promise. It **takes in two functions as parameters**. The **first** one is **invoked if the promise is fulfilled** and the **second** one(**optional) is invoked if the promise is rejected**.

Example: Handling Promise rejection using .then()

```
var promise = new Promise(function(resolve, reject) {
   resolve('Hello, I am a Promise!');
})

promise.then(function(promise_kept_message) {
```

```
        console.log(promise_kept_message);
           }, function(error) {

        // This function is invoked this time
        // as the Promise is rejected.
        console.log(error);   })
```

.catch() can be used for handling the errors(if any). It takes only one function as a parameter which is used to handle the errors (if any).
Example: Handling Promise rejection(or errors) using .catch()

```
const myPromise = new Promise((resolve, reject) => {
    if (Math.random()  > 0) {
        console.log('resolving the promise ...');
        resolve('Hello, Positive :)');
    }
    reject(new Error('No place for Negative here :('));
});

const Fulfilled = (fulfilledValue) => console.log(fulfilledValue);
const Rejected = (error) => console.log(error);
myPromise.then(Fulfilled, Rejected);

myPromise.then((fulfilledValue) => {
    console.log(fulfilledValue);
}}).catch(err => console.log(err));
```

8. Classes:

We can create a class in ES6 using the “class” keyword. Class creation and usage in ES5 was a pain in the rear, because there wasn’t a keyword class.

```
class Profile {
```

```
  constructor(firstName, lastName = '') { // class constructor
     this.firstName = firstName;
     this.lastName = lastName;
  }

  getName() { // class method
    console.log(`Name: ${this.firstName} ${this.lastName}`);
  }
}
let profileObj = new Profile('Kavisha', 'Talsania');
profileObj.getName(); // output: Name: Kavisha Talsania
```

9. Modules:

An ES6 module is a JavaScript file that executes in strict mode only. It means that any variables or functions declared in the module won't be added automatically to the global scope.

Exporting

To export a variable, a function, or a class, you place the export keyword in front of it as follow

```
// log.js
export let message = 'Hi';

export function getMessage() {
  return message;
}

export function setMessage(msg) {
  message = msg;
}
```

```
export class Logger {
}
```

In this example, we have the log.js module with a variable, two functions, and one class. We used the export keyword to exports all identifiers in the module.

Note that the export keyword requires the function or class to have a name to be exported. You can't export an anonymous function or class using this syntax.

JavaScript allows you to define a variable, a function, or a class first then export it later as follows:

```
// foo.js
function foo() {
   console.log('foo');
}

function bar() {
  console.log('bar');
}
export foo;
```

In this example, we defined the `foo()` function first and then exported it. Since we didn't export the `bar()` function, we couldn't access it in other modules. The `bar()` function is inaccessible outside the module or we say it is private.

Importing

Once you define a module with exports, you can access the exported variables, functions, and classes in another module by using the import keyword. The following illustrates the syntax:

```
import { what, ever } from './other_module.js';
```

In this syntax:

- First, specify what to import inside the curly braces, which are called bindings.
- Then, specify the module from which you import the given bindings.

Note that when you import a binding from a module, the binding behaves like it was defined using const. It means you can't have another identifier with the same name or change the value of the binding.
See the following example:

```
// greeting.js
export let message = 'Hi';

export function setMessage(msg) {
  message = msg;
}
```

When you import the message variable and setMessage() function, you can use the setMessage() function to change the value of the message variable as shown below:

```
// app.js
import {message, setMessage } from './greeting.js';
console.log(message); // 'Hi'

setMessage('Hello');
console.log(message); // 'Hello'
```

However, you can't change the value of the message variable directly. The following expression causes an error:

```
message = 'Hallo'; // error
```

Behind the scenes, when you called the setMessage() function. JavaScript went back to the greeting.js module and executed code in there and changed the message variable. The change was then automatically reflected on the imported message binding.

The message binding in the app.js is the local name for exported message identifier. So basically the message variables in the app.js and greeting.js modules aren't the same.

Import a single binding

Suppose you have a module with the foo variable as follows:

```
// foo.js
export foo = 10;
```

Then, in another module, you can reuse the foo variable:

```
// app.js
import { foo } from './foo.js';
console.log(foo); // 10;
```

However, you can't change the value of foo. If you attempt to do so, you will get an error:

```
foo = 20; // throws an error
```

Import multiple bindings

Suppose you have the cal.js module as follows:

```
// cal.js
export let a = 10,
        b = 20,
        result = 0;

export function sum() {
  result = a + b;
  return result;
}

export function multiply() {
  result = a * b;
  return result;
}
```

And you want to import these bindings from the cal.js, you can explicitly list them as follows:

```
import {a, b, result, sum, multiply } from './cal.js';
sum();
console.log(result); // 30

multiply();
console.log(result); // 200
```

Import an entire module as an object

To import everything from a module as a single object, you use the asterisk (*) pattern as follows:

```
import * as cal from './cal.js';
```

In this example, we imported all bindings from the cal.js module as the cal object. In this case, all the bindings become properties of the cal object, so you can access them as shown below:

```
cal.a;
cal.b;
cal.sum();
```

This import is called *namespace import*.

It's important to keep in mind that the imported module executes *only once* even import it multiple times. Consider this example:

```
import { a } from './cal.js';
import { b } from './cal.js';
import {result} from './cal.js';
```

After the first import statement, the cal.js module is executed and loaded into the memory, and it is reused whenever it is referenced by the subsequent import statement.

Limitation of import and export statements

Note that you must use the import or export statement *outside* other statements and functions. The following example causes a SyntaxError:

```
if( requiredSum ) {
  export sum;
}
```

Because we used the export statement inside the if statement. Similarly, the following import statement also causes a SyntaxError:

```
function importSum() {
```

```
  import {sum} from './cal.js';
}
```

Because we used the import statement inside a function.

The reason for the error is that JavaScript must *statically* determine what will be exported and imported.

Note that ES2020 introduced the function-like object import() that allows you to dynamically import a module

10.Spread Operator:

The **spread operator** is a new addition to the set of operators in JavaScript ES6. It takes in an iterable (e.g an array) and expands it into individual elements.

The spread operator is commonly used to make shallow copies of JS objects. Using this operator makes the code concise and enhances its readability.

Syntax

The spread operator is denoted by three dots, `...`.

Examples

Since the array data structure is widely used, it will be considered in all the subsequent examples.

1. Copying an array

The `array2` has the elements of `array1` copied into it. Any changes made to `array1` will not be reflected in `array2` and vice versa.

If the simple assignment operator had been used then `array2` would have been assigned a reference to `array1` and the changes made in one array would reflect in the other array which in most cases is undesirable.

```
let array1 = ['h', 'e', 'y'];
let array2 = [...array1];
console.log(array2);
```

Output: `[ 'h', 'e', 'y' ]`

2. Inserting the elements of one array into another

It can be seen that the spread operator can be used to append one array after any element of the second array. In other words, there is no limitation that `baked_desserts` can only be appended at the beginning or the end of the `desserts2` array.

```
let baked_desserts = ['cake', 'cookie', 'donut'];
let desserts = ['icecream', 'flan', 'frozen yoghurt', ...baked_desserts];
console.log(desserts);
//Appending baked_desserts after flan
let desserts2 = ['icecream', 'flan', ...baked_desserts, 'frozen yoghurt'];
console.log(desserts2);
```

```
Output
1.15s
[ 'icecream', 'flan', 'frozen yoghurt', 'cake', 'cookie', 'donut' ]
[ 'icecream', 'flan', 'cake', 'cookie', 'donut', 'frozen yoghurt' ]
```

3. Array to arguments

```
function multiply(number1, number2, number3) {
  console.log(number1 * number2 * number3);
}
let numbers = [1,2,3];
multiply(...numbers);
```

```
Output
1.04s
6
```

Instead of having to pass each element like `numbers[0]`, `numbers[1]` and so on, the spread operators allows array elements to be passed in as individual arguments.

```
//Passing elements of the array as arguments to the Math Object
let numbers = [1,2,300,-1,0,-100];
console.log(Math.min(...numbers));
```

```
Output
1.86s
-100
```

11.Rest Operator:

The **rest operator** is used to put the rest of some specific user-supplied values into a JavaScript array.
So, for instance, here is the rest syntax:

```
...yourValues
```

The text after the rest operator references the values you wish to encase inside an array. You can only use it before the last parameter in a function definition.

How Does the Rest Operator Work in a Function?

In JavaScript functions, rest gets used as a prefix of the function's last parameter.

Example:

```
// Define a function with two regular parameters and one rest parameter:
function myBio(firstName, lastName, ...otherInfo) {
  return otherInfo;
}
```

The rest operator (`...`) instructs the computer to add whatever `otherInfo` (arguments) supplied by the user into an array. Then, assign that array to the `otherInfo` parameter.

As such, we call `...otherInfo` a rest parameter.

Note: Arguments are optional values you may pass to a function's parameter through an invocator.

Example:

```
// Define a function with two regular parameters and one rest parameter:

function myBio(firstName, lastName, ...otherInfo) {
  return otherInfo;
}

// Invoke myBio function while passing five arguments to its parameters:
myBio("Oluwatobi", "Sofela", "CodeSweetly", "Web Developer", "Male");

// The invocation above will return:
["CodeSweetly", "Web Developer", "Male"]
```

In the snippet above, notice that `myBio`'s invocation passed five arguments to the function.

In other words, `"Oluwatobi"` and `"Sofela"` got assigned to the `firstName` and `lastName` parameters.

At the same time, the rest operator added the remaining arguments (`"CodeSweetly"`, `"Web Developer"`, and `"Male"`) into an array and assigned that array to the `otherInfo` parameter.

Therefore, `myBio()` function correctly returned `["CodeSweetly", "Web Developer", "Male"]` as the content of the `otherInfo` rest parameter.

12. Destructuring:

Assuming that you have a function that returns an array of numbers as follows:

```
function getScores() {
  return [70, 80, 90];
}
```

The following invokes the getScores() function and assigns the returned value to a variable:

```
let scores = getScores();
```

To get the individual score, you need to do like this:

```
let x = scores[0],
   y = scores[1],
   z = scores[2];
```

Prior to ES6, there was no direct way to assign the elements of the returned array to multiple variables such as x, y and z.

Fortunately, starting from ES6, you can use the destructing assignment as follows:

```
let [x, y, z] = getScores();

console.log(x); // 70
console.log(y); // 80
console.log(z); // 90
```

The variables x, y and z will take the values of the first, second, and third elements of the returned array.

Note that the square brackets [] look like the array syntax but they are not.

If the getScores() function returns an array of two elements, the third variable will be undefined, like this:

```
function getScores() {
  return [70, 80];
}

let [x, y, z] = getScores();

console.log(x); // 70
console.log(y); // 80
console.log(z); // undefined
```

In case the getScores() function returns an array that has more than three elements, the remaining elements are discarded. For example:

```
function getScores() {
  return [70, 80, 90, 100];
}

let [x, y, z] = getScores();

console.log(x); // 70
console.log(y); // 80
console.log(z); // 90
```

Array Destructuring Assignment and Rest syntax

It's possible to take all remaining elements of an array and put them in a new array by using the rest syntax (...):

```
let [x, y ,...args] = getScores();
console.log(x); // 70
console.log(y); // 80
console.log(args); // [90, 100]
```

The variables x and y receive values of the first two elements of the returned array. And the args variable receives all the remaining arguments, which are the last two elements of the returned array.

Note that it’s possible to destructure an array in the assignment that separates from the variable’s declaration. For example:

```
let a, b;

[a, b] = [10, 20];

console.log(a); // 10

console.log(b); // 20
```

Setting default values

See the following example:

```
function getItems() {

    return [10, 20];

}



let items = getItems();

let thirdItem = items[2] != undefined ? items[2] : 0;



console.log(thirdItem); // 0
```

How it works:

- First, declare the getItems() function that returns an array of two numbers.
- Then, assign the items variable to the returned array of the getItems() function.
- Finally, check if the third element exists in the array. If not, assign the value 0 to the thirdItem variable.

It’ll be simpler with the destructuring assignment with a default value:

```
let [, , thirdItem = 0] = getItems();
```

```
console.log(thirdItem); // 0
```

If the value taken from the array is undefined, you can assign the variable a default value, like this:

```
let a, b;

[a = 1, b = 2] = [10];

console.log(a); // 10

console.log(b); // 2
```

If the getItems() function doesn't return an array and you expect an array, the destructing assignment will result in an error. For example:

```
function getItems() {
    return null;
}

let [x = 1, y = 2] = getItems();
```

Error:

Uncaught TypeError: getItems is not a function or its return value is not iterable

A typical way to solve this is to fallback the returned value of the getItems() function to an empty array like this:

```
function getItems() {
    return null;
}

let [a = 10, b = 20] = getItems() || [];
```

```
console.log(a); // 10

console.log(b); // 20
```

Nested array destructuring

The following function returns an array that contains an element which is another array, or nested array:

```
function getProfile() {
    return [
        'John',
        'Doe',
        ['Red', 'Green', 'Blue']
    ];
}
```

Since the third element of the returned array is another array, you need to use the nested array destructuring syntax to destructure it, like this:

```
let [
    firstName,
    lastName,
    [
        color1,
        color2,
        color3
    ]
] = getProfile();
```

```
console.log(color1, color2, color3); // Red Green Blue
```

Array Destructuring Assignment Applications

Let’s see some practical examples of using the array destructuring assignment syntax.

1) Swapping variables

The array destructuring makes it easy to swap values of variables without using a temporary variable:

```
let a = 10,
    b = 20;


[a, b] = [b, a];


console.log(a); // 20
console.log(b); // 10
```

2) Functions that return multiple values

In JavaScript, a function can return a value. However, you can return an array that contains multiple values, for example:

```
function stat(a, b) {
   return [
      a + b,
      (a + b) / 2,
      a - b
   ]
```

```
}
```

And then you use the array destructuring assignment syntax to destructure the elements of the return array into variables:

```
let [sum, average, difference] = stat(20, 10);

console.log(sum, average, difference); // 30, 15, 10
```

```
(a, b) => {

  let chuck = 42;

  return a + b + chuck;

}
```