# ES6 FEATURES

ES6 or ECMAScript 2015 is the 6th version of the ECMAScript programming language. ECMAScript is the standardization of Javascript which was released in 2015 and subsequently renamed as ECMAScript 2015. ECMAScript and Javascript are both different.

**ECMAScript vs Javascript**

**ECMAScript:** It is the specification defined in ECMA-262 for creating a general-purpose scripting language. In simple terms, it is a standardization for creating a scripting language. It was introduced by ECMA International and is an implementation with which we learn how to create a scripting language.

**Javascript:** A general-purpose scripting language that confirms the ECMAScript specification. It is an implementation that tells us how to use a scripting language.

**ES6**

Javascript ES6 has been around for a few years now, and it allows us to write code in a clever way which basically makes the code more modern and more readable. It's fair to say that with the use of ES6 features we write less and do more, hence the term 'write less, do more' definitely suits ES6.

ES6 introduced several key features like const, let, arrow functions, template literals, default parameters, and a lot more. Let's take a look at them one by one.

1. **Const & let**
2. **Arrow functions**
3. **Template Literals**
4. **Array and Object Destructing**
5. **Default Parametes**
6. **Rest and Spread Operators**
7. **Classes**
8. **Modules**
9. **Promises**

## 1. const , let and var

const is a new keyword in ES6 for declaring variables. const is more powerful than var. Once used, the variable can't be reassigned. In other words, it's an **immutable variable** except when it used with objects.

This is really useful for targeting the selectors. For example, when we have a single button that fires an event, or when you want to select an HTML element in JavaScript, use const instead of var. This is because var is 'hoisted'. It's always preferable to use const when don't want to reassign the variable .

**Example:**

```
// Const
const name = 'Ramu   ';
console.log(name);     // Will print 'Ramu' to the console.

// Trying to reassign a const variable
name = 'Raju';
console.log(name);     // Will give TypeError.
```

let can be reassigned and take new value. It creates a **mutable variable**.

let is the same as const in that both are blocked-scope. It means that the variable is only available within its scope.

Using **let** statement, block scoped statement

**Example:**

```
let x = 10;
if(x==10){
        let x = 20;
        console.log(x); //prints 20
}
console.log(x); //prints 10;
```

Using **var** statement, function scoped statement

**Example:**

```
var x = 10;
if(x==10){
        var x = 20;
        console.log(x); //prints 20
}
console.log(x); //prints 20;
```

## 2. Arrow Functions

Arrow Functions are another interesting feature in ES6. It provides a more concise syntax for writing function expressions by opting out the function and return keywords.

Arrow functions are defined using a new syntax, the fat arrow (=>) notation. Let's see how it looks:

**Example:**

```
// Function Expression
var sum = function(a, b) {
  return a + b;
}
console.log(sum(2, 3)); // 5

// Arrow function
var sum = (a, b) => a + b;
console.log(sum(2, 3)); // 5
```

**Example-2:**

```
// No parameter, single statement
var hello = () => alert('Hello World!');
hello();
// Single parameter, single statement
```

```
var greet = name => alert("Hi " + name + "!");
greet("Peter");

// Multiple arguments, single statement
var multiply = (x, y) => x * y;
alert(multiply(2, 3));
```

## 3. Template Literals

Template literals provide an easy and clean way create multi-line strings and perform string interpolation. Now we can embed variables or expressions into a string at any spot without any hassle.

Template literals are created using back-tick (` `) (grave accent) character instead of the usual double or single quotes. Variables or expressions can be placed inside the string using the ${...} syntax.

**Example:**

```
// Without Template Literal
var name = 'Malla Reddy University';
console.log('Printed without using Template Literal');
console.log('My name is '+ name);
 // With Template Literal
const name1 = ' Malla Reddy University ';
console.log('Printed by using Template Literal');
console.log(`My name is ${name1}`);
```

**Example-2:**

```
let a = 10; let b = 20;
let result = `The sum of ${a} and ${b} is ${a+b}.`;
console.log(result);
```

## 4. Destructuring Assignment

The destructuring assignment is an expression that makes it easy to extract values from arrays, or properties from objects, into distinct variables by providing a shorter syntax.

There are two kinds of destructuring assignment expressions the ***array* and *object*** destructuring assignment.

**The array destructuring assignment**

Prior to ES6, to get an individual value of an array we need to write something like this:

**Example**:

```
// ES5 syntax
var fruits = ["Apple", "Banana"];
var a = fruits[0];
var b = fruits[1];
alert(a); // Apple
alert(b); // Banana
```

In ES6, we can do the same thing in just one line using the array destructuring assignment:

**Example:**

```
// ES6 syntax
let fruits = ["Apple", "Banana"];
let [a, b] = fruits; // Array destructuring assignment
alert(a); // Apple
alert(b); // Banana
```

**The object destructuring assignment**

In ES5 to extract the property values of an object we need to write something like this:

**Example-1:**

```
// ES5 syntax
var person = {name: "Peter", age: 28};
```

```
var name = person.name;
var age = person.age;
alert(name); // Peter
alert(age); // 28
```

But in ES6, you can extract object's property values and assign them to the variables easily like this:

**Example-2:**

```
// ES6 syntax
let person = {name: "Peter", age: 28};
let {name, age} = person; // Object destructuring assignment
alert(name); // Peter
alert(age); // 28
```

## 5. Default Parameters

In ES6, we can declare a function with a default parameter with a default value.

**Example**: This example will show how to define default parameters for a function.

```
function fun(a, b=1){
   return a + b;
}
console.log(fun(5,2));
console.log(fun(3));
```

## 6. Rest parameter and spread operator

**Rest Parameter:** It is used to represent a number of parameter in an array to pass them together to a function.

**Spread Operator:** It simply changes an array of n elements into a list of n different elements.

Rest and spread syntax use … (three dots) notation.

Spread syntax expands an array into separate elements, while a rest syntax condenses array elements into a single element.

**REST Example:**

```
function fun(...input){
   let sum = 0;
   for(let i of input){
      sum += i;
   }
   return sum;
}
console.log(fun(1,2)); // 3
console.log(fun(1,2,3)); // 6
console.log(fun(1,2,3,4,5)); // 15
```

**SPREAD Example:**

```
// Example 1: Creating a shallow copy of an array
const originalArray = [1, 2, 3];
const copyArray = [...originalArray];

// Example 2: Concatenating arrays
const array1 = [1, 2, 3];
const array2 = [4, 5, 6];
const combinedArray = [...array1, ...array2];

// Example 3: Passing array elements as individual arguments to a function
function displayValues(a, b, c) {
  console.log(a, b, c);
}
const values = [1, 2, 3];
```

```
displayValues(...values); // Outputs: 1 2 3
```

## 7. Classes

ES6 introduced classes in JavaScript. Classes in javascript can be used to create new Objects with the help of a constructor, each class can only have one constructor inside it.

**Example-1:**

```
Class Person{
 constructor(name, age){
 this.name = name;
 this.age = age;
    }
}
let p = new Person('John',28);
console.log(p);
```

**Example-2:**

```
// classes in ES6
class Vehicle{
   constructor(name,engine){
      this.name = name;
      this.engine = engine;
   }
}

const bike1 = new Vehicle('Ninja ZX-10R','998cc');
const bike2 = new Vehicle('Duke','390cc');

console.log(bike1.name); // Ninja ZX-10R
console.log(bike2.name); // Duke
```

## 8. Modules

Modules are executed within their own scope, not in the global scope. This means that variables, functions, classes, etc. declared in a module are not visible outside the module unless they are explicitly exported using one of the export forms. Conversely, to consume a variable, function, class, interface, etc. exported from a different module, it has to be imported using one of the import forms.

ES Modules rely on the import and export statements.

**Export**

You can export a function or variable from any file.

Let us create a file named person.js, and fill it with the things we want to export.

There are two types of exports: Named and Default.

**Named Exports**

You can create named exports two ways. In-line individually, or all at once at the bottom.

**In-line individually:**

**person.js**

```
export const name = "Jesse"
export const age = 40
```

All at once at the bottom:

**person.js**

```
const name = "Jesse"
const age = 40
export { name, age }
```

**Default Exports**

Let us create another file, named message.js, and use it for demonstrating default export.

You can only have one default export in a file.

**Example: message.js**

```
const message = () => {
  const name = "Jesse";
  const age = 40;
  return name + ' is ' + age + 'years old.';
};
export default message;
```

## 9. Promises

Promises in JavaScript provide a way to work with asynchronous code in a more structured and manageable manner. They represent the eventual completion or failure of an asynchronous operation and allow you to handle the result when it becomes available.

**Creating a Promise:**

You create a promise using the Promise constructor. It takes a function as an argument, which has two parameters: resolve and reject. These are functions you call to indicate the successful completion or failure of the asynchronous operation.

```
const myPromise = new Promise((resolve, reject) => {
  // Asynchronous operation
  const isSuccess = true;

  if (isSuccess) {
    resolve("Operation succeeded!");
  } else {
    reject("Operation failed!");
  }
});
```

**Handling Promise Results:**

You can use the .then() method to handle the result when the promise is resolved, and the .catch() method to handle errors when the promise is rejected.

```
myPromise
  .then((result) => {
    console.log("Success:", result);
  })
  .catch((error) => {
    console.error("Error:", error);
  });
```