JavaScript ES6

JavaScript **ES6** (also known as **ECMAScript 2015** or **ECMAScript 6**) is the newer version of JavaScript that was introduced in 2015.
ECMAScript is the standard that JavaScript programming language uses. ECMAScript provides the specification on how JavaScript programming language should work.
This tutorial provides a brief summary of commonly used features of ES6 so that you can start quickly in ES6.

JavaScript let

JavaScript `let` is used to declare variables. Previously, variables were declared using the `var` keyword.
To learn more about the difference between `let` and `var`, visit JavaScript let vs var.
The variables declared using `let` are **block-scoped**. This means they are only accessible within a particular block. For example,

```
// variable declared using let
let name = 'Sara';
{
    // can be accessed only inside
    let name = 'Peter';

    console.log(name); // Peter
}
console.log(name); // Sara
```

JavaScript const

The `const` statement is used to declare constants in JavaScript. For example,

```
// name declared with const cannot be changed
const name = 'Sara';
```

Once declared, you cannot change the value of a `const` variable.

JavaScript Arrow Function

In the **ES6** version, you can use arrow functions to create function expressions. For example,

This function

```
// function expression
let x = function(x, y) {
   return x * y;
}
```

can be written as

```
// function expression using arrow function
let x = (x, y) => x * y;
```

To learn more about arrow functions, visit JavaScript Arrow Function.

JavaScript Classes

JavaScript class is used to create an object. Class is similar to a constructor function. For example,

```
class Person {
  constructor(name) {
    this.name = name;
  }
}
```

Keyword `class` is used to create a class. The properties are assigned in a constructor function.

Now you can create an object. For example,

```
class Person {
  constructor(name) {
    this.name = name;
  }
}

const person1 = new Person('John');

console.log(person1.name); // John
```

To learn more about classes, visit JavaScript Classes.

Default Parameter Values

In the ES6 version, you can pass default values in the function parameters. For example,

```
function sum(x, y = 5) {

    // take sum
    // the value of y is 5 if not passed
    console.log(x + y);
}
```

```
sum(5); // 10
sum(5, 15); // 20
```

In the above example, if you don't pass the parameter for `y`, it will take **5** by default.

To learn more about default parameters, visit JavaScript Function Default Parameters.

JavaScript Template Literals

The template literal has made it easier to include variables inside a string. For example, before you had to do:

```
const first_name = "Jack";
const last_name = "Sparrow";

console.log('Hello ' + first_name + ' ' + last_name);
```

This can be achieved using template literal by:

```
const first_name = "Jack";
const last_name = "Sparrow";

console.log(`Hello ${first_name} ${last_name}`);
```

To learn more about template literals, visit JavaScript Template Literal.

JavaScript Destructuring

The destructuring syntax makes it easier to assign values to a new variable. For example,

```
// before you would do something like this
const person = {
    name: 'Sara',
    age: 25,
    gender: 'female'
}

let name = person.name;
let age = person.age;
let gender = person.gender;

console.log(name); // Sara
console.log(age); // 25
console.log(gender); // female
```

Using **ES6** Destructuring syntax, the above code can be written as:

```
const person = {
    name: 'Sara',
    age: 25,
    gender: 'female'
}

let { name, age, gender } = person;

console.log(name); // Sara
console.log(age); // 25
console.log(gender); // female
```

To learn more about destructuring, visit JavaScript Destructuring.

JavaScript import and export

You could export a function or a program and use it in another program by importing it. This helps to make reusable components. For example, if you have two JavaScript files named contact.js and home.js.

In contact.js file, you can **export** the `contact()` function:

```
// export
export default function contact(name, age) {
    console.log(`The name is ${name}. And age is ${age}.`);
}
```

Then when you want to use the `contact()` function in another file, you can simply import the function. For example, in home.js file:

```
import contact from './contact.js';

contact('Sara', 25);
// The name is Sara. And age is 25
```

JavaScript Promises

Promises are used to handle asynchronous tasks. For example,

```
// returns a promise
let countValue = new Promise(function (resolve, reject) {
   reject('Promise rejected');
});

// executes when promise is resolved successfully
countValue.then(
    function successValue(result) {
        console.log(result); // Promise resolved
    },
 )
```
Run Code

To learn more about promises, visit JavaScript Promises.

JavaScript Rest Parameter and Spread Operator

You can use the **rest parameter** to represent an indefinite number of arguments as an array. For example,

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

show('one', 'two', 'three', 'four', 'five', 'six')
```

You pass the remaining arguments using `...` syntax. Hence, the name **rest parameter**.

You use the **spread syntax** `...` to copy the items into a single array. For example,

```
let arr1 = ['one', 'two'];
let arr2 = [...arr1, 'three', 'four', 'five'];
console.log(arr2); // ["one", "two", "three", "four", "five"]
```

Both the rest parameter and the spread operator use the same syntax. However, the spread operator is used with arrays (iterable values).

However, the `sayName()` function waits for the execution of the `greet()` function