Show some ❤️ and ⭐ the repo to support the project

Table Of Contents

1. Var, let and Const
2. Template literals
3. Default Arguments
4. Arrow Functions
5. Array and Object Destructuring
6. Map, Reduce and Filter
7. Iterables and Looping
8. Rest and Spread Operator
9. Object Literals
10. Classes in ES6
11. Promises
12. Async and Await
13. "this" and "new" keyword

About the Author

1. Var, let and const

1.1 Var

- Var keyword was previously used for declaring variable in javascript.
- Variables declared with var can be re-initialised and re-declared too.
- It is not **recommended** to use `var` after release of `let` and `const`.

```
var a = 10;
```

```
for(var i=0;i<5;i++){
            var a = 20;
            console.log(a); //Returns 20
}

console.log(a); // Returns 20
```

1.2 LET

- "let" is used when you have to change the value of the variable later in the code.
- It has block scope.
- It can be re-initialised but not re-declared.

```
let a = 10;

// re-initialization
a = 30; // Updating a value to 30.

//re-declartion
let a = 20; // Throws Error

// Block 1
{
     let c = 10;
     console.log(c); // c=10
}

console.log(c); // Throws Error, c not defined.
```

1.3 CONST

- Const is used to define a constant variable which can't be changed throught the code.
- It has block scope.
- You can neither be re-initiased nor re-declared.

```
const a = 10;

// re-initialization
a = 30; // Throws Error, CONST variable can't be changed

//re-declartion
const a = 20; // Throws Error

// Block 1
{
     const c = 10;
     console.log(c); // c=10
}
```

```
    console.log(c); // Throws Error, c not defined.
```

2. Template Literals

Template literals are string literals allowing embedded expressions. You can use multi-line strings and string interpolation features with them. They were called "template strings" in prior editions of the ES2015 specification.

Template literals are basically the formatting of string in javascript. In ES5, formatting string was a tedious task as it involved a very manual formatting syntax.

Let's see an example how we used to format string in ES5.

```
    # TEMPLATE STRING (WITHOUT ES6)

    function greet(name){
        const greeting = 'Hello,' + ' ' + name + ' ' + Welcome to JavaScript
Course;
        return greeting;
    }

    greet('Deepak');

    // Hello, Deepak Welcome to JavaScript Course.
```

```
    # TEMPLATE STRING (WITH ES6)

    function greet(name){
        const greeting = `Hello, ${name} Welcome to JavaScript Course`;
        return greeting;
    }

    greet('Deepak');

    // Hello, Deepak Welcome to JavaScript Course.
```

Now, you see the diffference how easy it is to use format string with ES6 new syntax.

RECAP

- Template String are enclosed by back tick(``) instead of single or double quote.
- Template literals can contain placeholders. These are indicated by the dollar sign and curly braces (${expression}). The expressions in the placeholders and the text between the back-ticks (``) get passed to a function.

3. Default Arguments

Default argument or default parameter is the new feature in ES6. It allows you to set a default value for your function parameter/argument if **no value** or **undefined** of is passed.

Handling Default Argument with ES5

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

add() // NaN

// Handling Default Argument without ES6.

function add(a, b){
    const a = (typeof(a) !== 'undefined') ? a : 5;
    const b = (typeof(b) !== 'undefined') ? b : 10;
  return a+b;
}

add() // Returns 15
```

When no parameter is passed you can see we have to explicitly handle the error by setting default values of a & b. This doesn't look like a favourable way of handling default arguments.

Handling Default Argument with ES6

```
function add(a=5, b=10){
    return a+b;
}

add(); // a=5, b=10, sum = 15;

add(2, 3); // a=2, b=3, sum = 5;

add(4); // a=4, b=10, sum=14 ;
```

Default value of A and B will be only used when no parameter is passed.

4. Arrow Functions

An arrow function is a syntactically compact alternative to a regular function expression without its own binding to `this`, `super`,

```
**Using Regular Function Express (ES5)**

    // Example 1
    function add(a, b){
        return a+b;
    }

    add(5, 10);

    // Example 2

    const x = [1, 2, 3, 4, 5];

    const square = x.map(function(x){
        return x*x;
    });

    console.log(sqaure);
```

Using Arrow Functions (ES6)

```
    // Example 1
    const add = (a, b) => {
                return a+b;
    }

    add(5, 10)

    //Example 2

    const x = [1, 2, 3, 4, 5];

    const square = x.map(num => num*num);
    console.log(sqaure);
```

5. Array and Object Destructuring

Destructuring is a new feature introduced in ES6 to unpack values from arrays or properties from object. It helps in improving the readability and performance of our code.

Destructuring in ES5

```
    // Example 1 - Object Destructuring

    var user = {
        name : 'Deepak',
      username : 'dipakkr',
```

```
  password : 12345
}

const name = user.name; // Deepak
const username = user.username; // dipakkr
const password = user.password // 12345

//Example 2 - Array Destructing

*c*onst fruits = ["apple", "mango", "banana", "grapes"];

const fruit1 = fruits[0];
const fruit2 = fruits[1];
const fruit3 = fruits[2];
```

Destructuring in ES6

```
// Example 1 - Object Destructuring

var user = {
    name : 'Deepak',
  username : 'dipakkr',
  password : 12345
}

const {name, username, password} = user;
console.log(name);
console.log(username);
console.log(password);

//Example 2 - Array Destructing

const fruits = ["apple", "mango", "banana", "grapes"];

const [fruit1, fruit2, fruit3] = fruits;

console.log(fruit1); // apple
console.log(fruit2); // mango
console.log(fruit3); // banana
```

6. Map, Reduce and Filter

Map, Reduce and Filter are the array methods which was introduced in ES6. The common things among these three methods are that when these methods applied on an array, it returns a new array based on the given parameter.

Map Method

Let's understand the Map method by taking a simple example. Let's say you have users array that contains multiple user object. But, you just need the username of each user.

How will you do that? Here is one way to do it.

```
const users = [
  { name: 'Deepak', username: 'dipakkr', password: '123456'},
  { name: 'Rohan', username: 'rohan12', password: '198243' },
  { name: 'Sam', username: 'sam124', password: '123876' },
];

var usernames = [];

users.forEach(function(user) {
  usernames.push(user.username);
});

console.log(usernames); // [ 'dipakkr', 'rohan12', 'sam124', 'ro123' ]
```

Now, let's solve this problem with `map()` method.

```
const users = [
  { name: 'Deepak', username: 'dipakkr', password: '123456'},
  { name: 'Rohan', username: 'rohan12', password: '198243' },
  { name: 'Sam', username: 'sam124', password: '123876' },
];

const usernames = users.map(user => user.username);

console.log(usernames); // [ 'dipakkr', 'rohan12', 'sam124', 'ro123' ]
```

Filter Method

Filter methods take a function parameter which applies on each array element, then whichever element satisfies the parameter condition returns in the new array.

```
const number = [5, 1, 4, 10, 15, 20, 12];

const result = number.filter(num => num>10);

console.log(result); // [15, 20, 12];
```

7. Iterables and Looping

Here is the list of interables in JavaScript.

Iterable	Description
Array	Access each element by iterating over an array.
Map	Iterates over the key-value pair
Strings	Access each character by interating over a string
Sets	Iterates over the set elements
Arguments	Access each argument by interating over arguments

`for...of` is a new feature got introduced in ES6 to access the interables element more easily. The ***for...of*** statement simply creates a loop iterating over iterable objects.

Looping Without `for...of`

```
const array = [5, 10, 15, 20, 25, 30, 35];

for(var value in array){
        console.log(array[value]);
}

// To access the element of the array, We are using array[postion] notation.
```

Looping with `for...of`

```
const array = [5, 10, 15, 20, 25, 30, 35];

for(var value of a){
        console.log(value);
}
```

So, we can see we are able to access interable elements directly with for...of method.

8. Rest and Spread Operator

Spread and Rest Operators are denoted by `...` three dots. These three dots can be used in 2 ways, one as `Spread Operator` and other as `Rest Parameter`

⇒ **Rest Parameter**

- It collects all the remaining elements into an array.
- Rest Parameter can collect any number of arguments into an array.
- Rest Parameter has to be the last arguments.

Without Using Rest Parameter

```
// Write a Function to print sum of arguments.

function add() {
  var sum = 0;
  for (var i = 0; i < arguments.length; i++) {
    sum = sum + arguments[i];
  }
  return sum;
}

console.log(add(1, 2, 3, 4, 5)); // 15

console.log(add(1, 3, 4)); // 8
```

Example Using Rest Operator

```
function add(...args) {
  let sum = 0;
  for (let i of args) {
    sum += i;
  }
  return sum;
}

console.log(add(3, 4, 5, 10, 20)); // 42

console.log(add(1, 3, 4)); // 8
```

Spread Operator

- It allows iterables like `arrays / objects /strings` to be expanded into single arguments/elements.
- Spread operator is opposite of Rest Parameter. In Rest Parameter We were collecting the list of arguments into an array, while with spread operator we can unpack the array elements.

Let's see an example to understand `spread`

```
## EXAMPLE - 1

const cars = ['BMW', 'Honda', 'Audi'];
const moreCars = ['Maruti', 'Swift', ...cars];

console.log(moreCars);  // ['Maruti', 'Swift', 'BMW', 'Honda', 'Audi'];


## EXAMPLE - 2 //Copying one array to other

const array1 = [1, 2, 3];
const copiedArray = ...array1;
```

```
console.log(copiedArray); // [1, 2, 3]
```

9. **Object Literals**

Object literals are used to create an object in javascript. Enhancement in Object literals in ES2015 (ES6) release has made it more powerful.

- Object can be initialised by directly using the variable name. See Example 1 below.
- Object's method in ES5 require `function` statement. This is no longer required in ES6, you can directly return statement. See Example 2 below.
- Object literals key in ES6 can be dynamic. Any Express can be used to create a key.

Let's take a look at this example to see the working of Object literals.

Object Literals Without ES6 (ES5 Supported)

```
# Example 1

var username = 'dipakkr'
var name = 'Deepak Kumar'
var country = 'India'
var password = '123456'

var user = {
    username : username,
  name : name,
    country : country,
    password : password
}

# Example 2

var calculate = {
  sqaure :  function(a) { return a*a; },
  sum : function(a, b) { return a + b; }
};

console.log(calculate.square(5));  // 25
console.log(calculate.sum(4,5));   //  9
```

Object Literals with ES6

```
# Example 1

const username = 'dipakkr'
const name = 'Deepak Kumar'
const country = 'India'
```

```
const password = '123456'

const user = {
    username,
    name,
    country,
    password,
};


# Example 2

const calculate = {
    square(a) return a*a,
  sum(a, b) return a+b
}

console.log(calculate.square(5));   // 25
console.log(calculate.sum(5,7));    // 12
```

10. **Classes in ES6**

JavaScript introduced in ECMAScript 2015. Classes support prototype-based inheritance, constructors, super calls, instance and static methods

There are two ways to define classes in JavaScript.

1. Class Declaration
2. Class Expression

Class Declaration

In order to define class using-declaration method you need to use `class` keyword followed by className. The class name must start with Capital letter.

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

Class Expression

A class expression is another way to define a class. Class expressions can be named or unnamed. The name given to a named class expression is local to the class's body.

```

```

```
let Rectangle = class {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};

console.log(Rectangle.name);
```

Mozilla Developer has great explanation for javascript classes. Read more **here**

11. Promises

For supporting asynchronous programming, JavaScript uses a callback. However, the callback implementation has a major problem which is called as `Callback hell.` Promises come to rescue to solve the problem of callback hell.

Promises are a pattern that greatly simplifies asynchronous programming by making the code look synchronous and avoid problems associated with callbacks.

A Promise has three states.

- **pending**: Initial state, neither fulfilled nor rejected.
- **fulfilled**: It means that the operation completed successfully.
- **rejected**: It means that the operation failed.

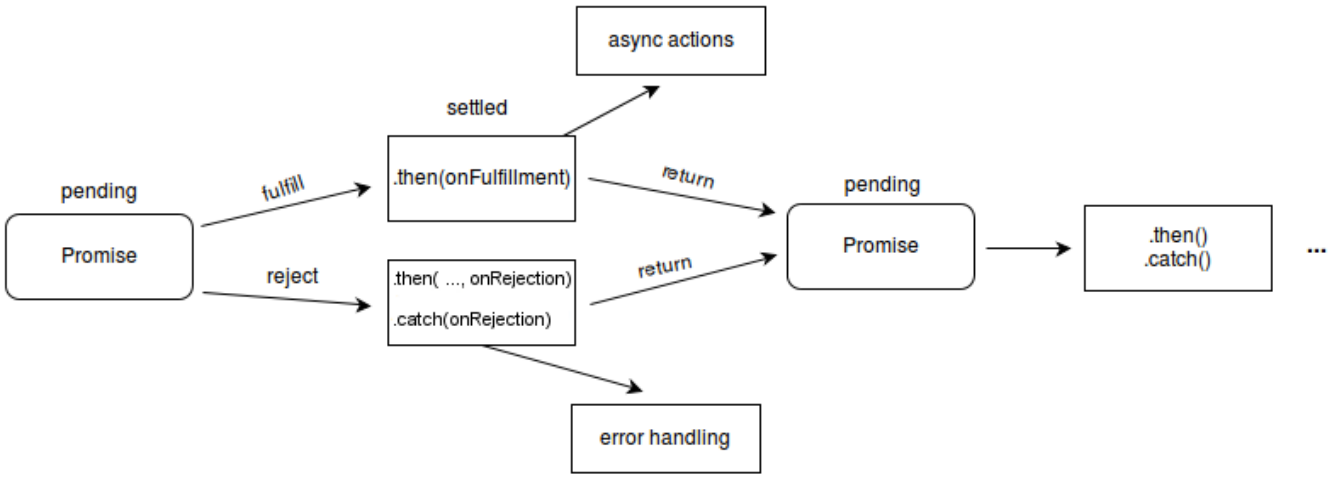

Image Credit : MDN

```
let promise = new Promise(function(resolve, reject) {
    setTimeout(() => resolve('Success ! '), 2000);
  });

  promise
    .then(function(result) {
      console.log(result);
    })
```

```
        .catch(function(error) {
          console.log(error);
        });
```

```
RESULT

Success !
```

To Read More about `Promises`, Checkout this link

About the Author

Hi, I am Deepak Kumar, a Full Stack JavaScript Developer, Freelancer and an aspiring Entrepreneur. I love building and scaling products that has real impact in community.

Join the Community and Stay updated on new release

Let's Connect ! - | LinkedIn | Instagram | Twitter

Licenses