

ES 6

ES6, also known as ECMAScript 2015, is a significant update to the JavaScript language, introducing new features and syntax that improve code readability, efficiency, and expressiveness

Essential JavaScript for React Part 1

1. JavaScript Variables
2. let ,var and const keywords
3. Functions and Arrow Functions
4. Objects
5. Arrays and array methods

JavaScript Variables

There are four ways to declare a JavaScript Variable:

- Using var
- Using let
- Using const
- Using nothing

In programming, a variable is a container (storage area) to hold data. For example,

```
• let num = 5;
```

Here, `num` is a variable. It's storing `5`.

In JavaScript, we use either `var` or `let` keyword to declare variables. For example,

```
var x;  
let y;
```

Here, `x` and `y` are variables.

Both `var` and `let` are used to declare variables. However, there are some differences between them.

Var

`var` is used in the older versions of JavaScript

`var` is function scoped

For example, `var x;`

Let

`let` is the new way of declaring variables starting in **ES6 (ES2015)**.

`let` is block

For example, `let y;`

Variable declared by `let` cannot be redeclared and must be declared before use whereas variables declared with `var` keyword are hoisted.

Rules for Naming JavaScript Variables

The rules for naming variables are:

Variable names must start with either a letter, an underscore `_`, or the dollar sign `$`. For example,

```
1. //valid  
2. let a = 'hello';  
3. let _a = 'hello';
```

```
4. let $a = 'hello';
```

Variable names cannot start with numbers. For example,

```
5. //invalid
6. Let 1a = 'hello'; // this gives an error
```

JavaScript is case-sensitive. So `y` and `Y` are different variables. For example,

```
7. let y = "hi";
8. let Y = 5;
9.
10. console.log(y); // hi
11. console.log(Y); // 5
```

Keywords cannot be used as variable names. For example,

```
12. //invalid
13. let new = 5; // Error! new is a keyword.
```

Note:

- Though you can name variables in any way you want, it's a good practice to give a descriptive variable name. If you are using a variable to store the number of apples, it's better to use `apples` or `numberOfApples` rather than `x` or `n`.
- In JavaScript, the variable names are generally written in camelCase if it has multiple words. For example, `firstName`, `annualSalary`, etc.

JavaScript Constants

The `const` keyword was also introduced in the **ES6(ES2015)** version to create constants. For example,

```
const x = 5;
```

Once a constant is initialized, we cannot change its value.

```
const x = 5;  
x = 10; // Error! constant cannot be changed.  
console.log(x)  
Run Code
```

Simply, a constant is a type of variable whose value cannot be changed.

Also, you cannot declare a constant without initializing it. For example,

```
const x; // Error! Missing initializer in const declaration.  
x = 5;  
console.log(x)  
Run Code
```

Note: If you are sure that the value of a variable won't change throughout the program, it's recommended to use `const`.

JavaScript Function

A function is a block of code that performs a specific task.

Suppose you need to create a program to create a circle and color it. You can create two functions to solve this problem:

- a function to draw the circle
- a function to color the circle

Dividing a complex problem into smaller chunks makes your program easy to understand and reusable.

JavaScript also has a huge number of inbuilt functions. For example, `Math.sqrt()` is a function to calculate the square root of a number. In this tutorial, you will learn about user-defined functions.

Declaring a Function

The syntax to declare a function is:

```
function nameOfFunction () {  
    // function body  
}
```

- A function is declared using the `function` keyword.
- The basic rules of naming a function are similar to naming a variable. It is better to write a descriptive name for your function. For example, if a function is used to add two numbers, you could name the function `add` or `addNumbers`.
- The body of function is written within `{}`.

For example,

```
// declaring a function named greet()  
function greet() {  
    console.log("Hello there");  
}
```

```
}
```

Calling a Function

In the above program, we have declared a function named `greet()`. To use that function, we need to call it.

Here's how you can call the above `greet()` function.

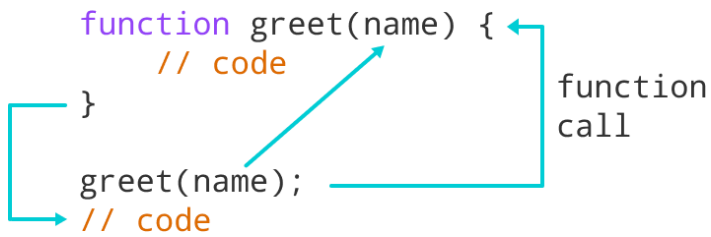
```
// function call  
greet();
```

```
function greet() {  
    // code  
}  
greet();  
// code
```

The diagram illustrates the relationship between a function declaration and its call. A teal arrow points from the closing curly brace of the `function greet() {` block to the text "function call". Another teal arrow points from the `greet();` line to the same "function call" text. A third teal arrow points from the `greet();` line to the `// code` comment below it.

Function Parameters

A function can also be declared with parameters. A parameter is a value that is passed when declaring a function.



Add Two Numbers

```
// program to add two numbers using a function  
// declaring a function  
function add(a, b) {  
    console.log(a + b);  
}  
  
// calling functions  
add(3,4);  
add(2,9);
```

In the above program, the `add` function is used to find the sum of two numbers.

- The function is declared with two parameters `a` and `b`.
- The function is called using its name and passing two arguments **3** and **4** in one and **2** and **9** in another.

Notice that you can call a function as many times as you want. You can write one function and then call it multiple times with different arguments.

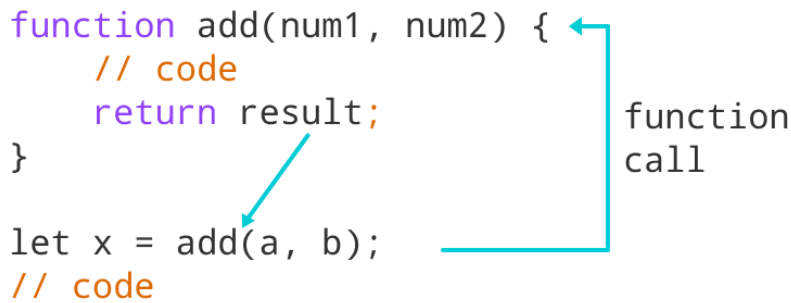
Function Return

The `return` statement can be used to return the value to a function call.

The `return` statement denotes that the function has ended. Any code after `return` is not executed.

If nothing is returned, the function returns an `undefined` value.

```
function add(num1, num2) {  
    // code  
    return result;  
}  
  
let x = add(a, b);  
// code
```



function call

Function Expressions

In Javascript, functions can also be defined as expressions. For example,

```
// program to find the square of a number  
// function is declared inside the variable  
let x = function (num) { return num * num };  
console.log(x(4));  
  
// can be used as variable value for other variables  
let y = x(3);  
console.log(y);
```

In the above program, variable `x` is used to store the function. Here the function is treated as an expression. And the function is called using the variable name.

The function above is called an anonymous function.

JavaScript Variable Scope

Scope refers to the availability of variables and functions in certain parts of the code.

In JavaScript, a variable has two types of scope:

1. Global Scope

2. Local Scope

3. A variable declared at the top of a program or outside of a function is considered a global scope variable.

4. Let's see an example of a global scope variable.

```
5. // program to print a text
6. let a = "hello";
7.
8. function greet () {
9.     console.log(a);
10.}
11.
12.greet(); // hello
```

In the above program, variable `a` is declared at the top of a program and is a global variable. It means the variable `a` can be used anywhere in the program. The value of a global variable can be changed inside a function. For example,

```
// program to show the change in global variable
let a = "hello";

function greet() {
    a = 3;
}

// before the function call
console.log(a);

//after the function call
greet();
console.log(a); // 3
```

the above program, variable `a` is a global variable. The value of `a` is `hello`.

Then the variable `a` is accessed inside a function and the value changes to **3**.

Hence, the value of `a` changes after changing it inside the function.

Note: It is a good practice to avoid using global variables because the value of a global variable can change in different areas in the program. It can introduce unknown results in the program.

In JavaScript, a variable can also be used without declaring it. If a variable is used without declaring it, that variable automatically becomes a global variable.

For example,

```
function greet() {  
    a = "hello"  
}  
  
greet();  
  
console.log(a); // hello
```

Local Scope

A variable can also have a local scope, i.e it can only be accessed within a function.

Local Scope Variable

```
// program showing local scope of a variable  
let a = "hello";  
  
function greet() {  
    let b = "World"  
    console.log(a + b);  
}
```

```
}  
  
greet();  
console.log(a + b); // error
```

In the above program, variable `a` is a global variable and variable `b` is a local variable. The variable `b` can be accessed only inside the function `greet`. Hence, when we try to access variable `b` outside of the function, an error occurs.

let is Block Scoped

The `let` keyword is block-scoped (variable can be accessed only in the immediate block).

Example: block-scoped Variable

```
// program showing block-scoped concept  
// global variable  
let a = 'Hello';  
  
function greet() {  
  
    // local variable  
    let b = 'World';  
  
    console.log(a + ' ' + b);  
  
    if (b == 'World') {  
  
        // block-scoped variable  
        let c = 'hello';  
  
        console.log(a + ' ' + b + ' ' + c);  
    }  
  
    // variable c cannot be accessed here  
    console.log(a + ' ' + b + ' ' + c);  
}
```

```
greet();
```

In the above program, variable

- `a` is a global variable. It can be accessed anywhere in the program.
- `b` is a local variable. It can be accessed only inside the function `greet`.
- `c` is a block-scoped variable. It can be accessed only inside the `if` statement block.

Hence, in the above program, the first two `console.log()` work without any issue.

However, we are trying to access the block-scoped variable `c` outside of the block in the third `console.log()`. This will throw an error.

JavaScript Arrow Function

Arrow function is one of the features introduced in the ES6 version of JavaScript. It allows you to create functions in a cleaner way compared to regular functions. For example,

This function

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

can be written as

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

using an arrow function.

Arrow Function Syntax

The syntax of the arrow function is:

```
let myFunction = (arg1, arg2, ...argN) => {  
  statement(s)  
}
```

Here,

- `myFunction` is the name of the function
- `arg1, arg2, ...argN` are the function arguments
- `statement(s)` is the function body

If the body has single statement or expression, you can write arrow function as:

```
let myFunction = (arg1, arg2, ...argN) => expression
```

Arrow Function with No Argument

If a function doesn't take any argument, then you should use empty parentheses. For example,

```
let greet = () => console.log('Hello');  
greet(); // Hello
```

Arrow Function with One Argument

If a function has only one argument, you can omit the parentheses. For example,

```
let greet = x => console.log(x);  
greet('Hello'); // Hello
```

Arrow Function as an Expression

You can also dynamically create a function and use it as an expression. For example,

```
let age = 5;  
  
let welcome = (age < 18) ?  
  () => console.log('Child') :  
  () => console.log('Ages');  
  
welcome(); // Child
```

Multiline Arrow Functions

If a function body has multiple statements, you need to put them inside curly brackets `{}`. For example,

```
let sum = (a, b) => {  
  let result = a + b;  
  return result;  
}  
  
let result1 = sum(5,7);  
console.log(result1); // 12
```

JavaScript Default Parameters

The concept of default parameters is a new feature introduced in the **ES6** version of JavaScript. This allows us to give default values to function parameters. Let's take an example,

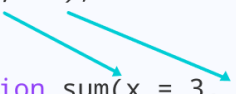
```
function sum(x = 3, y = 5) {  
  
    // return sum  
    return x + y;  
}  
  
console.log(sum(5, 15)); // 20  
console.log(sum(7));      // 12  
console.log(sum());       // 8  
R
```

In the above example, the default value of `x` is **3** and the default value of `y` is **5**.

- `sum(5, 15)` - When both arguments are passed, `x` takes **5** and `y` takes **15**.
- `sum(7)` - When **7** is passed to the `sum()` function, `x` takes **7** and `y` takes default value **5**.
- `sum()` - When no argument is passed to the **sum()** function, `x` takes default value **3** and `y` takes default value **5**.


Case 1: Both Argument are Passed

```
sum(5, 15);  
  
function sum(x = 3, y = 5) {  
    return x + y;  
}
```



Case 2: One Argument is Passed

```
sum(7);  
  
function sum(x = 3, y = 5) {  
    return x + y;  
}
```



Case 3: No Argument is Passed

```
sum();  
  
function sum(x = 3, y = 5) {  
    return x + y;  
}
```

Using Expressions as Default Values

It is also possible to provide expressions as default values.

Passing Parameter as Default Values

```
function sum(x = 1, y = x, z = x + y) {  
    console.log( x + y + z );  
}
```

```
sum(); // 4
```

In the above program,

- The default value of `x` is `1`

- The default value of `y` is set to `x` parameter
- The default value of `z` is the sum of `x` and `y`

If you reference the parameter that has not been initialized yet, you will get an error. For example,

```
function sum( x = y, y = 1 ) {  
    console.log( x + y );  
}  
  
sum();
```

Passing Function Value as Default Value

```
// using a function in default value expression  
  
const sum = () => 15;  
  
const calculate = function( x, y = x * sum() ) {  
    return x + y;  
}  
  
const result = calculate(10);  
console.log(result);           // 160
```

In the above program,

- **10** is passed to the `calculate()` function.
- `x` becomes `10`, and `y` becomes `150` (the sum function returns `15`).
- The result will be `160`.

JavaScript Objects

JavaScript object is a non-primitive data-type that allows you to store multiple collections of data.

Here is an example of a JavaScript object.

```
// object
const student = {
  firstName: 'khan',
  class: 12
};
```

Here, `student` is an object that stores values such as strings and numbers.

JavaScript Object Declaration

The syntax to declare an object is:

```
const object_name = {
  key1: value1,
  key2: value2
}
```

Here, an object `object_name` is defined. Each member of an object is a **key: value** pair separated by commas and enclosed in curly braces `{}`.

For example,

```
// object creation
const person = {
  name: 'Hassan',
  age: 20
};
console.log(typeof person); // object
```

-

Here, `name: 'Ali'` and `age: 20` are properties.

```
let person = {  
  name: 'Ali',  
  age: 20  
};
```

Accessing Object Properties

You can access the **value** of a property by using its **key**.

1. Using dot Notation

Here's the syntax of the dot notation.

```
objectName.key
```

For example,

```
const person = {  
  name: 'khan',  
  age: 20,  
};  
  
// accessing property  
console.log(person.name); // khan
```

2. Using bracket Notation

Here is the syntax of the bracket notation.

```
objectName["propertyName"]
```

For example,

```
const person = {  
  name: 'khan',  
  age: 20,  
};  
  
// accessing property  
console.log(person["name"]); // khan
```

JavaScript Nested Objects

An object can also contain another object. For example,

```
// nested object  
const student = {  
  name: 'ali',  
  age: 20,  
  marks: {  
    science: 70,  
    math: 75  
  }  
}  
  
// accessing property of student object  
console.log(student.marks); // {science: 70, math: 75}  
  
// accessing property of marks object  
console.log(student.marks.science); // 70
```

JavaScript Object Methods

In JavaScript, an object can also contain a function. For example,

```
const person = {  
  name: 'khan',  
  age: 30,  
  // using function as a value  
  greet: function() { console.log('hello') }  
}  
  
person.greet(); // hello
```

JavaScript Arrays

An array is an object that can store multiple values at once. For example,

```
const words = ['hello', 'world', 'welcome'];
```

Here, `words` is an array. The array is storing 3 values.

Create an Array

You can create an array using two ways:

1. Using an array literal

The easiest way to create an array is by using an array literal `[]`. For example,

```
const array1 = ["eat", "sleep"];
```

2. Using the new keyword

You can also create an array using JavaScript's `new` keyword.

```
const array2 = new Array("eat", "sleep");
```

Here are more examples of arrays:

```
// empty array
const myList = [ ];

// array of numbers
const numberArray = [ 2, 4, 6, 8];

// array of strings
const stringArray = [ 'eat', 'work', 'sleep'];

// array with mixed data types
const newData = ['work', 'exercise', 1, true];
```

You can also store arrays, functions and other objects inside an array. For example,

```
const newData = [
  {'task1': 'exercise'},
  [1, 2, 3],
  function hello() { console.log('hello')}
];
```

Access Elements of an Array

You can access elements of an array using indices (**0, 1, 2 ...**). For example,

```
const myArray = ['h', 'e', 'l', 'l', 'o'];

// first element
console.log(myArray[0]); // "h"
```

```
// second element
console.log(myArray[1]); // "e"
```

Add an Element to an Array

You can use the built-in method `push()` and `unshift()` to add elements to an array.

The `push()` method adds an element at the end of the array. For example,

```
let dailyActivities = ['eat', 'sleep'];

// add an element at the end
dailyActivities.push('exercise');

console.log(dailyActivities); // ['eat', 'sleep', 'exercise']
```

The `unshift()` method adds an element at the beginning of the array. For example,

```
let dailyActivities = ['eat', 'sleep'];

//add an element at the start
dailyActivities.unshift('work');

console.log(dailyActivities); // ['work', 'eat', 'sleep']
```

Change the Elements of an Array

You can also add elements or change the elements by accessing the index value.

```
let dailyActivities = [ 'eat', 'sleep'];

// this will add the new element 'exercise' at the 2 index
dailyActivities[2] = 'exercise';

console.log(dailyActivities); // ['eat', 'sleep', 'exercise']
```

Remove an Element from an Array

You can use the `pop()` method to remove the last element from an array.

The `pop()` method also returns the removed value. For example,

```
let dailyActivities = ['work', 'eat', 'sleep', 'exercise'];

// remove the last element
dailyActivities.pop();
console.log(dailyActivities); // ['work', 'eat', 'sleep']

// remove the last element from ['work', 'eat', 'sleep']
const removedElement = dailyActivities.pop();

//get removed element
console.log(removedElement); // 'sleep'
console.log(dailyActivities); // ['work', 'eat']
```

If you need to remove the first element, you can use the `shift()` method.

The `shift()` method removes the first element and also returns the removed element. For example,

```
let dailyActivities = ['work', 'eat', 'sleep'];

// remove the first element
dailyActivities.shift();

console.log(dailyActivities); // ['eat', 'sleep']
```

Array length

You can find the length of an array (the number of elements in an array) using the `length` property. For example,

```
const dailyActivities = [ 'eat', 'sleep'];

// this gives the total number of elements in an array
console.log(dailyActivities.length); // 2
```


Array Methods

In JavaScript, there are various array methods available that makes it easier to perform useful calculations.

Some of the commonly used JavaScript array methods are:

Method	Description
concat()	joins two or more arrays and returns a result
indexOf()	searches an element of an array and returns its position
find()	returns the first value of an array element that passes a test
findIndex()	returns the first index of an array element that passes a test
forEach()	calls a function for each element
includes()	checks if an array contains a specified element
push()	adds a new element to the end of an array and returns the new length of a array
unshift()	adds a new element to the beginning of an array and returns the new leng of an array
pop()	removes the last element of an array and returns the removed element

shift()	removes the first element of an array and returns the removed element
sort()	sorts the elements alphabetically in strings and in ascending order
slice()	selects the part of an array and returns the new array
splice()	removes or replaces existing elements and/or adds new elements

JavaScript Array Methods

```
let dailyActivities = ['sleep', 'work', 'exercise']
let newRoutine = ['eat'];

// sorting elements in the alphabetical order
dailyActivities.sort();
console.log(dailyActivities); // ['exercise', 'sleep', 'work']

//finding the index position of string
const position = dailyActivities.indexOf('work');
console.log(position); // 2

// slicing the array elements
const newDailyActivities = dailyActivities.slice(1);
console.log(newDailyActivities); // [ 'sleep', 'work' ]

// concatenating two arrays
const routine = dailyActivities.concat(newRoutine);
console.log(routine); // ["exercise", "sleep", "work", "eat"]
```

JavaScript forEach()

The `forEach()` method calls a function and iterates over the elements of an array. The `forEach()` method can also be used on Maps and Sets.

JavaScript forEach

The syntax of the `forEach()` method is:

```
array.forEach(function(currentValue, index, arr))
```

Here,

- `function(currentValue, index, arr)` - a function to be run for each element of an array
- `currentValue` - the value of an array
- `index (optional)` - the index of the current element

`arr (optional)` - the array of the current elements

forEach with Arrays

The `forEach()` method is used to iterate over an array. For example,

```
let students = ['ali', 'hassan', 'saeed'];

// using forEach
students.forEach(myFunction);

function myFunction(item) {

    console.log(item);
}
```

Output

```
ali
hassan
saeed
```

Updating the Array Elements

As we have seen in the above example, the `forEach()` method is used to iterate over an array, it is quite simple to update the array elements. For example,

```
let students = ['ali', 'hassan', 'saeed'];

// using forEach
students.forEach(myFunction);

function myFunction(item, index, arr) {

    // adding strings to the array elements
    arr[index] = 'Hello ' + item;
}

console.log(students);
```

Output

```
["Hello ali", "Hello hassan", "Hello saeed"]
```

forEach with Arrow Function

You can use the arrow function with the `forEach()` method to write a program. For example,

```
// with arrow function and callback

const students = ['ali', 'hassan', 'saeed'];

students.forEach(element => {
    console.log(element);
});

students.forEach((item, index, arr)=> {
    console.log("hello"+item);
});
```

JavaScript for...in loop

The syntax of the `for...in` loop is:

```
for (key in object) {  
    // body of for...in  
}
```

In each iteration of the loop, a key is assigned to the `key` variable. The loop continues for all object properties.

Iterate Through an Object

```
const student = {  
    name: 'khadija',  
    class: 7,  
    age: 12  
}  
  
// using for...in  
for ( let key in student ) {  
  
    // display the properties  
    console.log(`${key} => ${student[key]}`);  
}
```

WEEK #2

Essential JavaScript for React Part 2

1. Template literals
2. Spread and Rest Operators
3. Destructuring
4. JavaScript Map, Reduce, and Filter
5. Ternary Operators
6. ES Modules and Import / Export Syntax

JavaScript Template Literals

Template literals (template strings) allow you to use strings or embedded expressions in the form of a string. They are enclosed in backticks ```. For example,

```
const name = 'Ali';  
console.log(`Hello ${name}!`); // Hello Ali!
```

String interpolation

Without template literals, when you want to combine output from expressions with strings, you'd concatenate them using the addition operator `+`:

```
const a = 5;  
const b = 10;  
console.log("Fifteen is " + (a + b) + " and\nnot " + (2 * a + b) + ".");  
// "Fifteen is 15 and  
// not 20."
```

That can be hard to read – especially when you have multiple expressions.

With template literals, you can avoid the concatenation operator — and improve the readability of your code — by using placeholders of the form `${expression}` to perform substitutions for embedded expressions:

```
const a = 5;
const b = 10;
console.log(`Fifteen is ${a + b} and
not ${2 * a + b}.`);
// "Fifteen is 15 and
// not 20."
```

Spread Operator

The spread operator `...` is used to expand or spread an iterable or an array. For example,

```
const arrValue = ['My', 'name', 'is', 'Ali'];

console.log(arrValue); // ["My", "name", "is", "Ali"]
console.log(...arrValue); // My name is Ali
```

Copy Array Using Spread Operator

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

```
const arr1 = ['one', 'two'];
const arr2 = [...arr1, 'three', 'four', 'five'];

console.log(arr2);
// Output:
// ["one", "two", "three", "four", "five"]
```

Clone Array Using Spread Operator

In JavaScript, objects are assigned by reference and not by values. For example,

```
let arr1 = [ 1, 2, 3];
let arr2 = arr1;

console.log(arr1); // [1, 2, 3]
console.log(arr2); // [1, 2, 3]

// append an item to the array
```



```
arr1.push(4);

console.log(arr1); // [1, 2, 3, 4]
console.log(arr2); // [1, 2, 3, 4]
```

Here, both variables `arr1` and `arr2` are referring to the same array. Hence the change in one variable results in the change in both variables.

However, if you want to copy arrays so that they do not refer to the same array, you can use the spread operator. This way, the change in one array is not reflected in the other. For example,

```
let arr1 = [ 1, 2, 3];

// copy using spread syntax
let arr2 = [...arr1];

console.log(arr1); // [1, 2, 3]
console.log(arr2); // [1, 2, 3]

// append an item to the array
arr1.push(4);

console.log(arr1); // [1, 2, 3, 4]
console.log(arr2); // [1, 2, 3]
```

Spread Operator with Object

You can also use the spread operator with object literals. For example,

```
const obj1 = { x : 1, y : 2 };
const obj2 = { z : 3 };

// add members obj1 and obj2 to obj3
const obj3 = {...obj1, ...obj2};

console.log(obj3); // {x: 1, y: 2, z: 3}
```

Rest Parameter

When the spread operator is used as a parameter, it is known as the rest parameter.

You can also accept multiple arguments in a function call using the rest parameter. For example,

```
let func = function(...args) {  
  console.log(args);  
}  
  
func(3); // [3]  
func(4, 5, 6); // [4, 5, 6]
```

Here,

- When a single argument is passed to the `func()` function, the rest parameter takes only one parameter.
- When three arguments are passed, the rest parameter takes all three parameters.

You can also pass multiple arguments to a function using the spread operator.

For example,

```
function sum(x, y ,z) {  
  console.log(x + y + z);  
}  
  
const num1 = [1, 3, 4, 5];  
  
sum(...num1); // 8
```

If you pass multiple arguments using the spread operator, the function takes the required arguments and ignores the rest.

JavaScript Destructuring

The destructuring assignment introduced in **ES6** makes it easy to assign array values and object properties to distinct variables. For example,

Without Destructuring:

```
// assigning object attributes to variables
const person = {
  name: 'Khadija',
  age: 25,
  gender: 'female'
}

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

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

Using Destructuring :

```
// assigning object attributes to variables
const person = {
  name: 'Khadija',
  age: 25,
  gender: 'female'
}

// destructuring assignment
let { name, age, gender } = person;

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

Note: The order of the name does not matter in object destructuring.

For example, you could write the above program as:

```
let { age, gender, name } = person;
console.log(name); // khadija
```

For example,

```
let {name1, age, gender} = person;
console.log(name1); // undefined
```

If you want to assign different variable names for the object key, you can use:

Note: When destructuring objects, you should use the same name for the variable as the corresponding object key.

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

// destructuring assignment
// using different variable names
let { name: name1, age: age1, gender: gender1 } = person;

console.log(name1); // Khadija
console.log(age1); // 25
console.log(gender1); // female
```

Access Object Keys, Values & Entries

You often need to look through the properties and values of plain JavaScript objects.

Here are the common lists to extract from an object:

- The keys of an object is the list of property names.
- The values of an object is the list of property values.
- The entries of an object is the list of pairs of property names and corresponding values.

Let's consider the following JavaScript object:

```
const person = {  
  name: 'Hassan',  
  city: 'Rawalpindi'  
};
```

The keys of hero are ['name', 'city']. The values are [Hassan', 'Rawalpindi']. And the entries are [['name', 'Hassan'], ['city', 'Rawalpindi']].

Let's see what utility functions provide JavaScript to extract the keys, values, and entries from an object.

Object.keys()

Object.keys(object) is a utility function that returns the list of keys of object.

Let's use Object.keys() to get the keys of person object:

```
const person = {  
  name: 'Hassan',  
  city: 'Rawalpindi'  
};  
Object.keys(person); // => ['name', 'city']
```

Object.keys(person) returns the list ['name', 'city'], which, as expected, are the keys of person object.

Object.values()

Object.values(object) is the JavaScript utility function that returns the list of values of object.

Let's use this function to get the values of hero object:

```
const person = {
  name: 'Hassan',
  city: 'Rawalpindi'
};
Object.values(person); // => ['Hassan', 'Rawalpindi']
```

Object.values(person) returns the values of hero: ['Hassan', Rawalpindi].

Object.entries()

Object.entries(object) is an useful function to access the entries of object.

Let's extract the entries of person object:

```
const person = {
  name: 'Hassan',
  city: 'Rawalpindi'
};
Object.entries(person); // => [['name', 'Hassan'], ['city', 'Rawalpindi']]
```

Object.entries(person) returns the entries of person: [['name', 'Hassan'], ['city', 'Rawalpindi']].

Array Destructuring

You can also perform array destructuring in a similar way. For example,

```
const arrValue = ['one', 'two', 'three'];

// destructuring assignment in arrays
const [x, y, z] = arrValue;

console.log(x); // one
console.log(y); // two
console.log(z); // three
```

Assign Default Values

You can assign the default values for variables while using destructuring. For example,

```
let arrValue = [10];

// assigning default value 5 and 7
let [x = 5, y = 7] = arrValue;

console.log(x); // 10
console.log(y); // 7
```

In the above program, `arrValue` has only one element. Hence,

- the `x` variable will be **10**
- the `y` variable takes the default value **7**

In object destructuring, you can pass default values in a similar way. For example,

```
const person = {
  name: 'Ali',
}

// assign default value 26 to age if undefined
const { name, age = 26 } = person;

console.log(name); // Ali
console.log(age); // 26
```

Swapping Variables

In this example, two variables are swapped using the destructuring assignment syntax.

```
// program to swap variables
```

```
let x = 4;
```

```
let y = 7;
```

```
// swapping variables
```

```
[x, y] = [y, x];
```

```
console.log(x); // 7
```

```
console.log(y); // 4
```

```
Run Code
```

Skip Items

You can skip unwanted items in an array without assigning them to local variables. For example,

```
const arrValue = ['one', 'two', 'three'];
```

```
// destructuring assignment in arrays
```

```
const [x, , z] = arrValue;
```

```
console.log(x); // one
```

```
console.log(z); // three
```

```
Run Co
```

In the above program, the second element is omitted by using the comma separator `,`.

Assign Remaining Elements to a Single Variable

You can assign the remaining elements of an array to a variable using the spread syntax `.....`. For example,

```
const arrValue = ['one', 'two', 'three', 'four'];
```



```
// destructuring assignment in arrays
// assigning remaining elements to y
const [x, ...y] = arrValue;

console.log(x); // one
console.log(y); // ["two", "three", "four"]
Run Co
```

Here, `one` is assigned to the `x` variable. And the rest of the array elements are assigned to `y` variable.

You can also assign the rest of the object properties to a single variable. For example,

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

// destructuring assignment
// assigning remaining properties to rest
let { name, ...rest } = person;

console.log(name); // Khadija
console.log(rest); // {age: 25, gender: "female"}
R
```

Note: The variable with the spread syntax cannot have a trailing comma `,`. You should use this rest element (variable with spread syntax) as the last variable.

For example,

```
const arrValue = ['one', 'two', 'three', 'four'];

// throws an error
```

```
const [ ...x, y] = arrValue;

console.log(x); // error
```

JavaScript Map, Reduce, and Filter - JS Array Functions

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: 'ali', grade: 96 },  
  { name: 'hamid', grade: 84 },  
  { name: 'nasir', grade: 100 },  
  { name: 'noman', grade: 65 },  
  { name: 'anas', grade: 90 }  
];
```

```
const studentGrades = students.filter(student => student.grade >= 90);  
return studentGrades; // [ { name: 'ali', grade: 96 }, { name: 'nasir', grade: 100 },  
  { name: 'anas', 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
}
*/
```

Ternary Operator

A ternary operator evaluates a condition and executes a block of code based on the condition.

Its syntax is:

```
condition ? expression1 : expression2
```

The ternary operator evaluates the test condition.

- If the condition is `true`, **expression1** is executed.
- If the condition is `false`, **expression2** is executed.

The ternary operator takes **three** operands, hence, the name ternary operator.

It is also known as a conditional operator.

Let's write a program to determine if a student passed or failed in the exam based on marks obtained.

Example: JavaScript Ternary Operator

```
// program to check pass or fail

let marks = 78;

// check the condition
let result = (marks >= 40) ? 'pass' : 'fail';

console.log(`You ${result} the exam.`);
```

JavaScript Modules

As our program grows bigger, it may contain many lines of code. Instead of putting everything in a single file, you can use modules to separate codes in separate files as per their functionality. This makes our code organized and easier to maintain.

Module is a file that contains code to perform a specific task. A module may contain variables, functions, classes etc. Let's see an example,

Suppose, a file named **Hello.js** contains the following code:

```
// exporting a function
export function sayHello(name) {
    return `Hello ${name}`;
}
```

Now, to use the code of **Hello.js** in another file, you can use the following code:

```
// importing greetPerson from greet.js file
import { SayHello } from './Hello.js';

// using greetPerson() defined in greet.js
let displayName = sayHello('Yasir');

console.log(displayName); // Hello Yasir
```

Here,

- The `sayHi()` function in the **Hello.js** is exported using the `export` keyword

```
export function sayHello(name) {
  ...
}
```

- Then, we imported `sayHi()` in another file using the `import` keyword. To import functions, objects, etc., you need to wrap them around `{ }`.

```
import { sayHi } from './Hello.js';
```

You can only access exported functions, objects, etc. from the module. You need to use the `export` keyword for the particular function, objects, etc. to import them and use them in other files.

You can export members one by one. What's not exported won't be available directly outside the module:

```
export const myNumbers = [1, 2, 3, 4];

const animals = ['Panda', 'Bear', 'Eagle']; // Not available directly
outside the module

export function myLogger() {
  console.log(myNumbers, animals);
}
```

Or you can export desired members in a single statement at the end of the module:

```
export { myNumbers, myLogger};
```

Exporting with alias

You can also give an aliases to exported members with the as keyword:

```
export { myNumbers, myLogger as Logger}
```

Default export

You can define a default export with the default keyword:

```
export const myNumbers = [1, 2, 3, 4];  
const animals = ['Panda', 'Bear', 'Eagle'];  
  
export default function myLogger() {  
  console.log(myNumbers, pets);  
}
```

Importing with alias

You can also alias members at import time:

```
import myLogger as Logger from 'app.js';
```

Importing all exported members

You can import everything that's imported by a module like this:

```
import * as Utils from 'app.js';
```


This allows you access to members with the dot notation:

```
Utils.myLogger();
```