ES6 Functions

A function is the set of input statements, which performs specific computations and produces output. It is a block of code that is designed for performing a particular task. It is executed when it gets invoked (or called). Functions allow us to reuse and write the code in an organized way. [Functions in JavaScript](https://www.javatpoint.com/javascript-function) are used to perform operations.

In [JavaScript](https://www.javatpoint.com/javascript-tutorial), functions are defined by using function keyword followed by a **name** and **parentheses ()**. The function name may include digits, letters, dollar sign, and underscore. The brackets in the function name may consist of the name of parameters separated by commas. The body of the function should be placed within **curly braces {}.**

The syntax for defining a standard function is as follows:

1. function function\_name() {
2. //body of the function
3. }

To force the execution of the function, we must have to invoke (or call) the function. It is known as function invocation. The syntax for invoking a function is as follows:

1. function\_name()

Example

1. function hello() //defining a function
2. {
3. console.log ("hello function called");
4. }
5. hello(); //calling of function

In the above code, we have defined a function **hello().** The pair of parentheses **{}** define the body of the function, which is called a scope of function.

**Output**

hello function called

Let us try to understand different functions.

Parameterized functions

Parameters are the names that are listed in the definition of the function. They are the mechanism of passing the values to functions.

The values of the parameters are passed to the function during invocation. Unless it is specified explicitly, the number of values passed to a function must match with the defined number of parameters.

**Syntax**

The syntax for defining a parameterized function is:

1. function function\_name( parameter1,parameter2 ,…..parameterN) {
2. //body of the function
3. }

**Example**

In this example of parameterized function, we are defining a function **mul(),** which accepts two parameters **x** and **y,** and it returns their multiplication in the result. The parameter values are passed to the function during invocation.

1. function mul( x , y) {
2. var c = x \* y;
3. console.log("x = "+x);
4. console.log("y = "+y);
5. console.log("x \* y = "+c);
6. }
7. mul(20,30);

**Output**

x = 20

y = 30

x \* y = 600

Default Function Parameters

In ES6, the function allows the initialization of parameters with default values if the parameter is undefined or no value is passed to it.

You can see the illustration for the same in the following code:

**For example**

1. function show (num1, num2=200)
2. {
3. console.log("num1 = " +num1);
4. console.log("num2 = " +num2);
6. }
7. show(100);

In the above function, the value of **num2** is set to **200** by default. The function will always consider **200** as the value of **num2** if no value of **num2** is explicitly passed.

**Output**

100 200

The default value of the parameter **'num2'** will get overwritten if the function passes its value explicitly. You can understand it by using the following example:

**For example**

1. function show(num1, num2=200)
2. {
3. console.log("num1 = " +num1);
4. console.log("num2 = " +num2);
5. }
6. show(100,500);

**Output**

100 500

Rest Parameters

Rest parameters do not restrict you to pass the number of values in a function, but all the passed values must be of the same type. We can also say that rest parameters act as the placeholders for multiple arguments of the same type.

For declaring the rest parameter, the name of the parameter should be prefixed with the **spread operator** that has three periods (not more than three or not less than three).

You can see the illustration for the same in the following example:

1. function show(a, ...args)
2. {
3. console.log(a + " " + args);
4. }
5. show(50,60,70,80,90,100);

**Output**

50,60,70,80,90,100

Note: The rest parameters should be at last in the list of function parameters.

Returning functions

The function also returns the value to the caller by using the **return** statement. These functions are known as **Returning** functions. A returning function should always end with a **return** statement. There can be only one **return** statement in a function, and the **return** statement should be the last statement in the function.

When JavaScript reaches the **return** statement, the function stops the execution and exits immediately. That's why you can use the **return** statement to stop the execution of the function immediately.

**Syntax**

A function can return the value by using the **return** statement, followed by a value or an expression. The syntax for the returning function is as follows:

1. function function\_name() {
2. //code to be executed
3. **return** value;
4. }

**Example**

1. function add( a, b )
2. {
3. **return** a+b;
4. }
5. var sum = add(10,20);
6. console.log(sum);

In the above example, we are defining a function **add()** that has two parameters **a** and **b**. This function returns the addition of the arguments to the caller.

You will get the following output after the execution of the above code.

**Output**

30

Generator functions

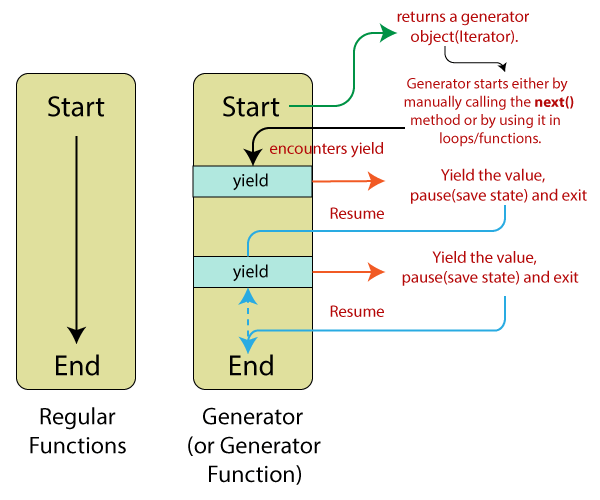
Generator (or Generator function) is the new concept introduced in ES6. It provides you a new way of working with iterators and functions.

ES6 generator is a different kind of function that may be paused in the middle either one or many times and can be resumed later.

ES6 Generators

Generator (or Generator function) is the new concept introduced in ES6. It provides you a new way of working with iterators and functions.

ES6 generator is a different kind of function that may be paused in the middle either one or many times and can be resumed later. When the standard function is called, the control rests with the called function until it returns, but the generator in [ES6](https://www.javatpoint.com/es6) allows the caller function to control the execution of a called function.



A generator is somewhere identical to the regular function except that:

* When the generator gets called, it doesn't run its code. Instead, it returns a special object which is called as '**Generator Object**' for managing the execution.
* The generator function can return (**or yield**) the control back to the caller at any point.
* Unlike the regular function, the generator can return (**or yield**) the multiple values, one after another, on the requirement.

Syntax

The syntax of the generator function is almost identical to regular function. The only real difference is that the generator function is denoted by **suffixing** the **function keyword** with an **asterisk (\*)**.

In the following syntax, we are showing you some of the valid ways of defining the generator function:

1. function\* mygenfun()    // Valid
2. {
3. yield 1;
4. yield 2;
5. ...
6. ...
7. }
9. function \*mygenfun()    // Valid
10. {
11. yield 1;
12. yield 2;
13. ...
14. ...
15. }
17. function\*mygenfun() // Valid
18. {
19. yield 1;
20. yield 2;
21. ...
22. ...
23. }

Example

1. function\* gen()
2. {
3. yield 100;
4. yield;
5. yield 200;
6. }
7. // Calling the Generator Function
8. var mygen = gen();
9. console.log(mygen.next().value);
10. console.log(mygen.next().value);
11. console.log(mygen.next().value);

**Output**

100

undefined

200

The yield statement

The yield statement suspends the function execution and sends a value back to the caller. It retains enough state to enable function to resume where it is left off. When it gets resumed, the function continues the execution immediately after the last yield run. It can produce a sequence of values.

next() method

In the above example, we have used the **next()**method, which is the main method of the generator. When you call the **next()**method along with an argument, it will resume the execution of the generator function, replacing the **yielded**expression where the execution was paused with the argument from the **next() method.**

The result of the **next()** method is always an object having two properties:

* **value:** It is the yielded value.
* **done:** It is a Boolean value which gives true if the function code has finished. Otherwise, it gives false.

For instance, here, we are creating a generator function and gets its yielded value.

Example

1. function\* show() {
2. yield 100;
3. }
5. var gen = show();   //here 'gen' is a generator object
6. console.log(gen.next()); // { value: 100, done: false }

**Output**

{ value: 100, done: false }

Generator object

The generator functions return generator objects. A generator object is an instance of the generator function, and it conforms to both the iterable and iterator interfaces.

The generator objects can be used either by calling the **next() method**or by using the generator object within a loop. The generator object is an iterator; that's why you can use it in **for…of**loops or in other functions accepting an iterable.

In the above **next() method**example, the variable **gen**is the generator object.

Return Statement in a Generator

Return is used to send a specified value back to its caller. It is used to end the function call execution and returns the result to the caller. Within a function, the statements defined after the return statements are not executed. That's why the return statement should be the last statement of the function.

Let us try to understand the **return** statement in a generator by using an example:

Example

1. function\* myGen()  {
2. yield 'First yield statement';
3. yield 'Second yield statement';
4. **return** 'Return statement';
5. yield 'Second yield statement';
6. }
7. let genobj = myGen();
9. console.log(genobj.next());    //returns {value: 'First yield statement', done: false}
11. console.log(genobj.next());   //returns {value: 'Second yield statement', done: false}
13. console.log(genobj.next());  //returns {value: 'Return statement', done: true}
15. console.log(genobj.next()); //returns {value: undefined, done: true}

**Output**

{ value: 'First yield statement', done: false }

{ value: 'Second yield statement', done: false }

{ value: 'Return statement', done: true }

{ value: undefined, done: true }

In the above example, we have defined a generator function **myGen()**in which we have defined four statements, including three **yield statements** and a **return statement.**Whenever we call the **next() method,**the function resumes until it hits the next **yield statement.**

You can notice how the first **next() method**returns **'First yield statement .'**When we call the **next() method a**second time, it resumes the execution and returns **'Second yield statement'**. After calling **the next() method**again, the function finds no more **yield**statement and returns the **'Return statement.'**But when we call the **next() method**fourth time, it will not consider the **yield**statement and returns **undefined** because it is written after the **return**statement.

You can see in the output of the above example that the **next() method**is not considering any statement after the **return statement.**

Generator function with for…of loop

Using for…of loop with generator function reduces the line of code. You can see the illustration for the same in the following example.

Example

1. "use strict"
2. function\* vowels() {
3. // here the asterisk marks this as a generator
4. yield 'A';
5. yield 'E';
6. yield 'I';
7. yield 'O';
8. yield 'U';
9. }
10. **for**(let alpha of vowels()) {
11. console.log(alpha);
12. }

**Output**

A

E

I

O

U

Anonymous function

An anonymous function can be defined as a function without a name. The anonymous function does not bind with an identifier. It can be declared dynamically at runtime. The anonymous function is not accessible after its initial creation.

An anonymous function can be assigned within a variable. Such expression is referred to as **function expression.** The syntax for the anonymous function is as follows.

**Syntax**

1. var y = function( [arguments] )
2. {
3. //code to be executed
4. }

**Example**

1. var hello = function() {
2. console.log('Hello World');
3. console.log('I am an anonymous function');
4. }
5. hello();

**Output**

Hello World

I am an anonymous function

Anonymous function as an argument

One common use of the anonymous function is that it can be used as an argument to other functions.

**Example**

Use as an argument to other function.

1. setTimeout(function()
2. {
3. console.log('Hello World');
4. }, 2000);

When you execute the above code, it will show you the output after two seconds.

**Output**

Hello World

Parameterized Anonymous function

**Example**

1. var anon = function(a,b)
2. {
3. **return** a+b
4. }
5. function sum() {
6. var res;
7. res = anon(100,200);
8. console.log("Sum: "+res)
9. }
10. sum()

**Output**

Sum: 300

Arrow functions

Arrow functions are introduced in ES6, which provides you a more accurate way to write the functions in JavaScript. They allow us to write smaller function syntax. Arrow functions make your code more readable and structured.

Function Hoisting

As variable hoisting, we can perform the hoisting in functions. Unlike the variable hoisting, when function declarations get hoisted, it only hoists the function definition instead of just hoisting the name of the function.

Let us try to illustrate the function hoisting in JavaScript by using the following example:

**Example**

In this example, we call the function before writing it. Let's see what happens when we call the function before writing it.

1. hello();
2. function hello() {
3. console.log("Hello world ");
4. }

**Output**

Hello world

In the above code, we have called the function first before writing it, but the code still works.

However, function expressions cannot be hoisted. Let us try to see the illustration of hoisting the function expressions in the following example.

1. hello();
3. var hello = function() {
4. console.log("Hello world ");
5. }

When you execute the above code, you will get a **"TypeError: hello is not a function."** It happens because function expressions cannot be hoisted.

**Output**

TypeError: hello is not a function

The Function Constructor

It is the way of defining a new function. The function statement is not the single way to define a new function; we can also dynamically define the function by using the **Function() constructor** along with the **new** operator.

It is less efficient than declaring a function using a **function expression** or **function statement.**

**Syntax**

The syntax of creating the function by using the **Function() constructor.**

1. var variable\_name = **new** Function(arg1, arg2..., "Body of the Function");

**Parameters used**

Its syntax includes:

**arg1, arg2, … argN:** These are the names that are used by the function as the name of the formal arguments. Each argument must be a string that corresponds to a valid identifier of JavaScript.

**Function Body:** It is a string that contains the [statements of JavaScript](https://www.javatpoint.com/javascript-if) comprises the function definition.

It can include any number of string arguments. The last argument is the body of the function, which can contain arbitrary statements of JavaScript, which is separated from each other by semicolons.

The function() constructor does not pass any argument that specifies a name for the function it creates.

**Example**

1. // creating a function that takes two arguments and returns the product of both arguments
2. var mul = **new** Function('a', 'b', 'return a \* b');
4. // calling of function
5. console.log("The product of the numbers is: " +mul(5, 5));

**Output**

The product of the numbers is: 25

Immediately Invoked Function Expression (IIFE)

It is a function in JavaScript that runs as soon as it is defined. An **IIFE (Immediately Invoked Function Expression)** can be used for avoiding the variable hoisting from within the blocks. It allows the public access to methods while retaining the privacy for variables defined in the function.

You can learn more about the IIFEs by clicking on this link [Immediately Invoked Function Expression (IIFE)](https://www.javatpoint.com/es6-immediately-invoked-function-expression)

JavaScript Functions and Recursion

When a function calls itself, then it is known as a **recursive function.** Recursion is a technique to iterate over an operation by having a function call itself repeatedly until it reaches a result.

It is the best way when we require to call the same function regularly with different parameters in a loop.

**Example**

1. function fact(num) {
2. **if** (num <= 1) {
3. **return** 1;
4. } **else** {
5. **return** num \* fact(num - 1);
6. }
7. }
8. console.log(fact(6));
9. console.log(fact(5));
10. console.log(fact(4));

**Output**

720

120

24

Function Expression v/s Function Declaration

The fundamental difference between both of them is that the function declarations are parsed before the execution, but the function expressions are parsed only when the script engine encounters it during the execution.

Unlike the function declarations, function expressions in JavaScript do not hoists. You cannot use the function expressions before defining them.

The main difference between a function declaration and function expression is the name of the function, which can be omitted in the function expressions for creating the anonymous functions.