Function

A function is a group of reusable code which can be called anywhere in your program. This eliminates the need of writing the same code again and again. It helps programmers in writing modular codes. Functions allow a programmer to divide a big program into a number of small and manageable functions. Like any other advanced programming language, JavaScript also supports all the features necessary to write modular code using functions. You must have seen functions like alert and write in the earlier chapters. We were using these functions again and again, but they had been written in core JavaScript only once. JavaScript allows us to write our own functions as well. This section explains how to write your own functions in JavaScript. Function Definition Before we use a function, we need to define it. The most common way to define a function in JavaScript is by using the function keyword, followed by a unique function name, a list of parameters thatmightbeempty, and a statement block surrounded by curly braces.

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.

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 `{}`.

```
// declaring a function named greet()
function greet() {
```

```
console.log("Hello there");
}
```

Calling functions

Defining a function does not *execute* it. Defining it names the function and specifies what to do when the function is called.

Calling the function actually performs the specified actions with the indicated parameters. For example, if you define the function square, you could call it as follows:

```
square(5);
```

The preceding statement calls the function with an argument of 5. The function executes its statements and returns the value 25.

Functions must be *in scope* when they are called, but the function declaration can be hoisted (appear below the call in the code). The scope of a function declaration is the function in which it is declared (or the entire program, if it is declared at the top level).

The arguments of a function are not limited to strings and numbers. You can pass whole objects to a function.

The showProps() function (defined in Working with objects) is an example of a function that takes an object as an argument.

A function can call itself. For example, here is a function that computes factorials recursively:

```
function factorial(n) {
if (n === 0 || n === 1) {
return 1;
} else {
return n * factorial(n - 1);
}
}
```

You could then compute the factorials of 1 through 5 as follows:

```
console.log(factorial(1)); // 1
console.log(factorial(2)); // 2
console.log(factorial(3)); // 6
console.log(factorial(4)); // 24
console.log(factorial(5)); // 120
```

There are other ways to call functions.

There are often cases where a function needs to be called dynamically, or the number of arguments to a function vary, or in which the context of the function call needs to be set to a specific object determined at runtime.

It turns out that *functions are themselves objects* — and in turn, these objects have methods. (See the Function object.) The call() and apply() methods can be used to achieve this goal.

Function hoisting

Consider the example below:

```
console.log(square(5)); // 25

function square(n) {

return n * n;

}
```

This code runs without any error, despite the square() function being called before it's declared. This is because the JavaScript interpreter hoists the entire function declaration to the top of the current scope, so the code above is equivalent to:

```
// All function declarations are effectively at the top of the scope
function square(n) {
return n * n;
}
console.log(square(5)); // 25
```

Function hoisting only works with function *declarations* — not with function *expressions*. The following code will not work:

```
console.log(square(5)); // ReferenceError: Cannot access 'square' before initialization
const square = function (n) {
```

```
return n * n;

};
```

Function scope

Variables defined inside a function cannot be accessed from anywhere outside the function, because the variable is defined only in the scope of the function. However, a function can access all variables and functions defined inside the scope in which it is defined.

In other words, a function defined in the global scope can access all variables defined in the global scope. A function defined inside another function can also access all variables defined in its parent function, and any other variables to which the parent function has access.

```
// The following variables are defined in the global scope

const num1 = 20;

const num2 = 3;

const name = "Chamakh";
```

```
// This function is defined in the global scope

function multiply() {

return num1 * num2;

}

console.log(multiply()); // 60



// A nested function example

function getScore() {

const num1 = 2;

const num2 = 3;

function add() {

return `${name} scored ${num1 + num2}`;

}

return add();

}

console.log(getScore()); // "Chamakh scored 5"
```

Example 1: Display a Text

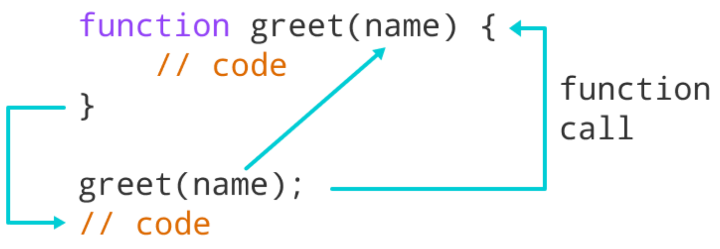
```
// program to print a text
// declaring a function
function greet() {
console.log("Hello there!");
}

// calling the function
greet();
```

Output

```
Hello there!
```

Function Parameters

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



Working of JavaScript Function with parameter

Example 2: Function with Parameters

```
// program to print the text
// declaring a function
function greet(name) {
console.log("Hello " + name + ":)");
}

// variable name can be different
let name = prompt("Enter a name: ");

// calling function
greet(name);
```

Output

Enter a name: Simon

Hello Simon :)

In the above program, the greet function is declared with a name parameter. The user is prompted to enter a name. Then when the function is called, an argument is passed into the function.

Note: When a value is passed when declaring a function, it is called parameter. And when the function is called, the value passed is called argument.

Example 3: 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);
```

Output

```
7

11
```

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.**

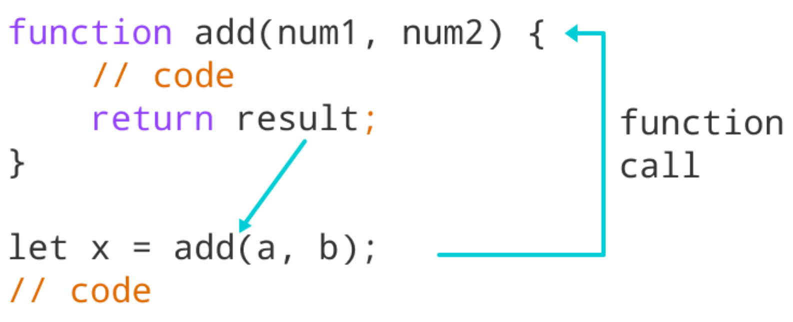
Notice that you can call a function as many times as you want. You can write one function and then call it multiple times with different arguments.

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
```

Working of JavaScript Function with return statement

Example 4: Sum of Two Numbers

```
// program to add two numbers
// declaring a function
function add(a, b) {
return a + b;
}

// take input from the user
let number1 = parseFloat(prompt("Enter first number: "));
let number2 = parseFloat(prompt("Enter second number: "));

// calling function
let result = add(number1,number2);

// display the result
console.log("The sum is " + result);
```

Run Code

Output

```
Enter first number: 3.4

Enter second number: 4

The sum is 7.4
```

In the above program, the sum of the numbers is returned by the function using the `return` statement. And that value is stored in the `result` variable.

Benefits of Using a Function

Function makes the code reusable. You can declare it once and use it multiple times.

Function makes the program easier as each small task is divided into a function.

Function increases readability.

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);
```

Output

```
16

9
```

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.

Types of functions in javascript?

- **Named function**
- **Anonymous function**
- **Immediately invoked function expression.**

JavaScript Function Methods

Method	Description
apply()	It is used to call a function contains this value and a single array of arguments.
bind()	It is used to create a new function.

call()	It is used to call a function contains this value and an argument list.
toString()	It returns the result in a form of a string.

JavaScript Anonymous Functions

The meaning of the word 'anonymous' defines *something that is unknown or has no identity*. In JavaScript, an anonymous function is that type of function that has no name or we can say which is without any name. *When we create an anonymous function, it is declared without any identifier*. It is the difference between a normal function and an anonymous function. Not particularly in JavaScript but also in other various programming languages also. The role of an anonymous function is the same.

Here, in this section, we will get to know about the anonymous function and its role in JavaScript. We will also learn and discuss its implementation.

Implementation of an Anonymous Function

An example is shown below that will make us understand how to use an anonymous and why it is different from a normal function:

Example:

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

```
let x = function () {
console.log('It is an anonymous function');
};
x();
```

The above code is the implementation of the anonymous function where:

The function is created for displaying the message as its output.

We have used the **function** keyword, which is used when we create any function in JavaScript, and the function is assigned to a variable **x** using 'let'.

The main focused point is that there is no function we have declared before. It is just the keyword function and parenthesis. In the case of a normal function, we use to name

the function as shown in the below code example:

```
function normale() {
console.log('It is a normal function');
}
normale();
```

Here, we have created a normale () function, which is a normal function. It is the difference between an anonymous function and a normal function.

Finally, we have invoked the created function.

So, in this way the basic implementation of an anonymous function is done.

Use of Anonymous Functions in JavaScript

We can use the anonymous function in JavaScript for several purposes. Some of them are given below:

Passing an anonymous function to other function as its argument

We can also use an anonymous function as an argument for another function. To understand better, let's implement a code under which we will pass the anonymous function as an argument value for another function:

```
setTimeout(function () {
console.log('Execute later after 1 second')
}, 1000);
```

The above code implements the use of anonymous function as an argument to a new function where:

The function setTimeout () will output the anonymous function after a second.

We have created an anonymous function and passed it to the setTimeout () as its argument.

Inside it, when the code gets executed, it will print the statement after a second of the execution time.

It is one such implementation and use of the anonymous function.

Immediate execution of a function

In order to invoke and execute a function immediately after its declaration, creating an anonymous function is the best way. Let' see an example to understand how we can do so:

```
(function() {
console.log('Hello');
})();
```

In the above code, the anonymous function is invoked immediately where it works as described in the following way:

The first step is to define the function expression, as shown below:

```
(function() {
console.log('Hello');
})
```

After defining the function, we can see the trailing parenthesis () followed by the terminator (;) that are used for invoking the defined function as shown below:

```
(function() {
console.log('Hello');
})();
```

In this way, the anonymous function can be invoked immediately.

Note: One can also pass the arguments in the function and invoke it too.

These are some uses of an anonymous function that concludes that an anonymous function is the one with no name, can be invoked immediately and can be used as an argument value in a normal function definition.

JavaScript Function Expression

The Javascript **Function Expression** is used to define a function inside any expression. The Function Expression allows us to create an anonymous function that doesn't have any function name which is the main difference between Function Expression and Function Declaration. A function expression can be used as an IIFE (Immediately Invoked Function Expression)which runs as soon as it is defined. A function expression has to be stored in a variable and can be accessed using *variableName.* With the ES6 features introducing Arrow Function, it becomes more easier to declare function expression.

Syntax for Function Declaration:

function *functionName(x, y)* { *statements...* return *(z)* };

Syntax for Function Expression (anonymous):

let *variableName* = function*(x, y)* { *statements...* return *(z)* };

Syntax for Function Expression (named):

let *variableName* = function *functionName(x, y)*

{ *statements...* return *(z)* };

Syntax for Arrow Function:

let *variableName* = *(x, y)* => { *statements...* return *(z)* };

Note:

A function expression has to be defined first before calling it or using it as a parameter.

An arrow function must have a return statement.

The below examples illustrate the function expression in JavaScript:

Example 1: Code for Function Declaration

```
function callAdd(x, y) {
```

```
let z = x + y;

return z;

}

console.log("Addition : " + callAdd(7, 4));
```

Output:

```
Addition : 11
```

Example2: Code for Function Expression (anonymous)

```
let calSub = function (x, y) {

let z = x - y;

return z;

}


console.log("Subtraction : " + calSub(7, 4));
```

Output:

```
Subtraction : 3
```

Example 3: Code for Function Expression (named)

```
let calMul = function Mul(x, y) {

let z = x * y;

return z;

}


console.log("Multiplication : " + calMul(7, 4));
```

Output:

```
Multiplication : 28
```

Example 4: Code for Arrow Function

```
let calDiv = (x, y) => {

let z = x / y;

return z;

}


console.log("Division : " + calDiv(24, 4));
```

Output:

```
Division : 6
```

Arrow functions in JavaScript

Arrow functions{()=>}

are a clear and concise method of writing normal/regular Javascript functions in a more accurate and shorter way. **Arrow functions** were introduced in the ES6 version. They make our code more structured and readable.

Arrow functions are anonymous functions i.e. they are functions without a name and are not bound by an identifier. Arrow functions do not return any value and can be declared without the function keyword. They are also called **Lambda Functions**.

Advantages of Arrow functions:

Arrow functions reduce the size of the code

The return statement and functional braces are optional for single-line functions.

It increases the readability of the code.

Arrow functions provide a lexical this binding, which means that they inherit the value of “this” from the enclosing scope. This feature can be advantageous when dealing with event listeners or callback functions where the value of “this” can be uncertain.

Syntax:

```
const gfg= () => {
console.log("Hi Geek!");
}

gfg();
```

Arrow functions without parameters:

```
const gfg= () => {
console.log("Hi from shrikant");
}
```

```
gfg();
```

Output

```
Hi from shrikant!
```

Arrow functions with parameters:

```
const gfg= (x,y,z) => {

console.log(x+y+z)

}


gfg(10,20,30);
```

Output

```
60
```

Arrow functions with default parameters:

```
const gfg= (x,y,z=30) => {

console.log(x+ " "+ y +" "+ z);
}
gfg(10,20);
```

Output

```
10 20 30
```

Limitations of Arrow functions:

Arrow functions do not have the prototype property

Arrow functions cannot be used with the new keyword.

Arrow functions cannot be used as constructors.

These functions are anonymous and it is hard to debug the code.

Arrow functions cannot be used as generator functions that use the yield keyword to return multiple values over time.

Closure in JavaScript

A closure is a feature of JavaScript that allows inner functions to access the outer scope of a function. Closure helps in binding a function to its outer boundary and is created automatically whenever a function is created. A block is also treated as a scope since ES6. Since JavaScript is event-driven so closures are useful as it helps to maintain the state between events.

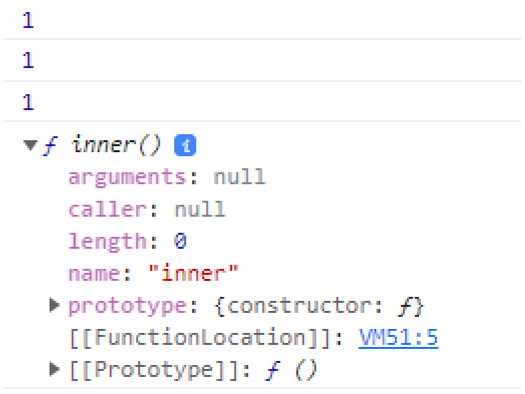
Prerequisite:: Variable Scope in JavaScript

Lexical Scoping: A function scope's ability to access variables from the parent scope is known as lexical scope. We refer to the parent function's lexical binding of the child function as "lexically binding."

Example 1: This example shows the basic use of closure.

```
function foo() {
let b = 1;
function inner() {
return b;
}
return inner;
}
let get_func_inner = foo();


console.log(get_func_inner());
console.log(get_func_inner());
console.log(get_func_inner());
```

Output: We can access the variable b which is defined in the function foo() through function inner() as the later preserves the scope chain of the enclosing function at the time of execution of the enclosing function i.e. the inner function knows the value of b through its scope chain. This is closure in action that is inner function can have access to the outer function variables as well as all the global variables.


```
1
1
1
▼ƒ inner() ℹ
    arguments: null
    caller: null
    length: 0
    name: "inner"
  ▶prototype: {constructor: ƒ}
    [[FunctionLocation]]: VM51:5
  ▶[[Prototype]]: ƒ ()
```

Closure in JavaScript

Definition of Closure:

In programming languages, closures (also lexical closures or function closures) are techniques for implementing lexically scoped name binding in languages with first-class functions. Operationally, a closure is a record storing a function[a] together with an environment:[1] a mapping associating each free variable of the function (variables that are used locally, but defined in an enclosing scope) with the value or reference to which the name was bound when the closure was created.[b] -Wikipedia

or

In other words, closure is created when a child function keep the environment of the parent scope even after the parent function has already executed

Note: Closure is the concept of function + lexical environment in which function it was created. so every function declared within another function then it has access to the scope chain of the outer function and the variables created within the scope of the outer function will not get destroyed.

Now let’s look at another example.

Example 2: This example shows the basic use of closure.

Javascript

```
function foo(outer_arg) {


function inner(inner_arg) {

return outer_arg + inner_arg;

}

return inner;

}

let get_func_inner = foo(5);


console.log(get_func_inner(4));
console.log(get_func_inner(3));
```

Output: In the above example we used a parameter function rather than a default one. Not even when we are done with the execution of **foo(5)** we can access the **outer_arg** variable from the inner function. And on the execution of the inner function produce the summation of **outer_arg** and **inner_arg** as desired.

```
9

8
```

Now let’s see an example of closure within a loop. In this example, we would store an anonymous function at every index of an array.

Example 3: This example shows the basic use of closure.

Javascript

```
// Outer function

function outer() {
```

```
let arr = [];

let i;

for (i = 0; i < 4; i++) {

// storing anonymous function

arr[i] = function () { return i; }

}


// returning the array.

return arr;

}


let get_arr = outer();


console.log(get_arr[0]());

console.log(get_arr[1]());

console.log(get_arr[2]());

console.log(get_arr[3]());
```

Output: Did you guess the right answer? In the above code, we have created four closures that point to the variable i which is the local variable to the function outer. **Closure doesn’t**

remember the value of the variable it only **points** to the variable or stores the **reference** of the variable and hence, returns the current value. In the above code when we try to update the value it gets reflected all because the closure stores the reference.

```
4
4
4
4
```

Let’s see the correct way to write the above code so as to get different values of i at different indexes.

Example 4: This example shows the basic use of closure

Javascript

```
// Outer function
function outer() {
function create_Closure(val) {
return function () {
```

```
return val;

}

}

let arr = [];

let i;

for (i = 0; i < 4; i++) {

arr[i] = create_Closure(i);

}

return arr;

}

let get_arr = outer();

console.log(get_arr[0]());

console.log(get_arr[1]());

console.log(get_arr[2]());

console.log(get_arr[3]());
```

Output: In the above code we are updating the argument of the function create_Closure with every call. Hence, we get different values of i at different indexes.

```
0

1
```

```
2

3
```

Note: It may be slightly difficult to get the concept of closure at once but try experimenting with closure in different scenarios like for creating getter/setter, callbacks, and so on.