Functions

Overview

JavaScript is based on **functional programming.** Therefore, functions are fundamental building blocks of JavaScript.
The function contains a set of statements that perform some task.

Syntax :
```
function functionName(parameters){
            // code
 }
```
Calling the function : functionName(arguments) ;

The function execution stops either when all the statements have been executed or a return statement is found. The return statement stops the execution of the function and returns the value written after the return keyword. . A function may or may not return some value after its execution.

Function Arguments

JavaScript is a dynamic language, and therefore it allows passing different numbers of arguments to the function and does not give error in this condition.

- **Passing fewer arguments** - In this case, when few arguments are passed, the other parameters that do not get any value assigned to them get value **undefined** by default.
- **Passing more arguments -** In this case, when more arguments are passed, the extra arguments are not considered.

Example :
```
function add(a, b, c) {
            return a+b+c ;
  }
console.log( add(10, 20) ) ; // Prints - NaN
console.log( add(10, 20, 30, 40) ) ; // Prints - 60
```

Default Parameters

If you pass fewer arguments than in the function, the remaining parameters default to **undefined in JavaScript**. But you can also set your default values to them.

Example :

```
function findInterest(p, r=5, t=1) {
            console.log( "Interest over", t, "years is:", (p*r*t)/100 );
 }
findInterest(1000); // Prints - Interest over 1 years is: 50
findInterest(1000, 7); // Prints - Interest over 1 years is: 70
findInterest(1000, 8, 2); // Prints - Interest over 2 years is: 160
```

Rest Parameters

The rest parameter syntax **(...variableName)** is used to represent an indefinite number of arguments. It is defined in an array-like structure.

Example: If we want to add **at least** 3 number

```
function addAtLeastThree(a, b, c, ...numbers) {
    var sum = a+b+c;
     for(var i=0; i<numbers.length; ++i) {
                sum += numbers[i];
         }
    return sum;
 }
```

❖ `console.log( addAtLeastThree(10, 20, 30, 40, 50) ) ;  // Prints - 150`

Hoisting

JavaScript provides an exciting feature called hoisting. This means that you can use a variable or function even **before it's a declaration.**

Hoisting is a mechanism in JavaScript where variables and function declarations are moved to the top of their scope before code execution.

NOTE: If you use a variable or function and do not declare them somewhere in the code, it will give an error.

Variable Hoisting

Variable hoisting means that you can use a variable even before it has been declared.

Example :
```
console.log(a); // Prints - undefined
//Other JavaScript Statements
 var a = 10;
```
- ❖ The above prints 'undefined' because only the variable declaration is moved to the top, not the definition.

Function Hoisting

Function hoisting means that you can use function even before function declaration is done.

Example :
```
console.log( cube(3) ); // Prints - 27
 //Other JavaScript Statements
function cube(n) {
      return n*n*n ;
 }
```

NOTE: You cannot use function hoisting when you have used function expression. If you use function expression and use function hoisting, then it will result in an error.

Function within Function

You can define a function within a function which we can also call a **nested function.** The nested function can be called inside the **parent** function only.

The nested function can access the variables of the parent function and as well as the global variables.
But the parent function cannot access the variables of the inner function.

This is useful when you want to create a variable that needs to be passed to a function. But using a global variable is not good, as other functions can modify it. So, using **nested functions will prevent the other functions from using this variable.**

Example: Using a count variable that can only be accessed and modified by the inner function

```
function totalCount( ) {
        var count = 0;
        function increaseCount( ) {
                count++;
                }
        increaseCount( ) ;
        increaseCount( ) ;
        increaseCount( ) ;
        return count ;
        }
        console.log(totalCount( )); // Prints - 3
```

Scope Chain

This tells how do interpreters look for a variable. This is decided according to the Scope Chain.

When a variable is used **inside a function**, the JavaScript interpreter looks for that variable inside the function. If the variable is not found there, it looks for it in the outer scope, i.e., the parent function. If it is still not found there, it will move to the outer scope of the parent and check it; and do the same until the variable is not found. The search goes on until the global scope, and if the variable is not present even in the global scope, the interpreter will throw an error.

Example :

```
 function a( ) {
    var i = 20;
    function b( ) {
        function c( ) {
                console.log(i); // Prints - 20
        }
        c( );
    }
    b( );
 }
a( );
```

Function Definition and Function Expression

Functions in JavaScript can be defined in two ways –

- **Function Definition -** Creating a function using function keyword and function name.
- **Function Expression -** Creating a function as an expression and storing it in a variable.

Function Definition

A function definition is creating a function in a normal way, which we have read until now.
Here, the parameters take values differently for different types of variables. We can either pass primitive value or non-primitive value.

- If the **value passed as an argument is primitive**, then it gets passed by value. This means that the changes to the argument does not reflect the changes globally and only remains local.

Example :
```
function abc(b) {
 b = 20;
 console.log(b); // Prints - 20
}
var a = 10;
abc(a);
console.log(a); // Prints - 10
```

- If the **value passed as an argument is non-primitive**, then it gets passed by reference. This means that change is visible outside the function.

Example :
```
function abc(arr2) {
arr2[1] = 50;
  console.log(arr2); // Prints - Array [10, 50, 30]
 }
var arr1 = [10, 20, 30];
abc(arr1);
console.log(arr1); // Prints - Array [10, 50, 30]
```

Function Expression

Variables can take primitive and non-primitive values. So the function is one of the possible values that a variable can have. A function expression is **used to assign the function to a variable.**

Example :
```
var add= function sum(a , b) {
                        return a + b ;
                         }
console.log(add( 2 , 4 )); // Prints – 6
```

However, note that the name **sum** that this function has can be used only inside the function to refer to itself; it can't be used outside the function.

The function expression, as shown above, is **named,** i.e. the function being assigned has a name. We can have anonymous function expressions as well, i.e. it does not have a name.

Example :
```
var factorial = function fac(n) {
            var ans = 1;
            for(var i = 2; i <= n; i++) {
                        ans *= i;
                        }
                        return ans ;
             }
console.log(factorial(3)) ; // Prints - 6
```

Passing Function As Argument (CallBack Function)

Functions in JavaScript are objects. So we can also pass a function as an argument to another function.

- Pass function as argument like - function abc(arg1, functionName);
- Define function as an argument like - function abc(arg1, function( ) {
                                                });

Example :

```
function abc(a, b, compute) {
                compute( a , b ) ;
        }
function multiple(a, b) {
        console.log(a*b) ;
        }
function add(a, b) {
        console.log(a+b) ;
        }

abc(5, 2, multiple) ; // Prints - 10
abc(5, 2, add) ; // Prints - 3
```

The function passed is also called the **callback function**. A callback is a code that is passed as an argument to another code, which is expected to execute the argument(function) at some convenient time.

NOTE: JavaScript is an event-driven language, meaning that instead of waiting for a response from a function, it keeps executing the code in sequence. So if you want to execute something after some line of code, then callbacks are very useful.

Arrow Function

Arrow functions allow us to write shorter function syntax :

Before :

```
print = function( ) {
            return “Coding Ninjas”;
        }
```

After :

```
print = ( ) => {
        return “Coding Ninjas” ;
    }
```

If the function has only one statement, and the statement returns a value, you can remove the brackets and the return keyword

```
print = ( ) => “Coding Ninjas” ;
```

Note: This works only if the function has only one statement.

Arrow Function With Parameters

If the function returns a single statement

```
sum = ( a, b ) => a+b ;
console.log( sum(2,3) ) ;  // 5
```

If the function returns multiple statements

```
sum = ( a, b ) => {
    console.log(a+2) ;
    console.log(b+2) ;
}

sum(2,3) ;

Output : 4
         5
```