Topic : Functions and Arrays

1. FUNCTIONS

JavaScript is based on **functional programming**. Therefore, functions are fundamental building blocks of JavaScript.

Function contains set of statements that ***perform some task.***

You define a function using **'function'** keyword. The syntax for creating a function is –

```
function functionName(parameters) {
  // SET OF STATEMENTS
}
```

You can call the above function as –

```
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 number of arguments to the function and does not give error in these condition -

- **Passing fewer arguments** - In this case, when few arguments are passed, the other parameters that does 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.

  Eg., when you have a add function given below –

```
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
```

1.1. Arguments Object

Although we will study about Objects in next session, but in context to functions you can have access to arguments passed to function.

The **argument** object is used to store the arguments passed to the function in an array like object. You can either use the parameter name or argument object to access the values. It is helpful in cases when you don't know the number of arguments passed to the function.

The number of arguments can be found using - **"arguments.length"**. The arguments can be accessed using the brackets notation as used in arrays - **"arguments[i]"**, where ***'i'*** is a number starting from 0.

Eg., the below code prints all the values of the passed to function –

```
function printAll() {
    for(var i=0; i<arguments.length; ++i) {
        console.log(arguments[i]);
    }
}

printAll('mango', 'apple');          // Prints - mango  apple
printAll('fire', 'water', 'ice', 'gas');    // Prints - fire
          water  ice  gas
```

1.2. Default Parameters

In JavaScript, if you ***pass less arguments*** than in the function, the ***remaining parameters*** defaults to ***'undefined'***. But you can also set your own default values to them.

Eg., the below function uses the default parameters when their values is not passed –

```
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
```

1.3. Rest Parameters

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

Eg., let's say in a situation when we want to add at least three numbers, then we can use something like the below function –

```
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
```

1.4. Hoisting

JavaScript provides a very interesting feature called as **hoisting**. This means that you can use a variable or function even before it's 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, then it will give an error.*

You can use hoisting with both variables and function as shown below-

1.4.1. Variable Hoisting

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

Eg., you can use variable as -

```
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 and not it's definition.

1.4.2. Function Hoisting

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

Eg., you can use function like this –

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

But 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.

2. SCOPE

Scope of a variable is part of code where that variable is accessible. **Scope of variable** depends where they are defined.

2.1. Global Scope

When a variable is ***defined*** *globally(i.e. not in any function)****, it can be used anywhere in the code.*** For eg.,

```
var i = 10;
function abc() {
    console.log(i);          // This will print 10
}
console.log(i);              // This will print 10
```

2.2. Function Scope

When a variable is ***defined*** inside a function, it is accessible only within that function.

Eg., see the below function –

```
var i = 0;
function abc() {
    var j = 1;
    console.log(i);         // This will print 0
    console.log(j);         // This will print 1
}

console.log(j); // This statement will throw an error as scope of
j is only within function abc.
```

Now let's see what happens when there are different variables with same name in different scopes. For eg.,

```
var i = 0;
function abc() {
    var i = 1;
    console.log(i);
}
```

Here, within function 'abc' we have 2 variables 'i' i.e. one whose scope is global and other whose scope is within function 'abc'.

Output of the above code will be **1** as preference will be given to variable that is in local scope(here scope of function 'abc'). If we remove line ***'var i = 1;'***, then the interpreter will look outside the function and output will be **0**.

3. FUNCTION WITHIN FUNCTION

You can define function within a function which we can also call as **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 to use this variable***.

Eg., using a count variable which 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
```

3.1. Scope Chain

Lets now discuss, how does interpreter looks 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, then it looks for it in the outer scope i.e. in the parent function. If it is still not found there, it will move to the outer scope of parent and check it; and do the same until the variable is not found. The search goes on till the global scope and if the variable is not present even in global scope then interpreter will throw error.

Eg., the below function shows how a variable is accessed when used inside a inner function -

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

```
}
a();
```

4. FUNCTION DECLARATION AND 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.

We have discussed function declaration and expression in detail below -

4.1. Function Definition

A function definition is creating a function in a normal way, which we have read until now. The syntax is -

```
function functionName(parameters) {
  // SET OF STATEMENTS
}
```

Here, the parameters take values differently for different types of variable. 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.
  Eg.,

```
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.
  Eg.,

```
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]
```

4.2. Function Expression

We have discussed that variables can take primitive and non primitive values. So function is one of the possible values that a variable can have. Function expression is used to assign the function to a variable.

Eg, the below code uses the function expression with function name –

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

However, note that the name "fact" 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 has a name.

The syntax is –

```
var variableName = function (parameters) {
  // SET OF STATEMENTS
}
```

Eg.,

```
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
```

5. PASSING FUNCTION AS ARGUMENT

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

There are two ways to pass a function –

- First, pass function as argument like - `function abc(arg1, functionName);`
- Second, define function as an argument like - `function abc(arg1, function() { ... });`

Example,

```
function abc(a, b, compute) {
    compute();
}

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 **callback function**. A callback is a piece of code that is passed as an argument to other code, which is expected to execute the argument(function) at some convenient time.

But ***why do we need to use callbacks?*** - 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. We will see callbacks in upcoming sections.

6. ARRAYS

Array is an **ordered collection of data**(can be primitive or non-primitive), used to store multiple values. This helps in storing indefinite number of values.
Each item/value has an **'index'** attached to it, using which we can access the values. In JavaScript ***'index'*** starts at ***0***.

The array also contains a property **'length'** that stores the number of elements present inside the array. It changes its value dynamically as the size of the array changes.

6.1. Creating an Array

There are two ways to create an array -

- **Using square brackets** - We can create an empty array as - `var arrayName = []` and array with initial values as - `var arrayName = [value1, value2, …, valueN]`
- **Using Array Object** - We can create an empty array as - `var arrayName = new Array()` and array with some length as - `var arrayName = new Array(N)`, where ***'N'*** is length of array.

Another way to create an array using 'Array' object is providing the values in it like - `var arrayName = new Array(value1, value2, …, valueN)`. This will create an array with these elements.

Array is heterogeneous, meaning it can ***contain different types of value*** at the same time. Also the array can store primitive and non-primitive values.
Eg., `var array = ["hammer", 85, {name: "Preeti"}, [0, 2, 6]]`

6.2. Accessing Element in Array

You can **access the individual elements** of the array **using the square bracket notation** like - `array[1]` will return ***'85'***.
This also allows you to **modify the value** like - `array[1] = 20` and the array now becomes - `["hammer", 20, {name: "Preeti"}, [0, 2, 6]]`

When using **array inside an array**, you can access the value of inner array directly like - `array[3][1]` will return ***'2'***.

If you ***access the array outside it's range***, i.e. you pass an invalid index(whether negative or greater than the length of array), then ***'undefined'*** is returned.

6.3. Placing Elements at Outside Array Range

When you use index outside the range to add elements to an array, the array behaves in a different manner.

When a value is assigned to ***positive index outside the range of array***, then the array stores this value at the specified index and all other indices before it are empty. The length of the array also changes to 'index+1'.

When a ***negative value is used***, then the array ***stores the element as a key-value pair***, where the negative index is the key and the element to be inserted is the value.

You can play around it and check the output.

EXTRA:

You must give a read to the below links on how arrays are stored in JavaScript - ***https://stackoverflow.com/questions/20321047/how-are-javascript-arrays-represented-in-physical-memory***

7. FUNCTIONS ON ARRAYS

Below are some important methods that can be used on arrays -

7.1. push Method

The **'push()'** method is used to ***add one or more elements to the end of the array*** and ***returns the new length of array***. It uses ***'length'*** property to add element.
If the array is empty, then ***'push()'*** method will create ***'length'*** property and also add element to it.
In case, you are adding multiple elements, separate them using a **comma**(**,**).

The syntax is - `array.push(element1, ... , elementN)`

7.2. pop Method

The **'pop()'** method is used to ***remove the last element from the array*** and ***return that element***. It also decreases the ***'length'*** of array by 1.
Using pop on an empty array returns ***'undefined'***.

The syntax is - `array.pop()`

7.3. shift Method

The **'shift()'** method is used to ***remove the first element from an array*** and ***return that element***.
If the ***'length'*** property is 0, ***'undefined'*** is returned.

The syntax is - `array.shift()`

7.4. unshift Method

The **unshift()'** method is used to ***add one or more elements to the beginning of the array*** and ***returns the new length of array***.
In case, you are adding multiple elements, separate them using a **comma(,)**.

The syntax is - `array.unshift(element1, ... , elementN)`

7.5. indexOf Method

The **'indexOf()'** method is used to ***return the first index at which the given element is found*** in the array. If the element is not found, then '-1' is returned.

By default the whole array is searched, but ***you can provide the start index from which the search should begin***. It is **optional**. If the index provided is negative, then the offset is set from the end of the array and search in opposite direction is done.

The syntax is - `array.indexOf(element, fromIndex)`

You can check the examples from the below link -
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/indexOf#Examples

7.6. splice Method

The **'splice()'** method is used to ***remove or replace or add element to an array***. If an element is removed, it returns the array of deleted elements. If no elements are removed, an empty array is returned.

The syntax is - `array.splice(start, deleteCount, item1, …, itemN)`

You can check the examples of adding, removing and replacing from the below link -
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/splice#Examples

7.7. reverse Method

The **'reverse'** method is used to ***reverse the content of the array*** and ***return the new reversed array.*** This means that the first element becomes last and vice-versa.

The syntax is - `array.reverse()`

7.8. sort Method

The **'sort()'** method is used to ***sort the elements of array*** and ***return the sorted array***. The sort is done by converting the elements to string and then comparing them.

The syntax is - `array.sort([compareFunction])`

Here, ***'compareFunction(a, b)'*** is optional and you can provide it to sort array according to the defined function.

You can visit the below link, to know more about 'compareFunction' -
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/sort#Description

7.9. join Method

The **'join()'** method is used to ***concatenate all the elements in an array*** and ***return a new string***.
By default, they are separated by **comma**(**,**), but you can also provide your own separator as a string. It is optional to provide a separator.

The syntax is - `array.sort(separator)`

If an element is ***'undefined'*** or ***'null'***, then it is converted into empty string.

7.10. toString Method

The **'toString()'** method is used to ***return the array in the form of a string***. The string contains all the elements separated by comma.

The syntax is - `array.toString()`

EXTRA:

You can find other methods that array uses from the below link - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array#Methods_2

8. ITERATING OVER ARRAYS

Iterating over the arrays help in accessing the values and also manipulate each one of them individually, using a lesser line of code. We have used two methods to iterate over the arrays -

8.1. For loop

The **'for'** loop is used normally to iterate over all the values of the array.

Eg.,

```
var arr = [10, 20, 30];
for(var i=0; i<arr; ++i) {
    console.log(arr[i]*2);
}
```

The above code will print the array values as doubled - 20 40 60

8.2. forEach Method

The **'forEach()'** method calls a function once for each array element.

The syntax is -

```
arr.forEach(function callback(currentValue, index, array) {
    /* Function Statements */
}, thisArg);
```

You can either provide function definition as shown in the syntax above. Or you can pass function name to it.

Eg., the below code will print the values on the console -

```
var items = ['apple', 'banana', 'orange'];
items.forEach(function(item) {
    console.log(item);
})
```

You can use the below link to find in detail about the parameters passed to the forEach method -
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach#Description