FUNCTIONS

In JavaScript, a function is a block of code that is designed to perform a specific task or to calculate a value. Functions are one of the fundamental building blocks of JavaScript programming and are used to organize code into reusable components.

BENEFITS OF FUNCTION

- *Faster development.*
- *Several programmers can work on individual programs at the same time.*
- *Easy debugging and maintenance.*
- *Easy to understand as each module works independently to another module.*
- *Less code has to be written.*
- *The scoping of variables can easily be controlled.*
- *Modules can be re-used, eliminating the need to retype the code many times*

DEFINING THE FUNCTION

Syntax

```
function functionName(Parameterlist) {
        Task;
}
```

It consists of the function keyword, followed byt the name of the function. A list of parameters to the function, enclosed in parentheses and separated by commas.The JavaScript statements that define the function, enclosed in curly brackets, { / … */ }.*

Note:

1. Parameters are essentially passed to functions by value — so if the code within the body of a function assigns a completely new value to a parameter that was passed to the function, the change is not reflected globally or in the code which called that function.
2. When you pass an object as a parameter, if the function changes the object's properties, that change is visible outside the function

```
<script>
function test(x){
x = x + 1
}
y= 5
test(y)
document.write(y)
</script>
```

e.g.1

```
<script>
 function welcome() {
  console.log("function");
 }
  // Function calling
 welcome();
</script>
```

CALLING FUNCTIONS

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

```
welcome();
```

e.g.
```
<script>
 function add(a, b) {
  return a + b;
 }

 let result = add(3, 5);
 console.log(result);

</script>
```

Return Value

A return statement is an optional part of a JavaScript function. If you wish to return a value from a function, you must do this. This should be the final statement of a function.

e.g.
```
    <script>
        function square(n) {
                s = n * n
                return s
        }

        num = parseInt(prompt("Enter a number")) ;
        x = square(num);
        document.write("square of " + num + " is " + x);

    </script>
```

ASI

In JavaScript, semicolons are used to terminate statements. While it's a good practice to include semicolons to explicitly indicate the end of a statement, JavaScript is forgiving in certain cases where semicolons are omitted. This is known as Automatic Semicolon Insertion (ASI).

ASI is a feature in JavaScript where the interpreter automatically inserts semicolons at the end of statements if they are missing.

Note: If we do not return the value in function then it us ***undefined***

Function expressions

A function expression in JavaScript is a way to define a function as part of an expression, rather than as a standalone declaration. In a function expression, a function is created and assigned to a variable or passed as an argument to another function.

e.g.
```
const add = function(a, b) {
  return a + b;
};
```

Function expressions are commonly used in scenarios :

- **Assigning Functions to Variables:** You can assign a function to a variable and use that variable to call the function.

- **Passing Functions as Arguments:** You can pass a function as an argument to another function.

- **Immediately Invoked Function Expressions (IIFE):** You can define a function expression and immediately invoke it.

e.g.

```
<script>
 // Assigning a function to a variable
 const addition = function (x, y) {
 return x + y;
 };
 console.log(addition(3, 4));

 // Assigning a function to a variable
 const multiply = function (x, y) {
 return x * y;
 };

 console.log(multiply(3, 4)); // Output: 12

 // Passing a function as an argument
 function calculate(operation, x, y) {
 return operation(x, y);
 }

 console.log(calculate(multiply, 5, 6));
 console.log(calculate(addition , 5, 6));

</script
```

Note

However, a name can be provided with a function expression. Providing a name allows the function to refer to itself(recursion), and also makes it easier to identify the function in a debugger's stack traces

	Function Declaration	Function Expression
1.	A function declaration must have a function name.	A function expression is similar to a function declaration however you can also create without the function name.
2.	Function declaration does not require a variable assignment.	Function expressions can be stored in a variable assignment.
3.	The function in function declaration can be accessed before and after the function definition.	The function in function expression can be accessed only after the function definition.
4.	**Function declarations are hoisted**	**Function expressions are not hoisted**

HOISTING

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

function square(n) {
  return n * n;
}
```

Hoisting

```
function square(n) {
  return n * n;
}
console.log(square(5)); // 25
```

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
```

Note:

Function hoisting only works with function declarations — not with function expressions. The code below will not work.

```
console.log(square); // ReferenceError: Cannot access 'square' before initialization

const square = function (n) {
  return n * n;
}
```

Note:

Hoisting is a kind of default behaviour in which all the declarations either variable declaration or function declaration are moved at the top of the execution scope just before executing the program's code.

e.g.

```
<script>
  var b = 20;
  document.write(a);
  document.write(b);
  var a = 10 ;
  document.write(a);
</script>
```

FUNCTION SCOPE

Function scope refers to the visibility and accessibility of variables and functions within a particular function. In JavaScript, variables and functions declared inside a function are scoped to that function, meaning they are only accessible within the body of that function. This concept is also known as local scope.

e.g.

```
<script>
  function f_scope(n) {
    let s = n * n;
    return s;
  }
  k = f_scope(5);
  document.write(k);
</script>
```

NESTED FUNCTIONS: *Nested functions refer to functions defined within another function.*

e.g.1

```
<script>
  function addSquares(a, b) {
    function square(x) {
    return x * x;
    }
    return square(a) + square(b);
  }
  const a = addSquares(2, 3); // returns 13
  const b = addSquares(3, 4); // returns 25
  const c = addSquares(4, 5); // returns 41
  document.write(a + "<br>");
  document.write(b + "<br>");
  document.write(c + "<br>");
</script>
```

Inner function forms a closure

Nested functions forms a closures, where inner functions "remember" the environment in which they were created, even after the outer function has finished executing. This can be particularly useful for creating modular and reusable code.

```
<script>
  function outside(x) {
    function inside(y) {
      document.write(x + " , " + y )
    }
    return inside;
  }

  const r_inside = outside(10);
  r_inside(4)
  r_inside(15)
  outside(20)(5)
</script>
```

e.g. 2

```
<script>
  function A(x) {
  function B(y) {
    function C(z) {
      return x + y + z;
    }
    return C(3);
  }
  return B(2);
 }

 let x = A(1); // Logs 6 (which is 1 + 2 + 3)
 document.write(x);
</script>
```

Name conflicts

When two arguments or variables in the scopes of a closure have the same name, there is a name conflict. More nested scopes take precedence. So, the innermost scope takes the highest precedence, while the outermost scope takes the lowest. This is the scope chain. The first on the chain is the innermost scope, and the last is the outermost scope.

e.g.

```
<script>
 function outside(x) {
  function inside(x) {
  return x * 2;
  }
  return inside;
 }

 let x = outside(5)(10); // returns 20 instead of 10
 document.write(x);
</script>
```

Default parameters

Default parameters in JavaScript allow you to specify default values for function parameters. This means that if a parameter is not provided when the function is called, the default value will be used instead. Default parameters were introduced in ECMAScript 6 (ES6).

e.g.

```
<script>
 function multiply(a, b = 1) {
  return a * b;
 }

 let x = multiply(5);
 document.write(x + " <br>");
 let y = multiply(5, 9);
 document.write(y);
</script>
```

JavaScript Rest Parameters

In JavaScript, rest parameters allow you to represent an indefinite number of arguments as an array within a function definition. This enables you to work with an arbitrary number of arguments passed to a function, making your code more flexible.

e.g.

```
<script>
  function sum(...input) {
    let sum = 0;
    for (i = 0; i < input.length; i++) {
      sum = sum + input[i];
    }

    return sum;
  }
  document.write(sum(1, 2) + "<br>");
  document.write(sum(1, 2, 3) + "<br>");
  document.write(sum(1, 2, 3, 4, 5) + "<br>");
  document.write(sum(1, 2, 3, 4, 5) + "<br>");
  document.write(sum(1, 2, 3, 4, 5, 6, 7, 8, 9, 5, 56, 67, 5, 6, 67));
</script>
```

Arrow function

An arrow function in JavaScript is a concise way to write **function expressions**. It provides a more compact syntax compared to traditional function expressions, especially when writing small anonymous functions. Arrow functions are commonly used in modern JavaScript codebases, especially for short, simple functions like event handlers, array methods, and callbacks.

```
<script>
 // normal function
 function add ( a , b ){
 return a+ b ;
 }

 //function expression
 const add = function ( a , b ){
   return a+ b ;
 }


 //arrow function
 const add = ( a , b )=>{
   return a+ b ;
 }


 const add = (a, b) => a + b;
</script>
```

Arrow functions have a few key characteristics

- **Concise syntax:** Arrow functions have a shorter syntax compared to traditional function expressions, which can make your code more readable and less verbose.

- **Implicit return:** If the function body consists of a single expression, you can omit the braces {} and the return keyword. The result of the expression will be implicitly returned.
  const add = (a, b) => a + b;

e.g.

```
<script>
 //expression function
 const square1 = function (n) {
   return n * n;
 };
```

```
  //Arrow function
  const square2 = (n) => {
    return n * n;
  };

  document.writeln(square1(5));
  document.writeln(square2(10));
</script>
```

// function expression

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

can be written as

// using arrow functions

```
let x = (x, y) => x * y;
```

Arrow Function Syntax

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

- **Arrow Function with No Argument**
  - If a function doesn't take any argument, then you should use empty parentheses.
    - let greet = () => console.log('Hello');
    - greet(); // Hello

- **Arrow Function with One Argument**
  - If a function has only one argument, you can omit the parentheses.
    - 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.

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

- **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
```

Serial no.	Difference	Normal function	Arrow function
1	**Syntactic**	function regular() {}	let arrow = () => {}

2	***Arrow function have lexical this whereas normal function does not***	<script>
		  let person ={
		    name :"target",
		    getname : function (){
		      console.log(this)
		      function local(){
		      console.log(this)
		      }
		      local()
		      local.call(this)
		    }
		  }
		  person.getname();
		  // const win = person.getname ;
		  // console.log(win())
		</script>
			<script>
			  let person ={
			    name :"target",
			    getname : function (){
			      console.log(this)
			      const local= ()=>{
			      console.log(this)
			      }
			      local()
			      local.call(this)
			    }
			  }
			  person.getname();
			  // const win = person.getname ;
			  // console.log(win())
			</script>

3	We can create factory method/constructor function using normal function but can not create using arrow function	<script>
		  function person(name,age) {
		    this.name = name
		    this.age = age
		  }
		  let ram = new Person('ram',20)
		console.log(ram)
		</script>
			<script>
			  const person =(name,age)=> {
			    this.name = name
			    this.age = age
			  }
			  let ram = new person('ram',20)
			console.log(ram)
			</script>

4	Hoisted	Hoisted	Not Hoisted

5.	Argument Object	We can access arguments object
		<script>
		  function sum() {
		
		console.log(arguments); // Accessing the arguments object
		
		    let total = 0;
		    for (let i = 0; i < arguments.length; i++) {
			We cannot access arguments object
			<script>
			  const sum = () => {
			    // Arrow function does not have access to the arguments object
			
			console.log(arguments); // Error: ReferenceError: arguments is not defined

		total +=
		arguments[i];
		}
		
		return total;
		}
		
		console.log(sum(2, 4, 6,23,5,3,5,35)); // Output: 12
		</script>
			let total = 0;
			for (let i = 0; i < arguments.length; i++) {
			// Error: ReferenceError: arguments is not defined
			total += arguments[i];
			}
			
			return total;
			};
			
			console.log(sum(2, 4, 6)); // Output: Error
			</script

CALL BACK FUNCTION

In JavaScript, you can also pass a function as an argument to a function. This function that is passed as an argument inside of another function is called a callback function.

e.g.

```
// function
function greet(name, callback) {
   console.log('Hi' + ' ' + name);
   callback();
}

// callback function
function callMe() {
   console.log('I am callback function');
}

// passing function as an argument
greet('Peter', callMe);
```

Akhilesh Kumar Gupta
(Technical Training specialist)
TechCarrel LLP