12 – Functions

Functions are one of the fundamental building blocks of JavaScript. A function is similar to a protocol – a set of statements that performs a task and returns a value. However, for a set of statements to qualify as a function, it should take some input and return an output wherein the input and output have some form of a relationship.

**Function Declarations**

A function definition, also called a function declaration, consists of the function keyword, followed by:

* The name of the function
* A list of the parameters (inputs) enclosed in parentheses
* The JS statements that define the function are enclosed in braces

function square(number) {

return number \* number;

}

**Function Expressions**

Another way of creating a function. In function expressions, having a name for the function is optional.

const square = function (number) {

return number \* number;

};

However, a name can be provided with a function expression. This allows a function to refer to itself, and also makes it easier to track in the debugger.

const factorial = function fac(n) {

return n < 2 ? 1 : n \* fac(n - 1);

};

**Function Hoisting**

Function hoisting refers to invoking/referring of a function before its declaration. This is possible because the JavaScript interpreter hoists the entire function declaration to the top of its scope.

However, function hoisting is possible only with function declarations – not with function expressions.

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

function square(n) {

return n \* n;

}

**Immediately Invoked Function Expressions (IIFE)**

An Immediately Invoked Function Expression (IIFE) is a function that is defined and executed immediately after it is created. It can be written for all three forms of writing a function: a function declaration, a function expression, and an arrow function.

Syntax: (function() { ... })();

**Advantages of IIFEs**

1) Encapsulation/Data privacy: Variables inside an IIFE are confined within the scope they are defined and do not pollute the global scope.

2) Run-once Initialization Code: Set-up code required to run only once is perfect when defined as IIFEs.

**Arrow Functions**

An arrow function expression is an alternative to a regular function expression with semantic differences and deliberate limitations:

* An arrow function doesn’t have its own bindings to “this”, “arguments”, or “super”. Thus, as they do not have a “this” binding, they shouldn’t be used as a method.
* Arrow functions cannot be used as constructors. Calling them with new throws a TypeError.

Syntax:

(param1,…,paramN) => {

statements

}

Example 1

a => a + 100;

The parentheses can only be omitted around the parameters if the function takes a single simple parameter. If it has multiple parameters, no parameters, default, destructured, or rest parameters, the parentheses around the parameter list are required.

The braces can only be omitted if the function directly returns an expression. If the body has statements, braces are required – and so is the return keyword.

**Expression v/s Statement**

An expression produces a value. Whereas, a statement performs an action, i.e., a complete unit of execution.

Expression – “Shaurya”.toUpperCase() // Produces a value (One that can be returned).

Statement – console.log(“Shaurya”) // Performs a task (No value returned).

const func = (x) => x \* x;

// expression body syntax, implied "return"

const func2 = (x, y) => {

return x + y;

};

// with block body, explicit "return" needed

**Closure**

Closure is a concept that refers to the combination of a function bundled together with variables from its lexical environment, even after the outer scope has finished executing.

Why Are Closures Beneficial?

1) Encapsulation/Data Privacy: It allows you to create private variables that cannot be accessed or modified from the outside.

Example:

function createCounter() {

let count = 0; // private variable

return function() {

count++;

return count;

};

}

const counter = createCounter();

console.log(counter()); // 1

console.log(counter()); // 2

console.log(counter()); // 3

In the above example, the variable “count” cannot be accessed from outside the function, nor can its value be changed arbitrarily other than through the function defined to increase its count by one.

2) Preserve State in Async Functions

Closures let you preserve variables across time in callbacks used to write asynchronous code.

Example:

function delayedMessage(msg, delay) {

setTimeout(function() {

console.log(msg);

}, delay);

}

delayedMessage("Hello after 1 second", 1000);

In the above example, the inner function remembers “msg” even after delayedMessage() has finished executing.

3) Modular and Clean Code

Closures help you write modules that expose only what’s necessary to the lexical environment (such as the user).

const userModule = (function() {

let username = "Shaurya"; // private – cannot be accessed from outside

return {

getUsername: () => username,

setUsername: (newName) => { username = newName; }

};

})();

console.log(userModule.getUsername()); // Shaurya

userModule.setUsername("Alex");

console.log(userModule.getUsername()); // Alex

In the above code, the user can directly call the methods defined in the function to perform operations without revealing the internal logic of the code.