CHAPTER 4

- Functions in JavaScript

A function is a block of reusable code designed to perform a specific task. It allows us to write modular, maintainable, and reusable code.

Function Declaration vs Function Expression ->

Function Declaration :
A function declaration is defined using the function keyword and can be hoisted, meaning it can be called before its definition in the code.

Example

```
function greet() {
console.log("Hello, world!");
}
greet();
```

- Function Expression :

A function expression assigns a function to a variable. Unlike function declarations, function expressions are not hoisted.

Example

```
const greet = function() {
console.log("Hello, world!");
};
greet();
```

- Arrow Functions In JavaScript (ES6)

Arrow functions provide a shorter syntax and have a lexical this binding.

Example

```
// Regular Function
function add(a, b) {
return a + b;
}

// Arrow Function
const add = (a, b) => a + b;
```

If a function has one parameter, parentheses () are optional.
If a function has one expression, {} and return are optional.
Arrow functions don't have their own this; they inherit it from their surrounding scope.

Lexical this Example

```
const obj = {
value: 10 ,
regularFunction: function() {
console.log(this.value) ; // ✅ 10
},
arrowFunction: () => {
console.log(this.value) ; // ❌ Undefined (inherits `this` from global scope)
}
};
obj.regularFunction() ;
obj.arrowFunction() ;
```

- Higher-Order Functions (Passing Functions as Arguments)

A higher-order function is a function that either:

Takes another function as an argument
Returns a function

Example using map() (which takes a function as an argument):

```
const numbers = [1, 2, 3, 4];
const squared = numbers.map(num => num * num);
console.log(squared); // [1, 4, 9, 16]
```

Example returning a function :

```
function multiplyBy(factor) {
```

```
return function(number) {
return number * factor ;
};
}

const double = multiplyBy(2) ;
console.log(double(5)) ; // 10
```

- Lexical Scope

Lexical Scope (or Static Scope) means that a function inherits the variables from its parent scope where it was defined, not where it was called.

Example of Lexical Scope:

```
function outerFunction() {
let outerVariable = "I am from outer!";

function innerFunction() {
console.log(outerVariable); // ✅ Can access outerVariable
}

innerFunction();
}

outerFunction();
```

💡 Key Point: innerFunction() can access outerVariable because it follows lexical scope, meaning it looks at where it was defined, not where it was called.

🔹 Closures

A closure occurs when an inner function remembers variables from its outer function, even after the outer function has finished execution.

Example of a Closure:

```
function counter() {
let count = 0; // Private variable

return function () {
count++;
console.log(`Count: ${count}`);
};
}

const increment = counter();
increment(); // Count: 1
increment(); // Count: 2
increment(); // Count: 3
```

🛠 Why Are Closures Useful?

Data Encapsulation (Private Variables)

The variable count is private inside counter().

Function Factories (Returning different behavior)

```
function multiplier(factor) {
return function (num) {
return num * factor;
};
}

const double = multiplier(2);
const triple = multiplier(3);

console.log(double(5)); // 10
console.log(triple(5)); // 15
```

3.Maintaining State in Asynchronous Code

```
function delayMessage(msg, delay) {
setTimeout(function () {
console.log(msg);
}, delay);
}

delayMessage("Hello after 2 seconds", 2000);
```

- Callbacks (Sync vs Async Execution)

A callback is a function passed as an argument to another function and executed later.

- Synchronous Callbacks

Executed immediately inside a function.

- Example:

```
function processUserInput(callback) {
let name = "Alice";
callback(name);
}

function greet(name) {
console.log(`Hello, ${name}!`);
}

processUserInput(greet); // Hello, Alice!
```

💡 Key Point: greet function is passed as a callback and executed immediately.

- Asynchronous Callbacks

Executed later, useful for handling tasks like API calls, file reading, and timers.

- Example using setTimeout():

```
console.log("Start");

setTimeout(() => {
console.log("This runs after 2 seconds");
}, 2000);

console.log("End");
```

💡 Key Point: JavaScript doesn't wait for setTimeout(), it moves on to the next task.