# Lesson:

# Closure

PW SKILLS

# Topics Covered

- Intro to closure with example & a real-life use case
  - Callbacks
  - Partial Application and Currying
  - Abstraction & Encapsulation with Closure
- Lexical Scope
- Scope chaining

# Intro to closure with example & a real-life use case

A closure is a combination of function and references to variables in its outer scope, even after the outer function has returned. Closures are created when a function is defined inside another function, and the inner function has access to the variables and parameters of the outer function.

When a function is executed, a new execution context is created, which includes the function's local variables and parameters. When the function completes, the execution context is destroyed, and its local variables and parameters are no longer available. However, if a nested function inside the function references a variable or parameter of the outer function, the inner function creates a closure and keeps a reference to that variable or parameter. The closure allows the inner function to access the outer function's variables and parameters, even after the outer function has returned and its execution context has been destroyed.

### Example 1:

```
function name() {
  let name = "mangesh ";
  return function () {
    return name.concat("thakare");
  };
}
const fullNameFun = name();
console.log(fullNameFun()); // output - mangesh thakare
```

In this example, the closure allows the fullName function to access and use the name variable from its outer scope, even after the name function has finished executing. This ensures that the fullName function retains access to the value of the name when it is called later.

## Example 2:

Let's create a program that returns the incremented value every time we invoke it.

```
function parent() {
  let count = 0;
  return function child() {
    count = count + 1;
    console.log(count);
  };
}
const increment = parent();
increment(); //1
increment(); //2
increment(); //3
```

**Use Cases**

**Callbacks:** Closures are commonly used in implementing callbacks. Functions passed as arguments to other functions (like in event listeners or setTimeout) "remember" the environment in which they were created, along with the variables they had access to at that time.

```
function greet() {
  let name = "Alice";

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

  setTimeout(sayHello, 2000);
}

greet();
```

In this example, the **greet** function contains a variable name and a nested function **sayHello**. The **sayHello** function is defined within greet, creating a closure that captures the **name** variable.

When greet is called, it sets a setTimeout function to execute sayHello after a 2-second delay. Even after the greet function completes execution, the sayHello function retains access to the name variable through the closure. So, after the delay, the sayHello function will still have access to the name variable and will log "Hello, Alice!" to the console.

**Partial Application and Currying:** They enable the creation of functions that remember a set of parameters and return a function waiting for additional arguments, aiding in partial application and currying.

```
function createMultiplier(multiplier) {
  return function (number) {
    return number * multiplier;
  };
}

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

console.log(double(5)); // Output: 10 (2 * 5)
console.log(triple(6)); // Output: 18 (3 * 6)
```

In this example, the createMultiplier function takes a multiplier argument and returns an inner function. The inner function, created within the createMultiplier context, captures the multiplier value through closure.

After invoking createMultiplier(2), the double function is created, which retains the multiplier value of 2. When double(5) is called, it multiplies 5 by the captured multiplier of 2, resulting in 10.

Similarly, createMultiplier(3) generates the triple function with a multiplier of 3. Therefore, triple(6) multiplies 6 by the captured multiplier of 3, resulting in 18. This illustrates the concept of currying and partial application using closures in JavaScript.

**Data Privacy:** Closures help in creating private variables and functions. This allows you to control and restrict access to certain data, exposing only necessary information via a public interface. For example, in the context of the module pattern, you can use closures to create encapsulated modules, revealing only the necessary functions or variables. We also call this use case **Abstraction & Encapsulation with Closure"**. These are the two most basic concepts of object-oriented programming in JS (OOJS). We will study OOJS in detail in later modules.

**Example 1:** Emulating private variables

This example demonstrates an important use of closures to maintain a private reference to a variable in the outer scope.

```
function secretPassword() {
  const password = "xh38sk";
  return {
    guessPassword: function (guess) {
      return guess === password;
    }
  };
}

const passwordGame = secretPassword();
console.log(passwordGame.guessPassword("heyisthisit?"));
// output - false
console.log(passwordGame.guessPassword("xh38sk"));
// output - true
```

This is a very powerful technique — it gives the closure function "guessPassword" exclusive access to the "password" variable while making it impossible to access the "password" from the outside.

The variable password remains private and cannot be accessed from outside the **secretePassword** function, **ensuring data privacy and encapsulation.**

**Example 2: Emulating private methods**

```
function createPerson(name) {
  // Private method
  function displayName() {
    console.log(`My name is ${name}`);
  }

  // Public interface
  return {
    introduce: function() {
      displayName();
      console.log("Nice to meet you!");
    }
  };
}

const person = createPerson("Alice");
person.introduce(); // Output: My name is Alice, Nice to meet you!
```

In this example, the **createPerson** function takes a **name** parameter and creates a private method displayName within its scope. This **displayName** method is only accessible within the createPerson function due to closure.

The createPerson function returns an object with a public method **introduce**. This method uses the private displayName method through the closure. When introduce is called, it invokes displayName to display the name of the person, then prints "Nice to meet you!" to the console. The displayName method remains private and cannot be accessed from outside the createPerson function, **ensuring data privacy and encapsulation**.

**Lexical Scope**

Every time the JavaScript engine creates an execution context to execute the function or global code, it also creates a new lexical environment to store the variable defined in that function during the execution of that function.

A lexical Environment is a data structure that holds an identifier-variable mapping. (here identifier refers to the name of variables/functions, and the variable is the reference to the actual object [including function type object] or primitive value).

A lexical environment has two components:

1. **Environment record:** which stores all the variables and functions declared within the function, and a reference to the outer environment, which is the lexical environment in which the function was defined.
2. **Reference to the outer environment:** It allows a function to access variables and functions declared in the outer environment

**Why lexical environment is important for closures?**

Lexical environment determines the scope in which a variable is defined and accessible. Closures allow a function to access and use variables from its parent function, and this is possible because the closure "closes over" the variables in the parent function's lexical environment, creating a snapshot of the relevant variables at the time the closure was created. Without a lexical environment, closures would not be able to access variables from their parent functions, severely limiting their functionality.

A lexical environment conceptually looks like this:

```
lexicalEnvironment = {
  environmentRecord: {
    <identifier> : <value>,
    <identifier> : <value>
  }
 outer: <Reference to the parent lexical environment>
}
```

# How do closures keep the lexical environment variables after the parent function has been removed?

When a function is created in JavaScript, it creates a new execution context which consists of the function's code and a reference to the outer lexical environment where the function was defined. This outer lexical environment is called the function's "lexical scope".

When the inner function is created, it captures a reference to the outer lexical environment and stores it in a "closure". This closure is a special object that contains the inner function and a reference to the outer lexical environment where the function was defined.

When the inner function is called, it uses this closure to access the variables in the outer lexical environment. The closure allows the inner function to access the variables as they were at the time the closure was created, even if the variables have since been changed or removed.

**Scope Chaining**

Every closure in JavaScript has access to three scopes:

- **Local Scope (Own Scope):** This refers to the innermost scope where variables are declared within the function.
- **Enclosing Scope:** This is the scope outside of the inner function and can be a block, function, or module scope. It encapsulates the environment where the inner function is created.
- **Global Scope:** The outermost scope that encloses all functions and variables.

In JavaScript, when an inner function is defined within an outer function, the inner function not only has access to its immediate outer function's scope but also to the scopes of all the surrounding functions (enclosing scopes). This creates a chain of function scopes known as the closure scope chain.

**Example:**

```
// global scope
const e = "Alex";

function func1(a) {
  return function func2(b) {
    return function func3(c) {
      // outer functions scope
      return function func4(d) {
        // local scope
        return `Values: ${a}, ${b}, ${c}, ${d}, ${e}`;
      };
    };
  };
}

const result = func1("Hello")("there")("from")("nested functions");

console.log(result); // Output: Values: Hello, there, from, nested functions, 10
```

The code provided illustrates a sequence of nested functions func1, func2, func3, func4, each capable of accessing variables and parameters from their respective outer functions' scope. This scenario explains the principle that closures possess access to all enclosing function scopes.