CLOSURE:

Definition:

A closure is a special kind of function in JavaScript. It allows a function to remember and access variables from an outer function, even after the outer function has finished running.

Key Points About Closures:

1. **A function within a function**: Closures happen when there is an inner function inside an outer function.
2. **Remembering variables**: The inner function "remembers" the variables of the outer function, even after the outer function is done.
3. **Useful for private variables**: Closures can be used to create private data that only the inner function can access.

Example:

```
function outerFunction() {
   let outerVariable = "I am from the outer function!";
   return function innerFunction() {
      console.log(outerVariable);          // innerFunction can access outerVariable
   };
 }
const myClosure = outerFunction();       // outerFunction executes
myClosure();                             // Logs: "I am from the outer function!"
```

Practical Uses of Closures:

1. **Data Hiding (Encapsulation):** Closures can create "private" variables that cannot be accessed from outside.

```

```

```
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
```

2. **Event Handlers**: Closures are often used in event listeners to "remember" some data.

```
function createClickHandler(message) {
    return function () {
        console.log(message);           // remembers the message
    };
 }
 const button = document.querySelector("button");
button.addEventListener("click", createClickHandler("Button clicked!"));
```

3. **Factory Functions**: Closures can generate functions with pre-configured behavior.

```
function multiplyBy(multiplier) {
    return function (value) {
```

```
      return value * multiplier;
   };
 }
const double = multiplyBy(2);
console.log(double(5));                    // result -10
```

Benefits of Closures:

1. **Maintain state**: Useful for keeping track of data across function calls.
2. **Data protection**: Variables inside closures are private to the function.
3. **Modularity**: Helps create reusable and self-contained pieces of code.

THIS KEYWORD:

Definition:

The this keyword in JavaScript is a reference to the object that is currently executing the code. Its value depends on how and where the function it resides in is called. It can refer to.

How this works in different scenarios:

1. **Global Context:**
   - In the global scope (outside of any function), this refers to the global object.
   - In browsers, this is the window object.

```
console.log(this);        // In a browser, it logs the global 'window' object.
```

2. **Object Context:**

- When a function is called as a method of an object, `this` refers to the object the method is called on.

```
const user = {
  name: "Alice",
  greet() {
    console.log(this.name);          // Refers to 'user' object
  }
};
user.greet();                        // Logs "Alice"
```

3. **Constructor Functions:**
   - In a constructor function, this refers to the instance being created.

```
function Person(name) {
  this.name = name;
}
const john = new Person("John");
console.log(john.name);          // Logs "John"
```

4. **Arrow Functions:**
   - Arrow functions do not have their own `this`. Instead, they inherit `this` from the surrounding (parent) scope.

```
const person = {
  name: "Bob",
  greet: () => {
    console.log(this.name);          // Undefined, as `this` is
inherited from global scope
  }
};
person.greet();                      // Logs undefined
```

5. **Event Listeners:**
    - In DOM event listeners, `this` refers to the element the event is bound to.

```
document.getElementById('myButton').addEventListener('click', function()
{
   console.log(this);          // The button element
});
```

6. **In Classes:**
    - Similar to constructor functions, `this` refers to the instance of the class.

```
class Animal {
   constructor(type) {
      this.type = type;
   }
   speak() {
      console.log(`This is a ${this.type}`);
   }
}
const dog = new Animal("dog");
dog.speak();                          // Logs "This is a dog"
```

7. **In Call, Apply, Bind Functions:**
    - **Bind Method:** Creates a new function with this bound to a specific value.
    - **Call Method**: Calls a function with this set to a specific value, and arguments provided individually.
    - **Apply Method:** Similar to call, but arguments are provided as an array.

```
let obj1 = {
   num1: 100,
   num2: 200
};

let func1 = function (param1, param2, param3) {
   console.log(this.num1 + this.num2 + param1 + param2 + param3);
};

func1.call(obj1, 300, 400, 500);          // Call function with obj1 as context

// Apply function
let arr = [300, 400, 500];
func1.apply(obj1, arr);         //Apply function with obj1 as context and arguments

// Bind function
let new_func = func1.bind(obj1);
new_func(300, 400, 500);                  // Call the bound function
```

EVENT BUBBLING AND EVENT CAPTURING:

Event Bubbling:

- Event bubbling describes the process where an event starts from the target element (where the event occurred) and moves upward to its parent elements in the DOM tree.
- It "bubbles" up from the innermost element to the outermost elements.

Example of Event Bubbling:

```
document.getElementById("child").addEventListener("click", () => {
  console.log("Child element clicked!");
});

document.getElementById("parent").addEventListener("click", () => {
  console.log("Parent element clicked!");
```

```
});
```

Output when you click the child element:

```
Child element clicked!
Parent element clicked!
```

In bubbling, both the child and parent event handlers execute, starting from the child (target).

Event Capturing (or Event Tricking):

- In capturing, the event moves from the outermost element to the innermost element.
- This is the opposite of event bubbling.
- To capture events during the capturing phase, pass { capture: true } as an option when adding an event listener.

Example of Event Capturing:

```
document.getElementById("parent").addEventListener("click", () => {
  console.log("Parent element clicked!");
}, { capture: true });

document.getElementById("child").addEventListener("click", () => {
  console.log("Child element clicked!");
}, { capture: true });
```

Output when you click the child element:

```
Parent element clicked!
Child element clicked!
```

Difference Between Bubbling and Capturing:

Feature	Event Bubbling	Event Capturing
Direction	From child (target) to parent elements.	From parent to child (target).
Default Behavior	Bubbling is the default behavior in JS.	Capturing requires { capture: true }.
Use Case	Useful for handling events on the target first.	Useful when you want to intercept events early.

Stopping Event Propagation:

You can stop the event from propagating further using event.stopPropagation().

Example:

```
document.getElementById("child").addEventListener("click", (event) => {
  console.log("Child clicked!");
  event.stopPropagation();     // Stops the event from bubbling or capturing further
});

document.getElementById("parent").addEventListener("click", () => {
  console.log("Parent clicked!");
```

```
});
```

Output when you click the child element:

```
Child clicked!
```

CALL FUNCTION ,APPLY FUNCTION AND BIND FUNCTION

Method	When to Use
call()	When you need to invoke a function immediately with a specific this value.
apply()	When you need to invoke a function immediately with arguments in array format.
bind()	When you need a function to be called later with a specific this value.

call():

Use when you want to invoke a function immediately and explicitly set the value of this.

Example:

```
const person1 = { name: "Alice" };
```

```
const person2 = { name: "Bob" };


function introduce(greeting) {
  console.log(`${greeting}, I'm ${this.name}`);
}


// Use call() to run introduce() with person2
introduce.call(person2, "Hello");
// Output: Hello, I'm Bob
```

apply():

Use when you want to invoke a function immediately and pass arguments as an array.

Example:

```
const numbers = [5, 12, 8, 21];


// Use apply() to find the maximum value in an array
const max = Math.max.apply(null, numbers);
console.log(max);                    // Output: 21
```

bind():

Use when you want to create a new function with a specific this value but not execute it immediately.

```
const car = {
  brand: "Toyota",
  getBrand() {
    console.log(this.brand);
  },
};

// Create a new function with bind()
const getCarBrand = car.getBrand.bind(car);

// Later use
setTimeout(getCarBrand, 1000);
// Output: Toyota
```