# Closure:

**A closure is a fundamental concept in JavaScript, where a function retains access to its lexical scope even after the function has finished executing. In simpler terms, a closure is created when an inner function has access to the variables and parameters of its outer function, even after the outer function has returned.**

**Here's a basic example to illustrate closures:**

```
function outerFunction() {
  const outerVariable = "I am from the outer function";

  function innerFunction() {
    console.log(outerVariable);
  }

  return innerFunction;
}

const closure = outerFunction();
closure(); // Output: I am from the outer function
```

**In this example:**

- **outerFunction defines a variable outerVariable and an inner function innerFunction.**
- **innerFunction has access to outerVariable even though it is defined within outerFunction.**

- **When outerFunction is called, it returns innerFunction, creating a closure.**
- **Later, when closure is called, it still has access to outerVariable, demonstrating the closure.**

**Here are some key points to understand about closures:**

**Access to Outer Scope:**

**Inner functions have access to variables and parameters of their outer functions, even after the outer function has finished executing.**

**Preservation of Variables:**

**Closures "remember" the environment in which they were created, including the variables in their lexical scope.**

**Data Privacy:**

**Closures can be used to create private variables and encapsulation. Since variables within an outer function are not accessible from outside, they can act as private variables.**

**Garbage Collection:**

**Closures may prevent garbage collection of the outer function's variables if the closure is still in use, which can lead to memory leaks if not managed carefully.**

**Here's another example illustrating private variables using closures:**

```
function createCounter() {
  let count = 0;
  return {
    increment: function () {
      count++;
      console.log("Count:", count);
    },
    decrement: function () {
      count--;
      console.log("Count:", count);
    },
    getCount: function () {
      return count;
    },
  };
}
const counter = createCounter();
counter.increment(); // Output: Count: 1
counter.increment(); // Output: Count: 2
counter.decrement(); // Output: Count: 1
console.log(counter.getCount()); // Output: 1
```

**In this example, count is a private variable that cannot be accessed directly from outside createCounter(), but the inner functions returned by createCounter() (e.g., increment, decrement, getCount) have access to it due to closures.**

Closures are powerful and commonly used in JavaScript for various purposes, such as encapsulation, creating private variables, and maintaining state in functional programming. Understanding closures is crucial for writing clean, maintainable JavaScript code.

### Example 1: Creating Memoization Function

Memoization is an optimization technique used to cache the results of expensive function calls and return the cached result when the same inputs occur again.

```
function memoize(func) {
  const cache = {};

  return function (...args) {
    const key = JSON.stringify(args);
    if (!(key in cache)) {
      cache[key] = func(...args);
    }
    return cache[key];
  };
}

const fibonacci = memoize(function (n) {
  if (n <= 1) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
});

console.log(fibonacci(10)); // Output: 55
```

**In this example, memoize() function creates a closure over the cache object. The returned function checks if the result for the given arguments already exists in the cache. If it does, it returns the cached result; otherwise, it calculates the result using the original function (fibonacci() in this case) and stores it in the cache.**

### Example 2: Implementing Private Variables and Methods

**Closures can be used to create objects with private variables and methods, providing encapsulation.**

```
function createPerson(name, age) {
  let _name = name;
  let _age = age;

  return {
    getName: function () {
      return _name;
    },
    getAge: function () {
      return _age;
    },
    celebrateBirthday: function () {
      _age++;
    },
  };
}

const person = createPerson("Alice", 30);
console.log(person.getName()); // Output: Alice
console.log(person.getAge()); // Output: 30
person.celebrateBirthday();
```

```
console.log(person.getAge()); // Output: 31
```

**In this example, _name and _age are private variables encapsulated within the closure. The returned object provides methods to access these variables but does not expose them directly.**

### Example 3: Callbacks in Asynchronous Operations

**Closures are commonly used in asynchronous operations, such as event handlers or AJAX requests, to maintain access to the surrounding context when the callback is executed.**

```
function fetchData(url, callback) {
  // Simulating AJAX request
  setTimeout(() => {
    const data = { name: "Example Data" };
    callback(data);
  }, 1000);
}

const container = document.getElementById("container");

fetchData("https://api.example.com/data", function
(data) {
  // Closure: has access to container even when executed
later
  container.innerHTML = `<p>${data.name}</p>`;
});
```

**In this example, the callback function passed to fetchData() has access to the container element, even though it is executed**

**asynchronously after the AJAX request completes, thanks to the closure.**

**Closures are versatile and can be used in various scenarios to maintain state, encapsulate functionality, and provide access to outer scopes in JavaScript functions.**