In JavaScript, **scope** refers to the accessibility or visibility of variables, functions, and objects in certain parts of the code during the execution. The scope determines where in the code a variable or function is available for use.

There are two main types of scope in JavaScript:

1. Global Scope:

- Variables declared outside of any function or block are in the global scope.
- These variables are accessible from anywhere in the code, including inside functions.
- Example:

```
  let globalVar = "I am global"; // Global scope

  function display() {
    console.log(globalVar); // Accessing global variable inside function
  }

  display(); // Output: "I am global"
```

2. Local Scope:

- Variables declared inside a function are in the **local scope** of that function.
- These variables are only accessible within that function.
- Example:

```
function myFunction() {
    let localVar = "I am local"; // Local scope
    console.log(localVar); // Accessible inside the function
  }

  myFunction(); // Output: "I am local"
  console.log(localVar); // Error: localVar is not defined
```

Types of Local Scope:

- **Function Scope:** Variables declared with var are function-scoped, meaning they are accessible only within the function in which they are declared.
- **Block Scope:** Variables declared with let and const are block-scoped, meaning they are only accessible within the block (e.g., inside loops, conditionals).

Lexical Scope:

- Lexical scope refers to the fact that a function's scope is determined by where it is defined, not where it is called.
- Example:

```
    function outer() {
          let outerVar = "I'm outer"; // outer scope

          function inner() {
            console.log(outerVar); // inner function can access outer
function's variable
          }

          inner();
        }

        outer(); // Output: "I'm outer"
```

3. Block Scope:

- Introduced in ES6, block-scoped variables are declared with let and const inside {} blocks, such as if statements or for loops.
- Example:

```
    if (true) {
          let blockVar = "I'm block-scoped";
          console.log(blockVar); // Accessible within the block
        }
        console.log(blockVar); // Error: blockVar is not defined outside the
block
```

4. Hoisting:

- Hoisting is a JavaScript mechanism where variables and functions are moved to the top of their scope during the compile phase.
- However, only var declarations are hoisted to the top. let and const are hoisted but are not initialized until the code execution reaches the declaration.
- Example:

```
console.log(a); // undefined
var a = 5;

console.log(b); // Error: Cannot access 'b' before initialization
let b = 10;
```

5. Closure scope

In JavaScript, **closure** is a function that "remembers" its lexical scope (the environment in which it was created) even when that function is executed outside of its original scope. This means that a closure allows a function to access variables from its outer (enclosing) function even after the outer function has finished execution.

How closure works:

When a function is defined inside another function, it forms a closure. The inner function has access to variables from its own scope, the outer function's scope, and the global scope. Even if the outer function has returned, the inner function can still access variables from the outer function because it "remembers" the environment in which it was created.

Example of closure:

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

    // The inner function forms a closure and can access outerVariable
    return function innerFunction() {
      console.log(outerVariable);
    };
  }

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

Key points about closure:

1. **Encapsulation**: Closures allow data to be hidden and only accessed through specific functions. This is a form of data encapsulation.
2. **Preservation of state**: Closures can help "preserve" the state of variables in an environment, even after the outer function has executed.
3. **Memory consumption**: Because closures preserve references to variables, they can sometimes lead to higher memory consumption if not managed carefully.

Example with parameters:

```
function multiplier(factor) {
    return function(number) {
```

```
      return number * factor; // factor is remembered even after multiplier
has finished execution
    };
  }

  const multiplyByTwo = multiplier(2);
  console.log(multiplyByTwo(5)); // Output: 10
```

In this example, the inner function remembers the factor argument of the outer function and continues to use it even after the outer function has returned. This is a closure in action!

Examples of closure scope in JavaScript:

1. Creating Private Variables:

```
function createCounter() {
    let count = 0;

    return function() {
      count++;
      return count;
    };
  }

  const counter = createCounter();

  console.log(counter()); // Output: 1
  console.log(counter()); // Output: 2
  console.log(counter()); // Output: 3
```

- In this example, the count variable is declared within the createCounter function.
- The inner function (returned by createCounter) has access to count even after createCounter has finished executing.
- This creates a closure, making count effectively private and accessible only through the returned function.

2. Creating Caching Mechanisms:

```
function memoize(fn) {
    const cache = {};
    return function(arg) {
      if (arg in cache) {
```

```
      return cache[arg];
    }
    const result = fn(arg);
    cache[arg] = result;
    return result;
  };
}

function factorial(n) {
  if (n === 0) {
    return 1;
  }
  return n * factorial(n - 1);
}

const memoizedFactorial = memoize(factorial);

console.log(memoizedFactorial(5)); // Calculate and store
console.log(memoizedFactorial(5)); // Retrieve from cache
```

- The `memoize` function creates a closure around the `cache` object.
- The returned function can access and modify `cache` to store and retrieve previously calculated results, improving performance for expensive function calls.

3. Creating Event Handlers:

```
const button = document.getElementById('myButton');

button.addEventListener('click', function() {
  console.log('Button clicked!');
});
```

- The event handler function (the anonymous function passed to `addEventListener`) forms a closure.
- It has access to the `button` variable even after the event listener is attached.

4. Currying:

```
function add(x) {
    return function(y) {
      return x + y;
    };
  }

  const add5 = add(5);
```

```
console.log(add5(3)); // Output: 8
```

- The `add` function returns another function that "remembers" the value of `x`.
- This creates a closure, allowing you to partially apply arguments to a function.

5. Creating Private Methods:

```
const myObject = {
    data: 10,
    publicMethod: function() {
      this.privateMethod(); // Accessing privateMethod
    },
    privateMethod: function() {
      console.log("This is a private method.");
    }
  };
```

- While not a true closure in the strictest sense, this pattern simulates private methods within an object.
- The `privateMethod` is only accessible within the object's methods, making it effectively private.

These examples demonstrate the power and versatility of closures in JavaScript. They enable you to create more modular, reusable, and efficient code.