In JavaScript, closure is a powerful and fundamental concept that allows a function to remember and access its lexical scope even when it's executed outside of its original scope. This means a function can access variables and parameters of its outer function(s) even after the outer function has finished execution. Closures are created every time a function is created, at function creation time.

To understand closures, it's important to grasp two key concepts in JavaScript: lexical scoping and first-class functions.

1. **Lexical Scoping**: Lexical scoping means that a function's variables are accessible within the function itself and within any inner functions. The scope of variables is determined by their physical location in the written code (hence "lexical").
2. **First-Class Functions**: JavaScript treats functions as first-class citizens, meaning they can be passed around like any other value (e.g., assigned to variables, passed as arguments to other functions, returned from functions).

## Example of Closure:

function outerFunction(outerVariable) {
return function innerFunction(innerVariable) {
console.log('Outer Variable: ' + outerVariable);
console.log('Inner Variable: ' + innerVariable);
}
}
const newFunction = outerFunction('outside');
newFunction('inside');

In this example, outerFunction creates a closure including the innerFunction and the outerVariable. Even after outerFunction has finished execution, innerFunction retains access to outerVariable. When newFunction, which is the returned innerFunction, is called with the argument 'inside', it still has access to outerVariable ('outside'), demonstrating the closure.

## Why Are Closures Useful?

* **Encapsulation**: They allow for the encapsulation of behavior and state. By using closures, you can create private variables that can only be accessed and modified by a defined set of functions.
* **Maintaining State in Async Operations**: Closures are particularly useful in asynchronous programming. For example, when setting up callbacks that need to access the state available at the time the callback was created.
* **Currying and Function Factories**: Closures enable currying and the creation of function factories. This means you can create functions that generate other functions with specific behaviors, which can be very powerful in functional programming.

In summary, closures are a core concept in JavaScript, enabling functions to capture and retain their lexical scope, thereby providing powerful tools for encapsulation, state management, and functional programming patterns.