Closure in JavaScript

A closure in JavaScript is a function that remembers and can access its lexical environment (the scope in which it was defined), even after its outer function has finished executing.

This means an inner function, forming the closure, maintains access to the variables and parameters of its containing function.

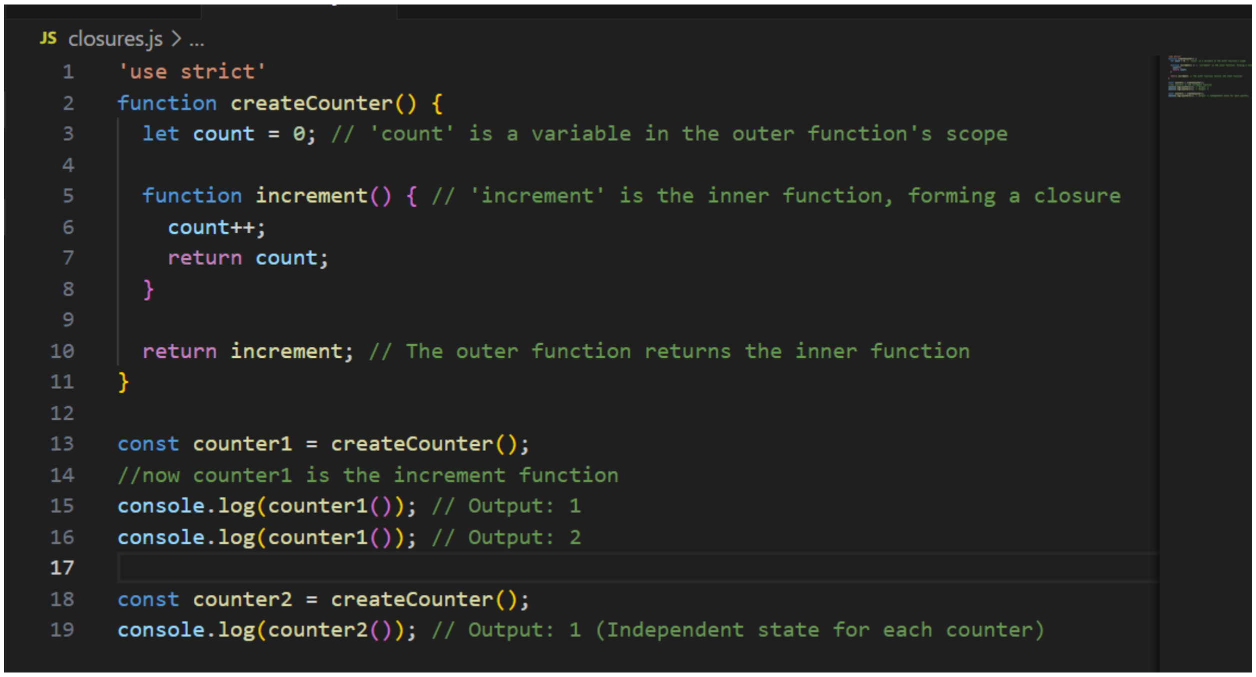
Key characteristics of closures:

- **Lexical Scoping:**

  Closures inherently follow lexical scoping, meaning a function's scope is determined where it is defined, not where it is called.
- **Access to Outer Scope:**

  The inner function within a closure can access variables from its own scope, the outer function's scope, and the global scope.
- **Persistence of State:**
  Closures allow for the creation of "private" variables and the maintenance of state. The inner function retains a reference to the outer function's variables, preserving their values even after the outer function has completed its execution.

Example:


closures.js > ...

```javascript
'use strict'
function createCounter() {
  let count = 0; // 'count' is a variable in the outer function's scope

  function increment() { // 'increment' is the inner function, forming a closure
    count++;
    return count;
  }

  return increment; // The outer function returns the inner function
}

const counter1 = createCounter();
//now counter1 is the increment function
console.log(counter1()); // Output: 1
console.log(counter1()); // Output: 2

const counter2 = createCounter();
console.log(counter2()); // Output: 1 (Independent state for each counter)
```

In this example, increment is a closure. When createCounter() is called, it returns the increment function. Even though createCounter() has finished executing, the increment function still "remembers" and can access the count variable from its lexical environment, allowing it to maintain its state independently for each counter instance.