Shadowing:

===========

In programming, shadowing occurs when a variable declared in a certain scope (e.g. a local variable) has the same name as a variable in an outer scope (e.g. a global variable). When this happens, the outer variable is said to be shadowed by the inner variable.

        function test(){

            const a=10

            if(true){

                const a=20

         console.log(a) // 20, outer variable a shadowed

            }

        }

        test()

In JavaScript, variables can be shadowed in both the **global** and **function** scope. Global variables can be shadowed by function-scoped variables, and function-scoped variables can be shadowed by block-scoped variables declared with the let or const keyword.

Closures, Lexical environment:

A closure is a feature of JavaScript that allows inner functions to access the outer scope of a function. Closure helps in binding a function to its outer boundary and is created automatically whenever a function is created. A block is also treated as a scope since ES6. Since JavaScript is event-driven so closures are useful as it helps to maintain the state between events.

**Lexical Scoping:** A function scope’s ability to access variables from the parent scope is known as lexical scope. We refer to the parent function’s lexical binding of the child function as “lexically binding.”

Or

The scope is where a variable is available in your code. Each block of code creates scope and a lexical environment. The outermost context where a variable is the lexical environment of that variable irrespective of order or sequence.

JavaScript cares about the lexical environment when you ask for a variable while running a line of code inside any particular execution context if it can’t find that variable in its block it will go at the outer reference or block and look for variables there. And that outer reference is where the function sits lexically is its outer environment.

Let’s see and understand closure through an example.

function foo() {

let b = 1;

function inner() {

return b;

}

return inner;

}

let get\_func\_inner = foo();

console.log(get\_func\_inner());

console.log(get\_func\_inner());

console.log(get\_func\_inner());

**Output:**We can access the variable **b**which is defined in**the function foo()** through **function inner()** as the later preserves the scope chain of the enclosing function at the time of execution of the enclosing function i.e. the inner function knows the value of **b** through its **scope chain**.   
**This is closure in action that is inner function can have access to the outer function variables as well as all the global variables.**

A closure looks like this:

1. An outer function
2. An inner function
3. A return statement that references the inner function

Like this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | function outer(){      function inner(){    }      return inner  } |