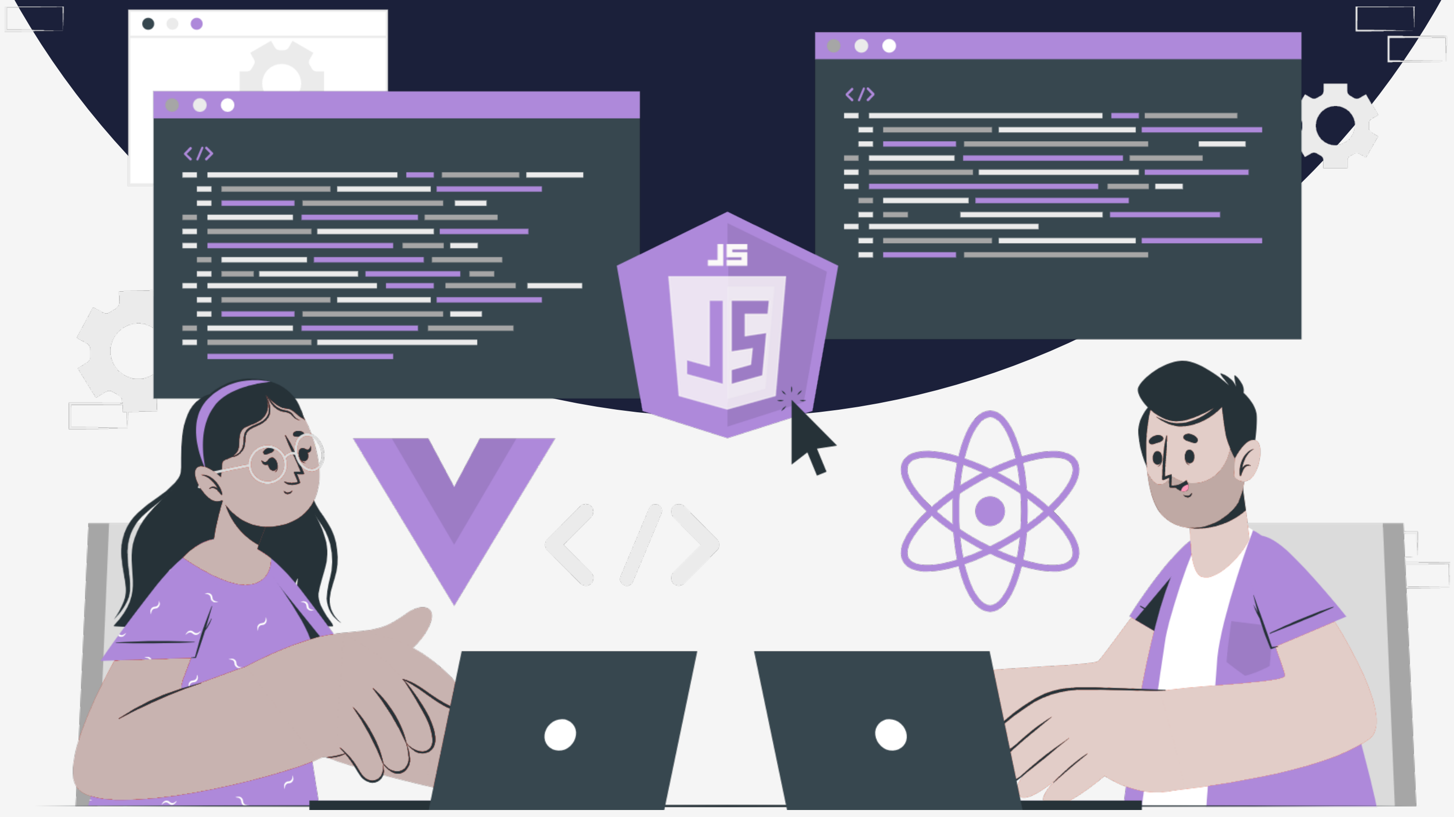
Lesson:

First Program in Closure and Understanding Closures



List of Content:

- Example to demonstrate closure
- How closures keep the lexical environment variables after the parent function has been removed
- Example 1
- Examples 2

Every function has an execution context, which comprises of an environment that gives meaning to the variables within that function and a reference to its parent's environment. A reference to the parent's environment makes all variables in the parent scope available for all inner functions, regardless of whether the inner function(s) are invoked outside or inside the scope in which they were created.

So, it appears as if the function "remembers" this environment (or scope) because the function literally has a reference to the environment (and the variables defined in that environment)!

```
function outer() {
  const a = 10;

  function inner() {
    const b = 20;
    console.log(a + b);
  }

  return inner;
}

const closureFn = outer();
closureFn(); // outputs 30
```

In this modified example, **"outer()"** defines a variable **"a"** and a function **"inner()"**, just like before. However, instead of calling **"inner()"** directly, **"outer()"** returns **"inner()"**. The variable **"closureFn"** is then assigned to the returned **"inner()"** function.

When **"closureFn()"** is called, it logs the sum of **"a"** and **"b"** to the console. However, **"b"** is not defined in the scope of **"closureFn()"**. How is **"closureFn()"** able to access **"a"** and **"b"**, then?

The answer lies in the concept of closure. When **"outer()"** is called, it creates a new execution context and defines a variable **"a"** and a function **"inner()"**. When **"inner()"** is defined, it creates a closure over the variables and functions in the outer scope, including **"a"**. This means that even after **"outer()"** returns, **"inner()"** still has access to **"a"**. When **"outer()"** returns **"inner()"**, **"closureFn"** is assigned to the returned **"inner()"** function, complete with its closure over **"a"** and **"inner()"**.

How closures keep the lexical environment variables after the parent function has been removed

When a function is created in JavaScript, it creates a new execution context which consists of the function's code and a reference to the outer lexical environment where the function was defined. This outer lexical environment is called the function's "lexical scope".

when the inner function is created, it captures a reference to the outer lexical environment and stores it in a "closure". This closure is a special object that contains the inner function and a reference to the outer lexical environment where the function was defined.

When the inner function is called, it uses this closure to access the variables in the outer lexical environment. The closure allows the inner function to access the variables as they were at the time the closure was created, even if the variables have since been changed or removed.

Example 1:
Let's create a program that returns the incremented value every time we invoke it.

```
function parent() {

    let count = 0;

    function child() {
        count = count + 1
        console.log(count);
    }

    return child;
}


const increment = parent()
increment() //1
increment() //2
increment() //3
```

Example 2:
One final example to demonstrate an important use of closures: to maintain a private reference to a variable in the outer scope.

```
function secretPassword() {

  const password = 'xh38sk';

  return {
    guessPassword: function(guess) {

      if (guess === password) {
        return true;

      } else {

        return false;

      }
    }
  }
}

const passwordGame = secretPassword();
passwordGame.guessPassword('heyisthisit?'); // false
passwordGame.guessPassword('xh38sk'); // true
```

This is a very powerful technique — it gives the closure function **“guessPassword”** exclusive access to the **“password”** variable while making it impossible to access the **“password”** from the outside.