Lesson:

Explaining what is closure?
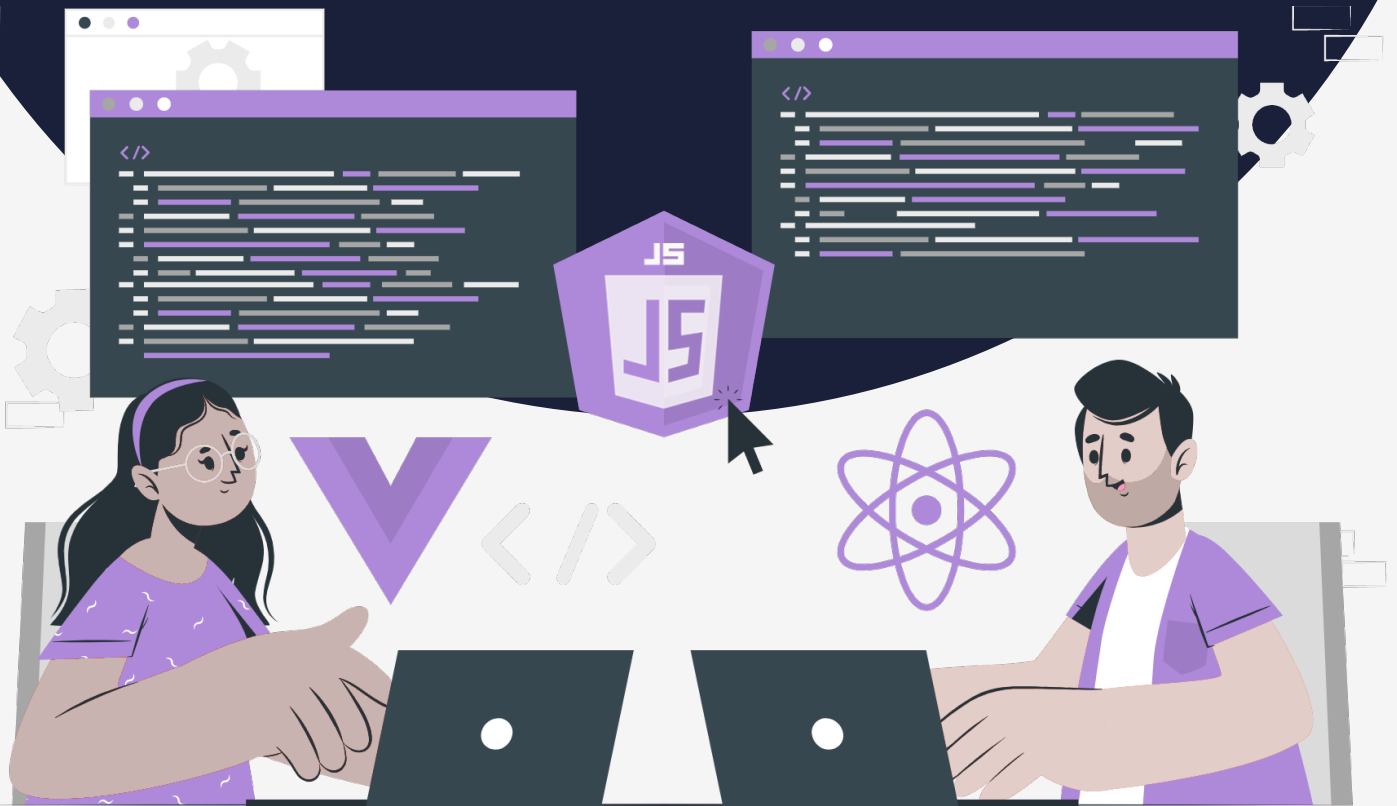
List of Content:

Lexical Environment
Every time the JavaScript engine creates an execution context to execute the function or global code, it also creates a new lexical environment to store the variable defined in that function during the execution of that function.

A lexical Environment is a data structure that holds an identifier-variable mapping. (here identifier refers to the name of variables/functions, and the variable is the reference to the actual object [including function type object] or primitive value).

A lexical environment has two components:

- **Environment record:** which stores all the variables and functions declared within the function, and a reference to the outer environment, which is the lexical environment in which the function was defined.
- **Reference to the outer environment:** It allows a function to access variables and functions declared in the outer environment

Why lexical environment is important for closures?

Lexical environment determines the scope in which a variable is defined and accessible. Closures allow a function to access and use variables from its parent function, and this is possible because the closure "closes over" the variables in the parent function's lexical environment, creating a snapshot of the relevant variables at the time the closure was created. Without a lexical environment, closures would not be able to access variables from their parent functions, severely limiting their functionality.

A lexical environment conceptually looks like this:

```
lexicalEnvironment = {
  environmentRecord: {
    <identifier> : <value>,
    <identifier> : <value>
  }
  outer: < Reference to the parent lexical environment>
}
```

Closures
A closure is a function that has access to variables in its outer scope, even after the outer function has returned. Closures are created when a function is defined inside another function, and the inner function has access to the variables and parameters of the outer function.

When a function is executed, a new execution context is created, which includes the function's local variables and parameters. When the function completes, the execution context is destroyed, and its local variables and parameters are no longer available. However, if a nested function inside the function references a variable or parameter of the outer function, the inner function creates a closure and keeps a reference to that variable or parameter. The closure allows the inner function to access the outer function's variables and parameters, even after the outer function has returned and its execution context has been destroyed.

```
function outerFunction() {
  const outerVar = 'I am in the outer function';

  function innerFunction() {
    console.log(outerVar); // logs "I am in the outer function"
  }

  innerFunction();
}


outerFunction()
```

In this example, the **"innerFunction"** has a reference to the **"outerFunction's"** lexical environment, which includes the **"outerVar"** variable. When **"innerFunction"** is invoked, it can access and log the value of **"outerVar"**.

Summary

In JavaScript, closures are created whenever a function is declared inside another function, and they are used extensively in modern web development. Understanding closures and how they interact with lexical environment and scope is essential for writing efficient and effective code.