Lecture-9

Closures

Async Prog.

Coding Blocks - Kartik Mathur

Class Agenda

01	Closures		
02	Lexical Scope, TDZ		
03	setTimeout & setInterval	04	Callbacks
05	Promises		
06	-	07	-

Closures

1

A **closure** is a function that **remembers its outer scope variables even after the outer function has executed**.

- This is useful in JavaScript for **data encapsulation, maintaining state, and functional programming**.

Closures

- Write a function counter that returns another function. Each time the inner function is called, it should increment and return a count.

```
function counter() {
    let count = 0;
    return function() {
        count++;
        return count;
    };
}

const count = counter();
console.log(count()); // 1
console.log(count()); // 2
console.log(count()); // 3
```

Closures-Basic Example

Function Currying!!

Function - Currying

Data Encapsulation!

- Create a function secureBankAccount that initializes a private balance and only allows deposit and withdrawal.

Function - Currying

Data Encapsulation!

- Create a function secureBankAccount that initializes a private balance and only allows deposit and withdrawal.

```
function secureBankAccount(initialBalance) {
    let balance = initialBalance;
    return {
        deposit: function(amount) {
            balance += amount;
            return balance;
        },
        withdraw: function(amount) {
            balance -= amount;
            return balance;
        },
        getBalance: function() {
            return balance;
        }
    };
}

const account = secureBankAccount(100);
console.log(account.deposit(50)); // 150
console.log(account.withdraw(30)); // 120
console.log(account.getBalance()); // 120
console.log(account.balance); // undefined (Encapsulation)
```

Closures

Data Encapsulation!

- Create a function secureBankAccount that initializes a private balance and only allows deposit and withdrawal.

```
function secureBankAccount(initialBalance) {
    let balance = initialBalance;
    return {
        deposit: function(amount) {
            balance += amount;
            return balance;
        },
        withdraw: function(amount) {
            balance -= amount;
            return balance;
        },
        getBalance: function() {
            return balance;
        }
    };
}

const account = secureBankAccount(100);
console.log(account.deposit(50)); // 150
console.log(account.withdraw(30)); // 120
console.log(account.getBalance()); // 120
console.log(account.balance); // undefined (Encapsulation)
```

Closures

- If a variable is declared inside a code block `{ . . . }`, it's only visible inside that block.

```
{
    // show message
    let message = "Hello";
    alert(message);
}

{
    // show another message
    let message = "Coding Blocks";
    alert(message);
}
```

Code block

The script as a whole have an internal (hidden) associated object known as the *Lexical Environment*.

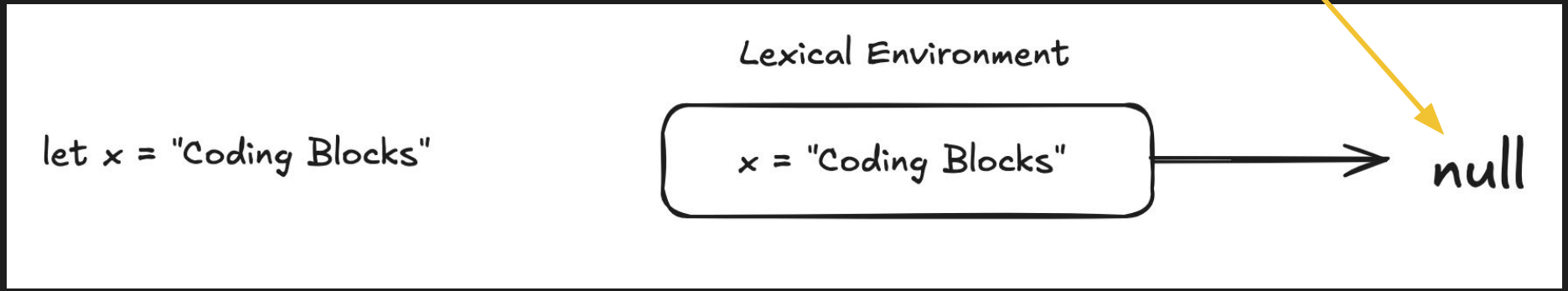
The Lexical Environment object consists of two parts:

1. **Environment Record**
   - An object that stores all local variables as its properties.
2. A reference to the ***outer lexical environment***, the environment associated with the outer code.

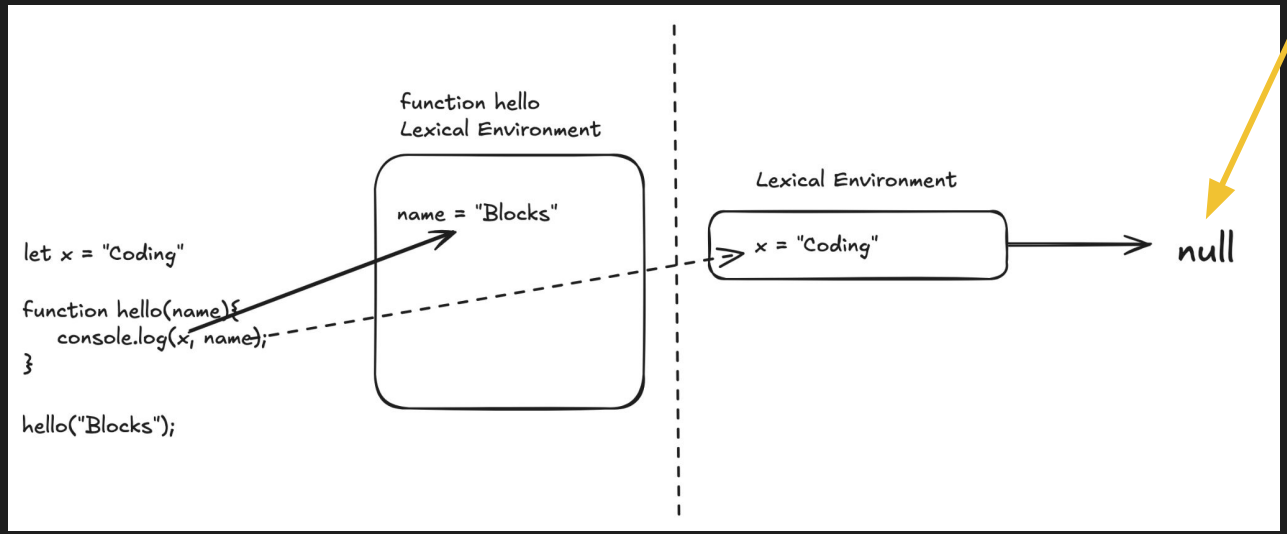
Lexical Environment

Lexical Environment - Global

1. Global Lexical Environment
    - Without function we just have this one only.



1. Global Lexical Environment
2. With functions



There is no outer lexical environment

Lexical Environment - With Functions

- All the functions have hidden property that is called as ENVIRONMENT
- It keeps the the reference of the lexical environment where it was created that is what we call “CLOSURES”

Closures

Implement a Memoization Function Using Closures

```
function memoize(fn) {
    let cache = {};
    return function (...args) {
        let key = args;
        if (!cache[key]) {
            cache[key] = fn(...args);
        }
        return cache[key];
    };
}

const factorial = memoize(function (n) {
    return n <= 1 ? 1 : n * factorial(n - 1);
});

console.log(factorial(5)); // 120
console.log(factorial(5)); // 120 (from cache)
```

Closures

Homework:

https://github.com/Kartik-Mathur/Assignment-WebDev-Batches/blob/main/Closure-Students-ManagementAssignment.md

https://github.com/Kartik-Mathur/WebDev-12Jan2025/blob/main/L8-Assignment-Closure/1_Assignment-Closures.pdf

Closures

TDZ: Temporal Dead Zone

The **Temporal Dead Zone (TDZ)** is the time between the **creation of the variable** in the Lexical Environment and its **declaration** in the code.

```
let x = 1;

function fun() {
    console.log(x); // ReferenceError: Cannot access 'x' before initialization
    let x = 2;
}
function fun(): void
fun();
```

This means they exist in the scope but remain **uninitialized** from the start of the block until their declaration is encountered.

- **Step 1: Entering the function `func()`**
  - A new **Lexical Environment** is created for the function execution.
  - Inside this environment, `let x` is recognized but remains **uninitialized** (TDZ starts).
- **Step 2: Executing `console.log(x);`**
  - The JavaScript engine looks for x in the function scope first.
  - Since `let x = 2;` exists in the function scope, JavaScript does not look in the outer scope (global x is ignored).
  - However, x is still in the **Temporal Dead Zone**, so accessing it before declaration throws a **ReferenceError**.
- **Step 3: `let x = 2;` is reached**
  - The variable x is now **initialized** and can be used normally after this point.

var gets hoisted and is **undefined**, so it works.

TDZ: Temporal Dead Zone Explained

NEXT CLASS:

1. setTimeout, setInterval
2. Callbacks
3. Promises

Asynchronous Programming