

Who am I?

Humera Tariq

PhD, MS, MCS (Computer Science), B.E (Electrical)

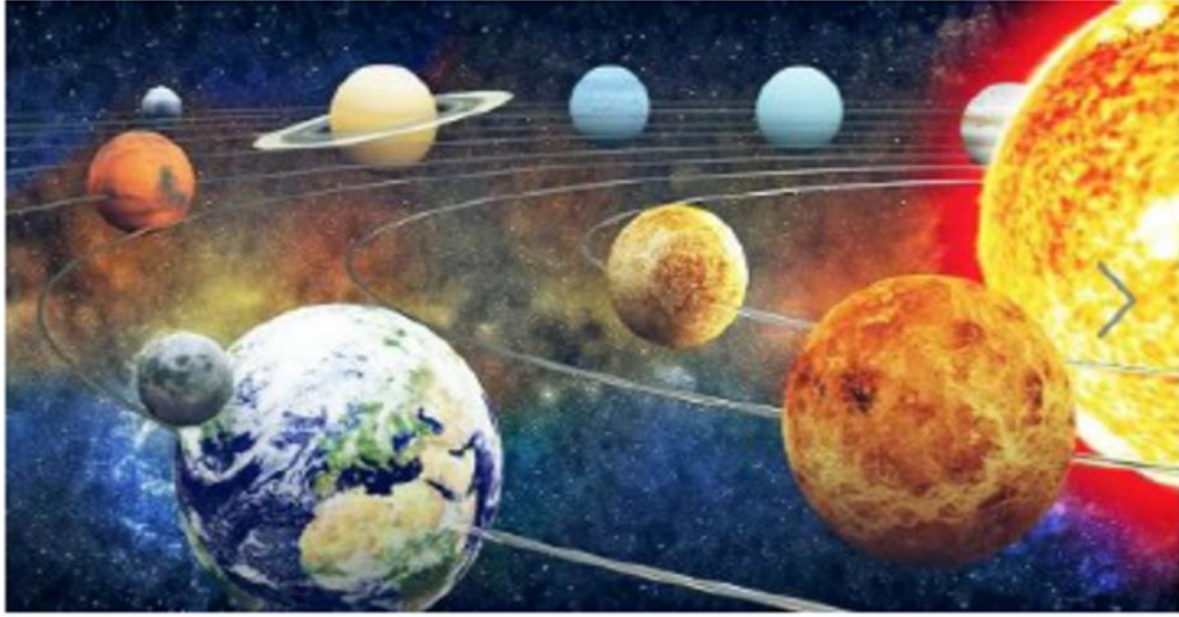
Postdoc (Medical Image Processing, Deep Neural Networks)

Email: humera@uok.edu.pk

Web: <https://humera.pk/>

Discord: <https://discord.gg/xeJ68vh9>

Starting in the name of Allah,



*the most beneficial,
the most merciful.*

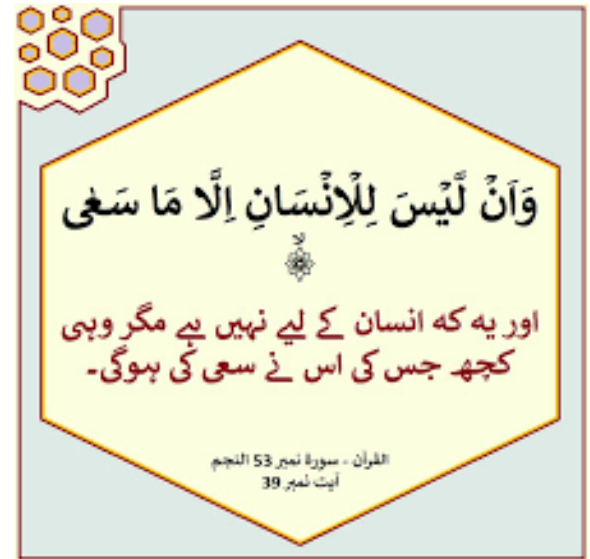
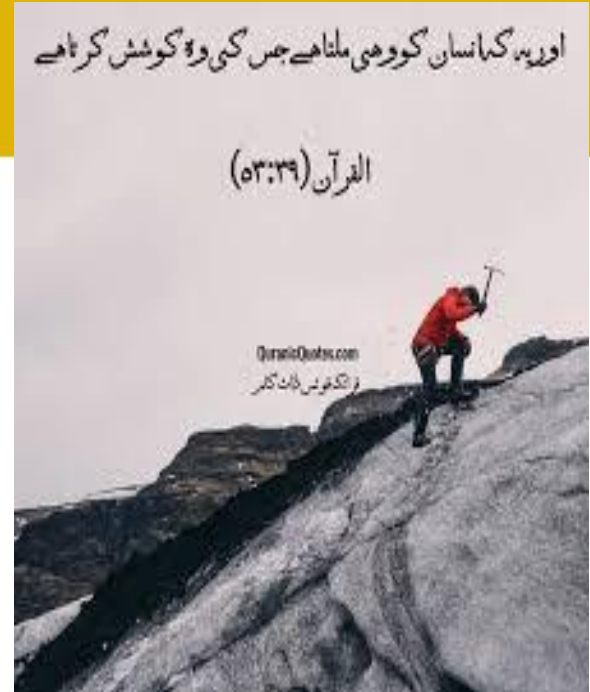
أَمْرٌ لِلْإِنْسَانِ مَا تَبَىٰ
تَبَىٰ ۚ

کیا انسان کو ہر وہ چیز حاصل ہے جس کی اس نے تمنا کی؟



UNIVERSITY OF
KARACHI

And there is not for man except that [good] for which he strives.





Week 12

Internet Application Development

Copyright © 2025, Humera Tariq

Department of Computer Science (DCS/UBIT)

University of Karachi

January 2025

Exams Guidance

JJ BOOK chap1 – chap4

Tables/Figures/captions/bolds

+

Exercises 1-13 (chap 2,3,4)

+

All Lecture slides

+

class discussions + relevant links + grp presentation

THEORY
PRACTICE

History of Javascript

Know your browser

The rendering engine

Javascript at runtime

JavaScript
Anatomy

Javascript engine

Call stack

Memory heap

Memory leak

Stack overflow

Garbage collection

Synchronous

Callback Queue

Event loop

3 Ways to promise

JavaScript
Foundations

JavaScript Pro

JavaScript
Under the Hood

Execution Context

Hoisting

Lexical Environment

Scope Chain

Closure

This

Let and Const

Arrow Function

Call, Apply, Bind

~~Function constructor~~

Prototype vs proto

Callback Object

HOC

IIFE

The 2 Pillars:
Closure
and
Prototypes

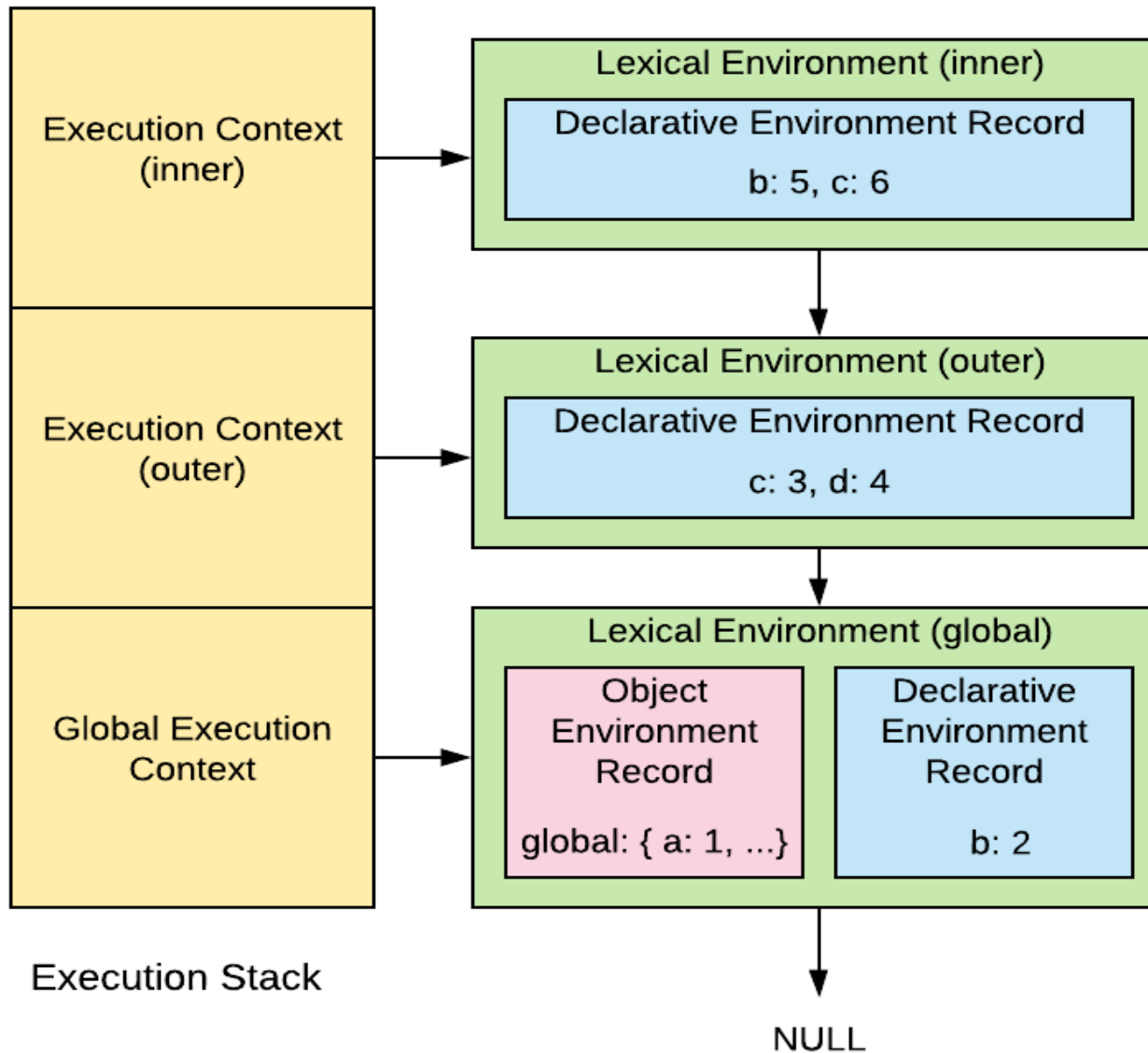
Execution context (EC)



Scope chain, closure, hof, prototype chain

When we call and enter a function a new _____ is created on the _____ to keep track of _____ as we execute the function's code.

How do we think if we want to log the value of variable 'a' from inner ?



```
var a = 1;
let b = 2;

function outer() {
  let c = 3;
  var d = 4;

  function inner() {
    let b = 5;
    let c = 6;

    // diagram refers to this

    console.log(a); // 1
    console.log(b); // 5
    console.log(c); // 6
    console.log(d); // 4
  }

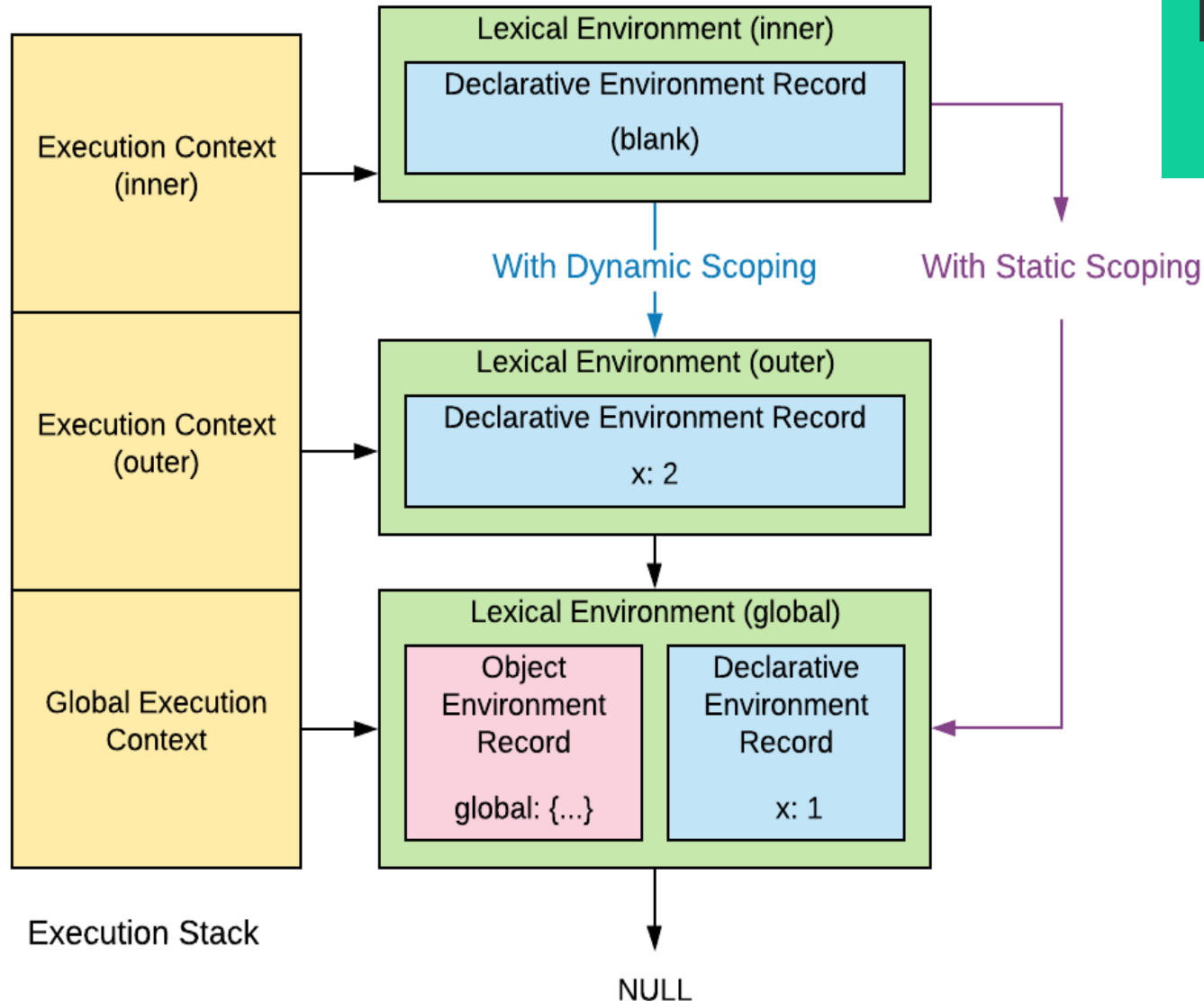
  inner();
}

outer();
```


Static vs. dynamic scoping

Static ~ Lexical ~ Compile time stuff

Static vs Dynamic Scoping (non-local variables)



The answer is _____.
Why?

```
const x = 1;

const inner = () => {
  console.log(x);
};

const outer = () => {
  const x = 2;
  inner();
};

outer();
```

Static vs Dynamic Scoping (non-local variables)

The answer is _____. Why? When `inner` is called, `x` is defined in _____ (outer/inner) EC which sits just below inner's EC, but Javascript doesn't care about that. Javascript uses _____ (static/dynamic) scoping, meaning it only cares about which variables are in-scope at the time a function is created.

```
const x = 1;

const inner = () => {
  console.log(x);
};

const outer = () => {
  const x = 2;
  inner();
};

outer();
```

When `inner` is created, lexically the top-level `x` (value 1) is in scope and `outer's` `x` (value 2) is not, so it only refers to that `x` (value 1) when it's called.

JavaScript (with static scoping — the real behavior)

Dynamic scoping (Bad/not preferred in modern programming):

"Hey, whenever you call me, I'll just *guess* where my variables are based *on the situation*."

Static (lexical) scoping:

"I know *exactly* where my variables are because of how and where I was *born (written)*."

```
let a = 10;

function foo() {
  console.log(a);
}

function bar() {
  let a = 20;
  foo(); // What will this print?
}

bar();
```

- ✓ *foo()* looks for a **where it was** _____, not where it was *called*.
- ✓ Since *foo* was _____ when *a* = _____ in **global scope**, it prints _____, **even though** *bar()* has a different *a* = _____ **locally**.

Static vs. dynamic scoping

Closure → static scoping

Closure → hof -> dynamic memory allocation

```
const x = 1;
```

```
const inner = () => {  
  console.log(x);  
};
```

```
const outer = () => {  
  const x = 2;  
  inner();  
};
```

```
outer();
```

```
JS closure.js > ...
```

```
1  const x = 1;
```

```
2
```

```
3  const outer = () => {
```

```
4    const x = 2;
```

```
5
```

```
6    const inner = () => {
```

```
7      console.log(x); // 2
```

```
8    };
```

```
9
```

```
10   return inner;
```

```
11 };
```

```
12
```

```
13 const foo = outer();
```

```
14
```

```
15 foo();|
```

Here, the inner function is defined inside the _____ function. The first thing to note is that now we'll be **console logging x with a value of 2, not 1**, because the **x** from the _____ function appears **first lexically** as we zoom out from the inner function.

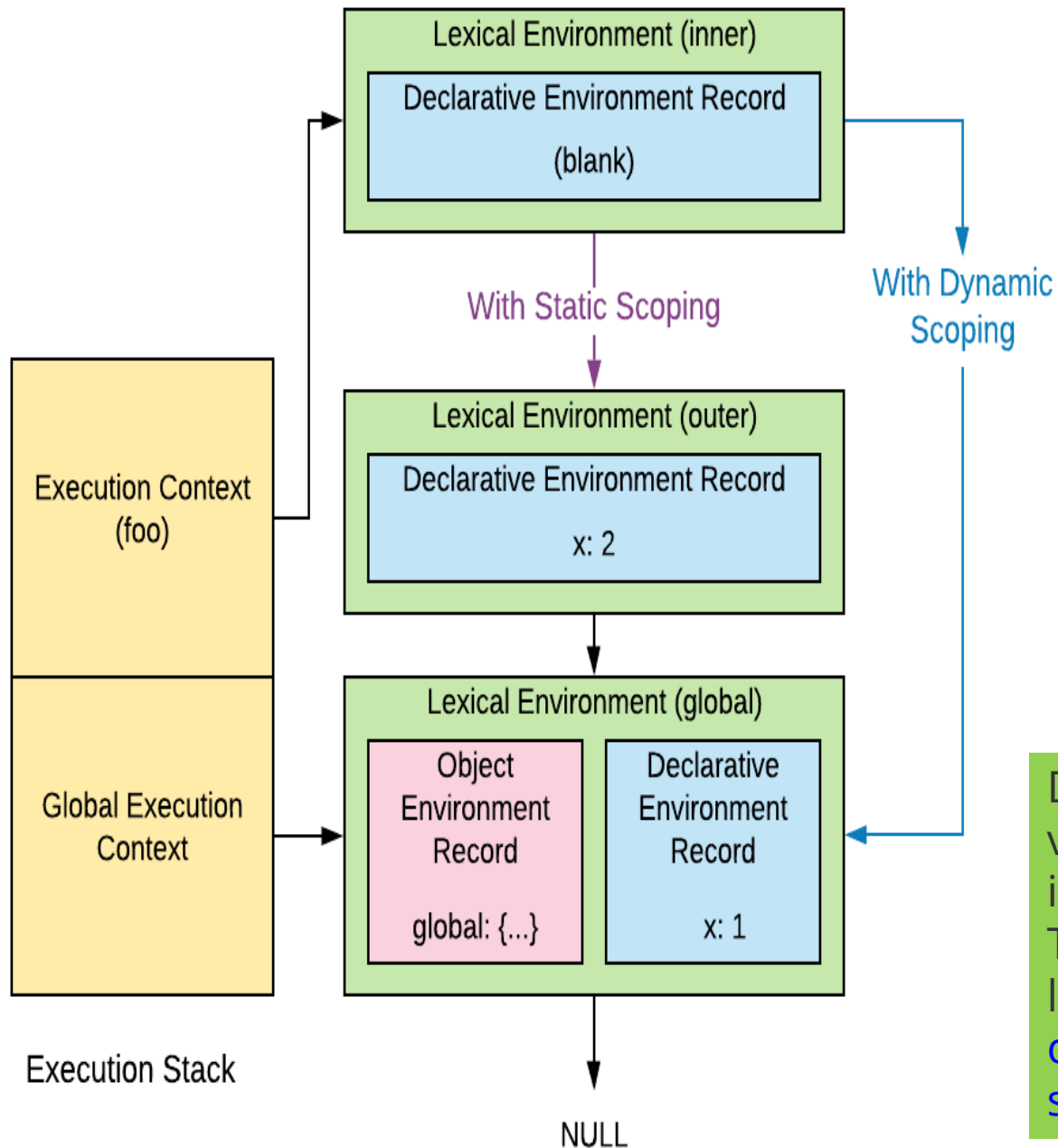
But the more important thing is that **by the time we call**, _____ and we've assigned our _____ function to **foo**, _____ execution context (**stack frame**) can no longer live on the _____ because **we're now outside that function completely**. **So how can foo know to output 2 when we reach console.log(x)?**

JS closure.js > ...

```
1  const x = 1;
2
3  const outer = () => {
4    const x = 2;
5
6    const inner = () => {
7      console.log(x); // 2
8    };
9
10   return inner;
11 };
12
13 const foo = outer();
14
15 foo();
```

This brings us to closures. *A closure is simply a function paired with a reference to its parent environment.* When a function makes reference to an outer function's variables, it's said to capture those variables, or 'close over' those variables (hence the term 'closure').

When we create our _____ function we actually create a _____ consisting of the _____ function and a _____ to the lexical environment of the _____ function.



When inner is created, it stores an internal `[[Scopes]]` property which captures the _____ of the _____ function. Then, when we call foo we use the `[[Scopes]]` property to traverse the _____ and find the value of x.

Dynamic scoping provides no way of closing over variables, because it will always traverse the call stack itself to find the value of a given variable at runtime. That means as soon as an outer function returns, its local variables are lost. Static scoping allows for closures, but by supporting closures we introduce some new complexities to memory management.

Memory fixed or not ???

JS closure.js > ...

```
1  const x = 1;
2
3  const outer = () => {
4    const x = 2;
5
6    const inner = () => {
7      debugger;
8      console.log(x); // 2
9    };
10
11   return inner;
12 };
13
14 const foo = outer();
15
16 debugger;
17
18 foo();
```

```
▼ Scope
▼ Script
  ▼ foo: () => { debugger; console.log(x); // 2 }
    arguments: (...)
    caller: (...)
    length: 0
    name: "inner"
    ► __proto__: f ()
      [[FunctionLocation]]: index.js:6
      ▼ [[Scopes]]: Scopes[3]
        ► 0: Closure (outer) {x: 2}
        ► 1: Script {x: 1, outer: f, foo: f}
        ► 2: Global {parent: Window, opener: null, t...
      ► outer: () => {...}
        x: 1
    ► Global
```

Window

```
▼ foo = () => {
  arguments = f ()
  caller = f ()
  length = 0
  name = 'inner'
  [[FunctionLocation]] = @ d:\____B
  > [[Prototype]] = f ()
  ▼ [[Scopes]] = Scopes[2]
    > 0 = Closure (outer) {x: 2}
    > 1 = Global {global: global, queu
  > module = Module {id: '.', path: 'D
  > outer = () => {
  > require = f require(path) {
  > this = Object
  x = 1
```

Fantastic Ideas– Memory + hof/closure as dynamic objects

```
function foo(a) {  
  return function(b) {  
    return a === b;  
  };  
}
```



```
const isFive = foo(5);    // Create a closure where a = 5  
  
console.log(isFive(5));  // true    -> because 5 === 5  
console.log(isFive(6));  // false   -> because 5 !== 6  
  
const isHello = foo("hello"); // a new closure where a = "hello"  
  
console.log(isHello("hello")); // true  
console.log(isHello("world")); // false
```

This proves that a is "remembered" correctly even though foo already finished.

In V8, a 'Context' is created inside a scope if there are any functions defined in that scope that reference any of the scope's variables. The Context contains all the variables in that scope which were captured by one or more functions.

That context then lives on the heap rather than the stack and the functions which captured any variables retain a reference to the context.

Why Closures Require Heap Allocation?

this vs closure?



```
JS heap.js > ...
1  const outer = () => {
2      const x = 2;
3      const y = 1;
4      const HUGE = { one: 1, two: 2 }; //
5
6      const bar = () => {
7          console.log(HUGE);
8      };
9
10     const inner = () => {
11         debugger;
12         console.log(x);
13     };
14
15     return inner;
16 };
17
18 const foo = outer();
19
20 // do lots of time-consuming stuff
21
22 foo();|
```

Problem:?????



- ✓ outer() runs, it creates **x**, **y**, **HUGE**, **bar**, and **inner**.
- ✓ It **returns inner**.

✓ **inner function** forms a closure over the **variables** it *uses* or *might need*.

✓ **inner** uses only closure, and **NOT** global scope directly.



Problem: **closure-related memory leaks**.

- ✓ *primitive variables that otherwise would have lived on the **stack** now live in the **heap***
- ✓ *large objects may now stick around in the **heap** longer than before*

- ✓ Even though inner doesn't *use* HUGE,
- ✓ **Because *outer()*'s whole environment is closed over**, and **HUGE** is part of that environment,
- ✓ **The huge object stays in memory** until **foo (the closure)** is garbage collected.
- ✓ This means: If HUGE is truly massive (MBs or GBs), your app **wastes memory unnecessarily**.

Example : How to fix /optimization:



If you **don't need** bar and HUGE after outer() runs, you should.

- ✓ Separate what inner actually needs
- ✓ Avoid keeping references to unnecessary variables.

```
const outer = () => {  
  const x = 2;  
  const inner = () => {  
    console.log(x);  
  };  
  return inner;  
};
```



Now, no reference to HUGE, and memory is not unnecessarily retained.

Who ?	Where It Lives	Purpose / Behavior
Global Execution Context	Stack initially, references objects on Heap	Special first context created when script starts. Holds window / global this.
Execution Context	Stack	Created each function call, handles local variables, this, scope. Dies after call unless closure.
Closure Context (V8 Context)	Heap	Captures variables needed by inner functions. Survives after outer function ends.
Lexical Environment ECMAScript specification/document	Abstract model → realized via Contexts	Invisible structure linking variable names to memory slots. Implemented through Execution or Closure Contexts.

Daniyal Ahmed B21110006025 4/22/2025 2:28 PM

Assalamoalikum,

In line with today's discussion, I'd like to share an analogy that really helped me understand the concept of closure.

Imagine a parent (the outer function) preparing a lunch box (the lexical environment) for their child (the inner function) before school. The parent packs it with a sandwich, juice, and snacks (these represent the variables). Even after the child goes to school — far away from the parent (i.e., the outer function has finished executing) — the child can still open the lunch box and access the food (the inner function can still access the variables from its outer scope).

I would really appreciate anyone who would like to add more to our knowledge as closures are a very important interview question in most JavaScript based job roles.

```
function parent() {  
  let lunchbox = { snacks: "Chips" };  
  
  const child = function() {  
    console.log("Snacks:", lunchbox.snacks);  
  };  
  
  // Inner function keeps a reference!  
  setTimeout(() => {  
    lunchbox.snacks = "Brownie"; // Magical update  
  }, 2000);  
  
  return child;  
}
```

```
const childWithWifiLunchbox = parent();  
  
childWithWifiLunchbox(); // Shows "Chips"  
setTimeout(() => {  
  childWithWifiLunchbox(); // Shows "Brownie" after 2 seconds  
, 3000);
```

Dr. Humera Tariq 4/22/2025 6:48 PM

Just enjoy your analogy with gpt 🤖 "It's like the child takes a magical lunchbox ✨ to school — one with Wi-Fi 📶! Even though the parent (outer function) is no longer around, the lunchbox (lexical environment) stays connected 📶 to the kitchen (the scope). So if the parent updates the snacks before leaving, the child (inner function) still sees those changes. The child doesn't have a copy — they have a live reference to what's inside."

Reduce Repeated Calls with Throttling and Debouncing

Debouncing (delay the execution)

Search

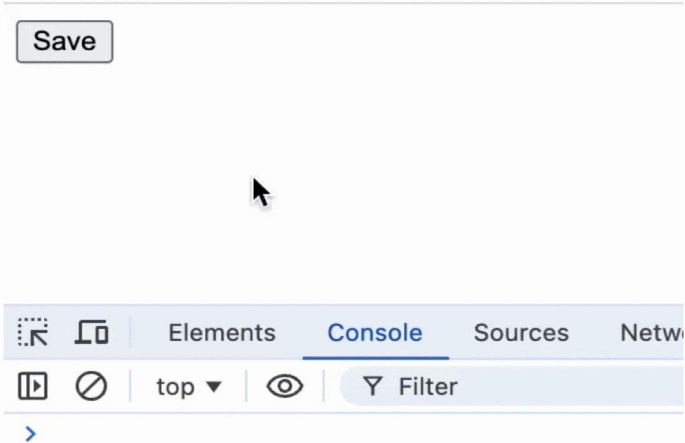
No of times event fired


```
/**
 * Function: debounce
 * -----
 * Returns a debounced version of the provided callback function.
 * The debounced function delays the execution of the callback until
 * after a specified delay
 * has elapsed since the last time the debounced function was invoked.
 *
 * Parameters:
 * - callback: The function to debounce.
 * - time: The delay in milliseconds.
 *
 * Returns:
 * - A debounced version of the callback function.
 */
```

Key Concepts in debouncing example 1

- *Closures*: The returned function maintains access to the *'interval'* variable defined in the *outer scope*.
- *Higher-Order Functions*: *'debounce'* takes a function as an argument and returns a new function. *(any one is fine)*

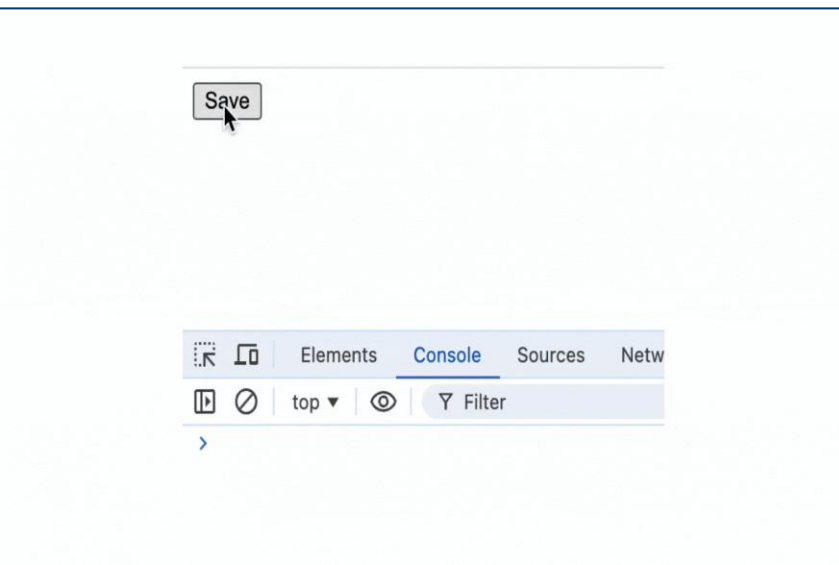
```
1 <!doctype html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8" />
5     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6   </head>
7   <body>
8     <button id="app">Save</button>
9     <!-- adds 'click' event listener to button -->
10    <script>
11      document.getElementById("app").addEventListener("click", () =>
12        console.log('document saved!'))
13    </script>
14  </body>
15 </html>
```



```

11 // define function that event handler will invoke on "click"
12 function saveDoc(){
13     console.log('document saved!')
14 }

```



```

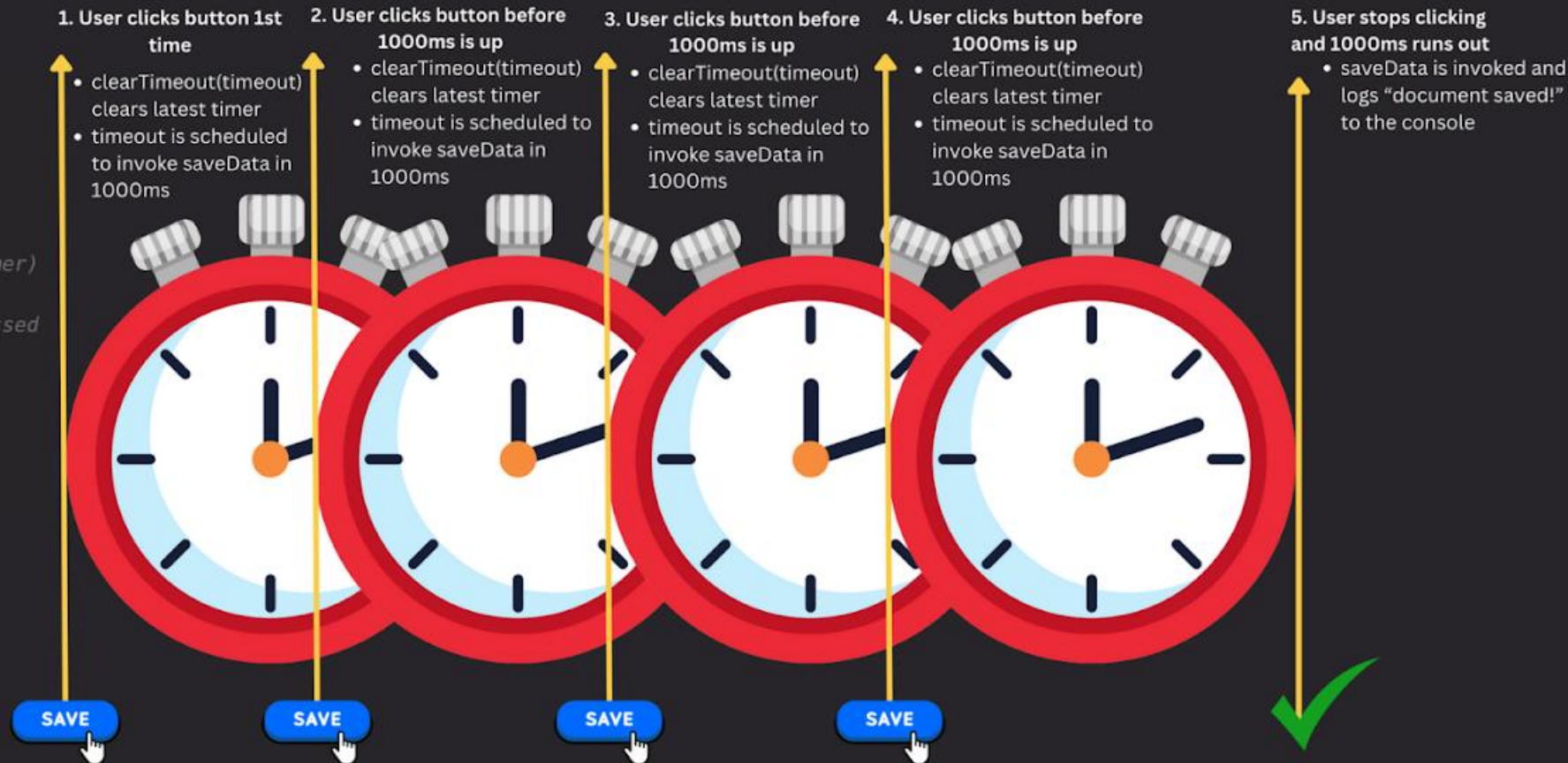
11 // define function that event handler will invoke on "click"
12 function saveDoc() {
13     console.log('document saved!');
14 }
15 //access button and add event listener.
16 //on click invoke debounce passing in saveDoc and interval
17 document
18     .getElementById('app')
19     .addEventListener('click', debounce(saveDoc, 2000));
20
21
22 function debounce(func, waitTime) {
23     //declare variable timeout and leave undefined
24     let timeout;
25
26     //return anonymous function definition
27     return function () {
28
29         //cancel function in setTimeout (cancel timer)
30         clearTimeout(timeout); RESET and prevent the callback from firing
31                                 too quickly
32         //schedule func to run after waitTime has passed (reset timer)
33         timeout = setTimeout(() => {
34             //in this case, "func" will be "saveDoc"
35             func();
36
37         }, waitTime);
38     };
39
40 }

```

- ✓ clear any existing timer;
- ✓ start a new timer;
- ✓ when timer finishes without being reset, run the function;

```
16 //on click invoke debounce passing in saveDoc and interval
17 document.getElementById("app").addEventListener("click", debounce(saveDoc, 1000))
```

```
18 //takes in a function and an interval (our timer)
19 function debounce(func, waitTime){
20   //declare variable timeout and leave undefined
21   let timeout;
22   //return anonymous function definition
23   return function(){
24     //cancel function in setTimeout (cancel timer)
25     clearTimeout(timeout)
26     //schedule func to run after waitTime has passed
27     timeout = setTimeout(() => {
28       //in this case, "func" will be "saveDoc"
29       func();
30     }, waitTime)
31   }
32 }
```



```
let debounceTimer;

function debounceExample() {
  // Clear any previous timer (reset the "stickiness")
  clearTimeout(debounceTimer);

  // Set a new timer (wait for 300ms before the ball bounces back)
  debounceTimer = setTimeout(() => {
    console.log("Ball is bouncing back!"); // Ball finally returns after silence
  }, 300);
}

// Simulate rapid throws (event triggers)
debounceExample(); // Ball will not bounce yet
debounceExample(); // Ball will not bounce yet
debounceExample(); // Ball will not bounce yet
// The ball will only bounce back after 300ms of silence (no more throws)
```


Limit Rate

<https://codesandbox.io/p/sandbox/throttle-example-elzis?file=%2Fsrc%2Findex.js>

```
let counter = 0;
let counterRegular = 0;
document.getElementById("btn").addEventListener(
  "click",
  throttle(function () {
    counter = counter + 1;
    document.getElementById("clickCount").innerText = "Click Count: " + counter;
  }, 1500)
);

document.getElementById("btnRegular").addEventListener("click", function () {
  counterRegular = counterRegular + 1;
  document.getElementById("clickCountRegular").innerText =
    "Click Count: " + counterRegular;
});
```

Throttled Button

Click me!

Click Count: 1

Regular Button

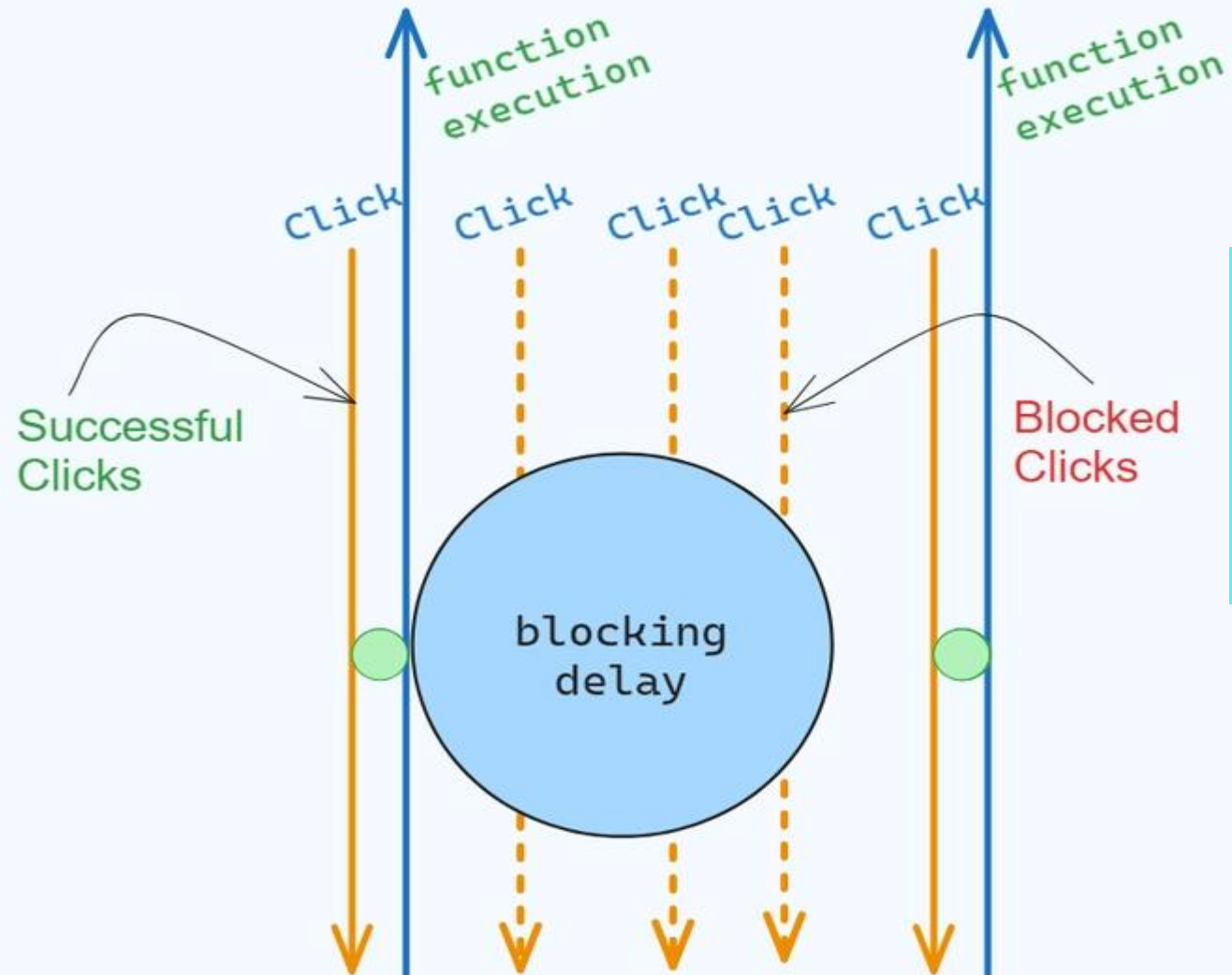
Click me!

Click Count: 13

```
src > index.js > ...
1  const throttle = (func, duration) => {
2    let shouldWait = false;
3    return (...args) => {
4      if (!shouldWait) {
5        func.apply(null, args);
6        shouldWait = true;
7        setTimeout(function () {
8          shouldWait = false;
9        }, duration);
10     }
11   };
12 };
13
```

Throttling is a technique that limits how often a function can be called in a given period of time. It is useful for improving the performance and responsiveness of web pages that have event listeners that trigger heavy or expensive operations, such as animations, scrolling, resizing, fetching data, etc.

- ✓ use a timer function such as `setTimeout`
- ✓ `difference of time` method (old method) to wrap the function being throttled.



```
if (enoughTimePassed)
{
  run the function
}
```

```
let lastTime = 0;

function throttleExample() {
  const now = Date.now();

  if (now - lastTime > 300) { // Throttle: Allow only once every 300ms
    console.log("Ball is thrown!"); // Ball is thrown and returns
    lastTime = now;
  } else {
    console.log("Wait! Ball is still being thrown too soon.");
  }
}

// Simulate rapid throws (event triggers)
throttleExample(); // This will go through
throttleExample(); // This will be blocked
setTimeout(throttleExample, 350); // This will go through after 350ms
```

```
// Define a function that fetches some data from an API
function fetchData() {
  console.log("Fetching data...");
  // Simulate an API call with a random delay
  setTimeout(() => {
    console.log("Data fetched!");
  }, Math.random() * 1000);
}

// Throttle the fetchData function with a delay of 5000 ms
const throttledFetchData = throttle(fetchData, 5000);

// Add an event listener to the window scroll event that calls the throttledFetchData function
window.addEventListener("scroll", throttledFetchData);
```

```
function throttle(mainFunction, delay) {
  let timerFlag = null; // Variable to keep track of the timer

  // Returning a throttled version
  return (...args) => {
    if (timerFlag === null) { // If there is no timer currently running
      mainFunction(...args); // Execute the main function
      timerFlag = setTimeout(() => { // Set a timer to clear the timerFlag after
        timerFlag = null; // Clear the timerFlag to allow the main function to be
      }, delay);
    }
  };
}
```

Debouncing vs. Throttling

Debounce: Only one event is triggered after a series of events.

Throttle: Only one event is triggered during a certain time period.



Anas Hussain B21110006017 4/29/2025 1:12 PM

As @Dr. Humera Tariq said that she was finding it difficult to differentiate between logical use cases of throttling and debouncing, Just wanted to share a quick breakdown that really helped me understand the difference:

Throttle is when you want to limit how often a function runs over time. Example: you've got a button that updates a value in your DB, but the DB can only handle 2 updates per second (a lot of such platforms have query limits, which if exceeded, throws you an error of "too many requests"). Throttling makes sure no matter how many times the user clicks, it only fires at a controlled rate, say, max 2 times/sec.

Debounce is when you want to wait until the user stops doing something before taking action. Like a search input field in a movie streaming app: without debouncing, if a user types "g", "go", "god", "godf", "godfa", "godfat", "godfath", "godfathe", "godfather" you might trigger 9 API calls. Debouncing waits until the user stops typing (like 300–500ms), and then fires just one clean call.



Daniyal Ahmed B21110006025 4/29/2025 1:45 PM

One real life application for this would be a search bar on an ecommerce web app:





In a product search bar, debouncing and throttling help manage API requests efficiently as a user types. With debouncing, the app waits until the user has stopped typing for a short delay, like 300 milliseconds, before making a single API call—this ensures only the final search term triggers a request, reducing unnecessary calls. In contrast, throttling allows the app to make one API call every fixed interval, such as once per second, regardless of how many keys the user presses, which means some characters typed during that interval may still cause intermediate searches. While debouncing is ideal for minimizing calls and focusing on the final input, throttling is better when updates should happen at a steady pace during continuous activity.







Nabeel Khan 4/29/2025 1:24 PM

Throttling vs. Debouncing: Event Handling Techniques

Throttling:

-  **Execution:** Runs a function at most once within a specified time interval.
-  **Logic:** Like a gate that opens periodically; ignores events that happen too soon.
-  **Use Case:** Control the *rate* of execution. Good for continuous events where you need updates, but not for every single micro-movement (e.g., scrolling, resizing).
-  **Think:** "Allow this to happen, but no more often than X times per second."

Debouncing:

-  **Execution:** Runs a function *after* a period of inactivity has passed since the *last* event.
-  **Logic:** Waits for the "dust to settle"; resets the timer if another event occurs before the delay is up.
-  **Use Case:** React to the *final* state after a rapid series of events. Good for actions that should only trigger when the user is "done" (e.g., typing in a search box, clicking a button repeatedly).
-  **Think:** "Wait for things to stop happening before doing this."

In short:

- **Throttling:** Limits how *often* something can happen.
- **Debouncing:** Delays something until events *stop* happening.

removeDuplicates.js

Identify the Higher Order function and justify your choice

HOF → closure

Unlike first-order functions, higher-order functions may require memory to be dynamically allocated at runtime. Why?

Higher-order functions allow for functions to be created _____ at runtime. In languages like **JavaScript**, **functions are first-class objects**, meaning, they can be saved in variables, passed to other (possibly higher-order) functions, and returned from (possibly higher-order) functions. To enable this, hof are typically managed on the _____ at runtime.

JS removeDuplicates.js X

JS removeDuplicates.js > ...

```
1 // Sample array with duplicate integers
2 const arr = [1, 2, 3, 2, 4, 3, 5, 1, 6];
3
4
5 const uniqueUsingSet = [...new Set(arr)];
6 console.log('Unique using Set:', uniqueUsingSet);
7 // Output: [1, 2, 3, 4, 5, 6]
```

1

By converting the array to a Set and then back to an array, duplicates are removed.

It checks if the current element is already in the uniqueArray and adds it if not.

```
10 const uniqueUsingLoop = [];
11 for (let i = 0; i < arr.length; i++) {
12     if (!uniqueUsingLoop.includes(arr[i])) {
13         uniqueUsingLoop.push(arr[i]);
14     }
15 }
16 console.log('Unique using Loop:', uniqueUsingLoop);
17
```

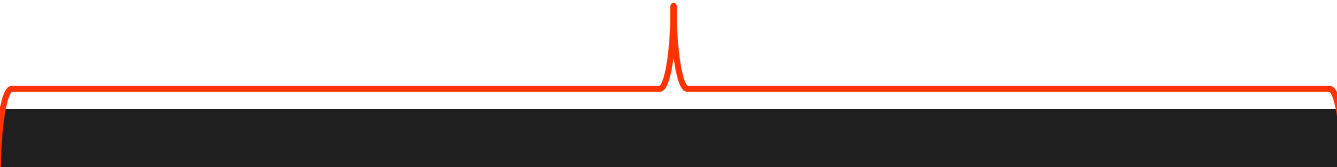
2

The filter() callback signature: `arr.filter(callbackFn, optional thisArg)`

3

- ✓ `filter` is a `hof` that takes a callback function as an argument.
 - Returns a Function
 - enabling function `factories` and `closures`
 - Returning functions is key to patterns like `currying`, `partial application`, and `higher-level abstractions`
- ✓ The `callback` determines whether each element should be included in the new array.

```
18
19  const uniqueUsingFilter = arr.filter((value, index, self) => self.indexOf(value) === index);
20  console.log('Unique using filter():', uniqueUsingFilter);
21
```



reducer() signature explained

```
// accumulator -> required  
// currentValue -> current element being processed in array  
// currentIndex -> current index of processed value  
// array -> a reference to the array reduce() was called upon  
const reducer = (accumulator, currentValue, currentIndex, array) => {}
```

Required. The accumulator accumulates the callback's return values; it is the accumulated value previously returned in the last invocation of the callback, or initialValue, if supplied (see below).

Required. The current element being processed in the array.

Optional. The index of the current element being processed in the array. Starts at index 0, if an initialValue is provided, and at index 1 otherwise.

Optional. The array reduce() was called upon.

reduce + includes, slow for large arrays

```
22 const uniqueUsingReduce = arr.reduce((accumulator, currentValue) => {  
23   if (!accumulator.includes(currentValue)) {  
24     accumulator.push(currentValue);  
25   }  
26   return accumulator;  
27 }, []);  
28 console.log('Unique using reduce():', uniqueUsingReduce);
```

*Higher-order functions can either _____ functions,
_____ functions, or _____ but not always both!*

forEach,map,reduce,filter accepts a _____ function as an
argument and _____ that function on _____ of the array.

✓ seen is now _____ or leaked outside the _____ .

✓ If you have multiple reduces in the same file, they might accidentally clash or _____ .

✓ It breaks _____ OOPS principal .

```
const numbers = [1, 2, 2, 3, 4, 4, 5];
const seen = new Set(); // has to be declared OUTSIDE

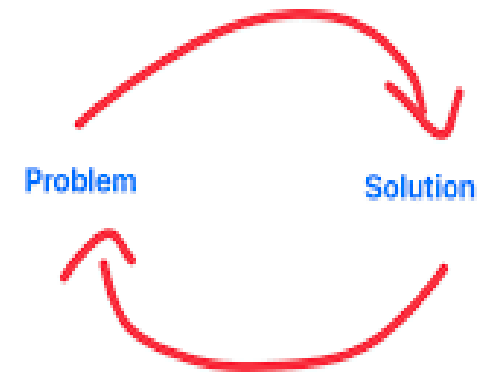
const uniqueNumbers = numbers.reduce((accumulator, currentValue) => {
  if (!seen.has(currentValue)) {
    seen.add(currentValue);
    accumulator.push(currentValue);
  }
  return accumulator;
}, []);

console.log(uniqueNumbers); // [1, 2, 3, 4, 5]
```



// createUniqueReducer: A factory function that creates a reducer function.
// seen: A Set that keeps track of seen elements, maintained via closure.
// Reducer Function: Adds currentValue to accumulator only if it hasn't been seen before.
// Now with the factory (createUniqueReducer()):
// seen is private inside the returned function.
// Each time you call createUniqueReducer(), you get a fresh, isolated seen set.
// This uses the concept of a closure!
// seen is closed over by the returned reducer function.

```
function createUniqueReducer() {  
  const seen = new Set();  
  return function(accumulator, currentValue) {  
    if (!seen.has(currentValue)) {  
      seen.add(currentValue);  
      accumulator.push(currentValue);  
    }  
    return accumulator;  
  };  
}  
  
const numbers = [1, 2, 2, 3, 4, 4, 5];  
const uniqueNumbers = numbers.reduce(createUniqueReducer(), []);  
  
console.log(uniqueNumbers); // Output: [1, 2, 3, 4, 5]
```



```
const uniqueUsingReduce = arr.reduce((accumulator, currentValue) => {  
  if (!accumulator.has(currentValue)) {  
    accumulator.add(currentValue);  
  }  
  return accumulator;  
}, new Set());
```

```
function createUniqueReducer() {  
  const seen = new Set();  
  return function(accumulator, currentValue) {  
    if (!seen.has(currentValue)) {  
      seen.add(currentValue);  
      accumulator.push(currentValue);  
    }  
    return accumulator;  
  };  
}  
  
const numbers = [1, 2, 2, 3, 4, 4, 5];  
const uniqueNumbers = numbers.reduce(createUniqueReducer(), []);  
  
console.log(uniqueNumbers); // Output: [1, 2, 3, 4, 5]
```

Same and different



"Reduce without Set is slow.
Reduce with Set or Filter with Set is fast.
Closure protects the Set."

Practice HOF with
reasoning

Question 1

```
var f = function (x) {  
    return function (y) { return y - x;  
};  
};  
  
var g = f(7);  
console.log(g(5));
```

Question 2

Given the following code:

```
var b = 6;  
  
var foo = function (a) {  
    a = b + a;  
    return function () { return a; };  
};  
  
b = 2;  
  
var bar = function () {  
    var b = 3;  
    return foo(b);  
};
```

what does bar() evaluate to?

(A) 5 (B) 6 (c) error (D) a function (E) 9

what does bar () () evaluate to?

(A) a function (B) 6 (c) error (D) 9 (e) 5

Question 3

This problem uses the same code as the previous two problems but focuses on the difference between static and dynamic binding rules.

Given the following code:

```
var b = 6;

var foo = function (a) {
  a = b + a;
  return function () { return a; };
};

b = 2;

var bar = function () {
  var b = 3;
  return foo(b);
};
```

what would `bar()()` evaluate to, if JavaScript were dynamically scoped?

(A) 5 (B) 6 (c) error (D) a function (E) 9

Practice closure with reasoning

"Closures allow JavaScript to emulate private variables by keeping a function's local state alive after its parent has returned, solving real memory safety and privacy problems."

Question 1

which of following is correct implementation of counter why or why not ?

```
let count = 0; // Global!

function increment() {
  count++;
  return count;
}

console.log(increment()); // 1
console.log(increment()); // 2

// But now
count = 999;
console.log(increment());
```

```
function createCounter() {
  let count = 0;

  return function() {
    count++;
    return count;
  };
}

const counter = createCounter();

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

Question 1

Write the output against each function call `loginAttempt()`;

```
function createLoginAttemptTracker() {  
  let attempts = 0;  
  
  return function() {  
    attempts++;  
    if (attempts > 3) {  
      console.log('Account locked!');  
    } else {  
      console.log(`Attempt ${attempts}`);  
    }  
  };  
}  
  
const loginAttempt = createLoginAttemptTracker();  
loginAttempt();  
loginAttempt();  
loginAttempt();  
loginAttempt();
```


Question: Explain what is the purpose of following program and write output.

```
function createMemoizedSquare() {  
  const cache = {};  
  
  return function(n) {  
    if (cache[n]) {  
      console.log('From cache:', cache[n]);  
      return cache[n];  
    }  
    const result = n * n;  
    cache[n] = result;  
    console.log('Calculated:', result);  
    return result;  
  };  
}  
  
const square = createMemoizedSquare();  
square(4);  
square(4);  
square(5);
```

Remove duplication practice with both front end and backend
(Self-study for exam)

Remove Duplicates using filter vs Closure +set + filter (Memory Safety)

```
function createUniqueCollector() {  
  const seen = new Set();  
  
  return function(value) {  
    if (!seen.has(value)) {  
      seen.add(value);  
      return true;  
    }  
    return false;  
  };  
}
```

```
18  
19   const uniqueUsingFilter = arr.filter((value, index, self) => self.indexOf(value) === index);  
20   console.log('Unique using filter():', uniqueUsingFilter);  
21
```

```
const arr = [1, 2, 2, 3, 4, 4, 5];  
const collector = createUniqueCollector();  
const uniqueArr = arr.filter(collector);  
  
console.log(uniqueArr); // [1,2,3,4,5]
```

To be Run single file server.js

npm init -y

npm install express

npm install --save-dev nodemon

npm install cors

package.json → "dev": nodemon server.js

package.json → type: module

npm run dev

```
package.json > {} devDependencies > nodemon
1  {
2    "name": "backend_likes",
3    "version": "1.0.0",
4    "description": "",
5    "main": "server.js",
6    "type": "module",
7
8
9    > Debug
10   "scripts": {
11     "test": "echo \"Error: no test specified\" && exit 1",
12     "start": "node server.js",
13     "dev" : "nodemon server.js"
14   },
15   "keywords": [],
16   "author": "",
17   "license": "ISC",
18   "dependencies": {
19     "express": "^5.1.0"
20   },
21   "devDependencies": {
22     "nodemon": "^3.1.10"
23   }
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS powershell

```
PS D:\_\_\_BSCS_IAD\_\_week12\backend_likes> curl http://localhost:5000/api/shared-posts

StatusCode      : 200
StatusDescription : OK
StatusDescription : OK
Content         : [{"userId":1,"postId":101,"action":"share"}, {"userId":2,"postId":101,"action":"share"}, {"userId":3,"postId":102,"action":"share"}, {"userId":2,"postId":102,"action":"share"}, {"userId":4,"postId":103
RawContent      : HTTP/1.1 200 OK
                  Connection: keep-alive
                  Keep-Alive: timeout=5
```

Server.js

```
import cors from 'cors'; // <-- ADD THIS
```

```
app.use(cors()); // <-- ALLOW all origins for now (good for development)
```

```
import express from 'express';
import { getSharedPosts } from './controllers/postController.js';

const app = express();

// Root path
app.get('/', (req, res) => {
  res.send('Hello, world!');
});

// Shared posts route
app.get('/api/shared-posts', _____ );

// Start the server
app.listen(5000, () => {
  console.log('Server running on http://localhost:5000');
});
```

EXPLORER

...

▼ BACKEND_LIKES

▼ controllers

JS postController.js

> node_modules

▼ services

JS uniqueService.js

{ } package-lock.json

{ } package.json

JS server.js

Server.js → postController.js → uniqueService.js

```
import { createUniqueReducer } from '../services/uniqueService.js';
```

```
export const getSharedPosts = (req, res) => {  
  const shares = [  
    { userId: 1, postId: 101, action: 'share' },  
    { userId: 2, postId: 101, action: 'share' },  
    { userId: 1, postId: 101, action: 'share' },  
    { userId: 3, postId: 102, action: 'share' },  
    { userId: 2, postId: 102, action: 'share' },  
    { userId: 4, postId: 103, action: 'share' }  
  ];
```

```
  // Apply the unique reducer to filter out duplicates  
  const uniqueShares = shares.reduce( _____ , []);  
  res.json( _____ ); // Return unique shared posts  
};
```

```
// Function to create a reducer for filtering unique posts  
export function createUniqueReducer() {  
  const seen = new Set(); // Using Set to track unique values  
  return function(accumulator, currentValue) {  
    const key = `${currentValue.userId}-${currentValue.postId}`;  
    if (!seen.has(key)) {  
      seen.add(key);  
      accumulator.push(currentValue);  
    }  
    return accumulator;  
  };  
}
```

Front end react-vite setup

`npm create vite@latest remove-duplicates --template react`

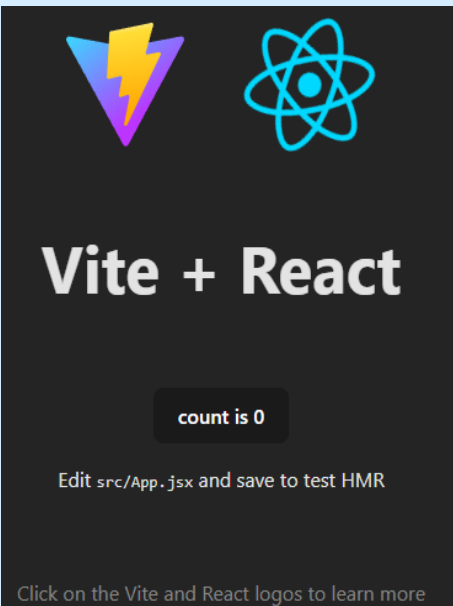
`cd remove-duplicates`

`npm install`

`npm run dev`

Shared Posts

- User 1 shared post 101
- User 2 shared post 101
- User 3 shared post 102
- User 2 shared post 102
- User 4 shared post 103



```
35   return (  
36     <div className="App">  
37       <h1>Shared Posts</h1>  
38       <ul>  
39         {shares.map((share, index) => (  
40           <li key={index}>`User ${share.userId} shared post ${share.postId}`</li>  
41         ))}  
42       </ul>  
43     </div>  
44   );
```

Front end react-vite setup

Shared Posts

- User 1 shared post 101
- User 2 shared post 101
- User 3 shared post 102
- User 2 shared post 102
- User 4 shared post 103

```
useEffect(() => {  
  const fetchShares = async () => {  
    // function definition  
    ...  
  };  
  
  fetchShares();  
  // function call  
}, []);
```

// Fetch shared posts from the backend API

```
useEffect(() => {  
  const fetchShares = async () => {  
    try {  
      const response = await axios.get('http://localhost:5000/api/shared-posts');  
      console.log('Fetched data:', response.data); // <-- ADD THIS LINE  
      const cleanedShares = removeDuplicates(response.data);  
      console.log('Cleaned unique shares:', cleanedShares); // <-- ADD THIS TOO  
      setShares(cleanedShares);  
    } catch (error) {  
      console.error('Error fetching shared posts:', error);  
    }  
  };  
}, []);
```

fetchShares();

}, []);

// Function to remove duplicates using Set


```
const removeDuplicates = (arr) => {  
  const seen = new Set();  
  return arr.filter(share => {  
    const unique = !seen.has(`${share.userId}-${share.postId}`);  
    if (unique) seen.add(`${share.userId}-${share.postId}`);  
    return unique;  
  });  
};
```


THANKS
for your
ATTENTION

Any
questions



UNIVERSITY OF
KARACHI



"Don't be satisfied with stories, how things have gone with others. Unfold your own myth." ~Rumi



UNIVERSITY OF
KARACHI



Department of Compute Science (UBIT Building), Karachi, Pakistan.

1200 Acres (5.2 Km sq.)

53 Departments

19 Institutes

25000 Students



My Homeland Pakistan

