Advanced Concepts: Scope, Closure, Module, Arrow Function

Dr Harshad Prajapati
26 Nov 2023

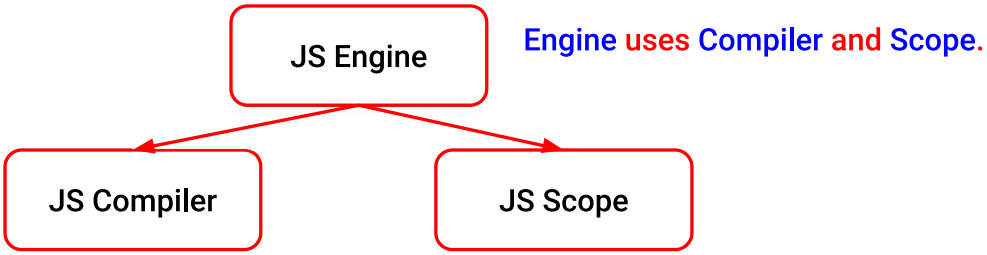
Scope in JavaScript

Scope in Programming Languages and in JavaScript

- Three types of scopes are possible in programming languages:
  - Block scope. (Class, Module, Block)
  - Function scope.
  - Global scope.
- Earlier, JavaScript had:
  - Function scope
  - Global scope
- Later, block scope was added in JavaScript.
- Block scope and Function scope in JavaScript are similar to Java.

How Scope is Handled in JavaScript

- There are three separate components while JavaScript code execution:
  - JS Engine: Responsible for start-to-finish compilation and execution of JS program.
  - JS Compiler: It helps engine. Responsible for parsing and code-generation.
  - JS Scope: It helps engine. It collects and maintains a lookup list of all the declared identifiers.



Declaring Variables in JavaScript

The Old Way of Declaring Variables using var keyword

- JavaScript traditionally used the var keyword for variable declaration.
- Variables declared with var have function scope and can be hoisted.
- Hoisting:
  - Hoisting is a behavior in JavaScript where variable and function declarations are moved, or hoisted, to the top of their containing scope during the compilation phase.
  - This means that we can use a variable or function before it's declared in the code.

Hoisting

- Assignments of values to variable remain in their original place.
- Function declarations are hoisted before variable declarations.

```
console.log(x); // Outputs undefined
var x = 5;
console.log(x); // Outputs 5
```

```
example(); // Outputs "Hello, hoisting!"
no usages
function example() {
    console.log("Hello, hoisting!");
}
```

```
var y;
console.log(y); // Outputs undefined
y = 10;
console.log(y); // Outputs 10
```


js-var-1.html

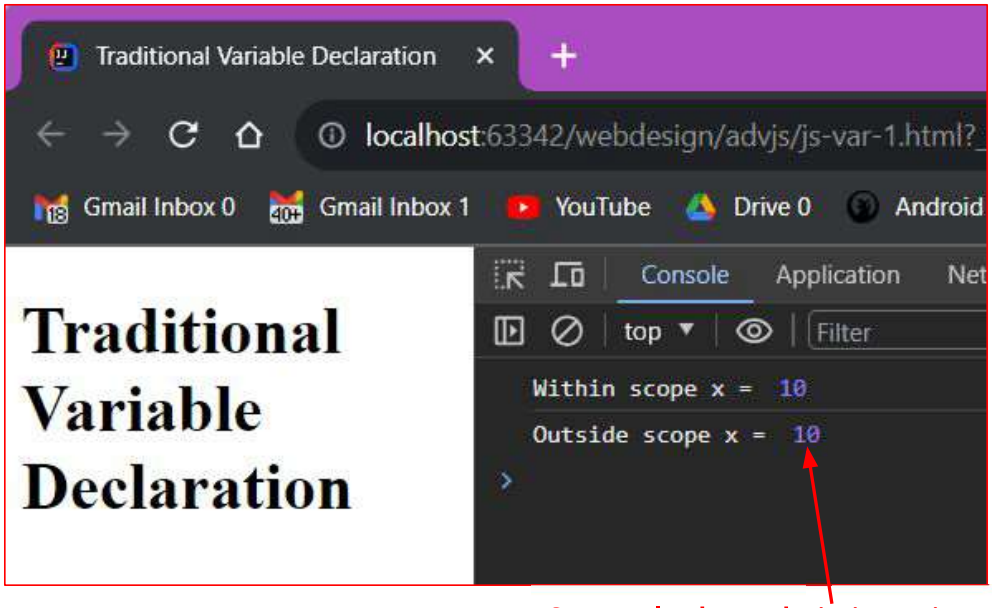
ja-var-1.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Traditional Variable Declaration</title>
</head>
<body>
    <h1>Traditional Variable Declaration</h1>
    <script>
        1 usage
        function example() {
            if (true) {
                var x = 10;
                console.log('Within scope x = ', x); // Outputs 10
            }
            console.log('Outside scope x = ', x); // Outputs 10 due to hoisting
        }
        example();
    </script>
</body>
</html>
```

The var declaration gets hoisted.
That is it goes up at the beginning of the function definition.

Strangely, due to hoisting, var is accessible outside scope also.

Running Application



Strangely, due to hoisting, x is accessible outside scope also.

Introduction of let and const Keywords

- Later two keywords for variable declaration were added:
  - let
  - const
- The let keyword allows to create a variable having block scope.
- The const keyword allows to create a constant having block scope.
  - Constants are immutable variables.

js-var-2.html

ja-var-2.html

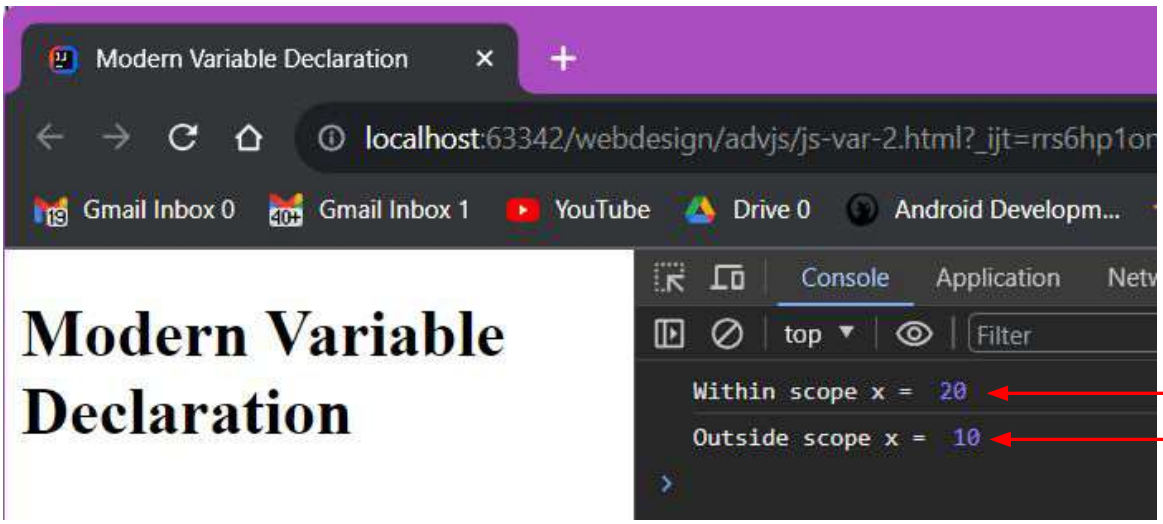
```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Modern Variable Declaration</title>
</head>
<body>
    <h1>Modern Variable Declaration</h1>
    <script>
        let x = 10;
        if (true) {
            let x = 20;
            console.log('Within scope x = ', x); // Outputs 20
        }
        console.log('Outside scope x = ', x); // Outputs 10 (outer scope variable)

        const PI = 3.14;
        // PI = 3.1415; // Error: Assignment to a constant variable
    </script>
</body>
</html>
```

Scope (outer) of variable x declared with let.

Scope (inner) of variable x declared with let.

We cannot change value of PI.
It is constant (immutable)

Running Application



x in inner scope

x in outer scope

JS Modules

What are Modules?

- Modules are a way to organize and structure code in JavaScript.
- They allow developers to break down their code into smaller, manageable pieces.

Why Modules?

- Encapsulation:
  - Modules encapsulate code, preventing variable and function name clashes.
- Reusability:
  - Modules enable code reuse, making it easier to use the same functionality in different parts of an application.
- Maintainability:
  - Code is easier to maintain and understand when organized into modules.

Module Formats

- CommonJS (Node.js):
  - Used in server-side JavaScript.
- ES6 Modules:
  - Native to modern browsers and widely adopted in front-end development.

We will use ES6 in JavaScript and React

Module Definition

- JS modules rely on the import and export statements
- A module in JavaScript is defined as a separate file.
- Defining a Module:
  - In CommonJS:
    - module.exports = { /* vars, objects, functions, etc. */ };
  - In ES6 Modules: ✔
    - export { /* vars, objects, functions, etc. */ };

Module Use

- Importing Modules:
  - CommonJS:
    - const module = require('./module');
  - ES6 Modules: ✔
    - import module from './module';

Export

- We can export a function or a variable from any file using export.
- Export variables individually:
  - ```
    export const name = "Tom";
    export const age = 40;
    ```
- Export all variables at once:
  - ```
    const name = "Tom";
    const age = 40;
    export {name, age};
    ```


Export

- There are two ways to export:
  - named.
  - default.
- We can have only one default export in a file.
  - Export named export:
    - export { myFunction1, myFynction2 } ;
  - Export default export:
    - export default myFunction1;

Import

- We have two ways to import:
  - named.
  - default.
- Import from named export (within curly braces)
  - import { name, age } from "./person.js";
  - The names of exported stuff (name and age) must match with imported names (name and age)
- Import from default export (without curly braces)
  - import message from "./message.js";
  - The name of exported stuff is not required to match with the imported name (message).

Use of modules

- Modules work with http or https protocols.
- We cannot use import / export in a web-page opened via the file:// protocol.
  - For testing, we open a page using live server.
  - Pages will be served using http protocol.

Example: Module

script1.js

```
// script1.js
// We cannot change message (let declaration)
// outside module (let has block (module) scope)
4 usages
let message : string = 'Hello'
2 usages
export default function greet(name) : void {
    console.log(`${message}, ${name}!`);
}

export { message };
```

Default export: While importing in other modules, we can use any other name instead of greet.

Named export: While importing in other modules, we have to use same name message while accessing it.

Irrespective of message is let or const, it cannot be modified in other modules.

script2.js

The exported name is greet; we changed it to greetPerson as it is default export.

```js
// script2.js
// Importing the greet function from script1.js
import greetPerson from './script1.js'; // default exported
import { message } from './script1.js'; // named exported
import * as mod1 from './script1.js'; // named exported

// Using the imported function
greetPerson( name: 'Tom Cruise');
console.log('The message in other module is ' + message);
console.log('The message in other module is ' + mod1.message);
```

Import named export

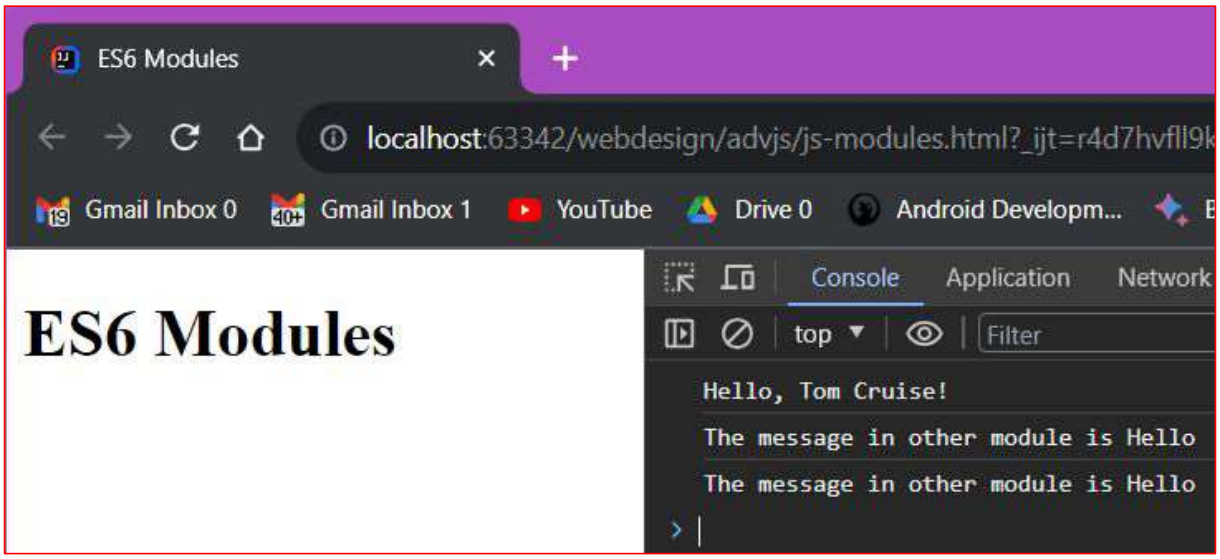
Access named export

js-modules.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport"
          content="width=device-width, initial-scale=1.0">
    <title>ES6 Modules</title>
    <script type="module" src="script2.js"></script>
</head>
<body>
    <h1>ES6 Modules</h1>
</body>
</html>
```

While importing a module, we use type=module

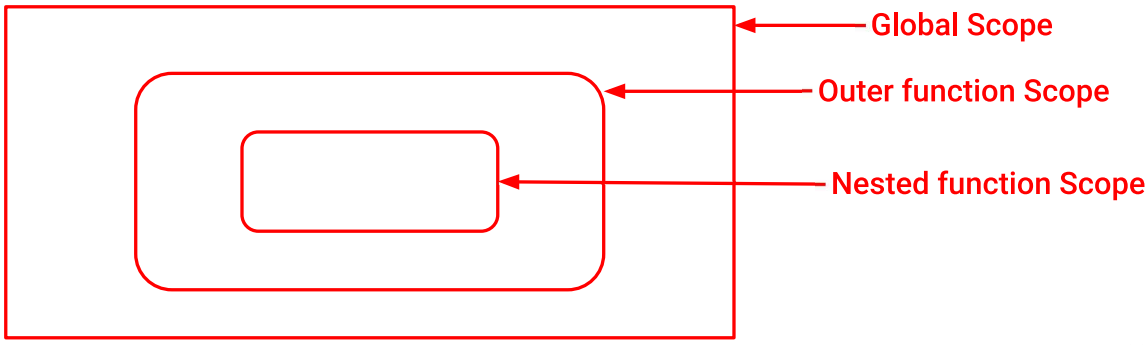
Running Application



Closure

Nested Function Scope

- In JavaScript, we can define a function inside another function.
- If a variable being accessed is not present in inner function,
  - it will be looked-up in outer function;
  - even if it is not present in outer function, it will be looked-up one level up, finally in global scope.



js-closure-1.html

```html
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Nested Function Scope</title>
</head>
<body>
    <h1>Nested Function Scope</h1>
    <script>
        let globalA = 10;
        1 usage
        function outer(){
            let outerA = 20;
            1 usage
            function inner(){
                let innerA = 30;
                console.log("innerA",innerA);
                console.log("outerA",outerA);
                console.log("globalA",globalA);
            }
            inner();
        }
        outer();
    </script>
</body>
</html>
```

A in Global Scope.
Accessible everywhere, in outer function and in inner function

A in outer Scope.
Accessible in itself (outer function) and in inner function.

A in inner Scope.
Accessible in itself (inner function).

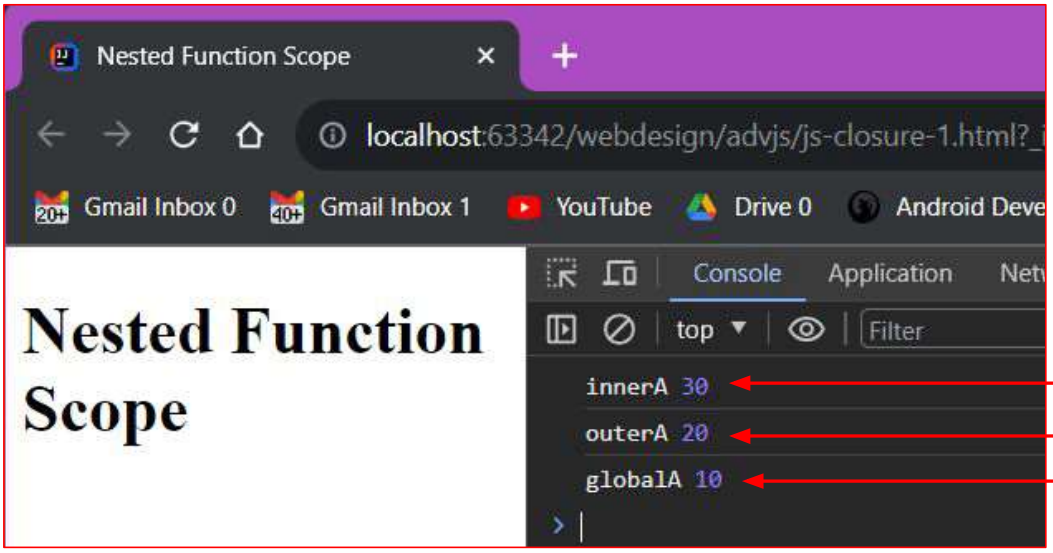
The inner function is not accessible outside outer() function.

In this scope, outer() is accessible, but inner() is not accessible.

Running Application

```
function inner(){
    let innerA = 30;
    console.log("innerA",innerA);
    console.log("outerA",outerA);
    console.log("globalA",globalA);
}
```

The inner() function can access all variables.

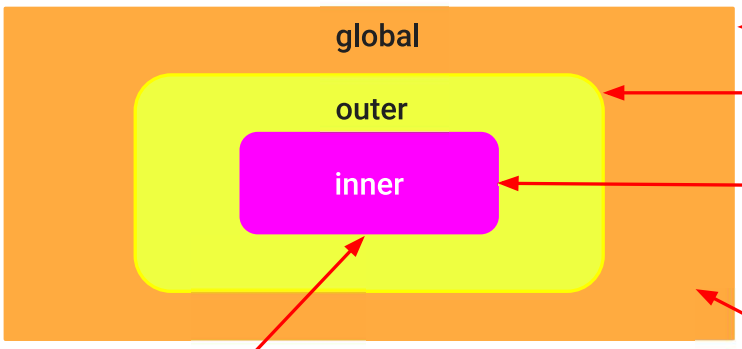


The inner function access its own variable

access variable in outer

access variable in global

Understanding of Scope



Global Scope

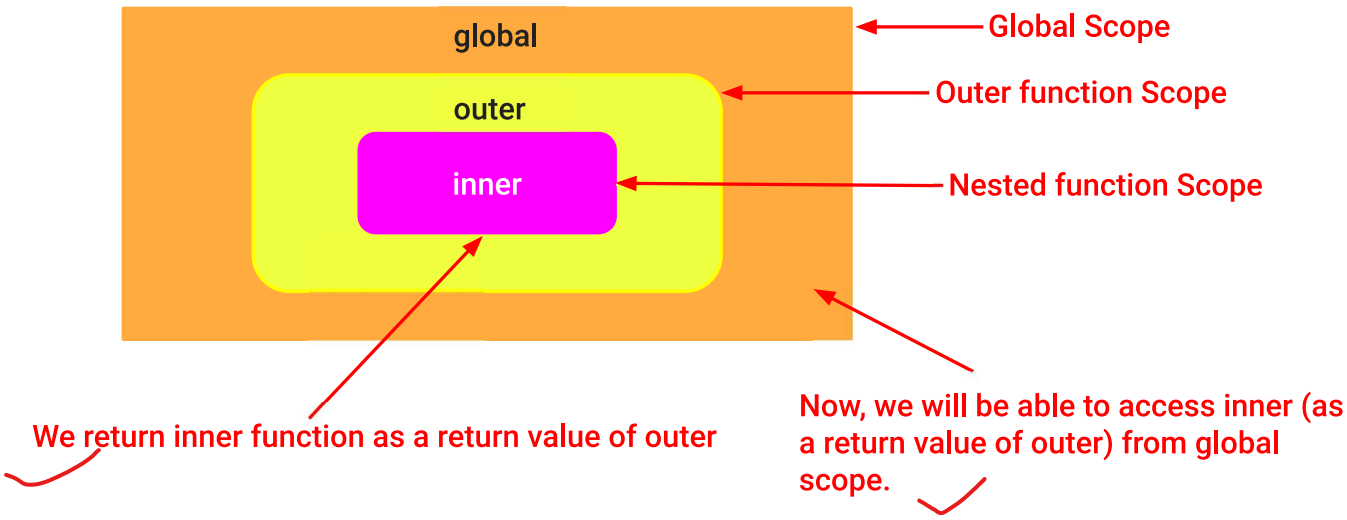
Outer function Scope

Nested function Scope

This inner has lexical scope within outer.
The lexical scope means, the scope in which we can access a defined thing with its name.

In previous example, we could not access inner() in global scope.

Understanding of Closure



js-closure-2.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Closure</title>
</head>
<body>
    <h1>Closure</h1>
    <script>
        1 usage
        function outer(){
            let outerA = 20;
            1 usage
            function inner(){
                let innerA = 30;
                console.log("innerA",innerA);
                console.log("outerA",outerA);
            }
            return inner;
        }
```

Lexical scope of inner is anywhere within outer.

Return inner function from outer (from its lexical scope)

js-closure-2.html

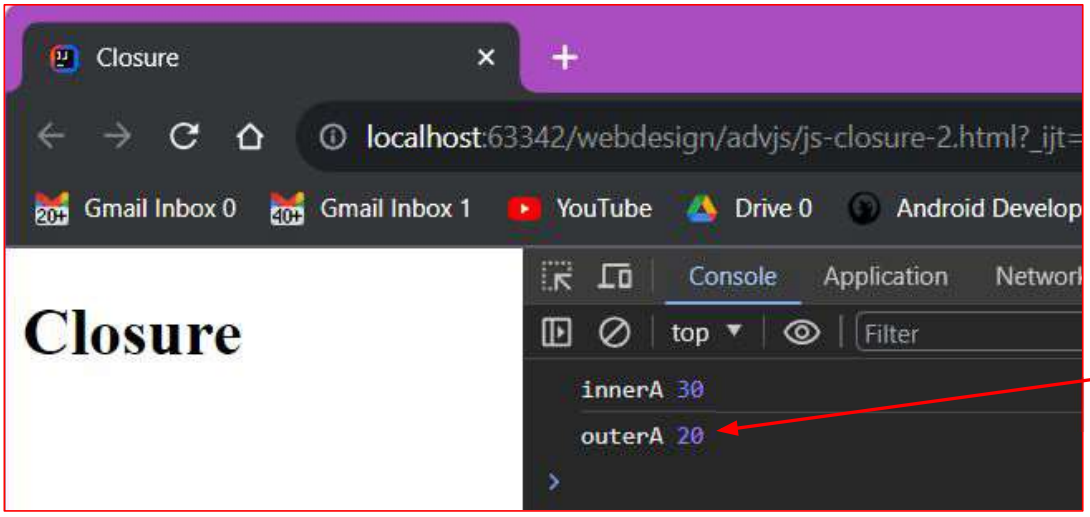
This name could be anything

The inner function has lexical scope anywhere within outer.
But, we access it as the return value of outer.

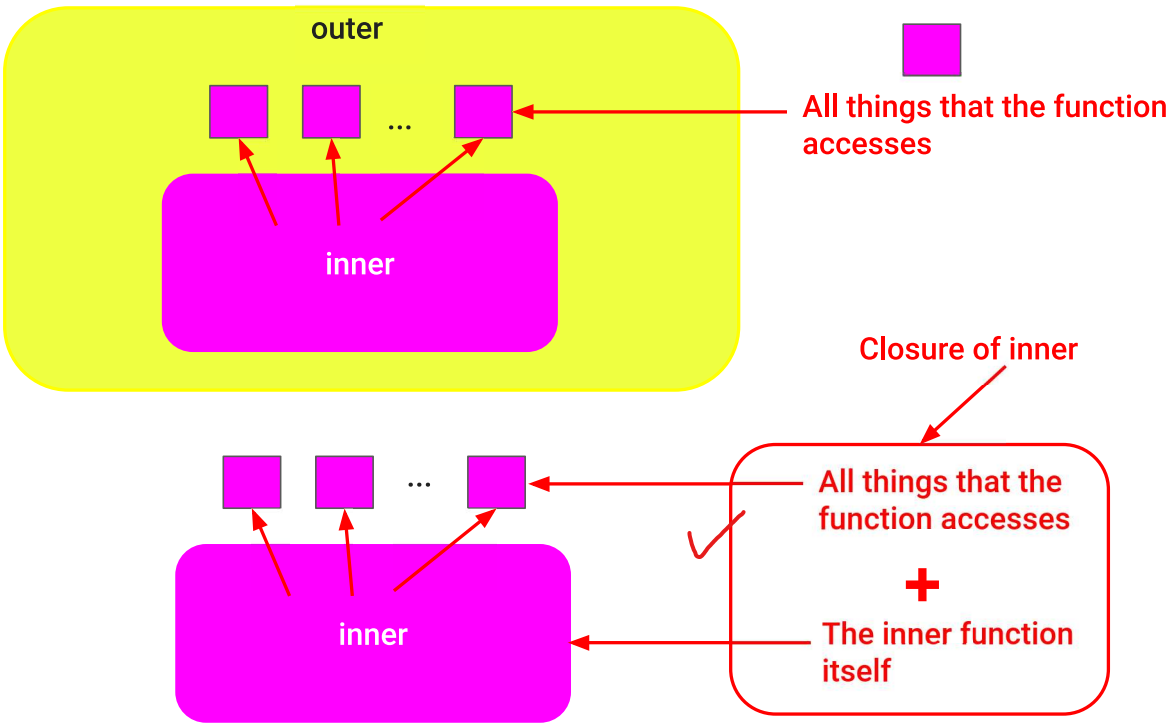
```
19        const inner = outer();
20        inner(); // outerA exists even though
21        // outer() has completed its execution
22    </script>
23 </body>
24 </html>
```

The execution of outer() completed in the previous statement.
But still outerA variable, defined inside outer(), is accessible when we execute inner()

Running Application



The outerA is accessible even though the execution of outer() is completed



If we transport an inner function outside of its lexical scope, the transported function will maintain a scope reference to where it was originally declared.

Closures

- In JavaScript it is possible to **return** a **function** from another function.
- When we return a function from another function, we are effectively **returning** a **combination** of the **function definition** along with the **function's scope**.
  - The **combination** of the **function** and its **scope chain** is called as a **closure**.
- A **closure** is the **combination** of a **function** bundled together with **references** to **surrounding state**.
- **Closures** are **created** every time a **function is created**, at function creation time.
- With closures, the **inner function** has **access** to the **variables** present in the **outer function scope** even after the **outer** function has **finished** executing.

Use of Function Closures

- A closure is a function having access to the parent scope, even after the parent function has closed.
- Global variables can be made local (private) with closures.
- Variables created with var, let, and const inside a function are local variables
  - Otherwise, global variables.
- Variable lifetime:
  - Global variables live until the page is discarded, like when you navigate to another page or close the window.

js-closure-3.html

js-closure-3.html

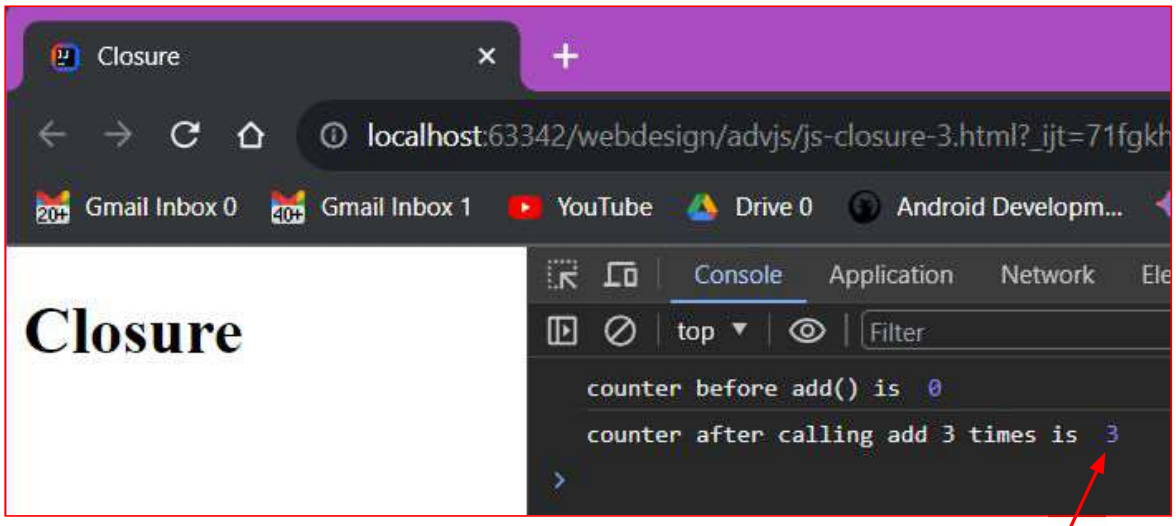
```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Closure</title>
</head>
<body>
    <h1>Closure</h1>
    <script>
        // Initiate counter
        let counter = 0;
        // Function to increment counter
        3 usages
        function add() {
            counter += 1;
        }
```

Counter is global, can be accessed outside add()

js-closure-3.html

```
16          console.log('counter before add() is ', counter);
17          // Call add() 3 times
18          add();
19          add();
20          add();
21          console.log('counter after calling add 3 times is ', counter);
22          // The counter should now be 3
23      </script>
24  </body>
25  </html>
```


Running Application



Counter is global, so its value is preserved

Immediately Invoked Function Expression

- A self-invoking function is IIFE (Immediately Invoked Function Expression).
- We define a function and call it at the same time.
- There are two ways to write IIFE.
  - (function(){ .. }())
  - (function(){ .. })()

js-closure-4.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Closure</title>
</head>
<body>
    <h1>Closure</h1>
    <script>
        const add = (function () {
            let counter = 0;
            return function () {counter += 1; return counter}
        })();
        console.log('counter is ', add());
        console.log('counter is ', add());
        console.log('counter is ', add());
        // the counter is now 3
    </script>
</body>
</html>
```

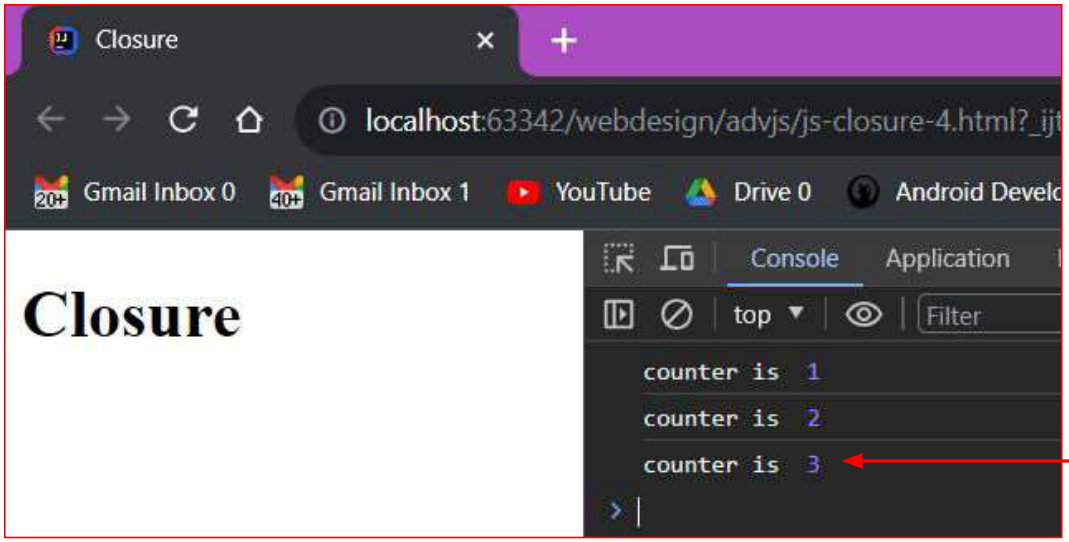
This is an anonymous outer function.

Counter is in the closure of the inner function.
It exists across function calls of add.

We invoke outer function

This is an anonymous inner function accessing variables defined in outer function.

Running Application



Counter is in the closure of the function.
It exists across function calls of add.
So value is 3 after calling add() three times.

Arrow functions and closures

- Arrow functions give access to their defining environment while regular functions give access to their calling environment.
- The value of the this keyword inside a regular function depends on how the function was called.
- The value of the this keyword inside an arrow function depends on where the function was defined.

Destructuring and Spread

Destructuring arrays and objects

- Which operator to use while destructuring?
  - [ ] to destructure array items
  - { } to destructure object properties
- Destructure array items.
  - const [first, second,, fourth] = [10, 20, 30, 40];
- Destructure object properties.
  - const { PI, E, SQRT2 } = Math;

Destructuring in React

- Destructure the useState and useEffect hook functions out of the React's API.
  - const { useState, useEffect } = React;
- The useState() returns an array of two elements.
  - const [state, setState] = useState();

The Rest Syntax

- Use of rest
  - const [first, ...restOfItems] = [10, 20, 30, 40];
  - The first element (10) is assigned to first variable.
  - And the rest elements [20,30,40] are assigned to an array variable restOfItems.

The Spread Syntax

- The spread syntax uses the same 3-dots to shallow-copy an array (array1) or an object (object1) into a new array (array2) or an object (object2).
- Using spread operator in an array:
  - const array2 = [newItem0, ...array1, newItem1, newItem2];
- Using spread operator in an object:
  - const object2 = {
    ...object1,
    newP1: 1,
    newP2: 2,
    };

Arrow Functions in JavaScript

Arrow Functions in JavaScript

- Arrow functions are a concise way to write anonymous functions in JavaScript.
- Introduced in ES6 (ECMAScript 2015) to provide a more concise syntax for writing function expressions.
- Traditional Function:

```
function add(a, b) {
    return a + b;
}
```

- Arrow Function:

```
const add = (a, b) => a + b;
```


Benefits of Arrow Functions:

- Conciseness: Arrow functions have a shorter syntax compared to traditional function expressions.
- Lexical this: Arrow functions don't have their own this context; they inherit it from the enclosing scope.

Features of Arrow Functions

- Implicit return:
  - If the arrow function has only one expression, the return statement is implicit.
  - const double = x => x * 2; ← Arrow function

    Input to arrow function Return from arrow function

JavaScript Array: ES6 functions

- map(): Creates a new array with the results of calling a provided function (lambda) on every element in the array.
- filter(): Creates a new array with all elements that pass the test implemented by the provided function (lambda).
- forEach(): Calls the provided function (lambda) for each element in the array.
- reduce(): Applies a function against an accumulator and each element in the array (from left to right) to reduce it to a single value.
- reduceRight(): Applies a function against an accumulator and each element in the array (from right to left) to reduce it to a single value.

JavaScript Array: ES6 functions

- find(): Returns the first element in the array that satisfies the provided testing function (lambda).
- findIndex(): Returns the index of the first element in the array that satisfies the provided testing function (lambda).
- some(): Checks if at least one element in the array satisfies the provided testing function (lambda).
- every(): Checks if all elements in the array satisfy the provided testing function (lambda).
- includes(): Checks if an array contains a certain element and returns true or false.

JavaScript Array: ES6 functions

- flat(): Creates a new array with all sub-array elements concatenated into it recursively upto a specified depth.
- flatMap(): Maps each element using a mapping function, then flattens the result into a new array.

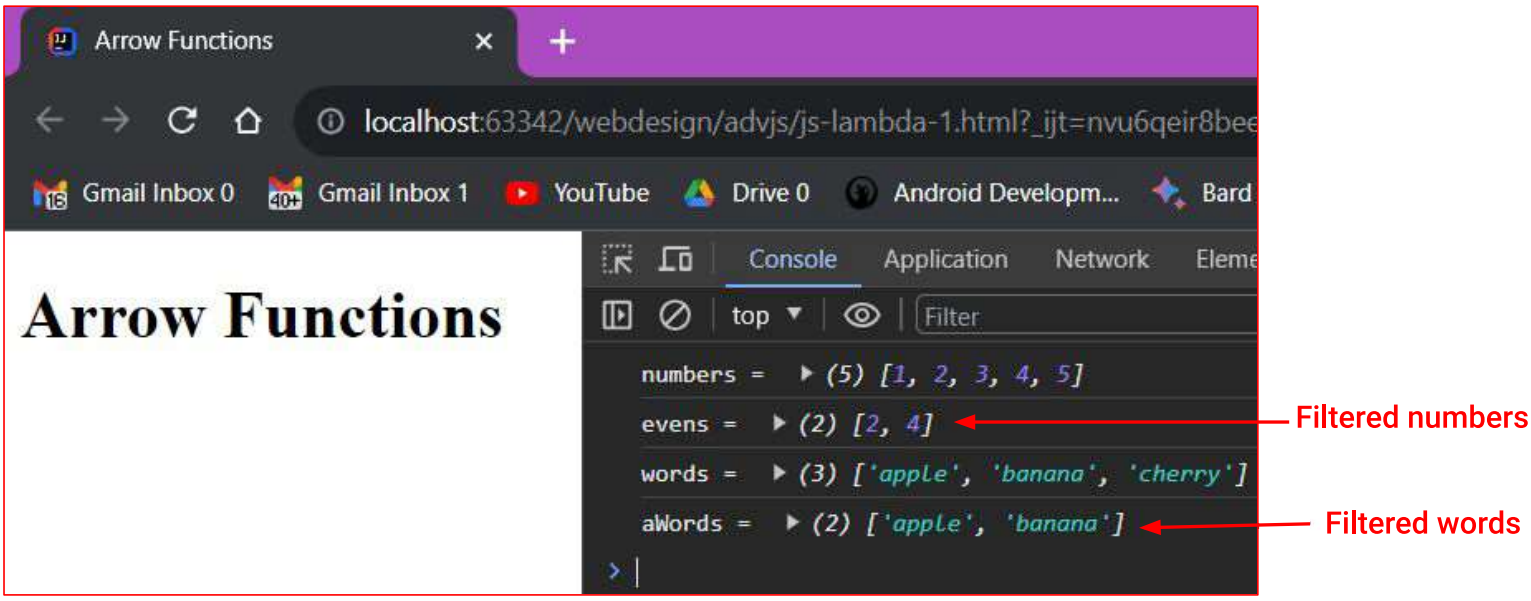
js-lambda-1.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Arrow Functions</title>
</head>
<body>
    <h1>Arrow Functions</h1>
    <script>
        const numbers = [1, 2, 3, 4, 5];
        const evens = numbers.filter(x => x % 2 === 0);
        console.log('evens = ',evens);
        // Result: [2, 4]

        const words = ['apple', 'banana', 'cherry'];
        const aWords = words.filter(word => word.includes('a'));
        console.log('words = ',words);
        // Result: ['apple', 'banana']
    </script>
</body>
</html>
```

Filter even numbers

Filter words that have 'a'

Running Application



js-lambda-2.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Arrow Functions</title>
</head>
<body>
    <h1>Arrow Functions</h1>
    <script>
        const numbers = [1, 2, 3, 4, 5];
        const sum = numbers.reduce((acc, curr) => acc + curr, 0);
        console.log('numbers = ',numbers);
        console.log('sum = ',sum);
        // Result: 15

        const words = ['Hello', ' ', 'World!'];
        const sentence = words.reduce((acc, word) => acc + word, '');
        console.log('words = ',words);
        console.log('sentence = ',sentence);
        // Result: 'Hello World!'
    </script>
</body>
</html>
```

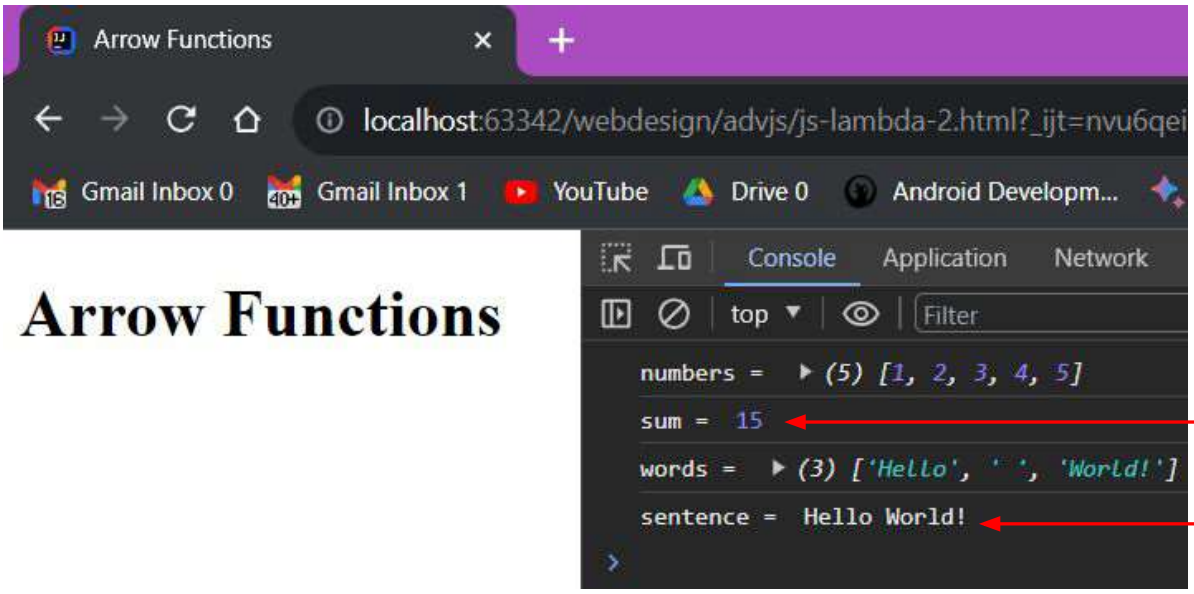
Reduce numbers to sum

Initial value of accumulator

Reduce words to sentence

Initial value of accumulator

Running Application



Reduce numbers to sum

Reduce words to sentence

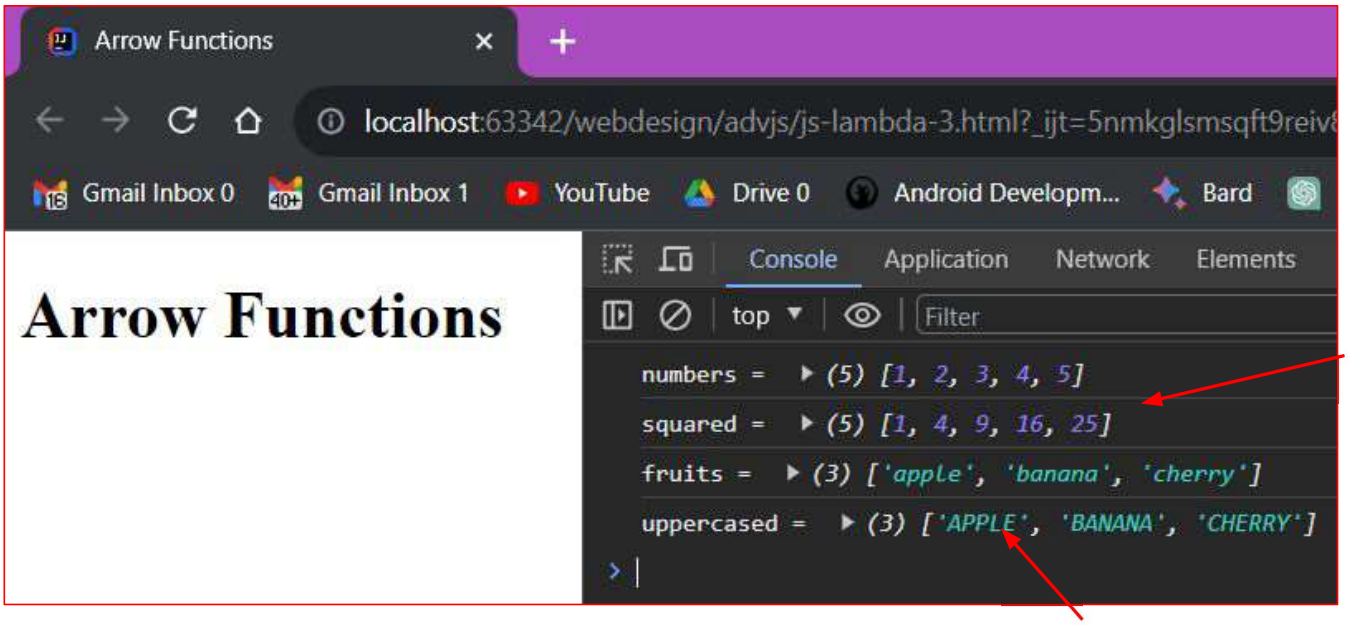
js-lambda-3.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Arrow Functions</title>
</head>
<body>
    <h1>Arrow Functions</h1>
    <script>
        const numbers = [1, 2, 3, 4, 5];
        const squared = numbers.map(x => x * x);
        console.log('numbers = ',numbers);
        console.log('squared = ',squared);
        // Result: [1, 4, 9, 16, 25]

        const fruits = ['apple', 'banana', 'cherry'];
        const uppercased = fruits.map(fruit => fruit.toUpperCase());
        console.log('fruits = ',fruits);
        console.log('uppercased = ',uppercased);
        // Result: ['APPLE', 'BANANA', 'CHERRY']
    </script>
</body>
</html>
```

Map numbers to square

Map words to uppercased

Running Application



numbers mapped to square

Words mapped to uppercased words

js-lambda-4.html

```html
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Arrow Functions</title>
</head>
<body>
    <h1>Arrow Functions</h1>
    <script>
        const numbers = [1, 2, 3, 4, 5];
        numbers.forEach(number => console.log(number));

        const allEven = numbers.every(number => number % 2 === 0);
        console.log('allEven = ',allEven);
```

forEach() element, print it

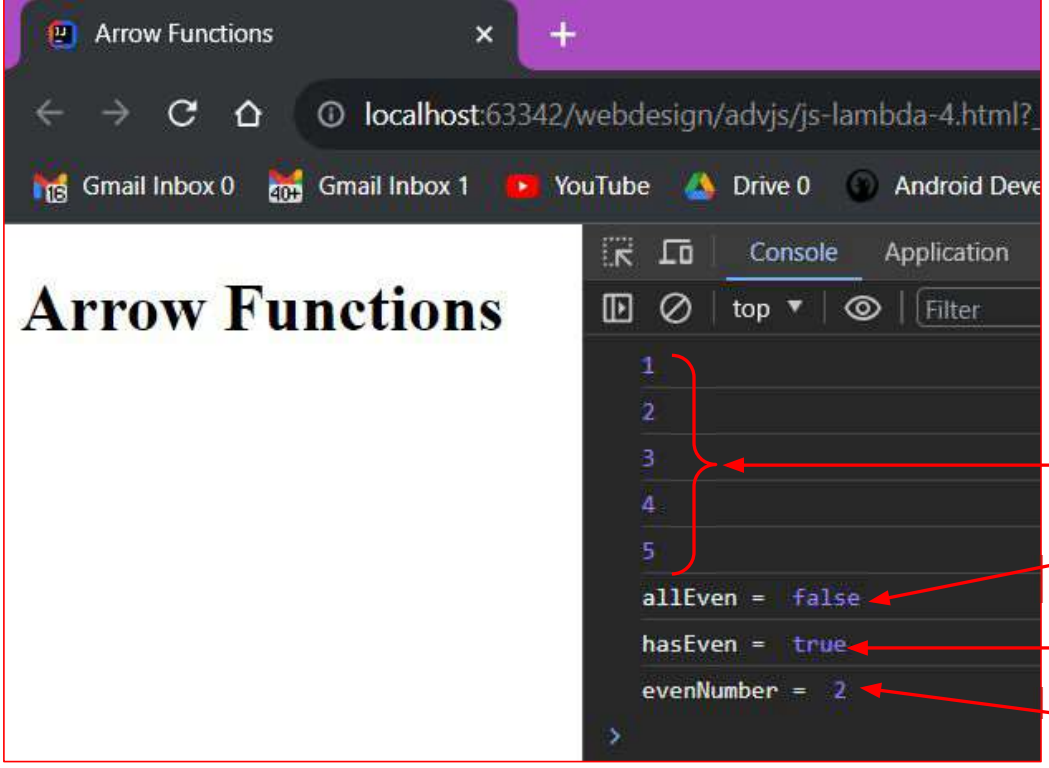
Test all numbers

js-lambda-4.html

```
        const hasEven = numbers.some(number => number % 2 === 0);
        console.log('hasEven = ',hasEven);
        // Result: true

        const evenNumber = numbers.find(number => number % 2 === 0);
        console.log('evenNumber = ',evenNumber);
        // Result: 2

    </script>
</body>
</html>
```

Check any one is satisfying condition

Find any one is satisfying condition



Printed using forEach()

Checked all numbers using every()

Checked any one is even using some()

Find any one even number using find()

js-lambda-5.html

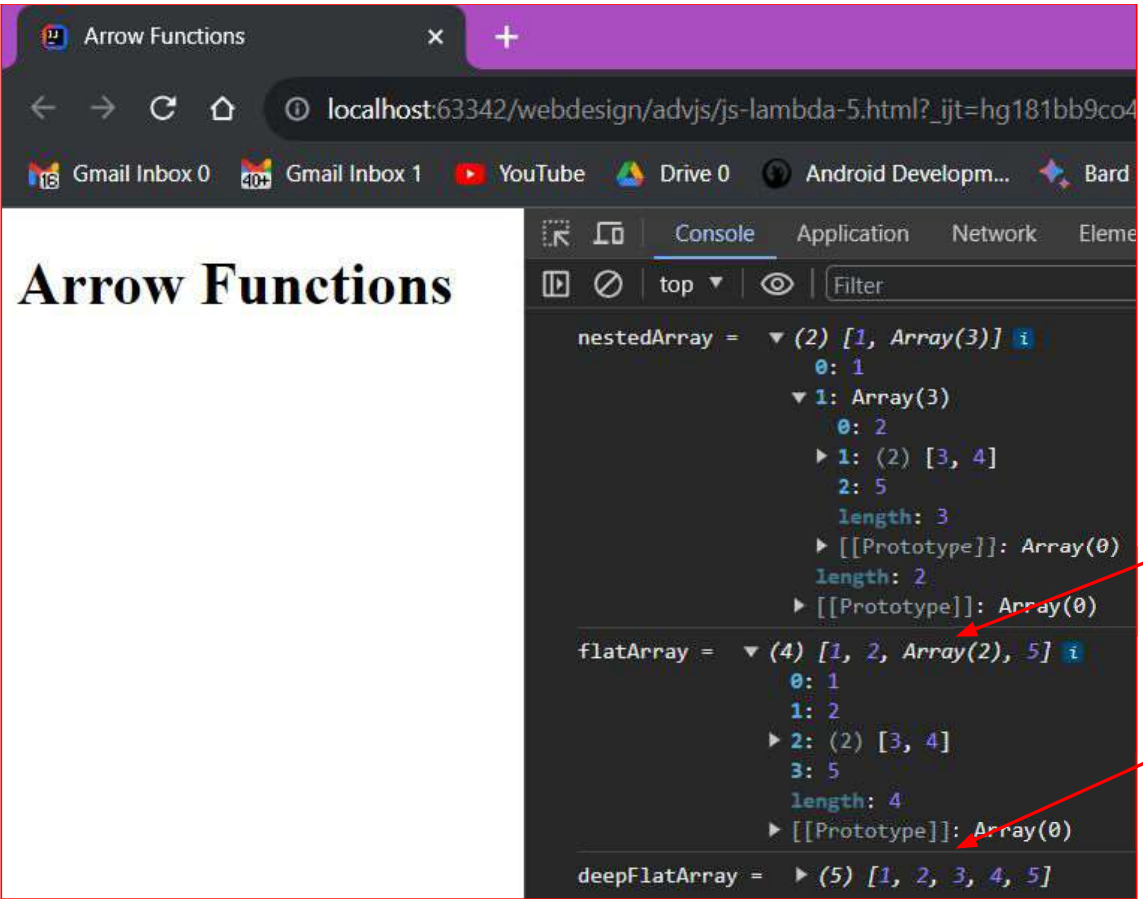
```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Arrow Functions</title>
</head>
<body>
    <h1>Arrow Functions</h1>
    <script>
        const nestedArray = [1, [2, [3, 4], 5]];
        console.log('nestedArray = ',nestedArray);

        // Using flat with default depth (1)
        const flatArray = nestedArray.flat();
        console.log('flatArray = ',flatArray);
        // Result: [1, 2, [3, 4], 5]
```

Flat recursively at depth 1

js-lambda-5.html

Flat recursively at depth 2

```
18        // Using flat with custom depth (2)
19        const deepFlatArray = nestedArray.flat(2);
20        console.log('deepFlatArray = ',deepFlatArray);
21        // Result: [1, 2, 3, 4, 5]
22    </script>
23 </body>
24 </html>
```

Flat recursively at depth 1. Means inner array at depth 1 is unwrapped

Flat recursively at depth 2. Means inner array at depth 2 is unwrapped

References

- https://developer.mozilla.org/en-US/docs/Web/JavaScript
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions