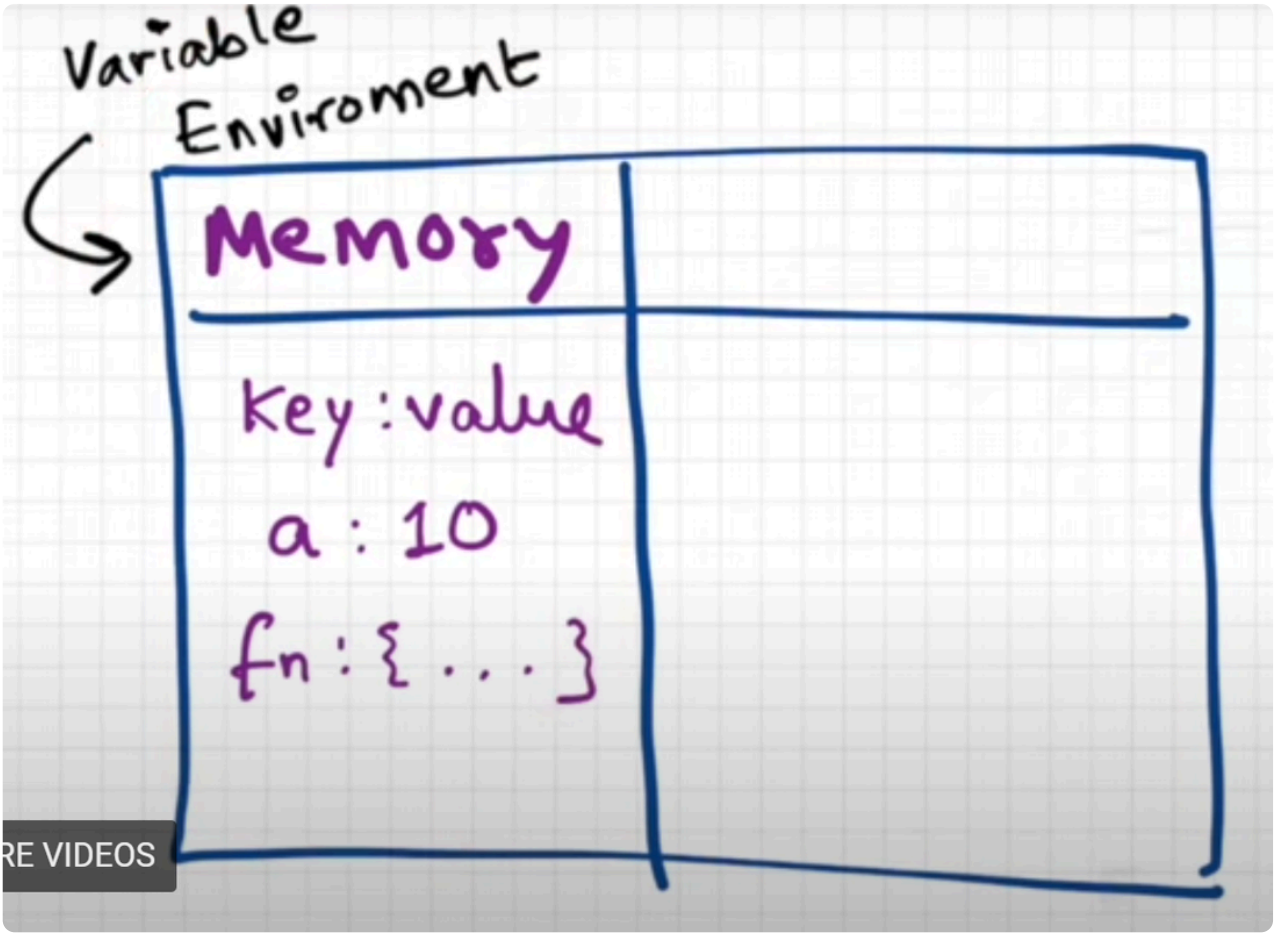
How JavaScript works and Execution Context

Everything in JavaScript happens inside an Execution Context

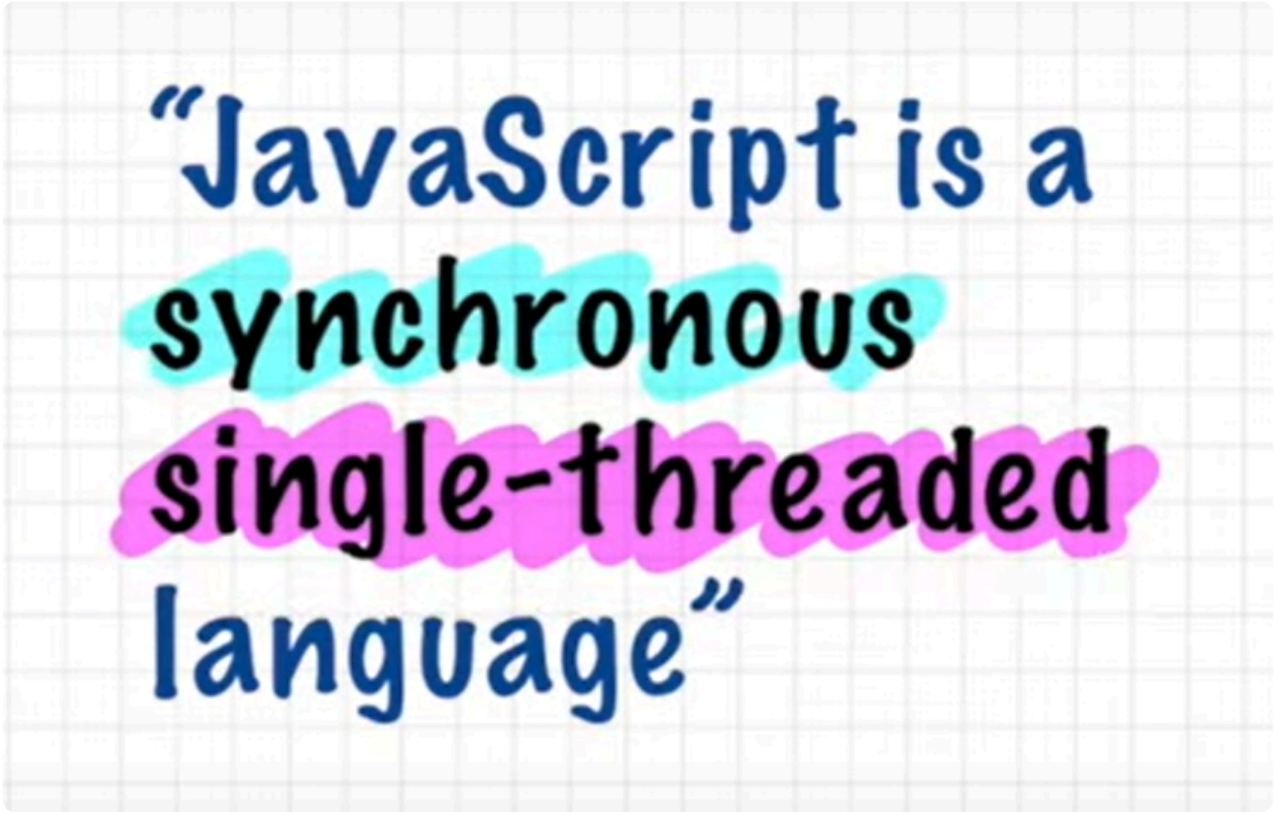
Execution context contains two components.

First components is **memory component**. It's also known as **variable environment**.
This **variable environment** is the place where all the variables and functions will be stored as key value pairs

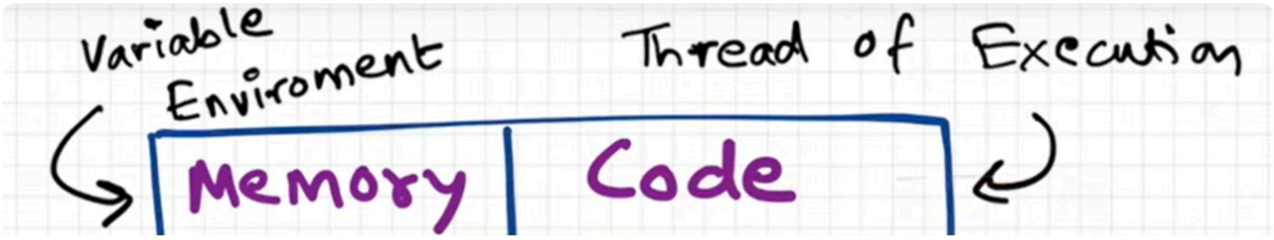


The second component of Execution context is **Code component**. Here is where JavaScript code executes line by line. It's also called as **Thread of Execution**.

> JavaScript is **synchrnonous single-threaded** language



- Single threaded refers to executing one line at at time (neither simultaneousity nor concurrency)
- Synchrnonous nature refers to line-by-line execution from top to bottom. The JS engine execute next line only after current line finished its execution



How JavaScript Code is Executed ?

- When we run a JavaScript program, a execution context is created.

- The creation of Execution Context happens in two phases.
- First phase is **Creation or Memory creation** phase.
- The second phase is **Code Execution** phase

Memory Creation phase

Let's consider the below program,

```
var n = 2;
function square(num){
    var ans = num * num;
    return ans;
}
var square2 = square(n);
var square4 = square(4);
```

- On the creation phase, memory will be created for each variables and functions line-by-line from top to bottom.
- On this memory creation, the variable's name will be stored as key and on the value, there will be a special value `undefined` will be stored for variable. For functions, the entire function will be stored as value

Code Exection phase

- In this second phase, once again JavaScript will run through the code from top to bottom, line by line

- Here is where real execution happens and all the calculations are done

- while executing `var n = 2`, value 2 will be stored in the identifier `n` in the memory component
- The program skips the 2,3,4 and 5th line as there is nothing to execute there.
- Once, it reaches the 6th line, here comes an function invocation. Functions are like mini programs in JavaScript.
- For executing a function, new execution context will be created inside the code component of parent execution context.
- Same two phases applicable here to create this mini execution context.
- In the first phase (creation phase), variables such as `num` (parameter), `ans` are allocated memory as a key-value pair with `undefined` stored as value.
- In second phase (code execution phase), As `n` (argument) is passed to the `num` (parameter), `num` got asssigned as 2 in the memory component overwriting the existing `undefined`
- After calculating `num * num` within code component, the resultant value 4 got assigned to `ans` variable in the memory component.
- `return` keywords states that, current context is over and we need to return the control again to the place where the function was invoked (parent execution context).
- When the `return` is encountered, it finds the `ans` from the local memory and it is returned to line no.6 and also the control goes back to the parent execution context. Now, the `undefined` in `square2` got overwritten with 4.

- As soon as the value is returned, the execution context for this `square` function will be deleted from the code component of parent execution context
- Now, the control comes to 7th line. Same happens here, a new execution context will be created and go through the two phase and finally returns valu 16.
- Now, `square4` 's value will be overwritten with 16

- After completion of entire program, the whole global execution context will be deleted

- Creation, deletion and management of this execution context will be managed by call stack in JavaScript
- Whenever a the program starts, Global Execution Context will be pushed onto the Call stack. When a function call happens, a new execution context will be created and pushed onto the stack
- When the function call returns, execution context will be popped out of the stack and will be deleted.
- Thus, *"Call Stack maintains the order of execution of execution contexts"*

Other names of call stack are:

- Call Stack
- Execution Context Stack
- Program Stack
- Control Stack
- Runtime Stack
- Machine Stack

Hoisting in JavaScript

> In interview context, **Hoisting** in JavaScript is a mechanism where variable and function declarations are moved to the top of their containing scope during the compilation phase, before the code is executed. This means you can use variables or functions before they are declared in the code.

In simple terms, hoisting refers to the process of creating memory for the variable during **memory creation phase**.

Hoisting in JavaScript is a mechanism where **variable and function declarations** are moved to the top of their containing scope during the compilation phase, before the code is executed. This means you can use variables or functions before they are declared in the code.

Key Points About Hoisting:

1. **Declarations are hoisted, not initializations**:
    - Variables declared with `var` are hoisted, but their initialization remains in place.
    - Variables declared with `let` and `const` are also hoisted but remain uninitialized (in a **temporal dead zone**).
    - Function declarations are fully hoisted, meaning you can invoke them before they appear in the code.

2. **Function expressions are not hoisted**:
    - Only the variable declaration is hoisted, not the function assigned to it.

Example with `var`:

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

- **Explanation**: The declaration `var x` is hoisted to the top, but the assignment `x = 5` is not.

Example with `let` and `const`:

```
console.log(y); // ReferenceError: Cannot access 'y' before initialization
let y = 10;
console.log(y); // Output: 10
```

- **Explanation**: The `let` declaration is hoisted, but it remains in a **temporal dead zone** until the line where it is declared is reached.

Undefined and not defined aren't the same

When the variable is not even defined, it'll be not defined. If it's defined later in the program and we trying to access them earlier than it was declared, we will get `undefined`.

We will get `undefined` throughout the program, if the variable is declared / defined but not initialised. Ex: `var a;` . Here, variable will be declared but not initialised. So throughout the program, the value stored in `a` is `undefined` .

```
var a = 10;
console.log(x); // ReferenceError: x is not defined
```

Example with Functions:

Function Declaration:

```
greet(); // Output: Hello, World!
function greet() {
  console.log("Hello, World!");
}
```

- **Explanation**: The entire function declaration is hoisted.

Function Expression:

```
hello(); // TypeError: hello is not a function
var hello = function () {
  console.log("Hi there!");
};
```

- **Explanation**: Only the declaration `var hello` is hoisted, but not the function assignment.
- On function expression, we can use named functions, anonymous function (mentioned above) and arrow functions as well.
- The main difference between named (or) anonymous function and arrow function is that, arrow functions lack `[[Constructor]]` internal method. It doesn't behave as constructor function. Also, arrow function do not have their own `this` .

Summary:

- Variables declared with `var` are hoisted but initialized as `undefined` .

- Variables declared with `let` and `const` are hoisted but remain uninitialized until the declaration line.
- Function declarations are fully hoisted.
- Function expressions are partially hoisted (only the variable name is hoisted).

Why JavaScript initialise those variables as undefined ?

> JavaScript has some primitive data types. If the data types are known, we can assign some default values such as 0 for `integers`, `null` for objects, `false` for boolean and so on. Without even knowing the data type, JavaScript itself won't assign a default value. Also, the language is dynamic language that doesn't support explicit mentioning of data type.
> Thus, a special data type called `undefined` is created and the value `undefined` is assigned to the variable of type `undefined` until it is explicitly initialised.

JavaScript is actually a *loosely typed* or *weekly typed* language. It means a variable can hold data of any type in javaScript. With same variable, we can store, Numbers, boolean, strings in different parts of the program.

From a JavaScript perspective, `undefined` is used in **three primary cases**, but there are subtle nuances. Let's explore them all:

The Three Primary Uses of `undefined` :

1. Default Value for Uninitialized Variables

When a variable is declared but not assigned a value, JavaScript automatically assigns it the value `undefined`.

Example:

```
let a;       // Declared but not initialized
console.log(a);  // undefined
```

- **Why**: This happens because memory is allocated for the variable during the execution context creation, but no value is explicitly assigned, so JavaScript defaults to `undefined`.

2. Function Parameters Without Arguments

If you call a function but do not pass an argument for one of its parameters, JavaScript assigns `undefined` to that parameter.

Example:

```
function greet(name) {
  console.log(name);  // undefined (if no argument is passed)
}

greet();  // No argument passed
```

- **Why**: JavaScript allocates memory for each parameter during the function's execution context creation, but if no argument is passed, the parameter remains `undefined`.

3. Accessing Non-Existent Object Properties

If you try to access a property of an object that does not exist, JavaScript will return `undefined`.

Example:

```
const person = { name: "John" };
console.log(person.age);  // undefined (no `age` property exists)
```

- **Why**: JavaScript checks the object for the requested property. If it doesn’t exist, it returns `undefined` without creating new memory for that property.

Other Contexts Where `undefined` Appears (Edge Cases):

While the above three are the most common cases, here are additional scenarios where `undefined` appears:

4. The `return` Value of Functions Without Explicit `return`

If a function doesn’t explicitly return a value, it implicitly returns `undefined`.

Example:

```
function sayHello() {
  console.log("Hello!");
}

const result = sayHello();
console.log(result);  // undefined
```

- **Why**: If no value is explicitly returned, JavaScript automatically assigns `undefined` as the return value.

5. Explicit Assignment of `undefined`

You can explicitly assign `undefined` to a variable or object property, though this is generally discouraged.

Example:

```
let x = undefined;  // Explicitly assigned
console.log(x);  // undefined
```

- **Why**: Even though you can assign `undefined` explicitly, it's better to use `null` if you intend to indicate "no value" explicitly.

6. `void` Operator

The `void` operator explicitly evaluates an expression and returns `undefined`.

Example:

```
console.log(void 0);  // undefined
```

- **Why**: `void` is used to ensure that an expression evaluates to `undefined`, regardless of its actual value.

7. Destructuring with Missing Properties

When destructuring an object or array, if the property or index doesn’t exist, the value defaults to `undefined`.

Example:

```
const { age } = { name: "John" };
console.log(age);  // undefined
```

- **Why**: The destructuring operation looks for `age` in the object, but since it doesn't exist, `undefined` is used.

Summary:

In JavaScript, `undefined` commonly appears in these cases:

1. **Uninitialized variables**.
2. **Function parameters without passed arguments**.
3. **Accessing non-existent object properties**.

Additionally, it can appear in edge cases like:

- Functions with no `return`.
- Explicit assignment.
- Using the `void` operator.
- Missing values in destructuring.

Apart from these, there are other places where `undefined` is used by JavaScript to handle some other cases as well. It's a big list that's not required to build up a basic foundation. If need to do so, there is nothing wrong in exploring them.

> Also, `undefined` is there in JavaScript for such specific context. So, in my point of view, explicitly assigning `undefined` to any variable is not a convention, it's not a good practise, it can lead to inconsistencies, and also, it's a **programming crime**.
> — Danusshkumar

How function works in JavaScript

Whenever we try to access a variable, the JavaScript do lookup on the current execution context where we try to access the variable. Once found, it will use that variables value while we do access to do some operations

```
var x = 1;

a();
b();
console.log(x);

function a() {
  var x = 10;
  console.log(x);
}

function b() {
  var x = 100;
  console.log(x);
}
/* Console logs:
 * 10
 * 100
 * 1
 * /
```

Shortest JS Program

The `window` **object** in JavaScript is a fundamental part of the web browser's JavaScript environment. It's created by the browser and serves as the **global object** for client-side JavaScript. Let's dive into its purpose, creation, and relationship with the **global execution context**.

1. What Is the `window` Object?

- The `window` object represents the **browser window** or **global scope** in a web environment.
- It provides methods, properties, and objects that are essential for interacting with the browser (e.g., managing the DOM, handling events, navigating to URLs, etc.).

2. Why Is the `window` Object Created?

- The `window` object is created by the browser when a web page loads to act as the **execution environment** for JavaScript code.
- It ensures a consistent interface to interact with the web page and browser.

- It provides access to:
  - The **DOM** (`document` , `body` , etc.).
  - Browser-specific APIs (`localStorage` , `alert()` , etc.).
  - Utility functions (`setTimeout` , `clearInterval` , etc.).

3. Link Between `window` and Global Execution Context

a. Global Execution Context

When a JavaScript program starts executing:

1. A **global execution context** is created by the JavaScript engine.
2. The global execution context consists of:
   - **Global Scope**: Contains all globally declared variables and functions.
   - **Global Object**: In a browser, this is the `window` object.
   - `this` **Keyword**: In the global execution context, `this` points to the global object (i.e., `window` in a browser).

b. `window` Is the Global Object

- All global variables and functions declared using `var` are properties of the `window` object.

```
var x = 10;
console.log(window.x); // 10
```

- In the global scope, the `this` keyword refers to `window` .

```
console.log(this === window); // true
```

c. Execution Flow

- When JavaScript code runs:
  1. The `window` object is created (by the browser).
  2. The global execution context is initialized, linking `this` to the `window` object.
  3. Global variables and functions become properties of the `window` object (if declared with `var`).

4. Key Features of the `window` Object

Feature	Description
DOM Access	`window.document` gives access to the DOM.
Global Variables/Funcs	Variables/functions declared with `var` become properties/methods of `window`.
Browser APIs	Provides APIs like `localStorage`, `fetch`, `alert`, `setTimeout`, etc.
Window Management	Functions like `open()`, `close()`, `resizeTo()`, etc., for managing browser windows.
Events	Handles events like `onload`, `onresize`, `onscroll`, etc.

5. `window` Object vs Global Variables

- `var` **Variables**: Added to `window`.

```
var foo = "Hello";
console.log(window.foo); // "Hello"
```

- `let` **and** `const` **Variables**: Not added to `window`.

```
let bar = "World";
console.log(window.bar); // undefined
```

6. Non-Browser Environments

In environments like Node.js, the global object is called `global`, not `window`.

- `global` serves the same purpose: managing the global scope and providing global utilities.
- There is no `window` object in Node.js because it doesn't have a browser context.

Summary

1. The `window` **object** is the global object in browser-based JavaScript.
2. It is created by the browser as the root of the global execution context.
3. All global variables (`var`) and functions are properties of `window` .
4. The `this` keyword in the global scope points to the `window` object.
5. It provides essential methods, properties, and APIs for interacting with the browser.

The Scope Chain, Scope & Lexical Environment

1. Lexical Environment & Its Data Structure:

- **Lexical Environment (LE)** refers to the environment in which variables and functions are declared, and it defines the scope in which the code is executed.
- Every **execution context** (global, function, etc.) has its own **lexical environment**.
- The **structure of a lexical environment** includes:
    1. **Environment Record (ER)**: Stores the variable bindings (e.g., `var` , `let` , `const` variables, functions, parameters).
    2. **Reference to the Outer Lexical Environment**: This creates the **scope chain**. Each lexical environment has a reference to its parent (or outer) lexical environment.

Example of Lexical Environment:

```
function outer() {
  let outerVar = "I'm from outer!";
  function inner() {
    let innerVar = "I'm from inner!";
    console.log(outerVar);  // Accessed via scope chain
  }
  inner();
}

outer();
```

Here:

- The `outer` function has a lexical environment with the `outerVar` variable.
- The `inner` function has its own lexical environment with the `innerVar` variable and a reference to the `outer` function's lexical environment.
- This creates a **scope chain** where `inner` can access variables from `outer`.

2. Relationship with `window` (Global Object):

- The **global object** (`window` in browsers) is part of the **global lexical environment**.
  - **Global Execution Context (GEC)** is created when the script runs in the global scope.
  - The **global lexical environment** holds references to the `window` object (or the global object in non-browser environments).
  - All global variables and functions are properties of the **global object**.

Example:

```
var globalVar = "I'm global";
console.log(window.globalVar);  // Accessed via the global object
```

- `window` is **not the same as the lexical environment**. It's an object that gives access to global variables and functions.
- The **lexical environment** only stores variable/function declarations and references to outer scopes, not the global object itself.

3. Lexical Environment & Memory Component:

- The **memory component** of an execution context refers to the **Environment Record (ER)**, which stores all the variables, functions, and parameters in memory.
- **Lexical Environment** essentially includes the memory component, as it holds the variable bindings and references to outer environments (for scope resolution).
- So, **Lexical Environment** and **Memory Component** can be viewed as essentially **the same thing**, but **Lexical Environment** has an additional reference to its outer environment.

Key Notes:

- **Execution Context**:
  - **Global Execution Context** has a global lexical environment (which contains `window` /global object).
  - **Function Execution Context** has a function-specific lexical environment, holding local variables and a reference to the outer lexical environment.

4. Indirect Access to Ancestor Lexical Environments:

- Lexical environments are **connected in a chain**. Each environment refers to its **immediate outer lexical environment** (parent scope).
- This chain enables **access to variables** from outer scopes indirectly.

Example of Scope Chain:

```
let a = 10;

function outer() {
  let b = 20;

  function inner() {
    let c = 30;
    console.log(a, b, c); // Accesses `a` from the global scope, `b` from
`outer`, `c` from itself
  }
  inner();
}

outer();
```

Here:

- **Global Lexical Environment** contains `a`.
- **Outer Lexical Environment** contains `b` and references the global environment.
- **Inner Lexical Environment** contains `c` and references the `outer` environment.

Thus, the scope chain gives indirect access to variables from ancestor environments.

5. Constructor Function's Object Creation:

- **Constructor functions** (when invoked with the `new` keyword) create a **new object**.
- The new object is stored **outside** the constructor's lexical environment but is **linked to the constructor function** via the `this` keyword.
  - **`this`** refers to the new object within the constructor.
  - The object is **created in heap memory** and not directly in the lexical environment.

Example:

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}

const person1 = new Person("Alice", 30); // `person1` is an object, created
in heap memory
```

- The new object (`person1`) is **not part of the lexical environment** of `Person` .
- The **lexical environment** of the `Person` constructor holds the function's parameters (`name` , `age`) and other local variables, but **not** the new object.

6. Functions & Objects:

- **Every function** has its own **execution context** and therefore its own **lexical environment**.
  - When the function is invoked, a new **execution context** is created.
  - The function's **lexical environment** contains references to variables defined inside it and a reference to its parent lexical environment.
  - **Objects** (created using constructors or other methods) are **not stored directly inside the lexical environment** of the function. Instead, they are created in heap memory and accessed through `this` .

7. Key Takeaways:

Topic	Explanation
Lexical Environment (LE)	Refers to the environment where variables and functions are declared. It consists of the **Environment Record** (variable bindings) and a reference to its parent (outer) environment.
Global Object (`window`)	The global object (`window` in browsers) is part of the **global lexical environment**. It provides access to global variables and functions.
Lexical Environment vs. Memory	The **lexical environment** is the **memory component** of the execution context. It stores variable/function declarations and references to outer environments.
Access to Ancestor Lexical Environments	Each lexical environment has a reference to its immediate outer environment, forming a **scope chain** that allows access to outer variables.
Constructor Function Objects	Objects created via constructor functions are **not stored in the constructor's lexical environment** but in **heap memory**. They are linked to `this` .
Functions and Execution Contexts	Every function has its own execution context, which includes its own lexical environment and a reference to its parent environment. Objects created inside functions are stored separately.

8. Final Note:

- **Lexical Environment** is essentially where the memory for variable bindings and function declarations is created.
- **Objects** (created by constructors or manually) are stored in **heap memory** and are **not directly tied** to the lexical environment but are **referenced via** `this` .

Lexical Environment and Memory Component: Are They the Same?

- While **lexical environment** and **memory component** are closely related, **they are not exactly the same**. They work together, but they represent different concepts:
    - **Lexical Environment**:
        - It's a **conceptual data structure** used to represent the **environment** for resolving names (variables, functions).
        - It defines the **scope** in which the code is executed and provides access to variables/functions in the scope chain.
        - It **includes** the environment record and references to outer environments.
    - **Memory Component**:
        - Refers to the **actual physical storage** allocated for variables, functions, and parameters.
        - When an execution context is created, memory is allocated for variables, and the environment record is populated.
        - It stores values (like primitive values or references to objects) and handles memory management.

Global Object & Global Execution Context

1. **Global Object & GEC**:
    - When the **Global Execution Context (GEC)** is created, the **global object** (like `window` in browsers) is **created by the JavaScript runtime** as well.
    - The **global object** itself is **not stored directly in the GEC's memory component**. Instead, it’s part of the **runtime environment** that provides access to global variables and functions.
2. **Global Variables in Memory Context**:
    - Inside the **memory component** of the **global execution context**, **global variables and functions** are stored (e.g., variables declared with `var`, functions declared with `function`).

    - These **global variables** are **stored in the memory component** of the **global execution context** and are **also accessible as properties of the global object** (e.g., `window.globalVar` in the browser).

3. **Global Object as an Access Point**:
    - **The global object** is **linked to the global execution context** but does not reside directly in its memory.
    - The **global object** essentially acts as the **access point** for global variables and functions, which are stored in the **memory component** of the **global execution context**.

- When a function `b` is defined inside `a`, it's like `b` is sitting inside lexical environment of `a`. In other words, `b` is lexically present inside `a`. So, the lexical parent of `b` is `a`.
- The caller environment may be different from a functions' lexical environment.
- Reference to parent's lexical environment is also stored in memory component of execution context only.
- Global lexical environment's parent reference is `null`.
- Scope chaining is nothing but the chaining of this lexical environment.
- We can visualise them in browser. For a execution context, when it is clicked, we can visually view it's lexical environment in the tab called "Scope". There, it's chained (visually it's not chained in browser, but placed one after other as list)

Five common scopes in JavaScript

1. Lexical Scope (also known as Static Scope):

- **Definition**: In **lexical scoping**, the scope of a variable is determined by its **location in the source code**, i.e., where the function or block is defined, not where it is called.
- **How it works**: JavaScript functions are lexically scoped. This means that a function can access variables in its **own scope**, its **parent scope**, and all the way up to the **global scope** based on the location in the code.

- **Example**:

```
function outer() {
    var outerVar = "I'm in the outer function";

    function inner() {
        console.log(outerVar);  // Accesses outerVar because of lexical
scoping
    }

    inner();
}

outer();
```

  - Here, the **inner** function can access `outerVar` because of lexical scoping. The scope is determined by the position of the functions in the code, not where they are called.

2. Dynamic Scope:

- **Definition**: In **dynamic scoping**, the scope of a variable is determined by the **call stack**—specifically, the **order in which functions are called** at runtime, not by where they are written in the source code.
- **How it works**: In languages that use dynamic scoping (such as older versions of JavaScript before ES6), a function would access the variables from the most recent function call, even if that variable is not in the lexical scope of the function.
- **Note**: **JavaScript does not use dynamic scoping**; it always uses lexical scoping. However, it's useful to understand the difference in languages that use dynamic scoping.
- **Example in Dynamic Scope (theoretical)**:

```
var x = "global x";

function first() {
    console.log(x);  // Looks for x in the dynamic call stack
}

function second() {
    var x = "local x";
    first();  // first() will access the dynamic scope
}

second();  // This would log "local x" if JavaScript used dynamic
scoping
```

- In a dynamically scoped language, `first()` would log `"local x"` because it looks up the stack and finds the `x` declared in `second()`, not the global `x`.

3. Block Scope (Introduced with ES6):

- **Definition**: **Block scoping** restricts the scope of variables to the block in which they are defined. In contrast to lexical scoping, variables declared using `let` or `const` are scoped to the nearest enclosing block, which could be an `if`, `for`, or any other block.
- **How it works**: Variables declared with `let` or `const` inside a block (such as a loop or conditional) are only accessible within that block, and not outside it.
- **Example**:

```
if (true) {
    let blockVar = "I'm inside a block";
    console.log(blockVar);  // Works fine inside the block
}
console.log(blockVar);  // ReferenceError: blockVar is not defined
```

4. Global Scope:

- **Definition**: The **global scope** refers to variables and functions that are accessible from anywhere in the JavaScript program.
- **How it works**: Variables and functions declared outside of any functions are in the global scope. In a browser, the global scope is represented by the `window` object. In Node.js, it's the `global` object.

- **Example**:

```
var globalVar = "I'm in the global scope";
function test() {
    console.log(globalVar);  // Accessible inside the function
}

test();
console.log(globalVar);  // Accessible outside the function as well
```

5. Function Scope:

- **Definition**: **Function scope** means that variables declared inside a function are only accessible within that function. This scope is typically used when declaring variables with `var`.
- **How it works**: Variables declared using `var` are scoped to the function they are declared in, meaning they cannot be accessed from outside that function.
- **Example**:

```
function test() {
    var functionVar = "I'm inside the function";
    console.log(functionVar);  // Works fine inside the function
}

console.log(functionVar);  // ReferenceError: functionVar is not defined
```

In Summary:

- **Lexical Scope**: Scope is determined by where the function is written in the code, not where it is called (this is what JavaScript uses).
- **Dynamic Scope**: Scope is determined by the call stack (not used in JavaScript).
- **Block Scope**: Variables declared using `let` and `const` are scoped to the nearest block (added in ES6).
- **Global Scope**: Variables and functions declared in the global context can be accessed from anywhere.
- **Function Scope**: Variables declared within a function are only accessible inside that function.

–

Scoped Explained in More Detail

Hierarchy of Scopes in JavaScript

1. Lexical Scope (The Foundation)

- **Definition**: The scope determined by the code structure at compile time.
- **Key Subtypes**:
  - **Global Scope**
  - **Function Scope**
  - **Block Scope**

2. Subtypes of Lexical Scope

These are all determined by where the code is physically written.

1. **Global Scope**
   - The outermost scope.
   - Variables and functions declared here are accessible throughout the program.
   - Corresponds to the global execution context.
   - In browsers, variables declared with `var` are attached to the `window` (or `globalThis`) object, but `let` and `const` are not.

2. **Function Scope**
   - Variables declared inside a function using `var` are scoped to the entire function.
   - This is **function-specific** and not influenced by blocks inside the function.

3. **Block Scope**
   - Introduced in ES6.
   - Variables declared with `let` and `const` are scoped to the block `{ ... }` in which they are defined.
   - Examples of blocks: loops, `if` statements, `try-catch` blocks.

3. Special Scopes

These are specific to certain JavaScript features and execution contexts.

1. **Script Scope**
    - A unique scope used for `let` and `const` declarations in the global context.
    - Unlike `var`, which is attached to the global object, `let` and `const` in the global scope are not properties of the global object.
    - Visible under the "Script" section in debugging tools.

2. **Module Scope**
    - Scope used for variables declared in ES6 modules.
    - Variables in modules are scoped to the module itself and are not added to the global object.

3. **Closure Scope**
    - Created when a function "remembers" variables from its **lexical environment** even after the outer function has finished executing.
    - Closures allow functions to maintain references to their **outer scope's variables**.

4. Dynamic Scopes

Dynamic scopes are conceptually different from lexical scopes, but they're relevant for some cases in JavaScript.

1. **`this` Scope**
    - The value of `this` is dynamically determined by the **context in which a function is called**, not where it is defined.
    - Examples:
        - In strict mode: `undefined` if not bound.
        - In non-strict mode: Defaults to the global object if not explicitly bound.

2. `with` **Scope (Deprecated)**
    - Creates a dynamic scope for an object, allowing its properties to be accessed as if they were variables.
    - Example:

```
with (someObject) {
  console.log(property); // Accessed dynamically
}
```

    - **Deprecated** and discouraged due to ambiguity and performance issues.

5. Execution Context-Related Scopes

These scopes are created as part of execution contexts during runtime.

1. **Global Execution Context Scope**
    - Corresponds to the global lexical environment.
    - Contains:
        - Global object.
        - Variables and functions declared globally.

2. **Function Execution Context Scope**
    - Created for each function execution.
    - Contains:
        - Local variables and parameters.
        - `this` binding.
        - Reference to its parent lexical environment (closure).

3. **Eval Scope**
    - Dynamically created when the `eval()` function is executed.
    - Allows execution of code in the scope where `eval` is called.
    - **Discouraged** due to security and performance issues.

```
Scopes in JavaScript
├── Lexical Scope
│   ├── Global Scope
│   ├── Function Scope
│   └── Block Scope
├── Special Scopes
│   ├── Script Scope
│   ├── Module Scope
│   └── Closure Scope
├── Dynamic Scopes
│   ├── `this` Scope
│   └── `with` Scope (Deprecated)
└── Execution Context Scopes
    ├── Global Execution Context Scope
    ├── Function Execution Context Scope
    └── Eval Scope
```

let & const in JavaScript, Temporal Dead Zone

- variables declared with `let` & `const` will be hoisted but not assigned any value (mentioned as `<value unavailable>` in browsers).
- **Temporal dead zone** is the time where the memory is created and variable not initialised for `let` & `const` variables

- If we try to access variables that are in temporal dead zone, `ReferenceError` will be thrown which is shown below:

```
basic-js.js:1 Uncaught ReferenceError: Cannot access 'a' before initialization
    at basic-js.js:1:13
```

- `ReferenceError` will also be thrown if we try to access the variable that is not declared. But, the message will be different.
- But, if we try to re-declare `let` variable, it will throw `SyntaxError`. But, this error is detected on parsing phase itself. Parsing phase is where the code not even get started (before even creation of global execution context). So, not even a single line will be executed.
- Also, `SyntaxError` will be thrown if same name is used in `var` and `let` variables as well.

- As `let` is block scope, declaration of `var` with same block is only ambiguous. Just for the reason `var` is function scoped, it doesn't affect the declaration of `let` with same name as of `var` which is defined outside the `let` 's block.
- But, `var` declartion inside same block as of `let` declared will give you `SyntaxError` .
- If there are two variables of `var` with same name (re-declaration of `var`), then it will give you no error.
- `let` is the only thing that will throw error only if a variable (`let` or `var`) with same name declared in same block scope

```
Uncaught
SyntaxError: Identifier 'var2' has already been declared (at basic-js.js:7:9)
```

Difference between let & const

- `const` behaves more strictly such that, declaration without assignment is not allowed
- assignment of values are initialisation is also not allowed.
- It will be constant throughout the scope.
- Missing initialisation on declaration will throw the below `SyntaxError`

```
Uncaught
SyntaxError: Missing initializer in const declaration (at basic-js.js:4:7)
```

- Assignment after initialisation to `const` will throw following `TypeError`

```
basic-js.js:5 Uncaught
TypeError: Assignment to constant variable.
    at basic-js.js:5:10
```

Explanation of Three Errors in JavaScript

- `ReferenceError` is where the variable we are trying to access is not exist or can't able to access them.
- `SyntaxError` refers that the syntax being wrong (not initialising `const` and re-declaration of `let` & const)

- `TypeError` occurs where we try to do something that the specific variable's type won't allow (re-initialisation of variable will be not allowed in the variable of type `const`)

Points to clarify about scopes and behaviours

1. **Block Scope (let & const)**:
    - Variables declared with `let` and `const` are block-scoped, meaning they are accessible only within the block where they are declared.
    - For each block, a **new inner lexical environment** is created, but no new execution context is created.

2. **Function Scope (var)**:
    - Variables declared with `var` are function-scoped, meaning they are accessible throughout the entire function, regardless of block boundaries within the function.

3. **Lexical Environment**:
    - A conceptual data structure tied to the memory component of an execution context.
    - It resolves variable scopes and manages block scopes, nested blocks, and function scopes.

4. **Block Scope Management**:
    - Block scopes are managed by layering **new inner lexical environments** over the current one, without creating a new execution context.
    - This allows precise management of variables within nested blocks.

5. **Script Scope**:
    - The "Script" scope in debugging tools refers to the **separate layer** for `let` and `const` variables declared in the global scope.
    - It visually separates them from `var` , which is attached to the global object.

- Conceptually, it behaves like a scope to avoid conflicts but is not a true scope by the runtime definition.

6. **Reason for Block Scope**:
   - Block scoping (introduced in ES6) prevents accidental variable leakage across blocks.
   - It ensures tighter control and predictable behavior when managing variables in nested blocks.

7. **New Lexical Environment Creation**:
   - A new **lexical environment** is created for every execution context (e.g., global, function, or eval).
   - Inside the same memory component, nested block scopes are abstracted with additional inner lexical environments layered on top.

8. **Var in Block Scopes**:
   - Variables declared with `var` inside a block (e.g., `if`, `for`) are stored in the same memory component as their parent function's lexical environment, ignoring block boundaries.

9. **Abstract Layer for let & const**:
   - For `let` and `const` inside blocks, the inner lexical environment abstracts block scoping, ensuring these variables are accessible only within the specific block.

10. **Execution Context vs. Lexical Environment**:
    - Execution context encompasses both the **code execution phase** and the **memory management phase**.
    - Lexical environments are the memory management structures within the execution context that handle variable resolution.

Block Scope and Shodowing in JavaScript

- Block can even be exist without any special syntax like `if`, `while` or `for` like in other programming languages
- Block is also called as **Compound Statement**. It means that block is used to group multiple statements into one.
- Generally, the syntax like `for`, `while` applies for the statement that is just present immediately below it (only one statement).
- But with the use of block, we can make cover multiple statement to be placed with corresponding `if`, `while` or `for` 's scope (or functionality).
- Without having a single line after `if`, `while` or `for` or not even a block will give us the `SyntaxError`. It requires a single statement to be present at there. But with blocks, we put multiple statements there and thus, the effect of using corresponding syntax (`if`, `while`, `for`) applies for all the statements present inside that block.

```
if(condition)
    statement 1 (immediately present after if);
```

```
if(condition){
    statement 1;
    statement 2;
    statement 3;
    statement 4;
}
```

Shadowing in JavaScript

- Redeclaration of `var` variable will overwrite the existing value. When the `var` outside the block is re-declared inside the block, and when we try to access them, inside the block or even outside the block (within same function), it will shows the updated value.
- This phenomenon is due to the function scoped nature of `var` and thus, this phenomenon is called **shadowing**.
- As `let` & `const` are block scoped, outside `let` & `const` (outside the block) will not interfere with the `let` & `const` declared inside the block. So, accessing the `let` & `const` inside the block will gives us the value assigned inside the block, and accessing the same outside the block will give us the

value assigned outside the block. This phenomenon is also called as **shadowing** (shadowing in `let` & `const`).

- But naturally, it's due to the scoping nature of variables. New inner lexical environment will be created and abstracted from outer environment when we use block. And this **shadowing** in `let` and `const` is the side effect of that inner lexical environment. In fact, for `let` & `const`, two variable memory will be created in two different scope (corresponding function or global scope and in block scope).

```
Output:
20
15
10
5
```

```
Output:
25
25
25
25
```

Illegal Shadowing

```
let a = 20;
{
    var a = 30;
}
// both are in same scope
```

This will throw an error :

```
Uncaught SyntaxError: Identifier 'a' has already been declared
```

In the above code, the scope of `let` is global scope. `var` is not block scoped in nature and comes under global scope. So, both variables are in global scope that gives the effect of `let` being redeclared and thus this is not allowed. This is called as **Illegal Shadowing** in JavaScript.

> **Note:** Redeclaration in same scope is not allowed for `let` . The redeclaration may be in the form of `let` or `var` .

```
let a = 20;
{
    let a = 30;
}
// this is allowed
// both are in different scope


var a = 20;
{
    var a = 30;
}
// this is also allowed
// both are in same scope, but it's `var`, redeclaration is allowed


var a = 20;
{
    let a = 30;
}
// this is also allowed
// both are in different scope
```

Closures in JavaScript

Functions bundled with its lexical environment is called **closures**.

Functions are **first class objects** in JavaScript. Here is the example for the same.

```
function a1(){
}
a1.authorName = "Danusshkumar";
console.log(a1.authorName);
```

1. Closure in JavaScript

A **closure** is a special type of function in JavaScript that "remembers" the environment in which it was created, even after the outer function has finished executing. This means that a function can maintain access to variables from its

outer (lexical) scope, even when the outer function has finished executing and returned.

Key Points:

- **Closure Definition:** A closure is formed when an inner function retains access to the variables from its outer function, even after the outer function's execution context has been removed from the call stack.
- **How Closures Work:** A closure "remembers" the lexical environment in which it was created. This includes variables from the outer function's scope that the inner function might use.
- **Creation of Closures:** Closures are created when the inner function is defined, not when it is executed. The lexical environment (a reference to the outer variables) is captured during function definition.

Example of a Closure:

```
function outer() {
  const outerVar = "I am from outer scope";

  function inner() {
    console.log(outerVar); // inner function has access to outer function's var
  }

  return inner; // The returned inner function is a closure
}

const closure = outer();
closure(); // Logs "I am from outer scope"
```

In this example, even though `outer()` has finished executing, the returned function `inner` still has access to the `outerVar` from its lexical environment because of the closure.

2. How Closures Are Internally Managed

- **Closure as an Internal Structure:** A closure is not an object that you define explicitly. Rather, it is an internal mechanism created by the JavaScript engine. It's an object-like structure that stores references to the variables needed from the parent lexical environment.

Example:

```
function outer() {
  let count = 0;  // Primitive
  const obj = { name: "JavaScript" };  // Object

  return function inner() {
    count++;  // Closure captures count
    console.log(obj.name);  // Closure captures reference to the object
  };
}

const closure = outer();
closure();  // count is incremented, and obj.name is logged
```

3. Garbage Collection and Closures

The JavaScript **Garbage Collector (GC)** is responsible for freeing up memory to objects and the variales that are no longer in referenced.

- **GC's Working Mechanism**
  - GC runs periodically, but its exact timing depends on the JavaScript engine’s internal heuristics and available system resources.
  - It identifies **unreachable objects**, meaning objects that cannot be accessed or referenced by any part of the program anymore. Those identified objects only be **garbage collected - destroyed**.

- **Closure's Part on holding reference**
  - When a function get called, new Execution context will be created and reference for the same is added into Call Stack
  - When a function returns, the execution context of the function will be deleted.
  - That means, the memory component of the execution context will be deleted.
  - On such deletion, Non-primitive data types (such as objects) are actually stored in Heaps and are referenced by memory component of current function's execution context.

- Primitive data type variables are stored somewhere and they are referenced in memory component
- These variables are refernced by memory components as they are declared and used on current function specifically as because current function is invoked.
- Now, as the function is returning, the execution context's memory component is yet to be deleted meaning that memory component will drop all its references for all the variables
- But, closures will still holds the references for the variables that its inner function needs. **Closure will only hold the reference to the variable that inner function needs. All other variables are not referenced by closures**.
- As garbage collector only destroys the variables that are not referenced, it will not delete the variables that are referenced by closures. But for other variables in same execution context, it's unreachable (no references) and will be deleted by garbage collector.

In other words,

1. **Inner Function and Closure**:
    - When an inner function is declared within an outer function, it forms a closure over the variables in the outer function's scope. A **closure** allows the inner function to "remember" and access variables from its surrounding lexical scope, even after the outer function has finished executing.

2. **Variable Scope in Closures**:
    - When the outer function returns, its execution context is typically destroyed, and its local variables are no longer accessible. However, if the inner function (closure) is still in scope (i.e., it is returned or passed around), it retains references to the variables from the outer function's scope, even though the outer function has finished executing.

3. **Garbage Collection**:

- JavaScript’s **garbage collector (GC)** works by identifying objects that are no longer referenced by any part of the program and then reclaiming the memory they occupy.
- When a closure holds references to variables from the outer function's scope, those variables are not garbage-collected immediately. The GC won’t remove them as long as the closure (or any reference to it) is still in scope. Essentially, the closure creates a "persistent" reference to those variables.
- As long as the closure is alive (i.e., it remains accessible and the references to those variables still exist in memory), the garbage collector will not reclaim the memory used by the variables that are part of the closure.

Example

Here’s an example that might help visualize this:

```
function outerFunction() {
  let outerVariable = "I am an outer variable";

  function innerFunction() {
    console.log(outerVariable);
  }

  return innerFunction;  // Returning the inner function (closure)
}

const closure = outerFunction();  // outerFunction executes, but
innerFunction is returned and still holds a reference to outerVariable

closure();  // This still logs "I am an outer variable", even though
outerFunction has finished execution
```

- When `outerFunction` is called, it creates `outerVariable` and `innerFunction`.
- The `innerFunction` is returned, and it forms a closure, keeping a reference to `outerVariable` even after `outerFunction` finishes execution.
- The variable `outerVariable` remains in memory as long as the `closure` is in scope and accessible. The garbage collector will not delete it because the closure still holds a reference to it.

Garbage Collection in Context of Closures

- **GC Behavior**: JavaScript's garbage collector follows a **reference-counting** and **mark-and-sweep** approach. It checks if there are any active references to variables and if there are none, it considers those variables eligible for garbage collection.

- **When will the GC remove variables?**:
  - Once the closure is no longer accessible (e.g., the closure is discarded or goes out of scope), then the references to the outer function's variables are no longer retained.
  - At this point, the garbage collector can safely reclaim memory from those variables, because no live references remain.

GC and Closures Example:

```
function outer() {
  let counter = 0;  // A primitive variable
  return function inner() {
    counter++;  // Closure references counter
    console.log(counter);
  };
}

const closure = outer();
closure(); // Logs 1
closure(); // Logs 2

// The outer function has returned, but counter is not deleted by GC because
it's referenced by the closure
```

Important Note:

- If, the inner function need to access it's immediate parent's variable and grand parent's variable, debugger may show it as two closures as two scopes are involved.
- Technically, it's a single closure that is handling those references.

4. Data Encapsulation via Closures

Closures are widely used to achieve **data encapsulation** in JavaScript, which helps in **hiding** data from the outside scope. This is one of the most common uses of closures, where you can expose only certain methods to interact with private data.

Example of Data Encapsulation:

```
function createCounter() {
  let count = 0;  // count is a private variable

  return {
    increment: function () {
      count++;  // count is accessed and modified through the closure
      return count;
    },
    decrement: function () {
      count--;
      return count;
    },
    getCount: function () {
      return count;
    }
  };
}

const counter = createCounter();
console.log(counter.increment()); // 1
console.log(counter.increment()); // 2
console.log(counter.decrement()); // 1
console.log(counter.getCount()); // 1
```

In this example, `count` is encapsulated within the closure, and the outer scope has no direct access to it. The only way to modify or access `count` is via the methods returned by the closure.

5. What Happens When the Outer Function is Called Multiple Times?

Every time the **outer function** is called:

- A **new execution context** is created.
- A **new lexical environment** is established.
- A **new closure** is created, associated with the new lexical environment.

- This ensures that variables in closures from different calls to the outer function do not interfere with each other.

Example of Multiple Closure Creation:

```
function outer() {
  let count = 0;
  return function inner() {
    count++;
    console.log(count);
  };
}

const closure1 = outer();
const closure2 = outer();

closure1(); // Logs 1
closure1(); // Logs 2

closure2(); // Logs 1 (Different closure, different scope)
```

Here, `closure1` and `closure2` maintain separate `count` values, demonstrating how closures are scoped to each call to `outer()`.

6. Closures with No Reference to Outer Variables

Closures can exist even if the inner function does not reference any variables from the outer scope. These closures still retain their lexical environment, but since they don't capture any external variables, they effectively become just regular functions that "remember" their original lexical environment.

Example: Closure with No Outer Variable Reference:

```
function outer() {
  const outerVariable = "I am from outer scope";

  // Inner function does not use outerVariable
  return function inner() {
    console.log("I am a closure without referencing outer variables.");
  };
}

const closure = outer();
closure();  // No outerVariable accessed
```

In this case, the inner function is still forms a closure, but it doesn't reference any variables from the outer scope.

7. Closures: Efficient or Inefficient?

- **Closures** themselves are generally **considered an efficient and powerful feature** in JavaScript. They allow for **data encapsulation**, **state retention**, and **function currying**.
- However, closures can lead to inefficiencies if they inadvertently capture unnecessary variables, leading to **memory leaks**.

Advantages of Closures:

- **Encapsulation and Privacy:** Closures allow you to encapsulate state, which is crucial for creating private variables.
- **Callback Functions:** Closures are used to create callbacks, event handlers, or asynchronous code like `setTimeout` or `Promise` handling.
- **Partial Application and Currying:** Closures support functional programming patterns like partial function application and currying.

Uses of Closures

1. Module Design Pattern
2. Currying
3. Functions like once
4. Memoization
5. Maintaining state in async world
6. setTimeouts
7. Iterators
8. and many more...

Disadvantages of Closures:

- **Memory Leaks:** Closures can lead to memory leaks if they hold references to large objects or variables that are no longer needed.
- **Increased Memory Usage:** Using many closures can increase memory usage, particularly if closures are repeatedly created in a loop.

- **Complexity in Debugging:** Since closures retain references to the outer scope, it may be harder to reason about their behavior, especially in large applications.

Conclusion

Closures are an essential and powerful concept in JavaScript, allowing functions to maintain access to their lexical environment even after their outer function has executed. They provide the ability to encapsulate data and create private variables, support callback functions, and enable functional programming patterns. However, managing closures efficiently is crucial to avoid memory leaks, especially when they inadvertently hold references to unnecessary data.

`setTimeOut` + Closures Interview Question

Time and tide waits for none. Just like that, JavaScript also waits for none. JavaScript as a synchronous programming language, it waits for no one eventhough a asynchronous code is present. When a asyncrhonous code is present, it will be moved to microtask or macrotask queue (generally called as callback queue) and waits until current synchronous code completes its execution.

After current code gets executed, event loop will fetch those in queue and loads them to call stack for execution.

Example 1

```
var i = 1;
setTimeout(function (){
    console.log(i);
}, 3000);
console.log("Hello world");
// console logs:
// Hello world
// 1 (after 3 seconds)
```

Example 2

```
function x() {
  for (var i = 1; i <= 5; i++) {
    setTimeout(function() {
      console.log(i);
    }, i * 1000);
  }
  console.log("Namaste JavaScript");
}

x();
// Prints 5 times "6" in regular intervals
```

- This is due to the combinatory behaviour of `var` and closures. The `setTimeout` function forms closure with `i` variable's reference.
- The `i` variable is evaulated on second parameter and corresponding value get stored on timing, but function only takes the reference of `i` by closures.
- At the time when function runs, the value of `i` will be 6. As `var` is function scoped, for all the iterations, the `var` is same and all the closures are pointing to same `i` only.
- That's why 6 is printed in regular interval

Simple solution for this problem is to make closure be formed with new variable with updated `i` value for that individual function.

Here are two ways to achieve that

Example 1.1

```
function x() {
  for (let i = 1; i <= 5; i++) {
    setTimeout(function() {
      console.log(i);
    }, i * 1000);
  }
  console.log("Namaste JavaScript");
}

x();
```

- As `let` is block scoped, for every iteration as block is executed as brand new, `i` variable is createed anew and assigned with updated value.

- As when a function is called everytime, the passed parameters are considered anew and stored in new memory location, loops behaves the same way.
- Every time the block runs, if a variable is created inside the block, that variable will be anew for that block as it's block scope. Same applies for variables declared on `for` 's syntax. They are block scoped and do not accessed outside the `for` loop.
- This block scoped nature in `for` loop is not invented by JavaScript but all languages works the same way.
- So, as the `let` is block scoped, the function will form cloosure with local `let` variable specific for that block.

Example 1.2

```
function x() {
  for (let i = 1; i <= 5; i++) {
      function close(x){
        setTimeout(function() {
          console.log(x);
        }, x * 1000);
      }
      close(i);
  }
  console.log("Namaste JavaScript");
}

x();
```

- With this, we can implement the same behaviour with `var` .
- Here, everytime the function `close` gets called, new execution context is created with new `x` for that execution context.
- So, the callback function inside `setTimeout` will form closure with the `x` variable of the `close` function, not the `i` variable.
- This is achieved only because, arguments are passed by value, not by reference. Also due to the fact that `var` is function scoped

Crazy JavaScript Interview

```
let a = 10;
function outer(){
    function inner(){
        console.log(a);
    }
    return inner;
}
let close = outer();
a = 20;
close();
// console log: 20 (closure stores and preserves the reference to the
required variable)



function Counter() {
  var count = 0;

  this.incrementCounter = function () {
    count++;
    console.log(count);
  };

  this.decrementCounter = function () {
    count--;
    console.log(count);
  };
}

var counter1 = new Counter();

counter1.incrementCounter();
counter1.incrementCounter();
counter1.decrementCounter();

// console logs:
// 1
// 2
// 1
```

Some questions and answers about behaviour

1. Constructor Functions and the `new` Keyword

- A function becomes a **constructor function** when it is **invoked with the `new` keyword**.
    - `new Counter()`:
        - Creates a **new object**.

    - Sets the `this` keyword inside the function to reference that new object.
    - Automatically returns the object (unless something else is explicitly returned).
- If the function is called **without** `new`, it behaves like a regular function:
    - `this` will refer to the global object in non-strict mode, or `undefined` in strict mode.

2. What Happens If a Constructor Function Explicitly Returns Something?

- **Case 1: Explicitly returns an object:**
    - If the constructor function returns an object explicitly, that object is returned instead of the one created by `new`.

```
function Counter() {
    return { custom: "This is a custom object" };
}
const obj = new Counter();
console.log(obj); // { custom: "This is a custom object" }
```

- **Case 2: Explicitly returns a primitive value:**
    - If a primitive value (e.g., a number, string) is returned, it is **ignored**, and the object created by `new` is returned instead.

```
function Counter() {
    return 42;
}
const obj = new Counter();
console.log(obj); // Counter {}
```

3. Why Is `count` Not Part of the Object?

- `var count` is a **local variable** within the `Counter` function and **not part of the object** created by `new`.
    - It is accessible only through the **closure** formed by `incrementCounter` and `decrementCounter`.

- It persists because:
  - Both `incrementCounter` and `decrementCounter` capture the reference to `count` through closures.
  - Even after the `Counter` function execution completes, the `count` variable remains in memory because the closures keep it referenced.

4. Do Both Functions Share the Same Variable?

- Yes! Both `incrementCounter` and `decrementCounter` form closures over the **same `count` variable**.
  - They don't create separate copies of `count`.
  - They **reference the same variable** in the memory created during the `Counter` function's execution.

5. Is This the Same as Returning an Object with Functions?

- Functionally, yes, but there are key differences:
  - Returning an object with functions explicitly:
    - The functions become properties of the returned object.
    - The variable `count` still persists because of closures.
  - Example:

```
function Counter() {
    var count = 0;
    return {
        incrementCounter: function () {
            count++;
            console.log(count);
        },
        decrementCounter: function () {
            count--;
            console.log(count);
        },
    };
}
const counter1 = Counter();
counter1.incrementCounter(); // 1
counter1.incrementCounter(); // 2
counter1.decrementCounter(); // 1
```

6. Key Differences Between Constructor Functions and Returning Explicit Objects

- `new` **Keyword:**
  - With a constructor function, you use the `new` keyword to create the object.
- **Prototypes:**
  - Constructor functions link the created object's prototype (`Counter.prototype`) to provide inheritance features.
  - Explicitly returning an object does not utilize prototypes unless you manually set them.
- **Memory:**
  - Both approaches rely on closures for variables like `count` , but the constructor function may involve slightly different optimizations at runtime.

Relation between Garbage Collector and Closure

Example 1

```
function outer(){
    var x = 0;
    function inner(){
        console.log("Hello world");
    }
}
var close = outer();
// Here x will be garbage collected after `outer` function returns
```

Example 2

```
function outer(){
    var x = 0;
    function inner(){
        console.log(x);
    }
}
outer();
// Here, inner function forms the closure and the function along with closure is returned, but as soon as the execution completed, we no longer have access to the returned inner function. Thus, we also don't have access to retuend clsoure. As closure is not referenced, the returned function, closure along with variable `x` will be garbage collected
```

Example 3

```
function outer(){
    var x = 0;
    function inner(){
        console.log(x);
    }
}
var close = outer();
// As long as we have access to close (until the program ends), `x` will not be garbage collected
```

Legacy Theoretical Closure Model by MDN

- Theoretical Closure model from official documentation says that entire lexical environment will be referenced by closure.
- This is followed by some legacy JS engines.
- Modern engines optimised the same that closure is formed only with neccessary variables.
- By the very less explanation from official documentation of JavaScript, Closure's theoretical model includes all the lexical environment's functions and variables

First Class Functions in JavaScript

```
// Function statement | Function declaration
function sum(a,b){
    return a + b;
}

// Function expression
// 1. Anonymous Function exression
let sum1 = function (a, b){
    return a + b;
}

// 2. Named Function Expression
let sum2 = function sumFunction(a, b){
    return a + b;
}

// 3. Arrow Function Expression
let sum3 = (a, b) => {
    return a + b;
}

// Difference between parameters and arguments
function print(a, b){
    console.log(a, b);
}
print("Hello", 5);
```

- Function Expression and Function Declaration differs in hoisting. Function declaration are fully hoisted whereas in function expression, variable is only hoisted with `undefined` stored if it is `var` and will be in *Temporal Dead Zone* if it is `let` & `const`. Function will be assigned to them only after execution of assignment statement
- For Named Function Expressions, we cannot access the function with its function name outside them. The function name is applicable only inside that function scope. `sumFunction(5,6)` outside the function will throw an `ReferenceError`. Instead we may call `sum2(5,6)`.
- In the definition of function `print`, `a` and `b` are parameters whereas value we passed ("Hello" and 5) are arguments

The ability of function to behave like as values is called **first class functions** in JavaScript. Functions can be returned from a function, can be passed as arguments to a function and so on.
For this ability, functions are also refered as **First class Citizens**.

Callback Functions in JS - Event Listeners

What is a Callback functions in JavaScript ?

- Function that is passed as an argument to other function is called **callback function**.
- With the feature of **callback**, async nature is made possible in JavaScript, eventhough JavaScript is a **synchronous and single-threaded language**.
- The function we passed may be called back later in our code on some other time. That's why it's called as **callback** functions.
- Every operations in JavaScript are done only after loaded into callstack. So, if any operation blocks this callstack, that is known as **Blocking the main thread**.
- When we write some heavy function in the main code, then on-the-flow, it will be executed. The codes below this heavy function may be some important code and that may be used to render the UI correctly. In such scenario, JavaScript will be busy processing our heavy function and rest of the code will not be executed at that time.
- In such time, the UI may freeze or UI may not respond. This is why, it's suggested to write our heavy resource intensive function in asynchronous, so that they will be executed after running the main code.
- This act of writing the heavy function without asynchronously is called **blocking the main thread** and it must be avoided.

Callbacks with Event Listeners

```
let count = 0;
document.getElementById("clickMe").addEventListener("click",
    function callBackFunction(){
        console.log("Button clicked: ", ++count);
    }
);
```

- Here, the `callBackFunction` is a callback function that forms closure with its environment.
- But, exposing the variable `count` globally is not an good practise (for security purposes).

- So, inorder to write them securely that `callBackFunction()` only access it, we're enclosing the entire event listener addition into a single function along with `count` variable:

```
function close(){
    let count = 0;
    document.getElementById("clickMe").addEventListener("click",
    function callBackFunction(){
        console.log("Button clicked: ", ++count);
    });
}
close();
```

Importance of removing event listeners

- Event listeners are heavy thing that consumes heavy memory. Eventhough the call stack is empty, event listeners and its callback function will still stores the closures of all its environment.
- Creating lots of event listeners will make our webpage slow.
- That's why removing the event listeners are important

Asynchronous JavaScript and Event Loop

All the APIs for accessing powerful browser stuffs shown in above picture are accessed by JavaScript using 'global' or 'window' object provided by browser.

So, we can either use `window.console.log("Hello world")` or `console.log("Hello world")` and these are just the same.

The only work of **event loop** is that it checks whether the call stack empty or not. If it is empty and if there is any function that's waiting in callback queue or microtask queue, then, that will be added to callstack by **event loop**.

Working of `fetch()`

`fetch()` 's callback functions won't be added into callback queue, instead it will be added to microtask queue. Because, it has higher priority.

microtask queue is given higher priority such that eventhough microtask queue contains 10 functions to be executed and callback queue contains only one, the callback queue will only be executed after the execution of all the 10 functions in the microtask queue.

Microtask queue is given priority over callback queue. So, what are all pushed to microtask queue ?

All the callback functions which comes through the promises will be added to microtask queue.

Mutation Observer will continously check for mutation in the DOM Tree. If there is any mutation there, some callback functions will be added to microtask queue.

In simple terms, callback functions from,

- Promises
- Mutation Observer

are added to microtask queue.

Callback queue is also called as **task queue** or **macrotask queue**

When the microtask creates some other microtask and adds more microtasks to the microtask queue, the callback function present in the callback queue (even the no.of function in callback queue is one), it will not get the chance to execute until all microtask including those that are created while the former microtask was running.

This condition is called **Starvation**

JS Engine Exposed - Google's V8 Architecture

JavaScript runtime environment contains all the components required to run the JavaScript code.

Those components are:

- JS Engine
- APIs
- Event loop
- Callback queue
- Microtask queue

- JavaScript can run in Browsers, servers, Watches, robotic devices and so on.
- Inorder to make JavaScript run in an device, we need JavaScript Runtime Environment
- JavaScript Engine is the heart of JavaScript runtime environment.
- Also, the API part of JRE can vary depend on what type of JRE it is.
- If it's browser, may be it have different set of API to do some DOM Manipulation, if it's Node, may be it may have different set of API.
- Some of the API functions may be common among them. Ex: `setTimeout`. Eventhough the names are same, they might have different implementation under the hood.

There are lots of JavaScript engines available and different browsers uses different JavaScript engine.

We even can create a JavaScript engine that must follow the ECMAScript standards.
The very first JavaScript engine was created by the developer of JavaScript and now, it's evolved a lot and called as *SpiderMonkey*, the one that is used in Mozilla Firefox

Architecture of V8

V8 was wrriten using **C++**

JS Engine took JS Code as input and do three steps to execute the code

Steps:

- Parsing
- Compilation
- Execution

Pasring

Parsing:

Written JavaScript code is parsed and splitted into tokens by **Syntax Parser**

```
let a = 5;
```

Tokens: `let` , `a` , `=` , `5` and `;`

After tokens are generated, **AST** (Abstract Syntax Tree) is generated from the token.

```
const bestJSCourse = "Namaste JavaScript";
```

Visualisation of **AST** for the above code:

Compilation and Execution

Interpreted Languages	Compiled Languages
In many programming languages, the code is executed using an interpreter.	Some other languages uses compiler to compile their code.
Interpreter executes the code line by line.	Compilers convert the source code into a optimized version of code and then it will be executed.
Advantage of Interpreter is that it's fast.	Compiler in the other hand is more efficient.

- When JavaScript was designed, it was first designed as an interpreted language. JS uses interpreter initially. As it's majorly used in browsers at that times, browser will not have to wait for JS to compile and then run. If so, there will be some performance issues such as UI freezing, etc. That's why it's first designed as Interpreted language.
- But, now JS engines use both interpreters and compilers and that makes JavaScript a JIT Compiled language (Just-in-Time compiled language).
- Both Compiler and Interpreters works to generate byte codes. Interpreters interprets the code into bytecode while Compiler do some optimisation as much as it can. This process is not an single pass process as it can go multiple passes in different manner.
- Different JS Engines uses their own algorithm of doing this. Different JS engines uses different algorithms to make both interpreters and compilers communicate effectively to do things.
- That's why it is **JIT - Just In Time Compilation**
- Some languages may use **AOT - Ahead Of Time compilation** where Compiler take a bunch of code which is going to be executed later and do as much optimisation as possible by it.
- Finally, the byte code goes to the Execution phase.
- Execution phase will not be possible without the two major components - **Callstack** and **Memory Heap**
- Memory Heap is where data are stored. It will be constantly in-sync with Callstack and Garbage collector

Fastest JavaScript Engine - V8

V8's interpreter is called **Ignition** and it's compiler is called **TurboFan**.
It's garbage collector is called **Orinoco**. They also have garbage collector called **Oilpan** which is used for different purpose

There are few languages which are procedural programming languages. There are some languages which are functional programming languages. There are some languages which are Objec Oriented Programming languages.

But, JavaScript supports all of this programming paradigms.

Mark & Sweep Algorithm

The **Mark and Sweep** algorithm is a fundamental approach used in garbage collection to manage memory by identifying and reclaiming unused objects. It consists of two main phases: **marking** and **sweeping**. Here's a step-by-step explanation:

1. Mark Phase

- **Objective**: Identify objects in memory that are still "reachable" or in use.
- **Process**:
    1. Start from a set of "root" objects (like global variables, function arguments, and local variables on the stack).
    2. Traverse the object graph recursively from these roots.
    3. Mark every object that can be reached directly or indirectly from the roots.
- **Key Points**:
    - Objects are considered "reachable" if there is a direct or indirect reference to them.
    - Any object not marked as reachable is considered unused or "garbage."

2. Sweep Phase

- **Objective**: Reclaim memory occupied by unreachable objects.
- **Process**:
    1. Traverse all objects in memory.
    2. For each object:
        - If it is marked as reachable, clear the "marked" status for the next garbage collection cycle.
        - If it is not marked, it is considered garbage and is deallocated or returned to the free memory pool.
- **Key Points**:

- This phase removes unused objects and frees up memory for reuse.

Advantages

1. **Simple and Effective**: Easy to implement and works well for many common scenarios.
2. **Non-moving (typically)**: Does not require relocating objects, which simplifies memory management.

Disadvantages

1. **Stop-the-World**: Both the mark and sweep phases may require pausing program execution, which can cause latency in real-time systems.
2. **Fragmentation**: Memory may become fragmented because unused objects are deallocated without compacting the remaining objects.
3. **Performance Overhead**: Can be slower than other advanced garbage collection techniques like generational garbage collection.

How It Works in Practice

- **Root Set**: Includes objects like global variables, the call stack, and active closures.
- **Traversal**: Common traversal methods include depth-first or breadth-first search.
- **Implementation**: Mark and sweep forms the foundation of modern garbage collectors, often enhanced with techniques like:
  - **Incremental collection**: Breaks the process into smaller steps to reduce latency.
  - **Generational collection**: Divides objects into "young" and "old" generations for more efficient cleanup.

Some Optimisation of JS Compilers

1. Inlining

- **What It Is**:
  - Replacing a function call with the actual code of the function.
  - This eliminates the overhead of calling a function and improves performance by avoiding the need to push/pull data on/from the call stack.

- **Example**:

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

const result = add(3, 5);
```

  After Inlining:

```
const result = 3 + 5;
```

- **Advantages**:
  - Reduces function call overhead.
  - Enables further optimizations, like constant folding or loop unrolling.

- **Limitations**:
  - Inlining large functions can increase code size (**code bloat**) and affect performance negatively.
  - Often used selectively by JavaScript engines.

2. Copy Elision

- **What It Is**:

- Avoiding unnecessary copying of data, especially objects, during assignment or return.
- Instead of creating a temporary copy, the JavaScript engine reuses the same memory or directly constructs the object in the final location.

- **Example**:

```
function createObject() {
  return { x: 10, y: 20 };
}

const obj = createObject();
```

  Optimization:

  - Instead of copying the object into `obj`, the object is directly created in the location where `obj` points.

- **Advantages**:

  - Saves time and memory by eliminating intermediate copies.
  - Improves performance for large objects or frequent object creation.

3. Inline Caching

- **What It Is**:

  - A technique to optimize property access by remembering the shape (structure) of objects and caching the result of previous lookups.
  - JavaScript engines, like V8, use this technique to speed up repeated property access on objects with the same structure.

- **Example**:

```
function accessProperty(obj) {
  return obj.name;
}

const user = { name: "Alice" };
console.log(accessProperty(user));
```

How It Works:

  - The first time `obj.name` is accessed, the engine resolves it and caches the location of `name` in the object.
  - Subsequent accesses to `obj.name` reuse the cached information if the object's shape (its structure and property layout) hasn't changed.

- **Advantages**:

  - Speeds up repeated property lookups by avoiding re-resolving the property each time.
  - Works well with objects that have consistent shapes.

- **Key Concept**:

  - Objects in JavaScript have a hidden structure, often called a "hidden class" or "shape."
  - If the shape changes (e.g., adding or removing properties), the cache is invalidated, and the lookup process starts over.

Trust Issues with `setTimeout()`

- Concurrency Model in JavaScript works this way such that the callback queue will only be loaded into memory after callstack becomes empty.
- Eventhough, we set `setTimeout(callbackFunction, 5000)`, if there is some code that takes 10 seconds to execute, the timer will expire and the `callbackFunction` will wait on the queue for next 5 seconds for main thread to completes its execution.
- So, `setTimeout` guarantees that the function will run atleast after 5 seconds but not exactly after 5 seconds.
- This is one of the reasons that we should not block our main thread with such code that runs for 10 seconds.
- `setTimeout(function(){}, 0)`, this syntax is used to intentionally put a function into callback. Eventhough the timer is 0 seconds, the function still go through all the steps a typical `setTimeout` callback functions do.
- If we have some less important function that's not needed to run as of now, or some function that do additional stuff that takes more computation, we

may put them like this.

- This will gives way for important code to run first and this additional code may run once the important code executed completely.
- If we want our resource intensive code not to block the UI rending, we can make them asynchronous like this.

Higher Order Function & Functional Programming

A function which takes function as a argument or returns function is called as **higher order function**.

Here is an example of writing program for calculating area, circumference and diameter of circles given radium array.

DRY (Don't Repeat Yourself) way of writing code

```
const radius = [3, 1, 2, 4];

const calculateArea = function(radius) {
  const output = [];
  for (let i = 0; i < radius.length; i++) {
    output.push(Math.PI * radius[i] * radius[i]);
  }
  return output;
};

console.log(calculateArea(radius));

const calculateCircumference = function(radius) {
  const output = [];
  for (let i = 0; i < radius.length; i++) {
    output.push(2 * Math.PI * radius[i]);
  }
  return output;
};

console.log(calculateCircumference(radius));

const calculateDiameter = function(radius) {
  const output = [];
  for (let i = 0; i < radius.length; i++) {
    output.push(2 * radius[i]);
  }
  return output;
};

console.log(calculateDiameter(radius));
```

Modular way of writing code - Functional programming

```
const radius = [3, 1, 2, 4];

const area = function (radius) {
  return Math.PI * radius * radius;
};

const circumference = function (radius) {
  return 2 * Math.PI * radius;
};

const diameter = function (radius) {
  return 2 * radius;
};

const calculate = function (radius, logic) {
  const output = [];
  for (let i = 0; i < radius.length; i++) {
    output.push(logic(radius[i]));
  }
  return output;
};

console.log(calculate(radius, area));
console.log(calculate(radius, circumference));
console.log(calculate(radius, diameter));
```

Using `.map()` method

```
const radius = [3, 1, 2, 4];

const area = function (radius) {
  return Math.PI * radius * radius;
};

const circumference = function (radius) {
  return 2 * Math.PI * radius;
};

const diameter = function (radius) {
  return 2 * radius;
};

const calculate = function (radius, logic) {
  const output = [];
  for (let i = 0; i < radius.length; i++) {
    output.push(logic(radius[i]));
  }
  return output;
};

console.log(radius.map(area));
console.log(radius.map(circumference));
console.log(radius.map(diameter));
```

Our own way of defining `map`

```
const radius = [3, 1, 2, 4];

const area = function (radius) {
  return Math.PI * radius * radius;
};

const circumference = function (radius) {
  return 2 * Math.PI * radius;
};

const diameter = function (radius) {
  return 2 * radius;
};

Array.prototype.calculate = function (logic) {
  const output = [];
  for (let i = 0; i < this.length; i++) {
    output.push(logic(this[i]));
  }
  return output;
};

console.log(radius.calculate(area));
console.log(radius.calculate(circumference));
console.log(radius.calculate(diameter));
```

Prototype and Prototypal Inheritance in JavaScript

What is Prototypal Inheritance?

- **Definition**: Prototypal inheritance allows objects to inherit properties and methods from other objects. This is achieved via the **prototype chain**.
- Every JavaScript object has an internal link (known as `[[Prototype]]` or `__proto__`) to another object, called its **prototype**.
- If a property or method is not found on the object itself, the JavaScript engine looks up the prototype chain to find it.

How Prototypal Inheritance Works

1. Prototype Chain

- When you access a property or method on an object:
  1. JavaScript first checks the object itself.
  2. If not found, it looks at the object's prototype (`__proto__`).
  3. This continues up the chain until the property is found or the chain ends at `null` (the top of the prototype chain).

2. Example

```
const obj1 = { name: "Dan" };
const obj2 = Object.create(obj1); // obj2's prototype is obj1

console.log(obj2.name); // Output: "Dan"
```

- `obj2` does not have a `name` property, so it looks in its prototype (`obj1`).

Changing the Prototype

- You can add or modify properties and methods on an object's prototype, and the changes are immediately reflected in all objects inheriting from it.

1. Adding Methods to Built-In Prototypes

```
Array.prototype.sum = function () {
  return this.reduce((a, b) => a + b, 0);
};

const arr = [1, 2, 3];
console.log(arr.sum()); // Output: 6
```

- Adding `sum` to `Array.prototype` makes it available to all arrays.

2. Changing the Prototype of an Object

```
const obj = { a: 1 };
const newPrototype = { b: 2 };

Object.setPrototypeOf(obj, newPrototype);

console.log(obj.b); // Output: 2
```

Behind the Scenes: Class-Based Syntax and Inheritance

1. Classes are Syntactic Sugar

- JavaScript's `class` syntax is a more readable way to define prototypes and constructors but still relies on prototypal inheritance under the hood.

2. How Classes Work

```
class Parent {
  greet() {
    console.log("Hello from Parent");
  }
}

class Child extends Parent {
  greet() {
    console.log("Hello from Child");
    super.greet(); // Call Parent's greet method
  }
}

const child = new Child();
child.greet(); // Output: "Hello from Child" followed by "Hello from Parent"
```

- **Behind the Scenes**:
  - The `Child` prototype inherits from `Parent.prototype`.
  - `super` allows accessing methods from the parent class (which is part of the prototype chain).

Understanding `Function`, `Array`, and Their Prototypes

1. Function Inheritance

```
Function.prototype.describe = function () {
  console.log(`This function is named: ${this.name}`);
};

function myFunc() {}
myFunc.describe(); // Output: "This function is named: myFunc"
```

- All functions inherit from `Function.prototype`.

2. Array Inheritance

```
Array.prototype.first = function () {
  return this[0];
};

const arr = [10, 20, 30];
console.log(arr.first()); // Output: 10
```

- All arrays inherit from `Array.prototype`.

How Inheritance Works Internally

1. **Object Creation**:
    - When you use `new` or `Object.create`, a new object is created with a prototype pointing to the constructor's prototype.

```
   const obj = Object.create(Array.prototype); // `obj` inherits from
`Array.prototype`
   console.log(obj instanceof Array); // true
```

2. **Prototype Chain**:
    - Objects contain a hidden `[[Prototype]]` property that links them to their prototype.
    - The `prototype` property exists on functions to define the prototype for instances created by the function.

3. **Garbage Collection and Performance**:
    - Prototypes are designed to share behavior efficiently, avoiding duplication of methods across instances.

Prototype Relationships, Constructor Functions, and Class Syntax

---

1. The `__proto__` Assignment and `Person.prototype`

When an object is created using a constructor function, the following relationships are established:

- **Object's `__proto__`**:
  - The object's `__proto__` is assigned a reference to the constructor's `prototype`.
  - This is how the object gains access to the properties and methods defined on the constructor's `prototype`.

Example:

```
function Person(name) {
  this.name = name;
}

Person.prototype.greet = function () {
  console.log(`Hello, my name is ${this.name}`);
};

const john = new Person("John");

// Relationships
console.log(john.__proto__ === Person.prototype); // true
```

- **Key Points**:
  - `john.__proto__` points to `Person.prototype`.
  - If you modify `Person.prototype`, all objects created using `new Person` will see the changes because `__proto__` holds a **reference** to the prototype.

Proof of Reflection:

```
Person.prototype.sayGoodbye = function () {
  console.log("Goodbye!");
};

john.sayGoodbye(); // Output: Goodbye!
```

2. Inheriting Properties from Another Object

If we want `Person` to inherit properties or methods from `Animal`, we can set `Person.prototype.__proto__` to `Animal.prototype`.

Example:

```
function Animal(type) {
  this.type = type;
}

Animal.prototype.makeSound = function () {
  console.log("Animal sound");
};

function Person(name) {
  this.name = name;
}

// Inheritance
Person.prototype.__proto__ = Animal.prototype;

const john = new Person("John");
john.makeSound(); // Output: Animal sound
```

- **Key Points**:
  - `Person.prototype` now inherits from `Animal.prototype`.
  - The `john` object will access properties/methods from `Animal.prototype` through the prototype chain.

3. How `extends` Works Internally

The `class` syntax provides a cleaner and more modern way to handle inheritance, but under the hood, it uses the same mechanism:

- **Internals of `extends`**:
  - `extends` sets up the prototype chain using `Object.setPrototypeOf`.
  - The child class's prototype (`Child.prototype`) is linked to the parent class's prototype (`Parent.prototype`).

Example:

```
class Animal {
  makeSound() {
    console.log("Animal sound");
  }
}

class Person extends Animal {
  constructor(name) {
    super(); // Calls the parent constructor
    this.name = name;
  }

  greet() {
    console.log(`Hello, I am ${this.name}`);
  }
}

const john = new Person("John");

// Relationships
console.log(Object.getPrototypeOf(Person.prototype) === Animal.prototype); // t
console.log(john.__proto__ === Person.prototype); // true
```

- **Under the Hood**:
  - `Object.setPrototypeOf(Person.prototype, Animal.prototype)` links `Person.prototype` to `Animal.prototype`.
  - The `super()` call ensures the parent constructor is executed during initialization.

4. Constructor Functions and Classes

- **Constructor Functions**:
  - The traditional way of defining an object template.
  - The `new` keyword automatically sets up the `__proto__` link and initializes the object.

Example:

```
function Person(name) {
  this.name = name;
}
```

- **Classes**:
  - A syntactic sugar over constructor functions.
  - The `class` syntax internally uses a constructor function for initialization and `extends` for inheritance.

Class Example:

```
class Person {
  constructor(name) {
    this.name = name;
  }
}
```

Equivalent Constructor Function:

```
function Person(name) {
  this.name = name;
}
```

Key Mistakes to Avoid

1. **Misunderstanding `__proto__` and `prototype`**:
   - `__proto__` is the internal property linking an object to its prototype.
   - `prototype` is a property of functions used to define the prototype for instances created by the function.

2. **Incorrect Inheritance Setup**:
   - Directly modifying `__proto__` works but is discouraged for performance reasons.
   - Use `Object.setPrototypeOf` or `extends` for clean inheritance.

`map`, `filter`, `reduce`

`map` method

```
// double the each elements
function double(x){
    return x * 2;
}

const arr = [5, 1, 3, 2, 6];

const output = arr.map(double);
```

`filter` method

```
// filter odd values
function isOdd(x){
    return x%2 === 1;
}

const arr = [5, 1, 3, 2, 6];

const output = arr.filter(isOdd);
```

`reduce` method

```
// sum of all values
const arr = [5, 1, 3, 2, 6];

const output = arr.filter(function (acc, curr){
    acc += curr;
    return acc;
}, 0);
// acc is the accumulator and curr is the current element
// we need to return after updating the accumulator
// Second argument of `filter` will be default initial value of accumulator
```

Example 1

```
const users = [
  { firstName: "akshay", lastName: "saini", age: 26 },
  { firstName: "donald", lastName: "trump", age: 75 },
  { firstName: "elon", lastName: "musk", age: 50 },
  { firstName: "deepika", lastName: "padukone", age: 26 },
];

// list of full names
// ["akshay saini", "donald trump", ...]

const output = users.map((x) => x.firstName + " " + x.lastName);

console.log(output);
```

Example 2

```
// frequency of ages of people
const users = [
  { firstName: "akshay", lastName: "saini", age: 26 },
  { firstName: "donald", lastName: "trump", age: 75 },
  { firstName: "elon", lastName: "musk", age: 50 },
  { firstName: "deepika", lastName: "padukone", age: 26 },
];


const output = users.map(function(acc, curr){
    if(acc[curr.age] === undefined){
        acc[curr.age] = 1;
    } else {
        acc[curr.age] += 1;
    }
    return acc;
}, {});

console.log(output);
```

Example 3

```
// first name of person with age < 30
const users = [
    { firstName: "akshay", lastName: "saini", age: 26 },
    { firstName: "donald", lastName: "trump", age: 75 },
    { firstName: "elon", lastName: "musk", age: 50 },
    { firstName: "deepika", lastName: "padukone", age: 26 },
  ];

  const output = users.filter(x => x.age < 30).map(x => x.firstName);

  // same using reduce
  const output2 = users.reduce(function (acc, curr){
      if(curr.age < 30)    acc.push(curr.firstName);
      return acc;
  }, []);

  console.log(output);
  console.log(output2);
```