

## What is Closures ?

A **closure** is the combination of a function bundled together (enclosed) with references to its surrounding state (the **lexical environment**). In other words, a closure gives a function access to its outer scope. In JavaScript, closures are created every time a function is created, at function creation time.

JS



```
function init() {  
  var name = "Mozilla"; // name is a local variable created by init  
  function displayName() {  
    // displayName() is the inner function, that forms a closure  
    console.log(name); // use variable declared in the parent function  
  }  
  displayName();  
}  
init();
```

JS


```
function makeFunc() {  
  const name = "Mozilla";  
  function displayName() {  
    console.log(name);  
  }  
  return displayName;  
}  
  
const myFunc = makeFunc();  
myFunc();
```

## Call

The call method calls a function with a given this value and arguments provided individually.

Syntax:


javascript

 Copy code

```
function.call(thisArg, arg1, arg2, ...)
```

Example:

javascript

 Copy code


```
const person = {  
  name: 'Mahesh',  
};  
  
function greet(greeting) {  
  console.log(greeting + ', ' + this.name);  
}  
  
greet.call(person, 'Hello'); // Hello, Mahesh
```

## Apply

The apply method calls a function with a given this value and arguments provided as an array (or an array-like object).

Syntax:


javascript

 Copy code

```
function.apply(thisArg, [argsArray])
```

Example:

javascript

 Copy code

```
const person = {  
  name: 'Mahesh',  
};  
  
function greet(greeting) {  
  console.log(greeting + ', ' + this.name);  
}  
  
greet.apply(person, ['Hello']); // Hello, Mahesh
```



## Differences and Use Cases

call **vs** apply: Both call and apply are used to invoke functions with a specified this context. The difference lies in how arguments are passed:


- call: arguments are passed individually.
- apply: arguments are passed as an array

## **bind**

The bind method creates a new function that, when called, has its this keyword set to the provided value, with a given sequence of arguments preceding any provided when the new function is called

Syntax:

javascript

 Copy code

```
function.bind(thisArg[, arg1[, arg2[, ...]]])
```

Example:

```
javascript Copy code  
  
const person = {  
  name: 'Mahesh',  
  greet: function(greeting) {  
    console.log(greeting + ', ' + this.name);  
  }  
};  
  
const greetMahesh = person.greet.bind(person, 'Hello');  
greetMahesh(); // Hello, Mahesh
```

## Higher Order Functions

Higher-Order Functions are functions that can accept other functions as arguments, return functions

Syntax:

```
function higherOrderFunction(callback) {  
  // Perform some operations  
  // Call the callback function  
  callback();  
}  
function callbackFunction() {  
  console.log("Callback function is executed.");  
}  
// Passing the callback function to the higher-order function  
higherOrderFunction(callbackFunction);
```

- **higherOrderFunction:** Takes a callback function, executes it, and performs operations.
- **callback:** A function passed as an argument, executed inside higherOrderFunction.
- **callbackFunction():** Logs "Callback function is executed."
- **Invocation:** Calls higherOrderFunction(callbackFunction), executing callbackFunction within higherOrderFunction.

## Difference between var and let keyword in javascript.

**Scope:** var is function-scoped, while let is block-scoped.

**Hoisting:** Both var and let are hoisted, but var initializes with undefined, and let remains uninitialized, causing a ReferenceError if accessed before declaration.

**Re-declaration:** var allows re-declaration within the same scope, while let does not.

**Global Object:** var declarations add properties to the global object, while let declarations do not.

### Var

**Function Scope:** Variables declared with `var` are function-scoped. This means they are accessible within the function they are declared in and not outside.

### Let

**Block Scope:** Variables declared with `let` are block-scoped. This means they are only accessible within the block they are declared in (e.g., within `{ }`).

## Explain Hoisting in javascript

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

Example:

```
javascript Copy code  
  
console.log(x); // undefined  
var x = 5;  
console.log(x); // 5
```

Example with `let` :

```
javascript Copy code  
  
console.log(y); // ReferenceError: Cannot access 'y' before initialization  
let y = 5;  
console.log(y); // 5
```

## What is memory leak

A memory leak in JavaScript occurs when the JavaScript engine retains memory that is no longer needed. This can cause the application's memory usage to grow over time, leading to performance issues and potentially causing the application to crash if it exhausts available memory. Memory leaks are particularly problematic in long-running applications like single-page applications (SPAs) or server-side applications using Node.js

Here are some common causes of memory leaks in JavaScript:.

1. **Unintentional Global Variables:** Declaring variables without `var`, `let`, or `const` in non-strict mode can create global variables that persist throughout the life of the application.

```
javascript Copy code  
  
function leakyFunction() {  
    leak = "This is a global variable";  
}
```

4. **Timers and Intervals:** Using `setInterval` or `setTimeout` and not clearing them properly can lead to memory leaks.

```
javascript Copy code  
  
function leakyFunction() {  
  setInterval(() => {  
    console.log("This will leak memory");  
  }, 1000);  
}
```

## What is Lazy loading

Lazy loading is a technique used by web pages to optimize load time. With lazy loading, a web page loads only required content at first, and waits to load any remaining page content until the user needs it. Lazy loading reduces the time it takes for a web page to open because the browser only loads a fraction of the content on the page at a time.

For example, if a web page has an image that the user has to scroll down to see, you can display a placeholder and lazy load the full image only when the user arrives to its location.

Benefits :

- Improved Performance
- Enhanced User Experience:
- Optimized Resource Utilization

Disadvantages:

- Complexity in Implementation
- SEO Implications

## Garbage collection

Garbage collection in JavaScript is a form of automatic memory management that helps reclaim memory occupied by objects that are no longer needed by the application. When you create variables or objects, they consume memory, and if those objects are no longer referenced, they become eligible for garbage collection.

## **lexical environment**

A lexical environment in JavaScript is a data structure that stores variables and functions defined in the current scope, along with references to all outer scopes. It is also known as the lexical scope.

The lexical environment is created when a function is defined and persists as long as the function or any of its closures remain accessible.

## **Global Variables**

In JavaScript, **global variables** are variables that are accessible from any part of the code, regardless of where they were declared

## **What are the different data types present in javascript**




## **Pass by Value**

When a function is called with primitive data types (like numbers, strings, booleans, null, undefined, and symbols), JavaScript passes a copy of the value. Changes made to the parameter inside the function do not affect the original variable.



javascript


 Copy code

```
function changeValue(x) {  
    x = 20; // Modifies the local copy  
}  
  
let num = 10;  
changeValue(num);  
console.log(num); // Output: 10 (original value remains unchanged)
```

## Pass by Reference

When a function is called with objects (including arrays and functions), JavaScript passes a reference to the original object. This means that if you modify the object inside the function, the changes will reflect in the original object.

javascript


 Copy code

```
function changeObject(obj) {  
    obj.property = "Changed!";  
}  
  
let myObj = { property: "Original" };  
changeObject(myObj);  
console.log(myObj.property); // Output: "Changed!" (original object is modified)
```

## Shallow Copy

A shallow copy creates a new object that is a copy of the original object, but it only copies the top-level properties. If the original object contains nested objects, the references to those nested objects are copied rather than the objects themselves. This means that changes to nested objects in either the original or the copied object will affect both.

javascript

 Copy code

```
let original = {
  a: 1,
  b: { c: 2 }
};

// Shallow copy using Object.assign
let shallowCopy = Object.assign({}, original);

// Shallow copy using the spread operator
// let shallowCopy = { ...original };

shallowCopy.b.c = 3;

console.log(original.b.c); // Output: 3 (original object is affected)
```

## Deep Copy

A deep copy creates a new object that is a complete copy of the original object, including all nested objects. Changes made to any nested object in the copied object do not affect the original object.

```
javascript Copy code

let original = {
  a: 1,
  b: { c: 2 }
};

// Deep copy using JSON methods
let deepCopy = JSON.parse(JSON.stringify(original));

deepCopy.b.c = 3;

console.log(original.b.c); // Output: 2 (original object is unchanged)
```

## Strict mode

Strict mode is a helpful feature in JavaScript that promotes better coding practices by enforcing a stricter set of rules. By catching common mistakes and preventing potentially problematic features, it helps developers write cleaner, safer code. Enabling strict mode is a good practice, especially in larger applications or when working in teams.

### Benefits of Strict Mode

- Disallows Duplicate Parameter Names
- Improved Error Handling:

## Pure Function

**Deterministic:** Given the same input, a pure function always returns the same output. It does not rely on any external state or data that can change over time.

**No Side Effects:** A pure function does not cause any observable side effects outside its scope. This means it doesn't modify any external variables or states, and it doesn't perform actions like logging to the console, modifying global variables, or altering the input parameters.

### Example of a Pure Function

```
function add(a, b) { return a + b; // Always returns the same result for the same inputs }
```

### Example of an Impure Function

```
let count = 0; function increment() { count += 1; // Modifies an external variable, causing side effects return count; }
```

## Promises

**Promise.all:** If any of the input promises is rejected, the entire returned promise will immediately reject with the reason of the first rejected promise. The resolution of `Promise.all` depends on all promises being fulfilled successfully.

**Promise.allSettled:** The returned promise always resolves, regardless of whether individual promises were fulfilled or rejected. The result is an array of objects representing the outcomes of all input promises, including both fulfilled and rejected ones.

**Promise.race:** Resolves as soon as the first promise in the input array settles, whether it's resolved or rejected. The returned promise adopts the outcome (either resolve or reject) of the first settled promise.


**Promise.any:** Resolves as soon as any one of the input promises resolves. It doesn't matter if the other promises reject. The returned promise adopts the resolved value of the first resolved promise.

## What is DOM?

- DOM stands for Document Object Model. DOM is a programming interface for HTML and XML documents.
- When the browser tries to render an HTML document, it creates an object based on the HTML document called DOM. Using this DOM, we can manipulate or change various elements inside the HTML document.

## Rest Operator


javascript

 Copy code

```
function sum(...numbers) {  
  return numbers.reduce((acc, curr) => acc + curr, 0);  
}  
  
console.log(sum(1, 2, 3, 4)); // Output: 10
```

## Spread Operator

javascript

 Copy code

```
const arr1 = [1, 2, 3];  
const arr2 = [4, 5, 6];  
const combined = [...arr1, ...arr2];  
console.log(combined); // Output: [1, 2, 3, 4, 5, 6]
```

## What is the difference between null and undefined?


**null:** Explicitly represents an empty or non-existent value.

**undefined:** Indicates that a variable has been declared but not yet assigned a value.

## What are generators in JavaScript?

Generators are functions that can be paused and resumed, allowing you to manage asynchronous operations more efficiently. They are defined with an asterisk `*` and use the `yield` keyword.

javascript


 Copy code

```
function* generatorFunction() {  
  yield 1;  
  yield 2;  
  yield 3;  
}  
  
const generator = generatorFunction();  
console.log(generator.next().value); // Output: 1  
console.log(generator.next().value); // Output: 2  
console.log(generator.next().value); // Output: 3
```

## Explain the concept of prototype in JavaScript

In JavaScript, every function and object has a prototype property, which is used to implement inheritance. The prototype property allows you to add properties and methods to objects.

javascript

 Copy code

```
function Person(name) {  
  this.name = name;  
}  
  
Person.prototype.greet = function() {  
  console.log(`Hello, my name is ${this.name}`);  
};  
  
const john = new Person('John');  
john.greet(); // Output: Hello, my name is John
```

## New Features in ES6

- [The let keyword](#)
- [The const keyword](#)
- [Arrow Functions](#)
- [The ... Operator](#)
- [For/of](#)
- [Map Objects](#)