JavaScript

Memory Management & Garbage Collection: -

Memory management in JavaScript is the process of allocating and freeing memory during the execution of a program. The JavaScript engine automatically handles memory allocation and garbage collection.

Memory Life Cycle: -

Allocation - Memory is allocated when variables, objects, and arrays are created.

Usage - The allocated memory is used for calculations, operations, and storing values.

Deallocation - When the memory is no longer needed, the JavaScript engine automatically frees it.

Memory Allocation in JavaScript: -

1.Primitive Values (Stored in Stack Memory)

```
let x = 10; // Number

let str = "Hello"; // String

let isActive = true; // Boolean
```

They are immutable, meaning their values cannot be changed.

2.Objects and Functions (Stored in Heap Memory)

```
let person = {
   name: "Alice",
   age: 25
};
function greet() {
  let message = "Hello"; // Memory allocated for 'message'
  console.log(message);
}
greet(); // Once execution is complete, 'message' is removed from memory.
```

The stack contains references (pointers) to these objects in the heap.

Garbage Collection in JavaScript: -

Garbage collection is the process of automatically reclaiming memory that is no longer in use. JavaScript uses automatic garbage collection with algorithms such as Mark-and-Sweep.

Mark-and-Sweep Algorithm: -

It marks all objects that are still accessible.

Unreachable objects are considered garbage and swept away (deallocated).

Example: Automatic Garbage Collection

```
function createUser() {
  let user = {
    name: "Bob",
    age: 30
  };
  return user;
}
let newUser = createUser(); // user object is accessible
newUser = null; // The object is no longer reachable, eligible for garbage collection
```

Common Memory Leaks in JavaScript: -

Even though JavaScript has automatic garbage collection, memory leaks can still occur.

1.Unintentional Global Variables

```
function test() {
  myVar = "I am global"; // No 'var', 'let', or 'const' (creates an implicit global variable)
}
test();
```

This creates a variable in the global scope, preventing it from being garbage-collected.

2.Unused References

```
let obj = { name: "Alice" };
obj = null; // Removing reference allows garbage collection
```

3.Closures Holding Unnecessary References

```
function outer() {
  let largeArray = new Array(1000000); // Large memory allocation
  return function inner() {
    console.log("Closure function");
  };
}
let myFunc = outer();
```

```
// largeArray is still in memory because 'myFunc' retains a reference to 'outer'
```

Solution: Explicitly nullify unused references inside closures.

Best Practices for Efficient Memory Management: -

Use Local Variables - Variables declared inside functions are cleared when the function execution ends.

Avoid Unnecessary Global Variables - Global variables stay in memory throughout the page's lifetime.

Nullify References - Explicitly set objects to null when they are no longer needed.

Functional Programming in JavaScript: -

Functional programming (FP) is a programming paradigm where functions are treated as first-class citizens and emphasize pure functions, immutability, and higher-order functions.

Core Concepts of Functional Programming: -

1.First-Class Functions

Functions can be assigned to variables, passed as arguments, and returned from other functions.

```
const sayHello = function() {
   return "Hello, World!";
};
console.log(sayHello());
```

2.Higher-Order Functions (HOFs)

Functions that take other functions as arguments or return functions.

```
function operate(a, b, operation) {
   return operation(a, b);
}
function add(x, y) {
   return x + y;
}
console.log(operate(5, 3, add)); // Output: 8
```

3. Pure Functions

A function is pure if it always produces the same output for the same input and has no side effects.

```
function square(n) {
    return n * n;
}
console.log(square(4)); // Output: 16
```

4. Immutability

Data should not be modified after creation; instead, new data should be created.

```
const numbers = [1, 2, 3];
const newNumbers = [...numbers, 4]; // Creating a new array
console.log(newNumbers); // [1, 2, 3, 4]
```

5. Function Composition

Combining multiple functions to create a new function.

```
const toUpperCase = str => str.toUpperCase();
const addExclamation = str => str + "!";
const excite = str => addExclamation(toUpperCase(str));
console.log(excite("hello")); // Output: "HELLO!"
```

Common Functional Programming Methods in JavaScript: -

1.map (Transforms an array by applying a function to each element)

```
const nums = [1, 2, 3, 4];
const squared = nums.map(num => num * num);
console.log(squared); // [1, 4, 9, 16]
```

2.filter (Filters elements based on a condition)

```
const words = ["apple", "banana", "cherry", "date"];
const longWords = words.filter(word => word.length > 5);
console.log(longWords); // ["banana", "cherry"]
```

3.reduce(Accumulates values into a single result)

```
const numbersArray = [1, 2, 3, 4];
const sum = numbersArray.reduce((acc, num) => acc + num, 0);
console.log(sum); // Output: 10
```

4.forEach(Iterates over an array without returning a new array)

```
const fruits = ["apple", "banana", "cherry"];
fruits.forEach(fruit => console.log(fruit));
```

5.every & some (Check conditions on array elements)

```
const nums2 = [2, 4, 6, 8];

console.log(nums2.every(num => num % 2 === 0)); // true (all are even)

console.log(nums2.some(num => num > 5)); // true (some are greater than 5)
```

Currying in Functional Programming: -

Transforming a function with multiple arguments into a series of functions, each taking a single argument.

Normal Function (Without Currying)

```
function multiply(a, b) {
   return a * b;
}
console.log(multiply(2, 3)); // Output: 6
```

Curried Version

```
function multiplyCurried(a) {
   return function (b) {
      return a * b;
   };
}
const double = multiplyCurried(2); // Returns a function
console.log(double(3)); // Output: 6
console.log(double(5)); // Output: 10
```

Arrow Function Syntax for Currying

```
const multiply = a => b => a * b;
const triple = multiply(3);
console.log(triple(4)); // Output: 12
```

It helps create reusable functions (double, triple, etc.).

It makes function composition easier in functional programming.