DAY 1

1. `let`:
    - Use `let` when the value of the variable can change.
      Example:

```
let num = 10.5; // Create a variable with value 10.5
console.log(num); // Prints 10.5
let name = "Rohit"; // Create a variable with value "Rohit"
name = "Mohit"; // Change the value to "Mohit"
console.log(name); // Prints "Mohit"
```

2.
3. `const`:
    - Use `const` when the value should not change.
      Example:

```
const id = 20; // Create a constant with value 20
console.log(id); // Prints 20
// id = 30; // This will cause an error because `const` values can't change
```

4.
5. `var`:
    - An old way to declare variables. Avoid it because it can behave unpredictably.
      Example:

```
var x = 10; // Create a variable with value 10
x = 20; // Change the value to 20
console.log(x); // Prints 20
```

6.

Key point:

- Use `let` and `const` for modern JavaScript.
- `const` is for things that don’t change, and `let` is for things that do.
-

DAY 2

Explanation of Primitive and Non-Primitive Data Types in JavaScript

---

Primitive Data Types

These are the most basic data types in JavaScript. They hold only a single value and are immutable (cannot be changed).

Number:
Used to represent numerical values.
Example:

```
let account_balance = 30;
console.log(account_balance); // Output: 30
```

1.

String:
Used for text or characters. Strings are wrapped in single (`'`) or double (`"`) quotes.
Example:

```
let str = "rohit negi is a bad boy, he doesn't know how to use zoom";
console.log(str); // Output: The full sentence
```

2.

Boolean:
Represents either `true` or `false`.
Example:

```
let Papa_ko_block_kara_hai = false;
console.log(Papa_ko_block_kara_hai); // Output: false
console.log(typeof Papa_ko_block_kara_hai); // Output: boolean
```

3.

Undefined:
A variable is `undefined` when it is declared but not assigned any value.
Example:

```
let account;
```

```
console.log(account); // Output: undefined
```

4.

Null:
Represents an intentional absence of value. It's an object type (quirk of JavaScript).
Example:

```
let bal = null;
console.log(typeof bal); // Output: object
```

5.

BigInt:
Used to represent very large integers beyond the range of `Number`. Append `n` to the value.
Example:

```
let a = 434314783612479182374983 2n;
console.log(a); // Output: 434314783612479182374983 2n
console.log(Number.MAX_SAFE_INTEGER); // Largest safe number
console.log(Number.MIN_SAFE_INTEGER); // Smallest safe number
```

6.

Non-Primitive Data Types

These are more complex types that can hold multiple values or functionalities.

Array:
A collection of items stored in a single variable. Each item has an index starting from 0.
Example:

```
let fruits = ["apple", "banana", "cherry"];
console.log(fruits); // Output: ["apple", "banana", "cherry"]
console.log(fruits[1]); // Output: banana
```

1.

Object:
A collection of key-value pairs, like a dictionary.
Example:

```
let person = {
  name: "Rohit",
```

```
    age: 25,
    isStudent: true
};
console.log(person); // Output: { name: "Rohit", age: 25, isStudent: true }
console.log(person.name); // Output: Rohit
```

2.

Function:
A block of code designed to perform a task. Functions can take input (parameters) and return a result.
Example:

```
function greet(name) {
    return "Hello, " + name + "!";
}
console.log(greet("Rohit")); // Output: Hello, Rohit!
```

3.

Key Difference:

- **Primitive types** store only a single value (immutable).
- **Non-primitive types** store collections of values or functions (mutable).

DAY 3

Data Types Recap

1. **Primitive Data Types**: Simple values like:
    - `Number`: Numbers like 10, 20, 50.
    - `String`: Text like `"rohit"` or `'hello'`.
    - `Boolean`: `true` or `false`.
    - `Null`: Represents "nothing."
    - `Undefined`: When a variable is declared but not assigned any value.
    - `BigInt`: For very large numbers.

- `Symbol`: A unique value used for specific purposes.

2. **Non-Primitive Data Types**: Complex types like:

Array: A list of items.
Example:
```
let arr = [10, 20, 50, "rohit", "mohit"];
console.log(typeof arr); // Output: object
```

-

Object: A collection of data in `key: value` format.
Example:
```
let obj = {
   user_name: "Rohit",
   account_number: 31242314213,
   balance: 420
};
console.log(obj);
```

-

Function: A reusable block of code.
Example:
```
let fun = function() {
   console.log("Hello Coder army");
   return 10;
};
console.log(fun()); // Output: Hello Coder army, 10
```

-

Type Conversion

String to Number:
Use `Number()` to convert a string to a number.
Example:

```
let account_balance = "100";
let num = Number(account_balance);
console.log(typeof num); // Output: number
```

1.

2. **Boolean to Number**:

    - `false` becomes `0`.
    - `true` becomes `1`.
      Example:

```
let x = false;
console.log(Number(x)); // Output: 0
```

3.

String That Can't Be a Number:
If the string contains non-numeric characters, it returns `NaN` (Not a Number).
Example:

```
let account = "100xs";
console.log(Number(account)); // Output: NaN
```

4.

Null to Number:
`null` becomes `0`.
Example:

```
let x1 = null;
console.log(Number(x1)); // Output: 0
```

5.

Undefined to Number:
`undefined` becomes `NaN`.
Example:

```
let x2;
console.log(Number(x2)); // Output: NaN
```

6.

Number to String:
Use `String()` to convert numbers or other types to strings.
Example:

```
let ab = 20;
console.log(String(ab)); // Output: "20"
```

7.

Boolean to String:
Example:

```
let ax = false;
console.log(String(ax)); // Output: "false"
```

8.
9. **String to Boolean**:
    - Empty strings (`""`) are `false`.
    - Non-empty strings (`"hello"`) are `true`.
      Example:

```
let abc = " ";
console.log(Boolean(abc)); // Output: true
```

10.

Math Operations

1. **Order of Operations**:
    - Multiplication and Division are done first, left to right.
    - Then Addition and Subtraction, left to right.
      Example:

```
console.log((((6 * (3 + 18)) / 6) - 9)); // Output: 18
```

2.

Modulo Operator (`%`):
Gives the remainder after division.
Example:

```
console.log(20 % 3); // Output: 2
```

3.
4. **Increment/Decrement**:
    - `++` adds 1.
    - `--` subtracts 1.

- Pre-increment (`++sum`): Increases value first, then uses it.
- Post-increment (`sum++`): Uses value first, then increases it.
  Example:

```
let sum = 20;
let total = ++sum; // Pre-increment
console.log(total); // Output: 21
console.log(sum); // Output: 21
```

5.

Assignment Operators

Used to update the value of a variable.
Example:

```
let x = 20;
x += 10; // Same as x = x + 10
console.log(x); // Output: 30
x /= 10; // Same as x = x / 10
console.log(x); // Output: 3
```

-

Summary:

- Understand **data types** (primitive and non-primitive).
- Practice **type conversion** to switch between types.
- Learn **math operators** for calculations.
- Use **assignment operators** to update variables easily.

DAY 4

Explanation of Comparison and Bitwise Operators in Simple Language

---

Comparison Operators

Used to compare two values. Results are `true` or `false`.

1. **`==` (Equality)**: Checks if the values are the same.
    - Converts types if needed.
      Example:

```
let a1 = 10;
let str1 = "10";
console.log(a1 == str1); // Output: true (converts "10" to 10)
```

2. **`===` (Strict Equality)**: Checks if both value **and type** are the same.
   Example:

```
let a1 = 10;
let str1 = "10";
console.log(a1 === str1); // Output: false (different types)
```

3.
4. **Other Comparisons**:
    - `<` (less than), `>` (greater than)
    - `<=` (less than or equal to), `>=` (greater than or equal to)
      Example:

```
console.log(5 < 10); // Output: true
console.log(20 >= 20); // Output: true
```

5.
6. **Null and Undefined**:
    - `null == undefined` is `true` because they both mean "no value."
    - `null === undefined` is `false` because they are different types.
      Example:

```
console.log(null == undefined); // Output: true
console.log(null === undefined); // Output: false
```

7.

Special Cases with Null:
Example:

```
console.log(null == 0); // Output: false
console.log(null < 0); // Output: false
console.log(null >= 0); // Output: true
```

8.

Undefined Comparison:
`undefined` doesn't compare with numbers.
Example:

```
console.log(undefined == 0); // Output: false
console.log(undefined < 0); // Output: false
```

9.
10. **NaN (Not a Number)**:

    - `NaN` is not equal to anything, even itself.
      Example:

```
console.log(NaN == NaN); // Output: false
```

11.

Logical Operators

Combine multiple conditions.

1. **`&&` (AND)**:

    - Both conditions must be true.
      Example:

```
let age = 18;
let money = 420;
console.log(age < 18 && money > 200); // Output: false
```

2.
3. `||` **(OR)**:

    - At least one condition must be true.
      Example:

```
console.log(age > 10 || money > 200); // Output: true
```

4.
5. **!** **(NOT)**:

    - Reverses the result.
      Example:

```
console.log(!(age > 10)); // Output: false
```

6.

Bitwise Operators

Work with binary numbers.

- Binary: Numbers written as 0s and 1s. For example, `5` in binary is `101`.

& **(AND)**:
Compares each bit and returns `1` if both bits are `1`.
Example:

```
console.log(4 & 5); // Output: 4
// 4 -> 100
// 5 -> 101
// Result -> 100 (4 in decimal)
```

1.

| **(OR)**:
Compares each bit and returns `1` if at least one bit is `1`.
Example:

```
console.log(11 | 14); // Output: 15
// 11 -> 1011
// 14 -> 1110
// Result -> 1111 (15 in decimal)
```

2.

^ **(XOR)**:
Returns `1` if bits are different, `0` if they are the same.
Example:

```
console.log(5 ^ 7); // Output: 2
```

```
// 5 -> 101
// 7 -> 111
// Result -> 010 (2 in decimal)
```

3.

<< (Left Shift):
Shifts bits to the left by a specified number of positions, filling with `0`.
Example:

```
console.log(5 << 3); // Output: 40
// 5 -> 101
// Shift left 3 -> 101000 (40 in decimal)
```

4.

>> (Right Shift):
Shifts bits to the right by a specified number of positions.
Example:

```
console.log(20 >> 2); // Output: 5
// 20 -> 10100
// Shift right 2 -> 101 (5 in decimal)
```

Summary:

- Use **comparison operators** to compare values.
- Use **logical operators** to combine conditions.
- Use **bitwise operators** to perform operations at the binary level.

Day 05

Primitive Data Types

Code:

```
 let a = 10;
let b = a;
b = 30;
console.log(b); // 30
console.log(a); // 10
```

1.
2. **Explanation:**

    - **Primitive data types** (like `number`, `string`, `boolean`, etc.) are **immutable** and stored by value.
    - When you assign `b = a`, the value of `a` (which is `10`) is copied into `b`. After this, `b` and `a` are completely independent variables.
    - Changing `b` to `30` does not affect `a`. So:
        - `console.log(b)` prints `30`.
        - `console.log(a)` prints `10`.

Non-Primitive Data Types (Objects)

Code:

```
 let obj1 = {
    id: 20,
    naming: "rohit"
};

let obj2 = obj1;

obj2.id = 30;

console.log(obj1); // { id: 30, naming: "rohit" }
console.log(obj2); // { id: 30, naming: "rohit" }
```

1.
2. **Explanation:**

    - **Non-primitive data types** (like objects and arrays) are **mutable** and stored by reference.
    - When you assign `obj2 = obj1`, you're copying the reference (or memory address) of `obj1` into `obj2`. Now both `obj1` and `obj2` point to the **same object in memory**.
    - Modifying `obj2.id = 30` changes the object in memory. Since `obj1` and `obj2` share the same reference, the change is reflected in both variables.
    - So:
        - `console.log(obj1)` prints `{ id: 30, naming: "rohit" }`.

- `console.log(obj2)` also prints `{ id: 30, naming: "rohit" }`.

Key Takeaways

1. **Primitive Data Types**:
   - Stored **by value**.
   - Independent after assignment.
   - Examples: `number`, `string`, `boolean`, `null`, `undefined`, `symbol`.
2. **Non-Primitive Data Types**:
   - Stored **by reference**.
   - Share the same reference after assignment unless explicitly cloned.
   - Examples: `object`, `array`, `function`.

DAY 06

Primitive vs Non-Primitive Data Types:

1. **Primitive Types:**
   - Immutable and directly hold values.
   - Example: `const num = 10;` (number type)
2. **Non-Primitive Types:**
   - Mutable and hold references to memory locations.

Example:
```
const obj = {

  id: 10,

  balance: 200

};
```

-
- Here, modifying a property (e.g., `obj.id = 11`) works because you're mutating the object.
- Assigning `obj = obj2;` throws an error since `obj` is declared as `const` and cannot be reassigned.

String Manipulations:

1. **String Declaration:**
   - Strings can be enclosed in double quotes (`" "`), single quotes (`' '`), or template literals (`` ` ` ``).
   - Template literals allow interpolation with `${variable}`.
2. **String Concatenation:**

Using `+` to combine strings:
```
let s1 = "hello";

let s2 = " Coder Army";

let s3 = s1 + s2; // Result: "hello Coder Army"
```

-

3. **Escape Characters:**
   - `\n` for a new line.
   - `\\` to include a backslash.
4. **Accessing Characters:**
   - Use bracket notation (`special[4]`) or `.charAt(index)`.

5. **String Methods:**
    - `toLowerCase()` and `toUpperCase()` to change case.
    - `indexOf()` and `lastIndexOf()` to find substrings.
    - `includes()` to check if a substring exists.
    - `slice(start, end)` and `substring(start, end)` to extract parts of a string.
        - `slice()` accepts negative indices.
    - `replace(search, replacement)` and `replaceAll(search, replacement)` to modify substrings.
    - `split(separator)` to split a string into an array.
    - `trim()` to remove whitespace from both ends of a string.
6. **String Properties:**
    - `length` gives the number of characters in the string.
7. **Special Cases:**
    - Strings created with `new String()` are of type `"object"`, not `"string"`.

Sample Outputs:

1. String concatenation:
    - `"hello coder army"`
    - `'hello coder army'`
2. Escape Characters:
    - `Rohit Bhaiya bhut bade badmash hai.`
    - `Wo bhut gande insaan hai.`
3. String methods:
    - `indexOf("Coder")` → 6
    - `lastIndexOf("Coder")` → 17
    - `slice(-6, 5)` → `loD`

This script effectively demonstrates the versatility of strings in JavaScript and the difference between primitive and non-primitive types.

Note

Notes on String Operations in JavaScript

1. Declaring Strings

- Strings can be declared using:
  - **Double quotes**: `"Hello"`
  - **Single quotes**: `'Hello'`
  - **Template literals**: `` `Hello` `` (allows string interpolation: `${expression}`)
- **Backslash (`\`)** is used for escape sequences like `\n` (new line) or `\\` (backslash).

---

2. Length of a String

`length`: Returns the number of characters in a string.

```
 let str = "Hello";
console.log(str.length); // Output: 5
```

-

---

3. Accessing Characters

- **Bracket Notation**: `str[index]`

`charAt(index)`: Returns the character at the specified index.

```
 let str = "Hello";
console.log(str[1]);        // Output: e
console.log(str.charAt(1)); // Output: e
```

---

4. Changing Case

- `toUpperCase()`: Converts all characters to uppercase.

`toLowerCase()`: Converts all characters to lowercase.

```
 let str = "Hello";
console.log(str.toUpperCase()); // Output: HELLO
console.log(str.toLowerCase()); // Output: hello
```

-

---

5. Searching in Strings

- `indexOf(substring)`: Finds the first occurrence of a substring; returns `-1` if not found.
- `lastIndexOf(substring)`: Finds the last occurrence of a substring.

`includes(substring)`: Checks if a substring exists in the string.

```
 let str = "Hello Coder";
console.log(str.indexOf("Coder"));       // Output: 6
console.log(str.lastIndexOf("o"));      // Output: 7
console.log(str.includes("Hello"));     // Output: true
```

-

---

6. Extracting Substrings

- **`slice(start, end)`**: Extracts part of a string, accepts negative indexes.
- **`substring(start, end)`**: Similar to `slice` but doesn't accept negative indexes.

`substr(start, length)`: Extracts a substring of the specified length.

```
 let str = "Hello World";
console.log(str.slice(0, 5));        // Output: Hello
console.log(str.substring(0, 5));   // Output: Hello
console.log(str.substr(6, 5));      // Output: World
```

-

---

7. Replacing Content

- **`replace(oldSubstring, newSubstring)`**: Replaces the first match.

`replaceAll(oldSubstring, newSubstring)`: Replaces all matches (requires ES2021+).

```
 let str = "Hello Hello";
console.log(str.replace("Hello", "Hi"));        // Output: Hi Hello
console.log(str.replaceAll("Hello", "Hi"));     // Output: Hi Hi
```

---

8. Splitting Strings

`split(delimiter)`: Splits a string into an array based on the delimiter.

```
 let str = "Hello,World,JavaScript";
```

```
console.log(str.split(",")); // Output: ["Hello", "World", "JavaScript"]
```

---

9. Trimming Strings

- `trim()`: Removes whitespace from both ends.

`trimStart()` / `trimEnd()`: Removes whitespace from the start or end, respectively.

```
 let str = "  Hello  ";
console.log(str.trim());        // Output: "Hello"
console.log(str.trimStart());   // Output: "Hello  "
console.log(str.trimEnd());     // Output: "  Hello"
```

DAY 07

JavaScript Numbers and Objects

Primitive and Object Comparison:

```
 let num1 = 231; // Primitive number

let num2 = new Number(231); // Number object

let num3 = new Number(231); // Another Number object


console.log(num1 == num2); // true: num2 is converted to a primitive

console.log(num2 == num3); // false: different object references
```

1. **Number Object Details:**

```
console.log(num2); // Logs Number object {231}
console.log(typeof num2); // "object"
```

2.

Number Methods

`toFixed`: Rounds to the specified decimal places.

```
let num = 231.68;
console.log(num.toFixed(3)); // "231.680" (3 decimal places)
```

1.

`toPrecision`: Specifies total significant digits.

```
console.log(num.toPrecision(4)); // "231.7" (4 significant digits)
```

2.

`toExponential`: Converts to scientific notation.

```
console.log(num.toExponential(2)); // "2.32e+2"
```

3.

`toString` and `valueOf`:

```
console.log(typeof num.toString()); // "string"
console.log(num.valueOf()); // 231.68 (primitive value)
```

4.

Math Object

Constants:

```
 console.log(Math.E); // Euler's number

console.log(Math.LN10); // Natural logarithm of 10

console.log(Math.PI); // 3.14159...

console.log(Math.LOG10E); // Base-10 logarithm of Euler's number
```

1.

`Math.floor` and `Math.ceil`:

```
 let num1 = 23.1;

console.log(Math.floor(num1)); // 23: Rounds down

console.log(Math.ceil(num1)); // 24: Rounds up
```

Random Number Generation

Generate a Random Integer Between 0-9:

```
 console.log(Math.floor(Math.random() * 10));
```

1.
2. **Generate Between Specific Ranges:**

1-10:

```
 console.log(Math.floor(Math.random() * 10) + 1);
```

-

11-20:

```
 console.log(Math.floor(Math.random() * 10) + 11);
```

Between `min` and `max` (e.g., 40-50):

```
 let min = 40, max = 50;

console.log(Math.floor(Math.random() * (max - min + 1) + min));
```

    -
3. **Custom Ranges:**

2-12:
```
console.log(Math.floor(Math.random() * 11) + 2);
```
    -

30-40:
```
console.log(Math.floor(Math.random() * (40 - 30 + 1) + 30));
```
    -

Ludo Dice Roll (1-6):

```
console.log(Math.floor(Math.random() * (6 - 1 + 1) + 1));
```

Summary Notes

- **Primitive vs. Object:** `==` compares values after converting objects to primitives. `===` does not.
- **Number Methods:**
    - `toFixed` rounds to fixed decimal places.
    - `toPrecision` focuses on significant digits.
    - `toExponential` converts to scientific notation.
- **Random Generation:**
    - `Math.random()` generates values between 0 (inclusive) and 1 (exclusive).
    - Scaling with multiplication adjusts the range.
    - Adding offsets shifts the range.
    - **Useful Math Constants:** `Math.PI`, `Math.E`, etc.

DAY 08

Accessing Array Elements:

```
 const arr = [2, 35, 1, 8, 9, "rohit", true, 8];
console.log(arr[1]); // Outputs: 35 (Element at index 1)
console.log(arr.at(-2)); // Outputs: true (Second last element)
```

1.
    - `at` method allows you to use negative indices to access elements from the end of an array.

Array Length:

```
 console.log(arr.length); // Outputs: 8 (Number of elements in the array)
```

2.

Cloning Arrays:

```
 const newarr = structuredClone(arr);
console.log(newarr == arr); // Outputs: false
```

3.
    - `structuredClone` creates a deep copy of the array.
    - `==` checks if both arrays are the same object in memory, which is false here since they are different objects.

---

4. **Array Operations:**

Push: Adds elements at the end.

```
 arr.push(30);
arr.push(50);
console.log(arr); // Outputs: [2, 35, 1, 8, 9, "rohit", true, 8, 30, 50]
```

    -

Pop: Removes the last element.

```
 arr.pop();
console.log(arr); // Removes and outputs: [2, 35, 1, 8, 9, "rohit", true, 8, 30]
```

-

Unshift: Adds elements to the beginning.

```
 arr.unshift(10);
console.log(arr); // Outputs: [10, 2, 35, 1, 8, 9, "rohit", true, 8, 30]
```

-

Shift: Removes the first element.

```
 arr.shift();
console.log(arr); // Outputs: [2, 35, 1, 8, 9, "rohit", true, 8, 30]
```

-

Delete Operation:

```
 delete arr[0];
console.log(arr); // Outputs: [empty, 35, 1, 8, 9, "rohit", true, 8, 30]
```

5.
    - The `delete` operator removes the element but leaves a "hole" (empty space) in the array.

Searching:

```
 console.log(arr.indexOf(8)); // Outputs: First index of 8 (3)
console.log(arr.lastIndexOf(8)); // Outputs: Last index of 8 (7)
console.log(arr.includes(10)); // Outputs: false (10 is not in the array)
```

6.

---

7. **Slicing and Splicing:**

Slice: Extracts part of an array (non-destructive).

```
 let a = arr.slice(2, 5);
console.log(a); // Outputs: [1, 8, 9]
console.log(arr); // Original array remains unchanged.
```

-

Splice: Removes or adds elements (destructive).

```
 let newsplice = arr.splice(2, 5);
console.log(newsplice); // Outputs: Removed elements [1, 8, 9, "rohit", true]
console.log(arr); // Outputs: Remaining elements [2, 35, 8, 30]
arr.splice(2, 0, "money", 90); // Adds elements at index 2
console.log(arr); // Outputs: [2, 35, "money", 90, 8, 30]
```

-

Joining and Converting Arrays:

```
 console.log(arr.toString()); // Outputs: Array as a comma-separated string
console.log(arr.join("*")); // Outputs: String with "*" as a separator
```

8.

Concatenation:

```
 let arr3 = arr1.concat(arr2, arr4);
console.log(arr3); // Outputs: Merged array of arr1, arr2, and arr4
```

9.

Flattening a 2D Array:

```
 let arr2d = [[1, 2, 3, [23, 432, 123, [331, 123, 123]]], [4, 5, 6], [7, 8, 9]];
let newarr = arr2d.flat(3); // Flatten array to 3 levels deep
console.log(newarr); // Outputs a 1D array
```

10.

Checking if a Variable is an Array:

```
 console.log(Array.isArray(abc)); // Outputs: true (abc is an array)
```

Creating a New Array with a Fixed Length:

```
 let ac = new Array(10);
```

console.log(ac.length); // Outputs: 10 (Array has space for 10 elements, all empty)

12.
    - `new Array(10)` creates an array with 10 empty slots (undefined values).

Note

Here's an explanation of each concept in the list you provided:

Array Methods and Properties:

1. **`length`:**
    - Represents the number of elements in an array.

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

    -

2. **`push()`:**
    - Adds an element to the end of an array.

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

    -

3. **`pop()`:**

- Removes the last element from an array and returns it.

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

-

4. **`unshift()`:**

    - Adds an element to the beginning of an array.

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

-

5. **`shift()`:**

    - Removes the first element from an array and returns it.

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

-

6. **`indexOf()`:**

    - Returns the index of the first occurrence of a specified element, or `-1` if the element is not found.

Example:

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

7. **`includes()`:**
    - Checks if an array contains a specified element.

Example:

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

8. **`slice()`:**
    - Extracts a portion of an array without modifying the original array.

Example:

```
 const arr = [1, 2, 3, 4];
console.log(arr.slice(1, 3)); // Output: [2, 3]
console.log(arr); // Output: [1, 2, 3, 4]
```

9. **`splice()`:**
    - Modifies an array by adding, removing, or replacing elements.

Example:

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

```
arr.splice(1, 2, 5, 6);
console.log(arr); // Output: [1, 5, 6, 4]
```

10. **`toString()`:**
    - Converts an array to a comma-separated string.

Example:
```
 const arr = [1, 2, 3];
console.log(arr.toString()); // Output: "1,2,3"
```

11. **`at()`:**
    - Introduced in ES2022, it accesses elements using positive or negative indices.

Example:
```
 const arr = [1, 2, 3];
console.log(arr.at(-1)); // Output: 3
```

12. **`join()`:**
    - Joins all elements of an array into a string with a specified separator.

Example:
```
 const arr = [1, 2, 3];
console.log(arr.join("-")); // Output: "1-2-3"
```

13. **`concat()`:**
    - Creates a new array by merging two or more arrays.

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

    -

14. **`flat()`:**
    - Creates a new array by flattening sub-arrays to a specified depth.

Example:
```
 const arr = [1, [2, [3, [4]]]];
console.log(arr.flat(2)); // Output: [1, 2, 3, [4]]
```

    -

Creating New Arrays:

1. **`Array.isArray()`:**
    - Checks if a variable is an array.

Example:
```
 const arr = [1, 2, 3];
console.log(Array.isArray(arr)); // Output: true
```

    -

2. **Creating Empty Arrays:**

Using `new Array(length)`:

```
 const arr = new Array(10);
console.log(arr); // Output: [empty × 10]
console.log(arr.length); // Output: 10
```

-
- Creates an array of a specified length, with all elements as `undefined`.

DAY 09

Working with `Date` Object

1. **Creating Date Objects:**

`new Date()` creates a `Date` object with the current date and time.

```
 const d = new Date();
console.log(d.toString()); // Outputs current date in string format
```

You can also create a `Date` object for a specific date/time:

```
 const specificDate = new Date("2022-10-20");
console.log(specificDate.toString()); // Outputs: Thu Oct 20 2022
```

`new Date(year, month, day, hours, minutes, seconds, milliseconds)` lets you specify all components of a date.

```
 const date = new Date(2024, 5, 28, 10, 12, 45, 231);
```

console.log(date.toString()); // Outputs: Fri Jun 28 2024 10:12:45 GMT+...

---

2. **Date Formatting:**

`toDateString()`: Outputs only the date part as a string.
console.log(d.toDateString()); // Outputs: Mon Dec 14 2024

`toISOString()`: Converts date to ISO 8601 format.
console.log(d.toISOString()); // Outputs: 2024-12-14T05:30:00.000Z

---

3. **Retrieving Components of a Date:**
   - `getDate()`: Day of the month (1-31).
   - `getDay()`: Day of the week (0 = Sunday, 6 = Saturday).
   - `getMonth()`: Month (0 = January, 11 = December).
   - `getFullYear()`: Full year (e.g., 2024).
   - `getMilliseconds()`: Milliseconds of the second (0-999).
   - `getMinutes()`: Minutes of the hour (0-59).
   - `getTime()`: Number of milliseconds since January 1, 1970.

---

4. **Modifying Date Components:**

Use `setDate()`, `setFullYear()`, and `setMonth()` to modify the `Date` object.

```
const d = new Date();
d.setDate(20);
d.setFullYear(2021);
d.setMonth(3); // 0 = Jan, 3 = Apr
console.log(d.toLocaleString()); // Outputs modified date
```

5. **Date Difference:**

Subtracting two `Date` objects gives the difference in **milliseconds**.

```
 const date1 = new Date();
const date2 = new Date("2025-04-21");
console.log(date2 - date1); // Outputs difference in milliseconds
```

---

6. **Countdown Timer Example:**
    - **Steps:**
        1. Calculate the difference between two dates in milliseconds.
        2. Convert milliseconds into days, hours, minutes, and seconds using mathematical operations.

Code:

```
 const date1 = new Date();
const date2 = new Date("2028-07-14T00:00:00");

const dateDiff = date2 - date1; // Difference in milliseconds

const days = Math.floor(dateDiff / (1000 * 60 * 60 * 24)); // Convert to days
const hours = Math.floor((dateDiff / (1000 * 60 * 60)) % 24); // Remaining hours
const minutes = Math.floor((dateDiff / (1000 * 60)) % 60); // Remaining minutes
const seconds = Math.floor((dateDiff / 1000) % 60); // Remaining seconds

console.log(`Olympics Countdown Time: Days: ${days}, Hours: ${hours}, Minutes:
${minutes}, Seconds: ${seconds}`);
```

Output:

```
 Olympics Countdown Time: Days: 1310, Hours: 5, Minutes: 30, Seconds: 45
```

Notes:

1. `Date.now()`:
    - Returns the current timestamp in milliseconds since 1st January 1970.

Example:
const now = Date.now();
console.log(now); // Outputs a large millisecond number

    -
2. **Date Arithmetic:**
    - Dates are represented as the number of milliseconds since the epoch (1st January 1970). This makes it easy to perform arithmetic on dates.
3. **Key Points:**
    - Month is **0-based** in the `Date` object (January = 0, December = 11).
    - Day and date components are **1-based** (1st = 1).

DAY 10

---

1. Creating Objects

Method 1: Object Literals

```
const obj = {
    0: 20,
    1: 50,
    2: 70,
    name: "rohit",
    account_balance: 420,
    gender: "Male",
    age: 30,
    "account number": 231230,
    undefined: 30,
    null: "mohan",
};
```

- **Object literal syntax** is the easiest and most common way to create objects.
- Properties can include:
    - **Numbers** (treated as string-like keys): e.g., `0`, `1`.
    - **Strings** (with quotes if spaces are in the key): e.g., `"account number"`.
    - **Special types** like `undefined` or `null` as keys.

Accessing Object Properties

```
console.log(obj["undefined"]); // 30
console.log(obj["null"]); // "mohan"
console.log(obj.gender); // "Male"
console.log(obj["account_balance"]); // 420
console.log(obj["account number"]); // 231230
console.log(obj['0']); // 20
console.log(obj[1]); // 50
console.log(obj[2]); // 70
console.log(obj);
```

- Access object properties:
    - **Dot notation**: `obj.gender`.
    - **Bracket notation**: `obj["account number"]`.

Arrays

```
const arr = [20, 50, 70];
console.log(arr);
```

- Arrays are objects where keys are numeric indices.

---

2. Second Method: `new Object()`

```
const person = new Object();
person.name = "Rohit";
person.age = 80;
person.gender = "Male";
console.log(person);

// Deleting properties
```

```
delete person.age;
console.log(person);

// Modifying properties
person.name = "Mohit";
console.log(person);
```

- Using `new Object()` creates an empty object.
- **Properties can be added, deleted, or modified** dynamically.

---

3. Third Method: Classes

```
class People {
   constructor(na, ag, gen) {
      this.name = na;
      this.age = ag;
      this.gender = gen;
   }
}

let per1 = new People("Rohit", 20, "Male");
let per2 = new People("Mohit", 30, "Female");
let per3 = new People("Aman", 21, "Male");
console.log(per1, per2);
```

- Classes are a blueprint for creating objects.
- Use the `constructor` to initialize properties when creating instances.

---

4. Object Methods

`Object.values()`

```
const arr = Object.values(obj);
console.log(arr);
// Output: ["rohit", 30, 420, "male"]
```

- Extracts and returns an array of **values** of an object.

`Object.entries()`

```
const arr2 = Object.entries(obj);
console.log(arr2);
// Output: [["name", "rohit"], ["age", 30], ["account_balance", 420], ["gender", "male"]]
```

- Returns an array of key-value pairs.

`Object.assign()`

```
const obj1 = {a: 1, b: 2};
const obj2 = {c: 3, d: 4};
const obj4 = {e: 5, f: 6};

const obj3 = Object.assign({}, obj1, obj2, obj4);
console.log(obj3);
// Output: {a: 1, b: 2, c: 3, d: 4, e: 5, f: 6}
```

- Merges objects into a new object. The first argument is the target object.

Spread Syntax (`...`)

```
const obj5 = {...obj1, ...obj2, ...obj4};
console.log(obj5);
// Output: {a: 1, b: 2, c: 3, d: 4, e: 5, f: 6}
```

- An alternative to `Object.assign()` to merge objects.

Notes

1. **Object Creation Methods**:
    - Object literals `{key: value}`.
    - `new Object()`.
    - Using `class` and `constructor`.

2. **Accessing Object Properties**:
    - Dot notation: `obj.key`.
    - Bracket notation: `obj["key"]`.
3. **Object Methods**:
    - `Object.values()`: Get all values as an array.
    - `Object.entries()`: Get key-value pairs as an array of arrays.
    - `Object.assign()`: Merge objects.
    - Spread syntax (`...`): Concise way to merge objects.
4. **Dynamic Object Manipulation**:
    - Adding, deleting, and updating properties.

DAY 11

Here is a detailed explanation of the code:

---

1. Object Shallow and Deep Copy

Code:

```
let obj1 = { a: 1, b: 2 };
let obj2 = obj1; // Shallow copy
obj2.a = 10;
console.log(obj2, obj1); // obj2 and obj1 share the same reference
```

- **Shallow Copy:** `obj2` is a reference to `obj1`. Changing `obj2.a` also modifies `obj1.a` since they both point to the same object.

Code:

```
let obj3 = structuredClone(obj1); // Deep copy
obj3.a = 20;
console.log(obj3, obj1); // obj3 is independent of obj1
```

- **Deep Copy:** `structuredClone` creates an entirely new copy of the object. Changing `obj3.a` does not affect `obj1`.

---

2. Nested Object and Shallow Copy Issue

Code:

```
const user = {
   name: "Rohit",
   balance: 420,
   address: { pincode: 246149, city: "kotdwar" }
};

const user2 = Object.assign({}, user); // Shallow copy
user2.address.pincode = 321314; // Modifies the original `user.address.pincode`
console.log(user.address.pincode); // Reflects the change
```

- **Explanation:** `Object.assign` only creates a shallow copy. The nested `address` object is still shared between `user` and `user2`.

---

3. Object Destructuring

Code:

```
let obj = {
   name: "Rohit",
   money: 430,
   balance: 30,
   age: 20,
   aadhar: "hfdsiohsai"
};

const { name, balance, age } = obj; // Extract properties
const { name: full_name, balance: amount, age: Umar } = obj; // Rename destructured properties
const { name, age, ...obj1 } = obj; // Rest operator to gather remaining properties
console.log(obj1); // {money: 430, aadhar: "hfdsiohsai"}
```

- **Explanation:** Object destructuring extracts or renames properties. The rest operator gathers remaining properties.

---

4. Array Destructuring

Code:

```
const arr = [3, 2, 1, 5, 10];
const [first, second] = arr; // Extract first and second elements
const [first, second, , third] = arr; // Skip an element
const [first, second, ...third] = arr; // Rest operator gathers remaining elements
console.log(third); // [1, 5, 10]
```

- **Explanation:** Array destructuring extracts elements or gathers remaining elements using the rest operator.

---

5. Nested Destructuring

Code:

```
let obj = {
   name: "Rohit",
   age: 20,
   arr: [90, 40, 60, 80],
   address: { pincode: 246149, city: "Kotdwar", state: "uk" }
};

const { address: { pincode, city } } = obj; // Nested destructuring
const { arr: [first] } = obj; // Destructuring array inside an object
console.log(first); // 90
```

- **Explanation:** You can destructure nested objects or arrays within objects directly.

---

6. Object Methods

Code:

```
let user = {
    name: "Rohit",
    amount: 420,
    greet: function () {
        console.log("Hello Coder Army");
    },
    meet: function () {
        return 20;
    }
};

console.log(user.greet()); // Logs "Hello Coder Army" and returns `undefined`
console.log(user.meet()); // Returns 20
```

- **Explanation:** `greet` logs a message and does not return a value, so `undefined` is logged. `meet` explicitly returns `20`.

---

7. Object `toString` Method

Code:

```
let obj = {
    name: "Rohit",
    amount: 420,
    greet: function () {
        return 10;
    }
};

console.log(obj.toString()); // Default toString implementation for an object
```

- **Explanation:** The default `toString` for objects returns `[object Object]`. To customize it, you can override the `toString` method.

---

8. Array as an Object

Code:

```
let arr = [2, 3, 1, 8];
arr.push(10); // Adds 10 to the end of the array
console.log(arr); // [2, 3, 1, 8, 10]
```

- **Explanation:** Arrays in JavaScript are objects, which is why they can have methods like `.push`. They also have properties like `.length`.

Let's break down the code and explain each part:

---

1. Setting `__proto__` in JavaScript

```
let user1 = {
   name: "Rohit",
   age: 20,
}

let user2 = {
   amount: 20,
   money: 50
}

user2.__proto__ = user1;
```

- **Explanation:**
    - `user1` is an object with properties `name` and `age`.
    - `user2` is another object with properties `amount` and `money`.
    - By setting `user2.__proto__ = user1;`, you are making `user1` the prototype of `user2`. This means `user2` will inherit all the properties and methods from `user1` through its prototype chain.
- **Prototype Chain:**

- When you try to access properties on `user2`, JavaScript will first look for them directly on `user2`. If it doesn't find them, it will look at `user2`'s prototype (`user1` in this case). If the property isn't found in `user1` either, JavaScript will continue searching up the prototype chain.

Example:

```
 console.log(user2.name); // "Rohit" (inherited from user1)
console.log(user2.amount); // 20 (directly in user2)
```

---

2. Prototype Chain of Arrays

```
let arr = [10, 20, 30, 40];
console.log(arr.__proto__ == Array.prototype);
console.log(arr.__proto__.__proto__ == Object.prototype);
console.log(arr.__proto__.__proto__.__proto__ == null);
```

- **`arr.__proto__ == Array.prototype`:**
  - Every array in JavaScript has a prototype, and for arrays, the prototype is `Array.prototype`.
  - `arr.__proto__` refers to the prototype of `arr`, which is `Array.prototype`. This is a built-in object that provides methods like `.push()`, `.pop()`, `.map()`, etc.
  - `Array.prototype` is itself an object, and it has methods and properties that are available to all arrays.
  - The statement `arr.__proto__ == Array.prototype` will return `true`.
- **`arr.__proto__.__proto__ == Object.prototype`:**
  - The prototype of `Array.prototype` is `Object.prototype`.
  - This is because in JavaScript, all objects (including arrays) inherit from `Object.prototype`. So, `Array.prototype` itself has `Object.prototype` as its prototype.
  - The statement `arr.__proto__.__proto__ == Object.prototype` will return `true`.

- **`arr.__proto__.__proto__.__proto__ == null`:**
    - The prototype of `Object.prototype` is `null`.
    - This is the end of the prototype chain in JavaScript. After `Object.prototype`, there is no higher-level prototype, and it is `null`.
    - The statement `arr.__proto__.__proto__.__proto__ == null` will return `true`.

Summary of the Prototype Chain in this Case

1. **For `arr`:**
    - `arr.__proto__` is `Array.prototype` (the prototype object for arrays).
    - `arr.__proto__.__proto__` is `Object.prototype` (the prototype object for all objects).
    - `arr.__proto__.__proto__.__proto__` is `null`, which indicates the end of the prototype chain.

Diagram of the Prototype Chain for `arr` (Array):

arr → [10, 20, 30, 40]
    __proto__ → Array.prototype
        __proto__ → Object.prototype
            __proto__ → null

This chain is crucial in understanding inheritance in JavaScript, where each object is linked to a prototype that it inherits methods and properties from.

DAY 12

Here’s an explanation of the code you've shared:

1. Basic Functions:

- `greet()` is a simple function that logs three messages to the console.

Functions are blocks of reusable code, and in this case, calling `greet()` will display:
Hello Coder Army

Mein badiya hu

Aur Kya chal rha hai

2. Function with Parameters:

The `sum()` function takes two parameters (`number1`, `number2`) and logs their sum.
Example:

```
sum(3, 4); // Output: 7

sum(10, 15); // Output: 25
```

3. Function with Return Value:

The `multiply()` function multiplies `number1` and `number2` and returns the result.
Example:

```
let result = multiply(4, 5); // Output: 20
```

4. Anonymous Function (Function Expression):

An anonymous function is a function that doesn't have a name. In the following code, the function is assigned to a variable `fun`:

```
const fun = function(){

  console.log("Hello Coder Army");

  return "Money";

}

console.log(fun()); // Output: "Hello Coder Army" and "Money"
```

5. Arrow Functions:

Arrow functions provide a shorter syntax for functions:

```
const sum = (number1, number2) => number1 + number2;

console.log(sum(3, 4)); // Output: 7
```

For the cube function, an arrow function takes one parameter (`number`) and returns its cube:

```
const cube = number => number * number * number;

console.log(cube(8)); // Output: 512
```

6. Spread Operator and Rest Parameter:

Spread Operator (`...`): Used to copy elements from an array into a new array.

```
let arr = [2, 3, 4, 5];

let arr2 = [...arr]; // arr2 is a copy of arr
```

Rest Operator (`...`): Used in function arguments to collect all arguments into an array.

```
const sum = function(...number) {
  console.log(number); // Logs all the arguments passed to the function
}

sum(2, 3, 4); // Output: [2, 3, 4]
```

7. Object Destructuring:

Object destructuring allows you to extract specific properties from an object:
```
let obj = { name: "Rohit", age: 30, amount: 420 };

const { name, amount } = obj;

console.log(name, amount); // Output: "Rohit 420"
```

8. Pass by Value vs Pass by Reference:

- **Pass by Value**: Primitive types (like numbers, strings) are passed by value, meaning changes inside a function don't affect the original value.

Pass by Reference: Objects are passed by reference, meaning changes inside a function will modify the original object.
```
function fun({ name, amount }) {

  console.log(name, amount);

}

fun(obj); // Logs: "Rohit 420"
```

9. Object Creation and Prototype:

`Object.create()` is used to create an object that inherits from another object:

```
let obj1 = { a: 1, b: 2 };

let obj2 = Object.create(obj1); // obj2 now inherits properties from obj1

console.log(obj2.a); // Output: 1 (inherited from obj1)
```

-
- In JavaScript, `obj2.__proto__` is the prototype of `obj2`, which points to `obj1`. This means `obj2` has access to properties of `obj1`.

Summary:

- You've explored basic functions, function expressions, arrow functions, destructuring, and handling objects.
- Understanding the difference between pass-by-value and pass-by-reference is key when working with primitive types and objects in JavaScript.

DAY 13

Here's an explanation of the different scopes and behaviors you're dealing with in the provided code:

1. Global Scope

In JavaScript, variables declared outside any function or block are in the global scope. These variables are accessible from anywhere in the code after they're declared.

```
let a = 10;  // global variable
var b = 20;  // global variable (var has function scope, but outside any function, it's globally scoped)
const c = 30;  // global constant
```

2. Function Scope (Local Scope)

Variables declared inside a function are only accessible within that function, and this is called **function scope**. The `let` and `const` declarations in a function are scoped to that function, meaning they won't interfere with the global variables of the same name.

```
function greet() {
    let a = 10;  // local to greet function
    var b = 20;  // local to greet function (but behaves differently due to var's scope)
    const c = 30;  // local to greet function
    console.log("Hello Function");
    console.log(a, b, c);  // This will print 10, 20, and 30 because they are local to this function
}
```

When you call `greet()`, it works with its local variables and prints their values. Outside the function, you cannot access `a`, `b`, or `c`, because they are confined to the scope of the function.

3. Block Scope

Variables declared with `let` or `const` are block-scoped, meaning they are accessible only within the nearest enclosing block (like a loop or conditional block). This is different from `var`, which is function-scoped.

```
if (true) {
   let a = 10;  // Block-scoped variable
   const c = 30;  // Block-scoped constant
}

console.log(amount);  // This will log 400 because 'amount' was declared globally, so it is accessible here.
```

In the `if` block above, `let a` and `const c` are only accessible inside the block, but they do not affect the global `amount` variable. The `console.log(amount)` prints `400` because the `amount` was declared globally and was not modified inside the `if` block.

4. Hoisting with `var` vs. `let` vs. `const`

- **`var`** declarations are **hoisted** to the top of the scope (function or global), but they are initialized to `undefined` before they are assigned a value. This means you can reference the variable before it's defined, but you'll get `undefined`.

```
console.log(amount);  // This will print 'undefined' because 'var amount' is hoisted but not yet assigned.
var amount = 400;
```

- **`let`** **and** **`const`** are also hoisted, but they are **not initialized** until the code execution reaches their declaration. This results in a **ReferenceError** if you try to use them before their declaration.

```
console.log(a);  // This will throw a ReferenceError because `let` and `const` are hoisted but not initialized.
```

```
let a = 10;
```

5. Function Expressions and Declarations

- **Function Declarations** (like `greet()`) are hoisted and can be used before they are defined.

```
function greet() {
   console.log("Hello Greet");
}
greet();  // This will work even if you call it before it's declared because of hoisting.
```

- **Function Expressions** (like `meet = function() {}`) are not hoisted the same way. The function `meet` cannot be called before it's assigned to a variable because the function is treated as an expression.

```
meet();  // This will throw an error because the function expression is not hoisted.
const meet = function() {
   console.log("Hello Meet");
};
```

Points:

1. **Global Scope**: Variables are accessible from anywhere in the code.
2. **Function Scope**: Variables are accessible only within the function where they are declared.
3. **Block Scope**: Variables declared with `let` and `const` are accessible only within the block (loops, conditionals).
4. **Hoisting**: `var` declarations are hoisted and initialized to `undefined`, whereas `let` and `const` are hoisted but not initialized, resulting in errors if accessed before assignment.
5. **Function Declaration vs. Function Expression**: Function declarations are hoisted, while function expressions (using `const` or `let`) are not.

Control Flow:

1. `if-else` Statement

The `if-else` statement is used to execute a block of code based on a condition. If the condition evaluates to `true`, the code inside the `if` block is executed. If it's `false`, the code inside the `else` block is executed.

Example:

```
let age = 7;


if (age >= 18) {
    console.log("Eligible for vote");
} else {
    console.log("Not Eligible for vote");
}
```

- If `age` is 18 or more, it prints `"Eligible for vote"`, otherwise it prints `"Not Eligible for vote"`.

2. `if-else if-else` Statement

You can use multiple `else if` conditions when you have more than two options to check. It allows checking multiple conditions sequentially.

Example:

```
let age = 49;


if (age < 18) {
    console.log("KID");
```

```
} else if (age > 45) {
    console.log("OLD");
} else {
    console.log("YOUNG");
}
```

- The first condition checks if `age` is less than 18 (KID).
- The second condition checks if `age` is greater than 45 (OLD).
- The `else` condition handles the case where the age is between 18 and 45, printing `"YOUNG"`.

3. `switch` Statement

The `switch` statement is used to evaluate multiple conditions based on the value of an expression. It can be used for multiple comparisons that are based on a single value.

Example:

```
let day = "0"; // Sunday

switch(day) {
    case "0":
        console.log("SUNDAY");
        break;
    case "1":
        console.log("MONDAY");
        break;
    case "2":
        console.log("TUESDAY");
```

```
        break;
    case "3":
        console.log("WEDNESDAY");
        break;
    case "4":
        console.log("THURSDAY");
        break;
    case "5":
        console.log("FRIDAY");
        break;
    case "6":
        console.log("SATURDAY");
        break;
    default:
        console.log("Not a Valid Day");
}
```

- `switch` matches the value of `day` with each case and executes the block that corresponds to it. The `break` statement exits the switch block after a match.
- The `default` case handles values that do not match any of the cases.

4. Loops

Loops are used when you want to repeatedly execute a block of code. There are different types of loops in JavaScript:

- **`for` Loop**: The `for` loop is used when the number of iterations is known.

Example (Printing "Hello Coder Army" 20 times):

```
for (let i = 0; i < 20; i++) {
    console.log("Hello Coder Army");
}
```

- **`for` Loop for Sum of First `n` Numbers**:

Example (Sum of first 10 numbers):

```
let sum = 0;
for (let i = 1; i <= 10; i++) {
    sum += i;
}
console.log(sum);  // Output: 55
```

- **Nested `for` Loop**: A loop inside another loop.

Example (Printing a pattern):

```
for (let j = 0; j < 6; j++) {  // Outer loop
    for (let i = 1; i <= 5; i++) {  // Inner loop
        console.log(i);
    }
}
```

This prints:

1

2

3

4

5

1

2

3

4

5

...

- **`while` Loop**: The `while` loop runs as long as the condition is true. It’s useful when you don't know the number of iterations in advance.

Example:

```
let i = 1;

while (i < 6) {

    console.log(i);

    i++;

}
```

- **`do-while` Loop**: Similar to the `while` loop, but the condition is checked after the code is executed, ensuring the loop runs at least once.

Example:

```
let i = 1;
do {
    console.log(i);
    i++;
} while (i < 6);
```

5. `for-in` Loop (Object Iteration)

The `for-in` loop is used to iterate over the keys of an object.

Example (Iterating over an object):

```
const obj = {
    name: "Rohit",
    age: 30,
    amount: 420,
    city: "Kotdwar"
};

for (let key in obj) {
    console.log(obj[key]);  // Prints the value of each property
}
```

Output:

Rohit

30

- You can also get the keys of an object using `Object.keys()` and iterate over them.

6. Notes on Loop Control

- `break`: Exits the loop or `switch` statement immediately.
- `continue`: Skips the current iteration of a loop and moves to the next iteration.

Summary of Key Concepts:

1. **`if-else`**: Conditional execution based on boolean expressions.
2. **`switch`**: Matches an expression to multiple possible cases.
3. **Loops**:
    - `for`: Fixed number of iterations.
    - `while`: Runs until the condition becomes `false`.
    - `do-while`: Executes at least once before checking the condition.
4. **`for-in`**: Iterates over object properties

DAY14

1

1. **Object.defineProperty()**:
    - This method is used to define or modify the properties of an object. It allows you to set specific characteristics of a property, like whether it's writable, enumerable, or configurable.
    - You can use it to make a property **read-only**, **non-enumerable**, or **non-configurable**.

2. **Writable**:
    - If `writable: false` is set on a property, the value cannot be changed. For example, when you try to change `obj.name` after setting it to `false`, the update won't be successful.
3. **Enumerable**:
    - If a property has `enumerable: false`, it won’t show up in `for...in` loops or `Object.keys()`. However, it can still be accessed directly. For example, setting `enumerable: false` for `name` means the property will not appear when you loop through the object, but its value can still be accessed.
4. **Configurable**:
    - This means the property can be deleted or its characteristics can be modified. If `configurable: false` is set, the property can’t be deleted or redefined.

Key Points in Your Code:

- **`Object.defineProperty()`** is used to define or modify how the properties behave.

For Example:

```
Object.defineProperty(obj, 'name', {
 value: "rohit",
 writable: true,   // you can change this value
 enumerable: true, // this property will show up in loops
```

```
  configurable: true // you can delete or modify this property
});
```

-

When you set `writable: false` for a property, you can't change its value:

```
 obj.name = "Mohit"; // This won't work if writable is false
```

-
- **Inheritance with `Object.create()`**:
    - When you use `Object.create(customer)`, `customer2` inherits properties from `customer`.
    - If you set `enumerable: false` for `name` in the `customer` object, it won't show up when you loop through `customer2` even though it's inherited.
- **The `for...in` loop** will iterate over both the object's own properties and those inherited from its prototype (unless `enumerable` is set to `false`).

Example Breakdown:

Setting a property as non-enumerable:

```
 Object.defineProperty(customer, "name", { enumerable: false });
```

1.
    - Now, the `name` property will not appear when you loop through `customer` using `for...in`.

2. **Inheritance and enumerable**:
    - `customer2` inherits from `customer`, so if a property is not `enumerable` in `customer`, it will also be hidden when looping through `customer2`.
3. **Modifying the prototype**:
    - You can modify properties of the `Object.prototype`, like making `toString` enumerable:

```
Object.defineProperty(Object.prototype, 'toString', { enumerable: true
});
```

Final Output:

In the loop:

```
for (let key in customer) {
   console.log(key);  // Only `age`, `account_number`, and `balance`
will be printed, not `name` because it's non-enumerable
}
```

2

1. Basic Object:

```
let obj = {
   name: "rohan",
   age: 23,
   gender: "male",
```

```
    city: "kotdwar"
};
```

- This is an object `obj` with 4 properties: `name`, `age`, `gender`, and `city`.

2. Using `for...in` Loop:

```
for(let key in obj) {
    console.log(key);
}
```

- The `for...in` loop iterates over **all the keys** in the object.
- In this case, it will print the keys: `name`, `age`, `gender`, and `city`. It doesn't print the values, only the property names.

Result:

```
 name
age
gender
city
```

-

If you want to print both the keys and their values, you can use:

```
for(let key in obj) {
    console.log(key, obj[key]);
}
```

This will print:

```
 name rohan
age 23
gender male
city kotdwar
```

-

3. `Object.keys(obj)`:

console.log(Object.keys(obj));

- `Object.keys()` is a method that returns an **array** of the object's own property names (keys).

So, `Object.keys(obj)` will give you:
["name", "age", "gender", "city"]

-

4. Inheritance with `Object.create()`:

let obj2 = Object.create(obj);
obj2.money = 420;
obj2.id = "Roh";

- `Object.create(obj)` creates a new object (`obj2`) that **inherits** all properties from `obj`.
- `obj2` also has its own properties: `money` and `id`.
- **Note**: Even though `obj2` inherits `name`, `age`, `gender`, and `city` from `obj`, they are not directly part of `obj2`'s own properties (they are inherited).

5. `Object.keys(obj2)`:

console.log(Object.keys(obj2));

`Object.keys(obj2)` returns only the **own properties** of `obj2` (not the inherited ones). So it will give:
["money", "id"]

-

6. `for...in` Loop with Inherited Properties:

```
for(let key in obj2) {
    console.log(key);
```

}

- The `for...in` loop will iterate over **both the own properties of `obj2` and the inherited properties from `obj`**.

So, it will print:
money
id
name
age
gender
city

-

Key Points:

- **`for...in` loop**: Iterates over **all properties** (own and inherited) of an object.
- **`Object.keys(obj)`**: Returns **only the own properties** of the object in an array.
- **Inheritance**: When using `Object.create()`, the new object inherits properties from the prototype, and `for...in` will loop through those inherited properties as well.

Summary:

- **`Object.keys()`**: Used when you want to get just the properties that belong directly to the object (not inherited).
- **`for...in`**: Loops over all properties, including inherited ones.
-

3

1. `for...in` with Arrays:

The `for...in` loop is generally used to iterate over the **properties of objects**. But in the case of an array, it will **loop over all enumerable keys** (indexes and other

properties), including any additional properties added to the array (like `name` and `age` in your example).

```
const arr = [10, 20, 40, 12, 30];
arr.name = "Rohit";
arr.age = 20;
```

- Normally, arrays are indexed by numbers (`0, 1, 2, 3, ...`), but here you added `name` and `age` as properties to the array.
- When you use the `for...in` loop, it will **iterate over all properties** (not just the indexed values):

```
for (let key in arr) {
   console.log(key);
}
```

Output:

```
0
1
2
3
4
name
age
```

- This includes the numeric keys (`0, 1, 2, 3, 4`) as well as the custom properties (`name`, `age`), because they are part of the array object.

Important Note: It's generally not a good practice to use `for...in` for arrays since it also picks up non-indexed properties. Instead, you should use a **`for` loop** or **`forEach()`** for iterating over array elements.

---

2. `Object.defineProperty()` and `Object.defineProperties()`:

`Object.defineProperty()`:

This method is used to **define or modify a single property** on an object. You can set various characteristics of the property, such as whether it's **writable**, **enumerable**, or **configurable**.

Syntax:

```
Object.defineProperty(obj, 'propertyName', {
    value: 'value',
    writable: true,     // Can this value be changed?
    enumerable: true,   // Should this property show up in for...in loop or Object.keys()?
    configurable: true  // Can this property be deleted or redefined?
});
```

Example:

```
const person = { name: 'Rohan' };

Object.defineProperty(person, 'age', {
    value: 25,
    writable: false,    // Cannot change age
    enumerable: true,   // Can show up in loops
    configurable: false // Cannot delete or modify this property later
});

person.age = 30;  // This won't work because writable is false
console.log(person.age);  // 25
```

`Object.defineProperties()`:

This method is similar to `defineProperty`, but it allows you to **define or modify multiple properties** at once.

Syntax:

```
Object.defineProperties(obj, {
    'propertyName1': {
        value: 'value1',
        writable: true,
        enumerable: true,
```

```
        configurable: true
    },
    'propertyName2': {
        value: 'value2',
        writable: false,
        enumerable: true,
        configurable: false
    }
});
```

Example:

```
const person = { name: 'Rohan' };

Object.defineProperties(person, {
    'age': {
        value: 25,
        writable: false,
        enumerable: true,
        configurable: true
    },
    'gender': {
        value: 'Male',
        writable: true,
        enumerable: true,
        configurable: true
    }
});

console.log(person.age);     // 25
console.log(person.gender);  // Male
```

- `Object.defineProperties()` allows you to set multiple properties' descriptors at once, rather than calling `defineProperty()` separately for each one.

Key Takeaways:

- **`for...in` loop**: Can be used on arrays, but it's not ideal because it iterates over **all properties**, not just array elements. For arrays, it's better to use a **`for` loop** or **`forEach()`**.

- **`Object.defineProperty()`**: Used to define or modify a **single property** of an object, with options like `writable`, `enumerable`, and `configurable`.

- **`Object.defineProperties()`**: Works the same as `defineProperty()`, but for **multiple properties at once**.

DAY 15

Callback Functions:

A **callback function** is a function you give to another function, which will then call it later.

For example:

```
function sayHello(callback) {
    console.log("Hello!");
    callback();  // Calls the callback function
}


const greet = function() {
```

```
    console.log("I am the callback function!");
};

sayHello(greet);  // sayHello calls the greet function
```

In this example:

- `sayHello` is a function that accepts another function (`callback`) as an argument.
- Inside `sayHello`, it calls `callback()`, which is the `greet` function.
- When you run `sayHello(greet)`, you first see `"Hello!"`, then `"I am the callback function!"`.

You can also pass **anonymous functions** (functions without names) like this:

```
sayHello(function() {
    console.log("I am the callback function!");
});
```

Or use **arrow functions** (a shorter way to write functions):

```
sayHello(() => {
    console.log("I am the callback function!");
});
```

setInterval:

The `setInterval()` function is used to run a function repeatedly, after a certain amount of time.

For example:

```
function fetchData() {
    console.log("I am fetching data!");
}

setInterval(fetchData, 5000);  // Calls fetchData every 5 seconds
```

In this example:

- `fetchData` is called every 5 seconds (5000 milliseconds) and prints `"I am fetching data!"` each time.

In short:

- A **callback function** is just a function you pass to another function to be run later.
- `setInterval()` calls a function over and over again at set intervals, like every 5 seconds.

1. `Object.defineProperty()`

```
let user = { name: "rohit", age: 30 };

Object.defineProperty(user, 'name', {
  writable: false,
});
```

```
user.name = "mohit";  // This will fail because 'name' is not writable.


console.log(Object.getOwnPropertyDescriptor(user, "name"));
```

- `Object.defineProperty()` is used to define or modify a property on an object with specific characteristics.
- Here, you are making the `name` property **non-writable**. So, when you try to change the `name` to "mohit", it won't work.
- `Object.getOwnPropertyDescriptor()` helps you view the details of the property, including if it's writable, configurable, etc.

2. `for...of` loop

- The `for...of` loop is used to loop over iterable objects like arrays, strings, or other objects that are **iterable**.

Example with an array:

```
 const arr = [10, 20, 11, 18, 13];

for (let value of arr) {

    console.log(value);  // Logs each value of the array

}
```

Example with a string:

```
 let str = "Rohit is Good Boy";

for (let value of str) {

    console.log(value);  // Logs each character of the string

}
```

Don't use `for...of` with objects directly. Instead, use `Object.keys()`, `Object.values()`, or `Object.entries()` for iteration:

```
 const obj = {
    name: "Chavvi",
    age: 22,
    gender: "female"
};


for (let key of Object.keys(obj)) {
    console.log(key, obj[key]);  // Logs the key and value of each property
}
```

3. `forEach()` method

- `forEach()` is a method available on arrays in JavaScript. It executes a function for each element of the array.

Example:

```
 let arr = [10, 20, 30, 40, 50];
arr.forEach((num, index, array) => {
    console.log(num);  // Logs each number
    console.log(index);  // Logs the index of each element
    array[index] = num * 2;  // Modifies the array (doubling each value)
});


console.log(arr);  // Logs the updated array [20, 40, 60, 80, 100]
```

- `forEach()` doesn't return anything, it just executes the callback function for each element.

4. `filter()` method

- `filter()` creates a new array with all elements that pass the test provided by the callback function.

Example with an array:

```
let arr = [10, 22, 33, 41, 50];

const result = arr.filter(num => num % 2 === 0);

console.log(result);  // Logs [10, 22, 50]
```

Filtering students based on marks:

```
const students = [
    {name: "Rohan", age: 22, marks: 70},
    {name: "Mohan", age: 24, marks: 80},
    {name: "Darshan", age: 28, marks: 30},
    {name: "Mohit", age: 32, marks: 40},
    {name: "Shadik", age: 12, marks: 90},
];

const result = students.filter(({marks}) => marks > 50);

console.log(result);
```

- The result is an array of students with marks greater than 50.

5. `map()` method

- `map()` creates a new array with the results of calling a function for every array element.

Example:

```
const arr = [1, 2, 4, 5];

const result = arr.map((num, index) => num * index);

console.log(result);  // Logs [0, 2, 8, 15]
```

It can also be chained:

```
const arr = [1, 2, 3, 4, 5, 6];

const result = arr.filter(num => num % 2 === 0)  // Filter even numbers

                  .map(num => num * num)      // Square them

                  .map(num => num / 2);       // Divide by 2

console.log(result);  // Logs [2, 8, 18]
```

6. `reduce()` method (Explanation)

- The `reduce()` method executes a reducer function (you provide) on each element of the array, resulting in a single output value.
- It's helpful for accumulating results (e.g., sum, product, etc.) or transforming an array into an object.

Syntax:

```
array.reduce((accumulator, currentValue, index, array) => {

   // logic to process each value

   return accumulator;  // Final result

}, initialValue);  // initialValue is optional
```

Example with sum:

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

const sum = arr.reduce((acc, curr) => acc + curr, 0);

console.log(sum);  // Logs 10
```

Example with an object transformation:

```
 const arr = [{a: 1}, {b: 2}, {c: 3}];

const result = arr.reduce((acc, curr) => {

    return {...acc, ...curr};  // Merge each object into one

}, {});

console.log(result);  // Logs {a: 1, b: 2, c: 3}
```

In summary:

- `reduce()` can be used for complex accumulations, whether it's numbers, objects, or arrays.
- `map()` is great for transforming data.
- `filter()` is ideal for extracting specific data based on conditions.
- `forEach()` is good for side-effects (like updating arrays, logging, etc.), but doesn't return anything.

DAY 16

1

In the code you provided, the goal is to dynamically add a property to the `obj` object based on the value of the `curr` variable. Since `curr` is set to `"apple"`, it

checks if `obj` already has a property named `"apple"`. If the property exists, it increments its value; otherwise, it adds the property and sets its value to `1`.

1. **Checking for the property**: The `if` statement checks if the object `obj` has a property `apple`. The `hasOwnProperty` method ensures that only properties directly belonging to `obj` are checked, not inherited properties.
2. **Incrementing or Adding the property**: If `"apple"` exists as a property in `obj`, its value is incremented by 1. If it doesn't exist, a new property `apple` is created with a value of `1`.

Given your current code:

```
let obj = {
    name: "rohit",
    age: 10,
    orange: 1,
}

let curr = "apple";

if (obj.hasOwnProperty(curr))
    obj[curr]++;
else
    obj[curr] = 1;

console.log(obj);
```

The output will be:

```
{
    name: "rohit",
    age: 10,
    orange: 1,
    apple: 1
}
```

This is because `"apple"` was not initially a property of `obj`, so it gets added with the value `1`.

2

The code you shared uses the `reduce` method to count how many times each fruit appears in the `arr` array. Here's a breakdown of how it works in a simple way:

Array:

let arr = ["orange", "apple", "banana", "orange", "apple", "banana", "orange", "grapes"];

This array has some fruits repeated multiple times.

`reduce` Method:

The `reduce` method is used to **accumulate** a value from the array. In your case, we are creating an **object** that counts the occurrences of each fruit in the array.

Code Explanation:

```
const result = arr.reduce((acc, curr) => {
    acc.hasOwnProperty(curr) ? acc[curr]++ : acc[curr] = 1;
    return acc;
}, {});
```

- `acc`: This is the accumulator, which starts as an empty object `{}`.
- `curr`: This is the current item (fruit) from the array being processed.

Steps:

1. The `reduce` method goes through each fruit in the array (`arr`).
2. For each fruit (`curr`), it checks if the accumulator (`acc`) already has that fruit as a property:
    - If it does, it **increments** the count of that fruit (`acc[curr]++`).
    - If it doesn't, it **adds** the fruit to the accumulator and sets its count to 1 (`acc[curr] = 1`).
3. After processing all elements in the array, the accumulator will contain the count of each fruit.

Final Object:

After the code runs, the result will be:

```
{
  orange: 3,
  apple: 2,
```

banana: 2,

grapes: 1

}

- `orange` appears 3 times.
- `apple` appears 2 times.
- `banana` appears 2 times.
- `grapes` appears 1 time.

Summary:

The `reduce` method is going through each element of the array and building an object where the keys are the fruits and the values are the number of times each fruit appears.

3

1. Using `Map` :

A `Map` is a collection of key-value pairs where:

- Keys can be any data type (e.g., number, string, object).
- Each key is **unique**.
- The value associated with the key can be anything (including objects, arrays, etc.).

In your code, you're using a `Map` to store some key-value pairs and perform various operations.

Code Breakdown:

const map1 = new Map([[4, "rohit"], ["Mohan", "rohan"], [30, 9], [63, 78]]);

This creates a `Map` with four entries:

- `4` -> `"rohit"`
- `"Mohan"` -> `"rohan"`
- `30` -> `9`
- `63` -> `78`

Operations with `Map`:

1. **Accessing `Map` Values**:
    - You can use `map1.get(key)` to get the value for a particular key.

```
console.log(map1.get(4));  // Output: "rohit"
```

2.
3. **Check if a Key Exists**:
    - `map1.has(key)` checks if a key exists in the `Map`.

```
console.log(map1.has(4));  // Output: true

console.log(map1.has("abc"));  // Output: false
```

4.
5. **Size of the `Map`**:
    - `map1.size` gives the number of key-value pairs in the `Map`.

```
console.log(map1.size);  // Output: 4
```

6.
7. **Deleting a Key-Value Pair**:
    - `map1.delete(key)` removes a specific key-value pair from the `Map`.

```
map1.delete(4);  // Removes the key-value pair (4, "rohit")

console.log(map1);
```

8.
9. **Clearing All Key-Value Pairs**:

    - `map1.clear()` removes all the key-value pairs from the `Map`.

```
map1.clear();

console.log(map1);  // Output: Map(0) {}
```

10.
11. **Iterating Over a `Map`**:

    - You can use a `for...of` loop to iterate over the key-value pairs.

```
for (let [key, value] of map1) {
   console.log(key, value);  // Logs each key-value pair
}
```

12.

Object vs `Map`:

- **Object**:
    - Keys can only be strings or symbols.
    - `Map` allows keys to be numbers, strings, or even objects.
- **`Map`**:
    - More flexible with key types.
    - Better for use cases where you need to store key-value pairs with non-string keys, as keys in a `Map` can be numbers, objects, etc.

Example:

```
const obj = {};

obj["name"] = "John";  // String as key

// obj[10] = "ten";  // This works, but the key gets converted to a string


const map = new Map();

map.set(10, "ten");  // Number as key

map.set("name", "John");

map.set({}, "object");  // Object as key
```

JavaScript Code Execution:

JavaScript code execution typically happens in the following way:

1. **Parsing**: JavaScript engines (like V8 in Chrome, SpiderMonkey in Firefox) first **parse** the code, converting it into an abstract syntax tree (AST).
2. **Compilation**: Then, the JavaScript engine compiles the code into machine-readable bytecode.
3. **Execution**: Finally, the code is executed in an event loop.

- **Single-threaded**: JavaScript is **single-threaded**, meaning it executes one operation at a time.
- **Event Loop**: When asynchronous code (like setTimeout, promises) is encountered, it gets queued and waits for the main thread to be free.
- **Stack and Heap**: JavaScript uses a **call stack** (for executing functions) and a **memory heap** (for storing objects and variables).

4

1. What is a `Set`?

A `Set` in JavaScript is a **collection of unique values**. It automatically removes duplicates from an array or collection. The values in a `Set` can be of any type, including objects, but each value can only appear **once** in a `Set`.

Example:

const set1 = new Set([10, 20, 30, 40, 10, 30]);

console.log(set1); // Output: Set { 10, 20, 30, 40 }

- In the above example, `10` and `30` are repeated, but in the `Set`, only unique values are stored.

2. Common `Set` Methods:

- **add(value)**: Adds a value to the `Set`. If the value already exists, it won't be added again.
- **delete(value)**: Removes a specific value from the `Set`.
- **has(value)**: Checks if a specific value exists in the `Set`.
- **clear()**: Removes all values from the `Set`.
- **size**: Returns the number of unique values in the `Set`.

Example:

let set1 = new Set();

set1.add(4); // Adds 4 to the set

set1.add(6); // Adds 6 to the set

set1.add("Rohit"); // Adds "Rohit" to the set

set1.add(30); // Adds 30 to the set

console.log(set1.size); // Output: 4 (because we added 4 unique items)

set1.delete(6); // Removes 6 from the set

console.log(set1.size); // Output: 3 (6 is removed)

3. Converting Array to Set:

You can use a `Set` to remove duplicates from an array. For example:

let arr = [10, 30, 20, 10, 40, 50, 30];

const set1 = new Set(arr); // Converts array to set, removing duplicates

arr = [...set1]; // Converts the set back to an array

console.log(arr); // Output: [10, 30, 20, 40, 50]

4. Set Operations:

You can perform set operations like **union** and **intersection**.

Union (All unique values from both sets):

You can combine two sets and remove duplicates:

let set1 = new Set([10, 20, 30, 40, 50]);

let set2 = new Set([10, 20, 70, 40]);

let set3 = new Set([...set1, ...set2]);

console.log(set3); // Output: Set { 10, 20, 30, 40, 50, 70 }

Intersection (Common values between both sets):

To find common values between two sets, you can use the `filter` method:

let set1 = new Set([10, 20, 30, 40, 50]);

```
let set2 = new Set([10, 20, 70, 40]);


// This filters only the elements that exist in both set1 and set2
const result = new Set([...set1].filter(num => set2.has(num)));
console.log(result); // Output: Set { 10, 20, 40 }
```

5. Iterating Over a Set:

You can use `for...of` or `forEach` to loop through all the values in a set.

Example using `for...of`:

```
for (let value of set1) {
    console.log(value); // Prints each value in the set
}
```

Example using `forEach`:

```
set1.forEach(value => console.log(value)); // Prints each value in the set
```

Summary of Methods:

- **add(value)**: Adds a value.
- **delete(value)**: Removes a value.
- **has(value)**: Checks if a value exists.
- **clear()**: Removes all values.
- **size**: Gets the size of the set.
- **forEach**: Iterates over all values in the set.

A `Set` is a great way to work with collections of unique values, especially when you need to remove duplicates or perform set operations like union and intersection.

DAY 17

Let’s break down the code step by step in simple terms:

```
let z;       // 'z' is declared, but not initialized. It is 'undefined' by default.
var x = undefined;   // 'x' is declared and explicitly set to 'undefined'.
let y;       // 'y' is declared, but not initialized. It is 'undefined' by default.
```

At this point:

- `z` is `undefined` because it hasn’t been assigned any value yet.
- `x` is explicitly set to `undefined`.
- `y` is also `undefined` because it hasn’t been given a value.

Now, the next steps:

```
console.log(x);    // Logs the value of 'x', which is 'undefined'.
z = 50;            // 'z' is now assigned the value 50.
```

At this point:

- `x` is still `undefined` when logged.
- `z` is now `50`.

```
x = 10;            // 'x' is now assigned the value 10.
y = 20;            // 'y' is now assigned the value 20.
```

At this point:

- `x` becomes `10`.
- `y` becomes `20`.

```
// a = 20;    // This line is commented out, so it doesn’t run.
console.log(z);    // Logs the value of 'z', which is 50.
console.log(x);    // Logs the value of 'x', which is now 10.
```

At the end:

- `z` is logged as `50`.
- `x` is logged as `10`.

Summary:

- `z` was initially `undefined` and then changed to `50`.
- `x` was initially `undefined`, then changed to `10`.
- `y` was initialized to `20`.
- The commented line (`a = 20;`) is ignored and doesn't affect the code.

hoisting.

```
console.log(x);
console.log(y);
var x = 10;
let y = 20;
```

What happens step by step:

1. **Hoisting with `var`**:
    - JavaScript **hoists** variable declarations (not their initializations) to the top of their scope.
    - When you use `var x = 10;`, only the declaration (`var x;`) is hoisted to the top, but the assignment (`= 10`) happens where

the code is written. So, at the time of the first `console.log(x)`, `x` is `undefined` because the declaration has been hoisted, but the value hasn't been assigned yet.

2. **Hoisting with `let`**:

    - Variables declared with `let` are **hoisted** too, but they are not initialized to `undefined`. Instead, they go into a "temporal dead zone" (TDZ) until the actual declaration is reached in the code. In this case, `let y = 20;` is hoisted, but JavaScript doesn't let you access `y` until it's initialized.
    - So, when you try to log `y` before it's initialized, you'll get a **ReferenceError** because of the TDZ.

Execution flow:

1. JavaScript hoists `var x;` and `let y;` to the top.
2. `console.log(x)` is executed, but since `x` is hoisted and hasn't been assigned yet, it prints `undefined`.
3. `console.log(y)` throws an error because `y` is in the temporal dead zone until its initialization.

The output:

undefined
// ReferenceError: Cannot access 'y' before initialization

Key takeaway:

- **`var`**: The variable is hoisted, and initialized to `undefined` until the code assigns it a value.
- **`let`**: The variable is hoisted, but it stays in the "temporal dead zone" until it's initialized in the code, so accessing it before initialization results in an error.

Sure! Let's simplify it:

Function Declarations (`greet`)

```
// greet();  // This works because the function is already available due to hoisting

function greet() {
    console.log("Hello World");
}

greet();  // Prints "Hello World"
```

- **Memory Allocation**: The function `greet` is available even before it's written in the code.
- **Code Execution**: When `greet()` is called, it works because the function is already stored in memory.

Function Expressions (`meet`)

```
var meet = function() {
    console.log("Hello Meet");
};

meet();  // Prints "Hello Meet"
```

- **Memory Allocation**: The variable `meet` is created, but it doesn't have the function assigned to it yet.
- **Code Execution**: The function gets assigned to `meet` only when the code runs, so you can call `meet()` after that.

Variable Declaration (`var x`)

```
var x;

console.log(x);  // Prints "undefined" because x isn't assigned yet

x = 10;  // Now x gets the value 10
```

- **Memory Allocation**: `x` is created, but it's `undefined` at first.
- **Code Execution**: When the code runs, `x` gets the value `10`. But the `console.log(x)` prints `undefined` because it happens before `x` is assigned a value.

Key Points:

1. **Function Declarations** can be used before they're written in the code.
2. **Function Expressions** only work after they're assigned a function.
3. **Variables (`var`)** are hoisted, but have the value `undefined` until you assign them a value.

DAY 18

1. **Global Object**:
    - In browsers like Chrome, the global object is `window`.
    - In Node.js, it's `global`.
    - `globalThis` is a standard global object that can be used in any environment (browser or Node.js).

2. `console.log("Hello World")`: This outputs `"Hello World"` to the console.

3. `Math.random()`: This generates a random number between 0 (inclusive) and 1 (exclusive).

4. `Object.freeze(obj)`: This method freezes the object `obj`, making it immutable (its properties can't be changed). So, even though you're trying to assign a new value to `obj.name`, it won't be updated.

5. **Output**: The line `console.log(obj)` will print `{ name: 10 }`, because the object has been frozen, and the attempt to change `obj.name` to 30 will fail silently.

In strict mode (`"use strict"`), you can't accidentally create global variables, and it helps catch common coding errors. For example, when you try to assign a value to `a` without declaring it, strict mode would throw an error (though you've commented that line out).

The code you provided explains the concept of the `this` keyword in JavaScript. Let's break it down:

1. Global Context (Outside Any Function):

- In **browsers**, `this` refers to the `window` object.
- In **Node.js**, `this` refers to the `module.exports` object.

// console.log(this); // In the browser, this will log the window object

2. Inside a Function:

- **Non-Strict Mode**: In a regular function (not in strict mode), `this` refers to the **global object** (`window` in the browser).
- **Strict Mode**: In strict mode (`"use strict"`), `this` will be `undefined` inside a function.

// Non-strict mode
function greet() {

```
    console.log(this);  // Logs the global object (window in the browser)
}

greet(); // Global object (window in browsers)

// Strict mode
"use strict";
function greetStrict() {
    console.log(this);  // undefined
}

greetStrict();  // undefined
```

3. Inside a Method (Object Context):

- When `this` is used inside an **object's method**, it refers to the **object** that owns the method.

```
const obj = {
    name: "Rohit",
    age: 20,
    meet: function() {
        console.log(this.name);  // Logs "Rohit"
    }
}

obj.meet();  // "Rohit"
```

4. Arrow Functions and `this`:

- **Arrow Functions** don't have their own `this`. They inherit `this` from their **lexical (surrounding) scope**. This is the key difference from regular functions.

In the following examples:

- **Arrow function in an object method**: The `this` inside the arrow function refers to the surrounding context (the global object, in the case of the browser, which is `window`).

```
let obj = {
    name: "Rohit",
    age: 11,
    greet: () => {
        console.log(this);  // `this` here refers to the global object (window in browsers)
    }
};

obj.greet();  // Logs the global object (window in browsers)
```

- **Arrow function inside a regular method**: When an arrow function is nested inside a regular function or method, it will still inherit `this` from the outer function's scope.

```
let obj = {
    name: "Rohit",
    age: 11,
    greet: function() {
        let ab = () => {
            console.log(this);  // `this` here refers to the object `obj`
        };
        ab();
    }
};

obj.greet();  // Logs the object obj, because the arrow function inherits `this` from greet()
```

Summary:

- In **non-strict mode**, `this` in a function refers to the global object (`window` in browsers).
- In **strict mode**, `this` in a function is `undefined`.
- Inside an object's **method**, `this` refers to the object.
- **Arrow functions** inherit `this` from the surrounding scope, which is different from regular functions that determine `this` based on how they are called.