JAVASCRIPT TOPICS

- **JavaScript Basics:**
  - Primitive vs Non-Primitive Data Types
  - Statically Typed vs Dynamically Typed Language
  - Undefined vs Not Defined
  - NaN
  - Null vs Undefined
  - Type of Errors
  - Strict Mode in JavaScript
  - IsNaN
  - Typeof Operator
  - Symbol()
  - BigInt
- **Variables and Scoping:**
  - Let vs Var
  - Const vs Let
  - Scope
  - Type of Scope
  - Scope Chain
  - Block Scope
  - Shadowing
  - Hoisting
  - Temporal Dead Zone (TDZ)
  - Lexical Scope
  - Lexical Environment
- **Functions:**
  - Function Statement
  - Function Expression-difference
  - Anonymous Function
  - Named Function Expression
  - First Class Function
  - First Order Function
  - Higher Order Function
  - Arrow Functions vs Normal Functions
  - Arrow Functions
  - Generator Functions
  - Self-Invoking Functions (Immediately Invoked Function Expression or IIFE)
  - Pure Functions
  - Function Object (also called normal function)
  - Callback Functions-practical
  - Callback Hell

- Async vs Sync Functions
- Promise
- Promise States
- Promise Chain
- async/await
- Factory Function
- Constructor Function
- This Keyword
- Call, Bind, Apply (Clarification needed on Bind)
- Currying
- Default Parameters
- Rest Operator
- Spread Operator
- Rest Parameter
- Pass by Value vs Pass by Reference
- Closure
- Immediately Invoked Function Expression (IIFE)

- **Objects and Classes:**

  - Object
  - Object Destructuring
  - Nested Destructuring
  - Object.defineProperty vs defineProperties
  - Object.fromEntries vs Object.entries()
  - Object.freeze
  - Object.seal
  - Check if Object is isFrozen or isSealed
  - Prototypal vs Classical Inheritance
  - Prototype Chaining
  - **proto** vs Prototype
  - Classes
  - Inheritance
  - Constructor
  - Accessor (Getter and Setter)
  - Enhanced Object Literals
  - Collections in JS (Map, WeakMap, Set, WeakSet)
  - Object vs Map
  - Map and HashMap

- **Strings and Arrays:**

  - String Functions
  - String Interpolation
  - Tagged Template
  - Template Literals
  - Trim
  - Slice vs Splice
  - Map, Reduce, Filter, forEach
  - For of vs For in

- Array Methods: Push, Pop, Shift, Unshift //code
- Static vs Dynamic Array
- Shallow Copy vs Deep Copy
- Array Sorting Functions

- **Control Structures:**

  - Loops: For, While, Do While
  - Conditional Operator (Ternary)
  - Default Parameters
  - Rest Operator
  - Unary Operator
  - Comma Operator
  - Nesting Templates

- **Event Handling:**

  - Event Listeners
  - Event Loop
  - Event Propagation (Bubbling, Capturing)
  - Stop Event Propagation
  - Stop Immediate Event Propagation
  - Event Delegation
  - Event Handling
  - Event Loop Working
  - Event Queue
  - Micro Task
  - Set Immediate
  - Clear Interval
  - Set Timeout vs Set Interval
  - Clear Timeout

- **JavaScript Engines and Performance:**

  - JS Engine Working
  - V8 Engine
  - Execution Context
  - Call Stack
  - Memory Management: Stack and Heap
  - Single vs Multi-Threading
  - Starvation

- **Modules and Imports:**

  - Module
  - Import and Export
  - Polyfill –old browser
  - ES6 Features
  - Strict Mode in JavaScript
  - Optional Chaining Operator

- Nullish Coalescing Operator

- **Web APIs and Asynchronous Operations:**
  - DOM
  - BOM
  - Fetch API
  - Axios vs Fetch
  - Service Worker
  - Local Storage vs Session Storage vs Cookies
  - JSON.stringify vs JSON.parse
  - HTTP Status Codes (Clarification needed)
  - Regular Expressions
  - Pure Functions

- **JavaScript Basics:**
  - Primitive vs Non-Primitive Data Types

Primitive Data Types:

Primitive data types are the most basic types of data. They are immutable, meaning their value cannot be changed after they are created. When you manipulate a primitive value, you are working directly with the value itself, not a reference to it.

Non-Primitive (Reference) Data Types:

Non-Primitive data types, also known as reference data types, are more complex data structures. They are mutable, meaning their content can be changed after they are created. Unlike primitive types, non-primitive data types store references to the actual data, not the data itself.

  - Statically Typed vs Dynamically Typed Language

Statically Typed Language:

- **Type Checking at Compile-Time:** In statically typed languages, the data type of a variable is known and checked during compile-time, before the code is run. This means that you must explicitly declare the type of each variable when writing the code.
- **Examples:** Languages like C, C++, Java, and Swift are statically typed.

Dynamically Typed Language:

- **Type Checking at Runtime:** In dynamically typed languages, the data type of a variable is determined at runtime, meaning you don't need to declare the type explicitly. The interpreter or compiler figures it out as the code runs.
- **Examples:** JavaScript, Python, Ruby, and PHP are dynamically typed languages.

- JavaScript is a **dynamically typed** language.
- This means that in JavaScript, you do not need to declare the type of a variable when you create it. The type is determined automatically by the JavaScript engine at runtime based on the value assigned to the variable. Additionally, the type of a variable can change during the execution of the program.

- Undefined vs Not Defined

    1. Undefined

        - When you create a variable (like a labeled box), JavaScript automatically puts **"undefined"** inside it.
        - It's like having an empty box—you know it exists, but there's nothing in it yet.
        - Example:

```
let myVariable; // The box exists, but it's undefined
```

        - If you try to access a variable that hasn't been declared or doesn't exist, JavaScript says it's **"not defined."**
        - It's like searching for a box that was never made.
        - Example:

```
console.log(myOtherVariable); // Not defined (the box doesn't exist)
```

- NaN
    - **NaN** is a special value that indicates something isn't a valid number.
    - It's like when you try to do math with words—it doesn't make sense!
    - JavaScript uses NaN to represent undefined or non-representable numeric values.

- Null vs Undefined
- Type of Errors
- **Syntax Error**:
    - This error occurs when you use predefined syntax incorrectly.
    - For example, missing a closing parenthesis or semicolon.
    - Example:

```
if (condition {
    // Syntax error: Missing closing parenthesis
}
```

- **Reference Error**:
    - When you try to access a variable that hasn't been declared or doesn't exist.

```
console.log(myVariable); // Reference error: myVariable is not defined
```

- **Type Error**:

- Happens when you use a value outside the scope of its data type.
- For instance, calling a method on an undefined variable.
- Example:

```
const myString = "Hello";
myString.push(" World"); // Type error: push is not a function
```

- **Evaluation Error**:
  - Rarely encountered nowadays due to modern JavaScript.
  - Occurs when using the `eval()` function.
  - Example (not recommended):

```
eval("console.log('Hello from eval!')");
```

- **Range Error**:
  - Occurs when a value is out of an acceptable range.
  - For instance, exceeding the maximum call stack size.
  - Example:

```
function recursiveFunction() {
    recursiveFunction(); // Range error: Maximum call stack size exceeded
}
```

- **URI Error**:
  - Happens when invalid characters are used in a URI function.
  - Example:

```
decodeURIComponent("%"); // URI error: URI malformed
```

- **Internal Error**:
  - Rarely encountered directly by developers.
  - Occurs when the JavaScript engine faces internal issues (e.g., stack overflow).
  - Example:

```
// Causes an internal error due to excessive recursion
```

- Strict Mode in JavaScript
  - **Strict mode** is like those safety features for your JavaScript code.
  - It helps prevent common mistakes and makes your code more reliable.
  - **Why Use Strict Mode?**
    - **Error Prevention**:
      - It catches silent errors and turns them into loud exceptions.
      - For example, if you misspell a variable name, strict mode will warn you.
    - **Optimizations**:
      - It helps JavaScript engines optimize your code.
      - Strict mode code can sometimes run faster.
    - **Future-Proofing**:
      - It avoids using features that might change in the future.
  - **How to Enable Strict Mode:**
    - You add `"use strict";` at the beginning of your script or function.

1. **No Accidental Globals:**
    - Without strict mode, if you forget to declare a variable with `let`, `const`, or `var`, JavaScript creates a global variable. Strict mode prevents this mistake.

```
javascript
Copy code
"use strict";
x = 10; // Error: x is not defined
```

2. **No Deleting Variables:**
    - You can't delete variables or functions in strict mode, which helps prevent accidental mistakes.

```
javascript
Copy code
"use strict";
let y = 5;
delete y; // Error: Can't delete a variable
```

3. **No Duplicate Parameters:**
    - You can't have two function parameters with the same name, which can lead to confusing code.

```
javascript
Copy code
"use strict";
function sum(a, a) { // Error: Duplicate parameter names
  return a + a;
}
```

4. **Safer `this` Keyword:**
    - In strict mode, `this` in global contexts will be `undefined` instead of the global object, which prevents accidental mistakes.

- IsNaN
    - **NaN** stands for "Not-a-Number."
    - It's a special value in JavaScript that indicates something isn't a valid number.
    - When certain calculations or operations result in an undefined or non-representable numeric value, JavaScript returns NaN.
        - **Checking for NaN:**
    - You can use the `isNaN()` function or the `Number.isNaN()` method to check if a value is NaN.
    - Example:

```
let result = Math.sqrt(-1);
console.log(isNaN(result)); // true
```

- Type of Operator

- **Arithmetic Operators:**
  - Used for mathematical calculations.
  - Examples:
    - `+` (addition)
    - `-` (subtraction)
    - `*` (multiplication)
    - `/` (division)
    - `%` (modulus, remainder)
    - `**` (exponentiation)
- **Assignment Operators:**
  - Assign values to variables.
  - Examples:
    - `=` (assign)
    - `+=` (add and assign)
    - `-=` (subtract and assign)
    - `*=` (multiply and assign)
    - `/=` (divide and assign)
- **Comparison Operators:**
  - Compare values and return `true` or `false`.
  - Examples:
    - `==` (equal to)
    - `===` (equal value and type)
    - `!=` (not equal)
    - `>` (greater than)
    - `<` (less than)
    - `>=` (greater than or equal to)
    - `<=` (less than or equal to)
- **String Operators:**
  - Used for string manipulation.
  - Example:
    - `+` (concatenation)
- **Logical Operators:**
  - Combine conditions.
  - Examples:
    - `&&` (logical AND)
    - `||` (logical OR)
    - `!` (logical NOT)
- **Bitwise Operators:**
  - Perform bitwise operations on binary numbers.
  - Examples:
    - `&` (AND)
    - `|` (OR)
    - `^` (XOR)
    - `~` (NOT)
- **Ternary Operator:**
  - A concise way to write conditional statements.
  - Example:
    - `condition ? expr1 : expr2`

- Symbol()
    - In JavaScript, a **Symbol** is like that magical key—it's a special value that's always different from any other value.
  - **Why Use Symbols?**
    - Symbols are great for creating hidden properties in objects.
    - Think of them as secret labels you attach to your objects.
    - Nobody else can accidentally use the same label because each Symbol is unique.
  - **Creating a Symbol:**
    - You can create a Symbol like this:

```
const mySymbol = Symbol();
```

- BigInt
  - A **BigInt** is a special type of number in JavaScript.
  - It allows you to work with **huge integers** that are beyond the range of regular numbers.
  - Regular numbers (like `1`, `42`, or `3.14`) have a limit—they can't represent really large integers accurately.
  - BigInts solve this problem by providing a way to handle **arbitrarily large whole numbers**.

- **Variables and Scoping:**

Let vs Var

1. **Scope:**
   - `let`:
   - Block-scoped: Variables declared with `let` are only available within the block where they're defined.
   - Useful for loops, conditionals, and any block of code.
   - `var`:
   - Function-scoped: Variables declared with `var` are accessible throughout the entire function where they're declared.

2. **Hoisting:**
   - `let`:
   - Not hoisted: Variables declared with `let` are not initialized until their definition is evaluated.
   - `var`:

- Hoisted: `var` declarations are moved to the top of their scope and initialized with `undefined`.

3. **Reassignment:**

- `let`:
- Can be reassigned after initialization.
- `var`:
- Also mutable, but less predictable due to hoisting.

- Const vs Let

`const` **(Constant):**

- When you declare a variable with `const`, it signals that the identifier won't be reassigned.
- It's suitable for values that should remain constant throughout their scope.

`let` **(Mutable):**

- Variables declared with `let` can be reassigned.
- It's commonly used for loop counters or temporary variables.
- `let` variables are block-scoped (limited to the block where they're defined).
  - **Key Differences:**
    - `const`:
      - Cannot be reassigned after initialization.
      - Requires an initial value.
      - Prevents accidental reassignment.
    - `let`:
      - Can be updated (reassigned) within its scope.
      - Doesn't require an initial value.
      - Provides flexibility for changing values.

- Scope

- Type of Scope

Global Scope:

- Variables declared outside any function become **global**.
- They are accessible from anywhere in your code.

- Example:

```
let carName = "Volvo"; // Global scope
function myFunction() {
    // Can also use 'carName' here
}
```

Local (Function) Scope:

- Variables declared within a function are **local** to that function.
- They can only be accessed within the function.
- Example:

```
function myFunction() {
    let toy = "ball"; // Local scope
    // Can use 'toy' here
}
```

Block Scope (Introduced by ES6):

- Variables declared with `let` and `const` have **block scope**.
- They are limited to the block (within `{}`) where they're defined.
- Example:

```
{
    let x = 2; // Block scope
    // Cannot use 'x' outside this block
}
```

- Scope Chain

1. JavaScript looks for variables or functions in the current scope.
2. If not found, it climbs up the scope chain to outer scopes.
3. Eventually, it reaches the global scope (the whole neighborhood).

- let toy = "ball"; // Global scope
- function play() {
- let toy = "doll"; // Local scope
- console.log(toy); // "doll" (from local scope)
- }
- play(); // Calls the local scope function
- console.log(toy); // "ball" (from global scope)

- Block Scope

1. A block scope is like a room within your code.
2. Whenever you see `{curly brackets}`, it's a block.
3. We can only acess Let and const within that scope.sha
4. Variables declared with `let` and `const` keywords have block scope.

5. Example:

JavaScript

```
{
    let x = 10; // Inside this block
    console.log(x); // Prints 10
}
// x is not accessible here
```

- **Why Is It Useful?**
  - Block scope helps prevent conflicts between variables.
  - Variables declared inside a block are only visible within that block.
  - It enhances code organization and modularity.

- Shadowing

1. When you declare a variable inside a function (inner scope) with the same name as a variable outside (outer scope), you're shadowing.
2. The inner variable takes precedence, hiding the outer one.

```
let name = "John"; // Outer variable

function greet() {
let name = "Alice" ; // Inner variable
console.log("Hello, " + name); // Prints "Hello, Alice"
}

greet();
```

- Hoisting

Hoisting means that variables and function declarations are moved to the top of their containing scope (either the global scope or function scope) before the code is executed. This is why you can access variables and functions before they are actually declared in the code.

1) greet(); // This works fine because the function is hoisted.

function greet() {

console.log("Hello!");

}

2) console.log(myVar); // Output: undefined (the declaration is hoisted, but not the value)
var myVar = 10;

If you use `let` or `const` instead of `var`, you'll get a `ReferenceError` because these variables are hoisted but not initialized, so they are in a "temporal dead zone" until the code reaches their declaration.

- Temporal Dead Zone (TDZ)

  const and let are in a Temporal Dead Zone until they are initialized some value. You might encounter SyntaxError, TypeError or ReferenceError while using them in your code.

  The **Temporal Dead Zone (TDZ)** is the area in a block of code where a variable is **declared** but **not yet initialized**. During this period, the variable is inaccessible, and any attempt to access it will result in a `ReferenceError`.

  ## Key Points:

1. **Hoisting:** Variables declared with `let` and `const` are hoisted to the top of their block scope, but they remain uninitialized until the code execution reaches the line where they are actually declared. This uninitialized state is what creates the TDZ.
2. **Initialization:** The TDZ ends when the variable is initialized, which occurs at the line where the variable is declared. After this point, the variable can be accessed normally.
3. **ReferenceError:** If you try to access a variable while it's in the TDZ (before it's initialized), you'll get a `Reference Error`.

- Lexical Scope

  What is Lexical Scope?

  - Lexical scope refers to the **definition area** of a variable or function.
  - In simpler terms, it's where a variable or function **gets created**.
  - Another name for lexical scope is **static scope**.
  - The place where an item is **invoked (called)** is not necessarily its lexical scope.

- Lexical Environment

1. **What is a Lexical Environment?**

a. A **lexical environment** is a data structure that stores all variables and function declarations within a specific scope.
b. It helps the interpreter determine which variables and functions are accessible in different parts of your program.
c. A lexical environment consists of two key components:
    i. **Environment Record**: An object that stores variables and functions declared within the current scope.
    ii. **Reference to the Outer Environment**: A pointer to the parent lexical environment, allowing access to variables and functions from outer scopes.
d. Whenever a function or block is executed, a new lexical environment is created.

- **Functions:**

  - Function Statement

    A **function definition** (also known as a **function declaration** or **function statement**) consists of the following components:

    1. **The `function` Keyword**:
        1. The `function` keyword is used to declare a function.
    2. **Function Name**:
        1. After the `function` keyword, we provide the **name of the function**.
        2. The function name follows the same naming rules as JavaScript variables (e.g., no spaces, starts with a letter or underscore, etc.).
    3. **Parameters**:
        1. Inside parentheses following the function name, we list the **parameters** (also called **arguments**) that the function accepts.
    4. **Function Body**:
        1. The function body is enclosed in curly braces `{ }`.
        2. Inside the braces, we write the JavaScript statements that define what the function does.
        3. These statements execute when the function is invoked (called).

  - Function Expression
    - A **function expression** in JavaScript is a way to define a function inside an expression. Unlike function declarations, which are hoisted and can be called before their definition in the code, function expressions are created when the execution reaches them. Here's a simple example:

```
const greet = function(name) {
  return `Hello, ${name}!`;
};

console.log(greet("Alice")); // Output: Hello, Alice!
```

- Anonymous Function

An **anonymous function** in JavaScript is simply a function without a name. Unlike regular named functions, which have an identifier, anonymous functions are defined using only the `function` keyword without any associated name.

```
const greet = function() {
  console.log("Welcome to GeeksforGeeks!");
};
```

https://youtu.be/KRjfdmsw9XQ?si=sP6J2MUBb4C6e6Z2

- Named Function Expression

A **Named Function Expression** in JavaScript is a function expression that has a name or identifier.

- First Class Function

A **First-Class Function** in JavaScript means that functions are treated like any other variable. You can:

- Store functions in variables.
- Pass them as arguments to other functions.
- Return them from functions.

```
const sayHello = function() {
    return "Hello!";
};

console.log(sayHello()); // Output: Hello!
```

Here, `sayHello` is a function stored in a variable, showing that functions are first-class citizens in JavaScript.

- First Order Function

A First Order Function in JavaScript is a regular function that doesn't take another function as one of its parameters and doesn't return another function within its body1. In simpler terms:

- It accepts parameters (of different data types, either primitive or non-primitive).
- It may or may not return a value based on the calculations performed on the passed parameters.

o Here's an example of a general first-order function in JavaScript:

```
function multiply(num1, num2) {
```

```
    console.log(num1 * num2);
}

multiply(4, 35); // Output: 140
```

- Higher Order Function

In JavaScript, a **higher-order function** (HOF) is a function that can either:

1. **Accept other functions as arguments**: HOFs allow dynamic behavior by customizing their actions based on the function passed as an argument. For example:

```
function higherOrderFunction(callback) {
  // Perform some operations
  console.log("Executing the higher order function...");
  // Call the callback function
  callback();
}

function callbackFunction() {
  console.log("Executing the callback function...");
}

higherOrderFunction(callbackFunction);
```

2. **Return a function as a result**: HOFs create functions dynamically based on conditions or parameters. For instance:

```
function createGreeter(greeting) {
  // Return a new function
  return function(name) {
    console.log(`${greeting}, ${name}!`);
  };
}

const greetHello = createGreeter("Hello");
greetHello("John"); // Output: Hello, John!

const greetGoodbye = createGreeter("Goodbye");
greetGoodbye("Alice"); // Output: Goodbye, Alice!
```

Higher-order functions enable abstraction, composition, and more flexible, reusable code.

- Arrow Functions vs Normal Functions
    - **Arrow Functions**:

- Introduced in ES6 (ECMAScript 2015).
- Concise syntax: `(parameters) => expression`.
- Lexical `this` binding: Inherits `this` from the surrounding context.
- Ideal for simple, one-liner functions.
- Example:

```
const multiply = (num1, num2) => num1 * num2;
```

- **Normal Functions**:
  - Traditional function syntax: `function functionName(parameters) { ... }`.
  - Create their own `this` binding based on the caller.
  - More flexible, suitable for complex logic and constructors.
  - Example:

```
function multiply(num1, num2) {
  return num1 * num2;
}
```

In summary, arrow functions are concise and great for straightforward tasks, while normal functions offer more flexibility and broader use cases

- Arrow Functions
  - Arrow functions are anonymous functions defined using the `=>` syntax.
  - They are often used for short, one-liner functions.
  - If the function has only one statement, you can omit the curly braces and the `return` keyword.
  - Unlike regular functions, arrow functions don't have their own `this`, `arguments`, or `super`.
  - They are commonly used as callbacks for methods like `map`, `filter`, and `sort`

- Generator Functions

Generator functions in JavaScript are special functions that can generate a sequence of values. When called, they return a **Generator Object** that follows the Iterable Protocol of ES6, similar to iterators[1]. Here's a brief overview:

- A generator function is defined using the `function*` syntax.
- It uses the `yield` keyword to generate values, pausing execution and sending values to the caller.
- The generator retains its state, allowing it to resume execution after each `yield`, continuing immediately after the last `yield` run.

Example of a generator function:

```
function* gen() {
  yield 1;
```

```
  yield 2;
  // ...
}
const iterator = gen();
console.log(iterator.next().value); // Output: 1
console.log(iterator.next().value); // Output: 2
// ...
```

- Self-Invoking Functions

  Self-invoking functions in JavaScript are code snippets that execute immediately when called upon. They are also known as **self-executing functions**. Here's a simple definition:

  - A self-invoking function is defined as an anonymous function enclosed within parentheses, followed by another set of parentheses.
  - It starts automatically without being explicitly called.
  - You cannot self-invoke a function declaration; it must be a function expression.

  Example of a self-invoking function:

```
(function() {
  let x = "Hello!";
  // I will invoke myself
})();
```

- Function Object

  In JavaScript, a **function object** is a special type of object that represents a function. Here's a simple definition:

- A function object includes a string that holds the actual code (the function body) of the function.
- Functions are first-class objects, which means they can be passed to other functions, returned from functions, and assigned to variables and properties.
- What distinguishes them from other objects is that functions can be **called**.

  In summary, think of function objects as callable "action objects" that allow you to execute specific code when invoked

- Callback Functions

  a **callback function** is a function that is passed as an argument to another function. When using a callback, you can call a function (let's call it the "parent" function) with a callback function, and the parent function will execute the callback after a specific task or operation is completed. Here's a simple example:

```
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
```

```
}

function myCalculator(num1, num2, myCallback) {
  let sum = num1 + num2;
  myCallback(sum);
}

myCalculator(5, 5, myDisplayer);
```

In this example:

- `myDisplayer` is the callback function.
- It is passed to `myCalculator` as an argument.
- After calculating the sum of `5` and `5`, `myCalculator` calls the `myDisplayer` callback to display the result in an HTML element with the ID "demo."

- Callback Hell

Callback Hell occurs when there are too many nested callback functions in asynchronous JavaScript code. This can make the code difficult to read, maintain, and debug. Callbacks are functions that are executed after another function has finished running, often used for asynchronous operations like API calls or reading files.

In **Callback Hell**, the code becomes like a pyramid structure due to multiple levels of nested callbacks, making it harder to follow. Here's an example:

```
doSomething(function(result) {
    doSomethingElse(result, function(newResult) {
        doAnotherThing(newResult, function(finalResult) {
            console.log(finalResult);
        });
    });
});
```

To avoid **Callback Hell**, you can use modern JavaScript features like **Promises** and **async/await**. Here's how the same logic can be handled with Promises:

```
doSomething()
.then(result => doSomethingElse(result))
.then(newResult => doAnotherThing(newResult))
.then(finalResult => console.log(finalResult))
.catch(error => console.error(error));
```

Using `async/await`:

```
async function handleTasks() {
      try {
            const result = await doSomething();
            const newResult = await doSomethingElse(result);
```

```
        const finalResult = await doAnotherThing(newResult);
        console.log(finalResult);
    } catch (error) {
        console.error(error);
    }
}
```

- This structure is much cleaner and easier to understand compared to deeply nested callbacks.

- Async vs Sync Functions
  - **Synchronous Functions**:
    - Execute sequentially, one line at a time.
    - Each operation waits for the previous one to complete.
    - Example:

```
console.log('One');
console.log('Two');
console.log('Three');
// Output: 'One', 'Two', 'Three'
```

  - **Asynchronous Functions**:
    - Execute in parallel or concurrently.
    - Operations can occur simultaneously.
    - Commonly used for tasks like fetching data, handling events, or making network requests.
    - Examples include Promises, async/await, and setTimeout.

- Promise

A **Promise** in JavaScript is a special object used to handle asynchronous operations. It's a way to deal with things like fetching data from a server or performing any task that takes time. Promises help you manage these tasks without blocking the rest of your code.

1. What is a Promise?

- A Promise is like a "promise" you make that something will be done in the future.
- It represents a value that may not be available yet but will be resolved at some point.

2. States of a Promise

A Promise can be in one of three states:

1. **Pending**: The initial state, where the operation hasn't finished yet.

2. **Fulfilled**: The operation was successful, and the promise has a result (e.g., data from a server).
3. **Rejected**: The operation failed, and the promise has an error.

3. How to Create a Promise

Here's a simple example of how to create a Promise:

```
let myPromise = new Promise(function(resolve, reject) {
  // Simulating a task, like fetching data from a server
  let success = true; // You can change this to 'false' to see rejection

  if (success) {
    resolve("Task completed successfully!"); // Fulfilled
  } else {
    reject("Task failed!"); // Rejected
  }
});
```

4. Using a Promise

Once you have a promise, you can handle it using `.then()` for success (fulfilled) and `.catch()` for failure (rejected):

```
myPromise
  .then(function(result) {
    console.log(result); // If promise is fulfilled
  })
  .catch(function(error) {
    console.log(error); // If promise is rejected
  });
```

5. Why Use Promises?

Promises help make your code cleaner and easier to manage when dealing with asynchronous tasks like:

- Fetching data from a server.
- Waiting for user input.
- Any task that takes time but shouldn't block the rest of your code.

Simple Example

Imagine you're ordering a pizza:

1. **Pending**: The pizza is being prepared.
2. **Fulfilled**: The pizza is ready and delivered to you.
3. **Rejected**: Something went wrong, and the pizza can't be delivered.

```
let pizzaOrder = new Promise(function(resolve, reject) {
  let pizzaReady = true; // Pizza is ready!
```

```
  if (pizzaReady) {
    resolve("Your pizza is here!"); // Success
  } else {
    reject("Sorry, we're out of pizza."); // Failure
  }
});

pizzaOrder
  .then(function(message) {
    console.log(message); // "Your pizza is here!"
  })
  .catch(function(error) {
    console.log(error); // If something went wrong
  });
```

1.States of a Promise:

1. **Pending**: Initial state (waiting for completion).
2. **Fulfilled**: Operation completed successfully (value available).
3. **Rejected**: Operation failed (error occurred).

- **How to Use Promises:**
  1. Create a Promise with `new Promise(executor)`.
  2. The `executor` function defines the async task.
  3. Attach `.then()` to handle success and `.catch()` for errors.

Example:

```
const fetchData = new Promise((resolve, reject) => {
  // Simulate fetching data (async operation)
  setTimeout(() => {
    const data = "Hello, Promises!";
    resolve(data); // Success
    // reject(new Error("Oops! Something went wrong.")); // Error
  }, 2000);
});

fetchData
  .then(result => console.log(result))
  .catch(error => console.error(error.message));
```

4. **Why Use Promises?**
   1. Better readability for async code.
   2. Avoids callback hell (nested callbacks).
   3. Enables structured error handling.

- Promise States

In JavaScript, a **Promise** represents the eventual completion (or failure) of an asynchronous operation and its resulting value[1]. Let's break down the states of a Promise:

1. **Pending**:

    1. Initial state when the promise is created.
    2. Neither fulfilled nor rejected yet.
2. **Fulfilled**:
    1. Indicates that the operation completed successfully.
    2. The promise has a value associated with it.
3. **Rejected**:
    1. Indicates that the operation failed (an error occurred).
    2. The promise has a reason (error) associated with it.

Remember, promises allow you to handle asynchronous tasks more effectively, making your code cleaner and easier to reason about

- Promise Chain

Promise chaining in JavaScript allows you to handle a sequence of asynchronous tasks one after another. It's a powerful way to manage complex code. Let's break it down:

1. **Basic Promise Chaining**:
    1. You create a chain of `.then()` handlers on a promise.
    2. Each `.then()` returns a new promise.
    3. The result from one `.then()` is passed to the next.
    4. Example:

```
new Promise(function(resolve, reject) {
  setTimeout(() => resolve(1), 1000);
})
  .then(function(result) {
    alert(result); // Shows 1
    return result * 2;
  })
  .then(function(result) {
    alert(result); // Shows 2
    return result * 2;
  })
  .then(function(result) {
    alert(result); // Shows 4
  });
```

2. **Returning Promises**:
    1. A `.then()` handler can create and return a new promise.
    2. The next `.then()` waits for that promise to settle.
    3. Example:

```
new Promise(function(resolve, reject) {
  setTimeout(() => resolve(1), 1000);
})
  .then(function(result) {
    alert(result); // Shows 1
    return new Promise((resolve, reject) => {
      setTimeout(() => resolve(result * 2), 1000);
    });
```

```
})
.then(function(result) {
  alert(result); // Shows 2
});
```

Remember, promise chaining simplifies handling asynchronous operations, making your code more organized and readable!

- async/await

Async/await in JavaScript is a powerful way to handle asynchronous operations with more readable and synchronous-like syntax. Let's break it down:

1. **Async Functions**:
    1. You declare a function as `async` using the `async` keyword.
    2. An `async` function always returns a promise, even if you don't explicitly return one.
    3. Example:

```
async function fetchData() {
  return 1;
}
```

2. **Await Keyword**:
    1. Used inside `async` functions to wait for a promise to settle (resolve or reject).
    2. Pauses function execution until the promise resolves.
    3. Example:

```
async function fetchAndProcessData() {
  let response = await fetch('/data.json');
  let data = await response.json();
  // Process data...
}
```

3. **Benefits**:
    1. Simplifies asynchronous code, making it appear more linear.
    2. Easier to read and write than using `.then()` with promises.
    3. Allows handling errors with `try`/`catch`.

Remember, `async/await` enhances code readability and helps manage complex asynchronous tasks effectively!

- Factory Function

a **factory function** is a function that returns an object. It's a way to create and customize objects in a controlled manner. Here's a simple explanation:

1. **What is a Factory Function?**
    1. A factory function creates and returns new objects.
    2. It allows you to set specific properties and behaviors for those objects.
    3. Unlike constructors or classes, it doesn't use `new` or `this`.
2. **Example: Creating a Person Object**

```
function createPerson(firstName, lastName) {
  return {
    firstName: firstName,
    lastName: lastName,
    getFullName() {
      return `${firstName} ${lastName}`;
    },
  };
}

const person1 = createPerson('John', 'Doe');
const person2 = createPerson('Jane', 'Doe');

console.log(person1.getFullName()); // Output: John Doe
console.log(person2.getFullName()); // Output: Jane Doe
```

3. **Advantages of Factory Functions:**
    1. Avoids code duplication when creating similar objects.
    2. Provides a clean way to manage object creation.
    3. Allows efficient memory usage by sharing methods.

By using factory functions, you can create multiple objects without repeating the same code

- Constructor Function

constructor function is used to create and initialize objects. It acts as a blueprint for creating multiple instances of similar objects. Here's a simple explanation:

1. **What is a Constructor Function?**
    1. A constructor function defines properties and behaviors for objects.
    2. It is called using the `new` keyword to create new instances.
    3. By convention, constructor function names start with an uppercase letter.
2. **Example: Creating a Person Object**

```
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
  this.getFullName = function() {
    return `${firstName} ${lastName}`;
  };
}

const person1 = new Person('John', 'Doe');
const person2 = new Person('Jane', 'Doe');
```

```
console.log(person1.getFullName()); // Output: John Doe
console.log(person2.getFullName()); // Output: Jane Doe
```

3. **Advantages of Constructor Functions:**
    1. Create multiple objects with shared properties and methods.
    2. Customize each instance by passing arguments.
    3. Encapsulate logic within the constructor.

Remember, constructor functions allow you to create consistent and reusable object structures

- This Keyword

the `this` keyword is a powerful concept that refers to the context in which a piece of code (usually a function) is executed. Let's explore its behavior:

1. **Context-Based Reference**:
    - `this` represents the object that is currently executing the function.
    - Its value depends on how and where the function is called.
    - Different scenarios determine what `this` refers to.
2. **Common Scenarios for** `this`:

- **In an Object Method**:
- Inside an object method, `this` refers to the object itself.
- Example:

```
const person = {
  firstName: "John",
  lastName: "Doe",
  getFullName() {
    return this.firstName + " " + this.lastName;
  }
};
console.log(person.getFullName()); // Output: John Doe
```

- **Alone (Global Scope)**:
    1. When used alone (outside any function or method), `this` refers to the global object (usually the browser window or Node.js global context).
    2. Example:

```
let x = this; // Refers to the global object
```

- **In a Function (Default)**:
    1. Inside a regular function, `this` also refers to the global object.
    2. Example:

```
function myFunction() {
  return this; // Refers to the global object
}
```

- **In a Function (Strict Mode)**:

- In strict mode, when used alone in a function, `this` is `undefined`.
- Example:

```
"use strict";
function myFunction() {
  return this; // Refers to undefined
}
```

- **In Event Handlers**:
- In HTML event handlers (e.g., `onclick`), `this` refers to the HTML element that received the event.
- Example:

HTML

```
<button onclick="this.style.display='none'">Click to Remove Me!</button>
```

3. **Explicit Function Binding**:
- Methods like `call()`, `apply()`, and `bind()` allow you to explicitly set `this` to a specific object.
- Example using `call()`:

```
const person1 = { fullName: function() { return this.firstName + " " + this.lastName; } };
const person2 = { firstName: "John", lastName: "Doe" };
person1.fullName.call(person2); // Returns "John Doe"
```

Remember, understanding `this` is crucial for effective JavaScript programming, especially when dealing with object-oriented patterns and event handling

- Call, Bind, Apply (Clarification needed on Bind)
  - These methods are used to manipulate the `this` context in JavaScript functions.
  - `bind`: Creates a new function with a specific `this` value and fixed arguments.
  - `call`: Invokes a function with a specified `this` value and individual arguments.
  - `apply`: Invokes a function with a specified `this` value and an array of arguments.
  - Example:

```
const person = {
  name: 'Alice',
  greet: function (message) {
    console.log(`${message}, ${this.name}!`);
  },
};

const greetAlice = person.greet.bind(person);
greetAlice('Hi'); // Output: Hi, Alice!
```

```
person.greet.call({ name: 'Bob' }, 'Hello'); // Output: Hello, Bob!
person.greet.apply({ name: 'Charlie' }, ['Hey']); // Output: Hey, Charlie!
```

Remember, understanding these concepts will enhance your JavaScript skills and make your code more efficient and expressive

- Currying

- Currying is a concept from lambda calculus, but don't worry—it's quite simple to implement in JavaScript.
- Currying transforms a function that takes multiple arguments into a sequence of functions, each expecting one argument.
- Instead of taking all arguments at once, a curried function takes the first argument and returns a new function. This new function takes the second argument and returns another new function, and so on, until all arguments are consumed.
- Example:

```
// Noncurried version
const add = (a, b, c) => {
  return a + b + c;
};
console.log(add(2, 3, 5)); // Output: 10

// Curried version
const addCurry = (a) => {
  return (b) => {
    return (c) => {
      return a + b + c;
    };
  };
};
console.log(addCurry(2)(3)(5)); // Output: 10
```

- Currying helps make code more readable, avoids repetitive variable passing, and promotes functional programming practices

- Default Parameters
    - Default parameters allow you to specify default values for function arguments.
    - If an argument is not provided when the function is called, the default value is used.
    - Example:

```
function greet(name = 'Guest') {
  console.log(`Hello, ${name}!`);
}
greet(); // Output: Hello, Guest!
greet('Alice'); // Output: Hello, Alice!
```

- Rest Operator
- Spread Operator

  The spread operator (`...`) in JavaScript is commonly used to create a shallow copy of an array or object. When applied to an array, it spreads out the elements into a new array. When used with an object, it creates a new object with the same properties as the original. It's a handy way to duplicate data without modifying the original.

- Rest Parameter
- Pass by Value vs Pass by Reference
- Closure

1. What is a Closure?

A closure is created when a function is defined inside another function (or block of code), and the inner function "closes over" or remembers the variables from its outer function. This allows the inner function to access and manipulate those variables, even after the outer function has finished executing.

2. How Closures Work

Closures occur because of how JavaScript handles variable scope. When a function is created, it remembers the environment in which it was created, including any variables that were in scope at that time. This is known as **lexical scoping**.

Here's an example to illustrate this:

```
function outerFunction() {
  let outerVariable = "I'm from the outer function";

  function innerFunction() {
    console.log(outerVariable); // The inner function can access the
outer function's variable
  }

  return innerFunction;
}

const closureExample = outerFunction(); // outerFunction() returns
innerFunction
closureExample(); // Output: "I'm from the outer function"
```

In this example:

- `outerFunction` defines a variable `outerVariable` and a function `innerFunction`.
- `innerFunction` accesses the `outerVariable` from its outer scope.
- Even after `outerFunction` finishes execution, the `closureExample` function (which is `innerFunction`) still "remembers" and has access to `outerVariable` because of the closure.

Closures are a fundamental concept in JavaScript, enabling functions to retain access to variables from their outer scope even after that scope has exited. This feature is essential for creating private variables, function factories, event handlers, and more. Understanding closures will improve your ability to write modular, flexible, and efficient JavaScript code.

- Immediately Invoked Function Expression (IIFE)

  An **Immediately Invoked Function Expression (IIFE)** is a JavaScript function that is executed immediately after it is defined. The key characteristic of an IIFE is that it is invoked immediately, without waiting for it to be called elsewhere in the code.

  Used for solve the problem of variable scope

  IIFE uses :-

  - creating private scopes
  - avoiding global variable pollution
  - preventing name collisions
  - encapsulating code (module pattern)
  - managing async operations.

  ## 1. Syntax of an IIFE

  An IIFE is a function expression that is immediately executed. The basic syntax looks like this:

```
 (function() {
  // Code here runs immediately
})();
```

  Here's what happens:

  - The function is wrapped in parentheses to turn it into a function expression (because normally a function declaration cannot be invoked immediately).
  - Immediately after the function expression, `()` invokes the function.

  Another variation is to use an arrow function:

```
 (() => {
  // Code here runs immediately
})();
```

2. Example of an IIFE

```
 (function() {
  const message = "Hello, I'm an IIFE!";
  console.log(message);
})(); // Output: "Hello, I'm an IIFE!"
```

- The function is defined and immediately invoked.
- The code inside the function executes right away, logging the message.

3. Why Use an IIFE?

IIFEs are often used for various reasons in JavaScript development:

1. Avoid Polluting the Global Namespace

IIFEs create a new scope and keep variables defined inside them out of the global scope. This prevents potential conflicts with other variables or functions in your code, making it especially useful in modular programming.

```
 (function() {
  const myVar = "I won't conflict with any global variables";
  console.log(myVar);
})();
```

Here, `myVar` is scoped to the IIFE and does not pollute the global namespace.

2. Data Privacy

IIFEs allow you to encapsulate and protect variables from being accessed or modified from outside the function. This can be used to create private variables or functions that are only accessible within the IIFE.

```
const counter = (function() {
  let count = 0; // private variable

  return {
    increment: function() {
      count++;
      console.log(count);
    },
    decrement: function() {
      count--;
      console.log(count);
    }
  };
})();

counter.increment(); // Output: 1
counter.increment(); // Output: 2
counter.decrement(); // Output: 1
```

In this example, `count` is private and can only be modified by the `increment` and `decrement` methods.

- **Objects and Classes:**

1. Object

An object is a collection of related data stored as key-value pairs. The keys (also called properties) are like labels, and the values are the actual data. You can create an object using curly braces `{}` or the `Object()` function.

```
const car = { brand: "Toyota", year: 2020 };
```

2. Object Destructuring

Object destructuring is a way to easily pull out specific values from an object and assign them to variables.

```
const car = { brand: "Toyota", year: 2020 };
const { brand, year } = car;  // brand = "Toyota", year = 2020
```

3. Nested Destructuring

If an object has another object inside it, you can use destructuring to get values from both the outer and inner objects.

```
const car = { brand: "Toyota", model: { name: "Corolla", year: 2020 } };
const { model: { name, year } } = car;  // name = "Corolla", year = 2020
```

4. Object.defineProperty vs defineProperties

- `Object.defineProperty()` is used to add or change one property of an object.
- `Object.defineProperties()` is used to add or change multiple properties at once.

```
Object.defineProperty(obj, "age", { value: 25 });
Object.defineProperties(obj, { height: { value: 170 }, weight: { value: 65
} });
```

5. Object.fromEntries vs Object.entries()

- `Object.entries()` turns an object into an array of key-value pairs.
- `Object.fromEntries()` takes an array of key-value pairs and turns it into an object.

```
const obj = { name: "John", age: 30 };
Object.entries(obj);  // [["name", "John"], ["age", 30]]
Object.fromEntries([["name", "John"], ["age", 30]]);   // { name: "John",
age: 30 }
```

6. Object.freeze

`Object.freeze()` locks an object so no one can add, remove, or change its properties.

```
const obj = { name: "John" };
Object.freeze(obj);
obj.name = "Jane";  // This won’t work.
```

7. Object.seal

`Object.seal()` stops new properties from being added or existing properties from being deleted, but you can still change the values of existing properties.

```
const obj = { name: "John" };
Object.seal(obj);
obj.name = "Jane";  // This works.
delete obj.name;  // This won't work.
```

8. Check if Object is Frozen or Sealed

- `Object.isFrozen()` checks if the object is locked and cannot be changed.
- `Object.isSealed()` checks if new properties can be added or deleted.

```
Object.isFrozen(obj);  // true or false
Object.isSealed(obj);  // true or false
```

9. Prototypal vs Classical Inheritance

- **Prototypal Inheritance**: Objects inherit directly from other objects in JavaScript.
- **Classical Inheritance**: In languages like Java, classes are used for inheritance. JavaScript uses prototypes instead of classes.

```
const car = { brand: "Toyota" };
const myCar = Object.create(car);  // myCar inherits from car.
```

10. Prototype Chaining

When you try to access a property on an object, and it's not there, JavaScript looks for it in the object's prototype. This is called prototype chaining.

11. `__proto__` vs `prototype`

- `__proto__` is the link to the object that your current object inherits from.
- `prototype` is a property of a function that defines the blueprint for new objects created by that function.

12. Classes

Classes are a way to create objects using a more structured method introduced in ES6.

```
class Car {
  constructor(brand, year) {
    this.brand = brand;
    this.year = year;
  }
}
```

```
const myCar = new Car("Toyota", 2020);
```

13. Inheritance

Inheritance allows a new class to use properties and methods of an existing class.

```
class Animal {
  speak() { console.log("Animal sound"); }
}
class Dog extends Animal {}
const myDog = new Dog();
myDog.speak();  // "Animal sound"
```

14. Constructor

The constructor is a function that sets up the initial values for a new object created from a class.

15. Accessor (Getter and Setter)

Getter and setter functions let you control access to the properties of an object. You use them to read or update the properties indirectly.

```
class Person {
  constructor(name) {
    this._name = name;
  }
  get name() {
    return this._name;
  }
  set name(value) {
    this._name = value;
  }
}
```

16. Enhanced Object Literals

This allows you to write object properties in a shorter form, where the property name is the same as the variable name, and you can also define methods directly.

```
const brand = "Toyota";
const car = {
  brand,
  drive() { console.log("Driving..."); }
};
```

17. Collections in JS (Map, WeakMap, Set, WeakSet)

- **Map**: A collection of key-value pairs where keys can be of any type.

- **WeakMap**: Similar to Map, but keys are weakly held (helping with memory management).
- **Set**: A collection of unique values.
- **WeakSet**: Similar to Set, but only objects are allowed as values, and they are weakly held.

18. Object vs Map

- **Object**: Key-value pairs, but the keys can only be strings or symbols.
- **Map**: Allows keys of any type (e.g., numbers, objects), and it's more efficient for large data collections.

19. Map and HashMap

In JavaScript, `Map` functions similarly to a hashmap found in other programming languages. It allows for any type of key and is more efficient for storing large amounts of key-value data.

- **Strings and Arrays:**

1. String Functions

These are built-in methods to work with and change strings. Some examples include:

- `charAt()`: Returns the character at a specific position in the string.
- `toUpperCase()`: Changes all characters in the string to uppercase.
- `concat()`: Joins two or more strings together.

2. String Interpolation

String interpolation lets you insert variables or calculations inside a string easily. You use backticks (`` ` `` `instead of quotes, and insert variables inside` `${}`` `.

```
const name = "John";
console.log(`Hello, ${name}`);  // "Hello, John"
```

3. Tagged Template

A more advanced feature of template literals where you can define a custom function (called a "tag") that processes the interpolated values before inserting them into the string.

4. Template Literals

A cleaner way to create strings that can span multiple lines and contain variables or expressions. Use backticks instead of single (`'`) or double (`"`) quotes.

```
const greeting = `Hello,
this is a template literal!`;
```

5. Trim

The `trim()` method removes any extra spaces from the start and end of a string. It's helpful when cleaning up user input or data.

```
const userInput = "  Hello  ";
console.log(userInput.trim());  // "Hello"
```

6. Slice vs Splice

- `slice()`: Extracts a part of a string or array without changing the original.
- `splice()`: Adds or removes elements from an array and changes the original array.

```
const arr = [1, 2, 3, 4];
arr.slice(1, 3);  // [2, 3], original stays the same
arr.splice(1, 2);  // removes elements, original array is modified
```

7. Map, Reduce, Filter, forEach

These are functions used to handle arrays:

- `map()`: Transforms each element in an array and returns a new array.
- `reduce()`: Combines array elements into a single value (e.g., sum).
- `filter()`: Creates a new array with only elements that meet certain conditions.
- `forEach()`: Loops through each element in an array and performs an action on them (doesn't return anything).

8. For...of vs For...in

- `for...of`: Loops over the values of an array or other iterable.
- `for...in`: Loops over the keys or properties of an object.

9. Array Methods: Push, Pop, Shift, Unshift

These methods let you add or remove items from an array:

- `push()`: Adds an item to the end of an array.
- `pop()`: Removes the last item.
- `shift()`: Removes the first item.
- `unshift()`: Adds an item to the beginning.

10. Static vs Dynamic Array

- **Static Array**: Has a fixed size, meaning you cannot add or remove items once it's created.
- **Dynamic Array**: Can grow or shrink in size as you add or remove items. In JavaScript, all arrays are dynamic.

11. Shallow Copy vs Deep Copy

- **Shallow Copy**: Copies only the outer object, so any nested objects still refer to the original.

- **Deep Copy**: Copies everything, including nested objects, so changes to the copy won't affect the original.

```
const arr = [1, { a: 2 }];
const shallowCopy = [...arr];  // shallow copy
const deepCopy = JSON.parse(JSON.stringify(arr));  // deep copy
```

12. Array Sorting Functions

JavaScript allows you to sort arrays using the `sort()` method, which can sort alphabetically or by a custom sorting function you provide.

```
const numbers = [3, 1, 4, 2];
numbers.sort((a, b) => a - b);  // sorts numbers in ascending order
```

- **Control Structures:**

1. Loops

Loops help you run a piece of code multiple times. In JavaScript, there are three main types:

- **For loop**: Runs a block of code a set number of times.
- **While loop**: Keeps running as long as a condition is true.
- **Do...while loop**: Similar to a while loop, but it runs the code at least once, even if the condition is false.

2. Conditional (Ternary) Operator

The ternary operator (`? :`) is a shorthand way to write an if-else condition. It checks a condition and gives one value if it's true and another value if it's false.
Example:

```
let isAdult = age >= 18 ? "Yes" : "No";
```

3. Default Parameters

When you create a function, you can give default values to the parameters. If no value is passed when the function is called, the default value will be used instead.
Example:

```
function greet(name = "Guest") {
  console.log("Hello, " + name);
}
greet();  // "Hello, Guest"
```

4. Rest Operator

The rest operator (. . .) allows you to group multiple arguments into an array. It is useful when you don't know how many arguments a function will receive.
Example:

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

5. Unary Operator

Unary operators are operators that work with just one value or operand. Some examples are:

- `+` (positive),
- `-` (negative),
- `typeof` (checks the data type),
- `!` (logical NOT, which flips true to false or false to true).

6. Comma Operator

The comma operator lets you evaluate several expressions and only returns the result of the last one. It's not commonly used, but you might see it in more complex code.
Example:

```
let a = (1, 2, 3);  // a becomes 3
```

7. Nesting Templates

This refers to using one template literal (with backticks) inside another. It's a way to build more complex strings with dynamic values or expressions inside them.
Example:

```
const name = "John";
const message = `Hello, ${`Mr. ${name}`}!`;  // "Hello, Mr. John!"
```

- **Event Handling:**

1. Event Listeners

Event listeners are functions that wait for something to happen, like a mouse click or a key press on an HTML element. When that event happens, the listener runs a function to handle it. For example, you can add a click listener to a button so that something happens when the button is clicked.

2.Event Loop

JavaScript is single-threaded, meaning it can execute only one operation at a time. However, it allows asynchronous tasks (like network requests, file reading, timers, etc.) using the event loop to avoid blocking the main thread.

How It Works:

1. **Call Stack**: JavaScript runs code in a stack, which is called the call stack. It executes functions sequentially. If a function is called, it gets added to the stack. When the function completes, it gets removed from the stack.
2. **Web APIs/Task Queue**: When an asynchronous function (e.g., `setTimeout`, `fetch`) is called, JavaScript sends the task to the Web API environment (provided by the browser). After the operation is completed, the callback (function to be executed) is moved to the task queue.
3. **Event Loop**: The event loop continuously checks the call stack. If the stack is empty and there are tasks in the task queue, the event loop pushes the first task from the queue onto the call stack to be executed. This allows JavaScript to handle asynchronous tasks without blocking the main thread.

Example:

```
console.log('Start');

setTimeout(() => {
  console.log('Callback from setTimeout');
}, 1000);

console.log('End');
```

- `console.log('Start')` and `console.log('End')` execute synchronously in the call stack.
- The `setTimeout` callback is sent to the Web API. After 1 second, it moves to the task queue.
- Once the call stack is empty, the event loop moves the callback from the task queue to the call stack, and it gets executed.

In Summary:

The event loop manages how JavaScript handles asynchronous tasks by coordinating the call stack and task queue, allowing non-blocking execution in a single-threaded environment.

3. Event Propagation (Bubbling and Capturing)

When an event happens, like a click, it can move through the page in two ways:

- **Capturing phase**: The event starts at the top (root) of the page and moves down to the element that was clicked.
- **Bubbling phase**: After reaching the clicked element, the event goes back up the page. You can control whether to handle the event during capturing or bubbling using a parameter in the event listener.

4. Stop Event Propagation

Sometimes, you might not want an event to keep moving through the page. To stop it, you can use `event.stopPropagation()`. This prevents the event from going further, either up or down the page.

5. Event Delegation

Instead of adding event listeners to every single child element, you can add one listener to a parent element and let it handle all events for the child elements. This is useful when there are many child elements or when new ones are added dynamically.

6. Event Handling

Event handling is just the process of responding to events. You set up event listeners to detect events, and then write code to say what should happen when the event occurs.

7. Event Queue and Micro Tasks

The **event queue** holds tasks (like user clicks) that need to be handled by the event loop.
Micro tasks, like promises, have higher priority than regular tasks (such as setTimeout). This means they get handled before the next event loop cycle.

8. Set Immediate and Clear Interval

- `setImmediate`: This runs a function right after the current event loop cycle finishes.
- `clearInterval`: This stops a repeating task that was set with `setInterval`.

9. Set Timeout vs. Set Interval

- `setTimeout`: Runs a function once after a delay.
- `setInterval`: Runs a function over and over again at regular time intervals.

10. Clear Timeout

To stop a function that was scheduled to run with `setTimeout`, use `clearTimeout`. This cancels the delayed task.

- **JavaScript Engines and Performance:**

1. JS Engine Working

Imagine the JavaScript (JS) engine as a chef in a kitchen. The engine takes your JavaScript code (like a recipe) and prepares it step by step.

- The chef has two main areas to work with:
  - **Memory Heap**: This is like where ingredients (data) are stored.

- **Call Stack**: This is where the cooking instructions (functions) are followed. When the code runs, the chef reads the instructions one by one, follows them, and removes them once they are done.

2. V8 Engine

Think of the V8 engine as a powerful tool (like a super-fast knife) used in Google Chrome and Node.js.

- The V8 engine turns JavaScript into machine language (like cutting ingredients quickly), making the code run faster.
- It also manages memory to make sure everything runs smoothly and quickly.

3. Execution Context

Picture the chef's workspace where all the ingredients and tools are laid out.

- The **execution context** is like this workspace, holding variables, functions, and the current step in the recipe.
- Every time the chef starts a new task (calling a function), a new workspace is set up.

4. Call Stack

Imagine a stack of plates in the kitchen.

- The **call stack** keeps track of which functions are being used at the moment.
- When a function is done, the chef removes that plate from the stack, just like when the task is completed.

5. Memory Management: Stack and Heap

- The **stack** is like a small kitchen shelf where you keep simple items like numbers and strings, and keep track of tasks (function calls).
- The **heap** is like a large pantry where more complex ingredients (like objects and arrays) are stored.
  The chef (JS engine) makes sure the kitchen stays organized and doesn't get cluttered.

6. Single vs. Multi-Threading

- **Single-threaded** means only one chef is working. It's simple but can be slow if there are too many tasks.
- **Multi-threaded** means multiple chefs are working at the same time, making it faster to handle many tasks at once.

7. Tree Shaking

Think of tree shaking as removing extra ingredients from a recipe that you don't need.

- It gets rid of unnecessary code during the bundling process, making the final file smaller and faster.

8. Starvation

Imagine a customer waiting too long for their meal because the kitchen is too busy with other orders.

- **Starvation** happens in programming when a process or task doesn't get enough CPU time, causing delays.

- **Modules and Imports:**

1. Module

Think of modules as separate recipes for different dishes in a cookbook.

- In JavaScript, **modules** allow you to organize your code into smaller, reusable pieces.
- Each module has its own set of functions, variables, or classes that are focused on one specific task.
- You can **import** (take ingredients from another recipe) or **export** (share your recipe with others) modules.

2. Import and Export

Imagine you're baking cookies and need chocolate chips from another recipe.

- **import** lets you bring in functions, variables, or data from other modules (like grabbing the chocolate chips).
- **export** allows you to share your own functions or data with other modules, like giving away your cookie recipe.

3. Polyfill

A **polyfill** is like a translator at a conference, helping people understand each other.

- It fills in missing features for older browsers so they can use newer JavaScript methods.
- For example, if an older browser doesn't understand a new feature, a polyfill makes it work so your code can run everywhere.

4. ES6 Features

ES6 (also called ECMAScript 2015) brought cool new tools to JavaScript to make coding easier:

- **Arrow functions**: Shorter, cleaner ways to write functions.
- **let** and **const**: Better ways to manage variables.
- **Template literals**: Easier way to work with strings (fancy strings). These features help you write better and more powerful code.

5. Strict Mode in JavaScript

Imagine a strict teacher who makes sure you follow the rules.

- `'use strict';` is like that teacher. It enforces stricter rules in your code.
- It helps you catch mistakes early and encourages better coding practices, like avoiding accidental global variables.

6. Optional Chaining Operator

Think of this as a safe way to check if something exists before using it.

- It's like checking if a door is open before walking into a room.
- Instead of writing `obj && obj.prop`, you can write `obj?.prop`. This won't break if `obj` is missing or `undefined`.

7. Nullish Coalescing Operator

Imagine you're picking pizza toppings.

- The **nullish coalescing operator** (`??`) picks the first available option that's not `null` or `undefined`.
- It's like saying, "I want pepperoni, but if that's not available, give me mushrooms."

- **Web APIs and Asynchronous Operations:**
  - DOM
  - BOM
  - Fetch API
  - Axios vs Fetch
  - Service Worker
  - Local Storage vs Session Storage vs Cookies
  - JSON.stringify vs JSON.parse
  - HTTP Status Codes (Clarification needed)

- **Advanced Topics:**
  - Regular Expressions
  - Normalize
  - Pure Functions
  - Decode and Encode URL in JavaScript
  - App Shell Model
  - PWAs (Progressive Web Apps)
  - Same Origin Policy
  - Zone.js
  - Trickling, Bouncing

String method

charAt()

2. charCodeAt()

3. codePointAt()

4. concat()

5. endsWith()

6. fromCharCode()

7. includes()

8. indexOf()

9. lastIndexOf()

10. localeCompare()

11. match()

12. matchAll()

13. normalize()

14. padEnd()

15. padStart()

16. repeat()

17. replace()

18. replaceAll()

19. search()

20. slice()

21. split()

22. startsWith()

23. substring()

24. toLocaleLowerCase()

25. toLocaleUpperCase()

26. toLowerCase()

27. toUpperCase()

28. toString()

29. trim()

30. trimEnd()

31. trimStart()

32. valueOf()