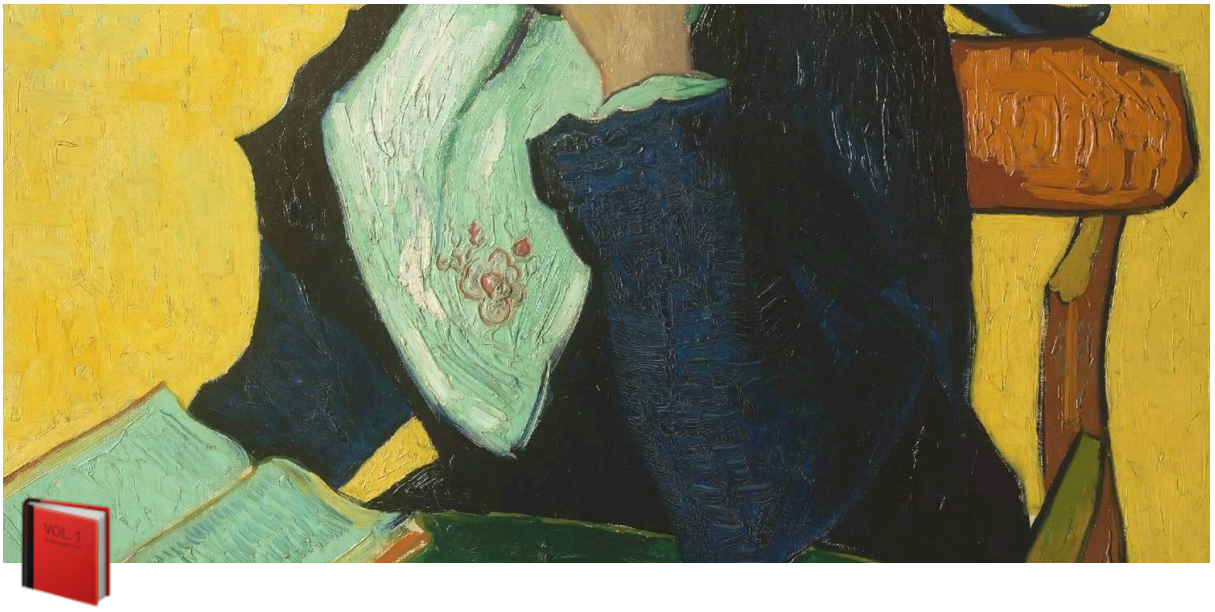

Theory JavaScript

Closure in JavaScript

What is a Closure?

- A closure is a function that retains access to variables from its outer function even after the function has finished execution.

Examples of Closures

Example 1: Basic Closure

```
function x() {
  let a = 7;
  function y() {
    console.log(a);
  }
  return y;
}

var z = x();
// After writing 10000 lines of code
z(); // 7
```

Explanation:

- `y` remembers the variable `a` from `x`.
- Even after `x()` is finished, `z()` still prints `7` because of closure.

Example 2: Closure with Updated Value

```
function x() {
  let a = 7;
  function y() {
    console.log(a);
  }
  a = 100;
  return y;
}

var z = x();
z(); // 100
```

Explanation:

Closures capture **references**, not values. Since `a` was updated to `100`, the closure reflects the latest value.

Example 3: Counter Function (Data Hiding)

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

  return function () {
    count++;
    console.log(count);
  };
}

const counter1 = createCounter();
counter1(); // 1
counter1(); // 2
counter1(); // 3
```

Explanation:

- `count` is private to `createCounter`.
- Only the returned function can access and modify it.
- This demonstrates **data hiding using closure**.

Example 4: Function Factory

```
function multiplyBy(factor) {
  return function (number) {
    return number * factor;
  };
}

const double = multiplyBy(2);
const triple = multiplyBy(3);

console.log(double(5)); // 10
console.log(triple(5)); // 15
```

Explanation:

- `multiplyBy` returns customized functions.
- `double` remembers `factor=2` and `triple` remembers `factor=3`.
- This is called a **function factory**.

Example 5: Loop with var vs let

```
for (var i = 1; i <= 3; i++) {
  setTimeout(function () {
    console.log("var i:", i);
  }, i * 1000);
}

for (let j = 1; j <= 3; j++) {
  setTimeout(function () {
    console.log("let j:", j);
  }, j * 1000);
}
```

Explanation:

- With `var`, the closure captures the same variable, so it prints `4,4,4`.
- With `let`, each iteration has its own scope, so closures print `1,2,3`.

What is JSON?

- JSON stands for **JavaScript Object Notation**.
- It is a **lightweight data format** used to store and exchange data between systems.
- It looks like a JavaScript object, but it is always written as a **string**.

JSON Methods in JavaScript

1. `JSON.stringify()`

Converts a JavaScript object/array into a JSON string.

```
const user = { name: "Sumit", age: 22, isStudent: true };
const jsonString = JSON.stringify(user);

console.log(jsonString);
// Output: '{"name":"Sumit","age":22,"isStudent":true}'
```

Real-world use:

When you send data to a **server** (like filling a form and submitting), you convert it into JSON string first.

2. `JSON.parse()`

Converts a JSON string back into a JavaScript object.

```
const jsonData = '{"name":"Sumit","age":22,"isStudent":true}';
const userObject = JSON.parse(jsonData);

console.log(userObject.name);
// Output: "Sumit"
```

Real-world use:

When your app receives data from an **API**, it usually comes as JSON string. You need to parse it to use it in JS.

What is Currying?

Currying is a technique in JavaScript where a function with **multiple arguments** is transformed into a sequence of **functions with a single argument each**.

Example 1: Normal Function vs Curried Function

Normal Function:

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

console.log(add(2, 3, 4)); // 9
```

Curried Function:

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

console.log(add(2)(3)(4)); // 9
```

Example 2: Using Arrow Functions (Cleaner Syntax)

```
const add = a ⇒ b ⇒ c ⇒ a + b + c;

console.log(add(2)(3)(4)); // 9
```

Why is Currying Useful?

1. **Reusability:**

```
function multiply(a) {
  return function (b) {
    return a * b;
  };
}

const double = multiply(2);
console.log(double(5)); // 10

const triple = multiply(3); // fixes a = 3
console.log(triple(5)); // 15

const quadruple = multiply(4); // fixes a = 4
console.log(quadruple(5)); // 20
```

   👉 We reused `multiply` to create a `double` function.

2. **Customization:**

   Helps in creating specialized functions by partially applying arguments.

3. **Functional Programming Style:**

   Used heavily in frameworks like **React, Redux, Lodash**.

What is Hoisting?

In JavaScript, hoisting is the process where variable and function declarations are moved to the top of their scope before execution. This means I can use functions before I define them, and variables declared with `var` will be accessible but initialized as `undefined`. However, `let` and `const` behave differently due to the Temporal Dead Zone.

🔷 Example 1: Hoisting with `var`

```
console.log(a); // undefined
var a = 10;
console.log(a); // 10
```

Explanation:

- JS engine sees `var a;` at the top (hoisted).
- At runtime:

```
var a;        // declaration hoisted
console.log(a); // undefined
a = 10;       // initialization
console.log(a); // 10
```

🔷 Example 2: Hoisting with `let` and `const`

```
console.log(b); // ❌ ReferenceError
let b = 20;
```

Explanation:

- `let` and `const` are also hoisted, but they remain in **Temporal Dead Zone** until initialized.
- That's why you get **ReferenceError** if accessed before initialization.

🔷 Example 3: Hoisting with Functions

```
sayHello(); // ✅ Works fine
function sayHello() {
  console.log("Hello!");
}
```

Explanation:

- Function **declarations** are fully hoisted with their definition.
- So you can call `sayHello()` before its definition.

🔷 Example 4: Function Expression vs Declaration

```
greet(); // ❌ TypeError: greet is not a function
var greet = function () {
```

```
  console.log("Hi!");
};
```

Explanation:

- Here `greet` is a **variable** (not a function declaration).
- `var greet` is hoisted → initialized as `undefined`.
- At runtime:

```
var greet; // undefined
greet();   // ❌ greet is undefined, not a function
greet = function() { ... };
```

Scope, Scope Chaining & Lexical Environment

🔷 1. What is Scope in JavaScript?

Scope means **where a variable or function is accessible** in your code.

Types of Scope in JavaScript:

1. **Global Scope** → Variables accessible everywhere.
2. **Local/Function Scope** → Variables accessible only inside a function.
3. **Block Scope (ES6 - `let`, `const`)** → Variables accessible only inside `{}`.

Example of Scope:

```
let globalVar = "I am Global";  // Global Scope

function testFunction() {
    let localVar = "I am Local";  // Function Scope
    console.log(globalVar);  // ✅ Accessible
    console.log(localVar);   // ✅ Accessible
}

console.log(globalVar);  // ✅ Accessible
console.log(localVar);   // ❌ Error (localVar is inside function)
testFunction();
```

🔷 2. What is Scope Chaining?

Scope Chaining means that **JavaScript searches for a variable in the current scope, if not found, it moves up to the parent scope, and so on until it reaches the global scope.**

Example of Scope Chaining:

```
let globalVar = "I am Global";

function outerFunction() {
    let outerVar = "I am Outer";
    function innerFunction() {
        let innerVar = "I am Inner";
        console.log(innerVar);  // ✅ Found in innerFunction
        console.log(outerVar);  // ✅ Found in outerFunction
        console.log(globalVar); // ✅ Found in Global Scope
    }
    innerFunction();
}

outerFunction();
```

🔷 How JavaScript searches for variables?

1. **innerFunction** → First searches in its own scope.
2. **If not found** → Moves up to **outerFunction**.
3. **If still not found** → Moves to **global scope**.
4. **If not found anywhere** → JavaScript throws an **error**.

🔷 3. What is Lexical Environment?

Lexical Environment is a **combination of the current scope and the reference to its outer scope (parent scope).**

Every time a function is created, a new **Lexical Environment** is created.

Example of Lexical Environment:

```
function outer() {
    let a = 10;
```

```
    function inner() {
        let b = 20;
        console.log(a);  // ✅ a is accessible due to Lexical Environment
    }

    inner();
}
outer();
```

✅ **Lexical Environment of `inner()` includes:**

1. **Its own variables** (`b = 20`).
2. **Reference to `outer()` variables** (`a = 10`).
3. **Reference to global scope.**

💬 Q: What is the Temporal Dead Zone in JavaScript?

✅ *"Temporal Dead Zone (TDZ) is the period between when a variable is declared using `let` or `const` and when it is initialized. Accessing the variable in this zone results in a `ReferenceError`."*

💬 Q: Why does `let` and `const` have a TDZ but `var` does not?

✅ *"`let` and `const` are hoisted but not initialized, so accessing them before assignment causes an error. `var` is hoisted and initialized with `undefined`, so it doesn't have a TDZ."*

Higher-Order Function

What is a HOF?

A Higher-Order Function is a function that either takes another function as an argument, or returns a function, or does both.

Why?

- In JavaScript, **functions are first-class citizens**.

  This means we can treat functions like values , (store them in variables, pass them around, return them).

- Because of this, we can build **higher-order functions** that allow powerful abstraction and reusability.

Example 1 — Taking a Function as an Argument

```
function higherOrder(fn) {
  return fn(5);
}

const square = (x) => x * x;

console.log(higherOrder(square)); // 25
```

✅ Here:

- `higherOrder` takes `square` (a function) as an argument.
- This is **higher-order** because it works with another function.

Example 2 — Returning a Function

```
function multiply(a) {
  return function(b) {
    return a * b;
  };
}

const double = multiply(2);
console.log(double(5)); // 10
```

✅ Here:

- `multiply` returns another function.
- The returned function **remembers** `a` because of closure.

Example 3 — Both

```
setTimeout(() => console.log("Hello"), 1000);
```

- `setTimeout` is a built-in **higher-order function**.

- It **takes a function as an argument** (callback).

Callback Function

1. What is a Callback Function?

A **callback function** is a function that is **passed as an argument** to another function and is **executed later** (either synchronously or asynchronously).

2. Why Callbacks? (Purpose)

- Reusability: Pass different behavior into the same function.
- Control flow: Run code *after* something else finishes.
- Asynchronous operations: Waiting for a task (like API call, file read, setTimeout).

Example 1 — Synchronous Callback

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

function sayBye() {
  console.log("Goodbye!");
}

greet("Sumit", sayBye);

// Output:
// Hello Sumit
// Goodbye!
```

✅ Here:

- `sayBye` is passed into `greet`.
- `greet` calls it after greeting.
- This is a **synchronous callback** (executed immediately after).

Example 2 — Asynchronous Callback

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

setTimeout(cb() ⇒ {
  console.log("This runs after 2 seconds");
}, 2000);

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

✅ Here:

- The arrow function `() ⇒ console.log(...)` is a **callback**.
- It is executed later by `setTimeout` after 2 seconds.
- Demonstrates **asynchronous callback**.

Cross-Question Prep

Q1: Difference between a callback and a higher-order function?

- Callback → the function you *pass in*.
- Higher-order function → the function that *receives or returns* another function.
- Solved by **Promises** and **async/await**.

Q3: Can callbacks be synchronous and asynchronous?

- Yes. Synchronous: executed immediately (like `Array.map(callback)`).
- Asynchronous: executed later (like `setTimeout`, `fs.readFile`).

What is diff between Sync & Async ?

JavaScript runs in a single thread, so it can execute one task at a time. Because of this, we have two types of operations: **Synchronous** and **Asynchronous**.

- **Synchronous**: Tasks are executed **one after another**. A new task cannot start until the current one is finished.
- **Asynchronous**: Tasks **don't block the execution**. While one task is waiting (like fetching data from server), JavaScript can continue running other code.

Interview-Friendly Example

Sync example

```
console.log("Task 1");
console.log("Task 2");
console.log("Task 3");
```

Async example

```
console.log("Task 1");
setTimeout(() => {
  console.log("Task 2");
}, 2000);
console.log("Task 3");
```

So the key difference is: **Synchronous blocks the code execution until the task is finished**, while **Asynchronous allows other tasks to run without waiting**. This is important in JavaScript because it prevents our UI or server from freezing when tasks take time, like API calls or file reading.

Real-Life Analogy Example

🍽️ Restaurant Analogy

- **Synchronous (Sync)**

  Imagine you go to a restaurant, and you order food.

  The waiter takes your order, goes to the kitchen, waits there until your food is ready, and then comes back to you.

  👉 During this waiting time, the waiter **cannot serve other customers**.

- **Asynchronous (Async)**

  Now imagine another waiter. He takes your order, gives it to the kitchen, and while the food is being cooked, he goes to serve other customers.

  When your food is ready, the kitchen staff calls the waiter, and he brings it to you.

  👉 This way, the waiter **doesn't waste time waiting**.

How to Say in Interview

“So, Sync is like a waiter who serves one customer at a time and waits until finished, while Async is like a smart waiter who multitasks – he doesn’t wait, he continues serving others, and comes back when the first customer’s order is ready.”

What are the ways to make the code Async?

In JavaScript, we can make code asynchronous in different ways. The main tools are **callbacks, promises, async/await, and some built-in APIs like setTimeout or fetch**.

Detailed Explanation (with examples)

1. **Callbacks (Old way)**

   We pass a function as an argument to be executed later.

```
setTimeout(() => {
  console.log("This runs after 2 seconds");
}, 2000);
```

👉 Problem: Too many callbacks lead to **callback hell**.

1. **Promises (Modern way)**

   A promise represents a value that may be available now, later, or never.

```
fetch("https://api.example.com/data")
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.log(error));
```

👉 Makes async code more readable than nested callbacks.

1. **Async/Await (Latest & Cleaner)**

   It’s syntactic sugar over promises.

```
async function getData() {
  try {
    const response = await fetch("https://api.example.com/data");
    const data = await response.json();
```

```
    console.log(data);
  } catch (error) {
    console.log(error);
  }
}
getData();
```

👉 Looks like synchronous code, but it's async under the hood.

1. **Event Listeners / Web APIs**

   For example:

```
document.getElementById("btn").addEventListener("click", () => {
  console.log("Button clicked!");
});
```

👉 The callback runs asynchronously when the event happens.

Wrap-Up (One-Liner)

"So in short, we can make code asynchronous using **callbacks, promises, async/await, and event-based or timer-based APIs** like `setTimeout`, `setInterval`, `fetch`, etc."

⚡Pro tip for interview: If asked "***which one should we use today?"***, you can say:

"**Async/Await with Promises is preferred** in modern JavaScript because it makes the code cleaner, easier to read, and avoids callback hell."

What are the Web Browser APIs?

Web Browser APIs are **extra powers provided by the browser** that allow JavaScript to do things like update UI (DOM API), fetch data (Fetch API), store data (Storage API), get location (Geolocation API), and more. They are not part of core JavaScript but are built into browsers.

Main Explanation with Examples

1. **DOM API (Document Object Model)**

   👉 Lets us change or access HTML & CSS.

```
document.getElementById("title").innerText = "Hello World!";
```

1. **Fetch API**

   👉 Used for network requests (AJAX).

```
fetch("https://api.example.com/data")
  .then(res ⇒ res.json())
  .then(data ⇒ console.log(data));
```

1. **Storage APIs**

   👉 Save data in the browser.

```
localStorage.setItem("theme", "dark");
console.log(localStorage.getItem("theme"));
```

1. **Geolocation API**

   👉 Get user’s location.

```
navigator.geolocation.getCurrentPosition(position ⇒ {
  console.log(position.coords.latitude, position.coords.longitude);
});
```

1. **Canvas API**

   👉 For 2D drawing & games.

```
const canvas = document.getElementById("myCanvas");
const ctx = canvas.getContext("2d");
ctx.fillRect(20, 20, 150, 100);
```

1. **Timers API**

   👉 Run code after a delay or repeatedly.

```
setTimeout(() ⇒ console.log("Runs after 2 sec"), 2000);
```

The Event Loop

- What is the Event Loop in JavaScript?

  → The Event Loop is a mechanism in JavaScript that manages the execution of asynchronous code. It ensures that tasks are executed in the correct order without blocking the main thread.

- How does the Event Loop work?

  → JavaScript has a Call Stack for executing synchronous code, Web APIs for handling asynchronous operations, and two queues (Microtask Queue & Callback Queue) for scheduling tasks. The Event Loop constantly checks the Call Stack, and when it's empty, it pushes tasks from these queues to execute them.

- What executes first: `setTimeout` or `Promise.then`?

  → `Promise.then` executes first because it goes to the Microtask Queue, which has higher priority than the Callback Queue where `setTimeout` callbacks are placed.

Promise Concept

What is Callback Hell?

Callback Hell happens when we use too many nested callbacks, making code hard to read, debug, and maintain.

Explanation in Simple Words

- A **callback** is just a function passed to another function to be executed later.
- But when we chain multiple async tasks using callbacks (one inside another), the code starts looking like a **pyramid of doom** → lots of indentation and messy structure.
- This is called **Callback Hell**.

Example (Show interviewer)

```
// Example of callback hell
getUser(1, function(user) {
```

```
  getOrders(user.id, function(orders) {
    getOrderDetails(orders[0], function(details) {
      processPayment(details, function(result) {
        console.log("Payment done:", result);
      });
    });
  });
});
```

👉 See how the code keeps going **right side → → →**.

Inversion of Control

Inversion of Control in callbacks means we give control of our code execution to someone else (the callback receiver), and we are no longer fully in charge of when or how our function will be called.

Explanation in Simple Words

- When we pass a **callback function** to another function/library, we trust that code to call it at the right time.
- But we **lose control** over it:
  - It might call our callback too early, too late, or multiple times.
  - It might forget to call it.
- This **loss of control** is called **Inversion of Control (IoC)**.

Example (Interview-Friendly)

```
function getData(callback) {
  // Simulating async code
  setTimeout(() => {
    // Sometimes it might call twice (bad control)
    callback("Data 1");
    callback("Data 2");
  }, 1000);
}

getData((data) => {
```

```
  console.log("Received:", data);
});
```

👉 Here, our callback is **dependent** on `getData` function.

If `getData` misbehaves (calls twice), we have no control.

✅ Real-World Analogy

Imagine you give your phone number to a delivery person and say: *"Call me when you arrive."*

- If he calls at the wrong time, multiple times, or doesn't call at all — **you have no control**.

  👉 That's Inversion of Control — you handed over control to someone else.

⚡ Pro Tip: If interviewer asks ***"How do Promises solve this?"*** → Say:

"With Promises, instead of someone else controlling when my callback runs, **I control it** by attaching `.then()` or using `await`. The function either resolves or rejects once, so no multiple/unexpected calls."

What is Promise?

A Promise in JavaScript is an object that represents the result of an asynchronous operation. It's a placeholder for a value that will be available **now, later, or never**.

Main Explanation

- A Promise has **3 states**:
  1. **Pending** → Initial state (waiting for result).
  2. **Fulfilled (Resolved)** → Operation completed successfully.
  3. **Rejected** → Operation failed with an error.
- Promises help us avoid **callback hell** and give more control over async tasks.

Example (Interview-Friendly)

```
const promise = new Promise((resolve, reject) => {
  let success = true;
```

```
  if (success) {
    resolve("Data fetched successfully!");
  } else {
    reject("Something went wrong!");
  }
});

promise
  .then(result => console.log(result)) // Runs if resolved
  .catch(error => console.log(error))  // Runs if rejected
  .finally(() => console.log("Done")); //Always executed
```

👉 Output (if success is true):

```
Data fetched successfully!
Done
```

Real-World Analogy

Think of a **pizza order** 🍕 again:

- You order pizza → Promise is **pending**.
- If pizza arrives → Promise is **fulfilled** (resolve).
- If delivery fails → Promise is **rejected**.
- Either way, at the end → Promise is **settled**.

👉 `.then()` is like *"What to do when pizza comes"*,

`.catch()` is like *"What to do if pizza fails"*,

`.finally()` is like *"Cleaning up after, no matter what."*

⚡ Pro Tip: If interviewer asks a **follow-up** like *"Why do we need promises if we already have callbacks?"* → Say:

"Callbacks suffer from inversion of control and callback hell. Promises solve these issues by providing a cleaner, more predictable way to handle async code."

What are different states of a `Promise` - `pending`, `fulfilled`, `rejected`

A Promise in JavaScript goes through different states during its lifecycle. These states are **pending, fulfilled, and rejected**.

Main Explanation (Simple Words)

1. **Pending**

- Initial state when the promise is created.
- The result is not yet available.

```
const promise = new Promise(() => {}); // Always pending
```

1. **Fulfilled (Resolved)**

- The async operation finished successfully.
- The promise gives a **value**.

```
Promise.resolve("Success")
  .then(result => console.log(result)); // Output: Success
```

1. **Rejected**

- The async operation failed.
- The promise gives an **error reason**.

```
Promise.reject("Error occurred")
  .catch(err => console.log(err)); // Output: Error occurred
```

👉 After a Promise is **fulfilled or rejected**, it is called **settled** (it won't change state again).

How to consume an existing `promise` ?

To consume an existing promise, we handle its result or error using **.then(), .catch(), .finally()** or by using **async/await**.

Main Explanation with Examples

1. **Using `.then()` and `.catch()`**

```
let promise = fetch("https://api.example.com/data");

promise
  .then(response => response.json())   // Runs if fulfilled
  .then(data => console.log(data))     // Chained success
  .catch(error => console.log(error)); // Runs if rejected
```

👉 `.then()` → handles success,

👉 `.catch()` → handles failure.

1. **Using** `.finally()`

```
promise
  .finally(() => console.log("Operation completed"));
```

👉 Runs regardless of success or failure → good for cleanup (like hiding a loading spinner).

1. **Using** `async/await` **(Modern way)**

```
async function consumePromise() {
  try {
    const response = await fetch("https://api.example.com/data");
    const data = await response.json();
    console.log(data); // Success
  } catch (error) {
    console.log(error); // Failure
  } finally {
    console.log("Operation completed");
  }
}
consumePromise();
```

👉 Cleaner, looks like synchronous code, but still async.

Wrap-Up (One-Liner)

"So, to consume an existing promise, we use `.then()`, `.catch()`, `.finally()`, or `async/await` depending on the style we prefer. In modern JavaScript, `async/await` is

most common because it's cleaner and more readable."

⚡ Pro Tip: If interviewer asks *"Which is better to consume a promise?"* → Answer:

"Both work, but **async/await** is preferred in modern JavaScript because it avoids chaining and makes async code look synchronous."

How to chain promises using .then

In JavaScript, we can chain promises using `.then()` so that the output of one async operation becomes the input of the next. This avoids deeply nested callbacks and makes code cleaner.

Main Explanation

- Each `.then()` returns a **new promise**.
- Whatever value you `return` inside `.then()` is passed to the next `.then()`.
- If any `.then()` fails, the error jumps to the nearest `.catch()`.

```
const resultant = new Promise((resolve, reject) => {
    setTimeout(() => resolve(10), 1000);
})
.then(num => {
    console.log(num);  // 10
    return num * 2;
})
.then(num => {
    console.log(num);  // 20
    return num * 3;
})
.then(num => {
    console.log(num);  // 60
});
```

⚡ Pro Tip: If interviewer asks ***"How is this better than callbacks?"*** → Say:

"With callbacks, we get nested code (callback hell). With `.then()` chaining, we get a flat structure that's easier to read and maintain."

Simple Promise Chaining Example

```
// Step 1: A function that returns a promise
function step1() {
  return new Promise((resolve) => {
    setTimeout(() => resolve("Step 1 done"), 1000);
  });
}

function step2() {
  return new Promise((resolve) => {
    setTimeout(() => resolve("Step 2 done"), 1000);
  });
}

function step3() {
  return new Promise((resolve) => {
    setTimeout(() => resolve("Step 3 done"), 1000);
  });
}

// Chaining promises
step1()
  .then((result1) => {
    console.log(result1); // Step 1 done
    return step2();
  })
  .then((result2) => {
    console.log(result2); // Step 2 done
    return step3();
  })
  .then((result3) => {
    console.log(result3); // Step 3 done
    console.log("All steps completed!");
  })
  .catch((error) => {
    console.log("Error:", error);
  });
```

How to handle errors in a promise chain using `.catch`

In a promise chain, we use `.catch()` to handle errors. If any promise in the chain is rejected, control jumps directly to the nearest `.catch()`.

Example

```
function step1() {
  return Promise.resolve("Step 1 done");
}

function step2() {
  return Promise.reject("Something went wrong in Step 2");
}

function step3() {
  return Promise.resolve("Step 3 done");
}

step1()
  .then((res) => {
    console.log(res); // Step 1 done
    return step2();
  })
  .then((res) => {
    console.log(res); // Will be skipped because step2 failed
    return step3();
  })
  .then((res) => {
    console.log(res); // Will also be skipped
  })
  .catch((error) => {
    console.log("Caught error:", error);
  })
  .finally(() => {
    console.log("Chain finished (success or failure)");
  });
```

What happens when an Error gets thrown inside `.then` when there is a `.catch`

When an error is thrown inside a `.then()` block, it automatically turns the promise into a **rejected state**. If there is a `.catch()` in the chain, that `.catch()` will handle the error.

What happens when an Error gets thrown inside `.then` when there is no `.catch`

If an error is thrown inside `.then()` and there is **no** `.catch()` in the chain, the promise becomes rejected and results in an **unhandled promise rejection**.

In browsers, you'll see it in the console as an error.

In Node.js, it may even **terminate the process** (depending on version & settings).

Example

```
Promise.resolve("Start")
  .then((data) => {
    console.log("✅ First then:", data);
    throw new Error("Something went wrong inside .then");
  })
  .then((data) => {
    console.log("This will never run:", data);
  });

// ❌ No .catch() here
```

🔷 Output in Browser Console

```
✅ First then: Start
Uncaught (in promise) Error: Something went wrong inside .then
```

🔷 Output in Node.js

```
(node:1234) UnhandledPromiseRejectionWarning: Error: Something went w
rong inside .then
```

👉 In newer Node.js versions, this can **crash the program**.

Why must `.catch` be placed towards the end of the promise chain?-

.catch should be placed towards the end of the promise chain because it ensures that **any error thrown anywhere in the chain** will be caught.

If you place `.catch` in the middle, it will only catch errors **before that point**, and the rest of the chain might still break if new errors occur.

How to consume multiple promises by chaining?

To consume multiple promises by chaining, we return a new promise inside each `.then()`. The next `.then()` waits for the previous one to finish before running. This way, we can execute multiple async tasks in sequence

Example – Chaining Multiple Promises

```
function taskOne() {
  return Promise.resolve("✅ Task One done");
}

function taskTwo() {
  return Promise.resolve("✅ Task Two done");
}

function taskThree() {
  return Promise.resolve("✅ Task Three done");
}

taskOne()
  .then((result1) => {
    console.log(result1);
    return taskTwo(); // chain next promise
  })
  .then((result2) => {
    console.log(result2);
    return taskThree(); // chain next promise
  })
  .then((result3) => {
```

```
  console.log(result3);
})
.catch((err) => {
  console.log("❌ Error:", err);
});
```

Explain Promise.all()

How to Start Your Answer

“ `Promise.all()` is used when we want to run multiple promises **in parallel** and wait until **all of them are resolved**. It returns a single promise that resolves into an array of results.”

Key Points

1. Accepts an **array of promises**.
2. Runs them **concurrently** (not one by one like chaining).
3. Resolves when **all succeed** → gives array of results.
4. Rejects immediately if **any one fails** → returns the first error.

Example

```
const p1 = Promise.resolve("🍎 Apple ready");
const p2 = new Promise((resolve) => setTimeout(() => resolve("🍌 Banana r
eady"), 1000));
const p3 = new Promise((resolve) => setTimeout(() => resolve("🍇 Grapes r
eady"), 2000));

Promise.all([p1, p2, p3])
  .then((results) => {
    console.log("✅ All fruits ready:", results);
  })
  .catch((err) => {
    console.log("❌ Error:", err);
  });
```

🔷 Output

```
✅ All fruits ready: [ '🍎 Apple ready', '🍌 Banana ready', '🍇 Grapes ready'
]
```

👉 Notice: It **waited 2 seconds** (because of grapes, the slowest one) before resolving.

✅ Error Case

```
const p1 = Promise.resolve("Task 1");
const p2 = Promise.reject("Task 2 failed ❌");
const p3 = Promise.resolve("Task 3");

Promise.all([p1, p2, p3])
  .then((results) => {
    console.log("✅ Success:", results);
  })
  .catch((err) => {
    console.log("❌ First error:", err);
  });
```

🔷 Output

```
❌ First error: Task 2 failed ❌
```

👉 Even though Task 1 & 3 succeeded, the **whole Promise.all failed** because of Task 2.

How to consume multiple promises by Promise.all?

To consume multiple promises using `Promise.all`, we pass them in an array. It runs them **in parallel** and gives us one final promise that resolves with all results when everything succeeds, or rejects immediately if any one fails.

Example – All Success

```
function taskOne() {
  return new Promise((resolve) => setTimeout(() => resolve("✅ Task One do
```

```
ne"), 1000));
}

function taskTwo() {
  return new Promise((resolve) => setTimeout(() => resolve("✅ Task Two do
ne"), 2000));
}

function taskThree() {
  return new Promise((resolve) => setTimeout(() => resolve("✅ Task Three d
one"), 1500));
}

Promise.all([taskOne(), taskTwo(), taskThree()])
  .then((results) => {
    console.log("All tasks done:", results);
  })
  .catch((err) => {
    console.log("❌ Error:", err);
  });
```

🔷 Output

```
All tasks done: [ '✅ Task One done', '✅ Task Two done', '✅ Task Three d
one' ]
```

👉 It waited **2 seconds (the slowest task)** before resolving.

Example – With One Failure

```
const p1 = Promise.resolve("Task 1 complete");
const p2 = Promise.reject("Task 2 failed ❌");
const p3 = Promise.resolve("Task 3 complete");

Promise.all([p1, p2, p3])
  .then((results) => {
    console.log("✅ Results:", results);
```

```
})
.catch((err) => {
  console.log("❌ Error caught:", err);
});
```

🔷 Output

```
❌ Error caught: Task 2 failed ❌
```

👉 Even though Task 1 & 3 worked, the **whole** `Promise.all` **failed** because of Task 2.

Why is error handling the most important part of using a promise?

Error handling is the most important part of promises because promises are used to handle **asynchronous tasks** where failures are common — like network requests, file reads, or database operations.

If we don't handle errors properly, they can become **unhandled rejections**, which may crash the program or leave it in an inconsistent state.

Key Points You Can Mention

1. **Async tasks are unreliable** →

   APIs can fail, servers may be down, files may not exist.

2. **Without error handling** →

   Promise rejections become **unhandled rejections**, leading to crashes or bugs.

3. **Promises unify error handling** →

   Just like `try...catch` for sync code, promises use `.catch()` (or `try...catch` with async/await) for async code.

4. **Production safety** →

   In real-world apps, proper error handling ensures the app doesn't crash and users see friendly error messages.

Example

```
fetch("https://wrong-api.com/data") // invalid URL
  .then((res) ⇒ res.json())
  .then((data) ⇒ console.log("Data:", data))
  // Without this catch → app will throw "UnhandledPromiseRejection"
  .catch((err) ⇒ {
    console.log("❌ Error handled gracefully:", err.message);
  });
```

👉 Instead of app crashing, user gets a controlled message.

How to promisify an asynchronous callbacks based function - eg. setTimeout, fs.readFile

Promisifying means converting a function that uses callbacks (like `setTimeout` or `fs.readFile`) into a function that returns a **promise** instead.

This makes it easier to use with `.then()`, `.catch()`, and `async/await`.

Example 1 – Promisify `setTimeout`

```
function delay(ms) {
  return new Promise((resolve) ⇒ {
    setTimeout(() ⇒ {
      resolve(`⏱️ Finished after ${ms}ms`);
    }, ms);
  });
}

// Usage
delay(2000).then((msg) ⇒ console.log(msg));
```

👉 Output (after 2s):

```
⏱️ Finished after 2000ms
```

Example 2 – Promisify `fs.readFile` (Node.js)

```
const fs = require("fs");
```

```
function readFilePromise(filePath) {
  return new Promise((resolve, reject) => {
    fs.readFile(filePath, "utf8", (err, data) => {
      if (err) reject(err);   // reject if error
      else resolve(data);     // resolve with file contents
    });
  });
}

// Usage
readFilePromise("sample.txt")
  .then((data) => console.log("📄 File content:", data))
  .catch((err) => console.error("❌ Error:", err));
```

🔷 Promise-based Utility Functions in JavaScript

1. `Promise.resolve()`

👉 It returns a promise that is **already resolved** with the given value.

- Useful when you want to wrap a normal value into a promise.

Syntax:

```
Promise.resolve(value)
```

Example:

```
Promise.resolve(42).then((res) => console.log(res));
// Output: 42
```

Here, `42` is wrapped inside a resolved promise.c+

2. `Promise.reject()`

👉 It returns a promise that is **already rejected** with the given reason (error).

- Useful to simulate an error quickly.

Syntax:

```
Promise.reject(reason)
```

Example:

```
Promise.reject("Something went wrong")
  .catch((err) ⇒ console.log(err));
// Output: Something went wrong
```

3. `Promise.all()`

👉 Runs multiple promises **in parallel** and returns a single promise.

- **Resolves** when **all promises resolve**.
- **Rejects immediately** if **any promise rejects**.

Example:

```
const p1 = Promise.resolve(10);
const p2 = Promise.resolve(20);
const p3 = Promise.resolve(30);

Promise.all([p1, p2, p3])
  .then((values) ⇒ console.log(values))
  .catch((err) ⇒ console.log(err));

// Output: [10, 20, 30]
```

⚠️ If even one promise fails, the whole chain rejects.

4. `Promise.allSettled()`

👉 Runs multiple promises in parallel, but **waits for all of them to finish**, whether **resolved or rejected**.

- Returns an array of objects with `{status, value}` or `{status, reason}`.

Example:

```
const p1 = Promise.resolve("Success ✅");
const p2 = Promise.reject("Error ❌");

Promise.allSettled([p1, p2])
  .then((results) => console.log(results));
```

✅ Output:

```
[
  { status: "fulfilled", value: "Success ✅" },
  { status: "rejected", reason: "Error ❌" }
]
```

5. Promise.any()

👉 Returns the **first fulfilled (resolved)** promise.

- If **all promises reject**, it throws an **AggregateError**.

Example:

```
const p1 = Promise.reject("Fail 1");
const p2 = Promise.resolve("First Success");
const p3 = Promise.resolve("Second Success");

Promise.any([p1, p2, p3])
  .then((res) => console.log(res))
  .catch((err) => console.error(err));

// Output: First Success
```

6. Promise.race()

👉 Returns the **first promise** that settles (resolved OR rejected).

- Doesn't wait for others.

Example:

```
const p1 = new Promise((resolve) => setTimeout(resolve, 100, "Slow Succes
s"));
const p2 = new Promise((resolve) => setTimeout(resolve, 50, "Fast Succes
s"));

Promise.race([p1, p2])
  .then((res) => console.log(res))
  .catch((err) => console.error(err));

// Output: Fast Success
```

Debouncing and Throttling

What is Debouncing?

- **Debouncing is a technique to control how often a function is executed.**
- Imagine you keep pressing a button many times very quickly → without debouncing, the function runs for every press (too many calls!).
- With **debouncing**, the function will only run **after you stop pressing for some time**.

Real-life Example

Think of an **elevator**:

- You press the button many times, but the elevator will **wait for a few seconds** before closing the door.
- If someone presses again during that wait, the timer resets.
- The door only closes when nobody presses for a while.

  That's debouncing! ✅

💻 Code Example (Search Box)

Without debouncing:

```
function searchApiCall(query) {
  console.log("Fetching results for:", query);
```

```
}

// Every keypress calls the API (too many calls!)
document.getElementById("search").addEventListener("input", (e) => {
  searchApiCall(e.target.value);
});
```

With debouncing:

```
function debounce(func, delay) {
  let timer;
  return function(...args) {
    clearTimeout(timer); // reset old timer
    timer = setTimeout(() => {
      func.apply(this, args);
    }, delay);
  };
}

function searchApiCall(query) {
  console.log("Fetching results for:", query);
}

// Only call API after user stops typing for 500ms
const debouncedSearch = debounce(searchApiCall, 500);

document.getElementById("search").addEventListener("input", (e) => {
  debouncedSearch(e.target.value);
});
```

What is Throttling?

- **Throttling is a technique to control the rate of function execution.**
- Unlike debouncing (which waits until activity stops), throttling makes sure the function runs at **regular intervals** no matter how often it's called.
- In short:
  - **Debounce → wait until user stops doing something.**

- **Throttle → execute function at fixed time intervals.**

Real-life Example

Think of a **security guard at a club**:

- People keep coming to enter the club very fast.
- The guard lets people in only **once every 5 seconds**, no matter how many people are waiting.
- That's throttling

💻 Code Example (Scroll Event)

Without throttling:

```
window.addEventListener("scroll", () => {
  console.log("Scroll position:", window.scrollY);
});
```

👉 Problem: Every tiny scroll fires an event → hundreds of times per second!

With throttling:

```
function throttle(func, limit) {
  let inThrottle;
  return function(...args) {
    if (!inThrottle) {
      func.apply(this, args);
      inThrottle = true;
      setTimeout(() => {
        inThrottle = false;
      }, limit);
    }
  };
}

function handleScroll() {
  console.log("Scroll position:", window.scrollY);
}
```

```
// Run only once every 1000ms
window.addEventListener("scroll", throttle(handleScroll, 1000));
```

What is AJAX ?

AJAX stands for Asynchronous JavaScript and XML. It is a technique used in web development to send and receive data asynchronously from a web server without having to reload the entire webpage. AJAX allows for updating parts of a webpage without refreshing the entire page. It is commonly used to fetch data from a server, submit form data, and update dynamic content on a web page.

How is it different from using React-Js?

AJAX vs ReactJS

- **AJAX** is just a technique to fetch data from the server without refreshing the page, but you need to update the DOM manually.
- **ReactJS** is a UI library that builds reusable components and updates the DOM automatically using **state** and **virtual DOM**.
- In fact, React still uses AJAX (fetch/axios) under the hood, but it simplifies **UI rendering and state management** compared to plain AJAX.

AJAX is mainly about asynchronous data fetching, while ReactJS is about building and managing UI efficiently. React still uses AJAX internally, but it handles UI updates automatically with state and virtual DOM, making apps more scalable.

What is Diffing ?

Diffing, also known as "DOM diffing" or "virtual DOM diffing," is a technique used by libraries like React.js to efficiently update the user interface. It involves comparing the previous state of the UI to the current state and identifying the minimum number of changes needed to update the UI.

Here's how diffing works:

- A virtual representation of the previous UI state is created, called the "virtual DOM."
- When the UI changes, a new virtual DOM is created to represent the new UI state.

- The two virtual DOMs are compared to identify the differences between them.
- Only the parts of the UI that have changed are updated in the actual DOM, rather than updating the entire UI.

By using diffing, React.js can perform UI updates more efficiently because it only updates the parts of the UI that have actually changed, minimising the amount of work needed to update the DOM. This can result in faster updates and improved performance in large-scale web applications.

What is Node.js, its features, and how it works ?

Node.js is an **open-source, cross-platform runtime environment** that lets you run **JavaScript outside the browser** (on the server). So, with Node.js, JavaScript is not just for frontend but also for **backend development**.

2. Features of Node.js

- **Asynchronous & Event-driven** → It uses a **non-blocking I/O model**, so tasks like reading files, databases, or APIs don't block other code.
- **Very fast** → It runs on **Google's V8 engine**, which makes execution of JavaScript really fast.
- **Single-threaded but scalable** → It uses **event loop + callbacks** to handle many requests at once without creating multiple threads.
- **Huge ecosystem** → Node Package Manager (**NPM**) provides thousands of ready-to-use modules.

3. How it works (Simple Explanation)

- Node.js works on a **single thread** using an **event loop**.
- When a request comes (like read file / DB call), Node.js **delegates it** to system resources.
- Meanwhile, it keeps listening for new requests.
- Once the task is done, the **callback function** is executed.

  👉 This is why Node.js can handle **thousands of requests at the same time** without slowing down.

✅ **Interview-ready short version:**

Node.js is a runtime that allows JavaScript to run on the server. It's asynchronous, event-driven, fast (because of V8), and single-threaded but scalable using an event loop. It's great for building real-time, scalable applications like chat apps or streaming services.

What is JavaScript?

JavaScript is a high-level, interpreting or client-side scripting language that allow to add logic in web pages and make web pages more interactive mode.

It's runs inside the browser and work with HTML & CSS to build responsive and dynamic web application.

What is TypeScript?

💬 Short Interview Answer:

> TypeScript is a superset of JavaScript that introduces static typing. It helps developers catch errors early during development and makes large projects more maintainable. It compiles into plain JavaScript so it works in any browser or JavaScript environment.

👴 If interviewer asks why TypeScript is better than JavaScript:

> In JavaScript, types are dynamic, so many bugs appear only at runtime. TypeScript checks types during compile-time, reducing errors and improving code reliability. It's especially useful in big projects or teams.

Explain LifeCycle of Class/Function

👴 Short Interview Answer (2–Minute Version):

> "React class components go through three main lifecycle phases — Mounting, Updating, and Unmounting.
>
> In the **Mounting phase**, methods like `constructor`, `render`, and `componentDidMount` run.
>
> In the **Updating phase**, the component re-renders when props or state change, using methods like `shouldComponentUpdate` and `componentDidUpdate`.
>
> In the **Unmounting phase**, `componentWillUnmount` is used for cleanup like removing timers or event listeners.

> These lifecycle methods help us control what happens at different points in a component's life."

What is Batch Function?

👱 **Short Interview Answer:**

> "Batching means React groups multiple state updates together to do a single re-render instead of multiple ones.
>
> It improves performance because React doesn't re-render the UI after every single state change.
>
> In React 18, batching happens automatically even inside async functions or promises."

What is Controlled and Uncontrolled Components ?

👱 **Short Interview Answer:**

> "In React, a controlled component is one where the form data is handled by React state, making the component predictable and easier to validate.
>
> An uncontrolled component lets the DOM handle the form data using refs.
>
> Controlled components are preferred for large forms because they give better control over user input."

💬 If interviewer asks which one you prefer:

> "I prefer controlled components because they keep the UI and data in sync through React state, making debugging and validation easier. But for simple inputs or performance-sensitive use cases, uncontrolled components can be faster."

Explain difference b/t var, const and let?

🧩 1️⃣ var

- ✅ **Function Scoped** → available inside the whole function.
- ❌ **Can be re-declared and updated.**

- ⚠️ **Hoisted** — but initialized as `undefined`.
- ❌ Not block-scoped (can create bugs).

🧩2️⃣ let

- ✅ **Block Scoped** (only works inside `{}`)
- ✅ Can be **updated**, ❌ but not **re-declared** in the same scope
- ⚠️ **Hoisted** but not initialized (can't use before declaration)

🧩3️⃣ const

- ✅ **Block Scoped** (like `let`)
- ❌ **Cannot be re-assigned or re-declared.**
- ⚠️ The value must be assigned during declaration.

What is DOM ?

👨 **Short Interview Answer:**

> "The DOM represents the actual structure of a webpage, and updating it directly is slow because the browser has to re-render the entire UI.
>
> The Virtual DOM, used by React, is an in-memory copy of the real DOM.
>
> When state changes, React updates the virtual DOM first, compares it with the previous version, and only updates the changed parts in the real DOM — making updates much faster and efficient."

React Native DOM Concept?

👨 **Short Interview Answer:**

> "In React for web, we have a real DOM and a Virtual DOM — React updates the Virtual DOM, compares it with the previous one, and only updates the changed parts in the real DOM.
>
> But in React Native, there's **no DOM** because it doesn't run in a browser.
>
> Instead, React Native uses a similar concept — it keeps a Virtual Tree in memory, figures out what changed, and then updates the **native UI elements** (like View, Text) through a bridge.

> This makes React Native apps perform like real native apps.”

🎯 onPressIn & Animation - Complete Deep Dive

onPressIn vs onPressOut - Kya hai?

```
// Home.jsx mein ye code hai
const onPressIn = () => {
  Animated.spring(scaleAnim, {
    toValue: 0.95,          // 95% size (5% shrink)
    useNativeDriver: true,   // Performance ke liye
  }).start();
};

const onPressOut = () => {
  Animated.spring(scaleAnim, {
    toValue: 1,             // 100% size (normal)
    useNativeDriver: true,
  }).start();
};
```

Event Lifecycle - Step by Step:

```
// User interaction flow:
1. User finger screen pe touch karta hai → onPressIn trigger
2. User finger screen pe rakha hai → onPressIn state
3. User finger uthata hai → onPressOut trigger
4. Animation complete → normal state
```

Animation Deep Dive:

1. Animated.Value - State Management

```
// Line 78: Animation value create kiya
const scaleAnim = useRef(new Animated.Value(1)).current;

// Ye value 1 se start hota hai (100% size)
```

```
// 0.95 pe jata hai (95% size)
// Phir wapas 1 pe aata hai (100% size)
```

2. Spring Animation - Physics Based

```
Animated.spring(scaleAnim, {
  toValue: 0.95,              // Target value
  useNativeDriver: true,      // Native thread pe run
  // Optional properties:
  tension: 100,               // Spring stiffness (default: 40)
  friction: 8,                // Spring damping (default: 7)
  mass: 1,                    // Spring mass (default: 1)
  delay: 0,                   // Animation delay
  duration: 200,              // Animation duration
}).start();
```

3. useNativeDriver - Performance Magic

```
// useNativeDriver: true
// ✅ Pros:
// - Animation UI thread pe run hota hai
// - 60fps smooth performance
// - JS thread block nahi hota
// - Battery efficient

// ❌ Cons:
// - Sirf transform aur opacity properties work karte hain
// - Layout properties (width, height, margin) work nahi karte

// useNativeDriver: false
// ✅ Pros:
// - Sab properties work karte hain
// ❌ Cons:
// - JS thread pe run hota hai
// - Performance issues
// - Frame drops possible
```

Complete Animation Flow:

```
// Step 1: Animation value create
const scaleAnim = useRef(new Animated.Value(1)).current;

// Step 2: Animation functions
const onPressIn = () => {
  Animated.spring(scaleAnim, {
    toValue: 0.95,          // Shrink to 95%
    useNativeDriver: true,   // Native thread
  }).start();
};

const onPressOut = () => {
  Animated.spring(scaleAnim, {
    toValue: 1,             // Back to 100%
    useNativeDriver: true,
  }).start();
};

// Step 3: Apply animation to component
<Animated.View
  style={[
    styles.offerCardWrapper,
    { transform: [{ scale: scaleAnim }] },  // Scale transform
  ]}
>
  <Pressable
    onPressIn={onPressIn}    // Touch start
    onPressOut={onPressOut}  // Touch end
  >
    {/* Content */}
  </Pressable>
</Animated.View>
```

Transform Properties - Detailed:

```
// Scale transform
{ transform: [{ scale: scaleAnim }] }

// Other transform options:
{ transform: [
  { scale: scaleAnim },          // Size change
  { scaleX: scaleAnim },         // Horizontal scale only
  { scaleY: scaleAnim },         // Vertical scale only
  { rotate: '45deg' },           // Rotation
  { translateX: 10 },            // Horizontal movement
  { translateY: 10 },            // Vertical movement
  { skewX: '10deg' },            // Skew effect
] }
```

Animation Types Comparison:

```
// 1. Spring Animation (Current)
Animated.spring(scaleAnim, {
  toValue: 0.95,
  useNativeDriver: true,
}).start();

// 2. Timing Animation
Animated.timing(scaleAnim, {
  toValue: 0.95,
  duration: 200,
  useNativeDriver: true,
}).start();

// 3. Decay Animation
Animated.decay(scaleAnim, {
  velocity: 0.5,
  deceleration: 0.997,
  useNativeDriver: true,
}).start();
```

📱 Modal in React Native - Complete Explanation

Modal Kya hai?

Modal ek overlay component hai jo main screen ke upar display hota hai. Ye user ko main content se temporarily distract karta hai aur specific action ke liye focus karta hai.

Home.jsx mein Modal Implementation:

```
// Lines 137-173: City selection modal
<Modal
  visible={showDropdown}          // Modal show/hide control
  transparent                     // Background transparent
  animationType="fade"            // Animation type
  onRequestClose={() => setShowDropdown(false)}  // Android back button
>
  <TouchableWithoutFeedback onPress={() => setShowDropdown(false)}>
    <View style={styles.modalOverlay}>
      <TouchableWithoutFeedback>
        <View style={styles.dropdownContainer}>
          <Text style={styles.dropdownTitle}>Select Your City</Text>
          <ScrollView>
            {cityList.map(city => (
              <TouchableOpacity
                key={city}
                style={styles.cityItem}
                onPress={() => {
                  setSelectedCity(city);
                  setShowDropdown(false);
                }}
              >
                <Text style={styles.cityText}>{city}</Text>
              </TouchableOpacity>
            ))}
          </ScrollView>
        </View>
      </TouchableWithoutFeedback>
    </View>
```

```
  </TouchableWithoutFeedback>
</Modal>
```

Modal Props - Detailed Explanation:

1. visible (Required)

```
visible={showDropdown}
// true: Modal show
// false: Modal hide
```

2. transparent

```
transparent={true}
// true: Background transparent (overlay effect)
// false: Background opaque (full screen)
```

3. animationType

```
animationType="fade"     // Fade in/out
animationType="slide"    // Slide from bottom
animationType="none"     // No animation
```

4. onRequestClose

```
onRequestClose={() => setShowDropdown(false)}
// Android back button handle karta hai
// Required prop for Android
```

Modal Structure Breakdown:

```
<Modal>
  {/* Outer TouchableWithoutFeedback - Background click */}
  <TouchableWithoutFeedback onPress={closeModal}>
    <View style={modalOverlay}>
      {/* Inner TouchableWithoutFeedback - Prevent event bubbling */}
```

```
      <TouchableWithoutFeedback>
        <View style={modalContent}>
          {/* Modal content */}
        </View>
      </TouchableWithoutFeedback>
    </View>
  </TouchableWithoutFeedback>
</Modal>
```

Why Nested TouchableWithoutFeedback?

```
// Outer TouchableWithoutFeedback
<TouchableWithoutFeedback onPress={() => setShowDropdown(false)}>
  <View style={styles.modalOverlay}>
    {/* Inner TouchableWithoutFeedback */}
    <TouchableWithoutFeedback>
      <View style={styles.dropdownContainer}>
        {/* Content */}
      </View>
    </TouchableWithoutFeedback>
  </View>
</TouchableWithoutFeedback>
```

Reason:

- **Outer**: Background click pe modal close
- **Inner**: Content click pe modal close nahi hoga (event bubbling prevent)

Modal Styles:

```
// Modal overlay (background)
modalOverlay: {
  flex: 1,
  backgroundColor: 'rgba(0,0,0,0.4)',  // Semi-transparent black
  justifyContent: 'center',
  alignItems: 'center',
},
```

```
// Modal content container
dropdownContainer: {
  width: '85%',                // 85% of screen width
  maxHeight: '60%',            // Maximum 60% of screen height
  backgroundColor: '#fff',     // White background
  borderRadius: 12,            // Rounded corners
  paddingVertical: 12,
  paddingHorizontal: 16,
  shadowColor: '#000',         // Shadow for depth
  shadowOpacity: 0.25,
  shadowRadius: 8,
  elevation: 5,                // Android shadow
},
```

Modal Types - Different Implementations:

1. Simple Alert Modal

```
const SimpleModal = () => {
  const [visible, setVisible] = useState(false);

  return (
    <Modal
      visible={visible}
      transparent
      animationType="fade"
      onRequestClose={() => setVisible(false)}
    >
      <View style={styles.overlay}>
        <View style={styles.modal}>
          <Text>Hello World!</Text>
          <TouchableOpacity onPress={() => setVisible(false)}>
            <Text>Close</Text>
          </TouchableOpacity>
        </View>
      </View>
    </Modal>
```

```
  );
};
```

2. Bottom Sheet Modal

```
const BottomSheetModal = () => {
  const [visible, setVisible] = useState(false);

  return (
    <Modal
      visible={visible}
      transparent
      animationType="slide"
      onRequestClose={() => setVisible(false)}
    >
      <View style={styles.overlay}>
        <View style={styles.bottomSheet}>
          <View style={styles.handle} />
          <Text>Bottom Sheet Content</Text>
        </View>
      </View>
    </Modal>
  );
};

const styles = StyleSheet.create({
  overlay: {
    flex: 1,
    backgroundColor: 'rgba(0,0,0,0.5)',
    justifyContent: 'flex-end',
  },
  bottomSheet: {
    backgroundColor: '#fff',
    borderTopLeftRadius: 20,
    borderTopRightRadius: 20,
    padding: 20,
    maxHeight: '80%',
  },
```

```
  handle: {
    width: 40,
    height: 4,
    backgroundColor: '#ccc',
    borderRadius: 2,
    alignSelf: 'center',
    marginBottom: 20,
  },
});
```

3. Full Screen Modal

```
const FullScreenModal = () => {
  const [visible, setVisible] = useState(false);

  return (
    <Modal
      visible={visible}
      animationType="slide"
      onRequestClose={() => setVisible(false)}
    >
      <View style={styles.fullScreen}>
        <TouchableOpacity onPress={() => setVisible(false)}>
          <Text>Close</Text>
        </TouchableOpacity>
        <Text>Full Screen Content</Text>
      </View>
    </Modal>
  );
};
```

Common Modal Patterns:

1. Confirmation Modal

```
const ConfirmationModal = ({ visible, onConfirm, onCancel }) => {
  return (
```

```
    <Modal visible={visible} transparent animationType="fade">
     <View style={styles.overlay}>
      <View style={styles.modal}>
       <Text>Are you sure?</Text>
       <View style={styles.buttonRow}>
        <TouchableOpacity onPress={onCancel}>
         <Text>Cancel</Text>
        </TouchableOpacity>
        <TouchableOpacity onPress={onConfirm}>
         <Text>Confirm</Text>
        </TouchableOpacity>
       </View>
      </View>
     </View>
    </Modal>
  );
};
```

2. Loading Modal

```
const LoadingModal = ({ visible, message = "Loading..." }) => {
  return (
    <Modal visible={visible} transparent animationType="fade">
     <View style={styles.overlay}>
      <View style={styles.loadingModal}>
       <ActivityIndicator size="large" color="#ff6b00" />
       <Text style={styles.loadingText}>{message}</Text>
      </View>
     </View>
    </Modal>
  );
};
```

📱 TouchableOpacity vs TouchableWithoutFeedback - Complete Difference

Basic Difference:

```
// TouchableOpacity - Visual feedback deta hai
<TouchableOpacity onPress={handlePress}>
  <Text>Press Me</Text>
</TouchableOpacity>

// TouchableWithoutFeedback - Visual feedback nahi deta
<TouchableWithoutFeedback onPress={handlePress}>
  <Text>Press Me</Text>
</TouchableWithoutFeedback>
```

Detailed Comparison:

1. TouchableOpacity

```
<TouchableOpacity
  onPress={handlePress}
  activeOpacity={0.7}        // Default: 0.2
  disabled={false}           // Enable/disable
  style={styles.button}
>
  <Text>Button Text</Text>
</TouchableOpacity>
```

Features:

- ✅ **Visual Feedback**: Opacity change on press
- ✅ **Customizable**: `activeOpacity` prop
- ✅ **Accessibility**: Built-in accessibility support
- ✅ **Easy to use**: Simple implementation
- ❌ **Performance**: Slightly slower (opacity animation)

2. TouchableWithoutFeedback

```
<TouchableWithoutFeedback
  onPress={handlePress}
  disabled={false}
```

```
  accessible={true}
  accessibilityLabel="Custom button"
>
  <View style={styles.button}>
    <Text>Button Text</Text>
  </View>
</TouchableWithoutFeedback>
```

Features:

- ✅ **Performance**: Faster (no opacity animation)
- ✅ **Custom Control**: Full control over visual feedback
- ✅ **Flexible**: Can add custom animations
- ❌ **No Visual Feedback**: Default behavior
- ❌ **Accessibility**: Manual accessibility setup required

Home.jsx mein Usage Examples:

1. TouchableOpacity Usage (Line 118-128)

```
<TouchableOpacity
  style={styles.locationButton}
  onPress={() => setShowDropdown(true)}
>
  <Text style={styles.locationText}>{selectedCity}</Text>
  <Image
    source={arrowDown}
    style={styles.locationArrow}
    resizeMode="contain"
  />
</TouchableOpacity>
```

Why TouchableOpacity here?

- User ko visual feedback chahiye ki button pressable hai
- Simple opacity change sufficient hai
- Accessibility automatically handle hoti hai

2. TouchableWithoutFeedback Usage (Lines 143, 145)

```
<TouchableWithoutFeedback onPress={() => setShowDropdown(false)}>
  <View style={styles.modalOverlay}>
    <TouchableWithoutFeedback>
      <View style={styles.dropdownContainer}>
        {/* Modal content */}
      </View>
    </TouchableWithoutFeedback>
  </View>
</TouchableWithoutFeedback>
```

Why TouchableWithoutFeedback here?

- Background click pe modal close karna hai
- Visual feedback nahi chahiye
- Event handling ke liye use kiya hai

💬 Interview me kaise explain kare:

> CSS Combinators are used to define the relationship between elements in HTML.
>
> They help us target elements based on how they are related to other elements.
>
> There are four main combinators — descendant (`space`), child (`>`), adjacent sibling (`+`), and general sibling (`~`).

34. What are pseudo-classes in CSS?

A Pseudo class in CSS is used to define the special state of an element. It can be combined with a CSS selector to add an effect to existing elements based on their states. For Example, changing the style of an element when the user

hovers over it, or when a link is visited. All of these can be done using Pseudo Classes in CSS.

```
{/* Show Resend Btn Or Resend Timer  */}
{showResend ? (
  /* Resend option */
  <Text style={styles.subHeading}>
    Not Received the OTP Yet{' '}
    <Text style={styles.resendText} onPress={handleResendOtp}>
      Resend?
    </Text>
  </Text>
) : (
  /* Timer */
  <Animated.View
  pointerEvents={showResend ? 'none' : 'auto'}
    style={{
      position: 'absolute',
      top: '65%',
      transform: [{ translateX: timerX }],
      opacity: timerOpacity,
      width: '100%',
      alignItems: 'center',
    }}
  >
    <Text style={styles.resendText}>
      Wait 00:{timer.toString().padStart(2, '0')} sec to resend
      OTP
    </Text>
  </Animated.View>
)}
```