Event Propagation -

1. **What is Event Propagation?**

Event Propagation defines **how events travel through the DOM tree** after being triggered.
It has **three phases**:

1. **Capturing Phase** → Event goes **from top (window/document) to target**.
2. **Target Phase** → Event reaches the actual element (target).
3. **Bubbling Phase** → Event bubbles back **from target to top (document/window)**.

2. **Event Bubbling (Default Behavior)**

- By default, events bubble up from child → parent → ancestor.
- Example:
-
- <button id="child">Click Me</button>
-
- <script>
- document.getElementById("parent").addEventListener("click", () => {
- console.log("Parent clicked");
- });
- document.getElementById("child").addEventListener("click", () => {
- console.log("Child clicked");
- });
- </script>
- **Output if child clicked:**
- Child clicked
- Parent clicked

3. **Event Capturing (Trickling Phase)**

- Less commonly used.
- If you pass true as the third argument in addEventListener, capturing happens.
- Example:
- document.getElementById("parent").addEventListener("click", () => {
- console.log("Parent capturing");
- }, true);
- **Now order:**
- Parent capturing
- Child clicked
- Parent clicked

4. **Stopping Propagation**

- **event.stopPropagation()** → Stops event from moving further (bubbling or capturing).
- **event.stopImmediatePropagation()** → Stops event + prevents other listeners on the same element.

5. **Event Delegation**

- Using event bubbling to manage events efficiently.
- Instead of adding event listeners to multiple child elements, add one to the parent.
- Example:
- document.getElementById("parent").addEventListener("click", (e) => {
- if(e.target.tagName === "BUTTON") {
- console.log("Button clicked:", e.target.textContent);
- }
- });

✍️ Short Notes: Event Propagation

- **3 phases**: Capturing → Target → Bubbling.
- **Default** → Bubbling (child → parent).
- **Capturing** → Use addEventListener(event, fn, true).
- **stopPropagation()** → Stops event flow.
- **stopImmediatePropagation()** → Stops + blocks other listeners.
- **Event Delegation** → Handle multiple child events via parent (performance optimization).

Higher Order Functions -

1. **Definition**

A **Higher Order Function (HOF)** is a function that:

1. **Takes another function as an argument (callback function)**, OR
2. **Returns a function as a result**.

👉 In short, HOFs **work with functions as values**.

2. **Why Important?**

- Makes code **reusable** & **cleaner**.
- Helps in **functional programming** style.
- Common in array methods (map, filter, reduce, etc.).

3. **Examples**

(a) Passing a function as an argument

```
function greet(name) {
  return "Hello, " + name;
}

function processUserInput(callback) {
  let name = "John";
  console.log(callback(name));
}

processUserInput(greet); // Output: Hello, John
```

(b) Returning a function

```
function multiplier(x) {
  return function(y) {
    return x * y;
  };
}

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

(c) Array Methods (Built-in HOFs)

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

const squares = numbers.map(n => n * n);      // [1, 4, 9, 16, 25]
```

```
const evens = numbers.filter(n => n % 2 === 0); // [2, 4]

const sum = numbers.reduce((a, b) => a + b, 0); // 15
```

4. **Popular Higher Order Functions**

- forEach() → Iterates over array.
- map() → Transforms each element → returns new array.
- filter() → Filters elements based on condition.
- reduce() → Reduces array to a single value.
- sort(), find(), every(), some() → Also HOFs.

5. **Benefits**

- Reusability.
- Less repetitive code.
- Improves readability.
- Essential for **functional programming** & **async operations** (like setTimeout, Promise.then).

✍️ Short Notes: Higher Order Functions

- **HOF** → A function that takes/returns another function.
- Examples → map, filter, reduce, forEach.
- Useful in → Reusability, functional programming, cleaner code.
- Real-life → setTimeout(fn, delay), addEventListener(event, fn).

Callback Hell -

1. **What is a Callback?**

- A **callback** is a function passed as an argument to another function, executed later.
- Common in asynchronous tasks (API calls, file reading, timers).

Example:

```
setTimeout(() => {

  console.log("Task done after 2 seconds");

}, 2000);
```

2. **What is Callback Hell?**

- **Callback Hell** happens when we have **too many nested callbacks**, making code:
  - Hard to read (pyramid shape).
  - Hard to debug.
  - Hard to maintain.

It is also called the **Pyramid of Doom**.

3. **Example of Callback Hell**

```
// Nested callbacks

getUserData(1, (user) => {

  getPosts(user.id, (posts) => {

    getComments(posts[0].id, (comments) => {

      getLikes(comments[0].id, (likes) => {

        console.log("Likes:", likes);

      });

    });

  });

});
```

👉 Here the code grows **horizontally** (nested structure), very hard to maintain.

4. **Problems with Callback Hell**

- Poor readability.
- Hard to handle errors.
- Difficult to scale for large projects.

5. **Solutions**

✅ Use **Named Functions** (avoid inline nesting)

```
getUserData(1, handleUser);



function handleUser(user) {

  getPosts(user.id, handlePosts);

}
```

```
function handlePosts(posts) {
  getComments(posts[0].id, handleComments);
}
function handleComments(comments) {
  console.log(comments);
}
```

✅ Use **Promises**

```
getUserData(1)
  .then(user => getPosts(user.id))
  .then(posts => getComments(posts[0].id))
  .then(comments => console.log(comments))
  .catch(error => console.error(error));
```

✅ Use **async/await** (cleanest way)

```
async function fetchData() {
  try {
    const user = await getUserData(1);
    const posts = await getPosts(user.id);
    const comments = await getComments(posts[0].id);
    console.log(comments);
  } catch (err) {
    console.error(err);
  }
}
fetchData();
```

✍️ Short Notes: Callback Hell

- **Callback Hell** = too many nested callbacks (pyramid of doom).
- Problems → unreadable, hard to debug, not scalable.
- Solutions →
  - Named functions.
  - Promises (then, catch).
  - async/await (best).

Promises in JS -

1. **What is a Promise?**

- A **Promise** is an **object** that represents the eventual **completion or failure** of an asynchronous operation.
- Think of it like a **guarantee of a future value**.

👉 Instead of relying on nested callbacks, Promises make async code **cleaner & manageable**.

2. **Promise States**

A Promise can be in **3 states**:

1. **Pending** → Initial state, neither fulfilled nor rejected.
2. **Fulfilled** → Operation completed successfully (resolved).
3. **Rejected** → Operation failed (error occurred).

3. **Creating a Promise**

```
const myPromise = new Promise((resolve, reject) => {

 let success = true;



 if (success) {

  resolve("Operation successful!");

 } else {

  reject("Something went wrong!");

 }
```

```
});
```

4. **Using Promises**

```
myPromise
  .then(result => console.log(result))   // "Operation successful!"
  .catch(error => console.error(error))  // "Something went wrong!"
  .finally(() => console.log("Done!"));  // Always runs
```

5. **Promise Chaining**

```
getUserData(1)
  .then(user => getPosts(user.id))
  .then(posts => getComments(posts[0].id))
  .then(comments => console.log(comments))
  .catch(err => console.error(err));
```

6. **Promise Methods**

- **Promise.all([p1, p2, ...])** → Runs all promises in parallel, waits until all are resolved (or rejected).
- **Promise.race([p1, p2, ...])** → Returns the first settled (resolved/rejected) promise.
- **Promise.allSettled([p1, p2, ...])** → Returns results of all promises, regardless of resolve/reject.
- **Promise.any([p1, p2, ...])** → Returns the first successfully resolved promise (ignores rejections).

7. **Promise Example with setTimeout**

```
function delay(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

delay(2000).then(() => console.log("Executed after 2 seconds"));
```

8. **Promises vs Callbacks**

Callbacks	Promises
Can cause **Callback Hell**	Avoids nested callbacks
Harder to read & maintain	Easier to read
Error handling tricky	Built-in .catch() for errors

✍️ Short Notes: Promises

- **Promise** = future value of async task.
- **States**: Pending → Fulfilled → Rejected.
- **Methods**:
  - .then() → success
  - .catch() → error
  - .finally() → always
- **Helpers**: Promise.all, Promise.race, Promise.allSettled, Promise.any.
- Cleaner alternative to callbacks.

Async/Await -

1. **What is Async/Await?**

- **Async/Await** is **syntactic sugar** built on top of **Promises**.
- Makes asynchronous code look and behave like **synchronous code** → easier to **read, write, and debug**.

2. **The async Keyword**

- Declares an **asynchronous function**.
- An async function always returns a **Promise**.

Example:

```
async function greet() {

 return "Hello World!";
```

```
}

greet().then(msg => console.log(msg)); // "Hello World!"
```

3. **The await Keyword**

- Used **inside async functions only**.
- Pauses execution until the Promise is resolved or rejected.

Example:

```
function delay(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}


async function run() {
  console.log("Start");
  await delay(2000);   // waits 2 sec
  console.log("After 2 seconds");
}
run();
```

4. **Async/Await Example with API**

```
async function fetchData() {
  try {
    let response = await fetch("https://jsonplaceholder.typicode.com/posts/1");
    let data = await response.json();
    console.log(data);
  } catch (error) {
    console.error("Error:", error);
  }
}
```

```
fetchData();
```

5. **Error Handling in Async/Await**

- Use **try/catch** for error handling.

```
async function getUser() {
  try {
    let response = await fetch("wrong-url"); // invalid URL
    let data = await response.json();
    console.log(data);
  } catch (err) {
    console.error("Something went wrong:", err);
  }
}
```

6. **Running Async in Parallel**

👉 If tasks are independent, use Promise.all with await.

```
async function runTasks() {
  const [posts, users] = await Promise.all([
    fetch("https://jsonplaceholder.typicode.com/posts").then(res => res.json()),
    fetch("https://jsonplaceholder.typicode.com/users").then(res => res.json())
  ]);

  console.log(posts.length, users.length);
}
runTasks();
```

7. **Async/Await vs Promises**

Promises	Async/Await
Uses .then() chaining	Uses await, looks synchronous
Nested .then() can still get messy	Cleaner, easier to read
Errors handled with .catch()	Errors handled with try/catch

✍️ Short Notes: Async/Await

- async → makes a function return a Promise.
- await → waits for a Promise to resolve/reject (only inside async).
- Cleaner alternative to .then() chaining.
- Use **try/catch** for error handling.
- For parallel tasks → Promise.all() with await.