JavaScript Notes (Digital Version)

Event Bubbling

When an event is triggered on a child element, it automatically travels upward towards its parent elements until it reaches the root. Using `event.stopPropagation()` we can stop this behaviour.

Synchronous Code *(Important)*

Synchronous means the code runs in a particular sequence exactly as written. It executes line-by-line and follows sequence-based programming.

Asynchronous Code

Asynchronous execution allows the next instruction to run immediately without blocking the flow. One section does not wait for another.

Event Looping

JavaScript is single-threaded, so it uses the **Event Loop** to handle asynchronous tasks. JS runs one line at a time in the call stack. When it encounters async tasks like `setTimeout` or `fetch`, they move to browser Web APIs. After completion, callbacks are moved to queues. - Promises → Microtask Queue - Timeouts → Macrotask Queue

Execution Order

1. Synchronous Code (Call Stack)
2. Microtasks
3. Macrotasks

Callback

A callback is a function that is passed as an argument to another function. Example: `array.forEach(function() {})`

Callback Hell

Nested callbacks stacked inside one another forming a pyramid structure. Difficult to manage.

Promises

A Promise represents the eventual completion of a task. It is an object in JS. It solves callback hell.

Promise States

- Pending
- Resolved → handled with `.then()`
- Rejected → handled with `.catch()`

`new Promise((resolve, reject) => { ... })` automatically creates two handlers.

Promise Chaining

Running async tasks one after another using `.then()`, where each `.then()` waits for the previous one. Example:

```
new Promise(resolve => resolve(1))
  .then(num => num + 1)
  .then(n => n * 5)
  .then(result => console.log(result)); // 10
```

Note: Arrow function with single expression returns automatically.

Async–Await

- An `async` function always returns a promise.
- `await` pauses execution until the promise is resolved.

Example:

```
function getData() {
  return new Promise(resolve => {
    setTimeout(() => resolve("Data"), 2000);
  });
}
```

```
async function showData() {
  const result = await getData();
  console.log(result);
}

showData();
```

IIFE (Immediately Invoked Function Expression)

A function that runs immediately after its creation.

Syntax:

```
(function(){})();
(() => {})();
```

Example:

```
(async function(){
  const result = await getData();
  console.log(result);
})();
```

Additional Important Topics (Extended Notes)

Microtask vs Macrotask (Detailed but Simple)

JavaScript processes asynchronous tasks in two separate queues:

Microtask Queue

Runs **before** macrotasks. Highest priority. Contains: - Promises (`.then`, `.catch`, `.finally`) - `queueMicrotask()`

Macrotask Queue

Runs **after** microtasks. Contains: - `setTimeout` - `setInterval` - DOM events - File readers

Why this matters?

Because JavaScript will *always* finish microtasks before moving to macrotasks.

Example:

```
console.log("1");
setTimeout(() => console.log("2"), 0);
Promise.resolve().then(() => console.log("3"));
console.log("4");
```

Output: `1 4 3 2`

Fetch API (Important Async Concept)

`fetch()` returns a Promise. Used to call APIs.

Example:

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

Using Async–Await:

```
async function getData() {
  try {
    const res = await fetch("https://api.example.com/data");
    const data = await res.json();
    console.log(data);
  } catch (err) {
    console.error(err);
  }
}
```

Error Handling in Async Code

Promises: use `.catch()` Async–await: use `try...catch`

Example:

```
async function load() {
  try {
    let res = await fetch("wrong-url");
  } catch (error) {
    console.log("Error happened:", error);
  }
}
```

Promise.all / Promise.race / Promise.allSettled

Used to run multiple async tasks.

Promise.all()

Runs all promises together → waits until all complete.

```
Promise.all([fetch(url1), fetch(url2)])
  .then(results => console.log("All Done"));
```

If **any one** fails → whole thing fails.

Promise.race()

Returns result of *first finished* promise.

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

Promise.allSettled()

Waits for all promises, even if some fail. Useful for safe execution.

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

---

setTimeout vs setInterval

setTimeout → runs once

```
setTimeout(() => console.log("Run once"), 2000);
```

setInterval → repeats

```
setInterval(() => console.log("Repeating"), 1000);
```

Event Capturing vs Event Bubbling (missing detail)

Capturing (Top → Bottom)

Event moves from window → document → parent → child.

Bubbling (Bottom → Top)

Event moves from child → parent → document → window.

Add listener with phases:

```
// capturing
addEventListener("click", handler, true);

// bubbling (default)
addEventListener("click", handler, false);
```

Call Stack, Web APIs, Callback Queue

Important flow of async execution: 1. JS runs code in Call Stack. 2. Async tasks go to Web APIs. 3. Finished tasks move callbacks to queues. 4. Event Loop pushes them back to Call Stack.

Simple Example:

```
console.log("A");
setTimeout(() => console.log("B"), 0);
console.log("C");
```

Output: `A C B`

If you need, I can also add diagrams (text-based), examples of debounce, throttle, or more async patterns.