Series 2: JavaScript Intermediate Concepts – Part 1

✅ 1. Event Loop

👨‍💻 Interview Q&A:

Q: What is the Event Loop in JavaScript?

> The event loop is a mechanism that allows JavaScript to perform **non-blocking operations**, like I/O, by **offloading tasks** and then checking when they're ready to run.

🧪 Example:

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

setTimeout(() => {
  console.log("Inside timeout");
}, 0);

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

Output:

- Start
- End
- Inside timeout

🎯 Analogy:

Think of a restaurant:

- The **call stack** is the chef making orders.

- The **event loop** is the waiter checking if any delayed orders (setTimeout) are ready.
- Only when the chef is free, the waiter hands in the next ready order.

✅ 2. Call Stack & Task Queue

👨‍💻 Interview Q&A:

Q: What is the Call Stack in JavaScript?

The call stack keeps track of function calls — last-in, first-out (LIFO).
Once it's empty, the event loop pushes callbacks from the task queue onto the stack.

🧪 Example:

```
function one() {
  two();
}
function two() {
  console.log("Inside two");
}
one();
```

🎯 Analogy:

Call stack is like a **stack of dishes** — you can only remove the top one.
Task queue is like a **line of people** waiting to be served once the dishes are cleared.

✅ 3. `setTimeout` and `setInterval`

👨‍💻 Interview Q&A:

Q: How do `setTimeout` and `setInterval` work?

Both are **asynchronous browser APIs**:

- `setTimeout` runs a function **once after a delay**.
- `setInterval` runs a function **repeatedly every X ms**.
  They're handled via the event loop — not instantly!

🧪 Example:

```
setTimeout(() => {
  console.log("Hello after 1s");
}, 1000);

const intervalId = setInterval(() => {
  console.log("Repeating...");
}, 2000);

// clearInterval(intervalId); // to stop it
```

🎯 Analogy:

- `setTimeout` is like an alarm set for a specific time.
- `setInterval` is like a **reminder app** that pings you every X minutes.

✅ 4. Callbacks

👨‍💻 Interview Q&A:

Q: What are callbacks in JavaScript?

A callback is a function passed as an argument to another function to be executed **later**, often after an async task finishes.

🧪 Example:

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

greet("Alice", () => {
  console.log("Callback called!");
});
```

🎯 Analogy:

A callback is like saying, “Call me when the pizza arrives.”
The function calls you **back** when the job is done.

✅ 5. Promises

👨‍💻 Interview Q&A:

Q: What is a Promise in JavaScript?

A Promise represents the **future result** of an asynchronous operation.
It can be in one of three states: **pending**, **fulfilled**, or **rejected**.

🧪 Example:

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => resolve("Done!"), 1000);
});

promise.then(result => console.log(result)); // "Done!"
```

🎯 Analogy:

A promise is like ordering food:
Pending = you're waiting,
Fulfilled = food is served,
Rejected = kitchen ran out.

✅ 6. `async/await`

👨‍💻 Interview Q&A:

Q: How does `async/await` work in JavaScript?

`async` makes a function return a Promise.
`await` pauses the function execution until the Promise settles (either resolved or rejected).
Cleaner alternative to `.then()` chains.

🧪 Example:

```
async function fetchData() {
  const response = await fetch('/api');
  const data = await response.json();
  console.log(data);
}
```

🎯 Analogy:

Using `await` is like saying, "Wait for the microwave to finish before you eat."
You pause until it's done.

✅ 7. Error Handling (`try/catch`, `.catch`)

👨‍💻 Interview Q&A:

Q: How do you handle errors in async code?

> In promises, use `.catch()` to catch errors.
> In `async/await`, wrap the code in `try/catch` for cleaner error handling.

🧪 Example:

```
async function getUser() {
  try {
    const res = await fetch('/user');
    const data = await res.json();
    console.log(data);
  } catch (err) {
    console.error("Error fetching user:", err);
  }
}
```

🎯 Analogy:

Error handling is like wearing a helmet — you hope nothing crashes, but you're ready if it does.

✅ 8. Currying

👨‍💻 Interview Q&A:

Q: What is currying in JavaScript?

> Currying is a technique where a function with multiple arguments is **transformed** into a sequence of functions, each taking a **single argument**.

🧪 Example:

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

```
  };
}

console.log(add(2)(3)); // 5
```

🎯 Analogy:

Currying is like ordering pizza step-by-step:
First choose size → then crust → then toppings → finally place the order.

✅ 9. Debouncing & Throttling

👨‍💻 Interview Q&A:

Q: What's the difference between debouncing and throttling?

- **Debounce**: Delay execution until the user stops triggering the event (good for search).
- **Throttle**: Ensures the function runs at most once every X ms (good for scroll events).

🧪 Example (Debounce):

```
function debounce(fn, delay) {
  let timer;
  return (...args) => {
    clearTimeout(timer);
    timer = setTimeout(() => fn(...args), delay);
  };
}
```

🎯 Analogy:

- Debounce = Wait until user stops typing before firing a search request.

- Throttle = Only allow scrolling logic to run once every 200ms, no matter how fast the user scrolls.
