Callbacks, Callback Chains, Promises, Async/Await — Complete Guide

Below is a clean, structured explanation with examples from **basic → advanced**, including **interview questions + answers**.

---

1. What is a Callback?

A **callback** is a function passed as an argument to another function, and it is executed after some operation completes.

Example 1 — Basic Callback (No `setTimeout`)

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

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

greet("John", sayBye);
```

Example 2 — Callback with `setTimeout` (Async Callback)

```
function fetchData(callback) {
  setTimeout(() => {
    console.log("Data fetched");
    callback();
  }, 1000);
}

fetchData(() => console.log("Process completed"));
```

Example 3 — Real-time Use Case (Simulating API Calls)

```
function getUser(id, callback) {
  setTimeout(() => {
    console.log("Fetched user");
    callback({ id, name: "Alice" });
  }, 1000);
}
```

```
function getOrders(user, callback) {
  setTimeout(() => {
    console.log("Fetched orders for", user.name);
    callback(["order1", "order2"]);
  }, 1000);
}

getUser(1, (user) => {
  getOrders(user, (orders) => {
    console.log("Orders:", orders);
  });
});
```

This type of nested callbacks is called a **callback chain** → also known as **callback hell**.

---

2. What is Callback Hell / Callback Chain?

Callback Hell happens when there are **multiple nested callbacks**, making the code:

- Hard to read
- Hard to maintain
- Hard to debug

Example of callback chain (API simulation):

```
step1(() => {
  step2(() => {
    step3(() => {
      step4(() => {
        console.log("Done");
      });
    });
  });
});
```

---

3. What is a Promise?

A **Promise** is an object representing the eventual completion (or failure) of an asynchronous operation.

A Promise has three states:

- **pending**
- **fulfilled**
- **rejected**

Basic Promise Example

```
const promise = new Promise((resolve, reject) => {
  const success = true;
  success ? resolve("Success!") : reject("Failed!");
});

promise.then(console.log).catch(console.error);
```

---

4. Convert Callback Examples → Promise Version

✅Example 1 Converted (Basic Callback → Promise)

```
function greet(name) {
  return new Promise((resolve) => {
    console.log("Hello " + name);
    resolve();
  });
}

greet("John").then(() => console.log("Goodbye!"));
```

✅Example 2 Converted (setTimeout Callback → Promise)

```
function fetchData() {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log("Data fetched");
      resolve();
    }, 1000);
  });
}

fetchData().then(() => console.log("Process completed"));
```

✅Example 3 Converted (Real-time API Chain → Promise Chain)

```
function getUser(id) {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log("Fetched user");
      resolve({ id, name: "Alice" });
    }, 1000);
  });
}

function getOrders(user) {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log("Fetched orders for", user.name);
      resolve(["order1", "order2"]);
    }, 1000);
  });
}

getUser(1)
  .then(getOrders)
  .then((orders) => console.log("Orders:", orders));
```

---

5. Converting Promises → Async/Await

Async/Await version of Example 1

```
async function greetUser() {
  await greet("John");
  console.log("Goodbye!");
}

greetUser();
```

Async/Await version of Example 2

```
async function run() {
  await fetchData();
  console.log("Process completed");
}

run();
```

Async/Await version of Example 3 (Real-time use case)

```
async function getUserOrders() {
  const user = await getUser(1);
  const orders = await getOrders(user);
  console.log("Orders:", orders);
}
```

```
getUserOrders();
```

Why Async/Await is Better?

- Looks like synchronous code
- Easier to debug
- Avoids callback hell
- Avoids long `.then()` chains

---

6. Interview Questions + Answers (Callbacks, Promises, Async/Await)

Q1. What is a callback?

A: A callback is a function passed as an argument to another function to be executed later.

Q2. What is callback hell and how do you fix it?

A: Callback Hell is nested callbacks leading to unreadable code. Fix using:

- Promises
- Async/Await
- Modular functions

Q3. What is the difference between synchronous & asynchronous callbacks?

A:

- Synchronous callbacks execute immediately (`Array.map`, `forEach`).
- Asynchronous callbacks execute later (`setTimeout`, API calls).

Q4. What is a Promise?

A: A Promise is an object representing the future result of an async operation.

Q5. What is the difference between *``* chaining and async/await?

A:

- `.then()` uses promise chaining
- `async/await` gives cleaner code and avoids nesting

Q6. What happens if you don't handle Promise rejection?

A: Node will throw an **UnhandledPromiseRejectionWarning**.

Q7. What is the difference between * `** and **` *?

A:

- `resolve` moves the promise to **fulfilled** state
- `reject` moves it to **rejected** state

Q8. Can async function run without await?

A: Yes, it returns a Promise immediately.

Q9. What is the event loop?

A: It is JavaScript's mechanism for handling async operations by moving callbacks from the task queue to the call stack.

Q10. Why is async/await preferred?

A:

- Avoids callback hell
- Avoids promise chaining
- Cleaner and more maintainable

---

7. Summary (for Interview Revision)

Concept	Description
Callback	Function passed inside another function
Callback Hell	Deeply nested callbacks
Promise	Object representing async result
then/catch	Promise chaining
async/await	Cleaner syntax for promises

---

8. Additional Interview Questions (Mid-level + Senior-level)

Mid-Level Questions

Q1. What is the difference between microtasks and macrotasks?

Answer:

- **Microtasks:** Promise callbacks (`then`, `catch`, `finally`), MutationObserver, queueMicrotask
- **Macrotasks:** setTimeout, setInterval, setImmediate, I/O, UI rendering

Execution order: Microtasks always run **before** the next macrotask.

Q2. What is Promise.all and when do you use it?

Answer: `Promise.all()` runs multiple asynchronous tasks in **parallel** and resolves when **all** of them complete.

```
Promise.all([
  fetchUser(),
  fetchOrders(),
  fetchNotifications()
]).then(([user, orders, notifications]) => {
  console.log(user, orders, notifications);
}).catch(err => console.error("One of the promises failed", err));
```

Use when all tasks are independent but results are needed together.

Q3. What is Promise.race and Promise.any?

Answer:

- **Promise.race()** → resolves/rejects when the **first promise completes**.
- **Promise.any()** → resolves when **first successful promise** completes; ignores rejections.

Q4. What are async function error-handling patterns?

Answer: Using `try...catch`:

```
async function run() {
  try {
    const data = await fetchData();
  } catch (err) {
    console.error(err);
  }
}
```

Senior-Level Questions

Q1. Explain the complete lifecycle of a promise.

Answer:

1. Starts as **pending**.
2. Moves to:
3. **fulfilled** → `resolve()` called
4. **rejected** → `reject()` called
5. Goes to **microtask queue**.
6. `.then()` or `.catch()` runs after current call stack.

Q2. What happens internally when you call async/await?

Answer:

- An `async` function **always returns a Promise**.
- `await` **pauses execution** of the async function until the promise resolves.
- Behind the scenes, it transforms into a chain of `.then()` calls.

Q3. Explain Event Loop with a real-world analogy.

Answer: Think of JavaScript as a chef (single thread). Tasks that take long (API calls, timers) go to assistants (web APIs). When done, assistants put callbacks into a queue. The chef picks tasks from the queue when free.

Q4. What are concurrency vs parallelism in JavaScript?

Answer:

- **Concurrency:** Handling multiple tasks by switching between them.
- **Parallelism:** Running tasks at EXACT same time (JS can't do this natively; worker threads needed).

---

9. Visual Comparison Tables

Callbacks vs Promises vs Async/Await

Feature	Callback	Promise	Async/Await
Syntax	Nested	Chained	Clean & readable
Error Handling	Hard	`.catch()`	`try/catch`
Readability	Low	Medium	High
Avoids callback hell?	❌	✔️	✔️
Debugging	Hard	Medium	Easy

Microtask vs Macrotask Execution Table

Type	Examples	Priority
Microtask	Promises, queueMicrotask	Highest
Macrotask	setTimeout, setInterval, I/O	Runs after microtasks

---

10. Detailed Example (Real World) — From Callback → Promise → Async/Await

Use Case: Fetch User → Fetch Orders → Fetch Shipping Info

A. Callback Version (Callback Hell Demonstration)

```
function getUser(id, callback) {
  setTimeout(() => {
    callback(null, { id, name: "Alice" });
  }, 1000);
}

function getOrders(user, callback) {
  setTimeout(() => {
    callback(null, ["order1", "order2"]);
  }, 1000);
}

function getShipping(order, callback) {
  setTimeout(() => {
    callback(null, "Shipped via Express");
  }, 1000);
}

getUser(1, (err, user) => {
  getOrders(user, (err, orders) => {
    getShipping(orders[0], (err, shipping) => {
      console.log(shipping);
    });
  });
});
```

Problem: deep nesting, poor readability.

B. Promise Version (Flattened and Structured)

```
function getUser(id) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (!id) return reject("Invalid user ID");
      resolve({ id, name: "Alice" });
    }, 1000);
  });
}

function getOrders(user) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (!user) return reject("User not found");
      resolve(["order1", "order2"]);
    }, 1000);
  });
}

function getShipping(order) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (!order) return reject("Order not found");
      resolve("Shipped via Express");
    }, 1000);
  });
}

getUser(1)
  .then(getOrders)
  .then((orders) => getShipping(orders[0]))
  .then((shipping) => {
    console.log("Shipping:", shipping);
  })
  .catch((err) => {
    console.error("Error:", err);
  });
```

Why this Promise version is better than callbacks: - No deep nesting — the flow is **linear**. - A single `.catch()` handles all errors in the chain. - Easier to extend (you can add more `.then()` steps).

C. Async/Await Version (Cleanest Implementation)

```
async function processOrder() {
  try {
    const user = await getUser(1);           // wait for user
    const orders = await getOrders(user);    // wait for orders
    const shipping = await getShipping(orders[0]); // wait for shipping info

    console.log("Shipping:", shipping);
  } catch (err) {
    console.error("Error while processing order:", err);
  }
}

processOrder();
```

Why the async/await version is best: - Code looks like normal synchronous code → easier to read. - Error handling with `try/catch` is very clear. - No callback hell and no long `.then()` chains.