Callback Functions

A **callback** is a function that is passed as an argument to another function and executed later.

Example

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

greet("Neeraj", () => {
  console.log("Welcome!");
});
```

Where callbacks are used?

- setTimeout()
- Event listeners
- API requests (old style)
- File reading

Callback Hell (Pyramid of Doom)

When too many callbacks are nested inside each other, code becomes:

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

Example (Callback Hell)

```
setTimeout(() => {
  console.log("1");

  setTimeout(() => {
    console.log("2");

    setTimeout(() => {
      console.log("3");

    }, 1000);
  }, 1000);
}, 1000);
```

This deep pyramid structure is called the **Pyramid of Doom**.

Promises is JavaScript

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

States of a Promise

1. **Pending**
2. **Fulfilled (resolved)**
3. **Rejected**

Creating a Promise

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

  if (success) resolve("Done");
  else reject("Error");
});
```

.then() and .catch()

Used to handle results of a promise.

.then() → for success

```
promise.then(result => {
  console.log("Success:", result);
});
```

.catch() → for errors

```
promise.catch(error => {
  console.log("Error:", error);
});
```

Promise Chaining

You can chain multiple `.then()` calls in sequence.

Example

```
new Promise(resolve => resolve(1))
  .then(num => {
    console.log(num);
    return num + 1;
  })
  .then(num => {
    console.log(num);
    return num + 1;
  })
  .then(num => console.log(num));
```

Each `.then()` returns a **new Promise**.

Attaching Multiple Handlers

You can attach multiple `.then()` to the same promise.
Each one will run independently.

Example

```
let p = Promise.resolve("Hello");

p.then(res => console.log("Handler 1:", res));
p.then(res => console.log("Handler 2:", res));
```

Promise API (Most Important Methods)

A. Promise.resolve(value)

Creates an already-resolved promise.

```
Promise.resolve(5).then(console.log);
```

B. Promise.reject(error)

Creates an already-rejected promise.

```
Promise.reject("Fail!").catch(console.log);
```

C. Promise.all([...])

Runs **multiple promises in parallel**.
Returns results **only if all succeed**.

```
Promise.all([
  fetch("/user"),
  fetch("/posts")
]);
```

If **any one fails**, entire Promise.all fails.

D. Promise.race([...])

Returns the result of the **first completed promise**
(success or fail).

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

E. Promise.allSettled([...])

Returns results when **all promises finish**
(regardless success or failure).

```
Promise.allSettled([p1, p2, p3]);
```

F. Promise.any([...])

Returns the **first fulfilled promise** (ignores failures).

```
Promise.any([p1, p2]).then(console.log);
```

Converting Callback Hell → Promises

Callback Hell

```
doTask1(function(result1) {
  doTask2(result1, function(result2) {
    doTask3(result2, function(result3) {
      console.log(result3);
    });
  });
});
```

Promise Version (Clean)

```
doTask1()
  .then(doTask2)
  .then(doTask3)
  .then(console.log)
  .catch(console.error);
```

Example: Promise with Timeout

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

delay(2000).then(console.log);
```

Error Handling in Chains

```
promise
  .then(step1)
  .then(step2)
  .catch(err => console.log("Error:", err));
```

JavaScript Notes – Async / Await & Error Handling

async Function

The **async** keyword is used to create a function that always returns a **Promise**.

Syntax

```
async function myFunc() {
  return "Hello";
}
```

This automatically becomes:

```
Promise.resolve("Hello")
```

await Keyword

await pauses the execution of an async function until a **Promise settles**.

Syntax

```
let result = await promise;
```

Example

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

async function start() {
  console.log("Waiting...");
  await delay(2000);
  console.log("Done!");
}

start();
```

Note: `await` works **only inside async functions**.

async + await Example

```
async function getUser() {
  let response = await fetch("https://api.example.com/user");
  let data = await response.json();
  console.log(data);
}

getUser();
```

Error Handling with try…catch

Errors generated inside async functions can be handled using **try–catch**.

Example

```
async function fetchData() {
  try {
    let res = await fetch("https://wrong-url.com/api");
    let data = await res.json();
    console.log(data);

  } catch (error) {
    console.log("Error Occurred:", error);
  }
}

fetchData();
```

finally Block

The **finally** block always runs:

- whether try succeeds
- or catch catches error

Used for cleanup code.

Example

```
async function processData() {
  try {
    let response = await fetch("/api");
    let data = await response.json();
    console.log(data);

  } catch (err) {
    console.log("Error:", err);

  } finally {
    console.log("Operation Finished");
  }
}
```

Throwing Custom Errors

You can throw your own meaningful error messages.

Syntax

```
throw new Error("Custom error message");
```

Example

```
function divide(a, b) {
  if (b === 0) {
    throw new Error("Division by zero is not allowed");
  }
  return a / b;
}
```

Handling the custom error:

```
try {
  divide(10, 0);
} catch (err) {
  console.log("Caught:", err.message);
}
```

Throwing Errors Inside async Functions

```
async function getData() {
  try {
    let response = await fetch("url");

    if (!response.ok) {
      throw new Error("Failed to fetch: " + response.status);
    }

    let data = await response.json();
    return data;

  } catch (err) {
    console.log("Error:", err);
  }
}
```

Using await with Promise Chains

```
async function run() {
  let step1 = await Promise.resolve("A");
  let step2 = await Promise.resolve("B");
  let step3 = await Promise.resolve("C");

  console.log(step1, step2, step3);
}
run();
```

Re-throwing Errors

You can catch an error and throw it again.

```
async function loadData() {
  try {
    let data = await fetch("/api");
    return data;

  } catch (err) {
    console.log("Handled locally, sending up...");
    throw err;   // re-throw
  }
}
```

Summary

Concept	Meaning
`async`	Makes a function return a Promise
`await`	Waits for a Promise result
`try`	Code that may fail
`catch`	Catches errors
`finally`	Runs always
`throw`	Creates custom errors

What is OOP? (Object-Oriented Programming)

OOP is a programming style where everything is represented as **objects**.

Objects contain:

- **Properties** → variables (data)
- **Methods** → functions (behavior)

Goals of OOP

- Organize code into reusable structures
- Reduce duplication
- Make code easy to maintain
- Provide real-world modeling

Pillars of OOP

1. **Encapsulation** – bundling data & methods inside an object
2. **Inheritance** – one class can use properties/methods of another
3. **Polymorphism** – same method behaves differently
4. **Abstraction** – hide unnecessary details

Objects in JavaScript

In JavaScript, objects are key–value pairs.

Example

```
let user = {
  name: "Neeraj",
  age: 23,
  greet() {
    console.log("Hello!");
  }
};
```

What is a Constructor?

A **constructor** is a special function used to create and initialize objects.

In JavaScript, constructors are usually written using:

- Function constructor (old)
- `class` constructor (modern, ES6)

Constructor Function (Old Style)

Syntax

```
function Person(name, age) {
  this.name = name;
  this.age = age;
  this.greet = function() {
    console.log("Hello " + this.name);
  };
}
```

Creating objects:

```
let p1 = new Person("Aman", 22);
let p2 = new Person("Rahul", 25);

p1.greet();
```

`new` keyword creates a new object and binds `this` to it.

Constructor in ES6 Classes (Modern)

Modern and recommended way.

Syntax

```
class Car {
  constructor(model, color) {
    this.model = model;
    this.color = color;
  }

  start() {
    console.log(this.model + " Started");
  }
}
```

Creating objects:

```
let c1 = new Car("BMW", "Black");
c1.start();
```

Points About Constructors

- Only one constructor per class
- Automatically called when object is created
- Used to set initial values
- No return statement required
- If you don't write a constructor, JS adds an **empty constructor** automatically

Default Constructor

If no constructor is defined:

```
class Student {}
```

JS internally creates:

```
constructor() {}
```

Example: Class With Constructor

```
class Student {
  constructor(name, city) {
    this.name = name;
    this.city = city;
  }

  info() {
    console.log(this.name + " from " + this.city);
  }
}

let s1 = new Student("Neeraj", "Lucknow");
s1.info();
```

Real-life Example

```
class BankAccount {
  constructor(owner, balance) {
    this.owner = owner;
    this.balance = balance;
  }

  deposit(amount) {
    this.balance += amount;
  }
}

let a1 = new BankAccount("Neeraj", 5000);
console.log(a1.balance);
```