02-07-25

4.Promises

A **Promise** in JavaScript represents a value that may be **available now**, **later**, or **never**. It has three states:

1. **Pending** – the initial state
2. **Fulfilled** – the operation completed successfully
3. **Rejected** – the operation failed

realworld example:

- **attending an interview**
  • If selected:
  fulfilled – "You’re hired!"
  • If not selected:
  rejected – "You’re not selected."

```
let result = new Promise((resolve, reject) ⇒ {
  let passed = false;

  setTimeout(() ⇒ {
    if (passed) {
      resolve("You're hired! ");
    } else {
      reject("Sorry, not this time ");
    }
  }, 2000);
});

result
```

```
  .then((result) => console.log(result))
  .catch((error) => console.log(error));
```

.ordering food

```
const orderfood= new Promise((resolve, reject)=> {
    let foodready= true;
    setTimeout(()=>{
        if(foodready){
            resolve("food is ready")
        }
        else{
            reject("sorry, food is out of stock")
        }
    },4000)
})
orderfood
.then((message)=>{
    console.log("Success:", message)
})
.catch((error)=>{
    console.log("failure:",error)
})
```

💡 .then() is used to handle the **success** (fulfilled) case of a promise.

.catch() is used to handle the **failure** (rejected) case of a promise.

Promise Concurrency

1. promise.all():
   wait for all to finish
   Fulfills when

all of the promises fulfill; rejects when **any** of the promises rejects.
example:
Downloading three files simultaneously and proceeding only after all are done.

2. promise.allsettled():
   wait for all regardless of success or failure
   Fulfills when
   all promises settle.
   example:
   uploading files, as we want to know which one is succeed and which one is failed

3. promise.race():
   take the fastest one
   Settles when
   any of the promises settles. fulfills when any of the promises fulfills; rejects when any of the promises rejects.

   example: Multiple content servers, use whichever responds first.

4. promise.any():
   the
   first success, and ignore failures.
   Fulfills when
   any of the promises fulfill; rejects when **all** of the promises rejects.
   example:
   payment gateways

examples:

promise.all():

```
const downloadFile = (file) ⇒
  new Promise((resolve) ⇒
    setTimeout(() ⇒ resolve(`${file} downloaded`), Math.random() * 2000)
  );
```

```
Promise.all([
  downloadFile('file1.pdf'),
  downloadFile('file2.pdf'),
  downloadFile('file3.pdf')
]).then((results) ⇒ {
  console.log('All files downloaded:', results);
});
```

promise.allsettled:

```
const uploadFile = (file, shouldFail = false) ⇒
  new Promise((resolve, reject) ⇒
    setTimeout(() ⇒ {
      shouldFail ? reject(`${file} failed`) : resolve(`${file} uploaded`);
    }, Math.random() * 2000)
  );

Promise.allSettled([
  uploadFile('photo1.jpg'),
  uploadFile('photo2.jpg', true),
  uploadFile('photo3.jpg')
]).then((results) ⇒ {
  results.forEach((result) ⇒ console.log(result.status, result.value || result.reason
});
```

```
const generatePDF = () ⇒ Promise.resolve('PDF ready');
const exportCSV = () ⇒ Promise.reject('CSV export failed');
const generateReport = () ⇒ Promise.resolve('Report ready');

Promise.allSettled([generatePDF(), exportCSV(), generateReport()]).then(console
```

promise.race():

```
const fetchFromServer = (server) ⇒
  new Promise((resolve) ⇒
    setTimeout(() ⇒ resolve(`Data from ${server}`), Math.random() * 3000)
  );

Promise.race([
  fetchFromServer('Server A'),
  fetchFromServer('Server B'),
  fetchFromServer('Server C')
]).then((fastestResponse) ⇒ {
  console.log('First response:', fastestResponse);
});
```

```
const googleMaps = () ⇒ new Promise((res) ⇒ setTimeout(() ⇒ res('Google'), 2
const openStreetMap = () ⇒ new Promise((res) ⇒ setTimeout(() ⇒ res('OSM'), 1
const hereMaps = () ⇒ new Promise((res) ⇒ setTimeout(() ⇒ res('HERE'), 300));

Promise.race([googleMaps(), openStreetMap(), hereMaps()])
.then(console.log);
```

promise.any():

```
const tryPayment = (gateway, shouldFail = false) ⇒
  new Promise((resolve, reject) ⇒
    setTimeout(() ⇒ {
      shouldFail ? reject(`${gateway} failed`) : resolve(`${gateway} succeeded`);
    }, Math.random() * 3000)
  );

Promise.any([
  tryPayment('PayPal', true),
  tryPayment('Stripe'),
  tryPayment('Razorpay', true)
]).then((result) ⇒ {
  console.log('Payment success:', result);
}).catch((err) => {
  console.log('All failed:', err);
});
```

```
const tryPayment = (gateway, success = true) =>
new Promise((resolve, reject) =>
setTimeout(() => {
success ? resolve(`${gateway} succeded`) : reject(`${gateway} failed`);
}, Math.random() * 3000)
);

Promise.any([
tryPayment('PayPal'),
tryPayment('Stripe'),
tryPayment('Razorpay', true)
]).then((result) => {
console.log('Payment success:', result);
}).catch((err) => {
console.log('All failed:', err);
});
```

```
const vendor1 = () => Promise.reject('Vendor 1 out of stock');
const vendor2 = () => Promise.resolve('Vendor 2 available');
const vendor3 = () => Promise.reject('Vendor 3 error');

Promise.any([vendor1(), vendor2(), vendor3()])
  .then((res) => console.log('Purchase from:', res))
  .catch((err) => console.error('All failed'));
```

`async` and `await`

async:

is a keyword that you put **before a function** to say it returns a **Promise**.

await

is used **inside async functions** to pause the code until the Promise is resolved or rejected.

example

```
function waitfrsai(){
    return new Promise((resolve)=>{
        setTimeout(()=>{
            resolve(" he came")
        }, 4000)
    })
}
async function gohome(){
    console.log("wait for sai")
    let wait= await waitfrsai()
    console.log(wait)
    console.log("leave together")
}
gohome()
```

Rule	Description
`await` only works inside `async` functions	Using it outside will cause an error
`async` function always returns a Promise	Even if you return a regular value

fetch()

fetch is a built-in JavaScript function used to get data from a server (usually through the internet).

ex: You're sending a message to a waiter (the server) asking for the menu (data), and when they bring it back, you read it.

syntax

```
fetch("https://api.example.com/data")
  .then(response => response.json()) // convert to JSON
  .then(data => console.log(data))   // use the data
  .catch(error => console.log("Error:", error));
```

json

JSON stands for **JavaScript Object Notation**.

It’s a **data format** used to send and receive information — especially between your browser and a server.

{
"name": "Deepthi",
"age": 25,
"city": "Hyderabad"
}

`fetch` + `JSON` Together

```
fetch("https://official-joke-api.appspot.com/random_joke")
.then(response => response.json())
.then(joke => {
console.log(joke.setup);
console.log(joke.punchline);
})
.catch(error => console.log("Failed to get joke", error));
```

now, trying with async and await

```
async function getDadJoke() {
  try {
    const response = await fetch("https://icanhazdadjoke.com/", {
      headers: {
        Accept: "application/json"
      }
    });

    const data = await response.json();
    console.log("Here's a dad joke for you:");
    console.log(data.joke);

  } catch (error) {
    console.log("Oops! Couldn't fetch a joke 😢", error);
  }
}

getDadJoke();
```

Why Use `try...catch`?

Because `await` **can fail**, just like `.then().catch()`.

Without `try...catch`:

```
async function getJoke() {
  const response = await fetch("https://invalid-url.com"); // this will throw
  const data = await response.json();
  console.log(data.joke); // won't be reached if there's an error
}

getJoke(); // This will crash with an error in the console
```

With `try...catch`:

```
async function getJoke() {
  try {
    const response = await fetch("https://icanhazdadjoke.com/", {
      headers: { Accept: "application/json" }
    });

    const data = await response.json();
    console.log("Joke:", data.joke);
  } catch (error) {
    console.log("Something went wrong:", error.message);
  }
}

getJoke();
```

6.Maps and Sets

MAP— Key-Value Storage

A **phonebook** where each person's name (key) maps to their phone number (value).

Features:

- Stores **key-value pairs**
- Keys can be **any type** (even objects or functions)
- Keeps the **order** of insertion
- More flexible than regular objects `{}`

Methods: `Map` provides specific methods for common operations:

- `new Map()`: Creates a new `Map`.

- `map.set(key, value)`: Adds or updates a key-value pair.
- `map.get(key)`: Retrieves the value associated with a key.
- `map.has(key)`: Checks if a key exists.
- `map.delete(key)`: Removes a key-value pair.
- `map.clear()`: Removes all key-value pairs.
- `map.forEach(callback)`: Iterates over each key-value pair.
- `map.keys()`, `map.values()`, `map.entries()`: Return iterators for keys, values, or key-value pairs respectively.

example:

1. E-Commerce Cart System

> Track items in a user's cart with quantity.

```
const cart = new Map();

cart.set("apple", 2);  // 2 apples
cart.set("banana", 1); // 1 banana

// Add another apple
cart.set("apple", cart.get("apple") + 1);

console.log(cart);
```

SET — Unique Values Collection

> A basket that holds only unique items — no duplicates allowed.

Features:

- Stores **only values** (no keys)

- **No duplicates**
- Keeps the **insertion order**
- Useful for filtering out duplicates

Methods: `Set` provides specific methods for operations:

- `new Set()`: Creates a new `Set`.
- `set.add(value)`: Adds a new unique value.
- `set.has(value)`: Checks if a value exists in the `Set`.
- `set.delete(value)`: Removes a specific value.
- `set.clear()`: Removes all values.
- `set.forEach(callback)`: Iterates over each unique value.
- `set.keys()`, `set.values()`, `set.entries()`: Return iterators for values (since keys and values are the same in a `Set`).

Example:

```
const fruits = new Set();

fruits.add("apple");
fruits.add("banana");
fruits.add("apple"); // duplicate, won't be added

console.log(fruits.size); // → 2
console.log(fruits.has("banana")); // → true

fruits.delete("apple");
console.log(fruits); // → Set { 'banana' }
```

It prevents duplicate form submission

```
const submittedUsers = new Set();

function submitForm(userId) {
if (submittedUsers.has(userId)) {
console.log("Already submitted!");
return;
}
submittedUsers.add(userId);
console.log("Form submitted!");
}

submitForm("user123"); // Form submitted!
submitForm("user123"); // Already submitted!
```

sample project

```
const cart = new Map();

const categories = new Set();

function addItem(name, category, price) {
  categories.add(category);

  if (cart.has(name)) {
    const item = cart.get(name);
    item.quantity += 1;
    cart.set(name, item);
  } else {
    cart.set(name, { price: price, quantity: 1 });
  }

  console.log(`${name} added to cart.`);
}
```

```
function viewCart() {
  console.log("🛒 Your Cart:");
  let total = 0;
  let count = 0;

  cart.forEach((item, name) => {
    const subtotal = item.price * item.quantity;
    total += subtotal;
    count += item.quantity;
    console.log(`- ${name}: ₹${item.price} × ${item.quantity} = ₹${subtotal}`);
  });

  console.log(`Total items: ${count}`);
  console.log(`Total price: ₹${total}`);
  console.log(`Categories browsed: ${[...categories].join(", ")}`);
}


addItem("Apple", "Fruits", 30);
addItem("Banana", "Fruits", 10);
addItem("Shampoo", "Toiletries", 100);
addItem("Apple", "Fruits", 30);

viewCart();
```

7.Template Literals

Template literals are a special way of writing **strings** using **backticks** (`` ` ``) instead of regular quotes.

They let you:

1. **Embed variables directly** in the string using `${...}`
2. **Write multi-line strings** easily

3. Use **expressions inside strings**

Examples

1. Embedding Variables

```
let user = "Deepthi";
let message = `Welcome , ${user}!`;
console.log(message);
```

2.Multi-line Strings

```
let poem = `Roses are red,
Violets are blue,
JavaScript is fun,
And so are you!`;

console.log(poem);
```

3. Expressions in Strings

```
let a = 5;
let b = 3;
console.log(`The sum of ${a} and ${b} is ${a + b}.`);
```

→ Output: `The sum of 5 and 3 is 8.`

4. Functions and Template Literals

```
function greet(name) {
  return `Hello, ${name}!`;
}
```

```
console.log(greet("Deepthi"));
```

Using Functions Inside `${}`

```
function greet(name) {
  return `Hi, ${name.toUpperCase()}!`;
}

const message = `Message: ${greet("deepthi")}`;
console.log(message);
```

8.Destructuring

Destructuring is a way to **unpack values** from arrays or objects into **separate variables** in a single line.

> Instead of accessing properties manually, you can "pick out" values directly.

Array Destructuring

Example:

```
const fruits = ["apple", "banana", "cherry"];

const [first, second] = fruits;
console.log(first);  // apple
console.log(second); // banana
```

Object Destructuring

```
const user = {
  name: "Deepthi",
  age: 25,
  city: "Hyderabad"
};

const { name, age } = user;
console.log(name); // Deepthi
console.log(age);  // 25
```

Function Parameters Destructuring

```
function greet({ name, city }) {
  console.log(`Hello, ${name} from ${city}`);
}

const person = { name: "Deepthi", city: "Hyderabad" };
greet(person);
```

Feature	Destructuring Used
Array value unpacking	`[a, b] = arr`
Object property unpacking	`{name, age} = obj`
Renaming	`{name: newName}`
Default values	`{key = default}`
Nested values	`{a: {b}} = obj`
Function parameters	`function({name}) {}`

example: user profile dashboard:

```
const apiResponse = {
  name: "Deepthi",
  email: "deepthi@example.com",
  role: "admin",
  stats: {
    posts: 42,
    followers: 1200,
    following: 300
  },
  location: {
    city: "Hyderabad",
    country: "India",
    pin: 500001
  }
};
const {
  name,
  email,
  stats: { posts, followers },
  location: { city }
} = apiResponse;

console.log(`Welcome, ${name}`);
console.log(`Email: ${email}`);
console.log(`You have ${posts} posts and ${followers} followers.`);
console.log(`You're from ${city}.`);
```

without destructuring

```
const name = apiResponse.name;
const email = apiResponse.email;
const posts = apiResponse.stats.posts;
const city = apiResponse.location.city;
```

example: printing the profile

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title> Profile </title>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 20px;
      background: #f4f4f4;
    }
    .card {
      background: white;
      padding: 20px;
      max-width: 400px;
      border-radius: 8px;
      box-shadow: 0 2px 8px rgba(0,0,0,0.1);
    }
    h2 {
      margin-top: 0;
      color: #333;
    }
    p {
      margin: 6px 0;
    }
  </style>
</head>
<body>

  <div class="card" id="profile">
  </div>

  <script>

    const apiResponse = {
```

```
    name: "Deepthi",
    email: "deepthi@example.com",
    role: "admin",
    stats: {
      posts: 42,
      followers: 1200,
      following: 300
    },
    location: {
      city: "Hyderabad",
      country: "India",
      pin: 500001
    }
  };


  const {
    name,
    email,
    role,
    stats: { posts, followers },
    location: { city, country }
  } = apiResponse;

  const profileDiv = document.getElementById("profile");

  profileDiv.innerHTML = `
    <h2>${name}</h2>
    <p><strong>Email:</strong> ${email}</p>
    <p><strong>Role:</strong> ${role}</p>
    <p><strong>Posts:</strong> ${posts}</p>
    <p><strong>Followers:</strong> ${followers}</p>
    <p><strong>Location:</strong> ${city}, ${country}</p>
  `;
</script>
```

```
</body>
</html>
```

Ex: Printing the profile by getting input from the user

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>User Profile Viewer</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      padding: 20px;
      background: #f4f4f4;
    }

    input {
      padding: 10px;
      font-size: 16px;
      margin: 5px 0;
      display: block;
      width: 300px;
    }

    button {
      padding: 10px 20px;
      font-size: 16px;
      margin-top: 10px;
      cursor: pointer;
    }

    .card {
      background: white;
      padding: 20px;
```

```
     max-width: 400px;
     border-radius: 8px;
     margin-top: 20px;
   }

   h2 {
     margin-top: 0;
     color: #333;
   }

   p {
     margin: 6px 0;
   }
  </style>
</head>
<body>

  <h2> Enter User Profile Info</h2>

  <input type="text" id="name" class="form" placeholder="Name"  />
  <input type="email" id="email" class="form" placeholder="Email" />
  <input type="text" id="role" class="form" placeholder="Role (e.g. admin, user)
  <input type="number" id="posts" class="form" placeholder="Number of Posts
  <input type="number" id="followers" class="form" placeholder="Number of Fo
  <input type="text" id="city" class="form" placeholder="City" />
  <input type="text" id="country" class="form" placeholder="Country" />

  <button onclick="showProfile()">Show Profile</button>

  <div class="card" id="profile">

  </div>

  <script>
   const fields= document.querySelectorAll(".form")
```

```
fields.forEach((field, index)=>{

    field.addEventListener("keydown", function(e){
        if(e.key==='Enter'){

        e.preventDefault()
        const nextid= fields[index+1]
        if(nextid){
            nextid.focus()
        }
        }



    })
})


function showProfile() {

  const name = document.getElementById("name").value.trim();
  const email = document.getElementById("email").value.trim();
  const role = document.getElementById("role").value.trim();
  const posts = document.getElementById("posts").value;
  const followers = document.getElementById("followers").value;
  const city = document.getElementById("city").value.trim();
  const country = document.getElementById("country").value.trim();

  const a = document.getElementById("profile");

  a.innerHTML = `
    <h2>${name || "No Name Entered"}</h2>
    <p><strong>Email:</strong> ${email || "N/A"}</p>
    <p><strong>Role:</strong> ${role || "N/A"}</p>
    <p><strong>Posts:</strong> ${posts || 0}</p>
    <p><strong>Followers:</strong> ${followers || 0}</p>
    <p><strong>Location:</strong> ${city || "Unknown"}, ${country || "Unknov
```

```
      `;
    }
  </script>

</body>
</html>
```

9.For...of loop

`for...of` is a clean, modern loop that lets you iterate over iterable items, like:

- Arrays
- Strings
- Maps, Sets
- NodeLists

Example with Array:

```
const fruits = ["apple", "banana", "mango"];

for (const fruit of fruits) {
  console.log(fruit);
}
```

Example with String:

```
const word = "Deepthi";

for (const char of word) {
  console.log(char);
}
```

Looping Through Input Fields

```
const fields = document.querySelectorAll(".form-field");

for (const field of fields) {
  field.style.border = "1px solid #ccc"; // add styles or listeners
}
```

```
const products = [
{ name: "Laptop", price: 40000 },
{ name: "Phone", price: 20000 },
{ name: "Tablet", price: 15000 },
];

for (const product of products) {
console.log(`${product.name} costs ₹${product.price}`);
}
```

for...in loop

The `for...in` loop is used to **iterate over the enumerable properties (keys) of an object**.

Iterating Over an Object

```
const user = {
  name: "Deepthi",
  age: 25,
  location: "India"
};

for (let key in user) {
  console.log(`${key}: ${user[key]}`);
}
```

Output:

```
name: Deepthi
age: 25
location: India
```

💡 Use it when:

- You're working with **objects**, not arrays.
- You want to iterate through **property names** (keys), not values directly.

Counting the number of keys in an object

```
const data = {
  name: "Product A",
  price: 100,
  stock: true
};

let count = 0;
for (let key in data) {
  count++;
}
console.log("Number of keys:", count);
```

Collect all keys into an array

```
const book = {
  title: "JavaScript Basics",
  author: "John Doe",
  pages: 200
};

const keys = [];
for (let k in book) {
  keys.push(k);
}
console.log(keys);
```

Check if an object has a specific key

```
const car = {
  brand: "Toyota",
  model: "Camry",
  year: 2020
};

const searchKey = "model";
let found = false;

for (let key in car) {
  if (key === searchKey) {
    found = true;
    break;
  }
}
console.log(`Key "${searchKey}" found:`, found);
```

`for...of`	`for...in`
Iterates **values**	Iterates **keys/indexes**

`for...of`	`for...in`
Best for arrays, strings, etc.	Best for objects
Skips properties like `length`	Includes all enumerable propertie

10. Default parameters

Default parameters allow you to **set a fallback value** for a function argument **if it's not provided** when the function is called.

Syntax:

```
function greet(name = "Guest") {
  console.log(`Hello, ${name}!`);
}
```

Example 1: With and Without Argument

```
greet("Deepthi"); // 👉 Hello, Deepthi!
greet();          // 👉 Hello, Guest!
```

- Default parameters fill in missing arguments
- Great for functions with optional inputs
- Keeps your code clean and bug-free

Form Field Defaults

```
function createProfile(name = "User", age = 18, country = "India") {
  return { name, age, country };
}

console.log(createProfile("Deepthi"));
```

Timer or Delay Utility

```
function wait(callback, time = 1000) {
  setTimeout(callback, time);
}
```

11.Rest and Spread operator

The spread operator is used to expand an iterable (like an array or string) into its individual elements, or to expand an object into its key-value pairs.

A. Spread in Arrays:

```
const fruits = ["apple", "banana"];
const moreFruits = [...fruits, "mango"];

console.log(moreFruits); // ["apple", "banana", "mango"]
```

B. Spread in Objects:

```
const user = { name: "Deepthi", role: "admin" };
const userWithLocation = { ...user, city: "Hyderabad" };

console.log(userWithLocation);
```

Clone an object or array without affecting the original

```
const original = { name: "Deepthi", city: "Hyderabad" };
const copy = { ...original };

copy.city = "Bangalore";
console.log(original.city);
```

Merge multiple arrays or objects

```
const frontend = ["HTML", "CSS"];
const backend = ["Node.js", "MongoDB"];
const fullStack = [...frontend, ...backend];

console.log(fullStack);
```

Rest Operator

The rest operator is used to collect multiple elements or properties into a single array or object. It is typically used in function parameters and destructuring assignments.

- **Function Parameters:** It allows a function to accept an indefinite number of arguments and gather them into an array.

```
 function logArguments(first, ...rest) {
     console.log(first);
      console.log(rest);
       }    logArguments(1, 2, 3, 4); //  1,  [2, 3, 4]
```

.object destructuring

```
const { name, ...details } = {
  name: "Deepthi",
  role: "admin",
  city: "Hyderabad"
};
console.log(name);    // "Deepthi"
console.log(details); // { role: "admin", city: "Hyderabad" }
```

Flexible product pricing function

```
function calculateTotal(product, ...prices) {
  const total = prices.reduce((sum, p) => sum + p, 0);
  return `${product} total: ₹${total}`;
}

console.log(calculateTotal("Laptop", 50000, 2000, 1500));
```

12.modules

JavaScript modules let you **split your code into multiple files**, each with its own scope. You can then **import** or **export** parts as needed.

Problem Without Modules	Solution With Modules
All code in one file = mess	Split by feature or purpose
Variable conflicts	Local scope per file
No reuse between projects	Easily import/export
Hard to test or scale	Cleaner architecture