Internet Programming IAE-1 QB Answers

Chapter 1 Introduction

1) Discuss HTTP and HTTPs protocol in detail with respect to Web Development.

HTTP and HTTPS Protocols in Web Development

1. Introduction

- **HTTP (HyperText Transfer Protocol)** → A set of rules used by browsers and servers to communicate and transfer web content (like HTML pages, images, videos).
- **HTTPS (HyperText Transfer Protocol Secure)** → The secure version of HTTP, which adds **encryption (SSL/TLS)** to protect the data being exchanged.

---

2. How HTTP Works

- **Client–Server Model:**
  - The **client** (e.g., browser) sends a request → "Get me this web page."
  - The **server** responds with the requested resource (HTML, CSS, JS, etc.).
- Works on **port 80** by default.
- Communication is **plain text** → data is not encrypted.

Key Features of HTTP:

- Stateless → Server doesn't remember past requests unless we use cookies/sessions.
- Faster, but **less secure**.
- Widely used for websites where security is not critical (e.g., blogs, tutorials).

---

3. How HTTPS Works

- Builds on HTTP, but adds **SSL/TLS encryption**.
- Works on **port 443**.
- Before data transfer:
  1. Browser and server perform a **handshake** to establish trust.
  2. A **digital certificate** (issued by Certificate Authorities like DigiCert, Let's Encrypt) is used to prove authenticity.
  3. All communication is encrypted.

Key Features of HTTPS:

- **Confidentiality** → Data is encrypted (hackers cannot read).
- **Integrity** → Data is not altered during transmission.
- **Authentication** → Ensures you are really talking to the correct server (no impersonation).

Importance in Web Development

- **User Trust:** Modern users look for the "lock symbol" before entering sensitive data.
- **Search Engine Optimization (SEO):** Google ranks HTTPS sites higher than HTTP.
- **Data Protection:** Essential for sites dealing with payments, personal info, or confidential content.

Real-World Example

- **HTTP Site:** A portfolio site showing only static content (no sensitive user data).
- **HTTPS Site:** Amazon, Flipkart, or Netbanking websites where user passwords, credit card numbers, and personal info must remain secure.

2) Explain DNS and its Working in detail.

DNS (Domain Name System) and Its Working

1. Introduction

- **DNS (Domain Name System)** is like the **phonebook of the Internet**.
- DNS translates **domain names → IP addresses**, allowing users to access websites using simple names instead of numbers.

👉 Example: When you type www.facebook.com, DNS resolves it into the IP address of Facebook's server.

---

2. Why DNS is Needed

- Without DNS, we would need to remember **long strings of numbers** (IP addresses).
- DNS makes the internet **user-friendly**, just like how your **contacts app** lets you search names instead of dialing numbers.

---

3. How DNS Works:

Let's say you type **www.example.com** into your browser:

1. **Browser Cache:** Checks if it already knows the IP from a previous visit.
2. **OS DNS Cache:** If browser doesn't know, your operating system checks its own cache.
3. **DNS Resolver (ISP):** If still unknown, request goes to your ISP's DNS resolver to find the IP.
4. **Root DNS Server:** Resolver asks the root server where to find info for the domain (.com, .net, etc.).
5. **TLD Server (.com):** Root directs to the TLD server, which gives the address of the authoritative server.
6. **Authoritative DNS Server:** Resolver asks this server for the actual IP of the domain.
7. **Response to Browser:** IP is returned to your browser, which then connects to the website.

3) Explain Json and XML Protocol

1. Introduction:

In **Web Development**, data often needs to be **exchanged between client and server** (e.g., from browser → web server or between APIs).

- **JSON (JavaScript Object Notation)** and **XML (eXtensible Markup Language)** are two widely used formats for **structuring and transferring data**.
- Both act as **protocols for data exchange**, but they differ in **syntax, readability, and usage**.

---

2. JSON (JavaScript Object Notation)

Definition

- Lightweight, text-based data format used to represent structured data.
- Based on **key-value pairs**.
- Designed to be **human-readable and machine-friendly**.

Features of JSON

- **Data Types:** Supports strings, numbers, arrays, booleans, and objects.
- **Language Independent:** Although derived from JavaScript, supported in almost every programming language.
- **Used in APIs:** Commonly used for RESTful APIs, AJAX calls, and web apps.

Example (JSON Data):

```
{
 "student": {
  "name": "Rahul",
  "age": 21,
  "subjects": ["Math", "Java", "Web Development"]
 }
}
```

3. XML (eXtensible Markup Language)

Definition

- Markup language that defines rules for representing structured data in a **hierarchical (tree) format**.
- Similar to HTML, but used for **data storage and transfer**, not presentation.

Features of XML

- **Self-Descriptive:** Tags define the meaning of data.
- **Hierarchy:** Data stored in nested elements (parent-child structure).
- **Extensible:** Developers can define their own tags.
- **Supports Metadata:** Attributes can describe additional details about data.
- Used in **SOAP web services**, config files, and older APIs.

Example (XML Data)

```
<student>
 <name>Rahul</name>
 <age>21</age>
 <subjects>
  <subject>Math</subject>
  <subject>Java</subject>
  <subject>Web Development</subject>
 </subjects>
</student>
```

4. Applications in Web Development

- **JSON:**
    - Modern web APIs (Google Maps, Twitter API, etc.).
    - Used in JavaScript-based frameworks (React, Angular).
    - Storing configuration in Node.js (package.json).

- **XML:**
    - Used in **SOAP-based services** (older web service standard).
    - Storing structured data (e.g., Android app config files).

4) Explain Dom with respect to Web development.

1. Introduction

- **DOM (Document Object Model)** is a **programming interface** provided by browsers that represents a web page as a **tree-like structure**.
- Every element in an HTML document (headings, paragraphs, links, images, etc.) is represented as an **object (node)** in this tree.
- Using DOM, developers can **access, modify, add, or delete elements** dynamically with the help of **JavaScript**.

Example: When you click a button and a message appears without reloading the page — that's DOM manipulation in action.

---

2. Structure of DOM

DOM represents the web page as a **hierarchical tree** of nodes:

1. **Document Node** → Root of the DOM tree (represents the whole HTML document).
2. **Element Nodes** → HTML tags (e.g., <h1>, <p>,).
3. **Attribute Nodes** → Attributes of elements (e.g., id="title").
4. **Text Nodes** → The actual text inside elements.

Example HTML:

```
<html>
 <body>
  <h1 id="heading">Hello World</h1>
  <p>Welcome to DOM</p>
 </body>
</html>
```

3. DOM Operations in Web Development

With JavaScript, we can perform **CRUD operations** on DOM:

a) Accessing Elements

- document.getElementById("heading") → Selects element by ID.
- document.getElementsByTagName("p") → Selects by tag.

- document.querySelector(".className") → Selects by CSS selector.

b) **Changing Content**
document.getElementById("heading").innerHTML = "Welcome!";
Changes <h1>Hello World</h1> to <h1>Welcome!</h1>.

c) **Changing Style**
document.getElementById("heading").style.color = "blue";
Updates CSS dynamically

4. Importance of DOM in Web Development
- **Dynamic Pages:** Enables changing page content without reloading.
- **Interactivity:** Essential for modern, interactive applications (e.g., form validation, animations).
- **JavaScript Integration:** DOM is the bridge between HTML/CSS and JavaScript.
- **Frameworks:** Libraries like **React, Angular, Vue** are built around efficient DOM manipulation (Virtual DOM).

5. Real-World Example
- On an **e-commerce site**, when you add an item to the cart:
    - The number in the cart icon increases dynamically.
    - This happens through **DOM manipulation** — JavaScript updates the cart count element in real-time.

5) Explain URL and URL in detail.

1. Introduction
- **URI (Uniform Resource Identifier):** A generic identifier used to uniquely identify a resource on the internet.
- **URL (Uniform Resource Locator):** A type of URI that not only identifies a resource but also provides its **location (address)** and how to access it.

Example:
- URI: urn:isbn:0451450523 (identifies a book by ISBN, but doesn't say where to find it).
- URL: https://www.example.com/index.html (gives the exact location of a web page).

2. What is URL?

Definition

- A **Uniform Resource Locator** specifies the **address of a resource** and the **protocol to access it**.
- It is what we usually type in a browser's address bar.

Structure of a URL

https://www.example.com:443/path/page.html?search=web#section1

Breakdown:

1. **Scheme/Protocol** → https (method of communication: HTTP, HTTPS, FTP, etc.).
2. **Domain Name (Host)** → www.example.com (server's address).
3. **Port (optional)** → 443 (default for HTTPS).
4. **Path** → /path/page.html (location of file on the server).
5. **Query String (optional)** → ?search=web (parameters passed to the server).
6. **Fragment (optional)** → #section1 (specific part of the page).

Example URLs

- https://www.google.com/ → Loads Google's homepage

3. What is URI?

Definition

- A **Uniform Resource Identifier** is a broader term that identifies a resource by **name, location, or both**.
- A URL is a **subset of URI**.

Types of URI

1. **URL (Locator)** → Tells *where* the resource is.
   Example: https://example.com/home.html

2. **URN (Name)** → Tells *what* the resource is, without saying where.
   Example: urn:isbn:0451450523 (book identifier).

Chapter 2 Java Script

1) Discuss the role of Java Script in the front end web development.

1. Introduction

- **JavaScript** is a **client-side scripting language** mainly used in **front-end web development**.
- While **HTML** structures the webpage and **CSS** styles it, **JavaScript makes it interactive and dynamic**.
- It runs directly in the user's browser, reducing server load and improving user experience.

---

2. Key Roles of JavaScript in Front-End

1. DOM Manipulation

- DOM = Document Object Model (tree structure of a web page).
- JavaScript can **add, remove, or modify elements** dynamically.
- Example: Changing text on a button click without reloading.
    - **document.getElementById("title").innerHTML = "Welcome User!";**

2. Event Handling

- JavaScript reacts to user actions: clicks, keypress, mouseover, etc.
- Example: Showing an alert when a button is clicked.
    - **document.getElementById("btn").onclick = () => alert("Button clicked!");**

3. Form Validation

- Ensures correct input before sending data to the server.
- Example: Checking if an email field is empty.
    - **if(email.value === "") alert("Email is required!");**

4. Creating Interactivity & Animations

- Enables pop-ups, image sliders, dropdown menus, and smooth transitions.
- Makes static pages feel **responsive and engaging**.

5. Asynchronous Data Loading (AJAX & Fetch API)

- JavaScript can **fetch data from the server** without reloading the page.
- Example: Loading new tweets on Twitter dynamically.
    - **fetch("data.json").then(res => res.json()).then(data => console.log(data));**

6. Integration with Frameworks/Libraries

- Modern frameworks like **React, Angular, Vue** rely on JavaScript.
- They allow developers to build **Single Page Applications (SPA)** where only parts of the page update, not the whole.

Real-World Examples

- **Google Maps:** Zooming and dragging without reloading.
- **E-commerce Sites:** Cart updates instantly when adding items.
- **Gmail:** Loads new emails dynamically.

2) Discuss various Java Script Features in details.

Key Features of JavaScript

1. **Lightweight & Interpreted Language**
   - No compilation required; code runs directly in the browser.
   - Makes development and debugging faster.

2. **Cross-Platform Compatibility**
   - Runs in all modern browsers (Chrome, Firefox, Edge, Safari).
   - Write once, run anywhere → Ensures wide usability.

3. **Dynamic Typing**
   - No need to specify variable types explicitly.
   - Example: **let x = 10; // number**
     x = "Hello"; // now string

4. **Object-Oriented (Prototype-Based)**
   - Supports objects, inheritance, and encapsulation.
   - Uses **prototypes** instead of classical classes (before ES6).
   - Example: **let car = { brand: "Toyota", model: "Camry" };**

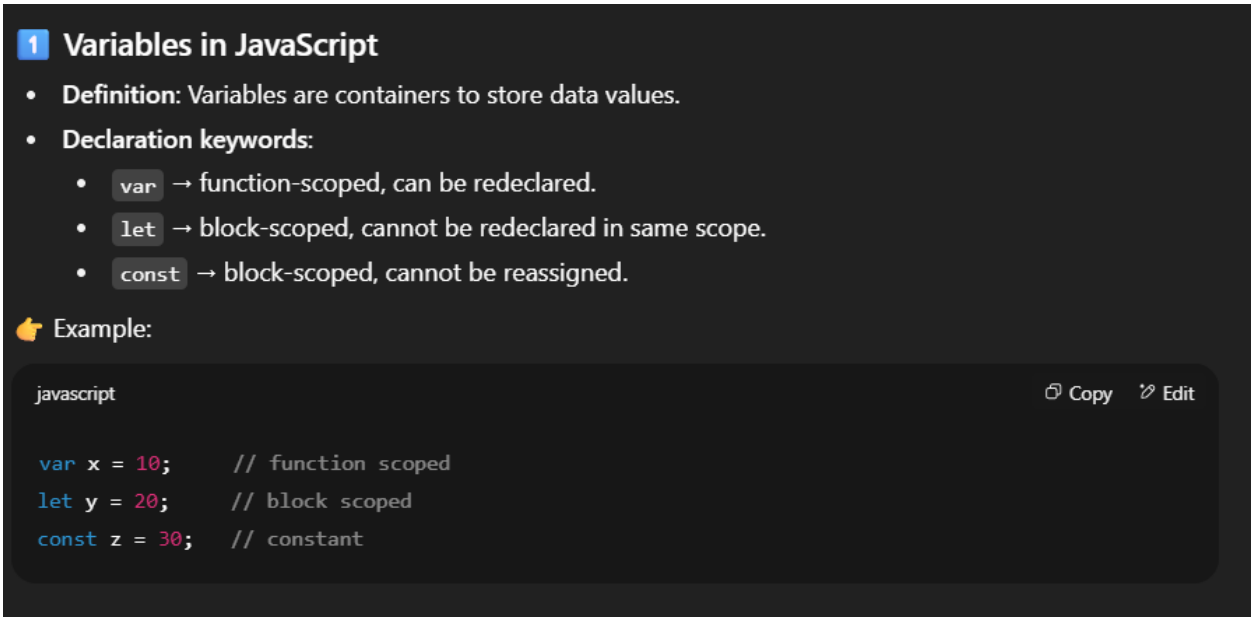
5. **Event-Driven**
   - Executes code when an event (click, keypress, hover) occurs.
   - Example: **document.getElementById("btn").addEventListener("click", () => alert("Clicked!"));**

6. **Asynchronous Programming**
   - JavaScript can handle tasks without blocking the main thread.
   - Uses **callbacks, promises, async/await**.
   - Example (Promise):
     - **fetch("data.json")**
     - **.then(res => res.json())**
     - **.then(data => console.log(data));**

7. **Client-Side Validation**
   - Ensures correct input before sending to server.
   - Example: Prevent empty forms from submission.

8. **Huge Ecosystem**
   - Rich libraries (jQuery) and frameworks (React, Angular, Vue).
   - Node.js allows JavaScript to run on the server side.

4) Discuss variables, data types , control structure and loop in Java Script with example



2 Data Types in JavaScript

JavaScript is **dynamically typed** → variable type is decided at runtime.

Primitive Types

1. **Number** → `let a = 42;`
2. **String** → `let name = "John";`
3. **Boolean** → `let isReady = true;`
4. **Undefined** → `let val;`
5. **Null** → `let data = null;`
6. **Symbol** (ES6) → unique value, `let sym = Symbol("id");`
7. **BigInt** (ES11) → for very large numbers, `let big = 123n;`

Non-Primitive Type

- **Object** → collection of key-value pairs.

  👉 Example:

javascript Copy Edit

```javascript
let person = { name: "Alice", age: 22 };
```

3 Control Structures in JavaScript

Used to control the flow of execution.

Conditional Statements

- `if`, `else if`, `else`

javascript Copy Edit

```javascript
let marks = 85;
if (marks > 90) {
  console.log("Grade A");
} else if (marks > 75) {
  console.log("Grade B");
} else {
  console.log("Grade C");
}
```

- `switch`

javascript Copy Edit

```javascript
let day = 2;
switch(day){
  case 1: console.log("Monday"); break;
  case 2: console.log("Tuesday"); break;
  default: console.log("Other day");
}
```

Loops in JavaScript

Loops are used for **repeated execution**.

for loop

javascript Copy Edit

```javascript
for(let i=1; i<=5; i++){
  console.log(i);
}
```

while loop

javascript Copy Edit

```javascript
let i = 1;
while(i <= 5){
  console.log(i);
  i++;
}
```

do...while loop

javascript Copy Edit

```javascript
let j = 1;
do {
  console.log(j);
  j++;
} while(j <= 5);
```

for...in loop (iterate over object keys)

javascript Copy Edit

```javascript
let person = {name: "John", age: 25};
for(let key in person){
  console.log(key, person[key]);
}
```

for...of loop (iterate over arrays/iterables)

javascript Copy Edit

```javascript
let arr = [10, 20, 30];
for(let value of arr){
  console.log(value);
}
```

3) Differentiate between ES5 and ES6 in detail with examples

📊 Difference Between ES5 and ES6

Feature	ES5 (ECMAScript 5)	ES6 (ECMAScript 2015)
Year of Release	2009	2015
Variable Declaration	Uses `var` (function-scoped, can be redeclared). 👉 Example: `var x = 10; var x = 20;`	Introduces `let` (block-scoped) and `const` (block-scoped, constant). 👉 Example: `let x = 10; const y = 20;`
Functions	Function expressions and declarations only. 👉 Example: `function add(a,b){ return a+b; }`	Introduces **Arrow Functions** with shorter syntax. 👉 Example: `let add = (a,b) => a+b;`
Classes	No direct support, used function constructors and prototypes. 👉 Example: `function Car(){this.brand="BMW";}`	Direct support for **classes**. 👉 Example: `class Car { constructor(){ this.brand="BMW"; } }`

Modules	Not supported directly. Needed external libraries.	Built-in **import/export** support.
		👉 Example:
		`export default function(){}`
		`import func from './file.js';`

Default Parameters	Not supported. 👉 Example: `function greet(name){ name = name`	
Template Literals	String concatenation done with `+`. 👉 Example: `"Hello " + name`	Uses backticks (`` ` ``) with placeholders `${}`. 👉 Example: `` `Hello ${name}` ``
Iteration	Uses traditional `for`, `for-in`.	Introduces `for-of` loop for arrays. 👉 Example: `for(let val of arr){ console.log(val); }`
Arrow Functions	Not available.	Introduced for concise function writing.
Promises	Not available; callbacks used for async code.	Introduced **Promises** for async programming. 👉 Example: `new Promise((resolve, reject)=>{ ... })`
Block Scope	Only function scope (`var`).	Block scope introduced with `let` and `const`.
Inheritance	Done via prototypes.	Done using `class` and `extends`.

5) Discuss function and arrow function in Java Script with example each.

1 Function in JavaScript

Definition

- A **function** is a block of code designed to perform a task.
- It allows **reusability** and makes programs more **modular and organized**.

Syntax

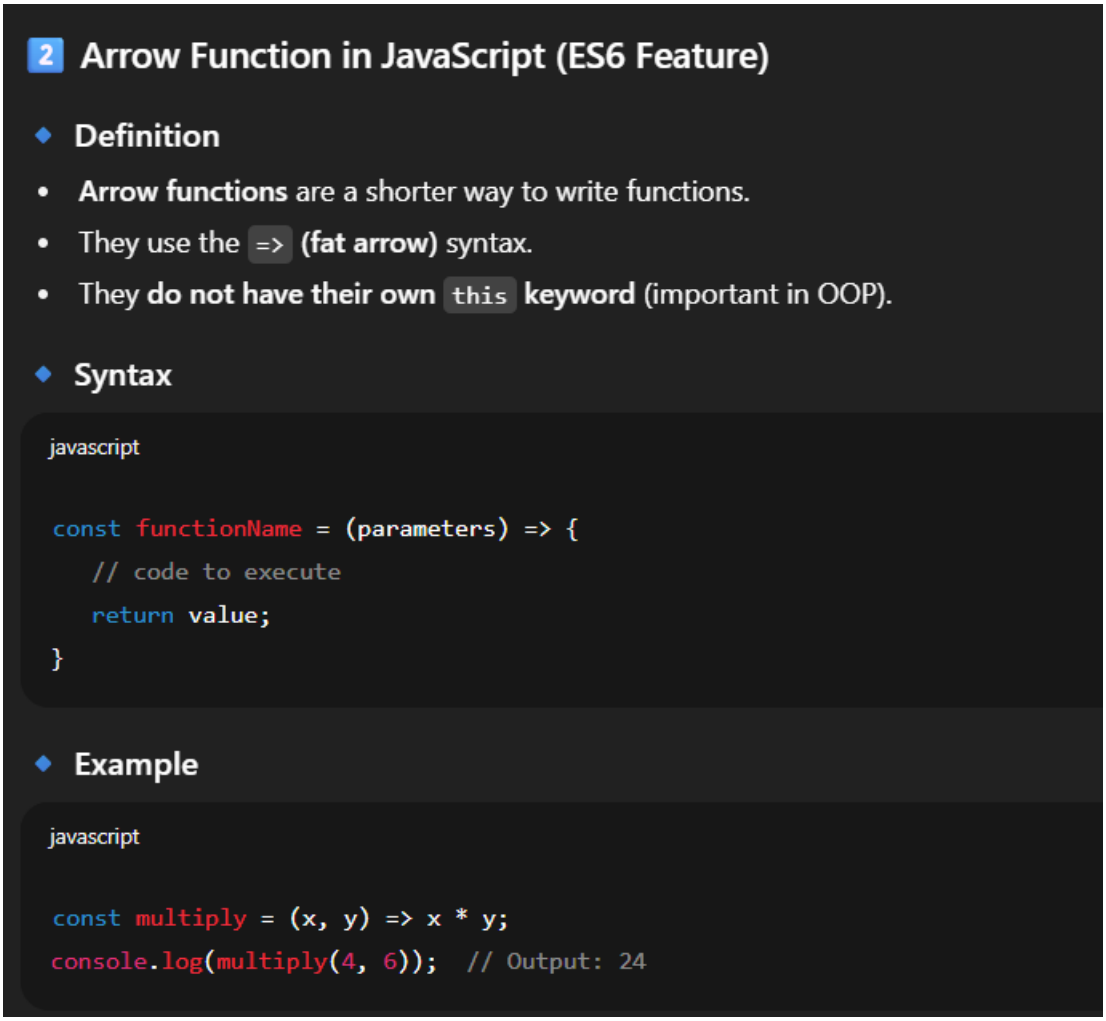
javascript

```javascript
function functionName(parameters) {
   // code to execute
   return value;
}
```

Example

javascript

```javascript
function add(a, b) {
  return a + b;
}
console.log(add(5, 3));   // Output: 8
```



6) Discuss Prototype Programming or Object Based Java Script programming in Java Script with examples.

Introduction

- JavaScript is an **object-based language**, not purely object-oriented.
- Instead of **classes (before ES6)**, JavaScript used **prototypes** to create and inherit objects.
- Every object in JavaScript has an internal link to another object called its **prototype**.

What is a Prototype?

- A **prototype** is like a **blueprint** for objects.
- It allows **object inheritance** → one object can access properties/methods of another.
- Think of it like a **family tree**: if a child doesn't know something, it asks the parent (prototype).

3 Object Creation using Prototype

🔹 Example 1: Using Object Literals

javascript

```javascript
let student = {
  name: "Alice",
  greet: function() {
    console.log("Hello, my name is " + this.name);
  }
};

student.greet();  // Output: Hello, my name is Alice
```

👉 Here, `student` is an object with properties and methods.

🔹 Example 2: Constructor Function with Prototype

javascript Copy

```javascript
function Person(name, age) {
  this.name = name;
  this.age = age;
}

// Adding method to prototype
Person.prototype.sayHello = function() {
  console.log("Hi, I'm " + this.name + " and I'm " + this.age + " years old.");
};

let p1 = new Person("John", 25);
let p2 = new Person("Sara", 30);

p1.sayHello();  // Output: Hi, I'm John and I'm 25 years old.
p2.sayHello();  // Output: Hi, I'm Sara and I'm 30 years old.
```

👉 Key Points:

- `Person` is a **constructor function**.
- `Person.prototype.sayHello` ensures the method is **shared** by all objects created from `Person`.

🔹 Example 3: Prototype Chain (Inheritance)

javascript

```javascript
let animal = {
  eats: true
};

let dog = Object.create(animal);  // dog inherits from animal
dog.barks = true;

console.log(dog.eats);   // true (inherited from prototype)
console.log(dog.barks);  // true (own property)
```

👉 Here, `dog` inherits property `eats` from `animal`.

Why Prototype Programming is Important?

- Memory-efficient → methods are stored in prototype, not copied in every object.
- Enables **inheritance** in JavaScript.
- Forms the **foundation of OOP in JavaScript** (before class syntax in ES6).

7) Discuss in details OOP programming in Java Script.

1️⃣ Introduction

- **OOP (Object-Oriented Programming)** is a programming paradigm based on the concept of **objects**.
- Objects combine **data (properties)** and **behaviors (methods)** into a single unit.
- JavaScript initially used **prototypes** for OOP, but with **ES6 (2015)**, the class keyword was introduced to make OOP more familiar.

2 Core OOP Concepts in JavaScript

(i) Class and Object

- **Class**: A blueprint for creating objects.
- **Object**: An instance of a class.

javascript

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

  display() {
    console.log(`Name: ${this.name}, Age: ${this.age}`);
  }
}

let s1 = new Student("Alice", 20);
s1.display(); // Output: Name: Alice, Age: 20
```

(ii) Encapsulation

- Wrapping **data** and **methods** into a single unit.
- In JavaScript, we can use **private fields (#)**.

javascript

```
class Account {
  #balance = 0;  // private property

  deposit(amount) {
    this.#balance += amount;
    console.log("Balance:", this.#balance);
  }
}

let acc = new Account();
acc.deposit(100); // Balance: 100
// acc.#balance ❌ (not accessible outside)
```

🔹 (iii) Inheritance

- Allows one class (child) to use properties/methods of another class (parent).

javascript

```javascript
class Person {
  constructor(name) {
    this.name = name;
  }
  greet() {
    console.log(`Hello, I'm ${this.name}`);
  }
}

class Teacher extends Person {
  constructor(name, subject) {
    super(name);  // call parent constructor
    this.subject = subject;
  }
  teach() {
    console.log(`${this.name} teaches ${this.subject}`);
  }
}

let t1 = new Teacher("John", "Math");
t1.greet();   // Hello, I'm John
t1.teach();   // John teaches Math
```

(iv) Polymorphism

- Same method name but behaves differently in different classes.

javascript

```javascript
class Animal {
  sound() {
    console.log("Some generic sound");
  }
}

class Dog extends Animal {
  sound() {
    console.log("Bark!");
  }
}

let a = new Animal();
let d = new Dog();

a.sound(); // Some generic sound
d.sound(); // Bark!
```

(v) Abstraction

- Hiding unnecessary details and showing only essential features.
- JavaScript doesn't have `abstract` classes directly, but we can simulate abstraction using **base classes** and methods that must be overridden.

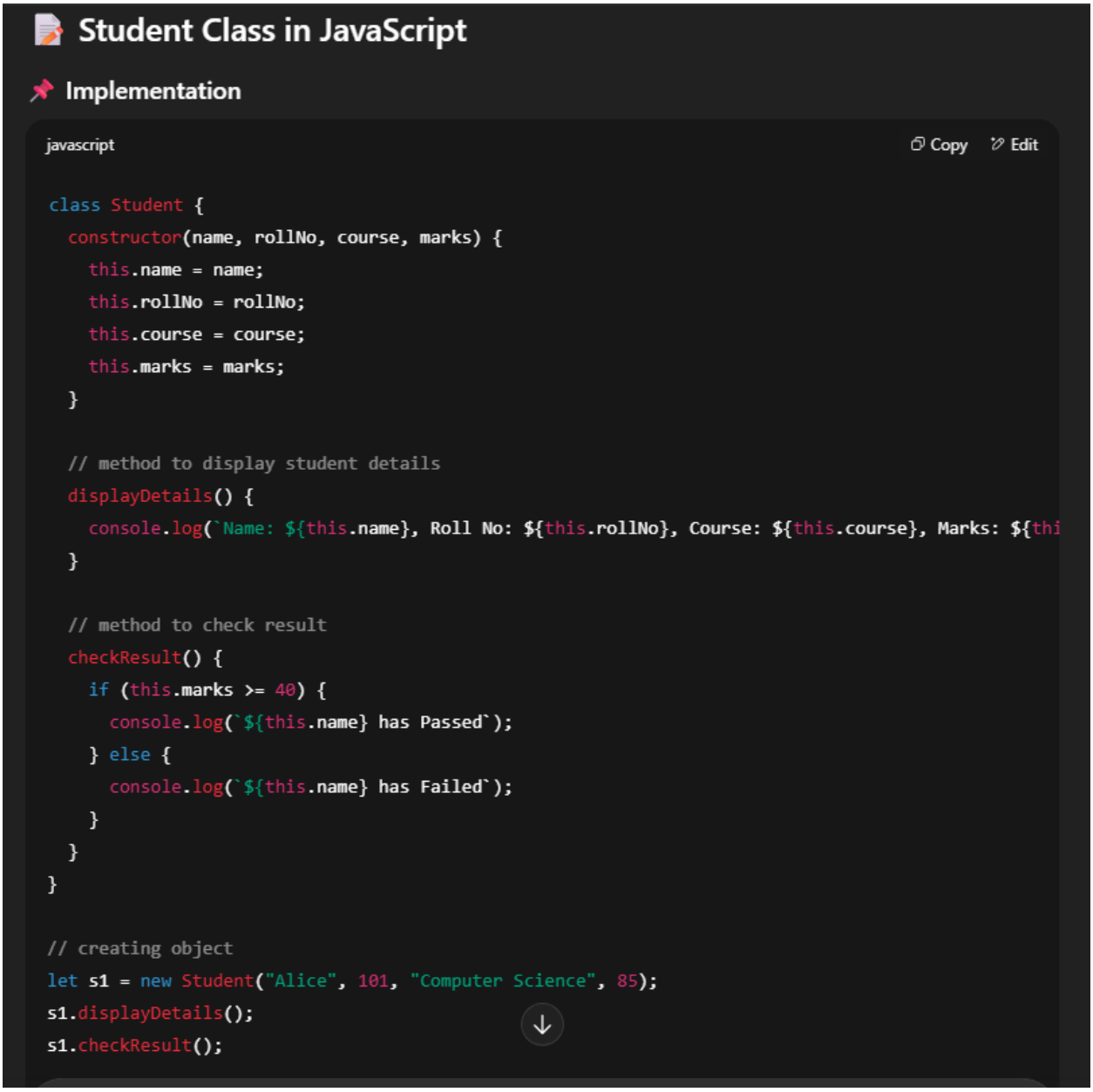
javascript Copy Edit

```javascript
class Shape {
  area() {
    throw "Area method must be implemented";
  }
}

class Circle extends Shape {
  constructor(radius) {
    super();
    this.radius = radius;
  }
  area() {
    return Math.PI * this.radius * this.radius;
  }
}

let c = new Circle(5);
console.log(c.area()); // 78.5
```

Advantages of OOP in JavaScript

- **Reusability** → Inheritance reduces code repetition.
- **Organization** → Code is modular and structured.
- **Scalability** → Easy to expand projects.
- **Data Security** → Encapsulation protects data.

8) Implement student's class in Java Script and explain in details.



📌 Explanation

1. **Class Definition**
   - `class Student {}` creates a blueprint for student objects.
2. **Constructor**
   - `constructor(name, rollNo, course, marks)` initializes the properties when an object is created.
3. **Properties**
   - `name`, `rollNo`, `course`, `marks` → represent student's information.
4. **Methods**
   - `displayDetails()` → prints student details.
   - `checkResult()` → checks if the student has passed (marks ≥ 40) or failed.
5. **Object Creation**
   - `new Student("Alice", 101, "Computer Science", 85)` creates an object with given values.
6. **Output**

yaml Copy Edit

```
Name: Alice, Roll No: 101, Course: Computer Science, Marks: 85
Alice has Passed
```

9) Implement inheritance in JS with Fees class.

📌 **Explanation**

1. **Parent Class –** `Student`
   - Holds basic student details (`name` , `rollNo`).
   - Method: `displayStudent()` prints student info.
2. **Child Class –** `Fees`
   - Extends the `Student` class using `extends` .
   - `super(name, rollNo)` → calls parent constructor to reuse initialization.
   - Adds new property `amount` .
   - Method: `displayFees()` shows the fees.
3. **Object Creation**
   - `new Fees("Alice", 101, 5000)` creates a Student with fee info.
4. **Output**

```yaml
Name: Alice, Roll No: 101
Fees Paid: 5000
```

10) Explain Asynchronous Java Script Programming.

📌 **Definition**

- **Asynchronous programming** means executing tasks **without blocking** the main thread.
- In JavaScript, code runs **line by line** (single-threaded), but async allows tasks like fetching data, timers, or file reading to run **in the background** while other code continues.

📌 **Why Needed?**

- If JavaScript were only synchronous, tasks like fetching data from a server would **freeze the browser** until completed.
- Asynchronous avoids this by **not waiting**; it continues executing other code and handles results later.

📌 **How It Works**

1. **Event Loop** – Keeps checking if async tasks are complete and executes their callback when ready.
2. **Callbacks / Promises / async-await** – Used to manage async operations.

📌 Example

javascript

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

setTimeout(() => {
  console.log("Async Task Done");
}, 2000);

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

Output

pgsql

```
Start
End
Async Task Done   // runs after 2 seconds, without blocking
```

11) Discuss callback with suitable example

📌 **Definition**

- A **callback** is a function that is **passed as an argument** to another function and is executed **after the completion of that function**.
- It is a way to handle **asynchronous tasks** in JavaScript.

📌 **Why Callbacks?**

- JavaScript is **non-blocking**. For tasks like API calls or timers, we don't know when they will finish.
- Instead of waiting, we use a **callback** that runs when the task is done.

📌 Example

javascript

```javascript
function fetchData(callback) {
  console.log("Fetching data...");

  setTimeout(() => {
    console.log("Data received!");
    callback();   // execute the callback after data is fetched
  }, 2000);
}

function processData() {
  console.log("Processing data...");
}

fetchData(processData);
```

Output

kotlin

```
Fetching data...
Data received!
Processing data...
```

📌 Explanation

1. fetchData takes a **callback function** as an argument.
2. setTimeout simulates data fetching (2 seconds delay).
3. Once done, it calls callback() (which is processData).
4. This ensures **orderly execution** of async tasks.

12) Discuss Promise with suitable examples

📌 Definition

- A **Promise** is an object in JavaScript that represents the **eventual completion (or failure)** of an asynchronous operation and its result.
- It helps to avoid **callback hell** and makes async code easier to manage.

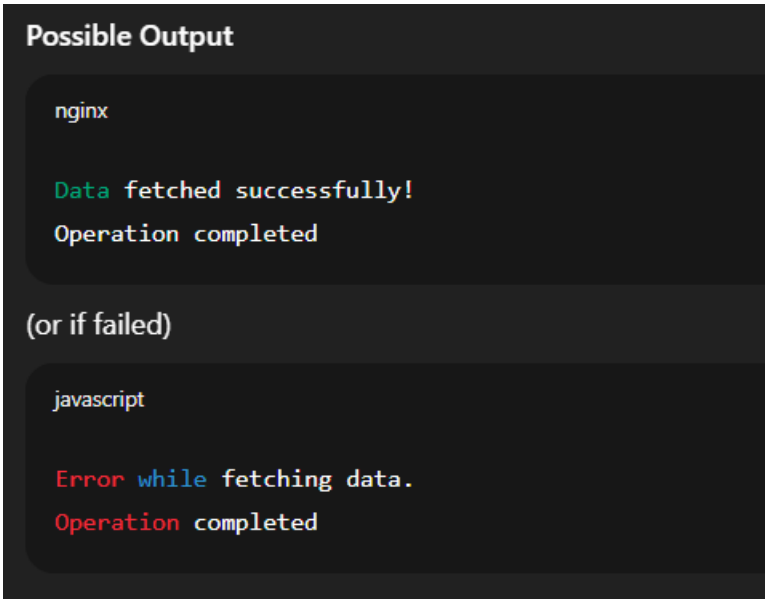
📌 States of a Promise

1. **Pending** → Initial state (operation not completed yet).
2. **Fulfilled** → Operation completed successfully.
3. **Rejected** → Operation failed.

📌 Example

javascript

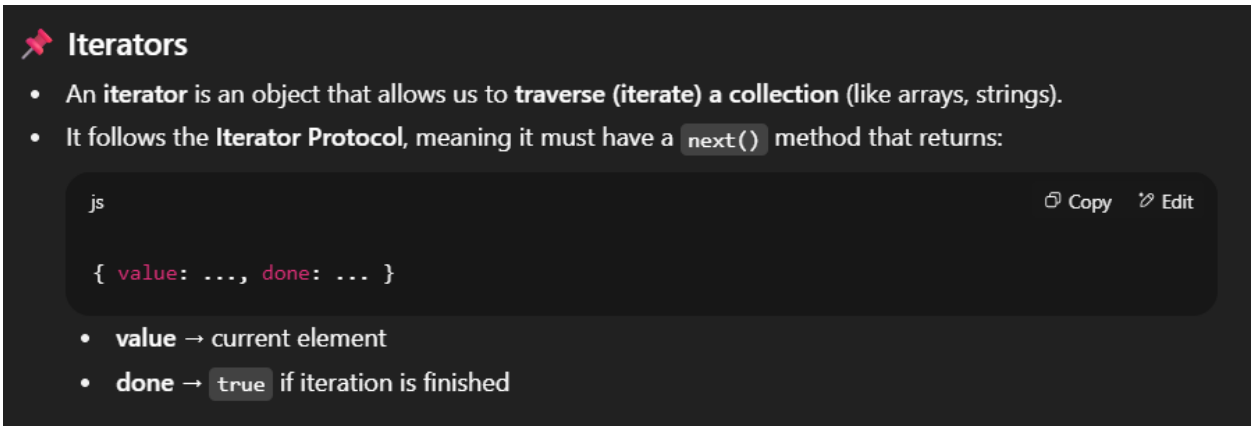
```javascript
// Creating a Promise
let myPromise = new Promise((resolve, reject) => {
  let success = true;

  setTimeout(() => {
    if (success) {
      resolve("Data fetched successfully!");
    } else {
      reject("Error while fetching data.");
    }
  }, 2000);
});

// Consuming a Promise
myPromise
  .then(result => console.log(result))   // handles success
  .catch(error => console.log(error))   // handles failure
  .finally(() => console.log("Operation completed"));
```



Explanation

- new Promise() takes two functions: **resolve** (success) and **reject** (failure).
- then() is used when the promise is resolved.
- catch() is used when the promise is rejected.
- finally() runs regardless of success or failure.

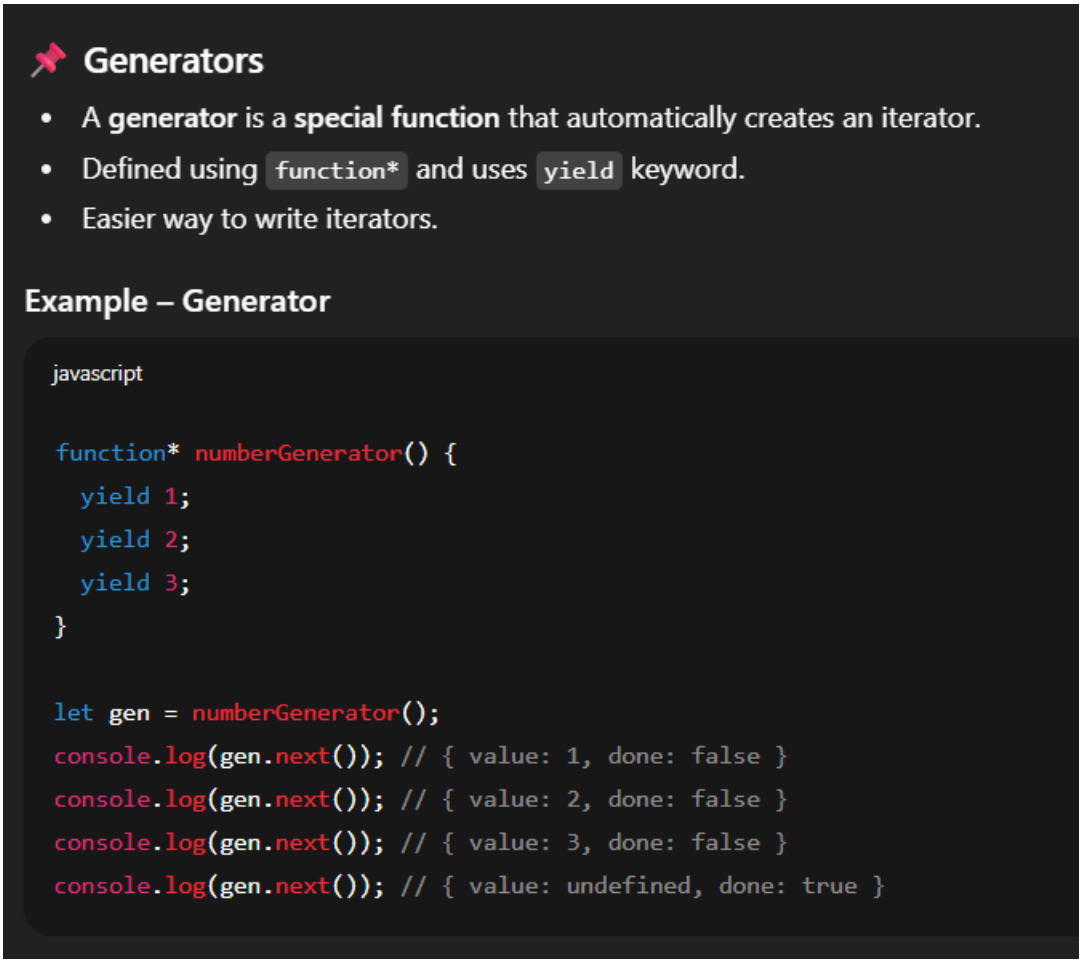
13) Explain Iterators and generator in JS with example



Example – Iterator

javascript

```javascript
function createIterator(array) {
  let index = 0;
  return {
    next: function () {
      if (index < array.length) {
        return { value: array[index++], done: false };
      } else {
        return { value: undefined, done: true };
      }
    }
  };
}

let iterator = createIterator([10, 20, 30]);
console.log(iterator.next()); // { value: 10, done: false }
console.log(iterator.next()); // { value: 20, done: false }
console.log(iterator.next()); // { value: 30, done: false }
console.log(iterator.next()); // { value: undefined, done: true }
```



14) Discuss client server with fetch in Js.

📌 **What is Fetch?**

- fetch() is a modern JavaScript API used to make **HTTP requests** from the **client (browser)** to the **server**.
- It is used for communication between **frontend (client)** and **backend (server)**.
- Returns a **Promise** → making it asynchronous and easy to handle.

📌 **How Client–Server Works with Fetch**

1. **Client (Browser)** sends a request using fetch() (GET, POST, etc.).
2. **Server** (like Node.js, PHP, Java backend, etc.) receives the request and processes it.
3. **Server** sends back a **response** (data in JSON, HTML, XML, etc.).
4. **Client** handles the response (update UI, store data, etc.).

📌 Example – GET Request

javascript

```javascript
// Client sending request to server
fetch("https://jsonplaceholder.typicode.com/posts/1")
  .then(response => response.json())   // convert response to JSON
  .then(data => console.log("Data from server:", data))
  .catch(error => console.log("Error:", error));
```

Output (Sample)

css

```
Data from server: { userId: 1, id: 1, title: "...", body: "..." }
```

📌 Example – POST Request

javascript

```javascript
// Sending data from client to server
fetch("https://jsonplaceholder.typicode.com/posts", {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({ title: "Exam Prep", body: "Using fetch()", userId: 1 })
})
.then(response => response.json())
.then(data => console.log("Server response:", data))
.catch(error => console.log("Error:", error));
```

Chapter 3 React Basics

1) Discuss the React with respect to front end development.

What is React?

- **React** is a popular **JavaScript library** (developed by Facebook) for building **user interfaces (UIs)**, especially **single-page applications (SPAs)**.
- It helps developers create **reusable UI components** that update efficiently when data changes.

Why React for Front-End?

- Traditional front-end (HTML + JS + jQuery) becomes **complex** with dynamic updates.
- React provides a **component-based** and **declarative** approach, making development **faster, scalable, and maintainable**.

Key Features of React in Front-End

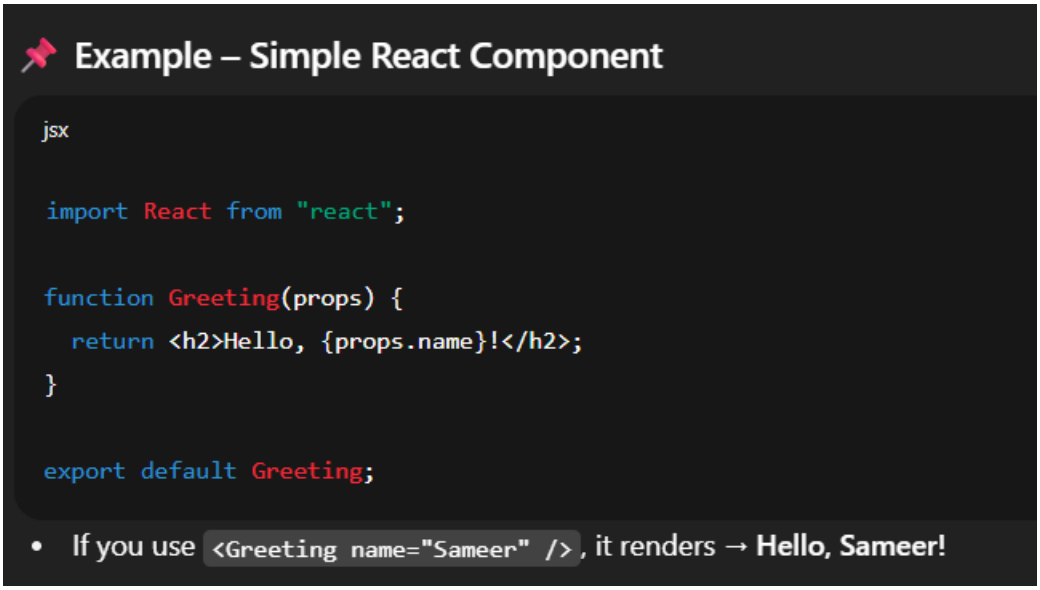
1. **Component-Based Architecture**
    - UI is divided into **small, reusable components** (e.g., Button, Navbar, Form).
    - Each component manages its own logic & rendering.

2. **Virtual DOM**
    - Instead of updating the real DOM directly, React uses a **Virtual DOM** (lightweight copy).
    - Only the **changed parts** are updated → **faster performance**.

3. **JSX (JavaScript XML)**
    - A syntax extension that lets you **write HTML inside JavaScript**.
    - Example: **const element = <h1>Hello, React!</h1>;**

4. **Unidirectional Data Flow**
    - Data flows in one direction (from parent → child components).
    - Makes debugging and maintenance easier.

5. **State & Props**
    - **State**: Stores component's own data (can change).
    - **Props**: Used to pass data from one component to another (read-only).



Advantages in Front-End Development

- **Reusable UI components** → saves time.
- **Efficient updates** → Virtual DOM ensures smooth performance.
- **Huge community & ecosystem** → plenty of tools, libraries, and support.
- **SEO-friendly** with server-side rendering (Next.js).

Real-World Use

- **Facebook, Instagram, Netflix, WhatsApp Web, Airbnb** use React for fast and interactive UIs.

2) Discuss various React Features in details.

1. Component-Based Architecture

- React applications are built using **components**.
- Each component is an independent, reusable block (like Lego pieces).
- Example: Navbar, Button, Footer → all can be written separately and reused.

2. Virtual DOM (Document Object Model)

- Traditional DOM updates are **slow**.
- React uses a **Virtual DOM**, a lightweight copy of the real DOM.
- When data changes, React compares (diffing) and updates **only the changed part** → faster performance.

3. JSX (JavaScript XML)

- React allows writing **HTML inside JavaScript**.
- Makes code easier to write and understand.
- Example: **const element = <h1>Hello React!</h1>;**

4. Unidirectional Data Flow

- Data flows **one way** (from parent to child via props).
- This makes debugging and managing the app easier.

5. State Management

- **State** stores data inside a component.
- When state changes, React **automatically re-renders** the component.
- Example: A counter button increases its value using state.

6. React Hooks (from React 16.8+)

- Hooks let you use **state and lifecycle methods** in functional components.
- Common hooks:
  - useState → manage state
  - useEffect → side effects (e.g., API calls)

7. React Native Support

- React can also be used for **mobile app development** (Android & iOS) using React Native.

8. Performance Optimization

- Virtual DOM + efficient rendering make React **faster** compared to traditional JS frameworks.

9. Large Community & Ecosystem

- Huge developer support, third-party libraries, and frameworks (like **Next.js** for SSR).

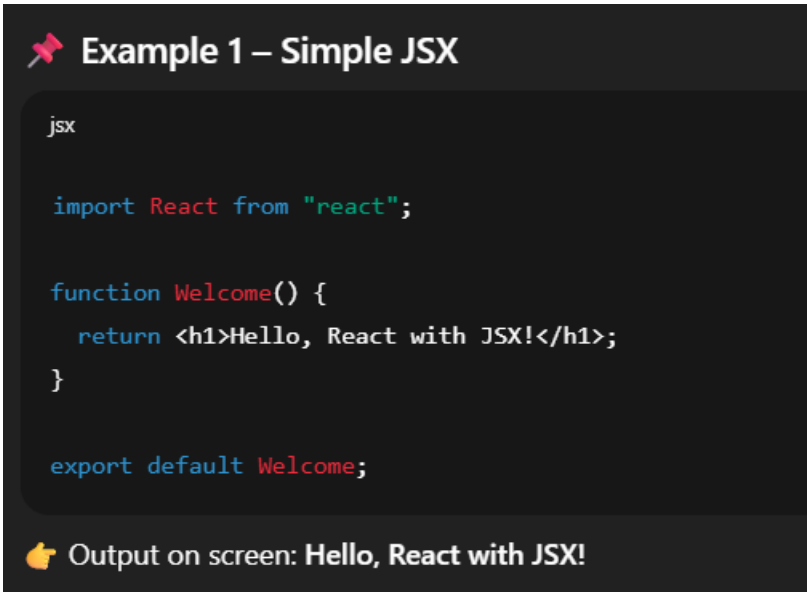
3) Discuss JSX with suitable example.

What is JSX?

- **JSX (JavaScript XML)** is a **syntax extension** in React that allows us to write **HTML-like code inside JavaScript**.
- It makes the code more **readable** and **intuitive**.
- JSX is not HTML; it gets **transpiled into JavaScript** by tools like **Babel** before running in the browser.

Why Use JSX?

1. **Readability** → Looks like HTML, easier for developers.
2. **Combines logic + UI** → Write markup and JavaScript in one place.
3. **Faster development** → Fewer lines of code.



📌 Example 2 – Using JavaScript inside JSX

```jsx
import React from "react";

function Greeting() {
  const name = "Sameer";
  return <h2>Welcome, {name}!</h2>;
}

export default Greeting;
```

👉 Here, `{name}` is a **JavaScript expression** inside JSX.
👉 Output: **Welcome, Sameer!**

4) Discuss Virtual DOM in detail in react.

📌 What is DOM?

- **DOM (Document Object Model)** is a tree-like representation of an HTML page.
- Example:

```html
<html>
  <body>
    <h1>Hello</h1>
  </body>
</html>
```

👉 Browser converts this into a **DOM Tree** that JavaScript can access and modify.

Problem with Real DOM

- The **Real DOM** is **slow** because:
    1. Every time data changes, the whole DOM (or a large part of it) is re-rendered.
    2. Frequent updates reduce performance in **large applications**.

What is Virtual DOM?

- The **Virtual DOM (VDOM)** is a **lightweight, virtual copy** of the Real DOM.
- React keeps this VDOM in memory.
- When something changes:
    1. React updates the Virtual DOM first.
    2. It then **compares (diffing)** the new Virtual DOM with the previous one.
    3. Only the **changed parts** are updated in the Real DOM (not the entire page).

How Virtual DOM Works (Step by Step)

1. **Render**: UI is first rendered in Virtual DOM.
2. **Diffing**: React compares old VDOM with the new VDOM.
3. **Reconciliation**: Only the changed elements are updated in the Real DOM.

📌 Example – Without and With Virtual DOM

Without Virtual DOM (Traditional DOM)

- If a small text changes (e.g., counter value), the browser **refreshes the entire UI block.**

With Virtual DOM (React)

jsx Copy Edit

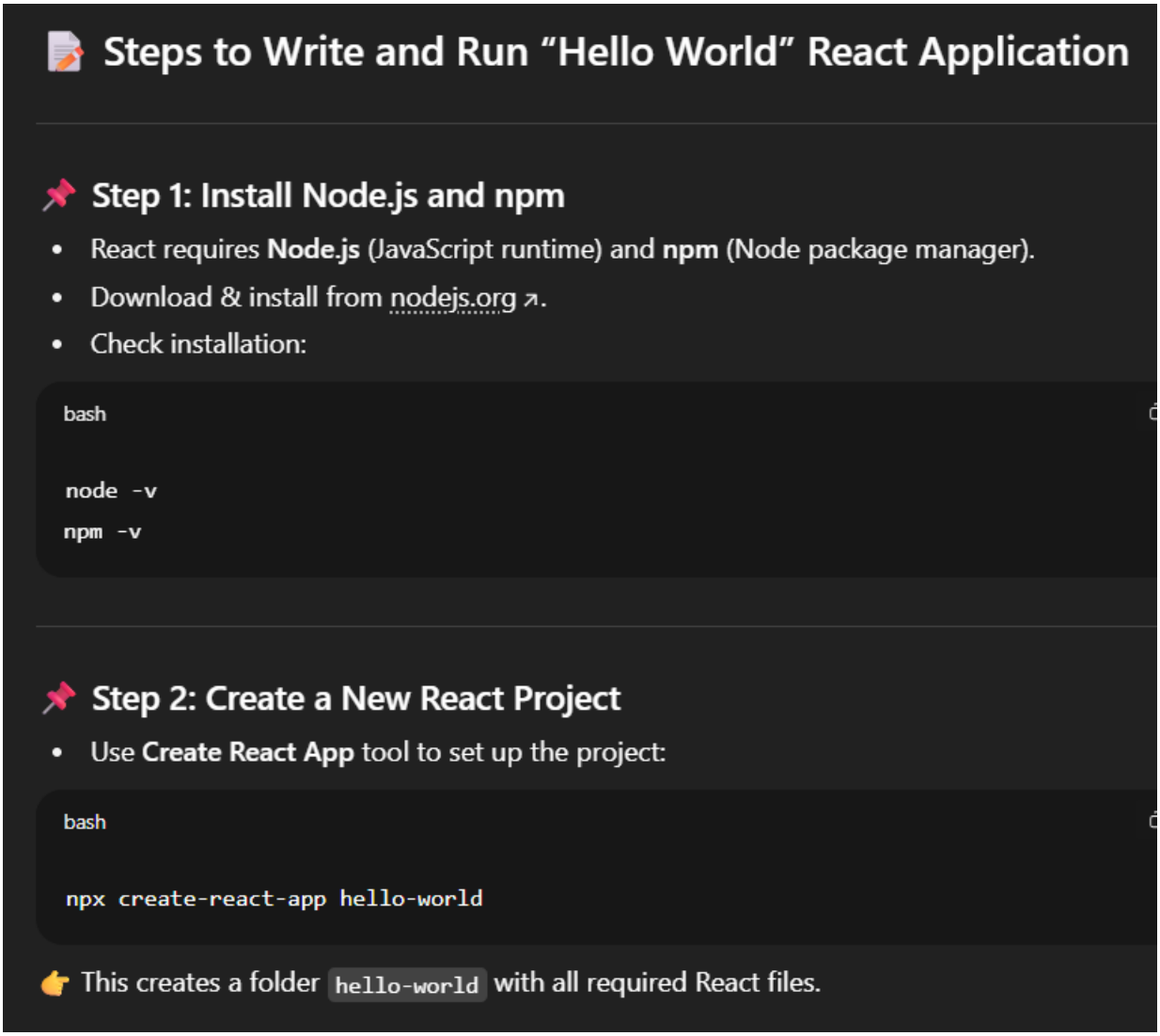
```jsx
import React, { useState } from "react";

function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <h2>Count: {count}</h2>
      <button onClick={() => setCount(count + 1)}>Increase</button>
    </div>
  );
}

export default Counter;
```

👉 Here, when `count` changes, React **only updates** `<h2>` in the real DOM instead of reloading the entire `<div>`.

Advantages of Virtual DOM

- **Faster performance** (minimal updates).
- **Efficient memory usage**.
- **Improves user experience** (no flickering or unnecessary reloads).
- Makes React suitable for **large-scale, dynamic applications**.

5) Discuss the details steps how to write and run "Hello World" React application.



📌 Step 3: Move into the Project Folder

```bash
cd hello-world
```

📌 Step 4: Start Development Server

```bash
npm start
```

👉 Runs the app on http://localhost:3000 ↗.

📌 Step 5: Modify App.js to Print "Hello World"

Open `src/App.js` and replace code with:

```jsx
import React from "react";

function App() {
  return (
    <div>
      <h1>Hello World</h1>
    </div>
  );
}

export default App;
```

📌 Step 6: View in Browser

- Save the file.
- Browser will automatically reload.
- You'll see: **Hello World** displayed. 🎉

6) Discuss what is components and its types in react.

What is a Component?

- A **Component** in React is a **reusable piece of code** that represents a part of the UI (User Interface).
- Example: A button, header, footer, or form can all be components.
- Components make applications **modular, maintainable, and reusable**.
- React apps are built by **combining multiple components together**.

📌 Types of Components in React

React mainly has **two types of components**:

---

1. Functional Components

- Defined as a **JavaScript function**.
- Return **JSX** (UI elements).
- **Simple, easy, and widely used** (especially with React Hooks).

✅ Example:

jsx

```jsx
import React from "react";

function Welcome() {
  return <h1>Hello, I am a Functional Component!</h1>;
}

export default Welcome;
```

2. Class Components

- Defined using **ES6 class** syntax.
- Can use **state** and **lifecycle methods**.
- Older way (before React Hooks), but still important.

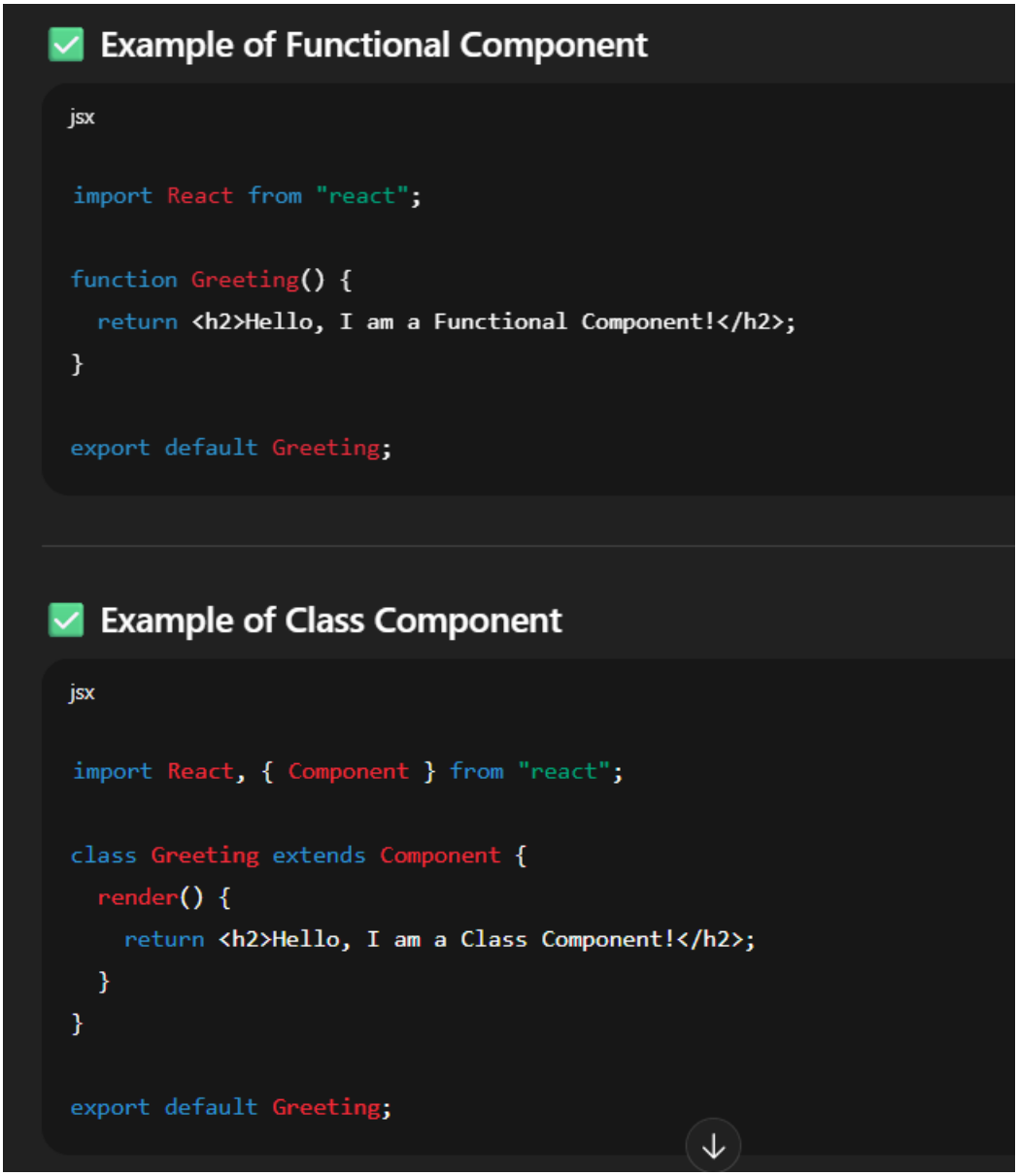
✅ Example:

```jsx
import React, { Component } from "react";

class Welcome extends Component {
  render() {
    return <h1>Hello, I am a Class Component!</h1>;
  }
}

export default Welcome;
```

7) Differentiate functional and class component in detail with example.

📝 Difference between Functional and Class Components in React

Aspect	Functional Component	Class Component
Definition	A simple JavaScript function that returns JSX.	A component created using ES6 `class` syntax that extends `React.Component`.
Syntax	Written as a **function**.	Written as a **class**.
State Management	Earlier **stateless**, but can now use **Hooks** (`useState`, `useEffect`).	Has a built-in **state object** to manage data.
Lifecycle Methods	Cannot directly use lifecycle methods (but Hooks can mimic them).	Supports lifecycle methods like `componentDidMount()`, `componentDidUpdate()`, etc.
Code Simplicity	Short, clean, and easy to write.	More verbose and complex.
Performance	Generally faster because they are simple functions.	Slightly slower due to class overhead.
Usage (Modern React)	Most recommended approach in modern React (with Hooks).	Used in older codebases, but less preferred now.



8) Discuss props with suitable example.

What are Props?

- **Props** stands for **Properties**.
- They are **inputs** to a React component.
- Used to **pass data from parent component to child component**.
- They make components **dynamic and reusable**.
- Props are **read-only** (cannot be modified inside the child component).

Key Points about Props

- Passed as **attributes** in JSX.
- Accessed inside a component using props.
- Similar to **function parameters** in JavaScript.
- Help in making **components reusable**.

Example of Props in React

Parent Component (`App.js`)

```jsx
import React from "react";
import Book from "./Book";

function App() {
  return (
    <div>
      <Book title="JavaScript Basics" author="John Doe" />
      <Book title="React for Beginners" author="Jane Smith" />
    </div>
  );
}

export default App;
```

Child Component (`Book.js`)

```jsx
import React from "react";

function Book(props) {
  return (
    <h2>
      Book: {props.title}, Author: {props.author}
    </h2>
  );
}

export default Book;
```

Output

```yaml
Book: JavaScript Basics, Author: John Doe
Book: React for Beginners, Author: Jane Smith
```

9) Discuss state in react with suitable example.

What is State?

- **State** is a **built-in object** in React used to store **dynamic data** of a component.
- Unlike **props** (which are passed from parent), **state is managed inside the component itself**.
- When state changes, the component **re-renders automatically**.
- State is **mutable** (can be updated using setState in class components or useState Hook in functional components).

Key Points about State

- Belongs **only to the component** that defines it.
- Changes in state trigger **UI updates**.
- Managed using:
    - this.state & this.setState() in **Class Components**
    - useState() Hook in **Functional Components**