Concurracy and Syncronization

🔷 1. What is Concurrency in Java?

Concurrency in Java means multiple threads executing independently, but possibly sharing resources.

- It allows a program to perform **multiple tasks at the same time**.
- For example, a web server handles multiple requests concurrently using threads.

🔷 2. What is Synchronization?

When multiple threads access **shared resources** (like variables or files), they can interfere with each other. This is called a **race condition**.

Synchronization is the mechanism to control thread access to shared resources, to prevent **data inconsistency**.

🔷 3. Thread Example Without Synchronization (Problem)

java

CopyEdit

```
class Counter {
  int count = 0;

  public void increment() {
    count++;
  }
}

public class Test {
  public static void main(String[] args) throws InterruptedException {
    Counter c = new Counter();
```

```
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) c.increment();
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) c.increment();
        });

        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Count: " + c.count);  // May not be 2000 due to race condition
    }
}
```

❗**Problem:**

Due to **race condition**, the final count may be **less than 2000**.

🔷 **4. Solution: Using synchronized Keyword**

```
java
CopyEdit
class Counter {
    int count = 0;

    public synchronized void increment() {
        count++;
```

```
    }
}
```

- synchronized ensures that only **one thread** can execute increment() at a time.
- It **locks** the object when one thread enters the method.

🔷 5. Synchronization Techniques in Java

✅ a. Synchronized Method

```
java
CopyEdit
public synchronized void increment() {
    count++;
}
```

✅ b. Synchronized Block

Useful for synchronizing only part of the code.

```
java
CopyEdit
public void increment() {
    synchronized(this) {
        count++;
    }
}
```

✅ c. Static Synchronization

For static methods shared across all instances.

```
java
CopyEdit
public static synchronized void increment() { }
```

✅ d. Using Lock Interface (Advanced)

java

CopyEdit

```
import java.util.concurrent.locks.ReentrantLock;

class Counter {
   private int count = 0;
   private ReentrantLock lock = new ReentrantLock();

   public void increment() {
      lock.lock();
      try {
         count++;
      } finally {
         lock.unlock();
      }
   }
}
```

🔷 6. Inter-thread Communication (wait/notify)

Sometimes, threads need to **communicate** (e.g., one waits for another to finish).

java

CopyEdit

```
class Shared {
   boolean ready = false;

   synchronized void waitForSignal() throws InterruptedException {
      while (!ready)
         wait();  // Wait until notified
```

```
        System.out.println("Received signal!");
    }

    synchronized void sendSignal() {
        ready = true;
        notify(); // Notify waiting thread
    }
}
```

🔷 7. Common Problems Without Synchronization

- **Race conditions**
- **Data inconsistency**
- **Thread interference**
- **Deadlocks** (if locks are not used properly)

🔷 Summary Table

Concept	Meaning
Concurrency	Multiple threads executing simultaneously
Synchronization	Mechanism to control access to shared resources
synchronized	Java keyword to lock method/block to one thread at a time
wait()/notify()	Used for communication between threads
ReentrantLock	Advanced locking mechanism with more control

Java thread Model

🔷 What Are Thread Priorities?

In Java, every thread has a **priority** that helps the **Thread Scheduler** decide the order in which threads should be executed.

- Priorities are represented by **integers from 1 to 10**:
    - Thread.MIN_PRIORITY = **1**
    - Thread.NORM_PRIORITY = **5** *(default)*
    - Thread.MAX_PRIORITY = **10**

⚠️ **Note:** Thread priorities are a **suggestion**, not a guarantee. Actual behavior may vary across operating systems and JVM implementations.

🔷 Why Use Thread Priorities?

Thread priorities help indicate **which thread is more important**. The **Thread Scheduler** may prefer higher-priority threads over lower-priority ones.

🔷 Setting and Getting Thread Priority

java

CopyEdit

Thread t = new Thread();

t.setPriority(8); // Set priority to 8

int p = t.getPriority(); // Get current priority

🔷 Priority Constants in Java

Constant	Value	Description
Thread.MIN_PRIORITY	1	Lowest priority
Thread.NORM_PRIORITY	5	Default priority

Constant	Value	Description
Thread.MAX_PRIORITY	10	Highest priority

🔷 **Example: Thread Priorities in Action**

java

CopyEdit

```
class MyThread extends Thread {
   public void run() {
      System.out.println(Thread.currentThread().getName() +
         " is running with priority " + getPriority());
   }
}

public class ThreadPriorityDemo {
   public static void main(String[] args) {
      MyThread t1 = new MyThread();
      MyThread t2 = new MyThread();
      MyThread t3 = new MyThread();

      t1.setName("Thread-1");
      t2.setName("Thread-2");
      t3.setName("Thread-3");

      t1.setPriority(Thread.MIN_PRIORITY);   // Priority 1
      t2.setPriority(Thread.NORM_PRIORITY);  // Priority 5
      t3.setPriority(Thread.MAX_PRIORITY);   // Priority 10
```

```
        t1.start();
        t2.start();
        t3.start();
    }
}
```

🔶 **Sample Output:**

csharp

CopyEdit

Thread-1 is running with priority 1

Thread-3 is running with priority 10

Thread-2 is running with priority 5

⚠️ **Important:** Execution order is **not guaranteed** — but higher-priority threads **may** run earlier or more frequently.

🔷 **Priority Inheritance in Java**

- A **new thread inherits** the priority of the **parent thread** by default.

java

CopyEdit

```
public class InheritedPriority {
    public static void main(String[] args) {
        Thread.currentThread().setPriority(7);

        Thread child = new Thread(() -> {
            System.out.println("Child thread priority: " +
                Thread.currentThread().getPriority());
        });

        child.start();  // Child inherits priority 7
```

```
    }
}
```

🔷 Real-World Use Cases

- Give **high priority** to threads handling **UI or real-time tasks**.
- Use **low priority** for **background or maintenance** tasks.
- Avoid setting all threads to MAX_PRIORITY — it may **starve** lower-priority threads.

🔷 Summary

Feature	Description
Default Priority	Thread.NORM_PRIORITY = 5
Range	1 to 10
Set Priority	setPriority(int level)
Get Priority	getPriority()
Scheduler Use	May prefer higher-priority threads
Not Guaranteed	JVM/OS may not strictly follow priority levels

🔷 What is Synchronization?

Synchronization in Java is a **mechanism** that ensures **only one thread** can access a **shared resource** (like a variable, file, or database) at a time.

It is used to **prevent race conditions** and ensure **data consistency** when multiple threads access shared data.

🔷 Why is Synchronization Needed?

✅ Problem Without Synchronization:

If two or more threads try to modify the same data at the same time, it can cause **unexpected behavior**.

java

CopyEdit

```
class Counter {
   int count = 0;

   public void increment() {
      count++;
   }
}
```

If two threads run increment() simultaneously, they may **read and write stale values**, resulting in **wrong output**.

🔷 How to Synchronize in Java

✅ 1. Synchronized Method

Locks the entire method so **only one thread can execute it at a time**.

java

CopyEdit

```
class Counter {
   int count = 0;

   public synchronized void increment() {
      count++;
   }
}
```

✅ 2. Synchronized Block

Locks only a **portion of code**, which is **more efficient** if only part of the method needs synchronization.

java

CopyEdit

```
class Counter {
    int count = 0;

    public void increment() {
        synchronized(this) {
            count++;
        }
    }
}
```

You can also synchronize on any object:

java

CopyEdit

```
synchronized(someObject) {
    // synchronized code
}
```

✅ **3. Static Synchronization**

If the method is static, synchronize on the **class object**.

java

CopyEdit

```
class Counter {
    static int count = 0;

    public static synchronized void increment() {
        count++;
    }
```

```
}
```

🔷 **Full Example: Synchronization vs No Synchronization**

❌ **Without Synchronization (Race Condition)**

java

CopyEdit

```
class Counter {
    int count = 0;

    public void increment() {
        count++;
    }
}

public class WithoutSync {
    public static void main(String[] args) throws InterruptedException {
        Counter c = new Counter();

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) c.increment();
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) c.increment();
        });

        t1.start();
        t2.start();
```

```
        t1.join();
        t2.join();

        System.out.println("Final count: " + c.count);  // Likely < 2000
    }
}
```

✅ **With Synchronization (Correct Output)**

```
java
CopyEdit
class Counter {
    int count = 0;

    public synchronized void increment() {
        count++;
    }
}

public class WithSync {
    public static void main(String[] args) throws InterruptedException {
        Counter c = new Counter();

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) c.increment();
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) c.increment();
        });
```

```
        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Final count: " + c.count);  // 2000
    }
}
```

🔷 Locking with ReentrantLock (Advanced)

For more control than synchronized, Java provides java.util.concurrent.locks.ReentrantLock.

```
java
CopyEdit
import java.util.concurrent.locks.ReentrantLock;

class Counter {
    private int count = 0;
    private ReentrantLock lock = new ReentrantLock();

    public void increment() {
        lock.lock();
        try {
            count++;
        } finally {
            lock.unlock();
        }
```

```
    }
}
```

🔷 Key Terms

Term	Meaning
Monitor	An internal lock used by Java to control access to synchronized code
Race Condition	A flaw that occurs when multiple threads access shared data without proper synchronization
Deadlock	When two or more threads are blocked forever, each waiting on the other
Thread-safe	Code is thread-safe when it functions correctly in a multi-threaded environment

---

🔷 Summary

Feature	Description
synchronized keyword	Prevents multiple threads from accessing code simultaneously
Synchronized Method	Locks the whole method
Synchronized Block	Locks only part of the method
Static Sync	Used for static methods
ReentrantLock	Advanced tool with manual locking and unlocking

📪 Messaging in Java (Inter-Thread Communication)

🔷 What is Messaging in Java?

Messaging in Java refers to **inter-thread communication**, where threads coordinate and **share data or signals** to perform tasks in a **synchronized and ordered** manner.

Java provides built-in methods for thread communication:

- wait()
- notify()
- notifyAll()

These methods belong to the Object class because every object in Java can be used as a **monitor** (lock).

🔷 Why Use Messaging?

When one thread needs to **wait for another thread to complete** some task, messaging is used to:

- **Avoid busy-waiting** (wasting CPU cycles)
- **Synchronize workflow** among threads
- Allow **efficient resource sharing**

🔷 Methods for Messaging

Method	Description
wait()	Causes current thread to wait until it is notified
notify()	Wakes up a single thread waiting on the object
notifyAll()	Wakes up all threads waiting on the object

⚠️ These methods must be called **inside a synchronized block or method**, or they will throw IllegalMonitorStateException.

🔷 ✅ Example: Producer-Consumer Problem (Messaging in Action)

java

CopyEdit

```
class MessageBox {
   private String message;
   private boolean empty = true;
```

```
    // Consumer
    public synchronized String read() {
        while (empty) {
            try {
                wait();  // Wait until a message is available
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        empty = true;
        notify();  // Notify producer to put a new message
        return message;
    }

    // Producer
    public synchronized void write(String message) {
        while (!empty) {
            try {
                wait();  // Wait until the box is empty
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        this.message = message;
        empty = false;
        notify();  // Notify consumer to read
    }
}
```

```
public class MessagingExample {
    public static void main(String[] args) {
        MessageBox box = new MessageBox();

        // Producer thread
        Thread producer = new Thread(() -> {
            String[] messages = { "Hello", "Welcome", "To", "Java", "Messaging" };
            for (String msg : messages) {
                box.write(msg);
                System.out.println("Produced: " + msg);
            }
        });

        // Consumer thread
        Thread consumer = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                String msg = box.read();
                System.out.println("Consumed: " + msg);
            }
        });

        producer.start();
        consumer.start();
    }
}
```

🧠 **What's Happening?**

- **Producer** puts a message only when the box is empty.

- **Consumer** reads the message only when it's available.
- wait() pauses the thread, releasing the lock.
- notify() wakes up the other thread.

🔷 Key Points

- wait() releases the **monitor lock** and pauses the thread.
- notify() wakes up **one** waiting thread.
- notifyAll() wakes up **all** waiting threads.
- These are used to implement coordination between threads **more efficiently than using sleep or busy waiting.**

🔷 Real-World Use Cases

Use Case	Explanation
Producer-Consumer	Message queue or buffer
Task Queue	One thread produces tasks, another executes them
Job Scheduling Systems	One thread adds jobs, others process them
Thread Pools	Threads wait for jobs using wait() and get notified on availability

---

🔷 Summary Table

Concept	Description
wait()	Pause thread and release lock until notified
notify()	Wake up a single waiting thread
notifyAll()	Wake up all waiting threads
synchronized	Required when using wait()/notify()

🔷 **Bonus: Messaging with BlockingQueue (Modern Alternative)**

Java provides high-level concurrency tools in java.util.concurrent.

java

CopyEdit

```
import java.util.concurrent.*;


public class BlockingQueueExample {
   public static void main(String[] args) {
      BlockingQueue<String> queue = new ArrayBlockingQueue<>(2);


      // Producer
      new Thread(() -> {
         try {
            queue.put("Hello");
            System.out.println("Produced: Hello");
         } catch (InterruptedException e) {}
      }).start();


      // Consumer
      new Thread(() -> {
         try {
            String msg = queue.take();
            System.out.println("Consumed: " + msg);
         } catch (InterruptedException e) {}
      }).start();
   }
}
```

✅ No need for manual wait/notify, as it's built into the BlockingQueue!

🧵 Main Thread in Java

🔷 What is the Main Thread?

- The **Main Thread** is the **initial thread** started automatically when a Java program begins execution.
- It is created by the **Java Virtual Machine (JVM)**.
- The entry point of the main thread is the main() method:

java

CopyEdit

public static void main(String[] args) { ... }

🔷 Characteristics of the Main Thread

Property	Description
Name	main
Created by	JVM
Entry Point	public static void main(String[] args)
Thread Group	main
Priority	5 (normal priority)
Lifecycle	Starts when the program begins and ends when main() completes or exits
Controls	It can create other threads and control them (start, join, etc.)

🔷 Accessing and Modifying Main Thread

You can get a reference to the main thread using:

java

CopyEdit

Thread t = Thread.currentThread();

✅ **Example: Main Thread Details**

java

CopyEdit

```
public class MainThreadExample {
   public static void main(String[] args) {
      Thread t = Thread.currentThread(); // Get the main thread
      System.out.println("Thread Name: " + t.getName());
      System.out.println("Thread Priority: " + t.getPriority());
      System.out.println("Thread Group: " + t.getThreadGroup().getName());
   }
}
```

🔶 **Output:**

yaml

CopyEdit

```
Thread Name: main
Thread Priority: 5
Thread Group: main
```

🔷 **Changing Main Thread Properties**

You can modify the thread name and priority:

java

CopyEdit

```
public class MainThreadModify {
   public static void main(String[] args) {
      Thread t = Thread.currentThread();

      t.setName("MyMainThread");
      t.setPriority(7);
```

```
        System.out.println("New Name: " + t.getName());
        System.out.println("New Priority: " + t.getPriority());
    }
}
```

🔷 Using Main Thread to Launch Other Threads

The main thread can start and manage other threads:

java

CopyEdit

```
class MyThread extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Child thread: " + i);
        }
    }
}

public class LaunchThread {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();  // Child thread starts

        for (int i = 1; i <= 5; i++) {
            System.out.println("Main thread: " + i);  // Main thread continues
        }
    }
}
```

✅ **Main thread** and **child thread** run **concurrently**.

🔷 Lifecycle of Main Thread

text

CopyEdit

[Created by JVM] → [Runs main()] → [Creates other threads (optional)] → [Terminates]

Even after the main thread terminates, the JVM stays alive **as long as other threads are running**, unless they are daemon threads.

🔷 Common Interview Question

Q: Can the main thread be restarted after it dies?

❌ **No.** Once the main thread finishes, it cannot be restarted like any other thread. You can **create new threads**, but you cannot restart the main() thread.

🔷 Summary

Aspect	Detail
Created By	JVM
Name	"main"
Can Modify	Yes (name, priority)
Entry Point	main() method
Role	Controls program start, can launch other threads
Lifecycle End	When main() exits, unless other threads are running

Implementing Thread using thread class and Runnable interface

Sure! Let's explore how to **implement threads in Java** using:

1. **Thread class (extending Thread)**
2. **Runnable interface (implementing Runnable)**

Both approaches are valid and commonly used.

🧵 Implementing Threads in Java

🔷 1. Extending the Thread Class

You can create a thread by **extending the Thread class** and **overriding its run() method**.

✅ Syntax:

java

CopyEdit

```
class MyThread extends Thread {
   public void run() {
      // Code executed by the thread
   }
}
```

✅ Example:

java

CopyEdit

```
class MyThread extends Thread {
   public void run() {
      for (int i = 1; i <= 5; i++) {
```

```
            System.out.println("Thread: " + i);
        }
    }
}

public class ThreadDemo1 {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start();  // Starts a new thread

        for (int i = 1; i <= 5; i++) {
            System.out.println("Main: " + i);
        }
    }
}
```

🔶 **Output (order may vary):**

makefile

CopyEdit

Main: 1

Thread: 1

Main: 2

Thread: 2

...

✅ **Notes:**

- start() starts the new thread.
- run() is called internally by the JVM.
- Each thread has its own call stack.

🔷 2. Implementing the Runnable Interface

You can also create a thread by **implementing the Runnable interface** and passing it to a Thread object.

✅ **Syntax:**

java

CopyEdit

```
class MyRunnable implements Runnable {
   public void run() {
      // Code executed by the thread
   }
}
```

✅ **Example:**

java

CopyEdit

```
class MyRunnable implements Runnable {
   public void run() {
      for (int i = 1; i <= 5; i++) {
         System.out.println("Runnable thread: " + i);
      }
   }
}


public class ThreadDemo2 {
   public static void main(String[] args) {
      MyRunnable r = new MyRunnable();
      Thread t1 = new Thread(r);
      t1.start();  // Starts a new thread
```

```
        for (int i = 1; i <= 5; i++) {
            System.out.println("Main: " + i);
        }
    }
}
```

🔷 Comparison: Thread vs Runnable

Feature	Thread Class	Runnable Interface
Inheritance	Extends Thread	Implements Runnable
Flexibility	Can't extend other classes	Can extend another class
Reusability	Less reusable	More reusable (can pass to many threads)
Recommended?	❌ Less preferred	✅ Preferred in real-world Java apps

---

🔷 When to Use What?

- ✅ Use Runnable when:
  - You want to implement **multithreading with inheritance from another class**.
  - You want **clean separation** of task logic from thread control.
- ❌ Avoid extending Thread unless:
  - Your class specifically **modifies or customizes thread behavior**.

🔷 Modern Approach (Lambda with Runnable – Java 8+)

java

CopyEdit

```
public class LambdaThread {
    public static void main(String[] args) {
        Runnable task = () -> {
```

```
        for (int i = 1; i <= 3; i++) {
            System.out.println("From Lambda Runnable: " + i);
        }
    };

    Thread t = new Thread(task);
    t.start();
  }
}
```

🔷 Summary

Aspect	Thread Class	Runnable Interface
Inheritance Limit	Single inheritance only	Can implement many interfaces
Flexibility	Less	More flexible
Preferred	Less in real apps	Recommended

🧵 Creating Multiple Threads using isAlive() and join() in Java

---

🔷 Overview

Java provides two useful methods for thread lifecycle management:

Method	Purpose
isAlive()	Checks if a thread is still running
join()	Waits for a thread to finish before continuing

🔷 What is isAlive()?

- public boolean isAlive()
- Returns true if the thread is **still running**.
- Returns false if the thread has **finished executing**.

✅ Example:

java

CopyEdit

```
Thread t = new Thread();
System.out.println(t.isAlive());  // false before start
t.start();
System.out.println(t.isAlive());  // true after start (maybe)
```

🔷 What is join()?

- public final void join() throws InterruptedException
- Waits for the **thread to die** (complete).
- It **blocks the current thread** until the target thread finishes.

✅ **Full Example: Multiple Threads using isAlive() and join()**

java

CopyEdit

```
class MyThread extends Thread {
   public void run() {
      for (int i = 1; i <= 3; i++) {
         System.out.println(getName() + " is running: " + i);
         try {
            Thread.sleep(500);  // Simulate some work
         } catch (InterruptedException e) {
            System.out.println("Interrupted: " + getName());
         }
      }
   }
}

public class ThreadDemo {
   public static void main(String[] args) {
      MyThread t1 = new MyThread();
      MyThread t2 = new MyThread();

      System.out.println("Before starting threads:");
      System.out.println("Is t1 alive? " + t1.isAlive());
      System.out.println("Is t2 alive? " + t2.isAlive());

      t1.start();
      t2.start();
```

```
        System.out.println("After starting threads:");
        System.out.println("Is t1 alive? " + t1.isAlive());
        System.out.println("Is t2 alive? " + t2.isAlive());

        try {
            // Wait for t1 and t2 to finish
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted");
        }

        System.out.println("After threads finish:");
        System.out.println("Is t1 alive? " + t1.isAlive());
        System.out.println("Is t2 alive? " + t2.isAlive());

        System.out.println("Main thread ends.");
    }
}
```

🔶 **Sample Output:**

yaml

CopyEdit

Before starting threads:

Is t1 alive? false

Is t2 alive? false

After starting threads:

Is t1 alive? true

Is t2 alive? true

Thread-0 is running: 1

Thread-1 is running: 1

Thread-0 is running: 2

Thread-1 is running: 2

Thread-0 is running: 3

Thread-1 is running: 3

After threads finish:

Is t1 alive? false

Is t2 alive? false

Main thread ends.

🔷 Use Cases

Method	Use When
isAlive()	You want to check if a thread is still working
join()	You want to **pause the main thread** until another thread completes

---

🔷 Summary

Method	Description
isAlive()	Returns true if thread is still running
join()	Waits for thread to finish before proceeding

- isAlive() → **non-blocking check**
- join() → **blocking wait**

🔷 Bonus Tip

You can also pass a timeout to join():

java

CopyEdit

```
t1.join(1000);  // Wait for t1 to finish, or 1000ms max
```

Use of JavaScript for creating web based applications in Java

Using **JavaScript** in **Java-based web applications** is very common and essential for creating **interactive, dynamic user interfaces**. Java handles the **server-side logic**, while JavaScript enhances the **client-side (browser) behavior**.

🔷 Why Use JavaScript in Java Web Applications?

Purpose	Java (Backend)	JavaScript (Frontend)
Handle business logic	✅	❌
Handle browser events (clicks)	❌	✅
Create dynamic UI (without reload)	❌	✅
Connect to databases	✅	❌
Send/receive data asynchronously	❌	✅ (AJAX, Fetch API)

🔷 Common Java Technologies Combined with JavaScript

Java Tech	Description
JSP/Servlets	Basic Java web components
Spring Boot (MVC)	Popular framework for REST APIs
JSF	JavaServer Faces for UI in Java

In all these, JavaScript enhances the **frontend behavior** of the HTML rendered by Java (JSP/Thymeleaf/etc.).

✅ Example: Java + JavaScript (Simple Form)

Backend (JSP + Servlet):

index.jsp:

jsp

CopyEdit

```
<html>
<head>
  <title>Form Example</title>
  <script>
    function validateForm() {
      var name = document.forms["myForm"]["username"].value;
      if (name === "") {
        alert("Name must be filled out");
        return false;
      }
    }
  </script>
</head>
<body>
  <form name="myForm" action="submit" method="post" onsubmit="return validateForm()">
    Name: <input type="text" name="username">
    <input type="submit" value="Submit">
  </form>
</body>
</html>
```

Servlet (SubmitServlet.java):

java

CopyEdit

```
@WebServlet("/submit")
public class SubmitServlet extends HttpServlet {
```

```
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        String name = request.getParameter("username");
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("Hello, " + name + "!");
    }
}
```

🔷 How JavaScript Helps Here

Task	Done by	Description
Input validation	JavaScript	Prevents empty form submission
Page rendering	JSP (Java)	Renders HTML
Form submission handling	Servlet (Java)	Processes data and sends response

---

🔷 Advanced Usage: JavaScript with Java REST API (AJAX)

Java Backend: Spring Boot REST Controller

java

CopyEdit

```
@RestController
public class UserController {
    @GetMapping("/greet")
    public String greetUser(@RequestParam String name) {
        return "Hello, " + name + "!";
    }
}
```

Frontend HTML + JavaScript:

html

CopyEdit

```
<!DOCTYPE html>
<html>
<head>
  <title>AJAX Example</title>
  <script>
    function fetchGreeting() {
      var name = document.getElementById("username").value;
      fetch("http://localhost:8080/greet?name=" + name)
        .then(response => response.text())
        .then(data => {
          document.getElementById("result").innerText = data;
        });
    }
  </script>
</head>
<body>
  Name: <input type="text" id="username">
  <button onclick="fetchGreeting()">Greet</button>
  <p id="result"></p>
</body>
</html>
```

✅ This is a modern **Java backend + JavaScript frontend** interaction using **AJAX** (Fetch API).

🔷 **Summary**

Java	JavaScript
Server-side logic	Client-side behavior
Data processing & storage	Event handling, DOM manipulation
Generates HTML (JSP, Thymeleaf)	Enhances HTML interactivity
REST API using Spring Boot	Calls APIs using fetch() or AJAX

🔷 Final Thoughts

JavaScript is **not a replacement** for Java in web apps — it's a **companion**. Java handles the heavy lifting on the server; JavaScript delivers a smooth user experience in the browser.

Reactjs , Angularhjs , Vue js

⚔️ ReactJS vs AngularJS vs Vue.js

These are the **three most popular frontend JavaScript frameworks/libraries** used for building modern web applications.

🔷 1. ReactJS

✅ Overview:

- Developed by: **Facebook**
- First Released: **2013**
- Type: **Library** (for building UI)
- Architecture: **Component-based**
- Language: **JavaScript + JSX**

✅ Key Features:

- **Virtual DOM** for fast UI rendering
- **JSX** (JavaScript + XML) syntax
- **One-way data binding**
- **Component-based architecture**
- **Unidirectional data flow**
- Strong ecosystem with **React Router**, **Redux**, etc.

❌ Limitations:

- Just the "View" — not full framework
- JSX has a learning curve
- Needs third-party libraries for routing, state management

✅ Applications:

- Facebook, Instagram, WhatsApp Web
- Single-page applications (SPAs)

- Dashboards and real-time UIs

🔷 2. AngularJS

✅ **Overview:**

- Developed by: **Google**
- First Released: **2010** (AngularJS), now evolved to **Angular 2+**
- Type: **Full framework**
- Architecture: **MVC / MVVM**
- Language: **JavaScript (AngularJS)**, **TypeScript (Angular 2+)**

✅ **Key Features:**

- **Two-way data binding**
- **Directives** (custom HTML tags)
- **Dependency injection**
- **Built-in routing and HTTP support**
- **Comprehensive CLI**
- **Real-time form validation**

❌ **Limitations:**

- Steep learning curve
- Complex syntax (especially in Angular 2+)
- Heavier and slower than React/Vue for small apps

✅ **Applications:**

- Google apps (like Gmail), Microsoft Office Web, PayPal
- Enterprise-level applications
- Form-heavy web apps

🔷 3. Vue.js

✅ **Overview:**

- Developed by: **Evan You**
- First Released: **2014**
- Type: **Progressive Framework**
- Architecture: **Component-based**
- Language: **JavaScript + HTML templates**

✅ **Key Features:**

- **Virtual DOM**
- **Two-way data binding** (like Angular)
- **Single-file components** (.vue files)
- **Vue CLI**
- Simpler syntax, beginner-friendly
- Lightweight and flexible

❌ **Limitations:**

- Smaller community than React/Angular
- Fewer enterprise-level tools
- May face integration issues in large-scale apps

✅ **Applications:**

- Alibaba, Xiaomi, GitLab
- Admin dashboards
- Lightweight SPAs

🔍 **Difference Between ReactJS, AngularJS, and Vue.js**

Feature	ReactJS	AngularJS	Vue.js
Developed By	Facebook	Google	Evan You (open-source)
First Release	2013	2010	2014

Feature	ReactJS	AngularJS	Vue.js
Type	Library (UI only)	Full Framework	Progressive Framework
Language	JavaScript + JSX	JavaScript (JS) / TypeScript	JavaScript
Data Binding	One-way	Two-way	Two-way
DOM	Virtual DOM	Real DOM (AngularJS), Shadow DOM	Virtual DOM
Learning Curve	Moderate	Steep	Easy to Moderate
Size	Small	Large	Small
Use Case	Dynamic SPAs	Complex enterprise apps	Lightweight and quick apps
Flexibility	High (requires add-ons)	Low (rigid structure)	High

---

✅ **When to Use Which?**

Use Case	Best Choice	Why?
Quick learning and prototyping	**Vue.js**	Easy syntax, fast setup
Large enterprise-grade app	**AngularJS** / Angular	Full ecosystem, powerful tooling
Highly dynamic UI (e.g., news feed)	**ReactJS**	Virtual DOM, fast rendering
SEO-friendly apps	**ReactJS (with Next.js)**	Server-side rendering
Form-heavy apps	**Angular**	Built-in validation and form modules

---

📝 **Summary Table**

Framework	Type	Language	Data Binding	Learning Curve	Best For
ReactJS	Library	JSX (JS)	One-way	Moderate	Dynamic SPAs
AngularJS	Framework	JS/TypeScript	Two-way	Hard	Enterprise apps
Vue.js	Framework	JavaScript	Two-way	Easy	Small to Medium apps

---

🔚 Conclusion

- **ReactJS** = Flexible and fast for UI
- **AngularJS** = Full solution for large projects
- **Vue.js** = Lightweight, beginner-friendly

React js

🔷 What is React?

- **ReactJS** (commonly just *React*) is a **JavaScript library** for building **component-based user interfaces**.
- Developed and maintained by **Facebook**.
- Released in **2013**.
- Focuses on the **View layer** in the MVC architecture.
- Enables developers to build **dynamic, interactive single-page applications (SPAs)** efficiently.

🔷 Core Concepts of React

1. Components

- React apps are built with **components**.
- Components are reusable, self-contained pieces of UI.
- They can be **functional** or **class-based**.

jsx

CopyEdit

```
// Functional Component
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

2. JSX (JavaScript XML)

- JSX is a syntax extension to JavaScript.
- It looks like HTML but allows you to write UI inside JavaScript.
- JSX gets compiled to React.createElement() calls.

jsx

CopyEdit

```
const element = <h1>Hello, world!</h1>;
```

3. Virtual DOM

- React uses a **virtual DOM** to optimize UI updates.
- Instead of manipulating the real DOM directly, React creates a lightweight copy.
- It compares the new virtual DOM with the previous version (diffing).
- Only the differences are updated in the real DOM — this improves performance.

4. One-way Data Binding (Unidirectional Data Flow)

- Data flows **downward** from parent components to child components via **props**.
- Makes app easier to debug and understand.

5. State

- Components can have **state** — data that changes over time.
- When state changes, React re-renders the component and updates the UI.

jsx

CopyEdit

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


 return (
  <>
   <p>You clicked {count} times</p>
   <button onClick={() => setCount(count + 1)}>Click me</button>
  </>
 );
}
```

6. Lifecycle Methods (Class Components)

- Special methods like componentDidMount(), componentDidUpdate() let you run code at different points in a component's life.

- In functional components, **React hooks** like useEffect() replace lifecycle methods.

🔷 Features of React

Feature	Description
Declarative UI	Write what UI should look like, React handles updates.
Component-Based	Build reusable UI components.
JSX Syntax	Write HTML-like code in JavaScript.
Virtual DOM	Efficient UI rendering.
Strong Community	Large ecosystem and many tools.
React Hooks	Simplify state and lifecycle management in functional components.

🔷 Advantages of React

- **Fast rendering** with Virtual DOM.
- **Reusable components** reduce code duplication.
- Easy to learn if you know JavaScript.
- Rich ecosystem: React Router, Redux, Next.js, etc.
- Strong backing by Facebook ensures continuous development.
- Good for building SPAs and mobile apps (React Native).

🔷 Limitations of React

- Only handles UI layer, you need additional libraries for routing, state management.
- JSX syntax may be confusing initially.
- Frequent updates and ecosystem changes can require learning new best practices.

- SEO requires server-side rendering (e.g., with Next.js).

🔷 Example React Application

jsx

CopyEdit

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

function App() {
 const [todos, setTodos] = useState([]);
 const [task, setTask] = useState("");

 const addTodo = () => {
  if (task !== "") {
   setTodos([...todos, task]);
   setTask("");
  }
 };

 return (
  <div>
   <h1>Todo List</h1>
   <input
    type="text"
    value={task}
    onChange={(e) => setTask(e.target.value)}
    placeholder="Add new task"
   />
   <button onClick={addTodo}>Add</button>
```

```
    <ul>
      {todos.map((todo, index) => (
        <li key={index}>{todo}</li>
      ))}
    </ul>
  </div>
 );
}

export default App;
```

🔷 React Ecosystem Highlights

Tool/Library	Purpose
React Router	Routing/navigation between pages
Redux	State management
Next.js	Server-side rendering & static site generation
React Native	Build mobile apps with React

🔷 Summary

Aspect	Details
Creator	Facebook
Release Year	2013
Type	JavaScript Library
Main Focus	Building reusable UI components
Key Feature	Virtual DOM, JSX
Data Flow	One-way data binding

Aspect	Details
Suitable For	SPAs, interactive UI, mobile apps

Angular js

🔷 What is AngularJS?

- **AngularJS** is a **JavaScript-based open-source front-end web framework**.
- Developed and maintained by **Google**.
- First released in **2010**.
- Designed to **build dynamic single-page applications (SPAs)**.
- Uses **Model-View-Controller (MVC)** or **Model-View-ViewModel (MVVM)** architecture.
- Uses **HTML as the template language**.
- Extends HTML with **directives** to add dynamic behavior.

🔷 Core Concepts of AngularJS

1. Two-Way Data Binding

- Synchronizes data between **Model (JavaScript objects)** and **View (HTML UI)** automatically.
- Changes in the model update the view, and changes in the view update the model.

html

CopyEdit

<input ng-model="name">

Hello, {{name}}!

2. Directives

- Special HTML attributes that extend HTML functionality.
- Examples: ng-model, ng-bind, ng-repeat, ng-show, etc.
- Let you manipulate the DOM declaratively.

3. Controllers

- JavaScript functions that control the data of AngularJS applications.

- Controllers define the business logic and data scope for views.

```
js
CopyEdit
app.controller('MyController', function($scope) {
 $scope.message = "Hello AngularJS!";
});
```

4. Modules

- Containers for different parts of an app (controllers, services, filters, directives).
- Helps organize code and manage dependencies.

5. Dependency Injection (DI)

- AngularJS has a built-in DI subsystem.
- Manages the components and their dependencies efficiently.

6. Templates

- Written in HTML with AngularJS-specific markup.
- Rendered dynamically based on the model data.

🔷 Features of AngularJS

Feature	Description
Two-way Data Binding	Keeps model and view in sync automatically
MVC Architecture	Separates application logic and UI
Directives	Extend HTML with custom behavior
Dependency Injection	Manages services and components easily
Filters	Format data for display (e.g., currency)
Form Validation	Built-in validation and error handling
SPA Support	Create fast, single-page web apps

🔷 Advantages of AngularJS

- **Simplifies front-end development** by extending HTML.
- Reduces amount of code thanks to two-way data binding.
- Built-in services like routing, form validation, and HTTP requests.
- Supported and maintained by Google.
- Large community and ecosystem.
- Encourages **test-driven development** (TDD).

🔷 Limitations of AngularJS

- **Performance issues** with complex and large apps due to two-way data binding overhead.
- **Steep learning curve** for beginners.
- Heavy use of scopes and digest cycle can confuse new developers.
- Debugging can be complex.
- AngularJS (version 1.x) is now largely superseded by **Angular (2+)**, a complete rewrite.

🔷 Example AngularJS Application

html

CopyEdit

```
<!DOCTYPE html>
<html ng-app="myApp">
<head>
 <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></script>
</head>
<body ng-controller="MainController">
 <h1>AngularJS Example</h1>

 <input type="text" ng-model="name" placeholder="Enter your name">
```

```
 <p>Hello, {{name}}!</p>

 <ul>
  <li ng-repeat="item in items">{{item}}</li>
 </ul>

 <script>
  var app = angular.module('myApp', []);

  app.controller('MainController', function($scope) {
   $scope.name = "User";
   $scope.items = ['AngularJS', 'React', 'Vue'];
  });
 </script>
</body>
</html>
```

🔷 **AngularJS vs Angular (2+)**

Aspect	**AngularJS (1.x)**	**Angular (2+)**
Language	JavaScript	TypeScript
Architecture	MVC	Component-based
Performance	Slower in complex apps	Improved with Ahead-of-Time (AOT) compilation
Mobile Support	Limited	Full support
Learning Curve	Moderate	Steeper but more structured
Tooling & Ecosystem	Older, less modular	Modern CLI, RxJS, Angular CLI

Summary

Aspect	Description
Creator	Google
Release	2010
Type	Front-end web framework
Architecture	MVC / MVVM
Data Binding	Two-way
Template Language	HTML + Angular directives
Use Case	Dynamic SPAs, form-heavy apps

Vue js

🔷 What is Vue.js?

- **Vue.js** is a **progressive JavaScript framework** used to build **user interfaces and single-page applications (SPAs)**.
- Created by **Evan You** and released in **2014**.
- Designed to be **incrementally adoptable**, meaning you can use as much or as little of Vue as you like.
- Focuses on the **ViewModel layer** of the MVVM architecture.
- Known for its simplicity, flexibility, and gentle learning curve.

🔷 Core Concepts of Vue.js

1. Reactive Data Binding

- Vue binds data to the DOM reactively.
- When the data changes, Vue automatically updates the DOM.

```
html
CopyEdit
<div id="app">
 {{ message }}
</div>

<script>
 new Vue({
  el: '#app',
  data: {
   message: 'Hello Vue!'
  }
```

```
});
</script>
```

2. Components

- Vue apps are built using reusable components.
- Components can have their own data, methods, templates, and lifecycle hooks.

```
js
CopyEdit
Vue.component('todo-item', {
 props: ['todo'],
 template: '<li>{{ todo.text }}</li>'
});
```

3. Directives

- Special tokens in the markup that tell Vue to do something.
- Examples include: v-bind, v-if, v-for, v-model, v-on.

```
html
CopyEdit
<input v-model="message" placeholder="edit me">
<p>The input is: {{ message }}</p>
```

4. Vue Instance

- The root Vue instance is created with new Vue() and controls a part of the DOM.

5. Computed Properties

- Vue computes properties based on reactive data.
- Useful for expensive computations or when you want to declaratively define derived data.

```
js
CopyEdit
computed: {
 reversedMessage() {
```

```
    return this.message.split('').reverse().join('');
  }
}
```

6. Lifecycle Hooks

- Vue offers lifecycle hooks such as created(), mounted(), updated(), and destroyed() to run code at specific stages of a component's lifecycle.

🔷 Features of Vue.js

Feature	Description
Reactive Data Binding	Keeps DOM and data in sync automatically
Component-based	Build reusable, modular UI components
Virtual DOM	Efficient rendering and updating
Template Syntax	Declarative HTML templates with directives
Single File Components	.vue files combining template, script & style
Transitions & Animations	Easy to add transitions and animations
Easy Integration	Can be integrated into existing projects easily

---

🔷 Advantages of Vue.js

- **Gentle learning curve** — easier for beginners compared to Angular.
- Lightweight and performant.
- Flexible and modular — can be used as a library or full framework.
- Great documentation and strong community.
- Supports both **one-way** and **two-way data binding**.
- Excellent tooling (Vue CLI, Vue Devtools).
- Single File Components (.vue) encapsulate HTML, CSS, JS.

🔷 Limitations of Vue.js

- Smaller community compared to React and Angular.
- Less enterprise adoption (though this is growing).
- Ecosystem still maturing compared to React.
- Some plugin integrations less mature.

🔷 **Example Vue.js Application**

```
html
CopyEdit
<div id="app">
 <h1>{{ title }}</h1>
 <input v-model="newItem" placeholder="Add item">
 <button @click="addItem">Add</button>

 <ul>
  <li v-for="(item, index) in items" :key="index">
   {{ item }}
   <button @click="removeItem(index)">Remove</button>
  </li>
 </ul>
</div>

<script src="https://cdn.jsdelivr.net/npm/vue@2"></script>
<script>
 new Vue({
  el: '#app',
  data: {
   title: 'My Todo List',
   newItem: '',
```

```
    items: []
  },
  methods: {
    addItem() {
      if(this.newItem.trim() !== '') {
        this.items.push(this.newItem);
        this.newItem = '';
      }
    },
    removeItem(index) {
      this.items.splice(index, 1);
    }
  }
 });
</script>
```

🔷 Vue.js Ecosystem Highlights

Tool/Library	Purpose
Vue Router	Client-side routing
Vuex	State management
Vue CLI	Project scaffolding & build tools
Nuxt.js	Server-side rendering and static site generation

🔷 Summary

Aspect	Details
Creator	Evan You

Aspect	Details
Release Year	2014
Type	Progressive JavaScript Framework
Architecture	MVVM
Data Binding	Reactive, supports two-way
Language	JavaScript
Use Case	SPAs, interactive UI, integrations