previous virtual DOM, and sees UserProfile component is same type. Comparing prop it observes that name is same, age changed from 23 to 24. Again, div is the same class and h2 with the same content and in p where only the age number is being changed.

8) What is event ~~degradation~~ delegation in Javascript and how does it optimize performance? Explain with an example of a click event on a dynamically added elements.

Ans: Event delegation is a process where instead of attaching event listeners to individual ~~elements~~, we attach a single event listener to a parent element to handle event for all of its children. This works by leveraging event bubbling in DOM. It optimizes performance in following way:

1) Reduced memory usage.

2) Faster Initialization.

3) Works with Dynamic Content

4) Simpler management

5) Throttling and Debouncing Events.

6) Passive Event Listeners for scrolling

7) Avoiding inline event handlers in HTML

8) Using event capturing for early handling

9) Cleaning up unused Event Listeners.

10) Using event delegation with closest selector.

Example: Click event on dynamically added elements

```
document.addEventListener('click',
  (event) => {
    if(event.target.matches('.btn')){
      console.log('Button clicked!',
        event.target.textContent);
    }
});
```

```
const newButton = document.createElement
                  ('button');

newButton.className = 'btn';
newButton.textContent = 'New Button';
document.body.appendChild(new
                  Button);
```

2) Explain how Java Regular Expressions can be used for input validation. Write a regex pattern to validate an email address and describe how it works using Pattern and Matcher classes.

Ans. Java Regular Expressions are used for input validation by defining a pattern that must conform to, and then checking if the provided input string matches that pattern. This ensures data integrity and prevents invalid input from being processed.

Email validation example:

```
import java.util.regex.*;

public class EmailValidation{
    private static final String EMAIL_
    REGEX= "^[A-Za-z0-9+_.-]+
        @[A-Za-z0-9.-]+\\.[A-Za-
            z]{2,}$";

    Private static final Pattern pattern
    =Pattern.compile(EMAIL_REGEX);
    public static boolean isValidEmail
        (String email){
        Matcher matcher=pattern
            matcher(email);
        return matcher.matches();
    }
    public ~~sad~~ static void main(String
        [] args){
```

```
String[] testEmails = {
    "user@example.com",
    "invalid.email",
    "another.user@domain.co.uk",
    "missing@tld."
};

for(String email: testEmails) {
    System.out.println(email + ": " +
        (isValidEmail(email) ? "Valid" :
            "Invalid"));
}
}
}
```

Here, the sample Pattern pattern = Pattern.compile(EMAIL_REGEX) compiles the regex string into a Pattern object, and Matcher matcher = pattern.matcher(inputString) creates a matcher that will match the input

against the pattern.

10) What are custom annotations in Java and how can they be used to influence program behaviour at runtime using reflection? Design a simple custom annotation & show how it can be processed with annotated elements.

Ans. Custom annotations in Java are user-defined annotations that allows us to attach metadata to classes, methods, fields, parameters or other elements. They don't directly change program behaviour when program reads them using reflection at runtime and takes action based on annotation informations

i) Creating a Simple Custom Annotation:

```
import java.lang.annotation.*;
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
```

```
public @interface LogExecution {
    String value() default "Method executed";
    boolean logParameters() default false;
}
```

ii) Using Annotation:

```
public class Calculator {
    @LogExecution("Adding two numbers")
    public int add(int a, int b) {
        return a+b;
    }
    @LogExecution(logParameters = true)
    public int multiply(int a, int b) {
        return a*b;
    }
    public int subtract(int a, int b) {
        return a-b;
    }
}
```

iii) Processing the Annotation at runtime:

```
Import java.lang.reflect.*;
public class AnnotationProcessor {
    public static void process(Object
        obj) {
        Class<?> clazz = obj.getClass();
        for(Method method : clazz.get
            DeclaredMethods()) {
            if(method.isAnnotationPresent
                (LogExecution.class)) {
                LogExecution Annotation =
                    method.getAnnotation
                    (LogExecution.class);
                String message = annotation
                    value();
                boolean logParams = annotation
                    logParameters();
```

```
try {
    System.out.println("[LOG] " + message);
    if (logParams) {
        System.out.println("Method: " +
                method.getName());
        for (Parameter param: method.get
                Parameters()) {
            System.out.println("Param: " +
                    param.getName() + " (" + param.getType().get
                    SimpleName() + ")");
        }
    }
} catch (Exception e) {
    e.printStackTrace();
}
}
}
}
}
public static void main(String[] args) {
```

```
        Calculator calculator = new Calculator();
        process(calculator);
    }
}
```

Output:

```
[LOG] Adding two numbers
[LOG] Method executed
    Method : multiply
        Param: arg0 (int)
        Param: arg1 (int)
```

11) Design the singleton design pattern in Java? What problem does it solve and how does it ensure only one instance of a class is created? Extend your answer to explain how thread safety can be achieved in a singleton implementation.

Ans. Singleton pattern is a creational design pattern that ensure a class has only one instance and provides a global point of access

following

to that instance. It solves ~~two~~ main problems:

1) Controlled Access to a Single Instance: Ensures that all clients use the same instance of a class.

2) Global Access Point: Provides a well-known access point to the instance.

3) Caching: When we need a centralized cache system.

4) Thread pools: Managing a pool of worker threads.

5) Resource management: When we need a shared resource (database connection, logger)

Thread-Safe Singleton Approaches:

a) Synchronized method:

```
class Singleton{
    public static Singleton instance;
    private Singleton(){}
```

```
public static synchronized Singleton
    getInstance() {
        if(instance==null){
            instance=new Singleton();
        }
        return instance;
    }
}
```

Synchronized ensures only one thread can access the method at a time.

b) Double Checked Locking (Efficient):

```
class Singleton{
    Private static volatile Singleton
        instance;
    private Singleton(){}
    public static Singleton get-
        Instance(){
        if(instance==null){
```

```
            instance = new Singleton();
        }
    }
    return instance;
    }
}
```

Here, first check avoids locking once the instance is initialized. Second check ensures only one instance is created.

c) Bill Pugh (Inner Static Helper Class):

```
class Singleton {
    private Singleton() {}
    private static class Holder {
        private static final Singleton
            INSTANCE = new Singleton();
    }
    public static Singleton getInstance()
    {
        return Holder.INSTANCE;
    }
}
```

This method ensures high performance b
lazy loading.

d) Enum Singleton (Simplest and Thread safe)

```
enum Singleton {
    INSTANCE;
    public void showMessage() {
        System.out.println("Hello fro
            Singleton!");
    }
}
```

Enum singletons are thread-safe, serializat
safe and easy to implement. JVM guaran-
tees that enum constants are instantiated
only once.

12) Describe how JDBC manages communication
between a Java application and relational data
base. Outline the step involved in executing a

a SELECT query and fetching results. Include error handling with try-catch and finally blocks. [Lab 4]

Ans. JDBC (Java Database Connectivity) provides a standard API for Java Applications to interact with relational databases. It acts as a bridge that translates Java method calls into database-specific operations.

Steps:

1) Load JDBC Driver: class.forName() is often still used for backward compatibility.

2) Establish connection: DriverManager.getConnection() creates a physical connection.
   - Connection String format: mysql://hostname:port/database,

3) Create statement: connection.createStatement() create a basic statement object

4) Execute query: Use executeQuery() for SELECT statements, and executeUpdate() for INSERT,

DELETE on UPDATE.

5) Process Results : Resultset maintain a cursor pointing to current row. next() moves cursor and returns false when no more rows.

6) Close Resources: It is closed in reverse order as:

Resultset → Statement → Connection

Execution of code:

```
import java.sql.*;
public class JdbcSelectExample {
    private static final String URL =
        "jdbc:mysql://localhost:3306/
            mydatabase");
    private static final String USER = "root";
    private static final String PASSWORD =
        "1234";
    public static void main (String[]
        args)
```

```
Connection connection=null;
Statement statement=null;
ResultSet resultSet=null;

try{
  Class.forName("com.mysql.cj.jdbc.Driven")
  DrivenManager.getConnection(URL,USER,
                PASSWORD);

  try{ statement=connection.createStateme
  nt();

String sqlQuery="SELECT id,name,email
                FROM users";

resultSet=statement.executeQuery
                (sqlQuery);

  while(resultSet.next()){
    int id=resultSet.getInt("id");
    String name=resultSet.get
               String("name");

    String email=resultSet.get
               String("email");
```

```
System.out.printf("ID: %d, Name: %s,
        Email: %s %n", id, name, email);
}

class (classNotFoundException e) {
    System.err.println("JDBC Driver
            not found: " + e.getMessage());
}
catch(SQLException e) {
    System.err.println("Database
            error: " + e.getMessage());
    e.printStackTrace();
}
finally {
    try {
        if(resultSet!=null)
            resultSet.close();
        if(statement!=null)
            statement.close();
        if(connection!=null)
            connection.close();
    }
```

```
        catch(SQLException e){
            System.err.println("Error closing resources
                : "+e.getMessage());
        }
    }
}
}
```

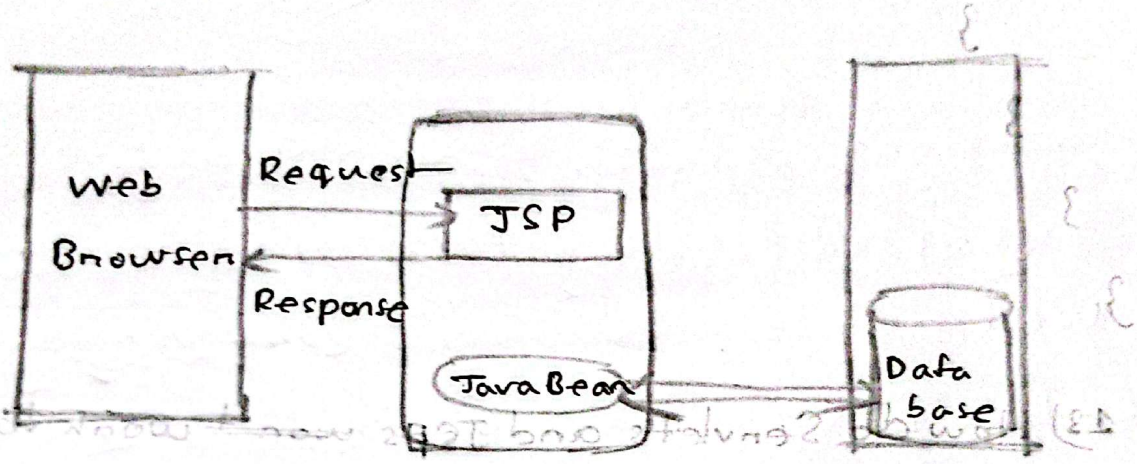
13) How do Servlets and JSPs ~~won't~~ work together in a web application following MVC architecture? Provide a brief use case showing servlet as a controller, JSP as a view and Java class as the model.

Ans. In Java web applications, Servlets and JSPs work together following MVC (Model-View-Controller) patterns:

1) Controller: Servlet handles logic and flow control.

2) View: JSP handles presentation

3) Model: Plain Java Class (POJO) that handles data and business logic.



Example:

i) Model (~~User.java~~ Data):

```
public class User {
    private String name;
    public User(String name) {
        this.name = name;
    public String getName() {
        return name;
    }
}
```

:) Controller (Servlet):

```
@WebServlet("\hello")
public class HelloServlet extends Http
        Servlet{
    protected void doGet(HttpServletRequest request, HttpServletResponse
            response) throws ServletException, IOException {
        User user = new User("Atif");
        request.setAttribute("user",
                user);
        request.getRequestDispatcher
                ("hello.jsp");
    }
}
```

::) View (JSP)

```
<!--- hello.jsp -->
<html>
<body>
```

<h1>Hello ${user.name}!</h1>

</body>

</html>

14) Explain life cycle of a Java Servlet. What are the roles of the init(), service(), and destroy() methods? Discuss how servlets handle concurrent requests and how thread safety issues may arise.

Ans. The Java Servlet lifecycle is managed through servlet container (e.g Tomcat, Jetty) and consists of three main phases:

1) Initialization
2) Request handling
3) Destruction

Roles of methods are:

1) init(ServletConfig config): It is called once when servlet is first loaded into memory. It is used for one-time initialization tasks

(e.g database connections) and receives a Servlet Config object containing servlet configuration parameters.

2) service(ServletRequest req, ServletResponse res): It handles each client request. Typically not overridden directly - the container calls this which then delegates to doGet(), doPost() etc.

3) destroy(): It is called when the servlet is being taken out of service. This method is used for cleanup activities (e.g closing database connections). All threads exit or timed out before this is called.

Handling concurrent requests:

i) The servlet container maintains a thread pool.

ii) For each incoming request, the container:

- Allocates a thread from the pool.

- Calls the servlets service() method in that thread.
- Returns thread to the pool when the response is completed.

Multiple threads can execute a servlet's method simultaneously, thread safety issue may arise:

1) Instance variables: All threads share the same servlet instance, so instance variables are shared

2) ServletContext attributes: Shared across all servlets in web application

3) Session attributes: Shared across requests from same client session

15) What problems can occur if shared resources (e.g variables) are accessed by multiple threads? Illustrate your answer with an example and suggest solution using synchronization

Ans: Since a single servlet instance handles multiple requests via threads, unsynchronized access to shared resources can lead to:

i) Race conditions: Threads interfering with each other.

ii) Data corruption: Give inconsistent or wrong values.

iii) Thread interference: Shows unpredictable behaviour.

Solution by synchronization:

```
public class SafeCounterServlet extends
        HttpServlet{
    private int count=0;
    protected void doGet (HttpServletRequest
        req, HttpServletResponse res) {
        synchronized (this);
        res.getWriter().write("Count:"+
                count);
    }
}
```

Here, this synchronization ensures only one thread executes the block at a time. It prevents race conditions but reduces throughput.

16) Describe how the MVC pattern seperates concerns in a Java web application? Explain the advantage of this structure in terms of maintainability and scalability, using student registration system as an example.

Ans. MVC (Model-View-Controller) seperates concerns in a Java web application by dividing into three components:

1) Model: It represents the applications data and business logic. In student registration system this would include:

i) Student entity classes (name, ID, course)

ii) Database operations (save/retrieve student records)

iii) Business rules (eligibility check, GPA calculations).

2) View: Handles presentation & user interface.

In example:

i) JSP/Thymeleaf showing registration forms.

ii) HTML tables displaying student lists.

iii) Confirmation messages after registration

3) Controller: Mediates between model and view.

For student registration:

i) Receives form submissions.

ii) Validates input

iii) Calls appropriate model methods.

iv) Selects which view to display.

Advantages for maintainability:

i) Isolated changes: Switch to JSP to angular without modifying registration rules.

ii) Easy debugging: Issue cleanly belong to one component.

iii) Code organization: Frontend works on views and backend on model.

Advantages for Scalability:

i) Component Independence: It scale views separately (CDN) for static resources.

ii) Reusability: Same model can serve:

- Web Interface
- Mobile App
- API customers

iii) Parallel Development:

- Designing view menus
- Enhancing controller logic
- Optimizing model performance

Example

```
//Model
class Student{
    String id,name;
}
//Controller
@WebServlet("/reg")
```

```
class RegServlet extends HttpServlet {
    List<Student> students = new ArrayList
                              <>();
    protected void doPost(HttpServletRequest req,
                          HttpServletResponse res) {
        Student s = new Student();
        s.id = req.getParameter("id");
        s.name = req.getParameter("name");
        students.add(s);
        res.sendRedirect("view.jsp");
    }
}

//View(view.jsp)
<html>
${students.forEach(s -> out.println
                   (s.id + " " + s.name))}
</html>
```

17) In a Java EE application, how does a servlet controller manage the flow between model and view? Provide a brief example that demonstrate forwarding data from a servlet to JSP and rendering a response. [Lab-5]

Ans. A servlet controller manage the flow between model and view in the following way:

1) Receives request: Handles HTTP request (GET/POST)

2) Processes Data: Interact with model (service / DB)

3) Forwards to View: Sends data to JSP for rendering.

Program:

```
// Controller
@WebServlet
public class StudentServlet extends
    HttpServlet {
```