1. **Concurrent Patterns in Java:**
   * **Question:** Imagine you are developing a web server that handles multiple client requests concurrently. Describe how you would implement this using Java's concurrency utilities. What pattern would you use, and why? Provide a code snippet to illustrate your approach.
2. **Concurrent Collections:**
   * **Question:** You have a multi-threaded application where threads frequently read from and update a shared map. Which concurrent collection would you choose to ensure thread safety and high performance? Justify your choice with an example of how you would implement this.
3. **Lambda Expressions:**
   * **Question:** Given a list of **Employee** objects, each having properties **id**, **name**, and **salary**, write a lambda expression to filter out employees with a salary greater than $50,000 and sort the remaining employees by name. Discuss how lambda expressions improve the readability and maintainability of this code.
4. **Stream API:**
   * **Question:** Using the Stream API, write a code snippet to process a list of transactions where each transaction has an **id**, **type**, and **amount**. Filter out transactions of type "debit" with an amount greater than $1000, map them to their IDs, and collect the result into a list. Explain the benefits of using the Stream API for this task.
5. **Introduction to Design Pattern:**
   * **Question:** You need to ensure that only one instance of a **DatabaseConnection** class is created throughout your application. Explain which design pattern you would use and provide a thread-safe implementation in Java. Additionally, discuss potential issues you might encounter with this pattern and how you would address them.
6. **GitHub:**
   * **Question:** Describe a scenario where you are working on a project with multiple team members, and you need to implement a new feature without affecting the main codebase. Explain how you would use GitHub's branching and pull request features to manage this workflow. Include the specific Git commands you would use.
7. **Introduction to JDBC:**
   * **Question:** Consider a scenario where you need to connect to a PostgreSQL database, execute a query to retrieve a list of users, and handle potential SQL exceptions gracefully. Write a code snippet to demonstrate this and discuss the importance of proper exception handling in JDBC.
8. **Statement vs. PreparedStatement:**
   * **Question:** You have identified a SQL injection vulnerability in your application caused by using a **Statement** object. Refactor the following code to use **PreparedStatement** to prevent SQL injection and explain how this change improves security.

String query = "SELECT \* FROM users WHERE username = '" + username + "' AND password = '" + password + "'";

Statement stmt = connection.createStatement();

ResultSet rs = stmt.executeQuery(query);

1. **ResultSet:**
   * **Question:** Given a **ResultSet** object containing user data, write a method that iterates through the **ResultSet** and prints each user's details. Additionally, explain how you would handle any SQL exceptions that might occur during this process to ensure robust error handling.
2. **Combining Concepts:**
   * **Question:** Design a small Java application that uses JDBC to connect to a database and retrieve product data, processes the data using the Stream API to filter and sort products by price, and then prints the results. Explain how you would structure your code to ensure it is modular, maintainable, and easy to test. Provide relevant code snippets.

🡨Answers🡪

1. **Concurrent Patterns in Java:**

* **Answer:** To handle multiple client requests concurrently, you can use the **ExecutorService** with a cached thread pool. This pattern is effective because it can dynamically adjust the number of threads according to the load, providing efficient resource utilization.

import java.io.IOException;

import java.net.ServerSocket;

import java.net.Socket;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class WebServer {

    private static final int PORT = 8080;

    private static final ExecutorService threadPool = Executors.newCachedThreadPool();

    public static void main(String[] args) throws IOException {

        ServerSocket serverSocket = new ServerSocket(PORT);

        while (true) {

            Socket clientSocket = serverSocket.accept();

            threadPool.submit(() -> handleRequest(clientSocket));

        }

    }

    private static void handleRequest(Socket clientSocket) {

        // Handle client request

    }

}

* **Concurrent Collections:**
* **Answer:** For a multi-threaded application requiring frequent reads and updates to a shared map, **ConcurrentHashMap** is the preferred choice. It allows concurrent read and update operations without locking the entire map, providing better performance than **Collections.synchronizedMap**.
* import java.util.Map;
* import java.util.concurrent.ConcurrentHashMap;
* public class ConcurrentMapExample {
* private static final Map<String, String> concurrentMap = new ConcurrentHashMap<>();
* public static void main(String[] args) {
* concurrentMap.put("1", "One");
* concurrentMap.put("2", "Two");
* Runnable task = () -> {
* for (Map.Entry<String, String> entry : concurrentMap.entrySet()) {
* System.out.println(entry.getKey() + ": " + entry.getValue());
* }
* };
* Thread thread1 = new Thread(task);
* Thread thread2 = new Thread(task);
* thread1.start();
* thread2.start();
* }
* }
* **Lambda Expressions:**
* **Answer:** Lambda expressions enhance code readability and maintainability by reducing boilerplate code. They allow you to express instances of single-method interfaces (functional interfaces) more concisely.
* import java.util.Arrays;
* import java.util.List;
* import java.util.stream.Collectors;
* class Employee {
* int id;
* String name;
* double salary;
* Employee(int id, String name, double salary) {
* this.id = id;
* this.name = name;
* this.salary = salary;
* }
* @Override
* public String toString() {
* return "Employee{" +
* "id=" + id +
* ", name='" + name + '\'' +
* ", salary=" + salary +
* '}';
* }
* }
* public class LambdaExample {
* public static void main(String[] args) {
* List<Employee> employees = Arrays.asList(
* new Employee(1, "John", 60000),
* new Employee(2, "Jane", 50000),
* new Employee(3, "Jack", 45000)
* );
* List<Employee> filteredSortedEmployees = employees.stream()
* .filter(e -> e.salary > 50000)
* .sorted((e1, e2) -> e1.name.compareTo(e2.name))
* .collect(Collectors.toList());
* filteredSortedEmployees.forEach(System.out::println);
* }
* }
* **Stream API:**
* **Answer:** The Stream API simplifies data processing tasks by providing a high-level abstraction for operations on collections. It allows you to chain multiple operations together in a declarative way.
* import java.util.Arrays;
* import java.util.List;
* import java.util.stream.Collectors;
* class Transaction {
* int id;
* String type;
* double amount;
* Transaction(int id, String type, double amount) {
* this.id = id;
* this.type = type;
* this.amount = amount;
* }
* public int getId() {
* return id;
* }
* public String getType() {
* return type;
* }
* public double getAmount() {
* return amount;
* }
* }
* public class StreamAPIExample {
* public static void main(String[] args) {
* List<Transaction> transactions = Arrays.asList(
* new Transaction(1, "debit", 1200),
* new Transaction(2, "credit", 800),
* new Transaction(3, "debit", 500)
* );
* List<Integer> debitTransactionIds = transactions.stream()
* .filter(t -> "debit".equals(t.getType()) && t.getAmount() > 1000)
* .map(Transaction::getId)
* .collect(Collectors.toList());
* debitTransactionIds.forEach(System.out::println);
* }
* }
* **Introduction to Design Pattern:**
* **Answer:** The Singleton pattern ensures that only one instance of a class is created. This is useful for managing shared resources like database connections. A thread-safe implementation can be achieved using double-checked locking.
* public class DatabaseConnection {
* private static volatile DatabaseConnection instance;
* private DatabaseConnection() {
* // Private constructor to prevent instantiation
* }
* public static DatabaseConnection getInstance() {
* if (instance == null) {
* synchronized (DatabaseConnection.class) {
* if (instance == null) {
* instance = new DatabaseConnection();
* }
* }
* }
* return instance;
* }
* }
* **GitHub:**
* **Answer:** In a collaborative project, you can use feature branches to isolate work on new features. This prevents conflicts with the main codebase and allows for focused development and review.

# Create a new feature branch

git checkout -b feature-branch

# Work on the feature and commit changes

git add .

git commit -m "Add new feature"

# Push the branch to GitHub

git push origin feature-branch

# Create a pull request on GitHub and merge after review

# Delete the feature branch locally and remotely after merge

git branch -d feature-branch

git push origin --delete feature-branch

* **Introduction to JDBC:**
* **Answer:** To connect to a PostgreSQL database, execute a query, and handle SQL exceptions, you can use the following code snippet:
* import java.sql.Connection;
* import java.sql.DriverManager;
* import java.sql.PreparedStatement;
* import java.sql.ResultSet;
* import java.sql.SQLException;
* public class JDBCConnectionExample {
* public static void main(String[] args) {
* String url = "jdbc:postgresql://localhost:5432/mydatabase";
* String username = "postgres";
* String password = "password";
* try (Connection connection = DriverManager.getConnection(url, username, password);
* PreparedStatement preparedStatement = connection.prepareStatement("SELECT \* FROM users");
* ResultSet resultSet = preparedStatement.executeQuery()) {
* while (resultSet.next()) {
* System.out.println("User: " + resultSet.getString("username"));
* }
* } catch (SQLException e) {
* e.printStackTrace();
* }
* }
* }
* **Statement vs. PreparedStatement:**
* **Answer:** Using **PreparedStatement** prevents SQL injection by parameterizing the query. Here's the refactored code:
* String query = "SELECT \* FROM users WHERE username = ? AND password = ?";
* try (PreparedStatement pstmt = connection.prepareStatement(query)) {
* pstmt.setString(1, username);
* pstmt.setString(2, password);
* ResultSet rs = pstmt.executeQuery();
* }

By using **PreparedStatement**, the parameters are properly escaped, preventing SQL injection attacks.

* **ResultSet:**
* **Answer:** To iterate over a **ResultSet** and print user details, you can use the following method:
* import java.sql.Connection;
* import java.sql.DriverManager;
* import java.sql.PreparedStatement;
* import java.sql.ResultSet;
* import java.sql.SQLException;
* public class ResultSetExample {
* public static void main(String[] args) {
* String url = "jdbc:mysql://localhost:3306/mydatabase";
* String username = "root";
* String password = "password";
* String query = "SELECT \* FROM users";
* try (Connection connection = DriverManager.getConnection(url, username, password);
* PreparedStatement preparedStatement = connection.prepareStatement(query);
* ResultSet resultSet = preparedStatement.executeQuery()) {
* while (resultSet.next()) {
* int id = resultSet.getInt("id");
* String name = resultSet.getString("name");
* String email = resultSet.getString("email");
* System.out.println("ID: " + id + ", Name: " + name + ", Email: " + email);
* }
* } catch (SQLException e) {
* e.printStackTrace();
* }
* }
* }

1. **Combining Concepts:**
   * **Answer:** To design a small Java application that connects to a database, retrieves product data, processes it using the Stream API, and prints the results, you can structure your code into separate classes for database connection, data retrieval, and processing. This ensures modularity, maintainability, and ease of testing.

**Code Structure**

1. **DatabaseConnection.java**: Handles database connection.
2. **Product.java**: Represents the product entity.
3. **ProductRepository.java**: Retrieves product data from the database.
4. **ProductService.java**: Processes the product data using the Stream API.
5. **Main.java**: Main application class to run the code.

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

public class DatabaseConnection {

    private static final String URL = "jdbc:mysql://localhost:3306/mydatabase";

    private static final String USER = "root";

    private static final String PASSWORD = "password";

    public static Connection getConnection() throws SQLException {

        return DriverManager.getConnection(URL, USER, PASSWORD);

    }

}

public class Product {

    private int id;

    private String name;

    private double price;

    public Product(int id, String name, double price) {

        this.id = id;

        this.name = name;

        this.price = price;

    }

    public int getId() {

        return id;

    }

    public String getName() {

        return name;

    }

    public double getPrice() {

        return price;

    }

    @Override

    public String toString() {

        return "Product{" +

                "id=" + id +

                ", name='" + name + '\'' +

                ", price=" + price +

                '}';

    }

}

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.util.ArrayList;

import java.util.List;

public class ProductRepository {

    public List<Product> getAllProducts() {

        List<Product> products = new ArrayList<>();

        String query = "SELECT \* FROM products";

        try (Connection connection = DatabaseConnection.getConnection();

             PreparedStatement preparedStatement = connection.prepareStatement(query);

             ResultSet resultSet = preparedStatement.executeQuery()) {

            while (resultSet.next()) {

                int id = resultSet.getInt("id");

                String name = resultSet.getString("name");

                double price = resultSet.getDouble("price");

                products.add(new Product(id, name, price));

            }

        } catch (SQLException e) {

            e.printStackTrace();

        }

        return products;

    }

}

import java.util.List;

import java.util.stream.Collectors;

public class ProductService {

    private ProductRepository productRepository;

    public ProductService(ProductRepository productRepository) {

        this.productRepository = productRepository;

    }

    public List<Product> filterAndSortProducts(double priceThreshold) {

        return productRepository.getAllProducts().stream()

                .filter(product -> product.getPrice() > priceThreshold)

                .sorted((p1, p2) -> Double.compare(p1.getPrice(), p2.getPrice()))

                .collect(Collectors.toList());

    }

}

import java.util.List;

public class Main {

    public static void main(String[] args) {

        ProductRepository productRepository = new ProductRepository();

        ProductService productService = new ProductService(productRepository);

        double priceThreshold = 50.0;

        List<Product> filteredSortedProducts = productService.filterAndSortProducts(priceThreshold);

        filteredSortedProducts.forEach(System.out::println);

    }

}

**Explanation:**

* **Modularity**: The code is divided into separate classes, each handling a specific responsibility (e.g., database connection, data retrieval, data processing).
* **Maintainability**: Each class is focused on a single responsibility, making the code easier to understand and modify.
* **Ease of Testing**: You can write unit tests for each class independently. For example, you can mock the database connection to test the **ProductRepository** and use a sample list of products to test the **ProductService**.

By structuring the code in this way, you ensure that each component is isolated and can be developed, tested, and maintained independently. This approach adheres to the principles of clean code and design patterns, promoting better software engineering practices.