Ques: - Design Pattern

Ans:-

Design patterns are categorized into three main types:

|  |  |  |
| --- | --- | --- |
| **Type** | **Purpose** | **Example** |
| **Creational** | Deal with object creation efficiently | Singleton, Factory, Builder, Prototype |
| **Structural** | Define relationships between classes | Adapter, Decorator, Composite, Proxy |
| **Behavioral** | Manage object interaction and responsibilities | Observer, Strategy, Command, Chain of Responsibility |

1. **Creational Patterns (How Objects Are Created)**

* **Singleton** → Ensures a class has only **one instance**.

public class **Singleton {**

private static **Singleton instance;**

**private Singleton() {}**

public static Singleton **getInstance() {**

if (instance == null) {

instance = new Singleton();

}

return instance;

}

**}**

* **Factory Method** → Creates objects without exposing instantiation logic.

interface **Vehicle {**

**void drive();**

**}**

class Carimplements **Vehicle {**

**public void drive() {**

System.out.println("Driving a car");

**}**

**}**

class **VehicleFactory {**

public static **Vehicle** createVehicle**() {**

return **new Car();**

**}**

**}**

* **Builder** → Step-by-step object creation (useful for complex objects).

public class **Car {**

private String **engine;**

private int **wheels;**

public static class **Builder {**

private String **engine;**

private int **wheels;**

public **Builder** setEngine(String engine) {

this.engine = engine; return this;

}

public **Builder** setWheels(int wheels) {

this.wheels = wheels; return this;

}

public **Car build() {**

**return new Car(this);**

**}**

**}**

**private Car(Builder builder) {**

this.engine = builder.engine;

this.wheels = builder.wheels;

**}**

**}**

1. **Structural Patterns (How Classes Are Organized)**

* **Adapter** → Converts one interface to another

class **OldSystem {**

void oldMethod() {

System.out.println("Old System Method");

}

**}**

interface **NewSystem {**

void **newMethod**();

**}**

class **Adapter** implements **NewSystem {**

private OldSystem oldSystem = new OldSystem();

public void **newMethod() {**

oldSystem.oldMethod();

**}**

**}**

* **Decorator** → Adds new behavior dynamically without modifying the original class

interface **Coffee {**

double **cost();**

**}**

class **SimpleCoffee** implements **Coffee {**

public double **cost() {**

return **5;**

**}**

**}**

class **MilkDecorator** implements **Coffee {**

private **Coffee** coffee;

**public MilkDecorator(Coffee** coffee**) {**

this.coffee = coffee;

**}**

public double **cost() {**

return **coffee.cost() + 2;**

**}**

**}**

**Ques: - State of circuit breaker**

**Ans:-**

|  |  |
| --- | --- |
| **State** | **Behavior** |
| **Closed** | Calls the service normally. |
| **Open** | Immediately returns the **fallback response**. |
| |  | | --- | | **Half-Open** |  |  | | --- | |  | | Tests a few requests before deciding to close or reopen. |

**resilience4j:**

**circuitbreaker:**

**instances:**

**myService:**

**failureRateThreshold:** 50 **#** 50% failure rate triggers **open state**

**waitDurationInOpenState:** 10s# Wait for **10s** before trying again

**permittedNumberOfCallsInHalfOpenState:** 3# Allow **3 test** calls in **Half-Open**

**slidingWindowSize:** 5 **#** Check **last 5** requests

**minimumNumberOfCalls:** 5 **#** Minimum calls before checking failure rate

**State Transition Flow**

1. **Closed → Open**: If failures exceed threshold.
2. **Open → Half-Open**: After the wait time.
3. **Half-Open → Closed**: If recovery is successful.
4. **Half-Open → Open**: If failures persist.

**Ques:- @Primary vs @Qualifier**

**Ans:-**

|  |  |  |
| --- | --- | --- |
| **Feature** | **@ Primary** | **@ Qualifier** |
| **Usage** | Marks a bean as the default | Selects a specific bean by name |
| **Scope** | Global (applies to all injections) | Local (applies only to the specific injection) |
| **Override Behavior** | Can be overridden by @Qualifier | Overrides @Primary when used |
| **Best Use Case** | When one bean is the **most commonly used** default | When you need **different beans in different places** |

@Component

public class EmailNotificationService implements **NotificationService** {

@Override

public void **sendNotification**(String message) {

System.out.println("Email Notification: " + message);

}

}

@Component

public class SmsNotificationService implements **NotificationService** {

@Override

public void **sendNotification**(String message) {

System.out.println("SMS Notification: " + message);

}

}

* **Injection with @Qualifier**

@Service

public class NotificationManager {

private final **NotificationService** notificationService;

@Autowired

public NotificationManager**(@Qualifier**("**emailNotificationService**") NotificationService notificationService) {

this.notificationService = notificationService;

}

public void notifyUser(String message) {

notificationService.sendNotification(message);

}

}

* **Using @Primary**

@Component

public class EmailNotificationService implements NotificationService {

@Override

public void sendNotification(String message) {

System.out.println("Email Notification: " + message);

}

}

@Component

@Primary // This bean will be used as the default

public class SmsNotificationService implements NotificationService {

@Override

public void sendNotification(String message) {

System.out.println("SMS Notification: " + message);

}

}

@Service

public class NotificationManager {

private final NotificationService notificationService;

@Autowired

public NotificationManager(NotificationService notificationService) {

this.notificationService = notificationService;

}

public void notifyUser(String message) {

notificationService.sendNotification(message);

}

}

* **SmsNotificationService because it is marked as @Primary.**

==================================================================================

**Ques:- Put vs Patch**

**Ans:-**

**Implementation-Level Difference**

If your service layer **intelligently merges** existing data in PUT, it can behave similarly to PATCH, but **strictly speaking**:

* **PUT is expected to replace the entire entity** unless handled otherwise.
* **PATCH is expected to modify only given fields**, which means you can even use alternative formats like JSON Merge Patch or JSON Patch (op: replace, op: remove, etc.).

#### **Example Where** PATCH **is More Flexible**

With PATCH, you can even send an operation-based update:

**[**

**{ "op": "replace", "path": "/name", "value": "John Smith" },**

**{ "op": "remove", "path": "/age" }**

**]**

**Note**: - This **is not possible** with PUT.

==================================================================================

**Ques: - IntStreamRange Vs IntStream.rangeClosed**

**Ans:-**

1. **IntStreamRange**

* IntStream.range(start, end) is a **Java 8 Stream API** method used to generate a sequence of numbers in a given range.
* Basic Syntax

**IntStream.range(start, end)**

* start → Inclusive (included)
* end → Exclusive (not included)
* Returns an **IntStream** of numbers from **start to end – 1**
* **Difference Between IntStream.range() and IntStream.rangeClosed()**

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Range** | **Example (1 to 5)** | **Includes end?** |
| **IntStream.range(start, end)** | [start, end-1] | 1, 2, 3, 4 | **No** |
| **IntStream.rangeClosed(start, end)** | [start, end] | 1, 2, 3, 4, 5 | **Yes** |

==================================================================================

**Ques:- String vs StringBuffer vs StringBuilder**

**Ans:-**

* **String**
* A String is **immutable**, meaning its value **cannot be changed** once created. Any modification creates a **new object** in memory.

public class StringExample {

public static void main(String[] args) {

String s1 = "Hello";

s1 = s1 + " World"; **// Creates a new String object**

System.out.println(s1); **// Output: Hello World**

}

}

* **Why is String Immutable?**
* Security (e.g., for passwords, class loading, etc.).
* Performance optimization (used in **String Pool**).
* Thread-safety (multiple threads can safely use the same String).
* **When to Use String?**
* When you don't need to modify the string frequently.
* When working with constants (e.g., "hello").
* When security and memory optimization are priorities.

1. **StringBuffer (Mutable & Thread-Safe)**

* StringBuffer is **mutable**, meaning its content can be changed **without creating a new object**.
* It is **thread-safe**, meaning it is synchronized for use in multi-threaded environments.

public class StringBufferExample {

public static void main(String[] args) {

StringBuffer sb = new StringBuffer("Hello");

sb.append(" World"); // Modifies existing object

System.out.println(sb); // Output: Hello World

}

}

1. **StringBuilder (Mutable & Faster, Not Thread-Safe)**

* StringBuilder is like StringBuffer but **not thread-safe**.
* It is **faster** than StringBuffer because it **does not use synchronization**.

### ****When to Use**** StringBuilder****?****

* When string modifications are needed **without multi-threading**.
* When performance is critical and synchronization is unnecessary.

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **String** | **StringBuffer** | **StringBuilder** |
| **Mutability** | Immutable | Mutable | Mutable |
| **Thread-Safe?** | Yes | Yes | No |
| **Performance** | Slow (New Object on Modification) | Medium | Fastest |
| **Usage** | Constants, Security | Multi-threading | Single-threaded, Fast Modifications |

**Ques:- @RequestHeader and @RequestAttribute**

**Ans:-**

1. **@RequestHeader**

* The @RequestHeader annotation is used to retrieve specific headers from an HTTP request.

@RestController

@RequestMapping("/api")

public class HeaderController {

@GetMapping("/greet")

public ResponseEntity<String> greetUser**(@RequestHeader("User-Agent")** String userAgent) {

return ResponseEntity.ok("Your User-Agent is: " + userAgent);

}

}

* The @RequestHeader("User-Agent") extracts the User-Agent header from the request.
* If the request does not include the header, an error occurs unless you provide a default value.
* You can make the header optional using required = false

**@RequestHeader(value = "User-Agent", required = false, defaultValue = "Unknown") String userAgent**

1. **@RequestAttribute**

* The **@RequestAttribute** annotation is used to access **request attributes** stored in the **HttpServletRequest** scope.
* These attributes are set within the same request lifecycle.

@RestController

@RequestMapping("/api")

public class AttributeController {

@GetMapping("/process")

public ResponseEntity<String> processRequest(HttpServletRequest request) {

request.setAttribute("transactionId", "TXN12345");

return ResponseEntity.ok("Transaction set");

}

@GetMapping("/fetch")

public ResponseEntity<String> fetchTransaction**(@RequestAttribute("transactionId") String transactionId**) {

return ResponseEntity.ok("Fetched Transaction ID: " + transactionId);

}

}

* The first endpoint (/process) sets an attribute transactionId in the request.
* The second endpoint (/fetch) retrieves the transactionId using @RequestAttribute.
* **When to Use What?**
* Use @RequestHeader when you need metadata about the request (like authentication tokens, user-agent info, etc.).
* Use @RequestAttribute when you need temporary request-scoped data set by another part of the application.

**Ques:- HashTable**

Ans:-

* A **hash table** stores data in an **array-like structure**, but instead of using numeric indexes (like arrays), it uses **keys** to compute an index using a **hash function.**

**How It Works**

1. The key is passed through a **hash function**.
2. The hash function returns an **index (bucket)** where the value is stored.
3. When retrieving, the same hash function is used to find the index.

**Key → Hash Function → Index → Value Stored**

"John" → hash("John") → 2 → "Engineer"

"Jane" → hash("Jane") → 5 → "Doctor"

**Ques:- Java passByValue or passByRefrence.**

**Ans:-**

* **Note:-** Java **always** uses **pass-by-value**.

## **Understanding Pass-by-Value**

In **pass-by-value**, a copy of the variable is passed to a method, meaning:

* **For primitives** → The actual value is copied.
* **For objects** → The reference (memory address) is copied, not the actual object.

🚨 **Java does NOT support pass-by-reference**, where a method can modify the original variable.

**Example: Pass-by-Value with Primitives**

public class PassByValueExample {

static void modify(int num) {

num = 10**; // Changes only the local copy**

}

public static void main(String[] args) {

int x = 5;

modify(x);

System.out.println(x**); // Output: 5 (Original value remains unchanged)**

}

}

* **The value of x remains 5 because modify() only changes a copy**

## **2. Pass-by-Value with Objects (Reference Variables)**

For objects, **the reference (memory address) is copied**, **not** the actual object itself.

* If you **change the object’s internal data**, the changes **persist** (because both references point to the same object).
* If you **reassign the reference**, the original object remains unchanged.

**class Car {**

**String model;**

**}**

public class PassByValueObjects {

static void modify(Car c) {

**c.model = "Tesla"; // Modifies the actual object**

}

public static void main(String[] args) {

**Car myCar = new Car();**

**myCar.model = "Toyota";**

**modify(myCar);**

**System.out.println(myCar.model); // Output: Tesla (Changed)**

}

}

* Since **both references point to the same object**, modifying c.model also affects myCar.model.

### ****Example: Reassigning Object Reference****

If we reassign the reference inside a method, the original object remains **unchanged**.

class Car {

String model;

}

public class PassByValueObjects {

static void modify(Car c) {

c = new Car(); // Creates a new object (does not affect original)

c.model = "Tesla";

}

public static void main(String[] args) {

Car myCar = new Car();

myCar.model = "Toyota";

modify(myCar);

System.out.println(myCar.model); // Output: Toyota (Unchanged)

}

}

* The **new object is created inside modify()**, but the original reference (myCar) is **not affected**.

**Ques:- Select 2nd highest salary of employee using sql.**

Ans:- Select salary

From ( select salary, DENSE\_RANK() OVER( ORDER By salary desc) AS rnk

From Emloyee

)

Where rnk =2 group by salary;

**Ques:- Can an Abstract Class Have a Constructor?**

**Ans:-**

* Yes, **an abstract class can have a constructor** in Java.
* Even though an **abstract class cannot be instantiated**, its **constructor is called when a subclass object is created**.
* **Why Does an Abstract Class Need a Constructor?**
* **To Initialize Common Fields** – If multiple subclasses share some common properties, the constructor initializes them.
* **To Enforce Initialization Logic** – Ensures that some logic (e.g., database connection setup) runs for every subclass.
* **To Call the Parent Constructor (super())** – Helps in maintaining a proper inheritance chain.

**abstract** class Animal {

String name;

**// Constructor in abstract class**

Animal(String name) {

this.name = name;

System.out.println("Animal Constructor Called: " + name);

}

**// Abstract method**

abstract void makeSound();

}

**// Subclass**

class Dog **extends Animal** {

Dog(String name) {

**super(name); // Calls the abstract class constructor**

System.out.println("Dog Constructor Called");

}

@Override

void makeSound() {

System.out.println(name + " barks!");

}

}

// Main class

public class Main {

public static void main(String[] args) {

Dog dog = **new Dog("Buddy"); // Creates an instance of Dog**

dog.makeSound();

}

}

* **Key Points:**
* **An abstract class can have a constructor** to initialize fields.
* **It is called when a subclass object is created** using super().
* **Even if the abstract class cannot be instantiated, its constructor is still executed** when a subclass is instantiated.