CREATIONAL & STRUCTURAL DESIGN PATTERN IN JAVA:-

Design patterns are common solutions to recurring problems in software design. They are categorized into three main types: **Creational**, **Structural**, and **Behavioral**. Here, we'll focus on **Creational** and **Structural** patterns in Java.

**Creational Design Patterns**

Creational patterns deal with object creation mechanisms, aiming to create objects in a manner suitable to the situation. These patterns increase flexibility and reuse of the code.

**1. Singleton Pattern**

Ensures that a class has only one instance and provides a global point of access to it.

**Example:**

public class Singleton { private static Singleton instance; private Singleton() { // Private constructor to prevent instantiation } public static Singleton getInstance() { if (instance == null) { instance = new Singleton(); } return instance; } }

**2. Factory Pattern**

Provides a way to create objects without specifying the exact class of the object being created.

**Example:**

interface Animal { void sound(); } class Dog implements Animal { public void sound() { System.out.println("Woof!"); } } class Cat implements Animal { public void sound() { System.out.println("Meow!"); } } class AnimalFactory { public static Animal createAnimal(String type) { if (type.equalsIgnoreCase("Dog")) { return new Dog(); } else if (type.equalsIgnoreCase("Cat")) { return new Cat(); } return null; } } public class FactoryPatternDemo { public static void main(String[] args) { Animal dog = AnimalFactory.createAnimal("Dog"); dog.sound(); Animal cat = AnimalFactory.createAnimal("Cat"); cat.sound(); } }

**3. Abstract Factory Pattern**

Works like the Factory Pattern but allows for the creation of families of related objects without specifying their concrete classes.

**4. Builder Pattern**

Constructs a complex object step by step. It separates the construction of a complex object from its representation.

**5. Prototype Pattern**

Creates new objects by cloning existing objects.

**6. Object Pool Pattern**

Manages a pool of reusable objects to reduce the overhead of object creation.

**Structural Design Patterns**

Structural patterns focus on the composition of classes or objects to form larger structures while keeping them flexible and efficient.

**1. Adapter Pattern**

Allows incompatible interfaces to work together by converting one interface into another.

**Example:**

java

interface MediaPlayer { void play(String audioType, String fileName); } class Mp3Player implements MediaPlayer { public void play(String audioType, String fileName) { System.out.println("Playing MP3 file. Name: " + fileName); } } class MediaAdapter implements MediaPlayer { AdvancedMediaPlayer advancedPlayer; public MediaAdapter(String audioType) { if (audioType.equalsIgnoreCase("vlc")) { advancedPlayer = new VlcPlayer(); } else if (audioType.equalsIgnoreCase("mp4")) { advancedPlayer = new Mp4Player(); } } public void play(String audioType, String fileName) { if (audioType.equalsIgnoreCase("vlc")) { advancedPlayer.playVlc(fileName); } else if (audioType.equalsIgnoreCase("mp4")) { advancedPlayer.playMp4(fileName); } } } public class AdapterPatternDemo { public static void main(String[] args) { MediaPlayer player = new MediaAdapter("vlc"); player.play("vlc", "example.vlc"); } }

**2. Bridge Pattern**

Separates an object’s abstraction from its implementation so that the two can vary independently.

**3. Composite Pattern**

Composes objects into tree structures to represent part-whole hierarchies.

**4. Decorator Pattern**

Adds new functionality to an object dynamically.

**5. Facade Pattern**

Provides a simplified interface to a larger body of code.

**6. Flyweight Pattern**

Reduces the cost of creating and managing a large number of similar objects.

**7. Proxy Pattern**

Provides a placeholder or surrogate to control access to another object.

**Summary Table**

| **Creational Patterns** | **Structural Patterns** |
| --- | --- |
| Singleton | Adapter |
| Factory | Bridge |
| Abstract Factory | Composite |
| Builder | Decorator |
| Prototype | Facade |
| Object Pool | Flyweight |
|  | Proxy |

**Creational Design Patterns**

1. **Singleton Pattern**
   * **Purpose**: Ensure a class has only one instance and provide a global point of access to it.
   * **Use Case**: Managing shared resources like configuration settings, database connections, etc.
   * **Example**: Logger service, where only one instance writes logs to a file.
   * **Real World**: A print spooler managing a single queue of print jobs.
2. **Factory Pattern**
   * **Purpose**: Create objects without specifying the exact class.
   * **Use Case**: When the exact class of the object to be created is determined dynamically.
   * **Example**: A ShapeFactory that creates circles, squares, or rectangles.
   * **Real World**: A car dealership creating different models of cars based on user requests.
3. **Abstract Factory Pattern**
   * **Purpose**: Provide an interface to create families of related or dependent objects without specifying their concrete classes.
   * **Use Case**: Systems that support multiple themes (e.g., dark and light modes).
   * **Example**: A GUI toolkit with factories for buttons, checkboxes, etc.
   * **Real World**: A furniture store where each furniture set includes related products (chair, table, sofa).
4. **Builder Pattern**
   * **Purpose**: Separate the construction of a complex object from its representation so the same construction process can create different representations.
   * **Use Case**: When building objects with many optional fields or configurations.
   * **Example**: Constructing a House object with optional features like a garden, pool, or garage.
   * **Real World**: Constructing a meal combo in a fast-food restaurant.
5. **Prototype Pattern**
   * **Purpose**: Create new objects by cloning existing ones.
   * **Use Case**: When object creation is expensive, and new objects share most of their attributes with existing ones.
   * **Example**: Cloning a Shape object with minor modifications.
   * **Real World**: Copying a document template to create new documents.
6. **Object Pool Pattern**
   * **Purpose**: Reuse objects from a pool rather than creating new ones to reduce overhead.
   * **Use Case**: Managing database connections or thread pools.
   * **Example**: A pool of reusable database connections.
   * **Real World**: A hotel reassigning cleaned rooms to new guests rather than building new rooms.

**Structural Design Patterns**

1. **Adapter Pattern**
   * **Purpose**: Allow incompatible interfaces to work together by wrapping one interface with another.
   * **Use Case**: Integrating new components into legacy systems.
   * **Example**: An AudioPlayer playing both .mp3 and .mp4 files by using an adapter.
   * **Real World**: A power adapter converting a 3-pin plug to fit a 2-pin socket.
2. **Bridge Pattern**
   * **Purpose**: Separate abstraction from implementation, allowing them to vary independently.
   * **Use Case**: Designing a system that should work with multiple implementations.
   * **Example**: A Shape abstraction implemented by Circle and Rectangle, with different rendering libraries.
   * **Real World**: TV remote control abstracted from the type of TV.
3. **Composite Pattern**
   * **Purpose**: Compose objects into tree-like structures to represent part-whole hierarchies.
   * **Use Case**: Managing hierarchies of objects, such as file systems or menus.
   * **Example**: A folder containing files and subfolders.
   * **Real World**: Organization charts showing employees and subordinates.
4. **Decorator Pattern**
   * **Purpose**: Dynamically add new responsibilities to an object without modifying its structure.
   * **Use Case**: Adding additional functionality to objects at runtime.
   * **Example**: A Coffee object that can be decorated with milk, sugar, etc.
   * **Real World**: Customizing a base pizza with extra toppings.
5. **Facade Pattern**
   * **Purpose**: Provide a simplified interface to a complex subsystem.
   * **Use Case**: Simplifying access to libraries or frameworks.
   * **Example**: A HomeTheater facade that controls multiple devices like TV, speakers, and DVD player.
   * **Real World**: A universal remote control.
6. **Flyweight Pattern**
   * **Purpose**: Minimize memory usage by sharing common data among objects.
   * **Use Case**: Managing large numbers of similar objects efficiently.
   * **Example**: Sharing graphical data for characters in a text editor.
   * **Real World**: Using shared sprites in a video game for efficiency.
7. **Proxy Pattern**
   * **Purpose**: Provide a placeholder for another object to control access or add functionality.
   * **Use Case**: Adding access control or lazy initialization.
   * **Example**: A virtual proxy loading large images on demand.
   * **Real World**: A credit card acting as a proxy for a bank account.

**Key Differences**

| **Creational Patterns** | **Structural Patterns** |
| --- | --- |
| Focus on object creation mechanisms. | Focus on object composition and relationships. |
| Examples: Singleton, Factory, Builder. | Examples: Adapter, Composite, Proxy. |