# Design Patterns in Java:

## **Core Java Design Patterns:**

## In core java, there are mainly three types of design patterns, which are further divided into their sub-parts:

## **1.Creational Design Pattern**

1. Factory Pattern
2. Abstract Factory Pattern
3. Singleton Pattern
4. Prototype Pattern
5. Builder Pattern

## **2. Structural Design Pattern**

1. Adapter Pattern
2. Bridge Pattern
3. Composite Pattern
4. Decorator Pattern
5. Facade Pattern
6. Flyweight Pattern
7. Proxy Pattern

## **3. Behavioural Design Pattern**

1. Chain Of Responsibility Pattern
2. Command Pattern
3. Interpreter Pattern
4. Iterator Pattern
5. Mediator Pattern
6. Memento Pattern
7. Observer Pattern
8. State Pattern
9. Strategy Pattern
10. Template Pattern
11. Visitor Pattern

# Creational design patterns:

Creational design patterns are concerned with**the way of creating objects.** These design patterns are used when a decision must be made at the time of instantiation of a class (i.e. creating an object of a class).

But everyone knows an object is created by using new keyword in java. For example:

StudentRecord s1=**new** StudentRecord();

Hard-Coded code is not the good programming approach. Here, we are creating the instance by using the new keyword. Sometimes, the nature of the object must be changed according to the nature of the program. In such cases, we must get the help of creational design patterns to provide more general and flexible approach.

# Factory Method Pattern:

A Factory Pattern or Factory Method Pattern says that just **define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate.** In other words, subclasses are responsible to create the instance of the class.

The Factory Method Pattern is also known as **Virtual Constructor.**

# Abstract Factory Pattern:

Abstract Factory Pattern says that just **define an interface or abstract class for creating families of related (or dependent) objects but without specifying their concrete sub-classes.**That means Abstract Factory lets a class returns a factory of classes. So, this is the reason that Abstract Factory Pattern is one level higher than the Factory Pattern.

An Abstract Factory Pattern is also known as **Kit.**

# Singleton design pattern in Java:

Singleton Pattern says that just **"define a class that has only one instance and provides a global point of access to it".**

In other words, a class must ensure that only single instance should be created and single object can be used by all other classes.

There are two forms of singleton design pattern

* **Early Instantiation:** creation of instance at load time.
* **Lazy Instantiation:** creation of instance when required.

Prototype Design Pattern:

Prototype Pattern says that **cloning of an existing object instead of creating new one and can also be customized as per the requirement**.

This pattern should be followed, if the cost of creating a new object is expensive and resource intensive.

Builder Design Pattern

Builder Pattern says that **"construct a complex object from simple objects using step-by-step approach"**

It is mostly used when object can't be created in single step like in the de-serialization of a complex object.

Object Pool Pattern

Mostly, performance is the key issue during the software development and the object creation, which may be a costly step.

Object Pool Pattern says that **" to reuse the object that are expensive to create".**

Basically, an Object pool is a container which contains a specified amount of objects. When an object is taken from the pool, it is not available in the pool until it is put back.**Objects in the pool have a lifecycle: creation, validation and destroy.**

A pool helps to manage available resources in a better way. There are many using examples: especially in application servers there are data source pools, thread pools etc.

import java.util.HashMap;

import java.util.Map;

// Singleton Pattern: DrawingManager ensures only one instance

class DrawingManager {

    private static DrawingManager instance;

    private DrawingManager() {}

    public static DrawingManager getInstance() {

        if (instance == null) {

            instance = new DrawingManager();

        }

        return instance;

    }

    public void drawShape(Shape shape) {

        shape.draw();

    }

}

// Abstract Factory Pattern: ShapeFactory interface and concrete factories

interface ShapeFactory {

    Shape createShape(String type);

}

class SimpleShapeFactory implements ShapeFactory {

    public Shape createShape(String type) {

        if (type.equalsIgnoreCase("circle")) {

            return new Circle();

        } else if (type.equalsIgnoreCase("rectangle")) {

            return new Rectangle();

        }

        return null;

    }

}

// Factory Method Pattern: Shape interface and concrete shapes

interface Shape extends Cloneable {

    void draw();

    Shape clone();

}

class Circle implements Shape {

    public void draw() {

        System.out.println("Drawing a Circle");

    }

    public Shape clone() {

        return new Circle();

    }

}

class Rectangle implements Shape {

    public void draw() {

        System.out.println("Drawing a Rectangle");

    }

    public Shape clone() {

        return new Rectangle();

    }

}

// Prototype Pattern: ShapeCache to clone shapes

class ShapeCache {

    private static Map<String, Shape> shapeMap = new HashMap<>();

    public static Shape getShape(String shapeId) {

        Shape cachedShape = shapeMap.get(shapeId);

        return (Shape) cachedShape.clone();

    }

    public static void loadCache() {

        Circle circle = new Circle();

        shapeMap.put("1", circle);

        Rectangle rectangle = new Rectangle();

        shapeMap.put("2", rectangle);

    }

}

// Builder Pattern: ComplexShape class with a builder

class ComplexShape implements Shape {

    private String partA;

    private String partB;

    private ComplexShape(Builder builder) {

        this.partA = builder.partA;

        this.partB = builder.partB;

    }

    public void draw() {

        System.out.println("Drawing a Complex Shape with " + partA + " and " + partB);

    }

    public Shape clone() {

        return new ComplexShape.Builder().partA(this.partA).partB(this.partB).build();

    }

    public static class Builder {

        private String partA;

        private String partB;

        public Builder partA(String partA) {

            this.partA = partA;

            return this;

        }

        public Builder partB(String partB) {

            this.partB = partB;

            return this;

        }

        public ComplexShape build() {

            return new ComplexShape(this);

        }

    }

}

// Object Pool Pattern: ShapePool to reuse shapes

class ShapePool {

    private Map<String, Shape> availableShapes = new HashMap<>();

    private Map<String, Shape> inUseShapes = new HashMap<>();

    public Shape acquireShape(String shapeId) {

        if (availableShapes.containsKey(shapeId)) {

            Shape shape = availableShapes.remove(shapeId);

            inUseShapes.put(shapeId, shape);

            return shape;

        }

        return null;

    }

    public void releaseShape(String shapeId) {

        if (inUseShapes.containsKey(shapeId)) {

            Shape shape = inUseShapes.remove(shapeId);

            availableShapes.put(shapeId, shape);

        }

    }

    public void loadShapes() {

        availableShapes.put("1", new Circle());

        availableShapes.put("2", new Rectangle());

    }

}

// Main class to demonstrate the patterns

public class ShapeDrawingApp {

    public static void main(String[] args) {

        // Singleton Pattern

        DrawingManager drawingManager = DrawingManager.getInstance();

        // Abstract Factory Pattern

        ShapeFactory shapeFactory = new SimpleShapeFactory();

        Shape circle = shapeFactory.createShape("circle");

        Shape rectangle = shapeFactory.createShape("rectangle");

        drawingManager.drawShape(circle);

        drawingManager.drawShape(rectangle);

        // Prototype Pattern

        ShapeCache.loadCache();

        Shape clonedCircle = ShapeCache.getShape("1");

        Shape clonedRectangle = ShapeCache.getShape("2");

        drawingManager.drawShape(clonedCircle);

        drawingManager.drawShape(clonedRectangle);

        // Builder Pattern

        ComplexShape complexShape = new ComplexShape.Builder()

                .partA("Part A")

                .partB("Part B")

                .build();

        drawingManager.drawShape(complexShape);

        // Object Pool Pattern

        ShapePool shapePool = new ShapePool();

        shapePool.loadShapes();

        Shape pooledCircle = shapePool.acquireShape("1");

        drawingManager.drawShape(pooledCircle);

        shapePool.releaseShape("1");

    }

}

# Structural design patterns

**Structural design patterns** are concerned with how classes and objects can be composed, to form larger structures.

The structural design patterns **simplifies the structure by identifying the relationships**.

These patterns focus on, how the classes inherit from each other and how they are composed from other classes.

## **Types of structural design patterns**

Adapter Pattern

An Adapter Pattern says that just **"converts the interface of a class into another interface that a client wants".**

In other words, to provide the interface according to client requirement while using the services of a class with a different interface.

The Adapter Pattern is also known as **Wrapper.**

Bridge Pattern

A Bridge Pattern says that just **"decouple the functional abstraction from the implementation so that the two can vary independently".**

The Bridge Pattern is also known as **Handle or Body.**