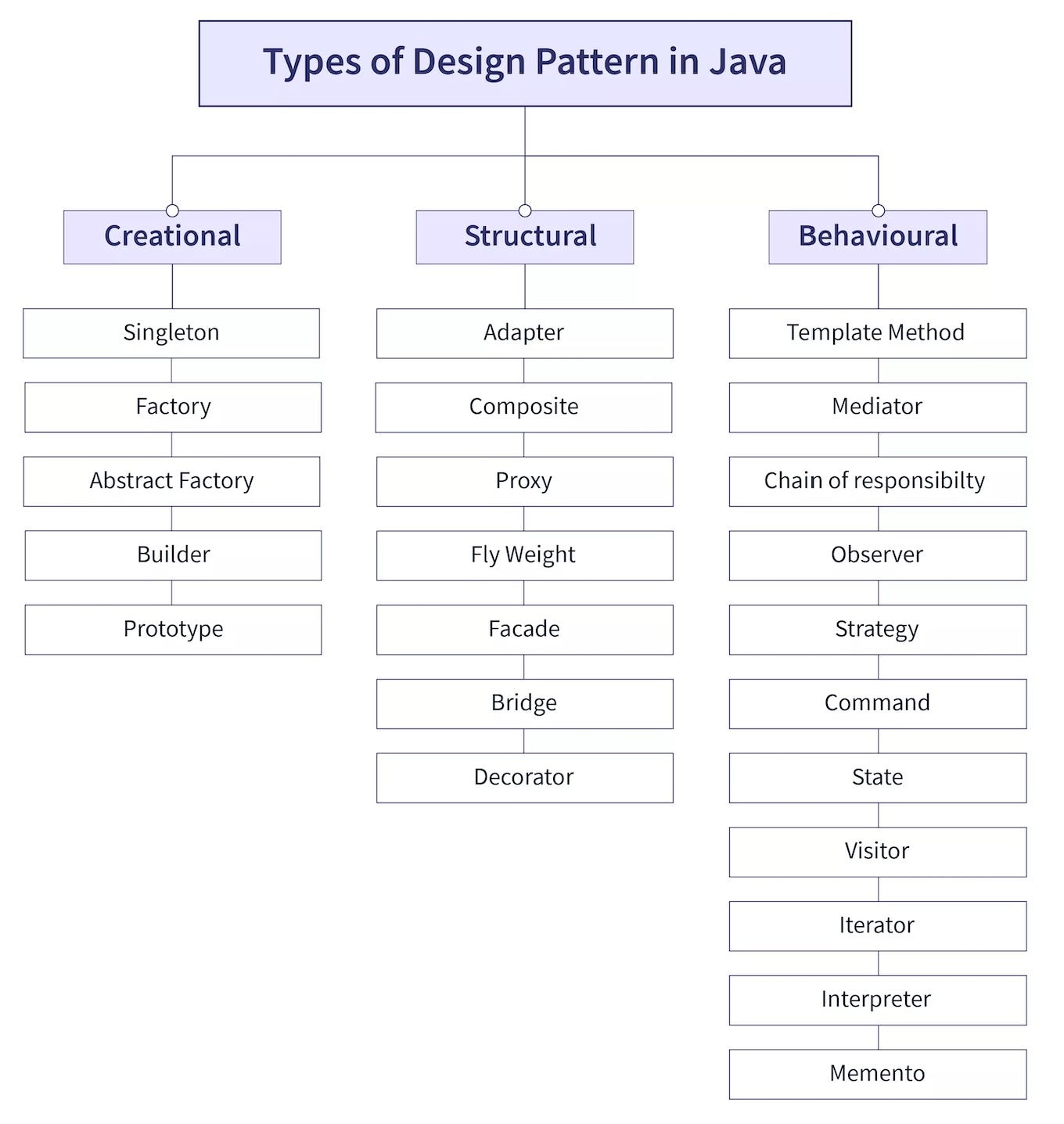
**Design Patterns**



**Creational**

**Singleton Design Pattern**

Singleton says that a class that has only one instance provide for global point of access.

Advantage is save the memory because object is not created only single instance will reuse at each request.

There are two form for Creation of Singleton Pattern.

Early (Create at load time) and Lazy (Create when required)

Rule for Creation Singleton Design Pattern

1. Create Private Static instance.
2. Create Private Constructor.
3. Create Static method for access.

Early or Eager Instantiation

In simple word instance will create at the time of loading the class by JVM.

public class EagerSingleton {  
 private static final EagerSingleton *instance* = new EagerSingleton();  
  
 private EagerSingleton()  
 {  
 System.*out*.println("EagerSingleton Constructor Call");  
 }  
  
 public static EagerSingleton getInstance()  
 {  
 return *instance*;  
 }  
}

Lazy Instantiation

It means the initialization will until the first time it call then it will create object.

public class LazySingleton {  
 private static LazySingleton *instance*;  
  
 private LazySingleton()  
 {  
  
 }  
  
 public static LazySingleton getInstance()  
 {  
 if (*instance* == null)  
 {  
 return *instance* = new LazySingleton();  
 }else {  
 return *instance*;  
 }  
 }  
  
}

Now in in this approach while the multithread environment works then we need to add Synchronized keyword.

To make singleton class thread safe that’s why **we make getInstance() method is made synchronized so that multiple thread can’t access it simultaneously.**

public class LazySingleton {  
 private static LazySingleton *instance*;  
  
 private LazySingleton()  
 {  
  
 }  
  
 public static synchronized LazySingleton getInstance()  
 {  
 if (*instance* == null)  
 {  
 return *instance* = new LazySingleton();  
 }else {  
 return *instance*;  
 }  
 }  
  
}

Now we also create a double checking lazy singleton class for thread safe.

public class LazyDoubleCheckSingleton {  
 private static LazyDoubleCheckSingleton *instance*;  
  
 private LazyDoubleCheckSingleton()  
 {  
  
 }  
  
 public static LazyDoubleCheckSingleton getInstance() {  
 if (*instance* == null) {  
 synchronized (LazyDoubleCheckSingleton.class) {  
 if (*instance* == null) {  
 return *instance* = new LazyDoubleCheckSingleton();  
 }  
 }  
 }  
 return *instance*;  
 }  
  
}

Now Lazy Inner Class Singleton

In this approach the inner class cannot get loaded until the getInstance() method call for the first time so that this solution is thread safe we don’t need to add manually the Synchronized.

So this is the most efficient approach for singleton design pattern with Thread safe.

public class LazyInnerSingleton {  
 private LazyInnerSingleton(){}  
 private static class SingletonHelper  
 {  
 private static final LazyInnerSingleton *instance* = new LazyInnerSingleton();  
 }  
  
 public static LazyInnerSingleton getInstance()  
 {  
 return SingletonHelper.*instance*;  
 }  
}

How to Prevent from Clone method of singleton class

**Clone**

**Reflection**

**Synchronized**

If we extend the clone class in singleton class then we should override clone method and throw CloneNotSupportException.

public class LazySingleton extends MyClone{  
 private static LazySingleton *instance*;  
  
 @Override  
 protected Object clone() throws CloneNotSupportedException {  
 throw new CloneNotSupportedException("Do not Clone this singleton class");  
 }  
  
 private LazySingleton()  
 {  
  
 }  
  
 public static synchronized LazySingleton getInstance()  
 {  
 if (*instance* == null)  
 {  
 return *instance* = new LazySingleton();  
 }else {  
 return *instance*;  
 }  
 }  
  
}

**Factory Design Pattern – Factory Method Design Pattern**

When there is super class and multiple subclasses and we want to get the object of the child classes based on input and requirement then there is use of factory pattern.

**In simple word we would say when there is need of any object of subclass on while on time requirement factory pattern class will create of that class object for us is known as Factory Design Pattern.**

Advantage

Focus on creating objects for Interface rather than implementation.

Loose coupling more robust code

**For Example**

If we have Transport company class which will transport the product by its sub class of Truck class.

Now in future we have other requirement for transportation by sea so we need to create call Ship.

But the problem is our App class have to create object by class name which is tight coupling.

Now this time we need Factory class for the crating object on the basis of requirements.

public interface Transport {  
 public void deliver();  
}

public class Truck implements Transport{  
 @Override  
 public void deliver() {  
 System.*out*.println("Deliver by Land in Box");  
 }  
}

public class Ship implements Transport{  
 @Override  
 public void deliver() {  
 System.*out*.println("Deliver by sea in a container");  
 }  
}

public class Factory {  
 public static Transport getTransport(String transportType)  
 {  
 if (transportType.trim().equalsIgnoreCase("Truck"))  
 {  
 return new Truck();  
 }  
 else if (transportType.trim().equalsIgnoreCase("Ship"))  
 {  
 return new Ship();  
 }  
 else  
 {  
 return null;  
 }  
 }  
}

public class MyApp {  
 public static void main(String[] args) {  
 Transport truck = Factory.*getTransport*("Truck");  
 truck.deliver();  
 Transport ship = Factory.*getTransport*("Ship");  
 ship.deliver();  
 }  
}

**Abstract Factory Design 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.

public interface Furniture {  
 public void getChair();  
 public void getSofa();  
}

public class ModernFurniture implements Furniture {  
 @Override  
 public void getChair() {  
 System.*out*.println("Modern Chair is Build");  
 }  
  
 @Override  
 public void getSofa() {  
 System.*out*.println("Modern Sofa is Build");  
 }  
}

public class VictorianFurniture implements Furniture {  
 @Override  
 public void getChair() {  
 System.*out*.println("Victorian Chair is Build");  
 }  
  
 @Override  
 public void getSofa() {  
 System.*out*.println("Victorian Sofa is Build");  
 }  
}

abstract public class FurnitureAbstractFactory {  
 public abstract Furniture createFurniture();  
}

public class ModernFurnitureAbstarctFactory extends FurnitureAbstractFactory{  
 @Override  
 public Furniture createFurniture() {  
 return new ModernFurniture();  
 }  
}

public class VictorianFurnitureAbstractFactory extends FurnitureAbstractFactory{  
 @Override  
 public Furniture createFurniture() {  
 return new VictorianFurniture();  
 }  
}

public class FurnitureFactory {  
 public static Furniture getFurniture(FurnitureAbstractFactory abstractFactory)  
 {  
 return abstractFactory.createFurniture();  
 }  
}

public class FurnitureTest {  
 public static void main(String[] args) {  
 Scanner sc = new Scanner(System.*in*);  
 String furnitureType;  
 Furniture furniture = null;  
 while (true)  
 {  
 System.*out*.println("Select Furniture [Modern,Victorian]");  
 furnitureType = sc.nextLine();  
 if(furnitureType.trim().equalsIgnoreCase("Modern"))  
 {  
 furniture = FurnitureFactory.*getFurniture*(new ModernFurnitureAbstarctFactory());  
 furniture.getChair();  
 furniture.getSofa();  
 } else if (furnitureType.trim().equalsIgnoreCase("Victorian")) {  
 furniture = FurnitureFactory.*getFurniture*(new VictorianFurnitureAbstractFactory());  
 furniture.getChair();  
 furniture.getSofa();  
 }  
 else {  
 System.*out*.println("Thank You");  
 break;  
 }  
 }  
 }  
}

**Builder Design Pattern**

It provide us to create complex objects on step-by-step basis, it allows us to produce different types and representation of an object using same construction process.

In simple word Builder design pattern provide us to build the complex object on step-by-step process.

Creation of Builder class

Create Class with fields and only getter method and tostring.

public class User {  
 private final String userId;  
 private final String userName;  
 private final String userE mail;

private User(UserBuilder builder) {  
 this.userId = builder.userId;  
 this.userName = builder.userName;  
 this.userEmail = builder.userEmail;  
}

public String getUserId() {  
 return userId;  
 }  
  
 public String getUserName() {  
 return userName;  
 }  
  
 public String getUserEmail() {  
 return userEmail;  
 }  
  
 @Override  
 public String toString() {  
 return "User{" +  
 "userId='" + userId + '\'' +  
 ", userName='" + userName + '\'' +  
 ", userEmail='" + userEmail + '\'' +  
 '}';  
 }  
  
}

Create Static inner class with same fields and only setter methods and constructor.

Now make those methods into method chaining and return this.

static class UserBuilder  
{  
 private String userId;  
 private String userName;  
 private String userEmail;  
  
 public UserBuilder()  
 {  
  
 }  
  
 //Method Chaining  
  
 public UserBuilder setUserId(String userId) {  
 this.userId = userId;  
 return this;  
 }  
  
 public UserBuilder setUserName(String userName) {  
 this.userName = userName;  
 return this;  
 }  
  
 public UserBuilder setUserEmail(String userEmail) {  
 this.userEmail = userEmail;  
 return this;  
 }  
  
 public User build()  
 {  
 User user = new User(this);  
 return user;  
 }  
}

Now create private constructor in outer class with static inner class as parameter.

private User(UserBuilder builder) {  
 this.userId = builder.userId;  
 this.userName = builder.userName;  
 this.userEmail = builder.userEmail;  
}

Lastly test those Builder class in main method.

public class Test {  
 public static void main(String[] args) {  
 User abc = new User

.UserBuilder()

.setUserId("1")

.setUserName("ABC")

.setUserEmail("abc@gmail.com")

.build();  
 System.*out*.println(abc);  
 }  
}

**Prototype Design Pattern**

It state that we will create copy of existing object.

Suppose we have object with complex logic and now we need new object but same as this old object then we will use cloning of that object.

In simple word prototype design pattern says we will cloning of that existing object rather than creating new one and we can also customize this clone object as per requirement.

For example

Consider one bike has speed, power, mileage, A+tires.

Now this bike can run only on Highways now there is new requirement for user to ride this bike on mountains or off-road. Now we need same bike but with other features so then we can’t create new bike from scratch now we can use this existing bike object and clone it as per our requirement.

Now here is condition for cloning existing object.

Existing object has so many fields with very complex logic, time consuming and very costly.

So that time if we need other object as same as existing object then we have to use cleanable interface.

Cleanable interface is functional interface which will says to JVM that this class will be cloned.

And we have to override clone method in this class.

public class Bike implements Cloneable{

@Override  
protected Object clone() throws CloneNotSupportedException {  
 return super.clone();  
}

}

public class User {  
 public static void main(String[] args) throws CloneNotSupportedException, InterruptedException {  
 Bike b1 = new Bike();  
 b1.setSpeed(350);  
 b1.setPowerComplexAndTimeTakingProcess("1200Hp");  
 b1.setMileage(25);  
 b1.setTierType("Racing");  
 System.*out*.println(b1);  
  
 try {  
 Bike b2 = (Bike)b1.clone();  
 b2.setTierType("Off-Road");  
 System.*out*.println(b2);  
 } catch (CloneNotSupportedException e) {  
 throw new RuntimeException(e);  
 }  
 }  
}

There is two types of Copy **Shallow Copy and Deep Copy**

**Shallow Copy**

Shallow copy mean if object having anther object reference so it will also copy that reference and whenever we will make any changes in this inner object field value the it will reflect in other copy or cloned object.

NetworkConnection1 connection1 = new NetworkConnection1();  
  
  
connection1.setIp("196.126.4.5");  
connection1.setImportantDataWithTimeTaking("Network Credentials");  
connection1.setDomains();  
  
NetworkConnection1 connection2 = (NetworkConnection1) connection1.clone();  
  
connection1.getDomains().remove(0);  
System.*out*.println("Here We remove domain in connection 1 but it will reflect on connection 2");  
System.*out*.println(connection1);  
System.*out*.println(connection2);

**Deep Copy**

Deep copy means we can clone Object who’s having another object reference and our clone method will also clone this inner object then this is known as Deep Copy.

In this we need to write a logic for cloning in clone method.

@Override  
protected Object clone() throws CloneNotSupportedException {  
 // logic for cloning  
 NetworkConnection2 networkConnection1 = new NetworkConnection2();  
 networkConnection1.setIp(this.getIp());  
 networkConnection1.setImportantData(this.getImportantData());  
  
 for (String d : this.getDomains())  
 {  
 networkConnection1.getDomains().add(d);  
 }  
  
 return networkConnection1;  
}

NetworkConnection2 connection3 = new NetworkConnection2();  
  
  
connection3.setIp("196.126.4.5");  
connection3.setImportantDataWithTimeTaking("Network Credentials");  
connection3.setDomains();  
  
NetworkConnection2 connection4 = (NetworkConnection2) connection3.clone();  
  
connection3.getDomains().remove(0);  
System.*out*.println("Here We remove domain in connection 3 but No changes on connection 4");  
System.*out*.println(connection3);  
System.*out*.println(connection4);