**Design Pattern**

**What is Design pattern:**

* **Design patterns are reusable solution to the problems that we encounter in day-to-day programming**
* **They are generally targeted at solving the problems of object generation and integration.**
* **Design patterns represent the best practices used by experienced object-oriented software developers.**
* **Design patterns are solutions to general problems that software developers faced during software development.**
* **During our software development we will face some kind of general problem, we have solution for the same which is called design pattern.**
* **We can just reuse the same in our development**
* **Simply says it is well-proved solution for solving the specific problem/task.**
* **By using the design patterns you can make your code more flexible, reusable and maintainable.**

### When should we use the design patterns?

* **We must use the design patterns during the analysis and requirement phase of SDLC(Software Development Life Cycle).**
* **Design patterns ease the analysis and requirement phase of SDLC by providing information based on prior hands-on experiences.**

### Categorization of design patterns:

**Basically, design patterns are categorized into two parts:**

1. **Core Java (or JSE) Design Patterns.**
2. **J2EE Design Patterns.**

## 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. **Singleton Pattern**
2. **Factory Pattern**
3. **Abstract Factory Pattern**
4. **Builder Pattern**
5. **Prototype Pattern**
6. **Object pool 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. Behavioral 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**
12. **Creational Design Pattern:**

* **This type of DP is deals with object creation and initialization.**
* **This pattern gives the more flexibility in deciding which objects needs to be created for the given case.**
* **Eg: Singleton, Factory, Abstract Factory, etc**

1. **Structural Design Pattern:**

* **This type of DP deals with class & object composition**
* **This pattern focusing on decoupling interface and implementation of classes and its objects**
* **Eg: Adapter, Bridge, Etc.**

1. **Behavioural Design Pattern:**

* **This DP deals with the communication between the classes and objects in loose couple manner**
* **Eg: chain of Responsibility, Command, Interpreter etc**

1. **Singleton Design Pattern:**

* **Singleton belongs to Creational Design Pattern**
* **This pattern is used when we need to create that only one instance of a particular class need to be created in JVM.**
* **All further References to the objects are referred to the same underlying instance**
* **Make sure the singleton class provide global access point to get the instance**

**Advantage of Singleton:**

* **It ensures there is only one object available across the application in a controlled state**
* **Used to create logging, caching, data base connection creation**

**Implementation:**

* **Ensure that only one instance of the class exists**
* **Provide global access to that instance by**

1. **Declaring all constructors of the class to be private**
2. **Providing static method that returns a reference to the instance**
3. **The instance is stored as a private static variable**

**1. Early Instantiation**

* The object creation takes place at the load time

**Class Singleton{**

**Private static Singleton inst= new Singleton();**

**Private Singleton{}**

**Public static Singleton getInstance{**

**return inst;**

**}**

**2.Lazy Instantiation**

* creation of instance when required.

**Eg : Class Singleton{**

**Private static Singleton inst=null;**

**Private Singleton{};**

**Public static Singleton getInstance{**

**If(inst==null)**

**inst= new Singleton();**

**return inst;**

**}**

### **3. Thread Safe Singleton Method**

* But the above approach can raise some concerns in the concurrent scenarios. Since singleton pattern is mainly used with multi-threads and if multiple threads enter the if condition at the same time it can raise issues.
* To avoid this we try to create a thread-safe singleton class by making the global access method synchronized. This ensures that only one thread is executing this method at any point in time. Refer to the below code to see the implementation:

**How to break Singleton Design pattern:**

**1.** **Serialization:-**

* [Serialization](https://www.geeksforgeeks.org/serialization-in-java/) can also cause breakage of singleton property of singleton classes.
* Serialization is used to convert an object of byte stream and save in a file or send over a network.
* Suppose you serialize an object of a singleton class. Then if you de-serialize that object it will create a new instance and hence break the singleton pattern.

class Singleton implements Serializable

{

    // public instance initialized when loading the class

    public static Singleton instance = new Singleton();

    private Singleton()

    {

        // private constructor

    }

}

public class GFG

{

    public static void main(String[] args)

    {

        try

        {

            Singleton instance1 = Singleton.instance;

            ObjectOutput out

                = new ObjectOutputStream(new FileOutputStream("file.text"));

            out.writeObject(instance1);

            out.close();

            // deserialize from file to object

            ObjectInput in

                = new ObjectInputStream(new FileInputStream("file.text"));

            Singleton instance2 = (Singleton) in.readObject();

            in.close();

            System.out.println("instance1 hashCode:- "

                                                 + instance1.hashCode());

            System.out.println("instance2 hashCode:- "

                                                 + instance2.hashCode());

        }

        catch (Exception e)

        {

            e.printStackTrace();

        }

    }

**Overcome serialization issue:-**

* To overcome this issue, we have to implement method readResolve() method.

|  |
| --- |
| class Singleton implements Serializable  {    // public instance initialized when loading the class      public static Singleton instance = new Singleton();      private Singleton()      {        // private constructor      }      // implement readResolve method      protected Object readResolve()      {        return instance;      }  }  public class GFG  {      public static void main(String[] args)      {          try          {              Singleton instance1 = Singleton.instance;              ObjectOutput out                  = new ObjectOutputStream(new FileOutputStream("file.text"));              out.writeObject(instance1);              out.close();              // deserialize from file to object              ObjectInput in                  = new ObjectInputStream(new FileInputStream("file.text"));              Singleton instance2 = (Singleton) in.readObject();           in.close();              System.out.println("instance1 hashCode:- "  + instance1.hashCode());              System.out.println("instance2 hashCode:- " instance2.hashCode());          }          catch (Exception e)          {    e.printStackTrace();        } }} |

1. **Cloning:**

* [Cloning](https://www.geeksforgeeks.org/clone-method-in-java-2/) is a concept to create duplicate objects.
* Using clone we can create copy of object.
* Suppose, we create clone of a singleton object, then it will create a copy that is there are two instances of a singleton class, hence the class is no more singleton.

**class SuperClass implements Cloneable**

**{**

**int i = 10;**

**@Override**

**protected Object clone() throws CloneNotSupportedException**

**{ return super.clone();**

**}**

**}**

**// Singleton class**

**class Singleton extends SuperClass**

**{**

**// public instance initialized when loading the class**

**public static Singleton instance = new Singleton();**

**private Singleton()**

**{ // private constructor**

**}**

**}**

**public class GFG**

**{**

**public static void main(String[] args) throws CloneNotSupportedException**

**{**

**Singleton instance1 = Singleton.instance;**

**Singleton instance2 = (Singleton) instance1.clone();**

**System.out.println("instance1 hashCode:- "**

**+ instance1.hashCode());**

**System.out.println("instance2 hashCode:- "**

**+ instance2.hashCode());**

**}**

**}**

**Overcome Cloning issue:-**

* To overcome this issue, override clone() method and throw an exception or return the same instance from clone method that is CloneNotSupportedException or return the same instance.
* Now whenever user will try to create clone of singleton object, it will throw exception and hence our class remains singleton.

**class SuperClass implements Cloneable**

**{**

**int i = 10;**

**@Override**

**protected Object clone() throws CloneNotSupportedException**

**{ return super.clone();**

**}**

**}**

**// Singleton class**

**class Singleton extends SuperClass**

**{// public instance initialized when loading the class**

**public static Singleton instance = new Singleton();**

**private Singleton()**

**{ // private constructor**

**}**

**@Override**

**protected Object clone() throws CloneNotSupportedException**

**{**

**throw new CloneNotSupportedException();**

**or**   **return instance;**

**}**

**}**

**public class GFG**

**{**

**public static void main(String[] args) throws CloneNotSupportedException**

**{**

**Singleton instance1 = Singleton.instance;**

**Singleton instance2 = (Singleton) instance1.clone();**

**System.out.println("instance1 hashCode:- "**

**+ instance1.hashCode());**

**System.out.println("instance2 hashCode:- "**

**+ instance2.hashCode());**

**}**

**}**

**2.Factory Design Pattern:**

* **IT is creational design pattern**
* **Factory Pattern is one of the most used design pattern in real world application, It is creational design pattern**
* **Factory pattern creates object without exposing the creation logic to the client and refer newly created object using a common interface.**
* **In Factory DP instead of we creating object we are allowing someone to create the object.**
* **Mainly it is used to when we have multiple sub classes of a Super class & based on input we want to return one class instance**
* **It provides the abstraction between implementation & client classes**
* **Remove the instantiation of client classes from client.**
* **Here Superclass can be interface, abstract class or basic class.**
* **Factory class has a static method which returns the instance of sub class based on input.**
* **Eg: let us assume we have a main method, in that main method we need a object of OS (interface)**

**Interface OS{**

**Void show(); by default all the methods in i/f are public abstract**

**}**

**Public class IOS implements OS{**

**Void show(){}**

**}**

**Public class Windows implements OS{**

**Void show(){}**

**}**

**Public class Android implements OS{**

**Void show(){}**

**}**

**FactoryMain{**

**p.s.v.m(){**

**OS o = new Windows();**

**o.show();**

**}**

**}**

* **We can achieve the low coupling by the factory design pattern**
* **Instead of we creating object by Os o = new Windows (),**
* **Here we are exposing the implementation class (Windows). In future we change the OS we have to change the code and recompile it**
* **Create below factory Class**

**Class OperatingSystemFactory{**

**Public OS getInstance(String str){**

**If(str.equals(“Open”)**

**return new Android();**

**else If(str.equals(“Close”)**

**return new IOS();**

**else**

**return new Windows();**

**}**

**}**

**Change main method like below**

**p.s.v.m(){**

**OperatingSystemFactory osf = new OperatingSystemFactory();**

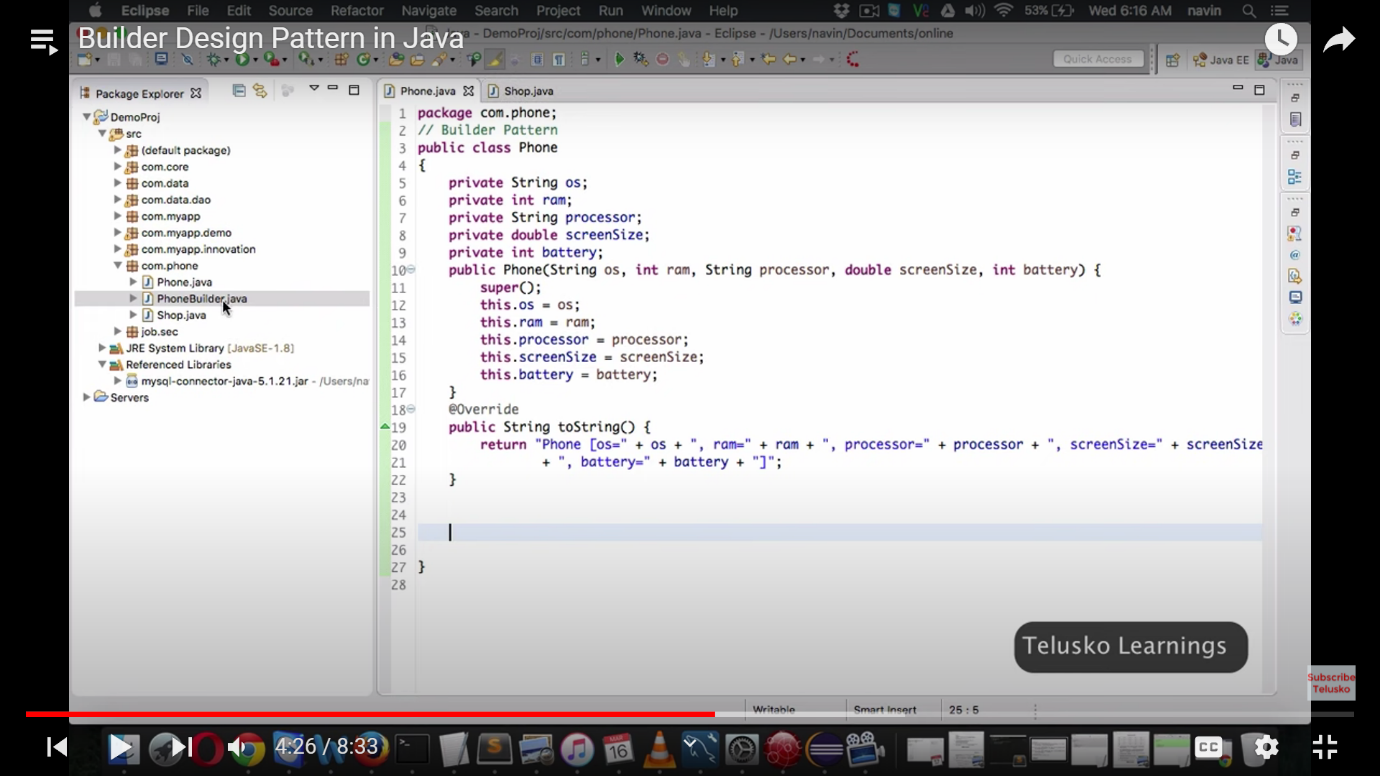
**OS o = osf.getInstance(“Open”); // to get android obj**

**o.show();**

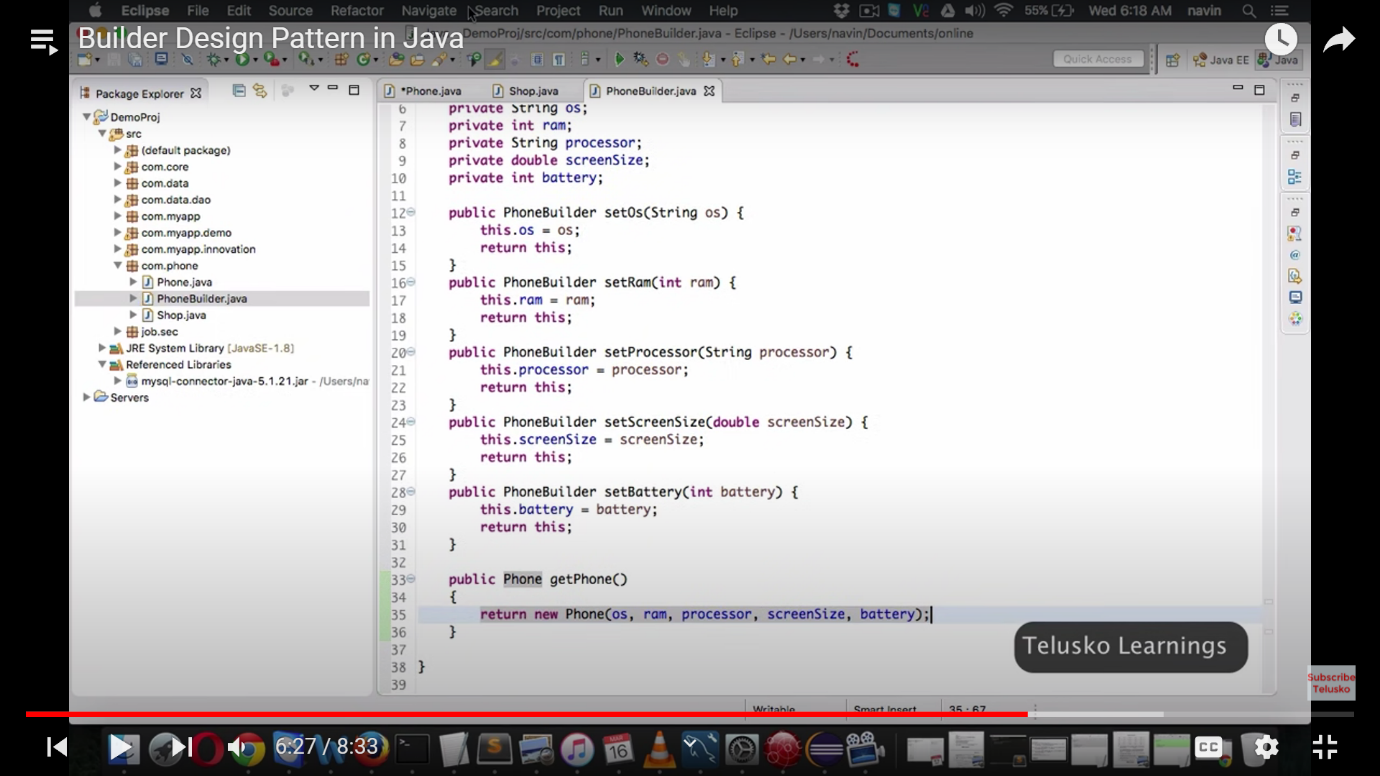
**}**

**4.Builder Design pattern**

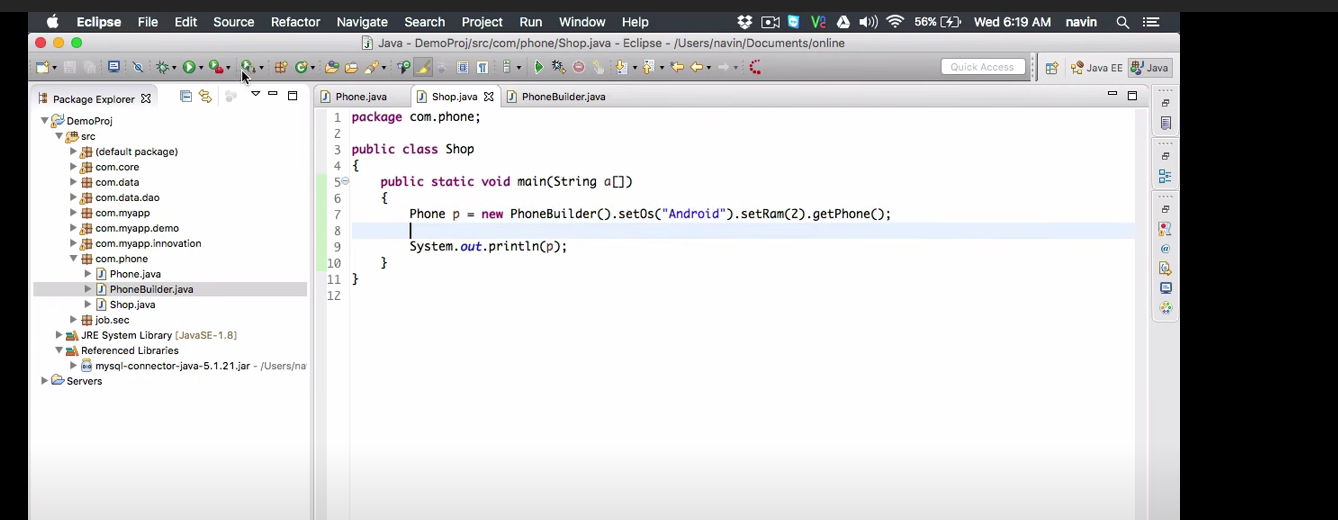
* **Builder design pattern is part of Creational design pattern**
* The builder pattern is a design pattern that allows for the step-by-step creation of complex objects using the correct sequence of actions.
* It provides clear separation between the construction and object representation.
* If there is too many attributes and we want to create object by passing only few attribute we can use builder design pattern.also here param sequence not matter.
* Eg: creating phone object,
* In below phone class we are having 5 attributes , and parameter constructor. So to create object for phone class I need to pass 5 param .
* It will be more complex if we have more attributes. We cant pass all the arguments at every time , it may produce errors, and we may want to pass only few arguments to create objects



* **To overcome this issue we need to create builder class.**
* **Here it is phoneBuilder class which has all the attributes, setter method to set the values and method to create and return phone object**



* **Now we can create phone object like below by passing only few params**

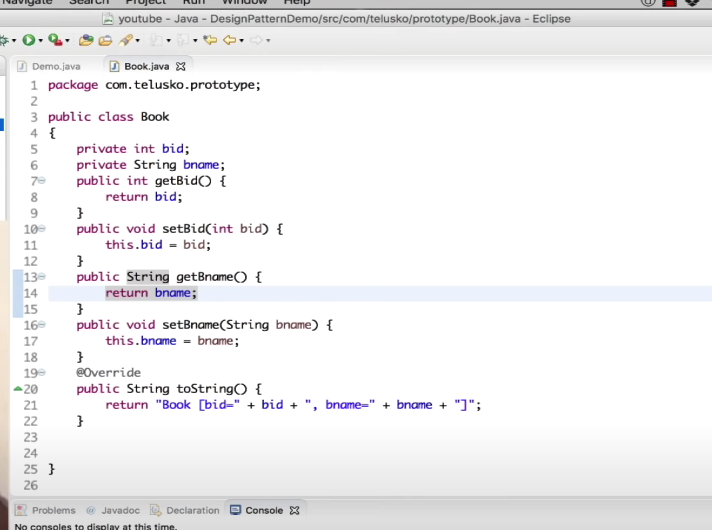


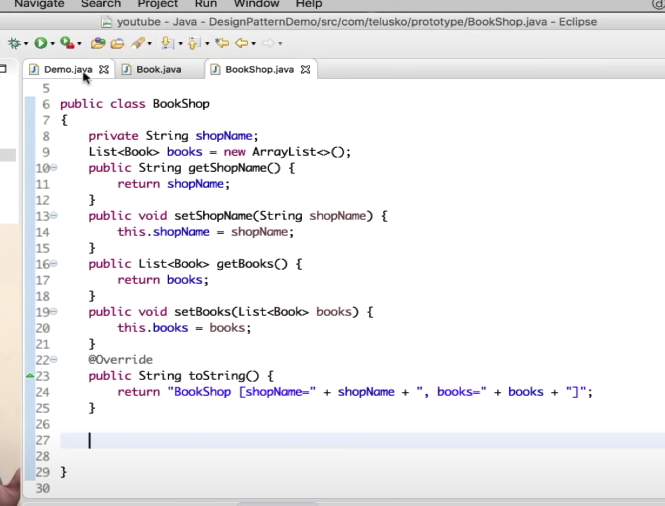
**5. Prototype Design pattern**

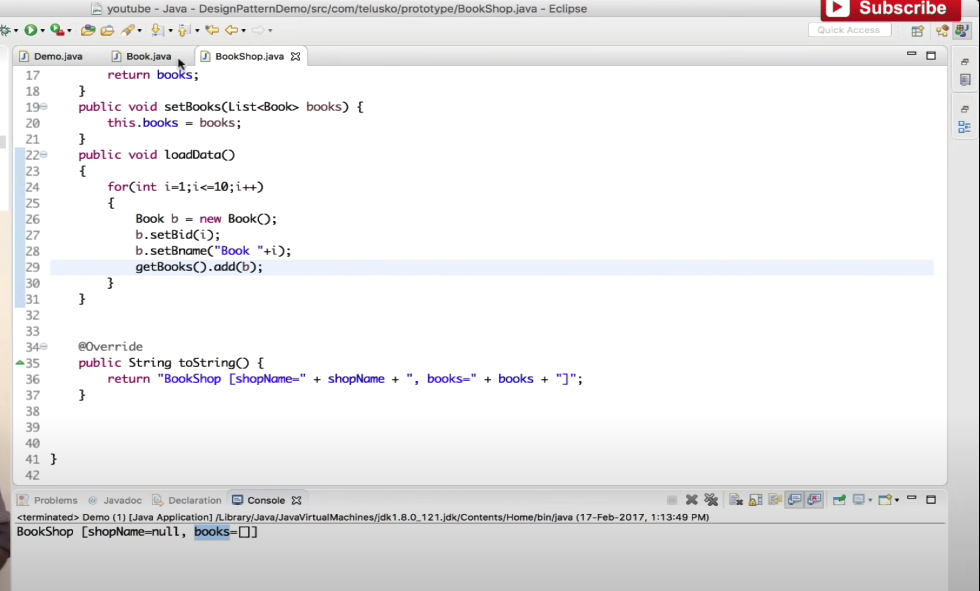
* This pattern comes under a **creational**pattern.
* prototypes design pattern allows you to create objects by **cloning** an existing object instead of creating a new object from scratch.
* This pattern is used when the process of **object creation is costly**. Like fetching data from data base
* when cloning, the newly copied object contains the same characteristics as its source object.
* After cloning, we can change the values of the new object’s properties as required

**Ref side :** [**https://medium.com/geekculture/overview-of-prototype-desing-pattern-3eafaf006fde#:~:text=prototypes%20design%20pattern%20allows%20you,characteristics%20as%20its%20source%20object**](https://medium.com/geekculture/overview-of-prototype-desing-pattern-3eafaf006fde#:~:text=prototypes%20design%20pattern%20allows%20you,characteristics%20as%20its%20source%20object)**.**

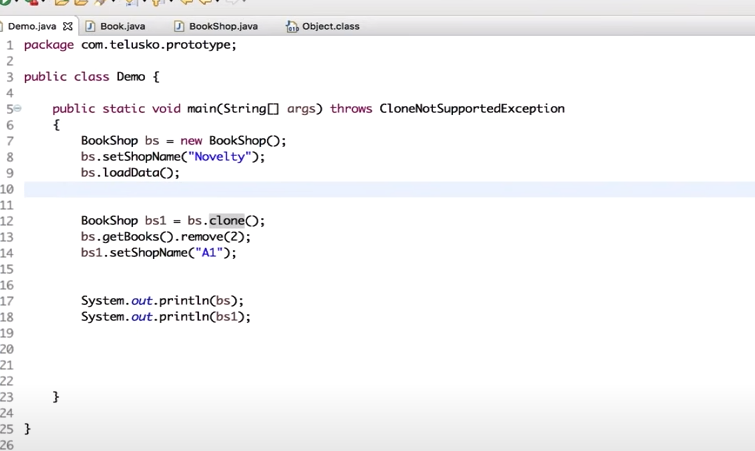
* **Create Book class**



* **Create Book shop class which will have book list**

,

* **Now start create the book shop object. Let us assume if we need to fetch book data from data base it will take time to load books object in book shop , it will leads to performance issue.**
* **So we can clone the existing bookshop object and add modification in new object like below**



* **Here bookshop need to implement clonable interface and override clone method.**

**Video ref:** [**https://www.youtube.com/watch?v=nZ76x13Nm8Q&list=PLsyeobzWxl7r2ZX1fl-7CKnayxHJA\_1ol&index=10**](https://www.youtube.com/watch?v=nZ76x13Nm8Q&list=PLsyeobzWxl7r2ZX1fl-7CKnayxHJA_1ol&index=10)

**Structural Design Pattern**

**1.Bridge Design Pattern**

**1. Bridge** DP is a structural design pattern

**2.**  that decouple the abstraction from its implementation

**3.** it used if we want runtime binding of the implementation and want to share the implementation among multiple objects.

**4.** we can hide the implementation from the client

**Video ref:**

[**https://www.youtube.com/watch?v=AvszFRYvvt0&t=196s**](https://www.youtube.com/watch?v=AvszFRYvvt0&t=196s)

**2.proxy design pattern**

**3.Facade design pattern**

* **It is structural DP**
* **Structural design pattern is improves the structure of java code**
* **Façade DP is used when there are multiple interfaces of similar kind of jobs, then we add a Façade Interface, it basically helps in routing to related interface**
* **Eg : google, chrome etc, Databases**
* **We will implement Façade helper class, which will route to method related to specific class based on input.**

**4.composite design patters**

* **It is structural Design pattern**
* **This used to composite individual objects and composition of objects**

**5.Decorator Design pattern**

* **Decorator design pattern is Structural Design pattern.**
* **Decorator design pattern is used when we want to modify functionality of an Object at runtime & it should not change individual object functionality.**
* **i.e, Adding different functionality is Dress**

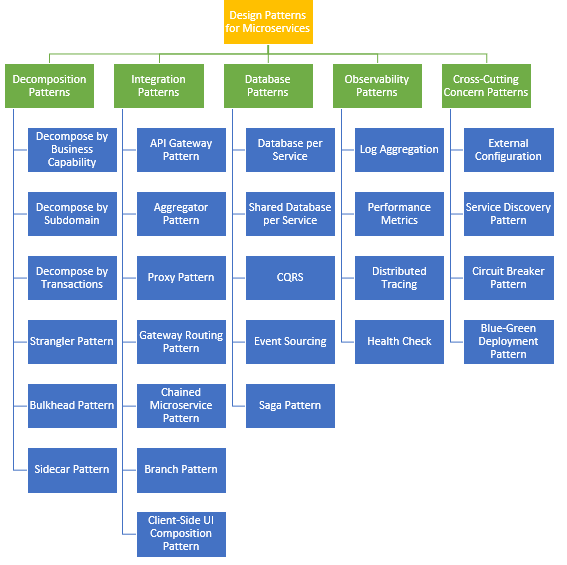
**6.FlyWeight Design Pattern**

**7. Adapter Design Pattern**

* **Adapter design pattern is Structural Design pattern**
* **Adapter design pattern used when objects offering same features, but has different interface.**
* **It allows existing class to be used with others without modifying their source code.**
* **Eg: webdriver adapter**

**MicroService Design Pattern**

**What are micro service design pattern**



**1.Saga**

* **The saga design pattern comes under data base patterns**
* **It is a way to manage data consistency across microservices in distributed transaction scenario. we have transaction across multiple micro service**
* **Eg: take uber eats application, if we try to order something payment failed in payment service that time we should not save any data for that order and payment, we can use saga pattern to handle this.**
* **2 ways of implementing SAGA design pattern**

**1. choreography**

**2. Orchestration pattern**