Design Patterns

Who needs this design pattern course?
ans) Every one

Where can we implement Design Pattern?
-->In languages ,technologies and frameworks which are dealing with Object oriented Programming , we can implement design patterns.. like java ,c# ,c++,phyton and etc...

What is design pattern?

Date pattern :: dd-MM-yyyy
Any thing that reoccuring is called pattern...

(a) Design Patterns are best solutions that come as set of rules for reocurring problems of Application development..
(b) Desging Patterns are best pratices to use software languages,technologies ,framworks and etc.. effectively to devlope s/w Applications or Projects..
(c) A Design pattern is a 3 part rule
  (a) Context
  (b) Problem
  (c) Solution

DEsign pattern is a best solution for the Problem that is raised in cerntain Context (Condition)

How to work with Design patterns

We build projects using apis given by Languages,technologies and frameworks

APIs in java are packages having classes,interfaces,enums and annotations...

note: APIs do not tell what is good ,what is bad.. So programmer should manually analyze
  (a) What is bad pratice or bad code
  (b) what is best pratice or best code
  (c) Wheather these pratices can solve all the problems in all contexts or not?
This kind fo analuzation must be documented for future reference...

History

Cristopher Alexendar ---> Civil Engineer --> 1970

1990s --> Four people --> GOF --> GO4
Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides
release book havingg 23 Patterns that is
Elements of reusable object oriented software...
(All these patterns are addersing only the problems related to standalone Apps development)
=> Any OOP language can be used to implment GOF patterns..
eg: Signleton , Factory Pattern, AbstractFactory , Command , Strategry and etc....

JEE ---> good for java based Layered Applications like web applications, Enterprise Apps and etc..
JEE Patterns are given by Sun Ms inspired by GOF Patterns.. to address of problems of JEE based layered Applications developement...

==>JEE patterns must be implemented in Java Language..

DAO , Compositive View, FrontController , ViewHelper and etc...

Desing Pattern Course Covers 25+ patterns
(Best of GOF Patterns)
(Best of JEE Patterns)

natarazjavaarena ----->FB group name
natarazjavaarena@gmail.com

PAttern Identification

->Candidate Pattern ----> Standard Design Pattern..

PAttern Template/ Pattern Characterstics documents...

Pattern NAme::
Problem ::
Forces ::
Solution ::
  (a) Structure :: Diagram
  (b) Strategy :: Psuedo code /sampe code
Consquences ::
related patterns::

PAtten Catalog

--->Maintains categorization of design patterns...

GOF Pattern Catalog contains
(a) Creational Patterns :: Solutions towards objs creation process
(b) Structural Patterns :: To compose big objects as combination of multiple objects..
(c) Behavioural Patterns :: Talks about communiation between classes / objects..

Loose coupling :: if the degree of dependency is less b/w two comps then they are called loosely coupled comps..
Tight coupling :: if the degree of dependency is more b/w two comps then they are called tightly coupled comps..

JEE Pattern Catalog

PRoblems of keeping multiple logics in single java class
(a) Clean seperation b/w logics will not be there.. So looks very clumzy
(b) The modifications done in one kind of logics will effect other logics..
(c) Maintence and enhancement of the project becomes complex..
(d) Parellel development is not possible , So productivity is bad
(e) It is not industry standard

To overcome thse problems use Layered Applications i.e keep diff logics in diff layers/classes and make them interacting with each other..

A Layer is a logical partiation of the Project having one or more classes/files holding certain type logics...

Advantages of Layered Applications
(a) Clean seperation b/w logics becoz of multiple layers .. So code is not clumzy
(b) The modifications done is one layer logics does not effect other layer logics
(c) Maintence and enhancement becomes easy
(d) Parellel developmnent is possible ,So productivity is good
(e) It is industry standard...

JEE Pattern Catalog
(a) Presentation tier patterns ::Related to Presentation logics
(b) Business tier patterns :: Related to business logic developemnt
(c) Integration tier patterns :: Related to Persisntence logic and related to interacting external services/sytstems like Gpay,Paypal and etc..

A type JEE Project maintains total 5 tiers

Client tier	Presentation tier	Business tier	Integration tier	Resource tier
(browser/ applet/ desktopApp)	(servlet,jsp)	(RMI,EJB,java classes, Spring)	(jdbc,jms, hibenrate spring, webServices)	(DBs, External sys)

Singleton Class

=>The java class that allows us to create only one object in any situation is called singleton class..

Problem:: Creating multiple objects for a java calss in these situations gives memmory issues, cpu utilization issues and performence issues..
(a) When class is not having any state
(b) when class is having only read-only state
(c) when class is having sharable state across the multiple other classes..

Solution :: Singleton class :: Create only object and use that object for mutliple times
|--->Does not allow to create more than 1 object in any situation..

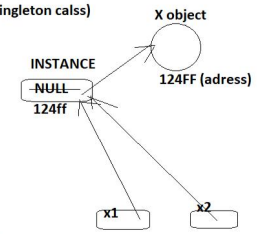
=>If we create only object for normal java class though it allows to create more objects then that class not called as singleton class... Singleton class should not allow to create more than 1 object though we are trying to create more objects..

=>SErvletContainer creates only object for our Servlet class it does not bean.. our Servlet class is singleton class.. ServletContainer just not intrtesed to create more than one object though our servlet class allow to create.

java.lang.Runtime , java.awt.Desktop , org.log4j.Logger are pre-defined singletin calss..

X ---->use-defined class (I want to make it as singleton calss)

```
public class  X{
  private static   X  INSTANCE;

  private X(){
  }

  public  static  X   getInstance(){

     if(INSTANCE==null)
        INSTANCE=new X();
    return INSTNACE;
   }
  }
```




```
outside
========
X  x=new X();  (x)

X  x1=X.getInstance();
X  x2=X.getInstnace();

x1.hashCode()  x2.hasCode()    //same hashCode

x1==x2   (true)
```

To develop singelton java class with minimum standards

(a) Class must be public (To make it visible outside the packages)
(b) Take only private constructor(s) (To stop new operator based object creation outside of the class)
(c) pubic static factory method (To check and create ,return that single object)
(d) private static reference variable of same class (INSTANCE) (To hold that one object ref for the verification of object creation)

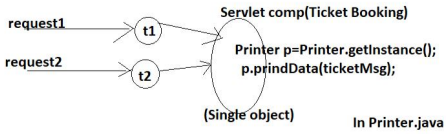
usecase:: To make all the employee company using same Printer ,, we must Printer class as singleton java class...

```
SingleDP  (EJP)
 |-->src
     |----->com.nt.sdp
              |--->Printer.java
     |--->com.nt.test
              |---->SingletonTest.java
```

ctrl+shift+c --> to enable/disable single line comments
ctrl+shift+/ --> To palce multiline comment
ctrl+shift+\---> To disable multiline comment
sysout + ctrl +space :: for S.o.p
systrace +ctrs+space :: for s.o.p(<message>)

=>Do not decide wheather object is created or not based on constructor execution... becoz when objects are created through deserialization and cloning no constructor executes...
=>when we create sub class object abstract class .. along with sub class constructor, the abstract class constructor executes.. but it does not mean obj is created for abstract class..

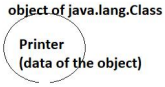
Singleton Java class in Multithread Env..




```
In Printer.java
===============

public  static Pinter  getInstnace(){
                          t1
  if(INSTANCE==null)
       INSTANCE=new Printer();
  return INSTANCE;
}
```

Solution for Multithreded env.. of Singleton java class is
(a) Use Sychronization (best) (Synchronized block is good with double null check)
(b) Eager Instantiation

Pinter.class --->Gives Object of java.lang.Class having Printer class name as the data of object...




```
//static factory method
      public  static  Printer  getInstance() {
        if(INSTANCE==null) {                    //1st NULL check
                synchronized (Printer.class) {
                    if(INSTANCE==null) //2nd NULL check
                          INSTANCE=new Printer();  //Lazy Instantiation  Good Pratice
            }//synchroized
         }//if
             return INSTANCE;
      }//method
```

Breaking singleton class Patterin using cloning concept

natarazjavaarena --->FB group

=>The Process creting new object based on existing object having existing object state as the intial state of new object is called cloning ...

=>In Java clong is possible only on cloneable objects... objects become cloneable only when their classes implements java.lang.Cloneable(I) (Which is an empty interface and marker interface)

=> The interface that makes underlying JVM/Container/framework to provide special runtime capabilities is called marker interface...

=>Most of the marker interfaces are empty interface.. it does not mean every empty is marker interface...

eg:: Cloneable (I),Serializable (I) and etc..

=> By seeing marker interface implementation the uneryling JVM/Container/Framewrok start providing special runtime capablities to the object of class.. not by using methods of marker interface..

To do cloning

(a) make the object as cloneable object
(The class of the object must implement java.lang.Cloenable(I))
(b) call clone() method of java.lang.Object class on existing cloneable object to get new object.

clone() is protected method java.lang.Object i.e it can be involed only from direct sub classes of java.lang.Object class..

Since the new object/duplicate created through cloning process will have invoking existing object state object as initial state.. So there is not need for seperate intialization for the object that is created through cloning.. So JVM does not invoke constructor..

To stop cloning on singleton java class object... override clone() as show below in singleton java class

```
//solving cloning problem of singleton (recomanded)
    @Override
    public Object clone()throws CloneNotSupportedException {
        throw new CloneNotSupportedException("Cloning is not allowed on this signleton java class");
    }

    /*//solving cloning problem of singleton (not recomanded)
    @Override
    public Object clone()throws CloneNotSupportedException {
        //return INSTANCE;
        return this;
    }*/
```

Breaking Singleton using DeSerialization

Serialization

It is the process of converting object data into bits and bytes.. So that the same object data can be sent over the network as bits and bytes or can be written files as bits and bytes...

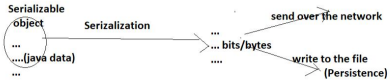
=>Flipart.com sends Card details to Paypall as Serialziable object data
=> Mobile games writes top 10 scores to files as Serilizable object data...

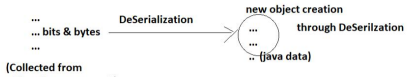
=>SErialization is possible only on Serializable objects.. i.e JVM connverts only Seraillizable objects data into bits and bytes.
=> Implementation class objects of java.io.Serializable(I) markter interface are called Serializable objects...

natarazjavaarena ---->FB Group

Deserilization

=>Converting bits and bytes into java data and creating new object having that java data as initial values is called Deserialization..
=> In DeSerialization process , constructor will not be executed though object is created. becoz data gathered through DeSerialization is already intialized with Object. So there is no need seeperated constructor based initalization to that object...





(Collected from either file or network)

We need the support of Streams to write the Data of Serialized object to File and also for DeSerilization

ObjectOutputStream -----> for Serialization
ObjectInputStream -----> for DeSerialization

High Level Streams..

=>Hight level streams internally take the support of LowLevel streams to complete read and write operations of the dest files...

FileInputStream,FileOuputStream and etc.. LowLevelStreams ... These stream can deal with only bytes and chars...

ObjectOutputStream,ObjectInputStream,DataOuputStream,DataInputStream and etc.. High LevelStreams .. These streams can work with diff data types values directly.. becoz they will be converted into bytes and chars thrugh low level streams..

Sample code for Serialization

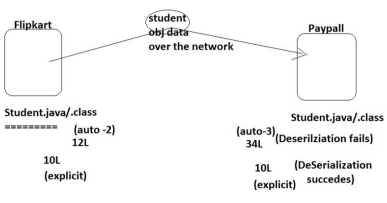
```
public class Student implements Serilizable{
  private static final long serialVersionUID=43434L;  // recomanded to place explicitly..
 private int sno;
 private String sname;

 public void assignData(int sno,String sname){
    this.sno=sno;
    this.sname=sname;
 }

 //toString()
   ....
}
```

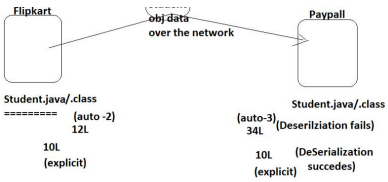
In App1.java (For Serialization)

```
Student st=new Student();
st.assignData(101,"raja");

ObjectOutputStream oos=
      new ObjectOutputStream(new FileOutputStream("stud.ser"));
 oos.writeObject(st);
  oos.flush();
  oos.close();
```

In App2.java (DeSerialization)

```
ObjectInputStream ois=
   new ObjectInputStream(new FileInputSTream("stud.ser"));

Student st=(Student)ois.readObject();
```





Solution ::
=======
override readResolve() method either returing already avaliable object or throwing exception.. in our singleton class ... This method is internally called by readObject(-) while performing deSerialization...

note: readResolve() is not declared in any interface or class of java api.. like main(-) method but it will be internally called by ois.readObject(-) on the invoking object of DeSerialization.. its signture is public Object readResolve()

```
In Printer class
==============
public Object readResolve(){
  //return this;
   return INSTACE;
}

  (or)
public   Object  readResolve(){   (best)
   throw new IllegalArgumentException("do not want to support DeSerialization");
}
```

note:: As part network /file related Deserlizatiion proecess the readResolve() method availablity will be verified in the Serilizable class , if not available new object will be created ..otherwise same/old object will be returned (or) exception will be thrown...

note:: As of now , there is no provision to stop Serialization of Serializable objects... but using readResolve() method we stop DeSerialization ..

Breaking singleton Using Reflection API

=>java.lang.reflect and its sub packages togather is called reflection API

=> Reflection API applications are like mirror devices which gives inter details given java class/interface/ enum/annotation dynamically and programatically and they can be used as inputs in the same application.

Limitation of new operator
==========

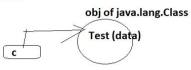
```
Test t=new Test();
```

==>It creates new object dynamically at runtime ,but expects the presence of class from compile time onwords.. i.e it can not create object for java class if the class name comes to app dynamically at runtime..

ServletContainer/Spring IOC container/EJB Container and etc.. are creating their component classes objects not by using new operator becoz they get their component classes dynamically at runtime... They use newInstance() of java.lang.Class for creating object...

create "Test" class object using newInstance() method

```
                                        args[0]
Class c=Class.forName("Test");
```

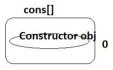


forName(-) is static factory method of java.lang.Class which makes the JVM to load given java class dynamically at runtime and return the object of java.lang.Class having the loaded name as data of the object .. So using "c" usisng multiple operations on that loaded class..

```
//create Objecct of loaded class
Object obj=c.newInstance();
 Test t=(Test) obj;
```

Accessing private constructior of singleton class outside class using reflection api and creating the object by breaking singleton behaviour

```
//Load the class
 Class  c=Class.forName("com.nt.sdp.Printer");
// get Constructors of the loaded class
  Constructor cons[]=c.getDeclaredConstructors();
//get Access to private constructor
 cons[0].setAccessible(true);  // breaking encapsulation...

 //create the object
  Printer p1=(Printer) cons[0].newInstance();
```



Each object of Constructor class represents one constructor of java class.

Solution for reflection api problem is

```
      private static boolean flag=false;
     //private constructor
          private  Printer() {
               if(flag==true)
                    throw new IllegalArgumentException("Object already created...");
               flag=true;
               System.out.println("Printer:: 0-param constructor");
          }
```

=>we can not reflection api singleton class brekage problem for ever.. becoz same reflection api can be used to access private flag variable and INSTANCE variable and can change their data..

Designing Perfect Singleton java class

```
package com.nt.sdp;

public class  Printer{
   private static Printer INSTNACE;
   private static boolean falg=false;
    private Printer(){
       if(flag==true)     //To stop reflection api based constructor access
          throw new IllegalArgumentException(" Object is already created");
       flag=true;
    }
   //static factory method
   public   static Printer getInstance(){

     if(INSTANCE==null){
        synchronized(Printer.class){
          if(INSTNACE==null)
             INSTANCE=new Printer();
        }//synchronized
     }// if
      return INSTNACE;
   }//getInstance

    //To stop cloning
   public Object clone(){
     throw new CloneNotSupportedException("cloning is not allowed");
   }
        //To stop DeSerialization
   public  Object readResolve(){
     //throw new IllegalArgumentException("DeSerilization not allowed");
     return INSTANCE;
   }

    //b.method

    public  void printData(String data){
      S.o.p(data);
    }
}
```

3 Approaches of developoing singleton java class

(a) Take normal class and add all constraints code as shown above (Traditional approach)
(b) Create Singleton java class object inside nested inner class (static inner class) of singleton java class
(c) Take Enum as singleton

Types of inner clases
==========
(a) Normal inner class (b) nested/static inner class (c) Local inner claass (d) Anonymous inner class

Developing Singleton Java class as ENUM

```
public enum Printer{
    INSTANCE;

  public void printData(String msg){
    S.o.p(msg);
  }
}
```

Client/Testapp

```
Printer p1=Printer.INSTANCE;
Printer p2=Printer.INSTNACE;
  S.o.p(p1.hashCode()+"  "+p2.hashCode());
  s.o.p("p1==p2?"+(p1==p2))
```

Advantages of taking Singleton as Enum

(a) Easy to develope
(b) It performs eager INSTNTITATION of its constants , So no multithreading issuses
(c) Enum does not allow cloning in any angle..
(d) Enum is inherintly Serilizable..But they allow to create only object per jvm, so there is no DeSerialization problem..
(e) we not create Enum objects through reflection api
if we try to create then it gives java.lang.IllegalArgumentException: Cannot reflectively create enum objects

Disadvantages

(a) Eager Instantiation of Constants.. may waste the resources thouhg object is not required...
(b) We should not place more than one constant in singleton Enum otherwise singleton behaviour will be broken
(c) we can not make other classes extending from singleton enum... (becoz all enums are final classes internally)

Best Code Tranditional Singleton Java class code

```
package com.nt.sdp;

public class Printer{
   private static Printer INSTNACE;
   private static boolean falg=false;

     private Printer(){
        if(flag==true)     //To stop reflection api based constructor access
            throw new IllegalArgumentException(" Object is already created");
         flag=true;
      }
     //static factory method
    public   static Printer getInstance(){

      if(INSTANCE==null){
         synchronized(Printer.class){
           if(INSTNACE==null)
              INSTANCE=new Printer();
         }//synchronized
     }// if
       return INSTNACE;
    }//getInstance


     //To stop cloning
    public Object clone(){
      throw new CloneNotSupportedException("cloning is not allowed");
    }
         //To stop DeSerialization
    public  Object readResolve(){
      //throw new IllegalArgumentException("DeSerilization not allowed");
      return INSTANCE;
     }

     //b.method

     public  void printData(String data){
       S.o.p(data);
     }
}
```

Best Singleton java class code usig nesteed inner class approach

```
public class Printer  {

    private static boolean flag=false;

    private Printer() {
        if(flag==true)
            throw new IllegalArgumentException("obj already created");
        flag=true;
        System.out.println("Printer:: 0-param constructor");
    }


    //static factory method
    public static  Printer  getInstance() {
        return  PrinterHolder.INSTANCE;
    }

    //b.method
    public  void printData(String msg) {
        System.out.println(msg);
    }

    //nested/static inner class
    private static class PrinterHolder{
        private static Printer INSTANCE=new Printer();

    }//class

    //to stop cloning
    public  Object clone() throws CloneNotSupportedException{
        throw new CloneNotSupportedException("cloning is not allowed");
    }


    //To stop Deserialization
    public  Object readResolve() {
        return PrinterHolder.INSTANCE;
        //throw new IllegalArgumentException("DerSerialization is not allowed");
    }

}
```

Best Enum Based Singleton Java class

```
public enum Printer  {

    INSTANCE;

    //b.method
    public  void printData(String msg) {
        System.out.println(msg);
    }

}
```

ClassLoaders
==========

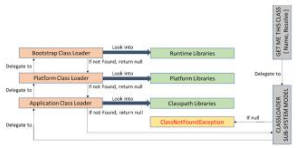
JVM internally uses ClassLoader SubSystem to load the java classes...

upto java 8 we have 3 classLoaders and from java9 we have 2 class loaders

(a) Bootstrap ClassLoader (Can load the classes from rt.jar file of <java_home>\jre\lib folder)
(b) Extension classLoader (Can load the classes from jar files added to <java_home>\jre\lib\ext folder)
(c) System/Application/Classpath ClassLoader
(Can load the classes from the directories and jar files
add to CLASSPATH env.. variable)

note: From java9 onwords Extension classLoader is removed from java .. So we do not "ext" folder in jdk's installation...

=>These classloaders are hierarchal...



3 Principles of ClassLoader Sub System
==========

(a) Delegation Principle :: Since ClassLoaders are hierarchal , So if class is not loaded by one ClassLoader then it delages another ClassLoader in the hierarchy
(b) Visibility Principle :: The classes loaded by Parent ClassLoader are visible to Child class Loaders but reverse is not true.
(c) Uniqueness Principle :: Only One ClassLoader will load the given class.. i.e same class will not be loaded by multiple class loaders...

=>Every ClassLoader is a class that extends from java.lang.ClassLoader class ..
=>To develop CustomClassLoader we need to take java class that extends from java.lang.ClassLoader class...
=> We are some readymade Custom ClassLoaders

eg: java.net.URLClassLoader ... (Useful to load classes from jars/directories that are not added to CLASSPATH)

Bydefault Custom class Loader is not in the hierarchy of STandard ClassLoaders (3)

With in a JVM we can break singleton java class behaviour by taking the support of CustomClassLoader... i.e We can create second object in the same jvm for singleton class with support of Customer ClassLoader...

https://github.com/nataraz123/NTDP413REpo.git

example
========

place singleton classes in jar file like sdpnit.jar file.. and give its location to Custom Class Loader

cmd> jar cf sdpnit.jar .

```
com.nt.sdp1.Printer p1=null;
Object p2=null,p3=null;
URLClassLoader loader1=null,loader2=null;
ClassLoader loader=null;
//get First object
p1=com.nt.sdp1.Printer.getInstance();

loader=p1.getClass().getClassLoader();
System.out.println(loader);

//create Custom ClassLoader and Load Singleton class and also create obj using reflection api
// loader1=new URLClassLoader(new URL[] {new URL("file:/E:/WorkSpaces/spring/NTDP914/NTDP912withoct4/SingletonDP2-ClassLoaders/sdp.jar")},p1.getClass().getClassLoader());
loader1=new URLClassLoader(new URL[] {new URL("file:/E:\\WorkSpaces\\spring\\NTDP914\\SingletonDP4-CustomClassLoader\\sdpnit.jar")},null); //Idependent ClassLoader
//loader1=new URLClassLoader(new URL[] {new URL("file:/E:\\WorkSpaces\\spring\\NTDP914\\SingletonDP4-CustomClassLoader\\sdpnit.jar")},loader);
Class<?> clazz1=loader1.loadClass("com.nt.sdp1.Printer");
Method method=clazz1.getDeclaredMethod("getInstance",new Class[] {});
p2= method.invoke(null);
```

```
//create Custom ClassLoader and Load Singleton class and also create obj using reflection api
loader2=new URLClassLoader(new URL[] {new URL("file:/E:\\WorkSpaces\\spring\\NTDP914\\SingletonDP4-CustomClassLoader\\sdpnit.jar")},loader1);
Class<?> clazz2=loader2.loadClass("com.nt.sdp1.Printer");
Method method1=clazz2.getDeclaredMethod("getInstance",new Class[] {});
p3= method1.invoke(null);

System.out.println(p1.hashCode()+"  "+p2.hashCode()+"  "+p3.hashCode());
```

Can we develop 100% Perfect singleton java class?

Not Possbile

=> Using traditional , inner class approach we can develop for 98%
=> Using enum we can develope for 99%

When should we take java class as singleton java class?
(or)
Realtime usecases of singleton java class

Ans)
==== a)When class is not having state
b) when class is having only ReadOnly state
c) When class is having sharable state across the multiple classes in synchronized enviroment.. (usecase is cache)

a)When class is not having state

problem::
==========
```
public class Arithmetic{

  public int sum(int x,int y){
     return x+y;
  }
}
```

solution::
========
```
public enum Arithmetic{
  INSTANCE;

  public int sum(int x,int y){
    return x+y;
  }
}
```

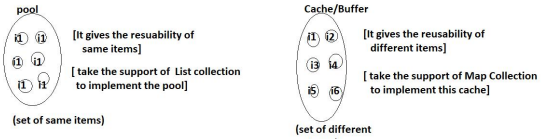
b) when class is having only ReadOnly state

problem:
=======
```
public class Circle{
  private final float PI=3.14f;

  public float calcArea(float radius){
     return PI*radius*radius;
  }
}
```

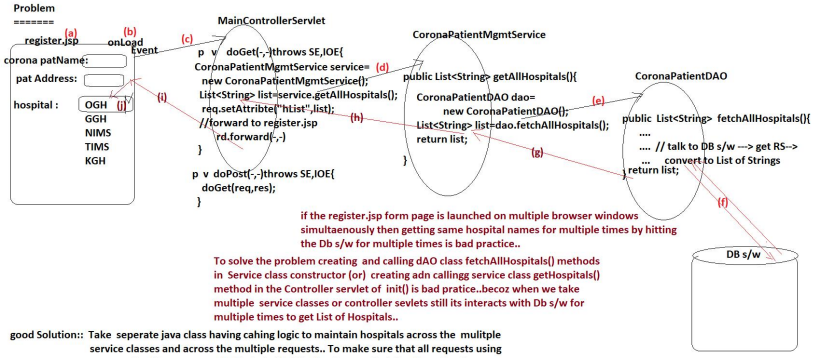
solution::
========
```
public enum Circle{
  private final float PI=3.14f;
  INSTANCE;
  public float calcArea(float radius ){
     return PI*radius*radius;
  }
}
```

```
public class Circle{
  private static Circle INSTANCE;
  private final float PI=3.14f;

  private Circle(){ }

  public static Circle getINSTANCE(){
    if(INSTANCE==null)
       INSTANCE=new Circle();
    return INSTANCE;
  }

  public float calcArea(float radius ){
     return PI*radius*radius;
  }
}
```

c) When class is having sharable state across the multiple classes in synchronized enviroment.. (usecase is cache) and ServiceLocator



Problem
=======



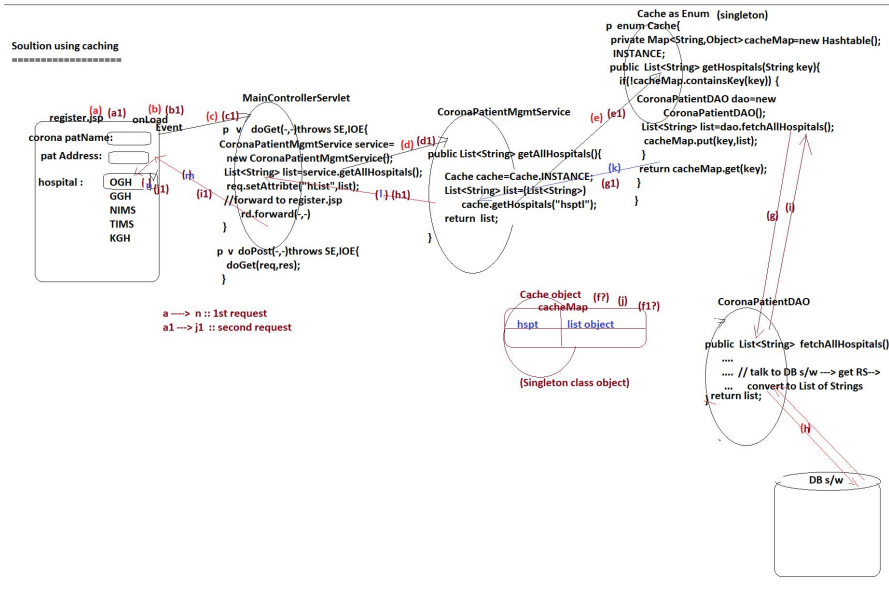
if the register.jsp form page is launched on multiple browser windows simultaenously then getting same hospital names for multiple times by hitting the Db s/w for multiple times is bad practice..

To solve the problem creating and calling dAO class fetchAllHospitals() methods in Service class constructor (or) creating adn callingg service class getHospitals() method in the Controller servlet of init() is bad pratice..becoz when we take multiple service classes or controller sevlets still its interacts with Db s/w for multiple times to get List of Hospitals..

good Solution:: Take seperate java class having cahing logic to maintain hospitals across the mulitple service classes and across the multiple requests.. To make sure that all requests using same cache it is reocomanded to take that Cache class as singleton java class/enum

---

Soultion using caching
==========



a ---> n :: 1st request
a1 ---> j1 :: second request

Factory Pattern /Simple Factory Pattern
==========

=>We can create object using new operator with /with out params/args.
We can create object using newInstance() method
We can create object having dependent object..
=> Instead of making multiple developers knowing the complex process of certail class obj creation with /with out dependents.. if we provide Factory to them.. It takes care of object creation process and simplifies job for developers..

=>Factory Pattern provides absttraction on Object creation process.. and creates,returns one of several related class objetc based on the data/inputs we supply.. Several related classes means all these classes should have common super class or common implementing interface...

=>BiscutFactory gives busicuts with out exposing their creational process
=>CarFActory gives cars with out exposing their manifacturing process
=> DriverManager.getConnection(-,-,-) is based factory pattern becoz it returns jdbc con object as the object created for one impl class of Connection(-) based on the data (url, user,pwd) that are supplied. In this it abstracts how it is giving dependent objs and how it is using them in connection object creation...

=>Spring IOC container is given based Factory Pattern.. factory.getBean(-)/ctx.getBean(-) gives Spring bean obj having depedent objects with out exposing their construction process..

=> All the statements in jdbc code that are given to create connection , statement ,ResultSEt MEtaData objects are working based on factory pattern..
=> In Hibernate SessionFactory ,Session object we create based on Factory pattern...

Connecction con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","...","...");
gives the object of T4CConnection class
(or)
Connecction con=DriverManager.getConnection("jdbc:mysql:///db1","...","...");
gives the object of MysqlConnection class..

Problem:: creating main and dependent calsses objs manually by knowing thir consturctor process and assigning dependent objs to main class maually in complex process...

solution:: use Factory Pattern.. havng hepler class factory method which takes the above process internally.. and gives main object having dependent object.. to developers when ever it is required.

While creating Car object based given data (car type) we should how to create their dependent objs like (Tyre obj, AC object and etc..) and we should also know how to use these dependent objs in main object creation... To simplify this use Factory pattern..

In Factory Pattern class , we generally take one static factory method having capality of returning one of several related classes objects based on the data that is passed..

```
public class CarFactory {

	//static factory method having factory pattern logic
	public  static  Car getCarInstance(String carType) {
		Car car=null;
		Tyre tyre=null;
		if(carType.equalsIgnoreCase("standard")) {
			tyre=new ApolloTyre();
			car=new StandardCar(tyre);
		}
		else if(carType.equalsIgnoreCase("luxory")) {
			tyre=new MechallinTyre();
			car=new LuxoryCar(tyre);
		}
		else if(carType.equalsIgnoreCase("sports")) {
			tyre=new MRFTyre();
			car=new SportsCar(tyre);
		}
		else {
			throw new IllegalArgumentException("invalid Car type");
		}

			return car;

	}//method

}//class
```

Factory Method desing Pattern

==========

=> It defines set rules and guidelines to standardize the object creation process when multiple factories are creating objects of same family classes..

Bajaj -->Bike Manifacturing company
|---------->NagPurBajaFactory
|---------->ChennaiBajajFactory

if differenf factories are following different standards in bike manifacturing then they will be differences in quality.. To overcome this problem we go for Factory Method desing pattern which defines set of rules /standards for all the factories to create objects having same quality.

NagpurBajajFactory
|----> assemble()
|---->paint();
|----> roadTest();

ChennaiBajajFactory
|---->paint()
|--->assemble()
|--->roadTest()
|--->oiling()

=>The method that creates and returns obj is called factory method..
=>Factory Method design pattern internally java's factory method.. but it is no way related to java's factory method. It is all about providing set of rules,guidelines to standardize the object creation process when multiple factories creating object of same family classes (Classes having common supper class or implementing common inferface)

Factory Method DP implementation is all about making multiple factory classes extends common super class.. and that super defines set rules as standards for object creation

```
public abstract class BajajFactory  {  //Factory method DP

    public abstract void paint();
    public  abstract  void assemble();
    public  abstract void roadTest();
    public  abstract void oiling();
    public  abstract BajajBike buildBike(String model);

    public  BajajBike  manifactureBike(String model){
        BajajBike bike=null;
        bike=buildBike(model);
            paint();
            assemble();
            roadTets();
            oiling();
        return bike;
    }
}
```

Abstract FActory Design Pattern

It is called super factory or factory of factories..

=>It provides one level higher abstraction on the top of factory.. i.e it make sure that multiple factories are returning related class objects.. It returns one of serveral factory class objects and that factory can be used to return one several related classes object.

DAO ---> Data Access object (The java class that contains purely persinstence logic)

```
          Course Info              Student info
             |-->DB                    |---->DB
             |-->Excel                 |--->Excel

         CourseExcelDAO             StudentExcelDAO
         CourseDBDAO                StudentDBDAO

         DBDAOFactory               ExcelDAOFactory
           |-->returns either          |--->returns either CourseExcelDAO or StudentExcelDAO
           CourseDBDAO or               class obj based on the data that is supplied..
           StudentDBDAO class obj
           based on the data that
           is supplied
                                                CouseExcelDAO
             StudentDBDAO
```

if use Simple Factory supports there is possibility of getting StudentDAO pointing to DB and CourseDAO pointing to Excel ... i.e we are going work with unrelaated class objects.. To overcome this problem create super factory called Abstract Factory..

Solution

```
public interface DAOFactory{
  public  DAO createDAO(String type);
 }

//factory1
public class DBDAOFactory implements DAOFactory{

   public DAO createDAO(String type){
        ....
        ....
     return dao;
    }
  }

//factory2
public class ExcelDAOFactory implements DAOFactory{

  public DAO  createDAO(String type){
      ...
      ...
   return dao;
  }
}

//Abstract Factory/super factory/ factory of factories
public class DAOFactoryCreator{

public static DAOFactory buildDAOFactory(String type){
       DAOFactory factory=null;
      if(type.equalsIgnoreCase("excel"))
           factory=new ExcelDAOFActory();
      else if(type.equalsIgnoreCase("DB"))
           factory=new DBDAOFactory();
       else
          throw new IllegalArguementException("Invalid factory type");
  return factory;
 }
}
```

ClientApp

```
DAOFactory factory=DAOFactoryCreator.buildDAOFactory("excel");
  DAO dao1=factory.createDAO("student");
  DAO dao2=factory.createDAO("course");
   dao1.insert();
   dao2.insert();
```

Abstract Factory or Super factory returns one of several related factories where each factory returns one of several related classes objects..

TEmplateMethod DP
==========

```
Task1 ---> x()
           y();     //we should remember multiple
           z();     method names and their sequence of invocation..
           a();     to complete the task
```

template method

```
t1(){          (template method)
  x();
  y();
  z();
  a();
}
```

task1 ----> t1()

Template method is a super class method definning algorithm by calling multiple methods in a sequnece to complet task.. In that algorithm keeps common methods with definitation.. and some methods as abstract methods so that sub classes can define..

```
Fresher Recruitment drive
      (a) Java Fresher
      (b) .net fresher
      (c) php fresher          Fresher RecruitmentProcess
                                   |--->conduct APT Test
                                   |---> Conduct GD
                                   |---> Conduct Technical interview
                                   |---> conduct  System Test
                                   |---> conduct  HR
```

```
//super class
==========
abstract
public class  HiringFresher{
 public  boolean  conductAptitudeTest(){
    ....
  return true/false;
 }
 public  boolean  conductGD(){
   ...
  return true/false;
 }
 public abstract  boolean  conductTechnicalTest();
 public  abstract boolean  conductSystemTest();
  public  boolean conductHRInterview(){
    ...
  return true/false;
 }

  //Template method DP
  public boolean  fresherRecruitmentProcess(){
    if(conductAptitueTest())
       if(conductGDTest())
            if(conductTechnical())
                 if(conductSystemTest())
                      if(conductHRInterview())
                           returnt true;
      return fasle;
  }

public class  HiringJavaFresher extends  HiringFresher{
   public  boolean conductTechnicalInterview(){
       S.o.p("Conducting java interview");
      return true/false;
    }
   public  boolean conductSystem(){
       S.o.p("Conducting java sytem test");
      return true/false;
    }
}
                    DotNet
public class  Hiring    Fresher extends  HiringFresher{
    public  boolean conductTechnicalInterview(){
        S.o.p("Conducting.net  interview");
       return true/false;
     }
    public  boolean conductSystem(){
        S.o.p("Conducting.net  sytem test");
       return true/false;
     }
}
public class  HiringPHPFresher extends  HiringFresher{
    public  boolean conductTechnicalInterview(){
        S.o.p("Conducting php interview");
       return true/false;
     }
    public  boolean conductSystem(){
        S.o.p("Conducting php sytem test");
       return true/false;
      }
}

public class  FresherHiringFactory{

  public static  HiringFresher  getInstance(String jobDomain){
      HiringFresher fresher=null;
       if(jobDomain.equalsIgnoreCase("java"))
          fresher=new HiringJavaFresher();
       else if(jobDomain.equalsIgnoreCase(".net"))
            fresher=new HiringDotNetFresher();
       else  if(jobDomain.equalsIgnoreCase("php"))
              fresher=new  HiringPhpFresher();
        else
            throw new IllegalAccessException("invalid  jobDomain");
   }
}

public class  Company1{

 p  s  v  main(String args[]){
    HiringFresher fresher=null;
    fresher=HiringFresherFactory.getInstance("java");
    fresher. fresherRecruitmentProcess();
  }
}
```

pre-defined methods acting as Template methods
==========

- All non-abstract methods of java.io.InputStream, java.io.OutputStream, java.io.Readerand java.io.Writer.
- All non-abstract methods of java.util.AbstractList, java.util.AbstractSet and java.util.AbstractMap.
- javax.servlet.http.HttpServlet, all the doXXX() methods by default sends a HTTP 405 "Method Not Allowed" error to the response. You're free to implement none or any of them.

Builder design Pattern

==========

Create complex object by using multiple small objects step by step by defining the a Process ,So that same process can be used to create diff complex objects of same category..

eg: Indian Meal object (soup obj, starter object, MainCourse object ,Desert object, pan object)
- |---> North Inidan meal
- |---> South Indian Meal
- |--->Jain Meal
- |--->Telugu Meal
- |--->Oriya Meal
- and etc...

eg:: Car object (Tyres obj , Engine object , body obj, Interior obj)
- |--->Ecomony Car
- |---->Luxory Car
- |--->Sports Car
- |--->Racing Car

4 comps of Builder DP

==========

(a) Proudct (Final complex object ---->Hou se) to get

(c) Buider (Interface having methods get small objects and final product object)
(like HouseBuilder)

(c) ConcreteBuilder ---> Implements Builder interface..
like WoodenHouseBuilder, IglooHouseBuilder, BrickHouseBuilder

(d) Director/Creator ----> actiavates one or other builder and builds the product..
/Delegator

House (Product)	HouseBuilder (I) (Builder)	
|----> Foundation object	ConcreteHouseBuiler	concreteBuilders
|----> Structure Object	IglooHouseBuilder	
|----> Roofing Object	WoodenHouseBuilder	
|-----> Interior object	MarbleHouseBuilder	

CivilEngineer (Director/Delegator)

HouseFactory

ClientApp

=>In Hibernate buildSessionFactory() method of Configuration class gives the complex object SessionFactory object having multiple sub objects like dialect , Datasource, caching and etc.. based on the cfgs done in xml file or properties file.. So this comes under Builder DP.

SessionFactory factory=cfg.buildSessionFactory()

=> Session ses=factory.openSession() falls under factory desing pattern

Decorator/wrapper Pattern

Using Inheritence for extension is always complex process.. That will make us to write more and more classes while adding new functionalaties every time..

```
public interface IceCream{
  public void prepare();
}

public class VanilaIceCream implements IceCream{

  public void prepare(){
     S.o.p("preparing vanila incream....");
  }
}

public class ButterScotchCream implements IceCream{

  public void prepare(){
     S.o.p("preparing butterscotch incream....");
  }
}

public class DryfuitVanilaIceCream extends VanilaIceCream{

  public void prepare(){
     super.prepare();
     addDryFruits();
  }
  private addDryFruits(){
     S.o.p("adding dryfruits....");
  }
}

public class DryfuitButterScotchIceCream extends ButterScothcIcream

  public void prepare(){
     super.prepare();
     addDryFruits();
  }
  private addDryFruits(){
     S.o.p("adding dryfruits....");
  }
}

public class HoneyVanilaIceCream extends VanilaIceCream{

  public void prepare(){
     super.prepare();
     addHoney();
  }
  private addHoney(){
     S.o.p("adding honey...");
  }
}

public class HoneyButterScothcCream extends ButterScotchCream{

  public void prepare(){
     super.prepare();
     addHoney ();
  }
  private addHoney(){
     S.o.p("adding honey....");
  }
}
```

2-->basic flavours
4 -->topings

in all angles we need to develop 240+ classes using inheritence concept supporting all permitations and combinations..

To solve the above problem use Decorator or Wrapper desing pattern.. which say use Composition support to add extra functionalaties to the object.. So will less no.of classes we can go for all combinations..

4 importan comps of Decorator/Wrapper Desing pattern

(a) Component Interface (IceCream(I))
(b) Concrete Component classes (VanilaIceCream , ButterScotchIceCream)
(c) Decorator (Abstract class having Compoennt interface ref variable and also implements component Interface)
(d) ConcreteDecorator (Extends from Decorator . adding addtional functionlalties like HoneyDecorator , ChocolateDecorator, DryFruitDecoarator, SambarDecorator)

IceCream.java (Component interface)

```
public interface IceCream{
  public void prepare();
}
```

VanilaIceCream.java (concrete component class)

```
public class VanilaIceCream implements IceCream{
    public void prepare(){
        S.o.p("preparing vanila Icecream");
    }
}
```

ButterScotchIceCream.java (concrete component class)

```
public class ButterScotchCream implements IceCream{
    public void prepare(){
        S.o.p("preparing butter scotch Icecream");
    }
}
```

IceDecorator.java (Decarator)

```
public abstract class IceCreamDecorator implements IceCream{
  private IceCream iceCream;

  public IceCreamDecorator(IceCream iceCream){
     this.iceCream=iceCream;
  }

  public void prepare(){
     iceCream.prepare();
  }
}
```

HoneyIceCreamDecorator.java (ConcreteDecorator)

```
public class HoneyIceCreamDecorator extends IceCreamDecorator{

   public HoneyIceCreamDecorator(IceCream icream){
      super(iceCream);
   }

   public void prepare(){
      super.prepare();
      addHoney();
   }

   private void addHoney(){
     S.o.println("adding hoeny");
   }
}
```

DryFruitIceCreamDecorator.java (ConcreteDecorator)

```
public class DryFruit IceCreamDecorator extends IceCreamDecorator{
   public DryFruit IceCreamDecorator(IceCream icream){
      super(iceCream);
   }

   public void prepare(){
      super.prepare();
      addDryFruit();
   }
   private void add DryFruit (){
     S.o.println("adding DryfrUll");
   }
}
```

Client app

```
  IceCream cream1=null;

  cream1=new DryFruitsIceCreamDecorator(new HoneyDecorator(new VanilaIceCream());

  IceCream cream2=new VanilaIcream();

  IceCream cream3=new DryFruitDecorator(new VanilaIcream());



  BufferReader reader=new BufferReader(new InputStreamReader(System.in));

  DataInputStream dis=new new DatanputSTream(new FileInputsTream("abc.txt"));
```

Reader(I)/Writer(I) --->Component Interface
FileReader,FileWriter --->concrete components

BufferReader,BufferWriter, Byte[]Reader,Byte[]Writer | ConcreateDecorators

Decorator (recognizable by creational methods taking an instance of some abstract/interface type which adds additional behaviour)

- All subclasses of java.io.InputStream, OutputStream, Reader and Writer have a constructor taking an instance of same type.
- java.util.Collections, the checkedXXX(), synchronizedXXX() and unmodifiableXXX()methods.
- javax.servlet.http.HttpServletRequestWrapper and HttpServletResponseWrapper
- javax.swing.JScrollPane

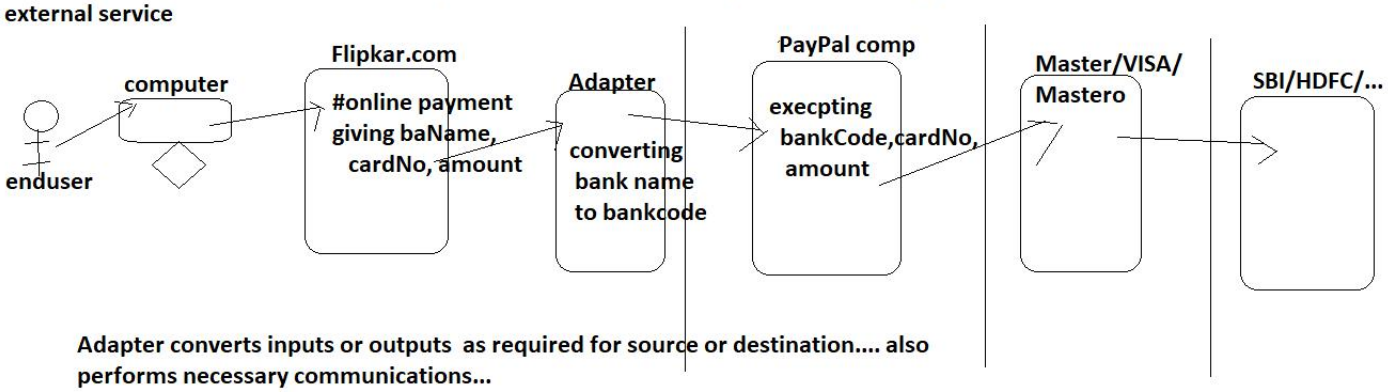
Adapter Design Pattern
==========

--> IT makes two incompitable interfaces(Apps/Projects) compitable with each other...

->To operate US Eletronic devices in india we purchase adapter which converts 120 v to 240vs

-> All the e-commerce websites will use payTM, PhonePe , Google Pay ,PayPall and etc.. as extenal services for online digital payment..

-> All stock brokering websites like crickBuzz, crickInfo and etc.. take support ICCScoreComp as external service



Adapter converts inputs or outputs as required for source or destination.... also performs necessary communications...

note:: While developing extenal comp... It always recomanded to take Interface , Impl class model so that we can expose that interface to remote clients to hold/refer external comp ref.. by achieving loose coupling...

Limitations of keeping logics directly in Local service class to interact with external service
==========

(a) Logic will not be reusable acorss the mulitiple local service class
(b) if Extenal service comp details are changed.. then we need to distrub multiple local service classes
(c) converting inputs as required for external comp and converting outputs as required for external should be done by Local service...

To overcome these problems use AdapterDP...

note:: In standalone Apps ... it is recomanded to take Adapter class obj as instance variable... in Local service class becoz only one user will operate the application a time...

In web applications... it is recomanded to take Adapter class obj as local variable in Local service class becoz multiple threads representing multiple requests will be there using adapter object..

note:: Instance variables of java class are not threadsafe .. where as local variables of java mehtod are thread safe..

Adapter (recognizeable by creational methods taking an instance of *different* abstract/interface type and returning an implementation of own/another abstract/interface type which *decorates/overrides* the given instance)

- java.util.Arrays#asList()
- java.util.Collections#list()
- java.util.Collections#enumeration()
- java.io.InputStreamReader(InputStream) (returns a Reader)
- java.io.OutputStreamWriter(OutputStream) (returns a Writer)
- javax.xml.bind.annotation.adapters.XmlAdapter#marshal() and #unmarshal()

two types of streams

1.byte streams
2.characters

```
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
```

character stream — adapter.. — byte stram

Fly weight
==========
Share the memory to support large number objects effeciently....

=> With Good coding algorithms we can use less memory for more operations and objects.

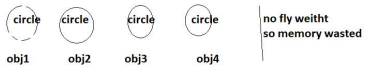
=>In paint app instead of using 100 objects fo Circle class to draw 100 circles.. this about using 1 object to draw 100 circle...

=> We need indentity Instrinic (sharable acorss the multiple objs)
state and extrinsic state (non-sharable)

Circle (class)
|--->lable -> "circle" (shrable---instrinsic state)
|---> radius,color, fillColor --->(non-sharable - extrinsic state)

-> take intrinsic state as member variables and extrinisic state as method params

```
public class Circle{
  private String lable="circle";  //intrinsic

  public void draw(int radius, String fillColor , String lineStyle){
     ....                                          extrinsic
     ....
  }                    obj1.draw(-,-,-) obj2.draw(-,-,-) obj3.draw(-,-,-) obj4.draw(-,-,-)
 }
```



no fly weitht
so memory wasted

Instead of 4 objects... take 1 object for Circle class call draw(-,-,-) method for 4 times to draw four ciles...

(1object)

```
obj.draw(34,"red","dotted");
obj.draw(4,"blue","dotted");
obj.draw(4,"yellow","plain");
obj.draw(14,"red"," dashed");
```

memory not wasted
becoz flyweight...

Problem
=========

Shape.java
==========

```
public interface Shape{
  public   void draw(int arg0, String fillColor, String lineStyle)
}
```

Circle.java
==========

```
public class class Circle implements Shape{
  private String label;
  public Circle(){
    label="circle";
  }
  public void draw(int radius, String fillColor, String lineStyle){
      S.o.p("drawing circle havng radius+" radius +" with fillcolor ::"+color+" ,line style::"+lineStyle);
  }
}
```

Square.java
==========

```
public class Square implements Shape{
  private String label;
  public Squre(){
    label="squre";
  }
  public void draw(int side, String fillColor, String lineStyle){
      S.o.p("drawing circle havng side legth+" side +" with fillcolor ::"+color+" ,line style::"+lineStyle);
  }
}
```

```
public class ShapeFactory{

  public static   Shape   getInstnace(String shapeType){
    Shape shape=null;
    if(shapeType.equals("circle")){
        shape=new Circle();
    }
    else if(shapeType.equals("square"){
        shape=new Square();
     }
    else{
      throw new IllegalArgumentException("invalid shape");
    }
    return shape;
  }
}
```

```
public class PaintApp{
  p  s v  main(String args[]){

    for(int i=1;i<=1000; ++i){
     Shape shape=ShapeFactory.getInstance("circle"))
      shape.draw(i+10, "red","dashed");
    }

     }
    }
```

Solution is Fly Weight DesignPattern that is Take factory to create relavent class objects by keeping the objects in cache..

```
public class ShapeFactory{
  private Map<String,Shape> cacheMap=new HashMap();
        synchronized
  public  static  Shape getInstance(String type){
      Shape shape=null;
    if(!cacheMap.containsKey(type)){
        if(type.equalsIgnoreCase("circle"))
          shape=new Circle();
        if(type.equalsIgnoreCase("square"))
          shape=new Square();
        cacheMap.put(type,shape);
     }
     return (Shape)cacheMap.get(type);
  }
}
```

cacheMap

circle	Circle class obj

```
public class PaintBrushApp{
  p  s v  main(String args[]){
    Shape shape1=null,shape2=null;
     for(int i=1;i<=50000;++i){
       shape1=ShapeFactory.getInstance("circle");
       shape1.draw(i+10,"red","dashed");
       }
      sysout("....................................");
     for(int i=1;i<=50000;++i){
       shape2=ShapeFactory.getInstance("square");
       shape2.draw(i+10,"red","solid");
       }
      sysout("....................................");
  }
}
```

Spring IOC container , ServletContaienr are designed based on fly weight DP.. becoz they not only create objs .. they also keep the objects in the internal cache for reusability...

Flyweight (recognizeable by creational methods returning a cached instance, a bit the "multiton" idea)

- java.lang.Integer#valueOf(int) (also on Boolean, Byte, Character, Short, Long and BigDecimal)

Proxy Design Pattern

==========

Proxy Internally holds real and provides access control to real .. In that process it can aslo define extra funcationalties...

Proxy internally uses real and working with proxy gives feel of working with real...(Proxy acts a real but it is not real)

Proxy object is called surrogate or wrapper obj hodling real object.. providing acess controller to read object..

3 types of Proxies

(a) Remote Proxies (Maintains access control to remote objects/services)

(b) Virtual Proxies (Maintains access control to create real objects)

(c) Protected Proxies (Maintains access controller to real objects by checking acess permissions of user)

During demonitization BankService is accesse through BankServiceProxy to allow to depoiste or withdraw money with restrictions..

(a) if we ask to draw to more than >4000 then gives only 4000

(b) if we ask to draw to more than <=4000 then allows request amount withdrawl

note:: These proxies can be generated manually or as inMemory proxies using jdk,cglib libraries....

The components Proxy Desing Pattern

==========

(a) Component Interface /Subject Interface (BankService) | (classes related

(b) Concrete Component class /Concrete Subject class (BankServiceImpl) | to real object)

(c) Proxy class (BankServiceProxy)

| -----> implements component interface

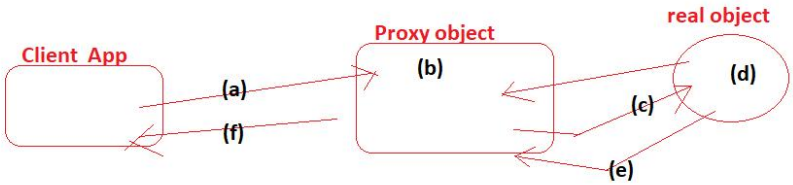
|-----> holds Concrete Component class object(real object)

(d) Factory ----> To give either real object or proxy object based on the condition.

note:: Decorator allows to multiple decoarations to real object with the support of composition here no access restriction to real object

Proxy provides access restriction real object .. In that process extra functionalities will also be provided.

While working with Adptter Client never feels he is dealing remote service.. becoz remote service and Adapter service ..(both are different) While working with the proxy we always feel working with real.



up

=>Spring AOP, Method Replacer, Lookmethod Injection and etc.. are developed based on Proxy DP..

Using CGLIB Libraries and JDK Libraries we can generat thse Proxy classes dynamically at runtime as InMemory classes .. (as the classes generated at run time allocating Memory in the JVM memory of RAM where java App runs)

Cglib library comes in the form of jar files .... (asm-7.1.jar (dependent), cglib-3.3.0.jar (main)) | internet..

=While Working with CgLib Libraries ,since the Proxy class as sub class fo Real class and overrides real class methods with new objects, So we can not real class final class and real class methods as final methods or static methods becoz we can not create sub class final classes and we can override final methods , abstract methods..

To overcome the above problem use .. JDK libraries to generate the Proxy class... becoz it can genereated proxy class by implementing component interface... So the real class can be taken as final class and real class methods can be taken final or static.. *(we need not to add jar files to classpath)*

Proxy (recognizeable by creational methods which returns an implementation of given abstract/interface type which in turn *delegates/uses* a *different* implementation of given abstract/interface type)

- `java.lang.reflect.Proxy`
- `java.rmi.*`
- `javax.ejb.EJB` *(annotation)*
- `javax.inject.Inject` *(annotations)*
- `javax.persistence.PersistenceContext` *(annotation)*

Strategy DesignPattern

=>It is all about developing classes of dependency management based bunch rules to make them as loosely coupled interchangable parts....
=> Taking classes having dependency is called dependency management....Where target class should be in a postion to change related depdenents having loose coupling....

Target class	Dependent class
Student	Java Course/DotnetCourse
Laptop	HardDisk
Flipkart	DTDC/FirstFlight
Vehicle	DieselEngine/Petrol Engine

The strategy pattern is used to create an interchangeable family of algorithms from which the required process is chosen at run-time

3 principles /rules of strategy dp

(a) Prefer/Favour Composition over Inheritance
(b) Always code to interfaces .... never code to Concreate classes/Implementation classes
(c) Our code must be open for extension and must be closed for modification.

(a) Prefer/Favour Composition over Inheritance

```
class A extends B{
  ....
  ....                    //inheritence  (IS-A relation)
}
class A{
  private B b=new B();     // composition (HAS-A relation)
  ...
  ....
}
```

=>Use Inheritence if class2 wants to use the entire behaviour of class1
=>Use Composition if class2 wants to use specific behaviour of class1

Limitations of Inheritence

(a) Some languages do not support multiple inheritence (can not use properties of multiple classes at a time)
(b) Code becomes easily brekable (fraile code)
(c) Testing problem
note: most of these problems can be solved using composition

#Rule1

(a) Some languages do not support multiple inheritence (can not use properties of multiple classes at a time)

java does not support multiple inheritence

```
public class  Flipkart extends  DTDC,BlueDart{
  .....
  .....
}
```

note:: Do not look interfaces in the angel inheritence/multiple inheritence ..Always look at them in the angel polymorphism and loose coupling

solution:

```
public class Flipkart{
  private DTDC  dtdc=new DTDC();
  private  BlueDart bDart=new BlueDart();
    ....
    ....
    ....
}
```

(b) Code becomes easily brekable (fraile code)

```
class  A{
        float
public  int  m1(){
  return 100;
 }
}

class  B extends  A{
 public  int m1(){
   return 1000;
  }
}

Class  C exntes  B{
  ...
  ...
  ...
}

class D exteds  C{
  .....
  ...
}

class  F extends  B{
  ...
  ...
}
```

If we change the return of m1() in "A" class form int to float .... It will distrub all classes that are there in the inhteritence hierarchy of class "A".

Solution:

```
class  A{
        float
public  int  m1(){
  return 100;
 }
}

class  B{
  private  A a=new A();

 public int m1(){
   int x=Math.round(a.m1());
   ...int na.m1();....
  return x+1000;
 }
}

class  C extends B{
  ....
  .....
}

class D exteds  C{
  .....
  ...
}
```

If we change m1() return type float from Int in "A" .... we just need modify one line in m1() method of "B" class.. and the classes inheriting from "B" class will not be effected.. So code is not that much fragile..

note:: Since java.lang.Object is the topmost class in the inheritence hierachy of every class... its method's signature will never be changed..

(c) Testing problem

=>Programmer's testing on his own piece of code is called unitTesting....

```
class A{
  public void m1(){
    ...
  }
  public void m2(){
    ..
  }
  public void m3(){
    ...
  }

class B extends A{

  public void m4(){
    ...
  }
  publid  void m5(){
    ...
  }
}

B b=new B();
b.m1();
b.m2();
b.m3();
b.m4();
b.m5();
```

HEre while testing "B" class... we should not only test "B" class methods... we should also test all inherited methods...from inheritence hierarchy.. (Here unit testing lengty and complex)

solution::

```
class A {
  public  int  m1(){
    return 1;
  }
  public void m2(){
    ..
  }
  public void m3(){
    ...
  }

class  B{
  private  A a=new A();
  public  int  m4(){
    int x=a.m1();
     return x+100;
  }
}
```

Here while doing unit testing on "B" class obj , we just need to worry about "B.methods" ....not at all other classes methods.... If "B" class are using any "A" class methods they will be participating in using Testing indirecly while doing unit testing on "B" class methods...

(b) Always code to interfaces ... never code to Concreate classes/Implementation classes

problem:

```
public class DTDC{
  ....
  ....
}
        //target class
public class Flipkart{
  private  DTDC dtdc=new DTDC();
  ...
}

public class BlueDart{
  ....
  ....
}
```

// To change DTDC to BlueDart and vice-versa..
In Flipkart class ... we need to modify source code..
(tight coupling:: The degree of dependency is more)

problem2:

```
                    //Dependent classes
interface Courier{
}

public class DTDC implements Courier{
}

public class BlueDart implements Courier{
}

                //target class
public class Flipkart{
  private Courier   courier=new DTDC();
  ...
  ...
}
```

note:: Still there is tight coupling..

Solution:

```
        //dependent classes (same as problem2)

//target class
public class Flipkart{
  private Courier courier;

  public void setCourier(Courier courier){
    this.courier=courier;
  }
  ...
}
```

Here the target and dependent classes are loosely coupled..becoz Interface model programming (POJI)

Client App

```
Flipkart fpkt=new Flipkart();
fpkt.setCourier(new DTDC());
fpkt.setCourier(new BlueDart());
```

If the degree dependency is less b/w target and dependent classes then they are called loosely coupled comps...

(c) Our code must be open for extension and must be closed for modification.

=>The Interface model programming that is suggested in rule no:2 allows to add more Dependents by implementing the common interface.. This tells our code is open for extension...
=> By taking target and dependent classes as final classes or by making their methods as final methods.. we can stop method overriding .. indirecly we can say our code is closed for modification.. (Take Flipkart , DTDC, BlueDart and etc.. classes as final classes)

While implementing Strategy DP we generally take on Factory pattern to abstract the following things
(a) To create and return one of the several related dependent classes objs based on the data that is supplied
(b) To create target class object and to assign Dependent class obj it..

```
Amazon
shopping(String[] items,
         float[] prices)

Payment Operations
  |---->DebitCardPayment
  |---->CreditCardPayment
  |---->UPIPayment
  |---->NetBankingPayment
  doPayment(id,name,amount)
```

AmazonFactory ----> should give Amazon object having One related Payment class object.

```
StrategyDP
  |------>com.nt.comps
          |---------->Amazon.java
          |---------->Payment.java
          |---------->DebitCardPayment.java
          |---------->CreditCardPayment.java
          |---------->UPIPayment.java
  |----> com.nt.factory
          |---------->AmazonFactory.java
  |---->com.nt.test
          |---------->StrategyDPTest.java
```

In spring all layered apps, all imp service,utility classes will be developed based on this strategy DP

note:: In Spring All major apps will be developed as Strategy DP Pattern.

Strategy (recognizeable by behavioral methods in an abstract/interface type which invokes a method in an implementation of a different abstract/interface type which has been passed-in as method argument into the strategy implementation)

=>java.util.Comparator#compare(), executed by among others Collections#sort().
=>javax.servlet.http.HttpServlet, the service() and all doXXX() methods take HttpServletRequest and HttpServletResponse and the implementor has to process them (and not to get hold of them as instance variables!).
=> javax.servlet.Filter#doFilter()

Prototype Design Pattern

==========

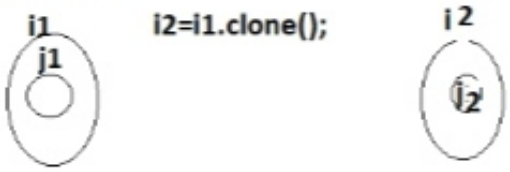
--->Instead of new objects from scratch .. just use cloning process

(a) Shallow cloning

clone() method by default performs shallow cloning..

Only objects are cloned but their internal objects/state are not cloned..

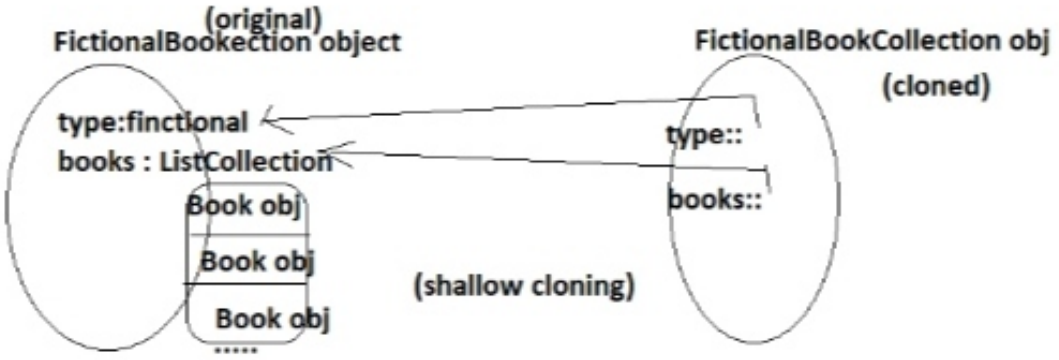
(b) Deep Cloning

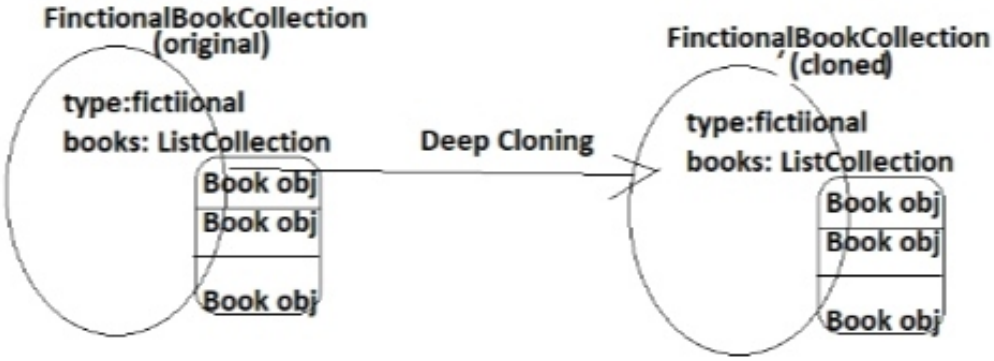


(deep cloning is good for prototype dp)

Here objects and their internal objects/state are cloned.

if we want to give buch of same objects for multiple times... first time u create them from scratch.. remaining all the times u get them from cloning process... So we can avoid CPu time wastage..





Prototype (recognizeable by creational methods returning a *different* instance of itself with the same properties)

- java.lang.Object#clone() (the class has to implement java.lang.Cloneable)