Java Design Patterns

INDEX

Java Design Pattern

Design Pattern

a. Design Patterns are set of rules that are given as best solutions for reoccurring problems of application development.
b. Design patterns best practices to develop software apps effectively.
c. Design patterns are 3-part rule having relation.
   A Context: An Environment/ surroundings/ situation that creates the problem.
   A Problem: Problem for which we need a solution like memory/ CPU/ performance issues and etc.
   A Solution: Solution code that solves the problem.

Anti-Pattern: The worst solution for the problem is called anti-pattern.

- Design pattern means the solution that is designed for reoccurring problem.

- Cristopher Alexander has given best solutions related to Civil Engineering industry.
- By inspiring from that GOF (4 Computer scientist) have given best solutions related to Object Oriented Programming.
- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides met together, and had a big conversion about the Design pattern. These gang of four people were called Gang of Four (GOF).

- Sun MS has given JEE patterns related to JEE Technologies based Layered application development.

Identifying and documenting Design Pattern

- Documented problem and its solution are initially called Candidate pattern.
- After checking in multiple projects and after comparing with existing patterns. We will make our candidate pattern as standard pattern.
- Pattern template document, details about all standard patterns must be maintained here.

Pattern Template/ Pattern Elements

a. Pattern Name: Having a concise, meaningful name for a pattern improves communication among developers.
b. Problem:
    - What is the problem and context where we would use this pattern?
    - What are the conditions that must be met before this pattern should be used?

c. Forces: List of reasons that makes the developer forces to use that pattern, justification for using the pattern.

d. Solution: Describes briefly what can be done to solve the problem.
    i. Structure: Diagrams describing the basic structure of the solution.
    ii. Strategies: Provides code snippets showing how to implement it.

e. Consequences: Describes result of using that pattern. Pros and cons of using it.
f. Related patterns: This section lists other related patterns and their brief description around it.

Some GOF patterns are: Singleton class, Factory, Bridge, Decorator, Strategy and etc.
Some JEE patterns are: DAO, FrontController, ApplicationController, Composite View and etc.

- GOF Patterns can be implemented in any OOP language including java.
- JEE patterns must be must implemented in Java, JEE environment.

Pattern Catalog

- If the subject expert or Research and Development (R&D) teams is giving more Design patterns belonging different categories, then it is recommended to categorized by preparing Pattern catalog.
- As of now GOF Pattern catalog is offering 3 categories of design patterns (total 23 patterns)
  a) Creational Patterns (solution related to objects creation)
  b) Structural Patterns (solutions related to complex objects creation)
  c) Behavioral Patterns (solutions related to interaction b/w classes and objects)
- JEE module and its technologies are given for developing Layered application.
- Layered means keeping different variety of logics in different components and making them interaction with other.
- A Layer is a logical partition/ physical partition that represents specific logic taking the support one more component (MVC is best architecture to develop layered application).

Creational Patterns: How an object can be created i.e., creational design patterns are the design patterns that deal with object creation mechanisms, trying to create Objects in a manner suitable to the situation. The basic form of Object creation (using new keyword) could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation. All the creational patterns define the best possible way in which an object can be, instantiated. These describes the best way to CREATE object instances. According to GOF, creational patterns can be categorized into five types.

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

   and etc.

Structural Patterns: Structural design patterns are design patterns that ease the design by identifying a simple way to realize relationships between entities. Structural Patterns describe how objects and classes can be combined to form larger structures. According to GOF, Structured Pattern can be realized in the following patterns:

1. Adapter Pattern

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

and etc.

Behavioral Patterns: Behavioral design patterns are design patterns that identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication i.e., prescribes the way objects interact with each other. They help make complex behavior manageable by specifying the responsibilities of objects and the ways they communicate with each other. The 11 behavioral patterns are:

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

Q. What is the difference b/w Architecture and Design Pattern?

Ans.

- Architecture speaks about involving multiple components and making them participating in communication to develop the application.
  E.g., MVC architecture
- Design Pattern speaks about problems in each component development and applying solutions for solving those problems.
  E.g., In each layer development of MVC architecture we use multiple design patterns like Singleton java class, View Helper Front controller and etc.

Advantages of Layered Applications:

a. Clean separation b/w logics because they will be more layers and components to maintain the logics.

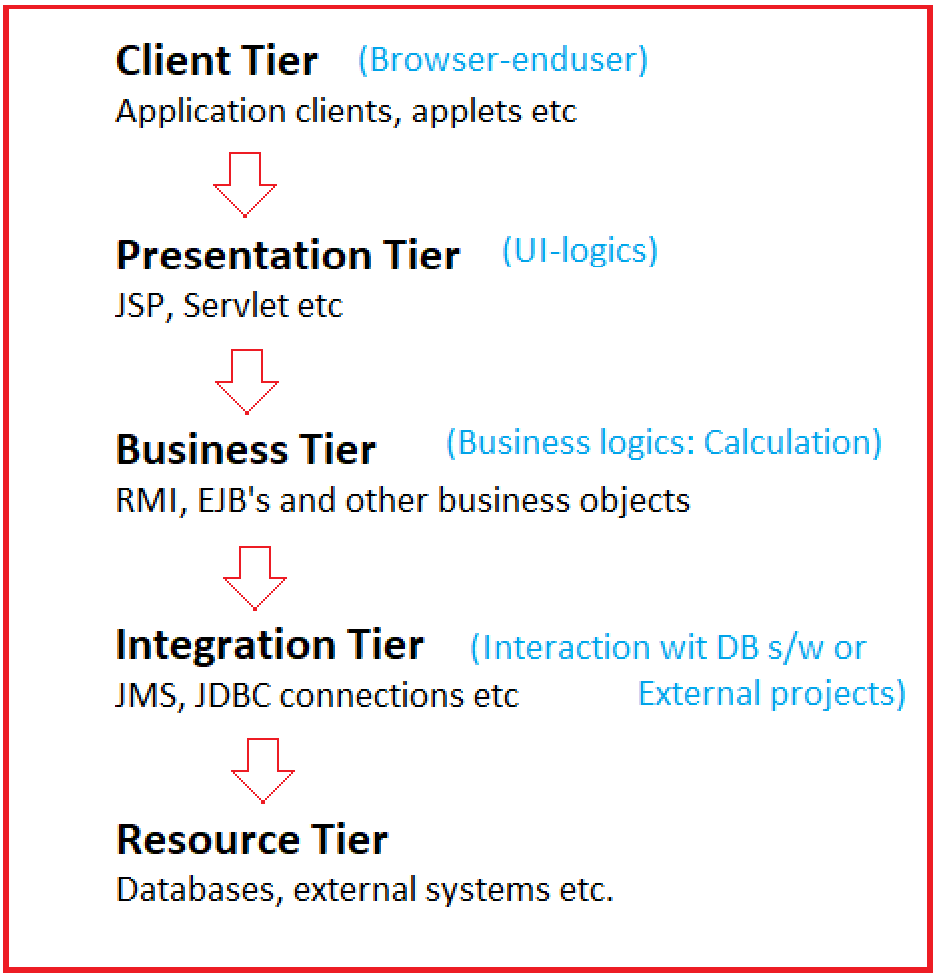
b. The modifications done in one-layer logics does not affect other layer logics.
c. Maintenance and Enhancement of project becomes easy.
d. Productivity will be good because parallel development.
e. It is industry standard approach to develop projects.

J2EE/ JEE Patterns (Sun MS JEE Patterns)

- J2EE Platform/ applications are multitiered system; we view the system in terms of tiers. A tier is a logical partition of related concerns. Tiers/ layers are used for separation of concerns. Each Tier handles its unique responsibility in the system. Each tier is a logical separation and has minimal dependency with other tiers. So, if we look into a J2EE Application it can be viewed as five several tiers as follows.

Five Tier Model of logical separation of concerns into tiers:

In J2EE the patterns are divided according to the functionality and the J2EE pattern catalog contains the following patterns.


Client Tier
(Browser-enduser)
Application clients, applets etc
Presentation Tier
(UI-logics)
JSP, Servlet etc
Business Tier
(Business logics: Calculation)
RMI, EJB's and other business objects
Integration Tier
(Interaction wit DB s/w or External projects)
JMS, JDBC connections etc
Resource Tier
Databases, external systems etc.

- In JEE Pattern Catalog, we have 3 categories of Design Patterns (21 patterns)
    a. Presentation-tier Patterns
    b. Business-tier Patterns
    c. Integration tier Patterns

Presentation-tier Patterns: This tier contains all the presentation tier logic required to service the clients that access the system. The presentation tier design patterns are

1. Intercepting Filter
2. Front Controller
3. Context Object
4. Application Controller
5. View Helper
6. Composite View
7. Service to worker
8. Dispatcher View

Business-tier Patterns: This tier provides business service required by the application clients; The Business tier design patterns are

1. Business Delegate
2. Service Locator
3. Session Façade
4. Application Service
5. Business Object
6. Composite Entity
7. Transfer Object
8. Transfer Object Assembler
9. Value List Handler

Integration Tier Patterns: This tier is responsible for communicating with external resources and systems such as data stores etc. The integration tier design patterns are

1. Data Access Object
2. Service Activator
3. Domain Store
4. Web Service Broker

We will deal with Best of JEE Patterns and GOF Patterns, they are

GOF Patterns

a. Singleton
b. Factory
c. Factory method
d. Abstract Factory
e. Bridge
f. Proxy
g. Decorator/Wrapper
h. Adapter
i. Strategy
j. Template
k. Flyweight
l. Prototype
m. Builder

JEE Patterns

n. FrontController
o. ApplicationController
p. View Helper
q. Composite View
r. ServiceLocator
s. Business Delegate
t. Service Facade
u. Message Facade
v. DAO (Data Access Object)
w. BO (Business Object)
x. DTO (Data Transfer Object)
y. VO (Value Object)

Singleton Design Pattern

- "Single" means 1 "ton" means object.
- The class which allows us to create e only one object in any situation is called "Singleton class".
- In Java, it is called as "Singleton Java class".

Problem:

- When class is having no state or read only state or sharable state then allowing to create multiple objects for that class will waste of the memory and will waste of CPU time because object creation in java is a costly process.

Solution:

- To overcome the above problem, make Java class as singleton java class, that must allow to create only one object in any situation.

Example:

- To make all employee of a company using same Printer. It is better to develop "Printer" java class as singleton java class.
- If someone creating only object for java class, even though that class allows to create more objects than that class is not called as singleton java class.
- Servlet component class is not singleton java class because Servlet container is happy with object but Servlet component class allows to create multiple objects if needed.
- The singleton java class must allow all the client apps/ underlying containers/ servers/ frameworks to create only one object in any situation even though they are attempting to create more objects for that class.

Note: If our servlet component class implements SingleThreadModel (I) then Servlet container may create Servlet instance pool (pool of Servlet class objects).

Some pre-defined Singleton java classes:

a. java.lang.Runtime (To make all people/ clients getting same runtime environment of current Java app)
b. java.awt.Desktop (To make all people/ clients getting same desktop of current computer)
c. org.apache.log4j.Logger (To make all the classes of a current java app using Logger environment to write log messages [confirmation/ debugging messages])

- state = data in variables
- behaviour = logics in methods
- type = class/ interface/ enum/ data type/ annotation

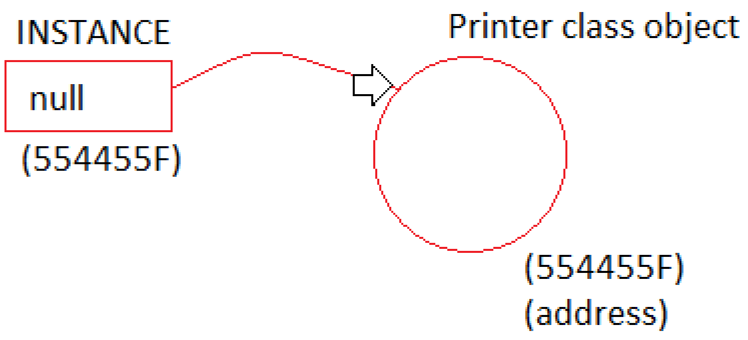
Procedure to Creating Singleton Java class

- To develop perfect singleton java class, we need to close all the doors that are there to create object for that java class and open only one door to check and create that single object (all these restrictions are outside of the class).

Singleton Java class with Minimum Standards:

```
public class Printer {
        private static Printer INSTANCE;
        private Printer () {
        }

        // public static factory method
        public static Printer getInstance () {
                //singleton logic
                if (INSTANCE==null)
                        INSTNACE=new Printer ();
                return INSTANCE;
        }

        //business method
        public void print (String msg) {
                System.out.println (msg);
        }
}
```

Factory Method: Any method that creates and returns object is called Factory method.



From Client app/ User class

```
//Printer p1 = new Printer (); //invalid
Printer p1 = Printer.getInstance(); //valid
Printer p2 = Printer.getInstance(); //valid
```

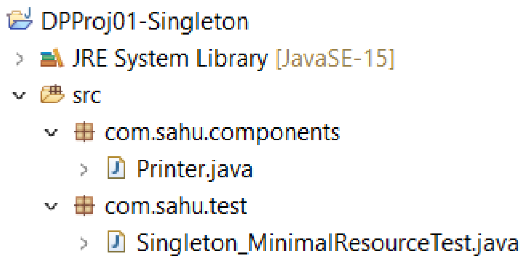
To develop Singleton java class with minimum standards:

a. Take private constructor inside the java class (To stop new operator-based object creation outside of the class).
b. Take private static member variable to hold current class object reference (This useful to check whether that single object is created or not).

c. Take public static factory method having singleton logic i.e., create and return object by checking whether object is already created or not.

- Recommended to take singleton java class as public to make that visible in more places and packages.
- private data, private constructors, private methods are visible within in the class but not outside the class.
- Only static member variables visible in static methods of the class without object.
- Generally, the one door we open for creating or accessing the object singleton java class is static factory method.

Directory Structure of DPProj01-Singleton:


```
DPProj01-Singleton
> JRE System Library [JavaSE-15]
v src
  v com.sahu.components
    > Printer.java
  v com.sahu.test
    > Singleton_MinimalResourceTest.java
```

- Develop the above directory structure package, class in src folder.
- Then use the following code with in their respective file.

Printer.java

```
package com.sahu.components;

public class Printer {

	private static volatile Printer INSTANCE;

	private Printer() {
		System.out.println("Printer.Printer()");
	}

	public static Printer getInstance() {
		if(INSTANCE==null)
			INSTANCE = new Printer();
		return INSTANCE;
	}

	//Business method
	public void print(String msg) {
		System.out.println(msg);
	}

}
```

Singleton_MinimalResourceTest.java

```
package com.sahu.test;

import com.sahu.components.Printer;

public class Singleton_MinimalResourceTest {

	public static void main(String[] args) {
		Printer print1 = Printer.getInstance();
		Printer print2 = Printer.getInstance();
		System.out.println(print1==print2);
		System.out.println(print1.hashCode()+" "+print2.hashCode());
	}

}
```

- Do not completely relay on constructor execution to confirm whether object is created or not because there are 1 or two exceptional cases.

  Case 1: If you create object for the sub class of abstract class, then abstract class constructor executes but it does not mean object is created for abstract class.
  Case 2: If you create object by using cloning process or deserialization process then no constructor will execute but it does not mean object is not created there.

Singleton Java class with eager instantiation:
Printer.java

```
package com.sahu.components;

public class Printer {

	private static Printer INSTANCE = new Printer ();

	private Printer () {
		System.out.println("Printer.Printer()");
```

```
        }

        public static Printer getInstance() {
                return INSTANCE;
        }
}
```

Singleton_EagerInstatiateTest.java

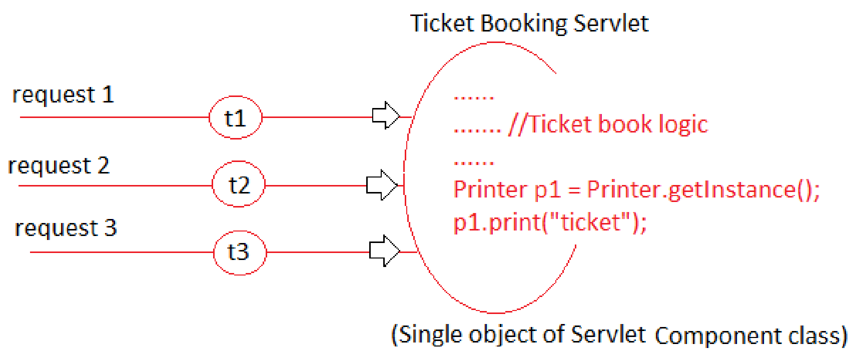
```
package com.sahu.test;

public class Singleton_EagerInstatiateTest {

        public static void main(String[] args) {
                try {
                        Class<?> clzz1 =
Class.forName("com.sahu.components.Printer");
                        Class<?> clzz2 =
Class.forName("com.sahu.components.Printer");

                        //Printer writer
                        System.out.println(clzz1==clzz2);
                        System.out.println(clzz2.hashCode()+"
"+clzz2.hashCode());
                } catch (ClassNotFoundException e) {
                        e.printStackTrace();
                }
        }

}
```

- Creating singleton java with eager Instantiation (creating object during class loading even though no one has asked for object creation) is bad practice. It will waste the memory and CPU time if that early created object for singleton is not used ever.

Singleton java class in multi-thread environment

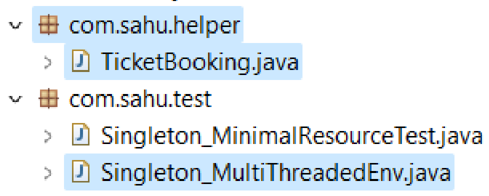
Problem:

- If multiple threads are acting one object simultaneously or concurrently then the data of the object may disturb that makes object as non-thread safe object.


Ticket Booking Servlet
request 1
t1
request 2
t2
request 3
t3
......
...... //Ticket book logic
......
Printer p1 = Printer.getInstance();
p1.print("ticket");
(Single object of Servlet Component class)

- Every Servlet component is Single instance (object) and multi-thread component.
- In multithread environment if thread switching is happening at if (INSTANCE==null) statement then there is a possibility creating multiple objects for singleton java class.

To implement this Problem:


```
com.sahu.helper
    TicketBooking.java
com.sahu.test
    Singleton_MinimalResourceTest.java
    Singleton_MultiThreadedEnv.java
```

- Take the lazy instantiation singleton class code.
- Create a package com.sahu.helper havingTicketBooking.java class and in com.sahu.test create a Singleton_MultiThreadedEnv.java test class.

TicketBooking.java

```java
package com.sahu.helper;

import com.sahu.components.Printer;

public class TicketBooking implements Runnable {

        @Override
        public void run() {
                Printer printer = Printer.getInstance();
                System.out.println(printer.hashCode());
        }

}
```

Singleton_MultiThreadedEnv.java

```
package com.sahu.test;

import com.sahu.helper.TicketBooking;

public class Singleton_MultiThreadedEnv {

        public static void main(String[] args) {
                TicketBooking ticket = new TicketBooking();
                Thread thread1 = new Thread(ticket);
                Thread thread2 = new Thread(ticket);
                Thread thread3 = new Thread(ticket);

                long start = System.currentTimeMillis();
                thread1.start();
                thread2.start();
                thread3.start();
                long end = System.currentTimeMillis();
                System.out.println("Time : "+(end-start)+" ms");
        }

}
```

Solution:

There 3 solutions to solve this problem

a. By enable eager instantiation on singleton class (Bad)
b. By taking static factory method as synchronized method (little bad)
c. By taking synchronized blocks inside static factory method (good solution)

Printer.java (Solution 2)

```
        public static synchronized Printer getInstance() {
                if(INSTANCE==null)
                        INSTANCE = new Printer();

                return INSTANCE;
        }
```

- Instead of making whole static factory method as synchronized method it is recommended to use synchronized block inside static factory method for better performance and for locking only the required code.

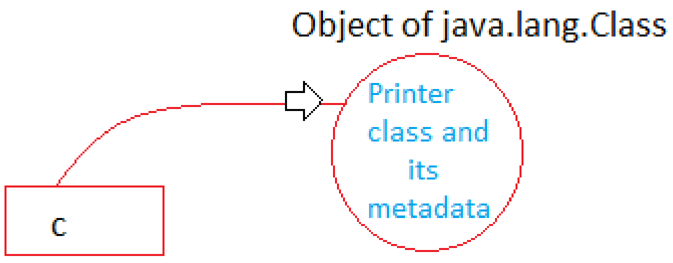
Printer.java (Solution 3)

```
public static Printer getInstance() {
        synchronized(Printer.class) {
                if(INSTANCE==null)
                        INSTANCE = new Printer();
        }

        return INSTANCE;
}
```

- In static method “this” is not visible so we can’t use in synchronized block, so we have to passes class name in the form of java.lang.Class object.

Class c= Printer.class;

"c" is ref variable pointing to the object of java.lang.Class having Printer class name and its Meta-Data as the data of object.



- In a running java app, if want to hold class or interface or enum or annotation (not as string value) with real meaning then we need to use the object of java.lang.Class. The easiest way to create that object is using ".class" property.

```
Class c1 = System.clas;
Class c2 = Date.class;
Class c3 = Printer.class;
```

Note:

- ✓ length, class are the built-in properties in java class generated by java compiler dynamically.
- ✓ "length" is useful to get array length.
- ✓ "class" is useful to get the object of java.lang.Class.
- ✓ Java is giving two-built-in reference variables this, super.
- ✓ Java is giving two built-in threads “main” thread and “gc” thread.
- ✓ In java app to hold numeric values we can use numeric data type

variables, to text content String class objects, floating points we can use float data type variable, similarly to hold class, interface, enum and annotation then we need to use the object of java.lang.Class.

- Instead of making every thread entering into synchronized block for creating object check object is already created or not outside the synchronized block for better performance. In that process we will get two NULL checks.

Printer.java

```java
public static Printer getInstance() {
        if (INSTANCE == null) {
                synchronized (Printer.class) {
                        if (INSTANCE == null)
                                INSTANCE = new Printer();
                }
        }
        return INSTANCE;
}
```

```java
synchronized (this) {
        ……
}
```

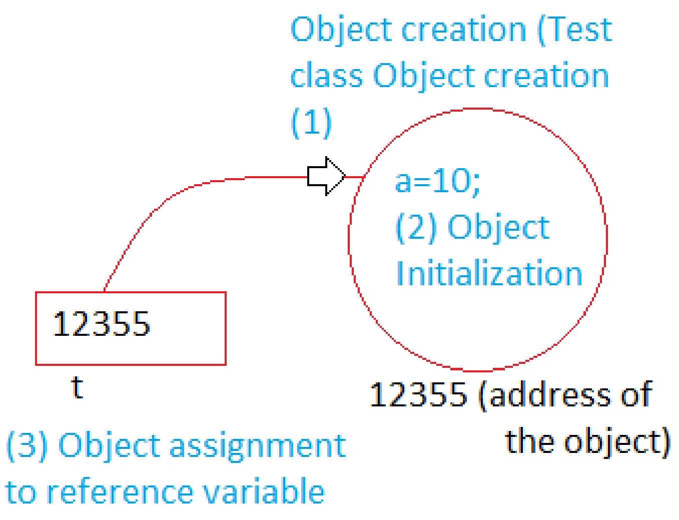
Object locking i.e., in the synchronized block only one thread can act on current object (this), here instance variables are locked.

```java
synchronized (Printer.class) {
        …………
}
```

Class locking i.e., in the synchronized block only one thread can act on static member variables of class (static variables are locked).

Object creation and initialization process is having 3 parts:

- It is recommended to take static member variable of singleton java class as volatile variable in order to make following activities as single task activities by applying do everything or nothing principle or we can say atomicity.
    a. Object creation
    b. Object initialization
    c. Object assignment


Object creation (Test class Object creation (1)
a=10;
(2) Object Initialization
12355
t
12355 (address of the object)
(3) Object assignment to reference variable

```
Test t = new Test();

class Test {
    int a;
    public Test() {
        a=10;
    }
    ...........
}
```

```
public class Printer {

    private static volatile Printer INSTANCE;

    ....................
    ....................
}
```

Q. What are the best solution to solve multi-threading issues of Singleton Java class?

Ans.

Solution 1:

- Using synchronized (-) inside the static factory method having double null check and enabling volatile behaviour on object creation process.

Printer.java

```
public class Printer {

    private static volatile Printer INSTANCE;

    private Printer() {
        System.out.println("Printer.Printer()");
    }

    public static Printer getInstance() {
        if (INSTANCE == null) {
            synchronized (Printer.class) {
                if (INSTANCE == null)
```

```
                                        INSTANCE = new Printer();
                        }
                }

                return INSTANCE;
        }

}
```

Solution 2: (Best Solution)

- Eager instantiation on singleton java class using its nested inner class having volatile behaviour (overall singleton java class is instantiated lazily only when static factory method is called by client)

Printer.java

```
public class Printer {

        private Printer() {
                System.out.println("Printer.Printer()");
        }

        class InnerPrinter {
                private volatile static Printer INSTANCE = new Printer();
        }

        //Factory method to get class object
        public static Printer getInstance() {
                return InnerPrinter.INSTANCE;
        }

}
```

Cloning

- It is the process of creating new object/ duplicate object based on existing object having existing state as new/ duplicate object state.
- Cloning possible only on cloneable objects. Object becomes cloneable object only when class of the object implements java.lang.Cloneable (I).
- Cloneable is a marker/ tag interface.
- The interface that makes underlying JVM/ server/ container/ framework

to provide special runtime capabilities to the implementation class objects is called Marker/ Tag interface.

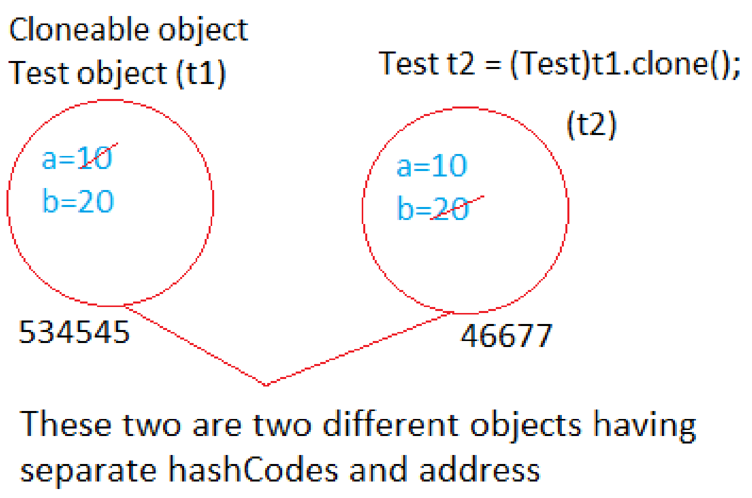
- Do not decide whether interface is marker interface or not based on its emptiness. Decide it based on the runtime capabilities that are coming to implementation class object from underlying JVM/ server/ container/ frameworks.

E.g.,		
	java.io.Serializable (I) java. lang. Cloneable (I)	Empty interfaces acting marker interface.
	java.rmi.Remote (I) java.lang.Runnable (I)	Interface with method acting as Marker interface

- Most of the Marker interfaces are empty interface. But we can't say every empty interface is marker interface unless until it makes underlying JVM/ container/ server/ framework providing special runtime capabilities to implementation class objects.
- To create user-defined marker interface, we must user-defined container on top of JVM providing special runtime capabilities to marker interface implement class objects.
- To do cloning on any cloneable object we need to call clone () method on that object, which basically protected method in java.lang.Object class.

Q. Why we should bother Cloning process on Singleton java class object if do not make our singleton java class implementing java.lang.Cloneable (I)?

Ans. If our singleton java class is extending from a pre-defined/ user-defined class and that class is implementing java.lang.Cloneable (I) then we can say our singleton java class also implementing Cloneable (I) and this makes the object of singleton java class as the cloneable object.

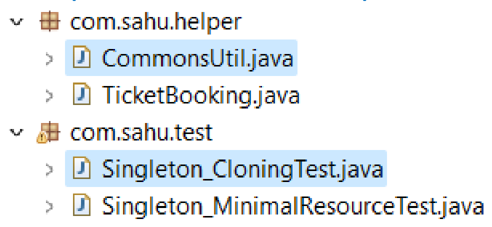


Cloneable object
Test object (t1)
Test t2 = (Test)t1.clone();
(t2)
a=~~10~~
b=20
a=10
b=~~20~~
534545
46677
These two are two different objects having separate hashCodes and address

Signature of clone() in java.lang.Object class

protected Object clone () throws CloneNotSupportException

To implement this clone problem:


```
com.sahu.helper
    CommonsUtil.java
    TicketBooking.java
com.sahu.test
    Singleton_CloningTest.java
    Singleton_MinimalResourceTest.java
```

- Create a class CommonsUtil.java in com.sahu.helper and another class Singleton_CloningTest.java in com.sahu.test.
- Extend Printer.java class from CommonsUtil.java.
- Then place the following code in their respective file.

CommonsUtil.java

```
package com.sahu.helper;

public class CommonsUtil implements Cloneable {

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

}
```

Singleton_ClonigTest.java

```
package com.sahu.test;

import com.sahu.components.Printer;

public class Singleton_CloningTest {

        public static void main(String[] args) {
                Printer printer1 = Printer.getInstance();
                try {
                        Printer printer2 = (Printer) printer1.clone();

                        System.out.println("Clone is created");
                        System.out.println(printer1.hashCode()+"
"+printer2.hashCode());
                        System.out.println("printer1==printer2 :
"+(printer1==printer2));
                }
                catch (CloneNotSupportedException e) {
                        e.printStackTrace();
                }
        }

}
```

Printer.java

```
package com.sahu.components;

public class Printer extends CommonsUtil {
```

Q. Why the constructor will not be executed when the object created through cloning process?

Ans. The object created through cloning process gets existing invoking object state/ data as its initial state i.e., new object created through cloning is already initialized with existing invoking object state so there is no need to separate initialization through constructor. So, the constructor will not be executed.

Q. When clone () is there in Object class why should we override that method having super.clone() logic?

Ans. clone () method of java.lang.Object class is protected method and it is visible only in the immediate sub class of inheritance hierarchy. So to use in other classes it is recommended to override having public modifier with super.clone() logic.

$$\begin{tikzcd}
\text{java.lang.Object (c)} \\
\text{com.sahu.helper.CommonsUtil (c)} \arrow[u, "\text{extends}"'] \\
\text{com.sahu.components.Printer (c)} \arrow[u, "\text{extends}"']
\end{tikzcd}$$

Q. How to stop cloning process of on the object of Singleton java class?
Ans. Override clone () method in singleton java class either throwing CloneNotSupportedException (good, more industry standard) or returning already created object reference.

Printer.java

```
public class Printer extends CommonsUtil {

	@Override
	public Object clone() throws CloneNotSupportedException {
		throw new CloneNotSupportedException("cloning is not allowed in Singleton class");
		//return InnerPrinter.INSTANCE;
	}

}
```

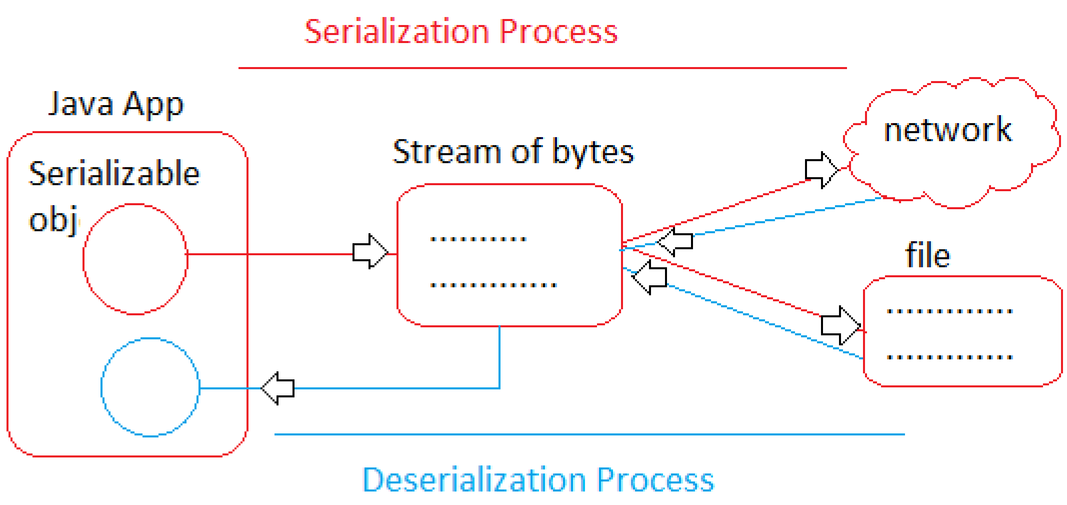
To develop perfect Singleton java class:

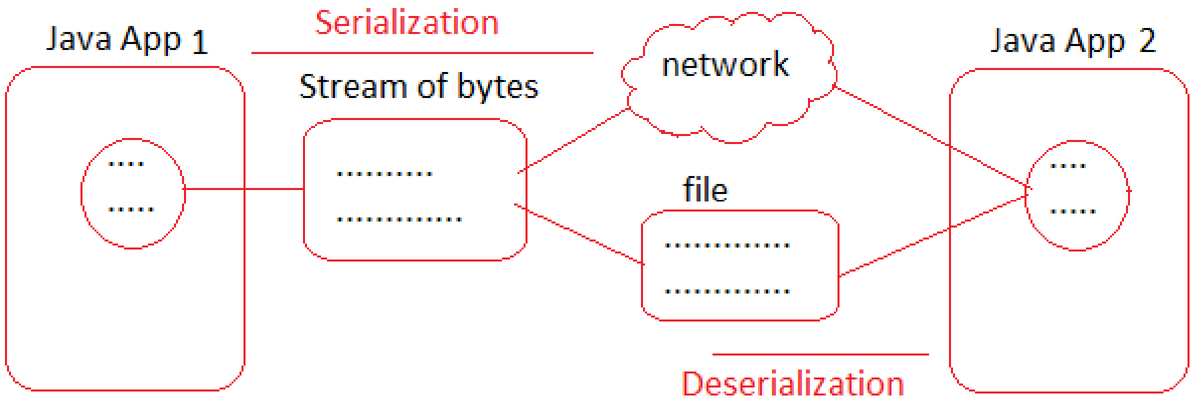
a. Protection from new operator (using private constructor).
b. Protection from Multi-Threaded environment (synchronization with double null check or inner class based eager instantiation).
c. Protection from Cloning process (override clone () method throwing CloneNotSupportedException).
d. Protection from Deserialization.
e. Protection from Reflection API environment.
f. Protection from ClassLoader environment.

Serialization & Deserialization

- The process of converting object data/ state to stream of bytes is called Serialization. These stream of bytes we can write over the network or to the file or somewhere.
- Serialization is possible only on Serializable objects. The object becomes Serializable object only when its class implements java.io.Serializable (I) (Marker Interface).
- Collecting stream of bytes from different place like network/ file and recreating object having state gathered from different locations is called Deserialization.

Note: Java notation object data/ state can't be written to files or can't be sent over the network directly. For that first we need to convert stream of bytes (0, 1, 0, 1 kids of data).





Object state ----> Stream of bytes (Serialization)
Stream of bytes ---> Object state (Deserialization)

Note: If the object is created/ recreated using Deserialization the constructor will be not be executed because the data read from network/ file will be utilized as the initial state of object by default. So separate initialization through constructor is not required.

- ✓ In serialization java notation object data will be converted into stream bytes.
- ✓ In deserialization the stream of bytes data will be converted into java notation data.
- ✓ To write Serializable object state to file through serialization we need to use java.io.ObjectOutputStream class. Similarly, to recreate/ create object having data collected from file through Deserialization process we need to use java.io.ObjectInputStream class.

Problem:

- We can break single java class behaviour by creating new objects through Deserialization process by performing Serialization on the single object of the singleton java class.

- Low level streams can read/ write only low-level data (chars/ bytes) from/ to Destination (files)
  E.g.,

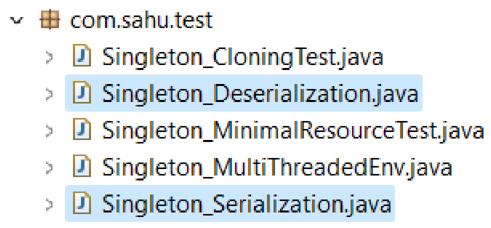
FileInputStream FileOutputSteam	(Byte streams)
FileWriter FileReader and etc.	(Character streams)

- High level streams can read/ write both low- and high-level data like floats, integers, Strings, objects and etc. from/ to destination (like files)
  E.g.,

DataInputStream DataOutputStream ObjectInputSteam ObjectOutputStream	(Byte streams)
ByteArrayFileWriter ByteArrayFileReader	(Character streams)

Note: High level stream internally uses low level streams.

To implement this Deserialization problem:



```
com.sahu.test
  Singleton_CloningTest.java
  Singleton_Deserialization.java
  Singleton_MinimalResourceTest.java
  Singleton_MultiThreadedEnv.java
  Singleton_Serialization.java
```

- Create Singleton_Serialization.java and Singleton_Deserialization.java under com.sahu.test package
- Implements Serializable interface in CommonsUtil.java class
- Then place the following code in their respective file.

CommonsUtil.java

```
public class CommonsUtil implements Cloneable, Serializable {
```

Singleton_Serialization.java

```
package com.sahu.test;

import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

import com.sahu.components.Printer;

public class Singleton_Serialization {

	public static void main(String[] args) {
		//Get Singleton class object
		Printer printer1 = Printer.getInstance();
		Printer printer2 = Printer.getInstance();
		try {
			//Write object data to file using ObjectOutStream
support (Serialization process)
			ObjectOutputStream oos = new
ObjectOutputStream(new FileOutputStream("printer_data.ser"));
			oos.writeObject(printer);
			oos.flush();
			oos.close();
			System.out.println("Serialization is done");
		} catch (IOException e) {
			e.printStackTrace();
		}
	}

}
```

Singleton_Deserialization.java

```
package com.sahu.test;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

import com.sahu.components.Printer;

public class Singleton_DeSerialization {
        public static void main(String[] args) {
                try {
                        //Write object data to file using ObjectInputStream
support (Serialization process)
                        ObjectInputStream ois1 = new ObjectInputStream(new
FileInputStream("printer_data.ser"));
                        Printer printer1 = (Printer) ois1.readObject();
                        printer1.print("Good morning");
                        System.out.println("Printer object hashCode :
"+printer1.hashCode());
                        ois1.close();
                        System.out.println("DeSerialization is done");
                        System.out.println("------------------------------");
                        ObjectInputStream ois2 = new ObjectInputStream(new
FileInputStream("printer_data.ser"));
                        Printer4 printer2 = (Printer4) ois2.readObject();
                        printer2.print("Nimu");
                        System.out.println("Printer object hashCode :
"+printer2.hashCode());
                        ois2.close();
                        System.out.println("DeSerialization is done");
                } catch (IOException e) {
                        e.printStackTrace();
                } catch (ClassNotFoundException e) {
                        e.printStackTrace();
                }
        }

}
```

Solution:

- Place readResolve() method in singleton Java class as shown below to stop Deserialization on the singleton java class.
- readResolve() does not belong anywhere like main(-) method called JVM. ois.readObject() method internally calls readResolve() method before performing actual Deserialization. So, we are using that method in Singleton java class either to return existing object or to throw exception.

Printer.java

```
public class Printer extends CommonsUtil2 {

	private static final long serialVersionUID = 64646364920L;

	//Solution for DeSerialization problem
	public Object readResolve() {
		//throw new IllegalArgumentException("Deserialization is not allowed in Singleton class");
		return InnerPrinter.INSTANCE;
	}

}
```

- If we don't place serialVersionUID in any class the JVM generates it dynamically and that will be changed accordingly we change the structure of class. So, we must maintain same structure of class for both serialization and deserialization process.
- If want to get freedom to modify class structure like adding new variables, new methods after Serialization and before Deserialization then it is recommended to place serialVersionUID manually in the java class before performing serialization itself. So, the changing in class structure does not stop Deserialization because JVM finds same serialVersionUID in the class even though structure in the class is modified.

```
		private static final long serialVersionUID = 64646364920L;
```

Reflection API

- Reflection means mirror image.
- Reflection API app/ code acts as mirror device to get complete internal details of given class/ interface/ enum/ annotation.
- We can say using reflection API, we can get Meta Data of given input (class/ interface/ enum/ annotation). Meta Data is data about data.
- In reflection API applications not only, we can get internal details of given class/ interface/ enum/ annotation, we can use them for dynamical activities like for instantiation, for method calling, for setting/ getting filed property values, for changing modifies and etc.
- Using Reflection API, we can get access to private data/ methods/ constructor of class/ object being from outside of the class/ object. This

is security breakage in java. So, we cannot say java is not 100% pure object oriented. For this we need to use setAccessible(true).

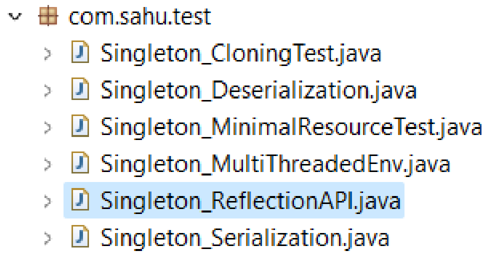
- API: Application Programming Interface.
- In Java API comes in the form of packages having classes, interfaces, enum and annotations
- APIs are the base for programmers to develop software applications E.g., lang API (java.lang pkg), reflection API (java.lang.reflect pkg), utility api (java.util pkg) and etc.

java.lang.reflect pkg:

- Each object Field class represents one member variable of given class.
- Each object of Constructor represents one constructor of given class.
- Each object of Method represents one method of given class.

  and etc.

To implement Reflection API code:



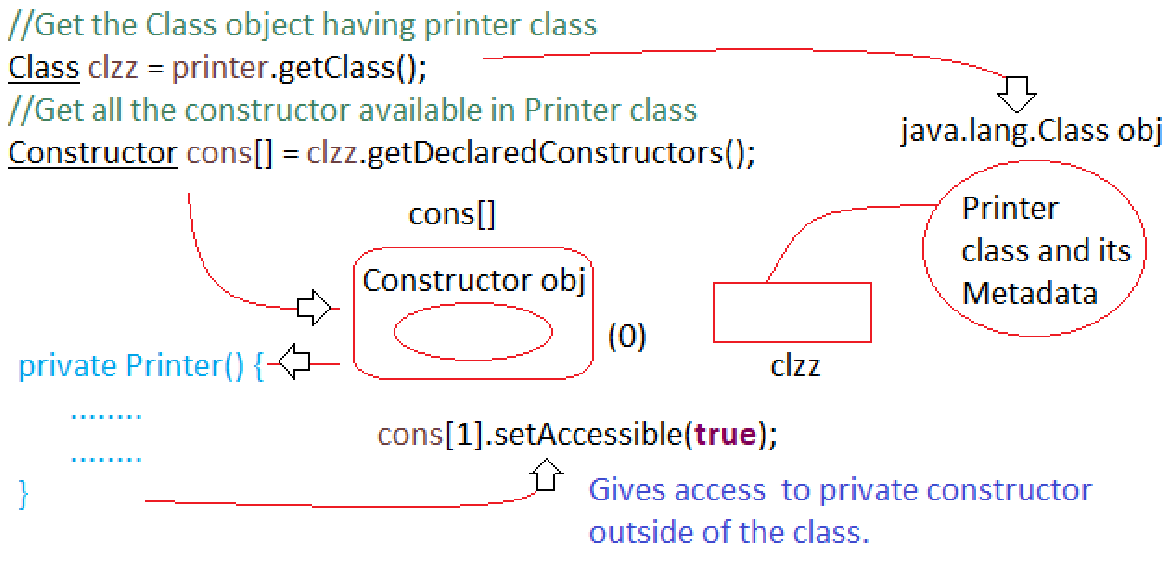
```
com.sahu.test
  Singleton_CloningTest.java
  Singleton_Deserialization.java
  Singleton_MinimalResourceTest.java
  Singleton_MultiThreadedEnv.java
  Singleton_ReflectionAPI.java
  Singleton_Serialization.java
```

- Create a class Singleton_ReflectionAPI.java under com.sahu.test package.
- Then place the following code in their respective file.

Singleton_ReflectionAPI.java

```
package com.sahu.test;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;

import com.sahu.components.Printer;

public class Singleton_ReflectionAPI {

	public static void main(String[] args) {
		//Get the printer class object using static factory method
		Printer printer = Printer.getInstance();
```

```
                    try {
                              //Get the Class objet having printer class
                              Class clzz = printer.getClass();
                              //Get all the constructor available in Printer class
                              Constructor cons[] = clzz.getDeclaredConstructors();
                              cons[1].setAccessible(true);
                              //Create Object using Reflection API
                              Printer printer1 = (Printer) cons[1].newInstance();
                              Printer printer2 = (Printer) cons[1].newInstance();
                              System.out.println("Objects are created using reflection
API");
                              System.out.println(printer.hashCode()+"
"+printer1.hashCode()+" "+printer2.hashCode());
                    }
                    catch (InstantiationException e) {
                              e.printStackTrace();
                    } catch (IllegalAccessException e) {
                              e.printStackTrace();
                    } catch (IllegalArgumentException e) {
                              e.printStackTrace();
                    } catch (InvocationTargetException e) {
                              e.printStackTrace();
                    }
          }

}
```



```
//Get the Class object having printer class
Class clzz = printer.getClass();
//Get all the constructor available in Printer class
Constructor cons[] = clzz.getDeclaredConstructors();

private Printer() {
      ........
      ........
}

cons[1].setAccessible(true);
```

java.lang.Class obj
Printer class and its Metadata
cons[]
Constructor obj
(0)
clzz
Gives access to private constructor outside of the class.

Solution:

- Write extra code in the singleton java class private constructor that is throw exception if object is already created.

Printer.java

```
public class Printer extends CommonsUtil {

	//Private constructor for avoid object creation using new keyword
	private Printer() {
		if (InnerPrinter.INSTANCE!=null)
			throw new RuntimeException("Object is already 
created!!!");
		System.out.println("Printer.Printer()");
	}

}
```

Printer.java

```
public class Printer extends CommonsUtil {

	private Printer() {
		if (INSTANCE!=null)
			throw new RuntimeException("Object is already 
created!!!");
		System.out.println("Printer.Printer()");
	}

}
```

Different Perfect Singleton Java classes

Version 1 (Legacy Approach): Static factory method with synchronized blocks having lazy instantiation.

Printer1.java

```
package com.sahu.components;

import com.sahu.helper.CommonsUtil;

public class Printer1 extends CommonsUtil {
    //Solution for InvalidClassException
    private static final long serialVersionUID = 2105646988646364920L;

    private static volatile Printer1 INSTANCE;

    //Private constructor for avoid object creation using new keyword
    private Printer1() {
        if (INSTANCE!=null)
            throw new RuntimeException("Object is already
created!!!");
        System.out.println("Printer1.Printer()");
    }

    //Factory method to get class object
    public static Printer1 getInstance() {
        //Solution for Multi-Threaded problem
        if (INSTANCE == null) {
            synchronized (Printer1.class) {
                if (INSTANCE == null)
                    INSTANCE = new Printer1();
            }
        }
        return INSTANCE;
    }

    // Solution for cloning problem
    @Override
    public Object clone() throws CloneNotSupportedException {
        //throw new CloneNotSupportedException("cloning is not
allowed in Singleton class");
        return INSTANCE;
    }

    //Solution for DeSerialization problem
    public Object readResolve() {
        // throw new IllegalArgumentException("Deserialization is not
allowed in Singleton class");
        return INSTANCE;
    }
```

```
        // Business logic/ method
        public void print(String msg) {
                System.out.println(msg);
        }

}
```

Version 2 (Bit Modern Approach): Nested inner class based eager instantiation

Printer2.java

```
package com.sahu.components;

import com.sahu.helper.CommonsUtil;

public class Printer2 extends CommonsUtil {

        //Solution for InvalidClassException
        private static final long serialVersionUID = 2105646988646364920L;

        //Private constructor for avoid object creation using new keyword
        private Printer2() {
                if (InnerPrinter.INSTANCE!=null)
                        throw new RuntimeException("Object is already 
created!!!");
                System.out.println("Printer2.Printer()");
        }

        //Solution for Multi-Threaded problem
        private static class InnerPrinter {
                private volatile static Printer2 INSTANCE = new Printer2();
        }

        //Solution for cloning problem
        @Override
        public Object clone() throws CloneNotSupportedException {
                throw new CloneNotSupportedException("cloning is not 
allowed in Singleton class");
                //return InnerPrinter.INSTANCE;
        }

        //Solution for DeSerialization problem
```

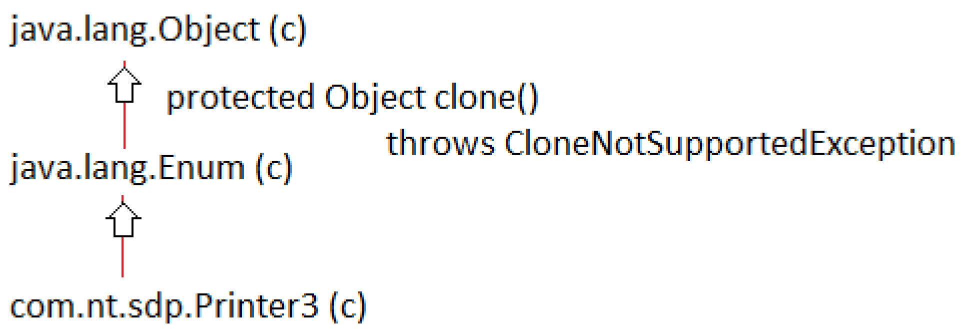
```
	public Object readResolve() {
		//throw new IllegalArgumentException("Deserialization is not a
allowed in Singleton class");
		return InnerPrinter.INSTANCE;
	}

	//Factory method to get class object
	public static Printer5 getInstane() {
		return InnerPrinter.INSTANCE;
	}

	//Business logic/ method
	public void print(String msg) {
		System.out.println(msg);
	}

}
```

Version3 (Enum Based): Latest Hot Cake with some limitations

Printer3.java

```
package com.sahu.components;

import java.io.Serializable;

public enum Printer3 implements Serializable {
	INSTANCE;

	//Static Factory method
	public static Printer3 getInstance() {
		return INSTANCE;
	}

	//Business method
	public static void print(String msg) {
		System.out.println(msg);
	}

}
```

Note: Every enum is internally java class extending java.lang.Enum class. This java.lang.Enum class implements java.io.Serializable (I). Every Enum is serializable indirectly.



Q. How Enum based Code is Perfect Singleton class even though we are writing less amount of code?

Ans.

a. The "INSTANCE" in the enum is eagerly INSTANTIATED So it is thread safe in multi-threaded environment.
b. We can't keep public constructors in the enum, so we can't create objects for Enums using "new" operator outside the class.
c. Every Enum extends from java.lang.Enum class internally and does not allows extending from any other classes explicitly, so we can't call protected clone() of java.lang.Object class on Enum INSTANCE. This indicates enum are not cloneable.
d. Enums supports Serialization. In Deserialization they return already created object.
e. For Enums we can't create object using Reflect API.
f. We can place business methods in the enum.

Note:

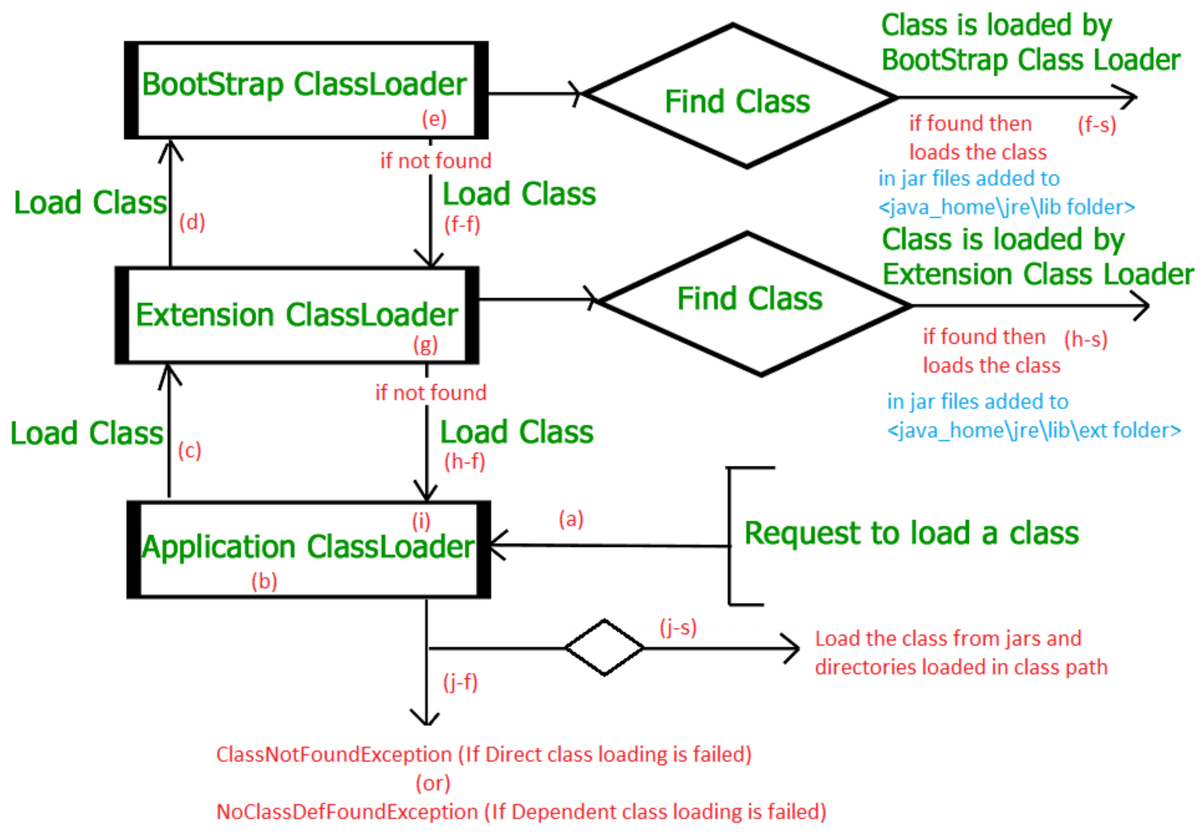
- ✓ Because of these reasons enum based code is simple code while developing singleton pattern.
- ✓ In Enum based singleton we can't use certain java features (Be careful about this limitation).

ClassLoader Subsystem

- JVM internally uses ClassLoader Subsystem to load the java classes that are required to execute Java app.
- In ClassLoader System sub system 3 ClassLoader are given
    a. BootStrap ClassLoader (capable loading classes from
    <java_home>\jre\lib folder jar files)

b. Extension ClassLoader (capable loading classes from <java_home>\jre\lib\ext folder jar files) (Removed from Java 9)

c. Application ClassLoader (capable loading classes from the jar files or directories added to the CLASSPATH environment variable)



BootStrap ClassLoader
(e)
Find Class
Class is loaded by BootStrap Class Loader
if found then loads the class
(f-s)
in jar files added to <java_home\jre\lib folder>
if not found
Load Class
(f-f)
Load Class
(d)
Extension ClassLoader
(g)
Find Class
Class is loaded by Extension Class Loader
if found then loads the class
(h-s)
in jar files added to <java_home\jre\lib\ext folder>
if not found
Load Class
(h-f)
Load Class
(c)
Application ClassLoader
(i)
(b)
(a)
Request to load a class
(j-s)
Load the class from jars and directories loaded in class path
(j-f)
ClassNotFoundException (If Direct class loading is failed)
(or)
NoClassDefFoundException (If Dependent class loading is failed)

Note:

- ✓ If the class loaded by parent ClassLoader then the same class will be not loaded by child ClassLoader (Principle of Uniqueness).
- ✓ The class loaded by parent ClassLoader is visible to child class loader but reverse is not possible (Principle of Visibility).
- ✓ When JVM gives a class to ClassLoader subsystem for Loading then it has to delegated all class loaders using class loader hierarchy (Principle of Delegation).

- We can create Custom ClassLoader by extending existing ClassLoaders and custom ClassLoaders also tries to satisfy all the 3 principles.
- URLClassLoader is readily available Custom ClassLoader. Which can load classes from the specified jar file and can also instantiate them as needed.

Procedure to create for Object Singleton java class using Custom class Loader (URLClassLoader):

Step 1: Make sure that jar file is created representing classes for whom you want to create the objects using Custom ClassLoader

Right click on the project --> export --> jar file --> ...... (creates jar file by the given name (sdp.jar)

Step 2: In client app use URLClassLoader as the Custom Loader to load the singleton class from given jar file and to create object by using Reflection API.

Singleton_ClassLoader.java

```
package com.sahu.test;

import java.lang.reflect.Method;
import java.net.URL;
import java.net.URLClassLoader;

import com.sahu.components.Printer;

public class Singleton_ClassLoader {

	public static void main(String[] args) throws Exception {
		//Using default class loader
		Printer printer = Printer.INSTANCE;
		System.out.println(printer.getClass().getClassLoader());
		System.out.println(printer.hashCode());
		System.out.println("--------------------------------");

		//Using Custom class loader
		URLClassLoader classloader = new URLClassLoader(new URL[]
{new URL("file:E:/JAVA/Workspace/DesignPattern/sdp.jar")}, null);
		//Load Class using Reflection API
		Class<?> clss =
classloader.loadClass("com.sahu.components.Printer");
		//get access to getInstance() method
		Method method = clss.getDeclaredMethod("getInstance", new
Class[] {});
		//invoke method
		Object obj = method.invoke(null);
		System.out.println(obj.getClass().getClassLoader());
```

```
                    System.out.println(obj.hashCode());
          }

}
```

- new URLClassLoader(new URL[] {new URL("file:E:/JAVA/Workspace/DesignPattern/sdp.jar")}, null);
  Here null is no parent class loader.
- clss.getDeclaredMethod("getInstance", new Class[] {});
  Here new Class [] {} is empty array because getInstance () is 0-param method.
- method.invoke(null);
  Here null because getInstance() is 0-param method.

Problem:

- Object obj = method.invoke(null);
  Since URLClassLoader is an independent ClassLoader. So, it loads Printer class separately and also creates object.

Solution:

- Create Custom ClassLoader by linking with ApplicationClassLoader where Singleton class is already loaded as parent class loader, because Uniqueness and Principle of Visibility the custom ClassLoader does not singleton class, so does not create new object rather it returns already created object.

Singleton_ClassLoader.java

```
                    //Using Custom class loader
                    URLClassLoader classloader = new URLClassLoader(new URL[]
{new URL("file:E:/JAVA/Workspace/DesignPattern/sdp.jar")},
printer.getClass().getClassLoader());
                    //Load Class using Reflection API
                    Class<?> clss =
classloader.loadClass("com.sahu.components.Printer");
                    //get access to getInstance() method
                    Method method = clss.getDeclaredMethod("getInstance", new
Class[] {});
                    //invoke method
                    Object obj = method.invoke(null);
```

Q. Can we develop perfect singleton java class?
Ans. 99% we can develop but 1% is not possible that is working with Custom ClassLoader to create the object of Singleton java class.

Note:
- ✓ In one JVM we can have multiple ClassLoaders as we have seen in above project.
- ✓ Spring's singleton scope is no way related to singleton java class. The IOC container of Spring does not make spring bean class as singleton java class if the scope is "singleton". The IOC container just creates only object for Spring bean and keeps that object in the cache for reusability though spring bean allows to create multiple objects
- ✓ It is like Servlet container creating only one object for our servlet component class through our servlet component class allows to create multiple objects.

Use-case of Singleton Java class

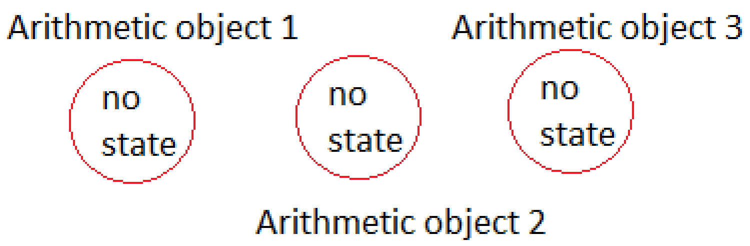
Q. When should we take java class as singleton java class?
Ans.
a) If class is not having any state (instance member variable)
b) If class is having just read only state (only final member variables)
c) If class is having sharable state across the multiple other classes in synchronized environment (use case: Cache implementation)

If class is not having any state (instance member variable):

Problem:

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

- This class is not having any state. So, creating multiple objects for this class without having state.is wastage of memory.



Solution 1: (You can apply all standards)

```
public class Arithmetic {
        private static Arithmetic INSTANCE;
        private Arithmetic () {
        }
        public static Arithmetic getInstance() {
                if (INSTANCE==null)
                        INSTANCE=new Arithmetic ();

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

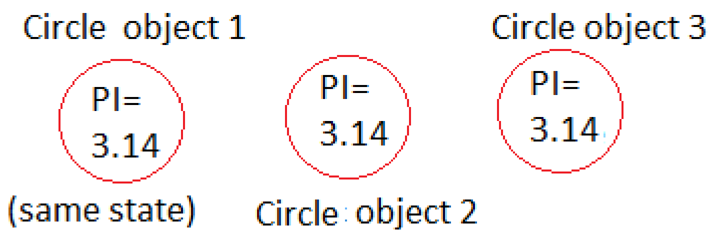
Solution 2:

```
public enum Arithmetic {
        INSTANCE;
        public static Arithmetic getInstance() {
                return INSTANCE;
        }
        public int sum (int x, int y) {
                return x+y;
        }
}
```

If class is having just read only state (only final member variables)

Problem:

```
public class Circle {
        private final float PI=3.14f;
        public float calcArea (float radius) {
                return PI*radius*radius;
        }
}
```



Solution 1: (You can apply all standards)

```
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;
        }
}
```

Solution 2:

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

If class is having sharable state across the multiple other classes in synchronized environment (use case: Cache implementation)

Problem:

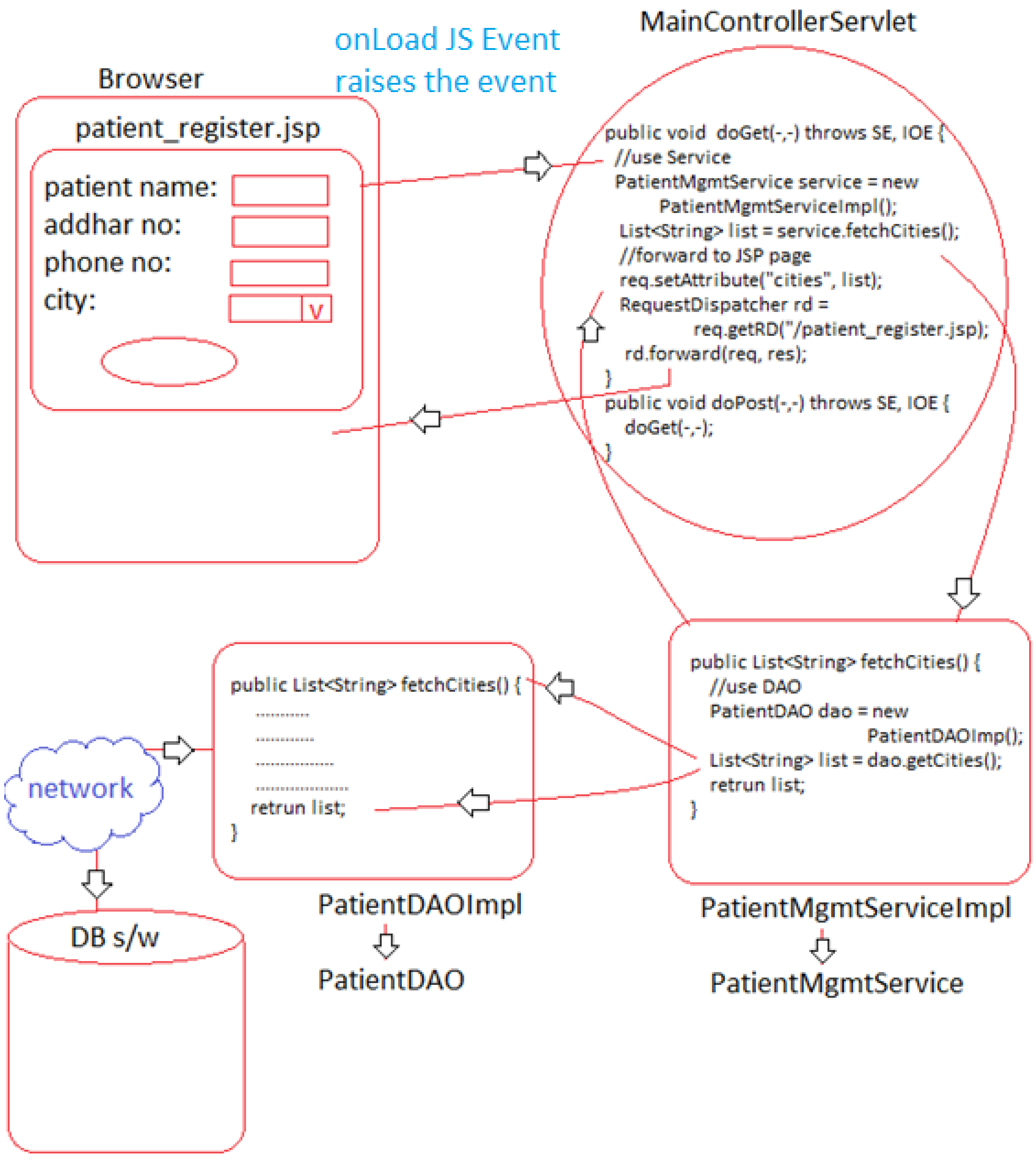
- If form page launched multiple times, then multiple times the app hits the DB software through DAO class to get same cities and to display in select box.

Solution:

1. Create DAO class object, call getCities() method in the constructor of service class. Similarly create service class obj and call fetchCities() method in the init() method of Controller Servlet component.
   (This looks like good solution if multiple service classes are not looking to use the above DAO logics).

2. Retrieve cities from DB s/w only for time through DAO class and keep them in Cache and use them across the multiple same requests. To make all the requests using same cache in all situations. It is recommended to develop the class "Cache" as singleton java class.

Problem:



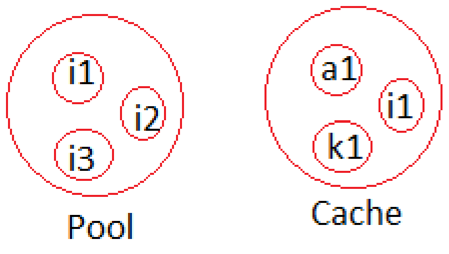
```
public void  doGet(-,-) throws SE, IOE {
  //use Service
  PatientMgmtService service = new
          PatientMgmtServiceImpl();
   List<String> list = service.fetchCities();
   //forward to JSP page
   req.setAttribute("cities", list);
   RequestDispatcher rd =
                  req.getRD("/patient_register.jsp);
    rd.forward(req, res);
}
public void doPost(-,-) throws SE, IOE {
    doGet(-,-);
}
```

```
public List<String> fetchCities() {
    ............
    ............
    ................
    ..................
   retrun list;
}
```

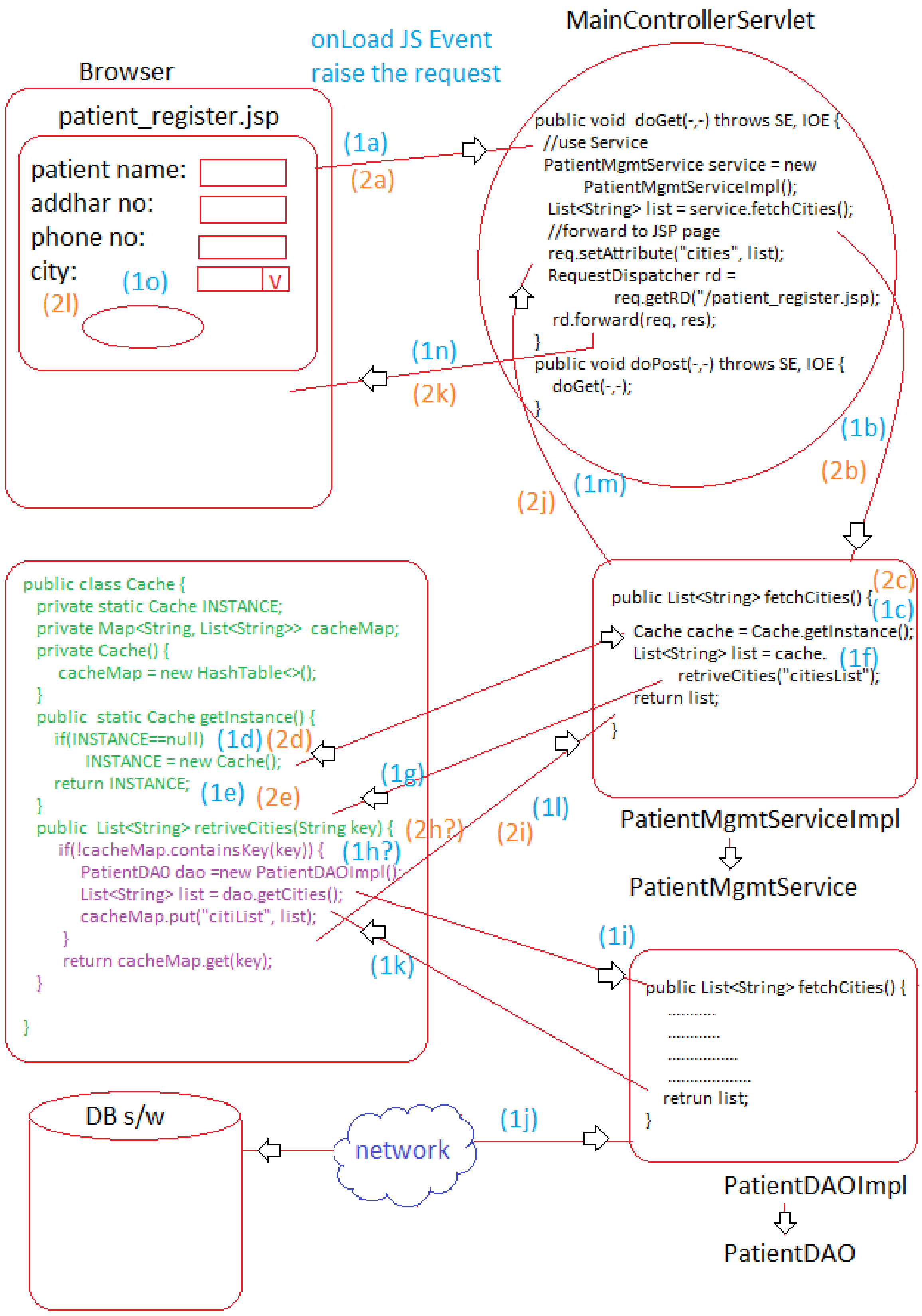
```
public List<String> fetchCities() {
   //use DAO
   PatientDAO dao = new
                    PatientDAOImp();
   List<String> list = dao.getCities();
   retrun list;
}
```

Set of same items is called as pool
(Use List collection for implementation)

Set of Different items is called as cache
(Use Map collection for implementation)



Solution 2: (Code using Cache)



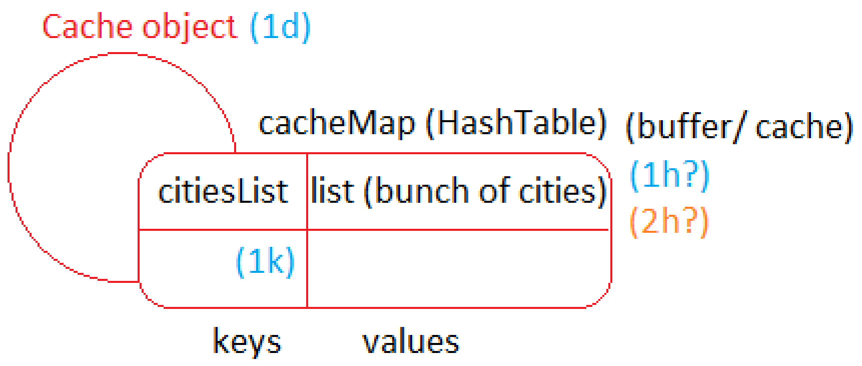
```
public void  doGet(-,-) throws SE, IOE {
  //use Service
  PatientMgmtService service = new
          PatientMgmtServiceImpl();
  List<String> list = service.fetchCities();
  //forward to JSP page
  req.setAttribute("cities", list);
  RequestDispatcher rd =
          req.getRD("/patient_register.jsp);
  rd.forward(req, res);
}
public void doPost(-,-) throws SE, IOE {
  doGet(-,-);
}
```

```
public List<String> fetchCities() {
  Cache cache = Cache.getInstance();
  List<String> list = cache.
          retriveCities("citiesList");
  return list;
}
```

```
public class Cache {
  private static Cache INSTANCE;
  private Map<String, List<String>>  cacheMap;
  private Cache() {
     cacheMap = new HashTable<>();
  }
  public  static Cache getInstance() {
     if(INSTANCE==null)
          INSTANCE = new Cache();
     return INSTANCE;
  }
  public  List<String> retriveCities(String key) {
     if(!cacheMap.containsKey(key)) {
        PatientDAO dao =new PatientDAOImpl();
        List<String> list = dao.getCities();
        cacheMap.put("citiList", list);
     }
     return cacheMap.get(key);
  }

}
```

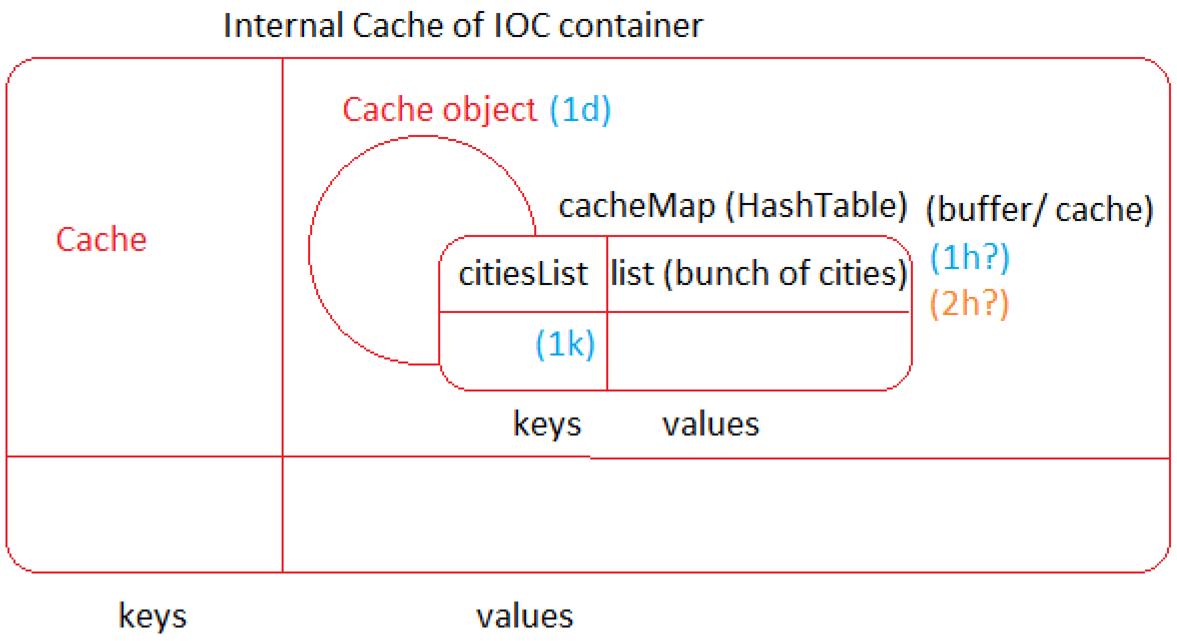
```
public List<String> fetchCities() {
   ............
   ............
   ................
   ..................
   retrun list;
}
```

1a to 1o: 1st Request
2a to 2l: 2nd Request
(Both are same request)



Note:

- ✓ If we develop the above Cache class as Spring Bean in Spring application then no need of taking the "Cache" class singleton java class just develop as normal class/ Spring bean having "singleton" scope.
- ✓ Because the IOC container itself creates and maintains Single scope Spring beans objects across the multiple requests and calls in its IOC container Internal cache of IOC container.



Factory Design Pattern

- There are basically 3 types of factory Design pattern
  1. Simple Factory Design Pattern
  2. Factory method Design Pattern (different java's factory method)
  3. Abstract Factory Design Pattern

Simple Factory Design Pattern

Problem:

1. Different objects for different classes will be created using different techniques but, I do not want expose those techniques end user.

   ```
   A a = new A ();
   B b = new (new C ());
   Calendar cal = Calendar.getInstance();
   Test t = new Test (10,20);
   ```

   - Making client app/ user app developer to know the process of creating object for class is going become bit heavy for client app/ user app developer.

2. Sometimes, we can't anticipate which class object has to be created until run time. So, creating objects for all the classes and using only one object from those objects is bad practice (wastage of memory and CPU time).

3. If the object creation is very complex then better not to expose that process directly to end user/ client app. Better to provide some middle man who provides abstraction on the object creation process.

Solution:

- Use Factory Pattern - It creates one of several related/ possible classes object based on data that we supply by providing abstraction on object creation process (if necessary, create and assign its dependent objects).

  E.g.,

  a. CarFactory provides abstraction on Car manufacturing process and gives car to the customer based on the model's name he chooses.
  b. BiscuitFactory provides abstraction on Biscuit creation process and gives to the customer based biscuit type he chooses.
  c. Connection con=DriverManager.getConnection(-,-,-);
     It creates and returns JDBC connection object for particular DB s/w based on the URL that we passed, but programmer need not to know any related to connection object creation process.
  d. factory.getBean(-) method if spring creates and return Spring bean object by hiding its object creational process based on the bean id we pass.

spring IOC container name is BeanFactory i.e., it is factory to create and return spring bean objects by providing abstraction on their object creation process.

e. Hibernate SessionFactory is given based Factory Design Pattern because it provides abstraction on Hibernate Session object process.

```
Session ses=factory.openSession();
```

f. In JDBC Programming, all jdbc objects created through factory pattern process.

```
Statement st=con.createStatement();
ResultSet rs=st.executeQeury(-);
PreparedSatatement ps=con.prepareStatement(-);
```

(In the creation of all these JDBC objects factory pattern is involved directly or indirectly)

Note:

- ✓ Spring IOC containers are designed based on two patterns
  - Factory Design pattern
  - Flyweight Design pattern.
- ✓ Abstraction is hiding the process/ implementation while utilizing the functionality.
- ✓ Factory Design pattern returns one of several related classes object based on the data that we pass. Here Related classes means the classes must implement either common interface or must extend from common class/ abstract class. Due to this
  we can refer the object returned by the factory by using that common interface ref variable/ common super class ref variable.

```
class A implements X {
        …………
        ………..
}
```

```
class B implements X {
        …………
        ………..
}
```

```
A a = new A ();
B b = new B ();

X x = new A (); (Good approach)
  x = new B ();
```

- Every class that is representing Factory Design Pattern will have on factory method accepting data and returning one of several related classes object based on the accepted data. This factory method contains commons super class/ interface as the return type.

```
public class MyFactory {
        static X createObject (String type) {
                if(type.equals("a"))
                        return new A ();
                else
                        return new B ();
         }
}
```

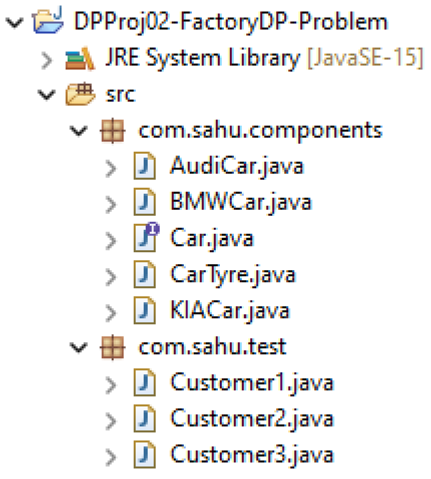
Client App/ user App

```
X x1 = MyFactory.createObject("a"); //get "A" object
X x2 = MyFactory.createObject("b"); //get "B" object
```

Note:

- ✓ If java method returns type, is an interface then method returns one implantation class object of that interface.
- ✓ If java method return type is an abstract class, then method return one sub class object of that abstract class.
- ✓ If java method return type is concrete class, then method can either return concrete class object or one of its sub class objects.

Directory Structure of DPProj02-FactoryDP-Problem:



```
DPProj02-FactoryDP-Problem
  JRE System Library [JavaSE-15]
  src
    com.sahu.components
      AudiCar.java
      BMWCar.java
      Car.java
      CarTyre.java
      KIACar.java
    com.sahu.test
      Customer1.java
      Customer2.java
      Customer3.java
```

- Develop the above directory structure package, class in src folder.
- Then place the following code with in their respective file.

Car.java

```
package com.sahu.components;

public interface Car {
	public void assemble();
	public void roadTest();
}
```

CarTyre.java

```
package com.sahu.components;

public class CarTyre {

	public CarTyre() {
		System.out.println("CarTyre.CarTyre()");
	}

	@Override
	public String toString() {
		return "Apollo Tyres";
	}

}
```

AudiCar.java

```
package com.sahu.components;

public class AudiCar implements Car {

	private CarTyre tyre;

	public AudiCar(CarTyre tyre) {
		this.tyre = tyre;
	}

	@Override
	public void assemble() {
		System.out.println("AudiCar.assemble()");
	}
	@Override
	public void roadTest() {
		System.out.println("AudiCar.roadTest()");
	}

	@Override
	public String toString() {
		return "AudiCar [tyre=" + tyre + "]";
	}

}
```

BMWCar.java

```
package com.sahu.components;

public class BMWCar implements Car {

	private CarTyre tyre;

	public BMWCar(CarTyre tyre) {
		this.tyre = tyre;
	}

	@Override
	public void assemble() {
		System.out.println("BMWCar.assemble()");
	}

	@Override
	public void roadTest() {
		System.out.println("BMWCar.roadTest()");
	}

	@Override
	public String toString() {
		return "BMWCar [tyre=" + tyre + "]";
	}

}
```

KIACar.java

```
package com.sahu.components;

public class KIACar implements Car {

	private CarTyre tyre;

	public KIACar(CarTyre tyre) {
		this.tyre = tyre;
	}

	@Override
	public void assemble() {
		System.out.println("KIACar.assemble()");
	}

	@Override
	public void roadTest() {
		System.out.println("KIACar.roadTest()");
	}

	@Override
	public String toString() {
		return "KIACar [tyre=" + tyre + "]";
	}

}
```

Customer1.java

```
package com.sahu.test;

import com.sahu.components.AudiCar;
import com.sahu.components.CarTyre;

public class Customer1 {

	public static void main(String[] args) {
		CarTyre tyre = new CarTyre();
		AudiCar car = new AudiCar(tyre);
		car.assemble();
```

```
                car.assemble();
                car.roadTest();
                System.out.println(car);
        }

}
```

Customer2.java

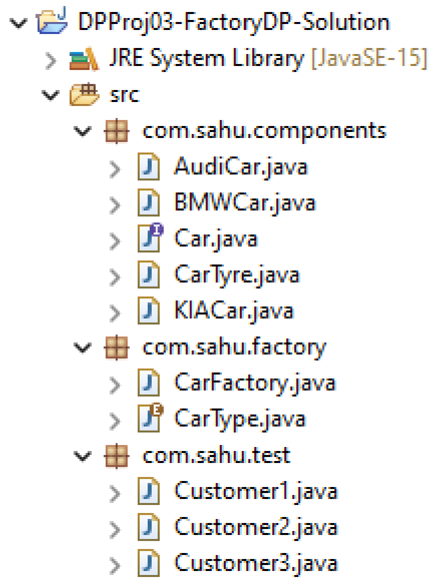
```
package com.sahu.test;

import com.sahu.components.BMWCar;
import com.sahu.components.CarTyre;

public class Customer2 {

        public static void main(String[] args) {
                CarTyre tyre = new CarTyre();
                BMWCar car = new BMWCar(tyre);
                car.assemble();
                car.roadTest();
                System.out.println(car);
        }

}
```

Customer3.java

```
package com.sahu.test;

import com.sahu.components.KIACar;
import com.sahu.components.CarTyre;

public class Customer3 {

        public static void main(String[] args) {
                CarTyre tyre = new CarTyre();
                KIACar car = new KIACar(tyre);
                car.assemble();
                car.roadTest();
                System.out.println(car);
        }
}
```

- As part of solution take a factory class having factory method that will provide abstraction on object creation process and better to take a Car type enum as below.

Directory Structure of DPProj03-FactoryDP-Solution:



```
DPProj03-FactoryDP-Solution
  JRE System Library [JavaSE-15]
  src
    com.sahu.components
      AudiCar.java
      BMWCar.java
      Car.java
      CarTyre.java
      KIACar.java
    com.sahu.factory
      CarFactory.java
      CarType.java
    com.sahu.test
      Customer1.java
      Customer2.java
      Customer3.java
```

- Develop the above directory structure package, class in src folder.
- Then use the following code with in their respective file.
- Copy the rest of code from previous application.

CarType.java

```java
package com.sahu.factory;

public enum CarType {
	AUDI, BMW, KIA;
}
```

CarFactory.java

```java
package com.sahu.factory;

import com.sahu.components.AudiCar;
import com.sahu.components.BMWCar;
import com.sahu.components.Car;
import com.sahu.components.CarTyre;
import com.sahu.components.KIACar;

public class CarFactory {
	public static Car getCar(CarType carType) {
		Car car=null;
		//Create Tyre object
		CarTyre tyre = new CarTyre();

		if (carType == CarType.BMW)
			car = new BMWCar(tyre);
		else if (carType == CarType.AUDI)
			car = new AudiCar(tyre);
		else
			car = new KIACar(tyre);

		car.assemble();
		car.roadTest();
		return car;
	}

}
```

Customer1.java

```
package com.sahu.test;

import com.sahu.components.Car;
import com.sahu.factory.CarFactory;
import com.sahu.factory.CarType;

public class Customer1 {

	public static void main(String[] args) {
		Car car = CarFactory.getCar(CarType.BMW);
		System.out.println(car);
	}

}
```

Customer2.java

```
package com.sahu.test;


import com.sahu.components.Car;
import com.sahu.factory.CarFactory;
```

```java
import com.sahu.factory.CarType;

public class Customer2 {

	public static void main(String[] args) {
		Car car = CarFactory.getCar(CarType.AUDI);
		System.out.println(car);
	}

}
```

Customer3.java

```java
package com.sahu.test;

import com.sahu.components.Car;
import com.sahu.factory.CarFactory;
import com.sahu.factory.CarType;

public class Customer3 {

	public static void main(String[] args) {
		Car car = CarFactory.getCar(CarType.KIA);
		System.out.println(car);
	}

}
```

Factory Method Design Pattern

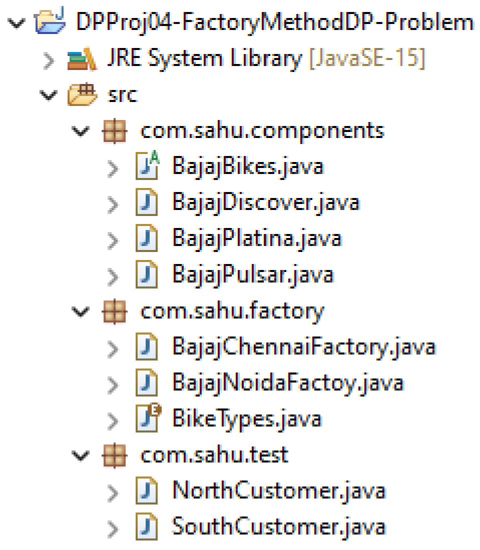
- Factory method is different from Factory method design pattern.
- Factory method creates and returns object to caller. It is required in multiple design patterns implementation like singleton class, factory design pattern, factory method pattern, strategy pattern and etc.
- Factory method design pattern defines set of rules and guidelines (standards) for creating objects of same family classes (classes belonging same inheritance hierarchy) when multiple factories are creating those objects.
- The classes that belong to same inheritance hierarchy either having common super class or common implementing interface are called same family classes.

Problem:

```
Bike (AC)
        |--> PulsorBike (CC)
        |--> DiscoverBike (CC)

Bajaj Company
        |--> ChennaiBajajFactory (South Customers)
        |--> NoidaBajajFactory (North Customers)
```

- If ChennaiFactory uses its own standards to manufacture for Bajaj familiarly bikes, and if NoidaFactory uses some other standards (fewer quality standards) to manufacture same old Bajaj family bikes. Then it will be problem to sell the Bajaj family bikes in the market.
- Customer may think that Chennai factory bikes very good in quality and NoidaFactory bikes are bit poor in quality. To overcome this problem, we need defined set of standards for all BajajFactories to follow by using Factory method design pattern.

Directory Structure of DPProj04-FactoryMethodDP-Problem:



```
DPProj04-FactoryMethodDP-Problem
  JRE System Library [JavaSE-15]
  src
    com.sahu.components
      BajajBikes.java
      BajajDiscover.java
      BajajPlatina.java
      BajajPulsar.java
    com.sahu.factory
      BajajChennaiFactory.java
      BajajNoidaFactoy.java
      BikeTypes.java
    com.sahu.test
      NorthCustomer.java
      SouthCustomer.java
```

- Develop the above directory structure package, class in src folder.
- Then use the following code with in their respective file.

BajajBikes.java

```java
package com.sahu.components;
public abstract class BajajBikes {
	public abstract void drive();
}
```

BajajDiscovr.java

```java
package com.sahu.components;

public class BajajDiscover extends BajajBike {

	public BajajDiscover() {
		System.out.println("BajajDiscover.BajajDiscover()");
	}

	@Override
	public void drive() {
		System.out.println("Driving Bajaj Discover bike");
	}

}
```

BajajPlatina.java

```java
package com.sahu.components;

public class BajajPlatina extends BajajBike {

	public BajajPlatina() {
		System.out.println("BajajPlatina.BajajPlatina()");
	}

	@Override
	public void drive() {
		System.out.println("Driving Bajaj Platina bike");
	}

}
```

BajajPulsar.java

```java
package com.sahu.components;

public class BajajPulsar extends BajajBike {

	public BajajPulsar() {
		System.out.println("BajajPulsar.BajajPulsar()");
	}

	@Override
	public void drive() {
		System.out.println("Driving Bajaj Pulsar bike");
	}

}
```

BikeTypes.java

```
package com.sahu.factory;

public enum BikeType {
	DISCOVER, PLATINA, PULSAR;
}
```

BajajChennaiFactory.java

```
package com.sahu.factory;

import com.sahu.components.BajajBike;
import com.sahu.components.BajajDiscover;
import com.sahu.components.BajajPlatina;

public class BajajChennaiFactory {

	public static void paint() {
		System.out.println("Painting Bajaj bike");
	}

	public static void assemble() {
		System.out.println("Assembling Bajaj bike");
	}

	public static void roadTest() {
		System.out.println("Road test Bajaj bike");
	}

	public static void oiling() {
		System.out.println("Oiling test Bajaj bike");
	}

	//factory method
	public static BajajBike createBike(BikeType bikeType) {
		BajajBike bike = null;
		if (bikeType == BikeType.DISCOVER)
			bike = new BajajDiscover();
		else if (bikeType == BikeType.PLATINA)
			bike = new BajajPlatina();
		else if (bikeType == BikeType.PULSAR)
			bike = new BajajDiscover();

		paint();
		assemble();
		oiling();
		roadTest();

		return bike;
	}

}
```

BajajNoidaFactory.java

```
package com.sahu.factory;

import com.sahu.components.BajajBike;
import com.sahu.components.BajajDiscover;
import com.sahu.components.BajajPlatina;

public class BajajNoidaFactory {

	public static void paint() {
		System.out.println("Painting Bajaj bike");
	}

	public static void assemble() {
		System.out.println("Assembling Bajaj bike");
	}

	public static void roadTest() {
```

```
                System.out.println("Road test Bajaj bike");
        }

        public static void oiling() {
                System.out.println("Oiling test Bajaj bike");
        }

        //factory method
        public static BajajBike createBike(BikeType bikeType) {
                BajajBike bike = null;
                if (bikeType == BikeType.DISCOVER)
                        bike = new BajajDiscover();
                else if (bikeType == BikeType.PLATINA)
                        bike = new BajajPlatina();
                else if (bikeType == BikeType.PULSAR)
                        bike = new BajajDiscover();

                assemble();
                paint();
                oiling();

                return bike;
        }

}
```

NorthCustomer.java

```
package com.sahu.test;

import com.sahu.components.BajajBike;
import com.sahu.factory.BajajNoidaFactory;
import com.sahu.factory.BikeType;

public class NorthCustomer {

        public static void main(String[] args) {
                BajajBike bike = 
BajajNoidaFactory.createBike(BikeType.PULSAR);
                bike.drive();
        }
}
```

SouthCustomer.java

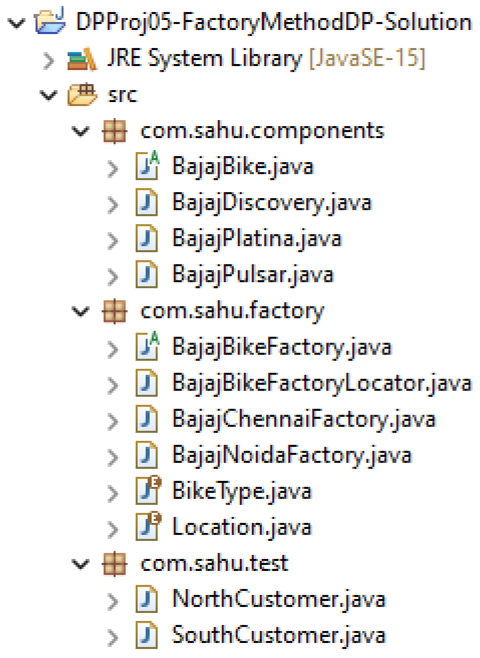
```
package com.sahu.test;

import com.sahu.components.BajajBike;
import com.sahu.factory.BajajChennaiFactory;
import com.sahu.factory.BikeType;

public class SouthCustomer {

	public static void main(String[] args) {
		BajajBike bike =
BajajChennaiFactory.createBike(BikeType.PULSAR);
		bike.drive();
	}

}
```

Solution:

- Here South Customer is getting quality Bajaj bike from Chennai factory whereas North customer is fewer quality bikes from Noida factory (because painting is after assembling and no road test is taking place).
- To overcome this problem, develop a super class for all factory classes defining the standards as shown below (Abstract class with abstract methods).

```
class Test {
	public void m1 () {(c)
		m2(); (d)
	}
	public void m2 () {
		………..
	}
}
class Demo extends Test {
	public void m2 () {
		……. (e)
	}
}
Demo d = new Demo (); (a)
d.m1 (); (b)
```

Directory Structure of DPProj05-FactoryMethodDP-Solution:


```
DPProj05-FactoryMethodDP-Solution
  JRE System Library [JavaSE-15]
  src
    com.sahu.components
      BajajBike.java
      BajajDiscovery.java
      BajajPlatina.java
      BajajPulsar.java
    com.sahu.factory
      BajajBikeFactory.java
      BajajBikeFactoryLocator.java
      BajajChennaiFactory.java
      BajajNoidaFactory.java
      BikeType.java
      Location.java
    com.sahu.test
      NorthCustomer.java
      SouthCustomer.java
```

- Develop the above directory structure package, class in src folder.
- Then use the following code with in their respective file.
- Copy the rest of code from previous application.

BajajBikeFactory.java

```
package com.sahu.factory;

import com.sahu.components.BajajBike;

public abstract class BajajBikeFactory {
	public abstract void paint();
	public abstract void assemble();
	public abstract void oiling();
	public abstract void roadTest();
	protected abstract BajajBike createBike(BikeType bikeType);

	//Method having factory method design pattern logic
	public final BajajBike orderBike(BikeType bikeType) {
		paint();
		assemble();
		oiling();
		BajajBike bike = createBike(bikeType);
		roadTest();
```

```
                return bike;
        }

}
```

Location.java

```
package com.sahu.factory;

public enum Location {
        CHENNAI, NODIA;
}
```

BajajBikeFactoryLocator.java

```
package com.sahu.factory;

public class BajajBikeFactoryLocator {

        public static BajajBikeFactory buildBikeFactory(Location location) {
                BajajBikeFactory bikeFactory = null;
                if (location.equals(Location.CHENNAI))
                        bikeFactory = new BajajChennaiFactory();
                else if (location.equals(Location.NODIA))
                        bikeFactory = new BajajNoidaFactory();

                return bikeFactory;
        }

}
```

BajajChennaiFactory.java

```
package com.sahu.factory;

import com.sahu.components.BajajBike;
import com.sahu.components.BajajDiscover;
import com.sahu.components.BajajPlatina;

public class BajajChennaiFactory extends BajajBikeFactory {
	@Override
	public void paint() {
		System.out.println("Painting Bajaj bike");
	}

	@Override
	public void assemble() {
		System.out.println("Assembling Bajaj bike");
	}

	@Override
	public void roadTest() {
		System.out.println("Road test Bajaj bike");
	}

	@Override
	public void oiling() {
		System.out.println("Oiling test Bajaj bike");
	}

	@Override
	public BajajBike createBike(BikeType bikeType) {
		BajajBike bike = null;
		if (bikeType == BikeType.DISCOVER)
			bike = new BajajDiscover();
		else if (bikeType == BikeType.PLATINA)
			bike = new BajajPlatina();
		else if (bikeType == BikeType.PULSAR)
			bike = new BajajDiscover();

		return bike;
	}

}
```

BajajNoidaFactory.java

```
package com.sahu.factory;

import com.sahu.components.BajajBike;
import com.sahu.components.BajajDiscover;
import com.sahu.components.BajajPlatina;

public class BajajNoidaFactory extends BajajBikeFactory {

	@Override
	public void paint() {
		System.out.println("Painting Bajaj bike");
	}

	@Override
	public void assemble() {
		System.out.println("Assembling Bajaj bike");
	}

	@Override
	public void roadTest() {
		System.out.println("Road test Bajaj bike");
	}

	@Override
	public void oiling() {
		System.out.println("Oiling test Bajaj bike");
	}

	@Override
	public BajajBike createBike(BikeType bikeType) {
		BajajBike bike = null;
		if (bikeType == BikeType.DISCOVER)
			bike = new BajajDiscover();
		else if (bikeType == BikeType.PLATINA)
			bike = new BajajPlatina();
		else if (bikeType == BikeType.PULSAR)
			bike = new BajajDiscover();

		return bike;
	}

}
```

NorthCustomer.java

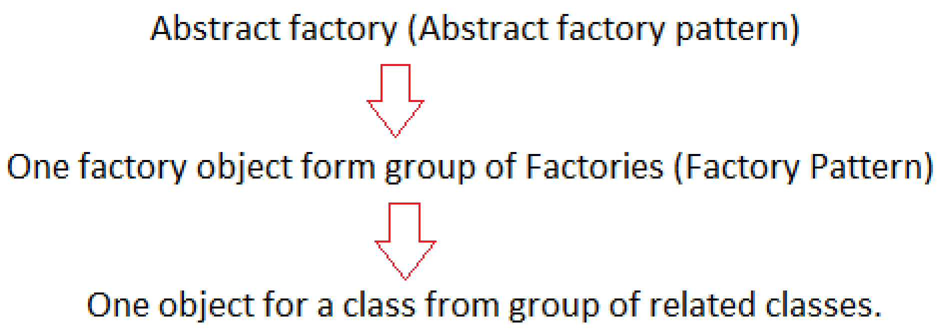
```
package com.sahu.test;

import com.sahu.components.BajajBike;
import com.sahu.factory.BajajBikeFactory;
import com.sahu.factory.BajajBikeFactoryLocator;
import com.sahu.factory.BikeType;
import com.sahu.factory.Location;

public class NorthCustomer {

	public static void main(String[] args) {
		BajajBikeFactory factory = 
BajajBikeFactoryLocator.buildBikeFactory(Location.NODIA);
		BajajBike bike = factory.orderBike(BikeType.DISCOVER);
		bike.drive();
	}

}
```

SouthCustomer.java

```
package com.sahu.test;

import com.sahu.components.BajajBike;
import com.sahu.factory.BajajBikeFactory;
import com.sahu.factory.BajajBikeFactoryLocator;
import com.sahu.factory.BikeType;
import com.sahu.factory.Location;

public class SouthCustomer {

	public static void main(String[] args) {
		BajajBikeFactory factory = 
BajajBikeFactoryLocator.buildBikeFactory(Location.CHENNAI);
		BajajBike bike = factory.orderBike(BikeType.PLATINA);
		bike.drive();
	}

}
```

Abstract/ Super Factory Design Pattern

- It is called super factory or factory of factories. It first returns one factory object from group related factories and each factory can be used to create object for group related classes.
- The Abstract factory pattern is one level of abstraction higher than the factory pattern.
- You can use this pattern to return one of several related classes of objects, each of can return several different objects on request.
- In other words, the Abstract factory is a factory object that returns one of several groups of classes. You might even decide to return from that group by using a simple factory.
- Abstract factory can be treated as a super factory or a factory of factories.
- Using factory design pattern, we abstract the object creation process of another class.
- Using the factory pattern abstract the objects creation process of family of classes.


Abstract factory (Abstract factory pattern)
One factory object form group of Factories (Factory Pattern)
One object for a class from group of related classes.

Problem:

DAO ------

ExcelStudentDAO
ExcelCourseDAO

DBStudentDAO
DBCourseDAO

(Factory 1)
ExcelDAOFactory
|--> createDAO(String type)

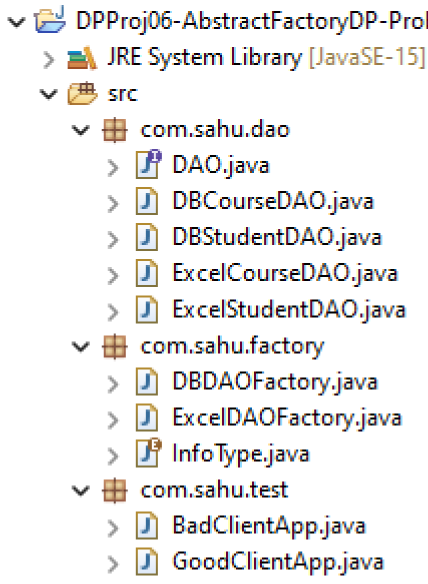
(Factory2)
DBDAOFactory
|--> createDAO(String type)

Client App

```
ExcelStudentDAO stDAO = ExcelDAOFactory.createDAO("student");
stDAO.insert(); //inserts student info excel sheet
DBCourseDAO courseDAO = DBDAOFactory.createDAO("course");
courseDAO.insert(); //inserts course info to DB table
```

Here client app/ user app developer is getting one wrong option i.e., inserting student details to excel and inserting course details to DB. That is not correct, both student and course details either should be inserted in same DB s/w or in same excel sheet.

Directory Structure of DPProj06-AbstractFactoryDP-Problem:



```
DPProj06-AbstractFactoryDP-Problem
  JRE System Library [JavaSE-15]
  src
    com.sahu.dao
      DAO.java
      DBCourseDAO.java
      DBStudentDAO.java
      ExcelCourseDAO.java
      ExcelStudentDAO.java
    com.sahu.factory
      DBDAOFactory.java
      ExcelDAOFactory.java
      InfoType.java
    com.sahu.test
      BadClientApp.java
      GoodClientApp.java
```

- Develop the above directory structure package, class in src folder.
- Then place the following code with in their respective files.

DAO.java

```
package com.sahu.dao;

public interface DAO {
	public void insert();
}
```

DBCourseDAO.java

```
package com.sahu.dao;

public class DBCourseDAO implements DAO {
	@Override
	public void insert() {
		System.out.println("DBCourseDAO.insert() - Inserting course 
details into DB s/w");
	}
}
```

DBStudentDAO.java

```
package com.sahu.dao;

public class DBStudentDAO implements DAO {

	@Override
	public void insert() {
		System.out.println("DBStudentDAO.insert() - Inserting student
details into DB s/w");
	}

}
```

ExcelCourseDAO.java

```
package com.sahu.dao;

public class ExcelCourseDAO implements DAO {

	@Override
	public void insert() {
		System.out.println("ExcelCourseDAO.insert() - Inserting course
details into excel sheet");
	}

}
```

ExcelStudentDAO.java

```
package com.sahu.dao;

public class ExcelStudentDAO implements DAO {

	@Override
	public void insert() {
		System.out.println("ExcelStudentDAO.insert() - Inserting
student details into excel sheet");
	}

}
```

InfoType.java

```
package com.sahu.factory;

public enum InfoType {
	STUDENT, COURSE;
}
```

DBDAOFactory.java

```
package com.sahu.factory;

import com.sahu.dao.DAO;
import com.sahu.dao.DBCourseDAO;
import com.sahu.dao.DBStudentDAO;

public class DBDAOFactory {

	public static DAO createDAO(InfoType type) {
		DAO dao = null;
		if (type.equals(InfoType.STUDENT))
			dao = new DBStudentDAO();
		else if (type.equals(InfoType.COURSE))
			dao = new DBCourseDAO();

		return dao;
	}

}
```

ExcelDAOFactory.java

```
package com.sahu.factory;

import com.sahu.dao.DAO;
import com.sahu.dao.ExcelCourseDAO;
import com.sahu.dao.ExcelStudentDAO;

public class ExcelDAOFactory {

	public static DAO createDAO(InfoType type) {
		DAO dao = null;
```

```
                DAO dao = null;
                if (type.equals(InfoType.STUDENT))
                        dao = new ExcelStudentDAO();
                else if (type.equals(InfoType.COURSE))
                        dao = new ExcelCourseDAO();

                return dao;
        }

}
```

BadClientApp.java

```
package com.sahu.test;

import com.sahu.dao.DAO;
import com.sahu.factory.DBDAOFactory;
import com.sahu.factory.ExcelDAOFactory;
import com.sahu.factory.InfoType;

public class BadClientApp {

        public static void main(String[] args) {
                DAO stDAO = ExcelDAOFactory.createDAO(InfoType.STUDENT);
                DAO crsDAO = DBDAOFactory.createDAO(InfoType.COURSE);

                stDAO.insert();
                crsDAO.insert();
        }

}
```

GoodClientApp.java

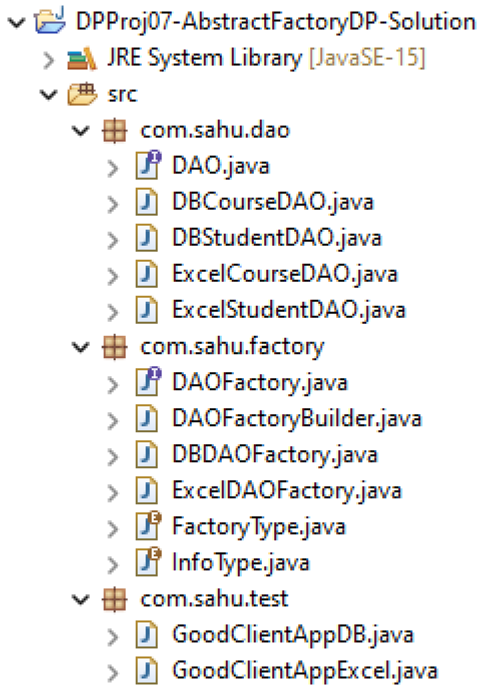
```
package com.sahu.test;

import com.sahu.dao.DAO;
import com.sahu.factory.ExcelDAOFactory;
import com.sahu.factory.InfoType;

public class GoodClientApp {
        public static void main(String[] args) {
                DAO stDAO = ExcelDAOFactory.createDAO(InfoType.STUDENT);
                DAO crsDAO = ExcelDAOFactory.createDAO(InfoType.COURSE);

                stDAO.insert();
                crsDAO.insert();
        }

}
```

Note: DAO means Java class having persistence logic.

Solution:

- To overcome this problem, we need to use Abstract factory design pattern.
- Which is nothing developing Factory class on the top of existing two DAO Factories and making client apps/ user apps getting group of DAO classes object from DAO factory given by Abstract factory design pattern.

Directory Structure of DPProj07-AbstractFactoryDP-Solution:



```
DPProj07-AbstractFactoryDP-Solution
  JRE System Library [JavaSE-15]
  src
    com.sahu.dao
      DAO.java
      DBCourseDAO.java
      DBStudentDAO.java
      ExcelCourseDAO.java
      ExcelStudentDAO.java
    com.sahu.factory
      DAOFactory.java
      DAOFactoryBuilder.java
      DBDAOFactory.java
      ExcelDAOFactory.java
      FactoryType.java
      InfoType.java
    com.sahu.test
      GoodClientAppDB.java
      GoodClientAppExcel.java
```

- Develop the above directory structure package, class in src folder.
- Then use the following code with in their respective file.
- Copy the rest of code from previous application.

DAOFactory.java

```java
package com.sahu.factory;

import com.sahu.dao.DAO;

public interface DAOFactory {
	public DAO createDAO(InfoType infoType);
}
```

DBDAOFactory.java

```java
package com.sahu.factory;

import com.sahu.dao.DAO;
import com.sahu.dao.DBCourseDAO;
import com.sahu.dao.DBStudentDAO;

public class DBDAOFactory implements DAOFactory {

	@Override
	public DAO createDAO(InfoType type) {
		DAO dao = null;
		if (type.equals(InfoType.STUDENT))
			dao = new DBStudentDAO();
		else if (type.equals(InfoType.COURSE))
			dao = new DBCourseDAO();

		return dao;
	}

}
```

ExcelDAOFactory.java

```java
package com.sahu.factory;

import com.sahu.dao.DAO;
import com.sahu.dao.ExcelCourseDAO;
import com.sahu.dao.ExcelStudentDAO;

public class ExcelDAOFactory implements DAOFactory {
        @Override
        public DAO createDAO(InfoType type) {
                DAO dao = null;
                if (type.equals(InfoType.STUDENT))
                        dao = new ExcelStudentDAO();
                else if (type.equals(InfoType.COURSE))
                        dao = new ExcelCourseDAO();

                return dao;
        }

}
```

FactoryType.java

```
package com.sahu.factory;

public enum FactoryType {
        DB,
        EXCEL;
}
```

DAOFactoryBuilder.java

```
package com.sahu.factory;

public class DAOFactoryBuilder {

        //Abstract factory logic method returning one of factory object
        public static DAOFactory buildDAOFactory(FactoryType factoryType) {
                DAOFactory daoFactory = null;
                if (factoryType.equals(FactoryType.DB))
                        daoFactory = new DBDAOFactory();
                else if (factoryType.equals(FactoryType.EXCEL))
                        daoFactory = new ExcelDAOFactory();

                return daoFactory;
        }

}
```

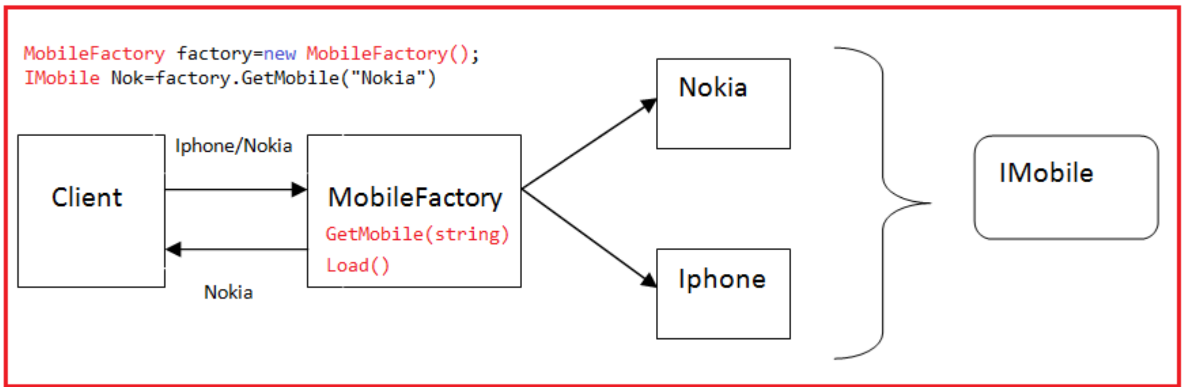
GoodClientApp.java

```
package com.sahu.test;

import com.sahu.dao.DAO;
import com.sahu.factory.DAOFactory;
import com.sahu.factory.DAOFactoryBuilder;
import com.sahu.factory.FactoryType;
import com.sahu.factory.InfoType;

public class GoodClientApp {

	public static void main(String[] args) {
		DAOFactory daoFactory =
DAOFactoryBuilder.buildDAOFactory(FactoryType.DB);
		DAO stDAO = daoFactory.createDAO(InfoType.STUDENT);
		DAO crsDAO = daoFactory.createDAO(InfoType.COURSE);

		stDAO.insert();
		crsDAO.insert();
	}

}
```

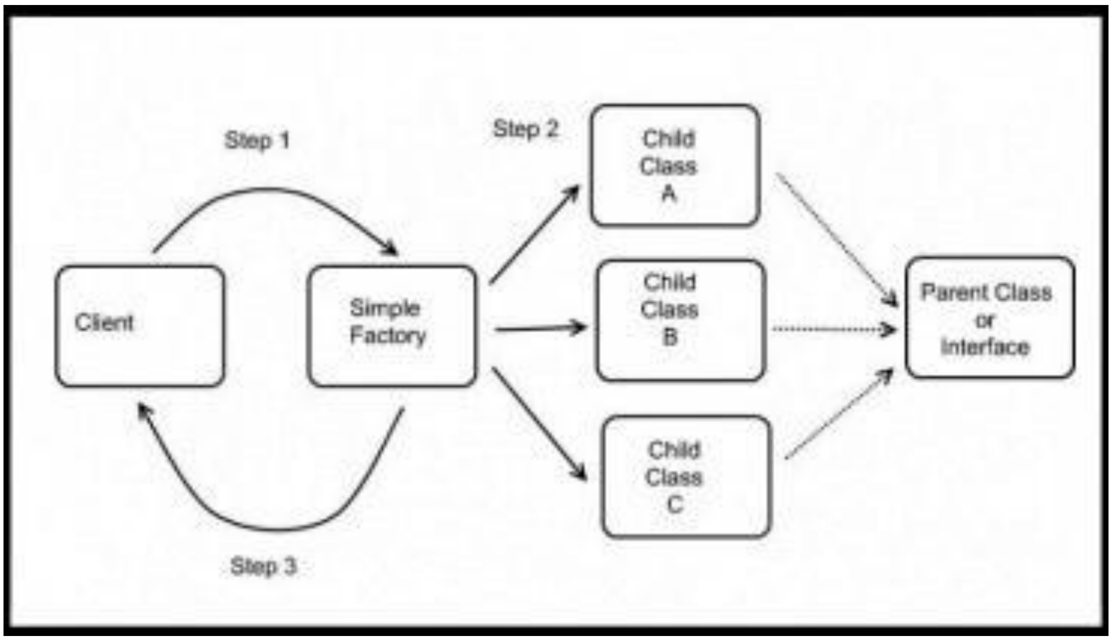
Simple Factory vs Factory Method vs Abstract Factory DP

Simple Factory

- The easiest way for me to remember this is that a simple factory is called directly by the class which wants to create an object (the calling class is referred to as the "client"). The simple factory returns one of many different classes. All the classes that a simple factory can return either inherit from the same parent class, or implement the same interface.



```
MobileFactory factory=new MobileFactory();
IMobile Nok=factory.GetMobile("Nokia")
```



Step One: You call a method in the factory. Here it makes sense to use a static method. The parameters for your call tell the factory which class to create.

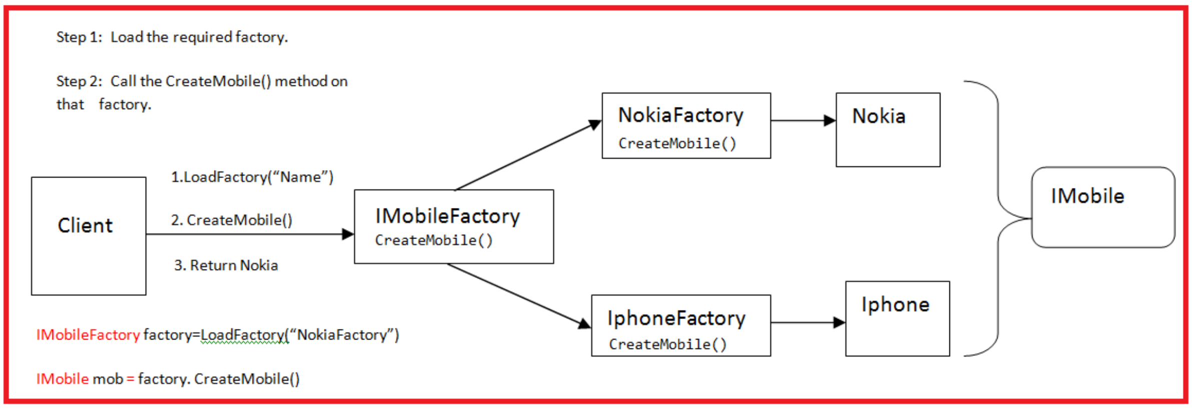
Step Two: The factory creates your object. The only thing to note is that of all the objects it can create, the objects have the same parent class, or implement the same interface.

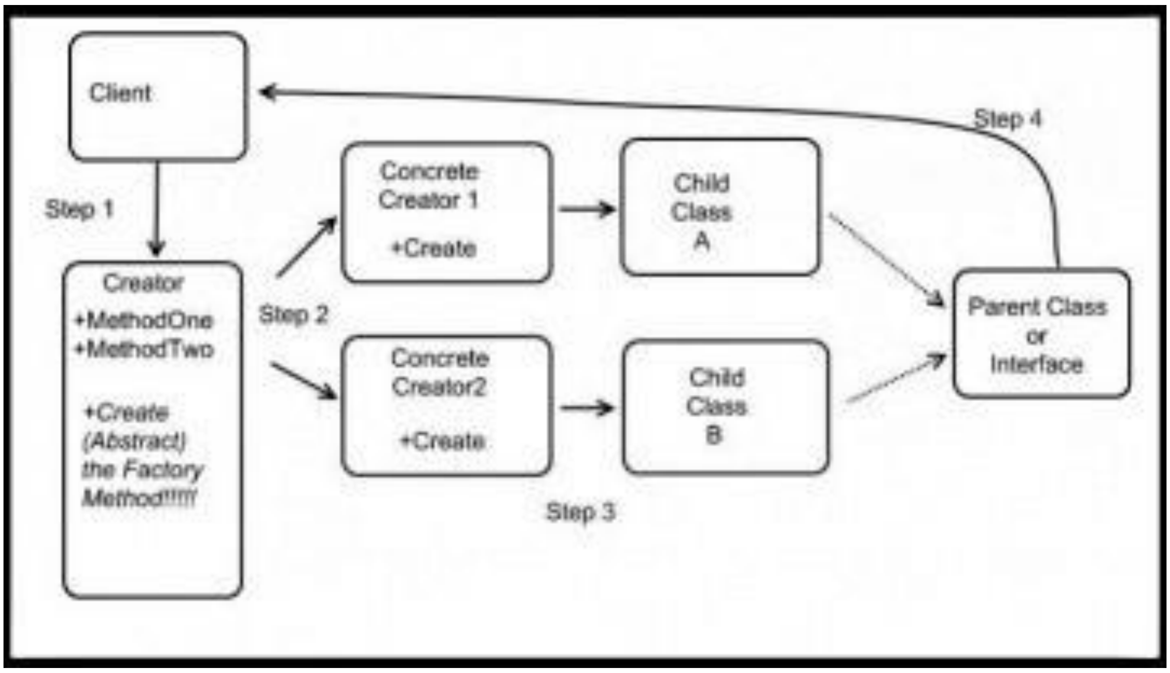
Step Three: Factory returns the object, and this is why step two makes sense. Since the client didn't know what was going to be returned, the client is expecting a type that matches the parent class /interface.

Factory Method

- The official definition of the pattern is something like: a class which defers instantiation of an object to subclasses. An important thing to note right away is that when we're discussing the factory pattern, we're not concentrating on the implementation of the factory in the client, but instead we're examining the manner in which objects are being created.
- In this example the client doesn't have a direct reference to the classes that are creating the object, but instead has reference to the abstract "Creator". (Just because the creator is abstract, doesn't mean this is the Abstract Factory!).
- It is a Factory Method because the children of "Creator" are responsible for implementing the "Create" method. Another key point is that the creator is returning only one object. The object could be one of several types, but the types all inherit from the same parent class.

- Factory Method DP defines standards form multiple factories that are creating object for same family classes.


Step 1: Load the required factory.
Step 2: Call the CreateMobile() method on that factory.
Client
1.LoadFactory("Name")
2. CreateMobile()
3. Return Nokia
IMobileFactory
CreateMobile()
NokiaFactory
CreateMobile()
Nokia
IphoneFactory
CreateMobile()
Iphone
IMobile
IMobileFactory factory=LoadFactory("NokiaFactory")
IMobile mob = factory. CreateMobile()



Step One: The client maintains a reference to the abstract Creator, but instantiates it with one of the subclasses. (i.e., Creator c = new ConcreteCreator1() ;).

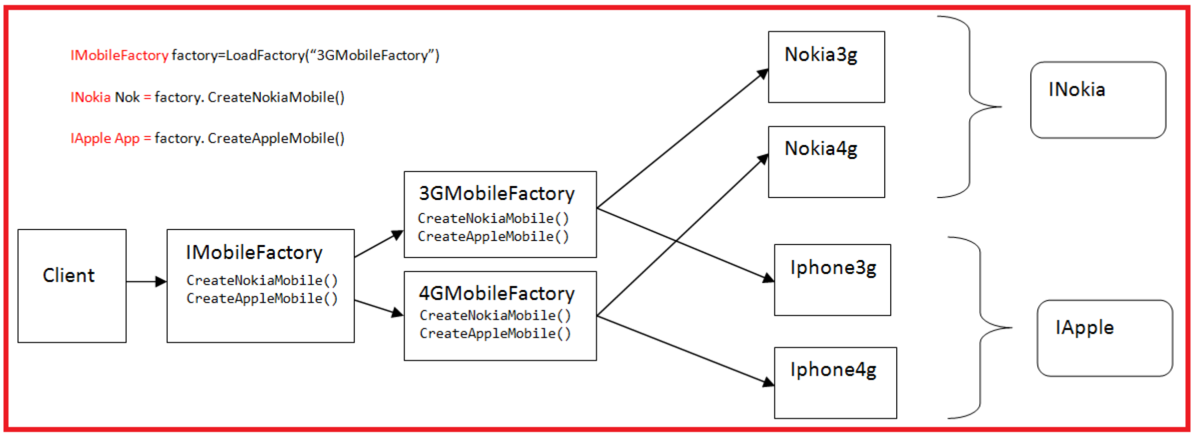
Step Two: The Creator has an abstract method for creation of an object, which we'll call "Create". It's an abstract method which all child classes must implement. This abstract method also stipulates that the type that will be returned is the Parent Class or the Interface of the "product".

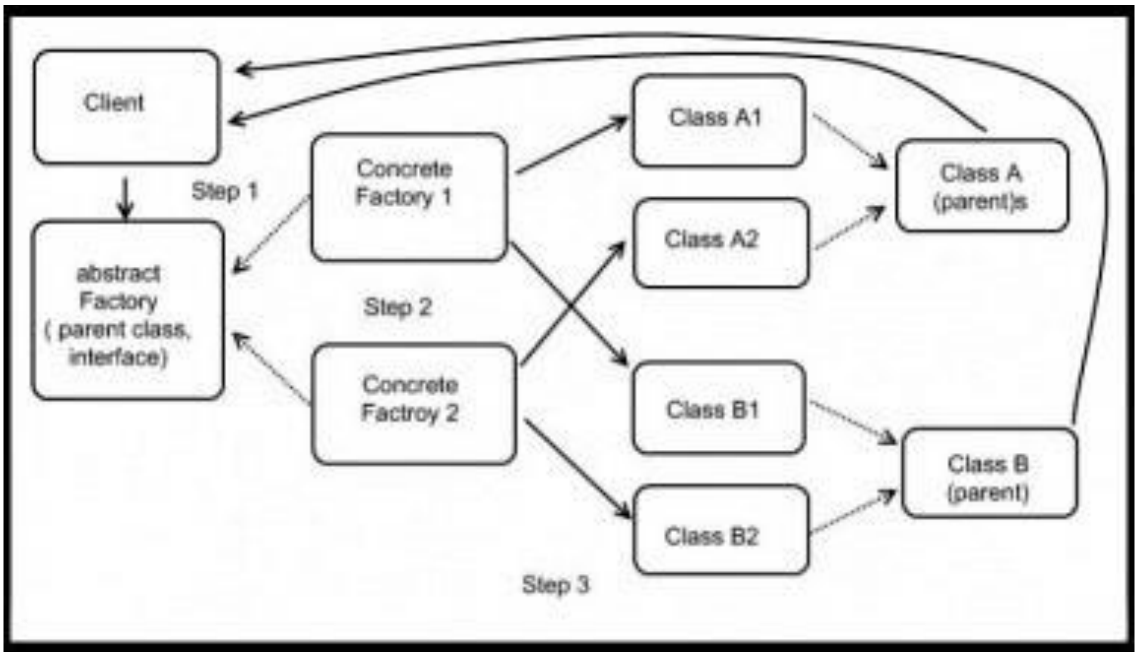
Step Three: the concrete creator creates the concrete object. In the case of Step One, this would be "Child Class A".

Step Four: the concrete object is returned to the client. Note that the client doesn't really know what the type of the object is, just that it is a child of the parent.

Abstract Factory

- This is biggest pattern of the three. I also find that it is difficult to distinguish this pattern from the Factory Method at a casual glance. For instance, in the Factory Method, didn't we use an abstract Creator? Wouldn't that mean that the Factory Method I showed was an actually an Abstract Factory? The big difference is that by its own definition, an Abstract Factory is used to create a family of related products (Factory Method creates one product).





Step One: The client maintains a reference to an abstract Factory class, which all Factories must implement. The abstract Factory is instantiated with a concrete factory.

Step Two: The factory is capable of producing multiple types. This is where the "family of related products" comes into play. The objects which can be created still have a parent class or interface that the client knows about, but the key point is there is more than one type of parent.

Step Three: The concrete factory creates the concrete objects.

Step Four: The concrete objects are returned to the client. Again, the client doesn't really know what the type of the objects are, just that are a child of the parents. See those concrete factories? Notice something vaguely familiar? They're using the Factory Method to create objects.

So, being as brief as I can:

- A Simple factory is normally caned by the client via a static method, and returns one of several objects that all inherit/ implement the same parent.
- The Factory method is really all about a “create” (object creation) method that is implemented by sub classes (multiple factories). It provides set of rules and guidelines to factories that are creating objects for same family classes.
- Abstract factory is about returning a family of related objects to the client suing same factory. It normally uses Super factory to create one factory class object and this factory class return other related classes objects.

Simple Factory DP: Provides abstraction on creating one of several related classes obj based on the data that is supplied.
Factory Method DP: Makes all factories to follows same standards while creating objects for same family classes.
Abstract Factory DP: Makes client app getting all related classes objects/ group of objects/ family objects coming from same factory.

Q. When did you use XXX factory patterns in your projects? (Simple Factory, Abstract Factory, Factory method DPs)
Ans. Realtime use-cases:

- I have MySQLCustomerDAO, OracleCustomeDAO, PostgreSQLDAO classes implementing same CustomeDAO (I) to return one DAO class object based on the DB s/w I, choose. I have implemented Factory DP.

- I have implemented Abstract factory pattern to make sure that StudentDAO, CourseDAO classes objects are coming from same ExcelDAOFactory or same OracleDAOFactory or MySQLDAOFactory.

Template Method Design Pattern

- It defines the skeleton of an algorithm in super class to complete a task/ job leaving some operations to implement for the sub classes.
- E.g.,

```
task1:  a ();
        b ();
        c ();
        d ();
```

(Here we should remember the method names and their order of invocation)

```
public void m1() {
        a ();
        b ();
        c ();                       task1: m1 ();
        d ();
}
```

- Template method is the method of super class having series of methods invoke to the complete a task. In those methods common operations/ logics related methods will be defined by super class itself but gives specific operations related methods to sub classes to override and to define logics as needed by sub classes.

  This algorithm for recruitment process.
  Fresher Recruitment Drive

```
        |--> Aptitude Test   |   Common tests for any fresher
        |--> GD Test         |
        |--> Technical Test  |   These tests will change domain to
        |--> Coding Test     |   domain (different for domain fresher)
        |--> HR Test         |   Common tests for any fresher
```

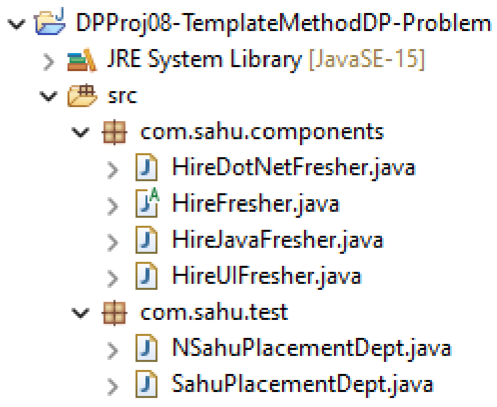
```
public abstract class HireFresher {        (abstract super class)
        public boolean aptitudeTest() {
                ……………………
                return true;
        }
```

```
            public boolean gdTest() {
                        ……………………
                        return ture;
            }
            public abstract boolean technicalTest();
            public abstract boolean codingTest();
            public boolean hrTest() {
                        ……………………
                        return true;
            }
            public boolean recruitFresher {
                        if (aptitudeTest())
                              if(gdTest())
                                   if(technicalTest())
                                         if(codingTest())
                                              if(hrTest())
                                                    return true;
                        return false;
            }
}

public class HireJavaFresher extends HireFresher {
            public boolean techinicalTest() {
                        //logic to conduct java technical Test
                        …………
                        return true;
            }
            public boolean codingTest() {
                        //logic to conduct java system test
                        …………
                        return true;
            }
}

public class DotNetFreshers extends HireFresher {
            public boolean techinicalTest(){
                        //logic to conduct .net technical Test
                        …………
                        return true;
            }
```

```
        public boolean codingTest() {
                //logic to conduct .net system test
                ………….
                return true;
        }
}
```

Problem: Here client has to take the entire pain, he has to remember the method, he should know how to call the method, he should remember the order of the method and he should know how to create the object.

Directory Structure of DPProj08-TemplateMethodDP-Problem:



```
DPProj08-TemplateMethodDP-Problem
  JRE System Library [JavaSE-15]
  src
    com.sahu.components
      HireDotNetFresher.java
      HireFresher.java
      HireJavaFresher.java
      HireUIFresher.java
    com.sahu.test
      NSahuPlacementDept.java
      SahuPlacementDept.java
```

- Develop the above directory structure package, class in src folder.
- Then use the following code with in their respective file.

HireFresher.java

```
package com.sahu.components;

public abstract class HireFresher {

        public boolean conductAptitudeTest() {
                System.out.println("Conducting aptitude test for fresher");
                return true;
        }

        public boolean conductGDTest() {
                System.out.println("Conducting GD test for fresher");
                return true;
        }

```

```
	public abstract boolean conductTechnicalTest();

	public abstract boolean conductCodingTest();

	public boolean conductHRTest() {
		System.out.println("Conducting HR test for fresher");
		return true;
	}

}
```

HireJavaFresher.java

```
package com.sahu.components;

public class HireJavaFresher extends HireFresher {

	@Override
	public boolean conductTechnicalTest() {
		System.out.println("HireJavaFresher conducting Java technical 
test");
		return false;
	}

	@Override
	public boolean conductCodingTest() {
		System.out.println("HireJavaFresher conducting Java coding 
test");
		return false;
	}

}
```

HireDotNetFresher.java

```
package com.sahu.components;

public class HireDotNetFresher extends HireFresher {

	@Override
	public boolean conductTechnicalTest() {
		System.out.println("HireDotNetFresher conducting .Net
technical test");
		return false;
	}

	@Override
	public boolean conductCodingTest() {
		System.out.println("HireDotNetFresher conducting .Net coding
test");
		return false;
	}

}
```

HireDotNetFresher.java

```
package com.sahu.components;

public class HireUIFresher extends HireFresher {

	@Override
	public boolean conductTechnicalTest() {
		System.out.println("HireUIFresher conducting UI technical
test");
		return false;
	}

	@Override
	public boolean conductCodingTest() {
		System.out.println("HireUIFresher conducting UI coding test");
		return false;
	}

}
```

SahuPlacementDept.java

```
package com.sahu.test;

import com.sahu.components.HireFresher;
import com.sahu.components.HireJavaFresher;
```

```
public class SahuPlacementDept {

	public static void main(String[] args) {
		HireFresher javaFresher = new HireJavaFresher();
		boolean result = javaFresher.conductAptitudeTest();
		if (result)
			result = javaFresher.conductGDTest();
		if (result)
			result = javaFresher.conductTechnicalTest();
		if (result)
			result = javaFresher.conductCodingTest();
		if (result)
			result = javaFresher.conductHRTest();
		if (result)
			System.out.println("Java fresher is selected");
		else
			System.out.println("Java fresher is not selected");
	}

}
```

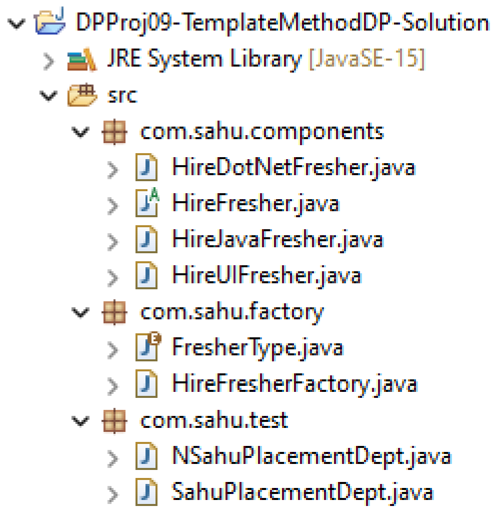
NSahuPlacementDept.java

```
package com.sahu.test;

import com.sahu.components.HireFresher;
import com.sahu.components.HireJavaFresher;

public class NSahuPlacementDept {

	public static void main(String[] args) {
		HireFresher uiFresher = new HireJavaFresher();
		boolean result = uiFresher.conductAptitudeTest();
		if (result)
			result = uiFresher.conductHRTest();
		if (result)
			result = uiFresher.conductGDTest();
		if (result)
			result = uiFresher.conductTechnicalTest();
		if (result)
```

```
                        System.out.println("Java fresher is selected");
                else
                        System.out.println("Java fresher is not selected");
        }

}
```

Solution: To overcome from this problem better to declare an algorithm in super class.

Directory Structure of DPProj09-TemplateMethodDP-Solution:



```
DPProj09-TemplateMethodDP-Solution
  JRE System Library [JavaSE-15]
  src
    com.sahu.components
      HireDotNetFresher.java
      HireFresher.java
      HireJavaFresher.java
      HireUIFresher.java
    com.sahu.factory
      FresherType.java
      HireFresherFactory.java
    com.sahu.test
      NSahuPlacementDept.java
      SahuPlacementDept.java
```

- Develop the above directory structure package, class in src folder.
- Then use the following code with in their respective file.
- Copy the rest of code from previous application.

HireFresher.java

```
package com.sahu.components;

public abstract class HireFresher {

        protected boolean conductAptitudeTest() {
                System.out.println("Conducting aptitude test for fresher");
                return true;
        }

        protected boolean conductGDTest() {
                System.out.println("Conducting GD test for fresher");
```

```
		return true;
	}

	protected abstract boolean conductTechnicalTest();

	protected abstract boolean conductCodingTest();

	protected boolean conductHRTest() {
		System.out.println("Conducting HR test for fresher");
		return true;
	}

	//Template method defining the skeleton of the algorithm
	public boolean recruitFresher() {
		System.out.println("HireFresher.recruitFresher()");
		if (conductAptitudeTest())
			if (conductGDTest())
				if (conductTechnicalTest())
					if (conductCodingTest())
						if (conductHRTest())
							return true;
		return false;
	}

}
```

FresherType.java

```
package com.sahu.factory;

public enum FresherType {
	JAVA, DOT_NET, UI;
}
```

HireFresherFactory.java

```
package com.sahu.factory;

import com.sahu.components.HireDotNetFresher;
import com.sahu.components.HireFresher;
import com.sahu.components.HireJavaFresher;
import com.sahu.components.HireUIFresher;

public class HireFresherFactory {

	public static HireFresher getInstance(FresherType fresherType) {
		HireFresher fresher = null;
		if (fresherType.equals(FresherType.JAVA)) {
			fresher =  new HireJavaFresher();
		}
		else if (fresherType.equals(FresherType.DOT_NET)) {
			fresher =  new HireDotNetFresher();
		}
		if (fresherType.equals(FresherType.UI)) {
			fresher =new HireUIFresher();
		}

		return fresher;
	}

}
```

NSahuPlacementDept.java

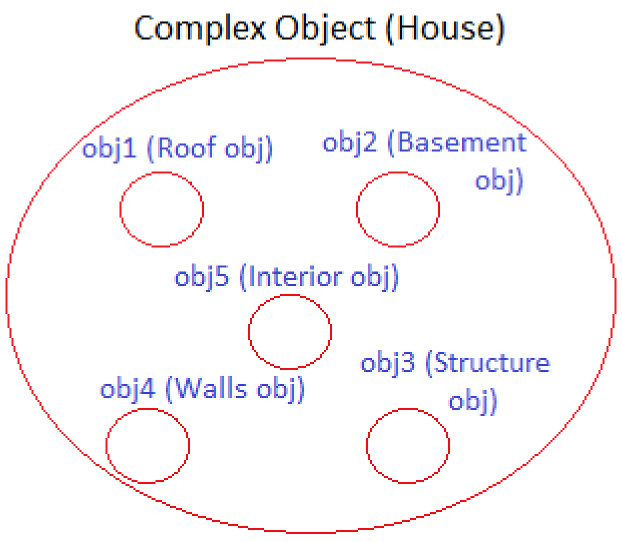
```
package com.sahu.test;

import com.sahu.components.HireFresher;
import com.sahu.factory.FresherType;
import com.sahu.factory.HireFresherFactory;

public class NSahuPlacementDept {

	public static void main(String[] args) {
		HireFresher javaFresher = 
HireFresherFactory.getInstance(FresherType.DOT_NET);
		boolean result = javaFresher.recruitFresher();
		if (result)
			System.out.println(".Net fresher is selected");
		else
			System.out.println(".Net fresher is selected");
	}

}
```

SahuPlacementDept.java

```
package com.sahu.test;

import com.sahu.components.HireFresher;
import com.sahu.factory.FresherType;
import com.sahu.factory.HireFresherFactory;

public class SahuPlacementDept {

	public static void main(String[] args) {
		HireFresher javaFresher =
HireFresherFactory.getInstance(FresherType.JAVA);
		boolean result = javaFresher.recruitFresher();
		if (result)
			System.out.println("Java fresher is selected");
		else
			System.out.println("Java fresher is selected");
	}

}
```

Builder Design Pattern

- It is a creational pattern that says, create complex object step by step by combining multiple independent objects by defining a process. So, that process can be used to construct multiple complex objects of same category.



If we define a process to construct the house by combining multiple individual objects using Builder pattern that process can be used to construct Wooden house, Ice house (Igloo), Concrete house and etc.

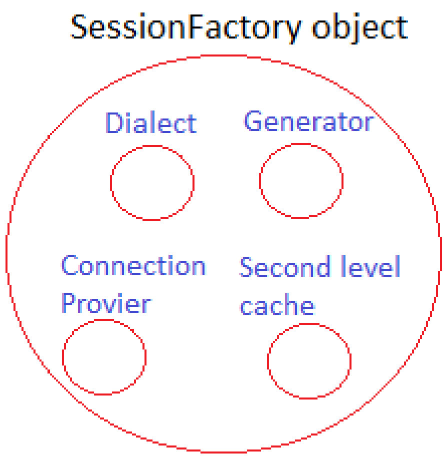
Indian Meal object: Roti +Rice + curries +Curd +salad
American Meal object: Bugger + French fries + Soft Drink

If we define a process to construct the Indian/ American meal object by combining multiple individual objects using Builder pattern. That process can be used to construct Veg Meal, Non-Veg meal, Kids Meal and etc.

4 participants of Builder Design Pattern

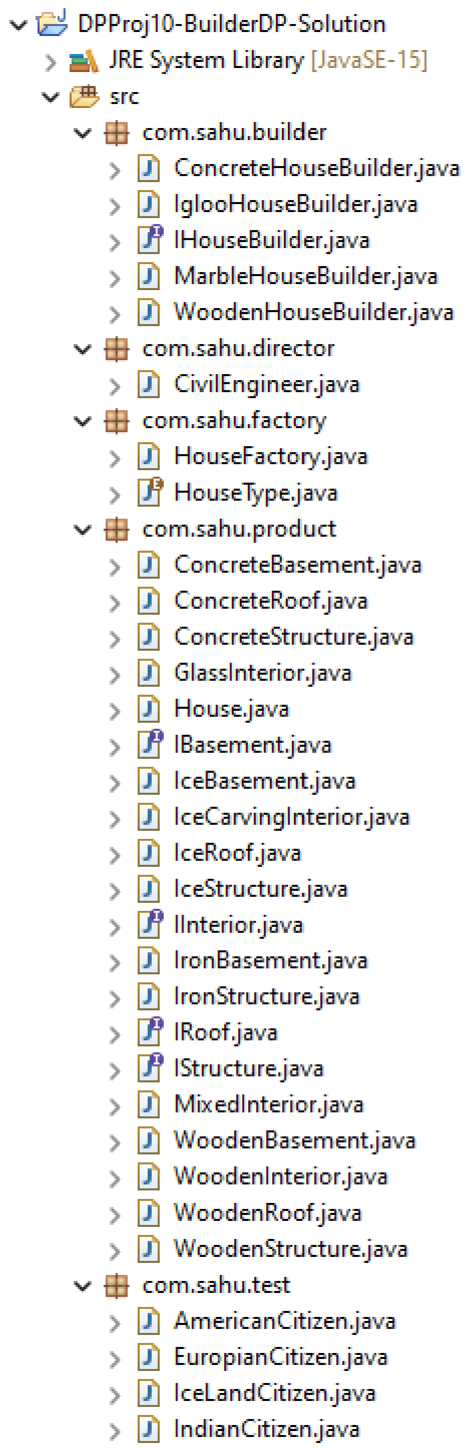
a. Product (The complex object to construct like House)
b. Builder (interface or abstract class that contains method declaration to constructor various simple objects like House plan)
c. Concrete Builder (Implements the Builder (I) and contains to create various simple/ independent objects that are required to create product (ConcreateHouseBuilder, IceHouseBuilder, WoodenHouseBuilder and etc.)
d. Director/ Delegator (Uses one or another Concrete Builder and defines the process to create the product, it is like CIVIL Engineer).

E.g., In Hibernate, that way we can configuration object to create SessionFactory by specifying various inputs like Connection factory, second level cache, Dialect, Generators, InMemory Metadata and etc. is called Builder Design Pattern.



House object needs
```
            |--> Basement object (ConcreateBasement, WoodenBasement,
                                   IceBasement and etc.)
            |--> Roof object (ConcreteRoof, WoodenRoof, IceRoof and etc.)
            |--> Structure object (ConcreteStructure, WoodenStructure,
                                   IceStructure and etc.)
            |--> Interiors object (WoodInteriors, MarbleInterior, GlassInterior and
                                   etc.)
```

Directory Structure of DPProj10-BuilderDP-Solution:



```
DPProj10-BuilderDP-Solution
    JRE System Library [JavaSE-15]
    src
        com.sahu.builder
            ConcreteHouseBuilder.java
            IglooHouseBuilder.java
            IHouseBuilder.java
            MarbleHouseBuilder.java
            WoodenHouseBuilder.java
        com.sahu.director
            CivilEngineer.java
        com.sahu.factory
            HouseFactory.java
            HouseType.java
        com.sahu.product
            ConcreteBasement.java
            ConcreteRoof.java
            ConcreteStructure.java
            GlassInterior.java
            House.java
            IBasement.java
            IceBasement.java
            IceCarvingInterior.java
            IceRoof.java
            IceStructure.java
            IInterior.java
            IronBasement.java
            IronStructure.java
            IRoof.java
            IStructure.java
            MixedInterior.java
            WoodenBasement.java
            WoodenInterior.java
            WoodenRoof.java
            WoodenStructure.java
        com.sahu.test
            AmericanCitizen.java
            EuropianCitizen.java
            IceLandCitizen.java
            IndianCitizen.java
```

- Develop the above directory structure package, class in src folder.
- Then use the following code with in their respective file.

IRoof.java

```
package com.sahu.product;
public interface IRoof {  }
```

ConcreteRoof.java

```java
package com.sahu.product;

public class ConcreteRoof implements IRoof {

	public ConcreteRoof() {
		System.out.println("ConcreteRoof.ConcreteRoof()");
	}

}
```

IceRoof.java

```java
package com.sahu.product;

public class IceRoof implements IRoof {

	public IceRoof() {
		System.out.println("IceRoof.IceRoof()");
	}

}
```

WoodenRoof.java

```java
package com.sahu.product;

public class WoodenRoof implements IRoof {

	public WoodenRoof() {
		System.out.println("WoodenRoof.WoodenRoof()");
	}

}
```

IBasement.java

```java
package com.sahu.product;

public interface IBasement {
}
```

ConcreteBasement.java

```
package com.sahu.product;

public class ConcreteBasement implements IBasement {

	public ConcreteBasement() {
		System.out.println("ConcreteBasement.ConcreteBasement()");
	}

}
```

IronBasement.java

```
package com.sahu.product;

public class IronBasement implements IBasement {

	public IronBasement() {
		System.out.println("IronBasement.IronBasement()");
	}

}
```

WoodenBasement.java

```
package com.sahu.product;

public class WoodenBasement implements IBasement {

	public WoodenBasement() {
		System.out.println("WoodenBasement.WoodenBasement()");
	}

}
```

IStructure.java

```
package com.sahu.product;

public interface IStructure {
}
```

ConcreteStructure.java

```
package com.sahu.product;

public class ConcreteStructure implements IStructure {

	public ConcreteStructure() {
		System.out.println("ConcreteStructure.ConcreteStructure()");
	}

}
```

IceStructure.java

```
package com.sahu.product;

public class IceStructure implements IStructure {

	public IceStructure() {
		System.out.println("IceStructure.IceStructure()");
	}

}
```

IronStructure.java

```
package com.sahu.product;

public class IronStructure implements IStructure {

	public IronStructure() {
		System.out.println("IronStructure.IronStructure()");
	}

}
```

IInterior.java

```
package com.sahu.product;

public interface IInterior {
}
```

GlassInterior.java

```java
package com.sahu.product;

public class GlassInterior implements IInterior {

	public GlassInterior() {
		System.out.println("GlassInterior.GlassInterior()");
	}

}
```

IceCarvingInterior.java

```java
package com.sahu.product;

public class IceCarvingInterior implements IInterior {

	public IceCarvingInterior() {
		System.out.println("IceCarvingInterior.IceCarvingInterior()");
	}

}
```

MixedInterior.java

```java
package com.sahu.product;

public class MixedInterior implements IInterior {

	public MixedInterior() {
		System.out.println("MixedInterior.MixedInterior()");
	}

}
```

WoodenInterior.java

```java
package com.sahu.product;

public class WoodenInterior implements IInterior {

	public WoodenInterior() {
		System.out.println("WoodenInterior.WoodenInterior()");
	}

}
```

WoodenStructure.java

```
package com.sahu.product;

public class WoodenStructure implements IStructure {

	public WoodenStructure() {
		System.out.println("WoodenStructure.WoodenStructure()");
	}

}
```

IceBasement.java

```
package com.sahu.product;

public class IceBasement implements IBasement {

	 public IceBasement() {
		System.out.println("IceBasement.IceBasement()");
	}

}
```

House.java

```
package com.sahu.product;

public class House {
	private IBasement iBasement;
	private IStructure iStructure;
	private IRoof iRoof;
	private IInterior iInterior;

	public void setBasement(IBasement iBasement) {
```

```
                this.iBasement = iBasement;
        }

        public void setStructure(IStructure iStructure) {
                this.iStructure = iStructure;
        }

        public void setRoof(IRoof iRoof) {
                this.iRoof = iRoof;
        }

        public void setInterior(IInterior iInterior) {
                this.iInterior = iInterior;
        }

        @Override
        public String toString() {
                return "House [basement=" + iBasement + ", structure=" +
iStructure + ", roof=" + iRoof + ", interior=" + iInterior
                                + ", getClass()=" + getClass() + ", hashCode()=" +
hashCode() + ", toString()=" + super.toString()
                                + "]";
        }

}
```

IHouseBuilder.java

```
package com.sahu.builder;

import com.sahu.product.House;

public interface IHouseBuilder {
        public void constructRoof();
        public void constructBasement();
        public void constructStructure();
        public void constructInterior();
        public House getHouse();
}
```

ConcreteHouseBuilder.java

```java
package com.sahu.builder;

import com.sahu.product.ConcreteBasement;
import com.sahu.product.ConcreteRoof;
import com.sahu.product.ConcreteStructure;
import com.sahu.product.GlassInterior;
import com.sahu.product.House;

public class ConcreteHouseBuilder implements IHouseBuilder {

	private House house;

	public ConcreteHouseBuilder(House house) {

	System.out.println("ConcreteHouseBuilder.ConcreteHouseBuilder()");
		this.house = house;
	}

	@Override
	public void constructRoof() {
		house.setRoof(new ConcreteRoof());
	}

	@Override
	public void constructBasement() {
		house.setBasement(new ConcreteBasement());
	}

	@Override
	public void constructStructure() {
		house.setStructure(new ConcreteStructure());
	}

	@Override
	public void constructInterior() {
		house.setInterior(new GlassInterior());
	}

	@Override
	public House getHouse() {
		return house;
	}

}
```

IglooHouseBuilder.java

```
package com.sahu.builder;

import com.sahu.product.House;
import com.sahu.product.IceBasement;
import com.sahu.product.IceCarvingInterior;
import com.sahu.product.IceRoof;
import com.sahu.product.IceStructure;

public class IglooHouseBuilder implements IHouseBuilder {

	private House house;

	public IglooHouseBuilder(House house) {
		System.out.println("IglooHouseBuilder.IglooHouseBuilder()");
		this.house = house;
	}

	@Override
	public void constructRoof() {
		house.setRoof(new IceRoof());
	}

	@Override
	public void constructBasement() {
		house.setBasement(new IceBasement());
	}

	@Override
	public void constructStructure() {
		house.setStructure(new IceStructure());
	}

	@Override
	public void constructInterior() {
		house.setInterior(new IceCarvingInterior());
	}

	@Override
	public House getHouse() {
		return house;
	}

}
```

MarbleHouseBuilder.java

```
package com.sahu.builder;

import com.sahu.product.ConcreteBasement;
import com.sahu.product.ConcreteRoof;
import com.sahu.product.House;
import com.sahu.product.IronStructure;
import com.sahu.product.MixedInterior;

public class MarbleHouseBuilder implements IHouseBuilder {

	private House house;

	public MarbleHouseBuilder(House house) {

	System.out.println("MarbleHouseBuilder.MarbleHouseBuilder()");
		this.house = house;
	}

	@Override
	public void constructRoof() {
		house.setRoof(new ConcreteRoof());
	}

	@Override
	public void constructBasement() {
		house.setBasement(new ConcreteBasement());
	}
	@Override
	public void constructStructure() {
		house.setStructure(new IronStructure());
	}

	@Override
	public void constructInterior() {
		house.setInterior(new MixedInterior());
	}

	@Override
	public House getHouse() {
		return house;
	}

}
```

WoodenHouseBuilder.java

```
package com.sahu.builder;

import com.sahu.product.House;
import com.sahu.product.WoodenBasement;
import com.sahu.product.WoodenInterior;
import com.sahu.product.WoodenRoof;
import com.sahu.product.WoodenStructure;

public class WoodenHouseBuilder implements IHouseBuilder {

	private House house;

	public WoodenHouseBuilder(House house) {

	System.out.println("WoodenHouseBuilder.WoodenHouseBuilder()");
		this.house = house;
	}

	@Override
	public void constructRoof() {
		house.setRoof(new WoodenRoof());
	}
```

```
        @Override
        public void constructBasement() {
                house.setBasement(new WoodenBasement());
        }

        @Override
        public void constructStructure() {
                house.setStructure(new WoodenStructure());
        }

        @Override
        public void constructInterior() {
                house.setInterior(new WoodenInterior());
        }

        @Override
        public House getHouse() {
                return house;
        }

}
```

CivilEngineer.java

```
package com.sahu.director;

import com.sahu.builder.IHouseBuilder;
import com.sahu.product.House;

public class CivilEngineer {

        private IHouseBuilder builder;

        public CivilEngineer(IHouseBuilder builder) {
                this.builder = builder;
        }

        //Builder design pattern main method, Defining the process to
construct complex object
        public House buildHouse() {
                builder.constructBasement();
                builder.constructStructure();
                builder.constructRoof();
                builder.constructInterior();

                return builder.getHouse();
        }

}
```

HouseType.java

```
package com.sahu.factory;

public enum HouseType {
        WOODEN, ICE, CONCRETE, MARBLE;
}
```

HouseFactory.java

```
package com.sahu.factory;

import com.sahu.builder.ConcreteHouseBuilder;
import com.sahu.builder.IHouseBuilder;
import com.sahu.builder.IglooHouseBuilder;
import com.sahu.builder.MarbleHouseBuilder;
import com.sahu.builder.WoodenHouseBuilder;
import com.sahu.director.CivilEngineer;
import com.sahu.product.House;

public class HouseFactory {

        public static House createHouse(HouseType type) {
                IHouseBuilder builder = null;
                House house = new House();
                if (type.equals(HouseType.WOODEN))
                        builder = new WoodenHouseBuilder(house);
                else if (type.equals(HouseType.ICE))
                        builder = new IglooHouseBuilder(house);
                else if (type.equals(HouseType.CONCRETE))
                        builder = new ConcreteHouseBuilder(house);
                else if (type.equals(HouseType.MARBLE))
```

```
				builder = new MarbleHouseBuilder(house);

			//Create Director object
			CivilEngineer engineer = new CivilEngineer(builder);
			house = engineer.buildHouse();
			return house;
		}

}
```

AmericanCitizen.java

```
package com.sahu.test;

import com.sahu.factory.HouseFactory;
import com.sahu.factory.HouseType;
import com.sahu.product.House;

public class AmericanCitizen {
		public static void main(String[] args) {
			House house =
HouseFactory.createHouse(HouseType.WOODEN);
			System.out.println(house);
		}
}
```

EuropianCitizen.java

```
package com.sahu.test;

import com.sahu.factory.HouseFactory;
import com.sahu.factory.HouseType;
import com.sahu.product.House;

public class EuropianCitizen {
		public static void main(String[] args) {
			House house =
HouseFactory.createHouse(HouseType.MARBLE);
			System.out.println(house);
		}
}
```

IceLandCitizen.java

```
package com.sahu.test;

import com.sahu.factory.HouseFactory;
import com.sahu.factory.HouseType;
import com.sahu.product.House;

public class IceLandCitizen {

	public static void main(String[] args) {
		House house = HouseFactory.createHouse(HouseType.ICE);
		System.out.println(house);
	}

}
```

IndianCitizen.java

```
package com.sahu.test;

import com.sahu.factory.HouseFactory;
import com.sahu.factory.HouseType;
import com.sahu.product.House;

public class IndianCitizen {

	public static void main(String[] args) {
		House house =
HouseFactory.createHouse(HouseType.CONCRETE);
		System.out.println(house);
	}

}
```

Predefined Builder DP Implementation

```
Configuration cfg =new Configration();
cfg.configure(".com/sahu/cfgs/hibernate.cfg.xml");

SessionFactory factory=cfg.buildSessionFactory();
                                    Builder Design pattern
```

Builder Design Pattern (recognizable by creational methods returning the instance itself (Core Java level API methods)

- java.lang.StringBuilder#append() (unsynchronized)
- java.lang.StringBuffer#append() (synchronized)
- java.nio.ByteBuffer#put() (also on CharBuffer, ShortBuffer, IntBuffer, LongBuffer, FloatBuffer and DoubleBuffer)
- javax.swing.GroupLayout.Group#addComponent()
- All implementations of java.lang.Appendable
- java.util.stream.Stream.Builder

Singleton Design Pattern (recognizable by creational methods returning the Same instance (usually of itself) every time)

- java.lang.Runtime#getRuntime()
- java.awt.Desktop#getDesktop()
- java.lang.System#getSeurityManager()

Factory Design Pattern (recognizable by creational methods returning an implementation of an abstract/ interface type)

- java.util.Calendar#getInstance()
- java.util.ResourceBund1e#getBundle()
- java.text.NumberFormat#getInstance()
- java.nio.charset.Charset#forName()
- java.net.URLStreamHandlerFactory#createURLStreamHandler(String) (Returns singleton object per protocol)
- java.util.EnumSet#of()
- javax.xml.bind.JAXBContext#createMarshaller() and other similar methods
- java.sql.DriverManager#getConnection(String url,string user,String pws)
- IoC Container supplied getBean(-) method

Abstract Factory Design Pattern (recognizable by creational methods returning the factory itself which in turn can be used to create another abstract/ interface type)

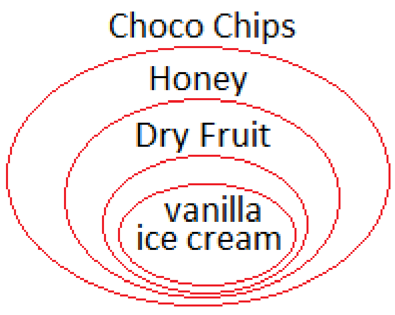
- javax.xml.parsers.DocumentBuilderFactory#newInstance()
- javax.xml.transform.TransformerFactory#newInstance()
- javax.xml.xpath.XPathFactory#newInstane()

Template method Design Pattern (recognizable by behavioral methods which already have a "default" behaviour defined by an abstract type)

- All non-abstract methods of java.io.InputStream, java.io.OutputStream, java.io.Reader and 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.
- Non abstract methods in JdbcTemplate, NamedPameterJdbcTemplate, JndiTemplate, MongoTemplate and etc.

Decorator/ Wrapper Design Pattern

- It is a structural pattern, that adds new responsibilities or functionalities to existing object with out effecting the other objects. i.e., it adds dynamic responsibilities or functionalities to existing object as the wrapper around the existing object, so it is called Decorator or Wrapper design pattern.



Here Choco chips, Honey, Dry Fruits and etc. are the wrappers around the vanilla Ice-cream object.

Problem: Adding new functionalities/ responsibilities on the existing object through inheritance is bad practice. we will end up getting huge number of classes as shown below.
E.g.,

Icecream.java
```
public interface Icecream {
          public void prepare();
}
```

VanillaIcecream.java
```
public class VanillaIcecream implements Icecream {
          public void prepare() {
                    S.o.p("Preparing Vanilla Ice-cream");
          }
}
```

ButterscotchIcecream.java

```
public class ButterscotchIcecream implements Icecream {
		public void prepare() {
				S.o.p("Preparing Butter Ice-cream");
		}
}
```

StrawberryIcecream.java

```
public class StrawberryIcecream implements Icecream {
		public void prepare() {
				S.o.p("Preparing Straberry Ice-ream");
		}
}
```

DryFruitVanillaIcecream.java

```
public class DryFruitVanillaIcecream extends VanillaIcecream {
		public void prepare() {
				super.prepare();
				addDryFruits();
		}

		public void addDryFruits(){
				S.o.p("Adding Dry fruits");
		}
}
```

HoneyVanillaIcecream.java

```
public class HoneyVanillaIcecream extends VanillaIcecream {
		public void prepare() {
				super.prepare();
				addHoney();
		}

		public void addHoney(){
				S.o.p("Adding Honey");
		}
}
```

ChocochipsVanillaIcecream.java

```
public class ChocochipsVanillaIcecream extends VanillaIcecream {
```

```
            public void prepare() {
                        super.prepare();
                        addChocochips();
            }

            public void addChocochips(){
                        S.o.p("Adding Choco chips");
            }
}
```

HoneyDryFruitVanillaIcecrem.java

```
pubic class HoneyDryFruitVanillaIcecrem extends DryFruitVanillaIcecream {
            public void prepare() {
                        supr.prepare();
                        addHoney();
            }

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

3 basic Flavors (Vanilla, Chocolate, Butterscotch)
3 Toppings (Dry fruit, Honey, Choco chips)

6! combinations are possible = 6*5*4*3*2*1 = 720 classes
(We are really not ready to develop 720 classes to support various permutation and combinations.)

Solution: To overcome the problem of developing huge number of classes go for Decorator/ Wrapper design pattern which builds the solutions using Composition and bit of inheritance.

```
class A {           class B extends A {       (Inheritance IS-A relationship)
      ……….                 ……………
}                   }

class A {           class B {
    ………….              A a = new A ();       (Composition: HAS-A relationship)
                         ……….
}                   }
```

The components of Decorator/ Wrapper design pattern

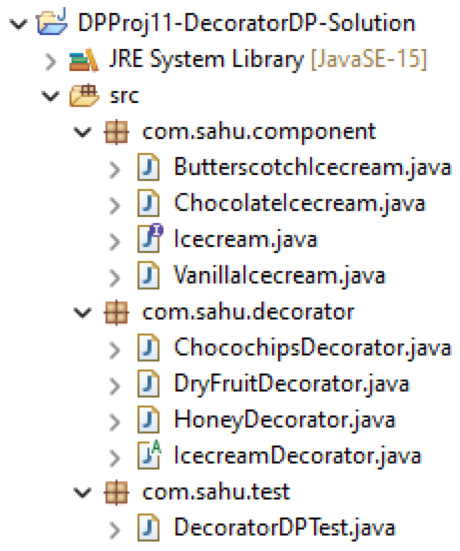
a. Component: It is the basic interface that defines object type (like Icecream (I))

b. Concrete Component: Implements component (I) and defines the basic component flavors/ root objects (like ButterscotchIcecream, VanillaIcecream, ChocolateIcecream, StrawberryIcecream and etc.)

c. Abstract Decorator: Abstract class implementing component (I) and also containing component (I) type property (It is the base class for all decorator/ wrapper classes that are there to add external functionalities or responsibilities to the object) It is like IcecreamDecorator (AC)).

d. Concrete Decorator: Extends from Abstract Decorator defining additional functionalities or responsibilities on the top of Concrete Components (these are like HoneyDecorator, DryFruitDecorator, ChocochipsDecorator and etc.).

Directory Structure of DPProj11-DecoratorDP-Solution:


```
DPProj11-DecoratorDP-Solution
  JRE System Library [JavaSE-15]
  src
    com.sahu.component
      ButterscotchIcecream.java
      ChocolateIcecream.java
      Icecream.java
      VanillaIcecream.java
    com.sahu.decorator
      ChocochipsDecorator.java
      DryFruitDecorator.java
      HoneyDecorator.java
      IcecreamDecorator.java
    com.sahu.test
      DecoratorDPTest.java
```

- Develop the above directory structure package, class in src folder.
- Then use the following code with in their respective file.

Icecream.java

```
package com.sahu.component;
public interface Icecream {
	public void prepare();
}
```

ButterscotchIcecream.java

```
package com.sahu.component;

public class ButterscotchIcecream implements Icecream {

	@Override
	public void prepare() {
		System.out.println("Prepare Butterscotch Icecream");
	}

}
```

ChocolateIcecream.java

```
package com.sahu.component;

public class ChocolateIcecream implements Icecream {

	@Override
	public void prepare() {
		System.out.println("Prepare Chocolate Icecream");
	}

}
```

VanillaIcecream.java

```
package com.sahu.component;

public class VanillaIcecream implements Icecream {

	@Override
	public void prepare() {
		System.out.println("Prepare Vanilla Icecream");
	}

}
```

IcecreamDecorator.java

```
package com.sahu.decorator;
```

```java
import com.sahu.component.Icecream;

public abstract class IcecreamDecorator implements Icecream {

	private Icecream icecream;	//Composition

	public IcecreamDecorator(Icecream icecream) {
		this.icecream = icecream;
	}

	@Override
	public void prepare() {
		icecream.prepare();
	}

}
```

ChocochipsDecorator.java

```java
package com.sahu.decorator;

import com.sahu.component.Icecream;

public class ChocochipsDecorator extends IcecreamDecorator {

	public ChocochipsDecorator(Icecream icecream) {
		super(icecream);
	}

	@Override
	public void prepare() {
		super.prepare();
		addChocochips();
	}

	public void addChocochips() {
		System.out.println("Adding Chocochips");
	}

}
```

DryFruitDecorator.java

```java
package com.sahu.decorator;

import com.sahu.component.Icecream;

public class DryFruitDecorator extends IcecreamDecorator {

	public DryFruitDecorator(Icecream icecream) {
		super(icecream);
	}

	@Override
	public void prepare() {
		super.prepare();
		addDryFruits();
	}

	public void addDryFruits() {
		System.out.println("Adding Dry Fruits");
	}

}
```

HoneyDecorator.java

```java
package com.sahu.decorator;

import com.sahu.component.Icecream;

public class HoneyDecorator extends IcecreamDecorator {

	public HoneyDecorator(Icecream icecream) {
		super(icecream);
	}

	@Override
	public void prepare() {
		super.prepare();
		addHoney();
	}
```

```
        public void addHoney() {
                System.out.println("Adding Honey");
        }

}
```

DecoratorDPTest.java

```
package com.sahu.test;

import com.sahu.component.ButterscotchIcecream;
import com.sahu.component.Icecream;
import com.sahu.component.VanillaIcecream;
import com.sahu.decorator.ChocochipsDecorator;
import com.sahu.decorator.DryFruitDecorator;
import com.sahu.decorator.HoneyDecorator;

public class DecoratorDPTest {

        public static void main(String[] args) {
                Icecream icecream1 = new VanillaIcecream();
                icecream1.prepare();
                System.out.println("----------------------");
                Icecream icecream2 = new DryFruitDecorator(new
VanillaIcecream());
                icecream2.prepare();
                System.out.println("----------------------");
                Icecream icecream3 = new HoneyDecorator(new
DryFruitDecorator(new VanillaIcecream()));
                icecream3.prepare();
                System.out.println("----------------------");
                Icecream icecream4 = new ChocochipsDecorator(new
HoneyDecorator(new DryFruitDecorator(new VanillaIcecream())));
                icecream4.prepare();
                System.out.println("----------------------");
                Icecream icecream5 = new ChocochipsDecorator(new
HoneyDecorator(new DryFruitDecorator(new ButterscotchIcecream())));
                icecream5.prepare();
        }

}
```

Note: Here by just taking 8 resources (3 concreate components classes and 3 concreate decorator classes and 1 component Interface and abstractor decorator class) we can bring the effect of same 720 classes of inheritance.

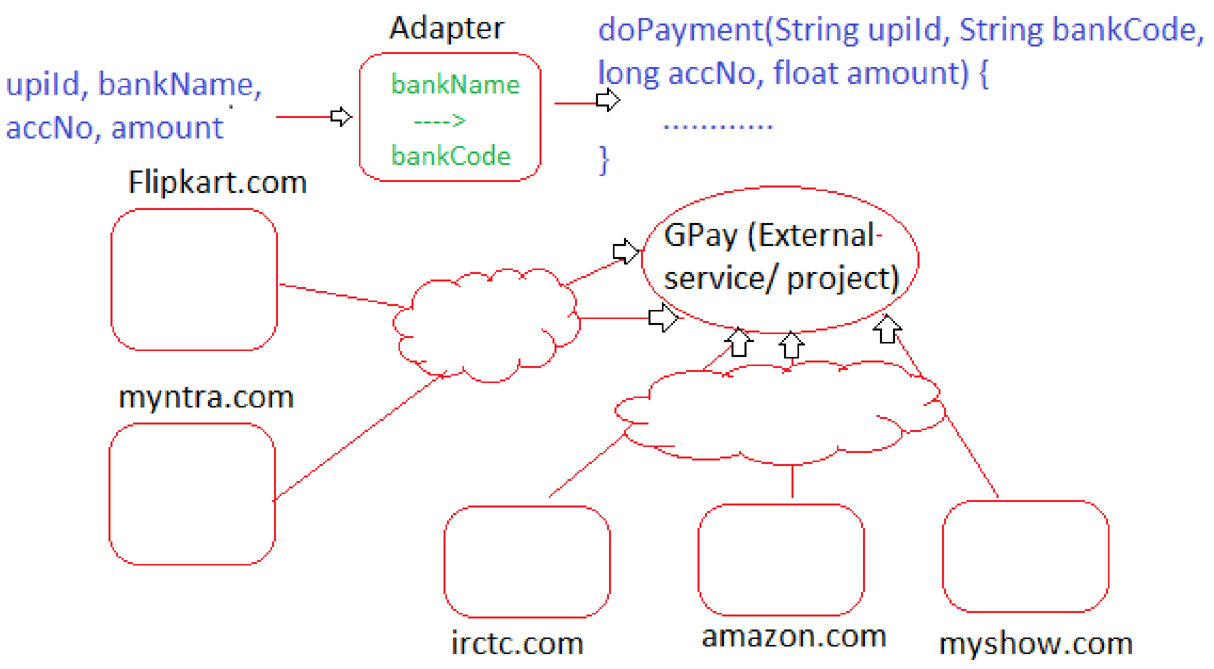
```
BufferedReader br = new BufferedReader(
                                    new InputStreamReader(System.in));
DataInputSrtream dis = new DataInputStream(
                                    new FileInputStream("e:/abc. txt"));
Scanner sc= new Scanner (System. in);
ObjectInputStream ios = new ObjectInputStream(
                                    new FileInputStream("abc. txt"));
```

Note:

- ✓ Java IO streams is designed based on Decorator/ Wrapper design patten.
- ✓ In that InputStream/ OutputStream/ Reader/ Writer are component interfaces.
- ✓ FileInputStream/ FileOutputStream/ FileReader/ FileWriter are the concrete component classes.
- ✓ BufferedReader, BufferedWriter, DataInputStream, DataOutputStream and etc. are the concreate decorator classes.

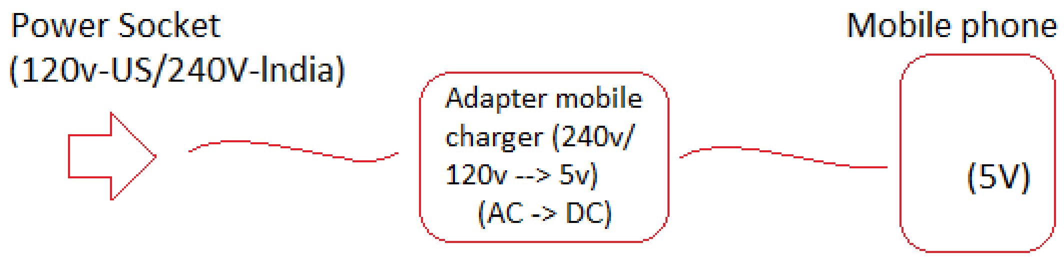
Adapter Design Pattern

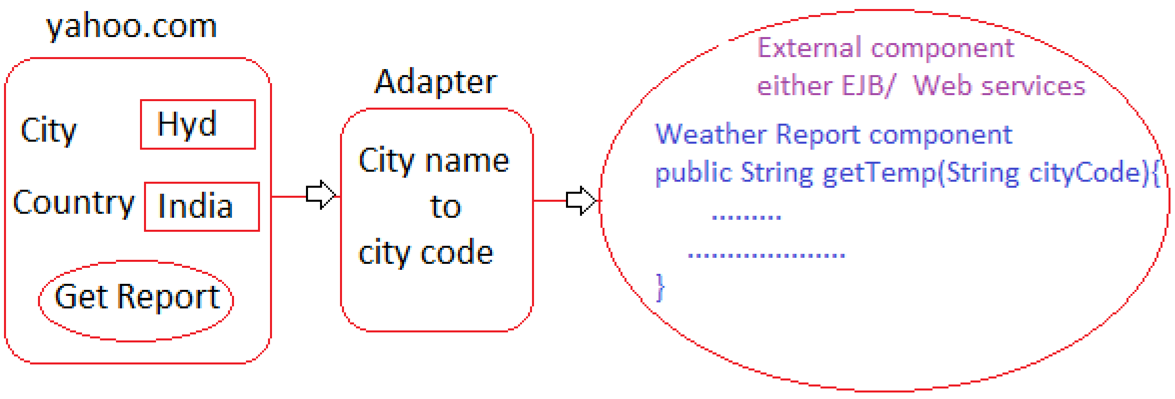
- The project/ services that are external to all the projects and whose services can be used in multiple projects are called external services or external projects.


Adapter
upiId, bankName, accNo, amount
bankName ----> bankCode
doPayment(String upiId, String bankCode, long accNo, float amount) {
....
}
Flipkart.com
myntra.com
GPay (External-service/ project)
irctc.com
amazon.com
myshow.com

- GPay, Paytm, Weather report component, BSE/ NSE share component, ICC score component (all these are called distributed services or external services or external projects because all these services will be used multiple local and remote applications).
- Adapter design pattern makes two unrelated interfaces/ components talking to each other.

Note: While making our local projects taking with external projects there is possibility of having incompatibility. In order to solve that incompatibility, we need to develop one Adapter class (Adapter design pattern).





Adapter class contains the following logics:

a. Converting client project supplied inputs as required for external project inputs.
b. Getting external component reference either directly or using ServiceLocator.
c. Calling business methods on external component reference and getting the results.
d. Converting external component supplied results as required for the Local project (if necessary). E.g., converting temperature from Fahrenheit degrees (◦F) to Celsius degrees (°C).

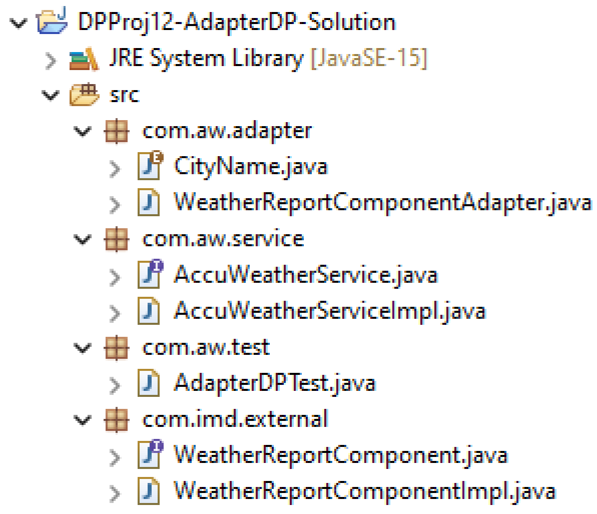
Note:

- ✓ Java Adapter class is not way related to Adapter design pattern.
- ✓ In Java Adapter class is a class that implements interface and provide

null method definitions. So instead of implementing interface we can make our class extending from adapter class and we can override only those methods in which we are interested in.

✓ Adapter design pattern is all about acting as bride/ mediator having logic for converting inputs and outputs, in the process of interacting with external component.

Directory Structure of DPProj12-AdapterDP-Solution:



```
DPProj12-AdapterDP-Solution
  JRE System Library [JavaSE-15]
  src
    com.aw.adapter
      CityName.java
      WeatherReportComponentAdapter.java
    com.aw.service
      AccuWeatherService.java
      AccuWeatherServiceImpl.java
    com.aw.test
      AdapterDPTest.java
    com.imd.external
      WeatherReportComponent.java
      WeatherReportComponentImpl.java
```

- Develop the above directory structure package, class in src folder.
- Then use the following code with in their respective file.

WeatherReportComponent.java

```java
package com.imd.external;

public interface WeatherReportComponent {
	public double getTemperature(Integer cityCode);
}
```

WeatherReportComponentImpl.java

```java
package com.imd.external;

public class WeatherReportComponentImpl implements
WeatherReportComponent{

	private static WeatherReportComponent INSTANCE;

	private WeatherReportComponentImpl() { }
```

```
        public static WeatherReportComponent getInstance() {
                if (INSTANCE==null)
                        INSTANCE = new WeatherReportComponentImpl();

                return INSTANCE;
        }

        @Override
        public double getTemperature(Integer cityCode) {
                //Get temperature from DB s/w or from IOT device
                double temp = 0.0;
                if (cityCode==100)
                        temp = 100 - Math.random();
                else if (cityCode==101)
                        temp = 101 - Math.random();
                else if (cityCode==102)
                        temp = 102 - Math.random();
                else if (cityCode==103)
                        temp = 130 - Math.random();

                return temp;
        }

}
```

CityName.java

```
package com.aw.adapter;

public enum CityName {
        HYD, DELHI, MUMBAI, CHENNAI;
}
```

WeatherReportComponentAdapter.java

```
package com.aw.adapter;

import com.imd.external.WeatherReportComponent;
import com.imd.external.WeatherReportComponentImpl;
```

```
        public double fetechTemperature(CityName cityName) {
                //Get city ode based on the city name (we can get by taking
from DB s/w or Web services component)
                int cityCode = 0;
                if (cityName.equals(CityName.HYD))
                        cityCode = 100;
```

AccuWeatherService.java

```
package com.aw.service;

import com.aw.adapter.CityName;

public interface AccuWeatherService {
	public String showTemperature(CityName cityName);
}
```

AccuWeatherServiceImpl.java

```
package com.aw.service;
```

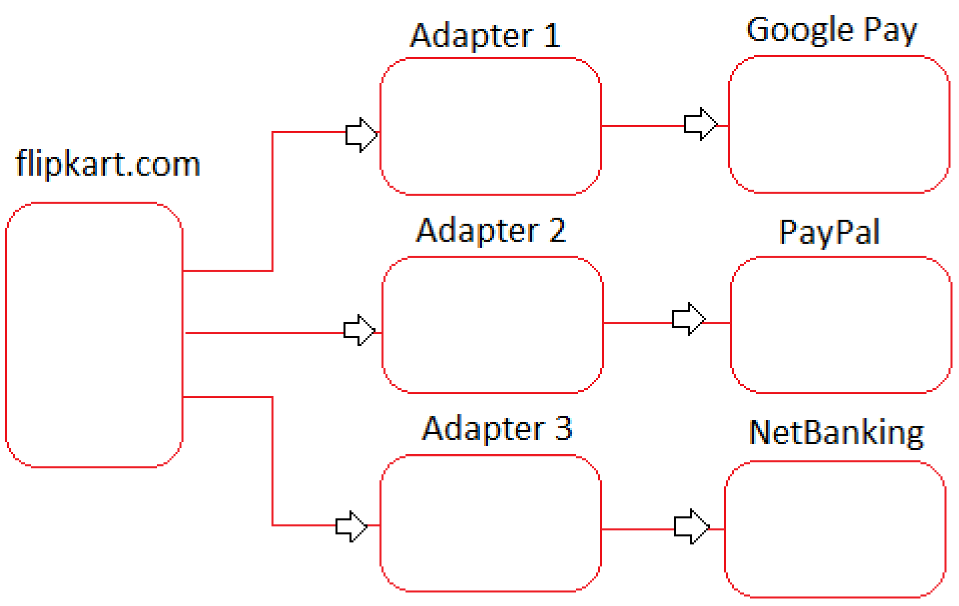
```
import java.text.DecimalFormat;

import com.aw.adapter.CityName;
import com.aw.adapter.WeatherReportComponentAdapter;

public class AccuWeatherServiceImpl implements AccuWeatherService {
```

AdapterDPTest.java

```
package com.aw.test;

import com.aw.adapter.CityName;
import com.aw.service.AccuWeatherService;
import com.aw.service.AccuWeatherServiceImpl;

public class AdapterDPTest {

	public static void main(String[] args) {
		//Create service class object
		AccuWeatherService service = new AccuWeatherServiceImpl();
		//invoke method
		System.out.println(service.showTemperature(CityName.HYD));

	System.out.println(service.showTemperature(CityName.DELHI));

	System.out.println(service.showTemperature(CityName.MUMBAI));

	System.out.println(service.showTemperature(CityName.CHENNAI));
	}

}
```

Note:

- ✓ Since we are developing external service component class as ordinary java class that to, we found no state in that class. So, we have taken that external Singleton java class.
- ✓ If those class/ logics are developed as real EJB component or webservice component then we need to worry single object creation, the underlying EJB container or Webservice f/w implementation will take care of that process.
- ✓ Always make External component/ Distributed component is having one interface and one implementation class. So, that we can pass/ supply interface to client projects/ local projects to make to receive and hold external component references.
- ✓ Every local project will take Adapter classes on 1 per external component basics.



```
DataInputStream dis = new DataInputStream(
                          new FileInputStream(System.in));
```

⇩

(All byte streams are interlinked)

(CharacterStream)

⇧

```
BuffereredReader reader = new BufferedReader(
                       new InputStreamReader(System.in));
```

⇩ Adapter converting byte stream to char stream

⇩ (Byte Stream)

Adapter Design Patten (recognizable by creational method taking an instance of different abstract/ interface type and returning an implementation of own/ anther interface type with decode)

- 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()
- The Streaming API stream () method

Q. What is the difference factory method and factory pattern?

Ans.

- Factory method creates and returns either its own class object, related class obj, unrelated class object but always returns same class object. Factory method can be static or non-static method.
  E.g.,
  ```
    Class c = Class.forName("java.util.Date"); //static factory method
    Calendar cal = Calendar.getInstance(); // static factory method

    String s = "hello";
    String s1 = s.concat("123"); //gives he110123
                                            //non-static factory method
    StringBuffer sb = new StringBuffer("hello");
    String s2 = sb.substring(0,3); //gives hell // non-static factory method
  ```
  (All these above methods are giving same class object always irrespective data that we are passing or not passing.)

- Factory pattern creates and returns one of several related classes object based on data that is passed, in this process it provides abstraction on object creation process. Factory pattern internally uses either static or non-static factory method but it returns one of the several related classes object based on data is passed.
  E.g.,
  ```
    Car car1 = CarFactory.getCar("swifit");
    Car car2 = CarFactory.getCar("baleno"); //static factory method

    Connection con =                        //static factory method
          DriverManager.getConnection(url,username,password);
  ```

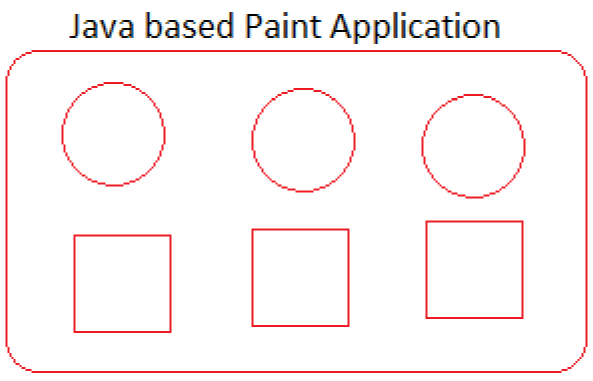
```
Statement st = con.createStatement();   //non-static factory method
```

(These are also factory patterns because they creating and returning one of the several related classes obj based on the data is supplied.)

Note: Every Factory pattern uses factory method but we can't say every factory method is factory pattern. Here we are not at all taking about factory method design pattern we are just talking about factory method of java.

Flyweight Design Pattern

- Flyweight: removing/ reducing /decreasing the weight.
- This design pattern says use JVM memory of RAM effectively while creating huge number of objects. So, that less memory can be used to maintain more objects.
- IOS mobile RAM size is small but performance of iPhone is much better compared to Android mobile with huge RAM size because the algorithms are there for Memory Management.
- According to GOF, flyweight design pattern intent is: "Use sharing to support large numbers of fine-grained objects efficiently".



Normal application
3 circles: 3 objects for circle class
3 squares: 3 objects for square class

Flyweight DP application
3 circles: 1 object for circles class and keep that object in cache and use it for 3 times by passing radius as the argument value to draw(-).
3 squares: 1 object for square class and keep that object in cache and use it for 3 times by passing side length as the argument value to draw(-).

Use the Flyweight design pattern, in the following situation:

- The number of objects to be created by application should be huge.
- The object creation is heavy on memory and it can be time consuming too.
- The object properties can be divided into intrinsic and extrinsic

1properties.

- The properties that declared inside the class as member variables and common for all objects created for the are called Intrinsic properties/ data (sharable properties/ data).
- The data/ properties that is coming to object from user/ client app as java method argument value because it is not common for all objects of that class is called extrinsic data/ properties (non-sharable properties/ data).

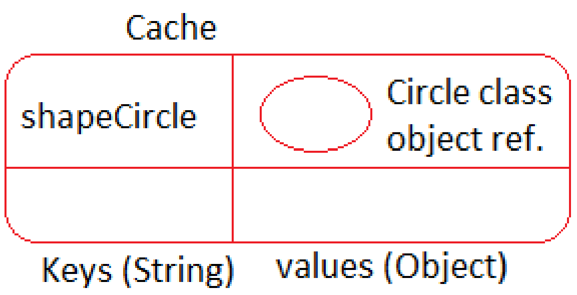
Circle class

- label: circle (intrinsic property) because all objects circle class can have same label "circle".
- radius: 30/ 50/ 80 (extrinsic property) because every time the radius of circle drawing will be changed by end-user.
  - fillColor (extrinsic)
  - lineStyle (extrinsic)
  - lineColor (extrinsic)
  - defaultId (intrinsic)

- According Flyweight pattern, first we should identity intrinsic (sharable) and extrinsic (non-sharable) data or properties of the object/ class. While designing class we should declare intrinsic properties as member variable (non-static) and we should extrinsic data/ properties as java method param values.

```
public class Circle {
        private String label; (intrinsic data/ property)

        public Circle () {
                label="circle";
        }

        public void draw (float radius, String fillColor, String lineSyle) {
                                        (Extrinsic data/ property)
                ............
        }
}
```

- After designing classes by identifying and keeping intrinsic and extrinsic properties, now we need to take Flyweight factory class, to create single object for these classes to keep them in the cache for reusability. So, every class single object can be used for multiple times to complete tasks.



Spring IoC container acting Flyweight DP based container while working singleton scope beans, because it creates single object for Singleton scope Spring bean and keeps that object in the internal cache IoC container for reusability.

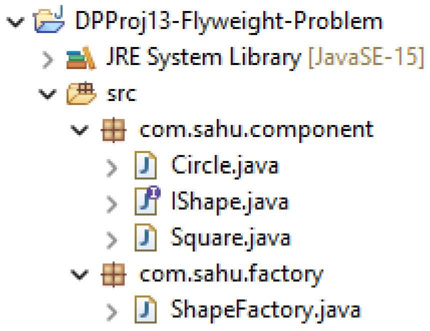
Servlet container is also designed based on Flyweiht design pattern.

Q. What is difference b/w singleton java class and Flyweight pattern?

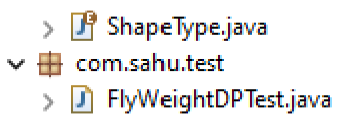
Ans.

- Singleton java class allows to create only object in any situation, and its object will not be placed in any cache. (The private static <class> INSTANCE property itself acts as cache, in case singleton java class).
- In Flyweight pattern, the classes are normal classes if needed we can create multiple objects for those classes having different intrinsic states. But we generally create one object and keeps that object in the internal cache for reusability.
- If there is a need of taking a class without state or really only state or sharable state then go for singleton java class.
- If there is a need of taking multiple classes of same family and reusing their single objects by keeping in cache then go for flyweight pattern.
- If Spring IoC containers and Servlet container are not given based on Flyweight design pattern then developing multiple or all servlet classes, multiple or all Spring bean classes as singleton java classes manually is very complex process for the programmers.

Directory Structure of DPProj13-FlyWeightDP-Problem:



```
DPProj13-Flyweight-Problem
  JRE System Library [JavaSE-15]
  src
    com.sahu.component
      Circle.java
      IShape.java
      Square.java
    com.sahu.factory
      ShapeFactory.java
```



```
> ShapeType.java
v com.sahu.test
  > FlyWeightDPTest.java
```

- Develop the above directory Structure package, class in src folder.
- Then use the following code with in their respective file.

IShape.java

```java
package com.sahu.component;

public interface IShape {
	public void draw(float arg0, String fillColor, String lineStyle);
}
```

Circle.java

```java
package com.sahu.component;

public class Circle implements IShape {

	private String label;

	public Circle() {
		System.out.println("Circle.Circle()");
		label = "CIRCLE";
	}

	@Override
	public void draw(float radius, String fillColor, String lineStyle) {
		System.out.println("Drawing "+label+" with radius "+radius+"
having fill color "+fillColor+", line style "+lineStyle);
	}

}
```

Square.java

```java
package com.sahu.component;

public class Square implements IShape {

	private String label;
	public Square() {
		System.out.println("Square.Square()");
		label = "SQUARE";
	}

	@Override
	public void draw(float side, String fillColor, String lineStyle) {
		System.out.println("Drawing "+label+" with side length
"+side+" having fill color "+fillColor+", line style "+lineStyle);
	}

}
```

ShapeType.java

```
package com.sahu.factory;

public enum ShapeType {
	CIRCLE, SQUARE;
}
```

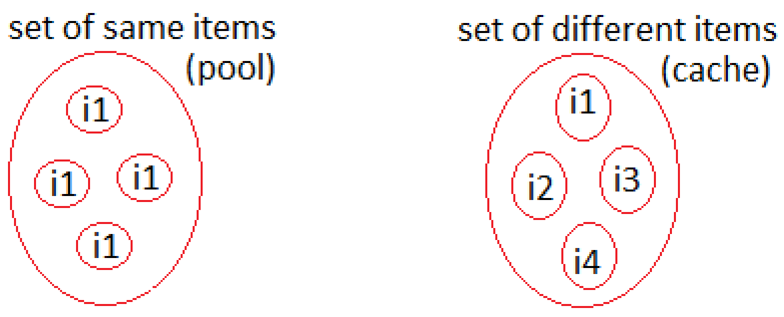
ShapeFactory.java

```
package com.sahu.factory;

import com.sahu.component.Circle;
import com.sahu.component.IShape;
import com.sahu.component.Square;

public class ShapeFactory {

	public static IShape getShape(ShapeType shapeType) {
		IShape shape = null;
		if (shapeType.equals(ShapeType.SQUARE))
			shape = new Square();
		else if (shapeType.equals(ShapeType.CIRCLE))
			shape = new Circle();

		return shape;
	}

}
```

FlyWeightDPTest.java

```
package com.sahu.test;

import com.sahu.component.IShape;
import com.sahu.factory.ShapeFactory;
import com.sahu.factory.ShapeType;

public class FlyWeightDPTest {

	public static void main(String[] args) {
		for (int i=1; i <=500; i++) {
			IShape shape = 
ShapeFactory.getShape(ShapeType.CIRCLE);
			shape.draw(i+10, "Red", "Dotted");
		}
		System.out.println("--------------------");
		for (int i=1; i <=500; i++) {
			IShape shape = 
ShapeFactory.getShape(ShapeType.SQUARE);
			shape.draw(i+10, "Red", "Dotted");
		}
	}

}
```

Problem: To draw 500 circles, this app is using 500 objects for circle/ class, which lead to memory issues and system hang-up.



[Gives the reusability of same items]
(To implement this use List collection or Array)

[Gives the reusability of different items]
(To implement this use Map)

Solution: Place caching logic Flyweight factory class.

Directory Structure of DPProj14-FlyweightDP-Solution:

- Copy & paste the DPProj13-FlyweightDP-Problem project and rename to DPProj14-FlyweightDP-Solution.
- No extra class or package added.
- Then use the following code with in their respective file.

ShapeFactory.java

```
package com.sahu.factory;

import java.util.HashMap;
import java.util.Map;

import com.sahu.component.Circle;
import com.sahu.component.IShape;
import com.sahu.component.Square;

public class ShapeFactory {

	private static Map<ShapeType, IShape> cacheMap = new
HashMap<>();

	public static IShape getShape(ShapeType shapeType) {
		IShape shape = null;
		if (!cacheMap.containsKey(shapeType)) {
			if (shapeType.equals(ShapeType.SQUARE))
				shape = new Square();
			else if (shapeType.equals(ShapeType.CIRCLE))
				shape = new Circle();

			//Keep shape object in cache
			cacheMap.put(shapeType, shape);
		}
		shape = cacheMap.get(shapeType);

		return shape;
	}

}
```

Note: No run the test class you can see only object will created.

Cache Map (using HashMap)

circle	Circle class object reference
square	Square class object reference
key (String)	values (IShape object)

Note: Here, there will not be any multithreading issue though multiple threads are acting on single object of each Shape class object because only sharable/ common intrinsic data will be there inside the object, the entire extrinsic data is coming to object from outside the object.

Flyweight Design Patter (recognizable by creational methods returning a cached instance, a bit the "multitone" idea.)

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

```
Integer i1 = Integer.valueOf(10);
Integer i2 = Integer.valueOf(10);
Integer i3 = Integer.valueOf(10);
```

Creates one Integer wrapper class objects having value 10 and reuses for multiple times.

```
Integer i1 = new Integer(10);
Integer i2 = new Integer(20);      3 objects will be created
Integer i3 = new Integer(30);
```

Note: public static Integer valueOf(int i) - Returns an Integer instance representing the specified int value. If a new Integer instance is not required, this method should generally be used in preference to the constructor Integer(int), as this method is likely to yield significantly better space and time performance by caching frequently requested values. This method will always cache values in the range -128 to 127, inclusive, and may cache other values outside of this range.

Strategy Design Pattern

- In project development, we develop multiple classes having dependency.
- The class that takes the services/ support of other class is called target class and the class that acts as helper class is called dependent class.

- Strategy design pattern says, develop the classes of dependency management (target, dependent classes) as the loosely coupled interchange parts (We can replace one dependent with another dependent without disturbing the source code of target class).

```
target class                dependent class
Myntra <-----------------> DTDC/ BlueDart/ DHL/ FedEx/....
Vehicle <-----------------> DieselEngine/ PetrolEngine/ CNGEngine/ ....
Student <----------------> JavaCourse/ DotNetCourse/ PHPCourse/....
DAO class <-------------> ApacheDBCP/ HikariCP/ TomcatCP/....
Service class <---------> OracleEmployeeDAO/ MysqlEmployeeDAO/....
```

- It is not Spring design pattern, but become popular because of Spring.
- It is GOF design pattern and can be implanted in any OOP language.

Developing the classes of dependency management (target and dependent classes) based on Strategy design pattern is all about implementing 3 rules/ principles

1. Favor Composition over Inheritance
2. Always code to interfaces/ abstract classes, never code to implementation classes/ concreate classes to achieve loose coupling
3. Our code must be open for extension and must be closed for modification.

Favor Composition over Inheritance

```
class A {                class B extends A {
      ..........               ...............        Inheritance (IS-A relation)
      ..........               ...............
}                        }

class A {                class B {
      ..........               A a = new A ();        Composition (HAS-A relation)
      ..........               ...............
}                        }
```

- If class1 wants to use all properties and methods of (total functionality) classes2 then go for inheritance (class1 extends class2).
- If class1 wants to use partial properties and partial methods of class2 then go for composition.

Limitation of Inheritance:

- Some OOP languages does not support multiple inheritance through classes.
- With inheritance code becomes fragile
- With inheritance code testing is bit complex.

Some OOP languages do not support multiple inheritance through classes:

Problem:

```
class A {          class B {          class C extends A, B {
    ……                 ……….                  ………..    Invalid statement (Java
}                  }                  }                 does not support multiple
                                                        inheritance though classes)
```

Solution:

```
class C {
    A a-new A ();          (In composition we can work with
    B b=new B ();          multiple other classes functionalities)
    ……….
}
```

With inheritance Code becomes fragile (easily breakable):

Problem:

```
class A {                    class B extends A {        class C extends B {
    public int x1() {            public int x1() {          public int x1() {
        ……                           ……..                       ………….
        return 100;                  return 1000;               return 500;
    }                            }                          }
}                            }                          }
```

- If u modify the signature of x1() method in class A then all the classes that are in the inheritance hierarchy will be broken/ disturbed (This makes code as easily breakable).
- If the signatures of java.lang.Object class methods are disturbed then, entire Java classes hierarchy will be collapsed.

Solution:

```
class A {                          class c extends B {        class D extends C {
   public ~~int~~ float m1() {         public int m1() {          public int m1() {
        ………                               ……..                       ……..
        return 100;                        return 1000;               return 500;
   }                                   }                          }
}                                  }                          }
```

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

- If modify the return type of m1() method class A from int to float, then only one line code of B class will be distributed, we can solve that problem by correcting the code, other classes will not be distributed.

With inheritance code testing is bit complex:

Problem:

```
class A {                           class B extends A {
     public int m1() {                   public int m3() {
         ...........                         ..........
         return 400;                         return 100;              B b = new B ();
     }                                   }                            b.m1();
     public int m2() {                   public int m4() {            b.m2();
         ...........                         ...........              b.m3();
         return 500;                         return 500;              b.m4();
     }                                    }
}                                   }
```

- Here while testing sub classes, we need to perform testing not only on direct methods of sub class, we should perform unit testing on inherited methods of other classes that are there in the hierarchy of inheritance classes.

Note:

- ✓ Testing means checking actual results with expected results if matched test result is positive if not matched test result is negative.
- ✓ Programmer testing, on his own piece code is called unit testing.
- ✓ We can do unit testing either manually or by taking the support of JUnit tool.

Solution:

```
class A {                             class B extends A {
      public int m1() {                     private A a = new A ();
           ..........                       public int m3() {
           return 400;                           int x = a.m1();           B b = new B ();
      }                                          .......                   b.m3();
      public int m2() {                          return x+100;
           ..........                       }
           return 500;                }
      }
}
```

- Here we need to do unit testing only on B class methods. i.e., only m3() should tested. This reduces burden on the programmer.

Always code to interfaces/ abstract classes, never code to implementation classes/ concreate classes to achieve loose coupling

Problem:

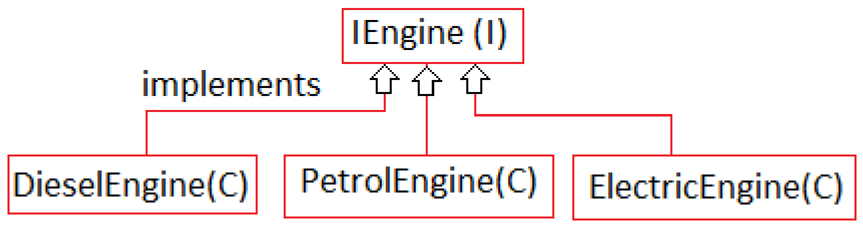
Target class

```
public class Vehicle {
           private DieselEngine engine = new DieselEngine();
           ..........              To change dependent from DieselEngine to
           ..........              PetrolEngine we need to modify the source code
                                   target class i.e., there is tight coupling.
}
```

Dependent classes

```
public class DieselEngine {           public class PetrolEngine {
           ..........                            ...........
           ........                              ........
}                                     }
```

Solution:



- Make all Dependent classes implementing common interface and that interface reference variable as a "HAS-A" property in target classes supporting composition as show above.

Target class

```
public class Vehicle {
        private IEngine engine; //HAS-A property of type interface

        //setter method
        public void setEngine(Engine engie){
                this.engine=engine;
        }
        …….
        ……..
}
```

In Client app

```
//create dependent class object
IEngine engg = new DieselEngine();
//create target class object
Vehicle vehicle1 = new Vehicle();
//assign or inject dependent class obj to target class obj
vehicle1.setEngine(engg);

//To change dependent
engg = new PetrolEngine();
vehicle1.setEngine(engg);
```

- Here we can change one dependent object to another dependent object without touching the source code of target class. This indicates target and dependent classes having loose coupling.

Our code must be open for extension and must be closed for modification

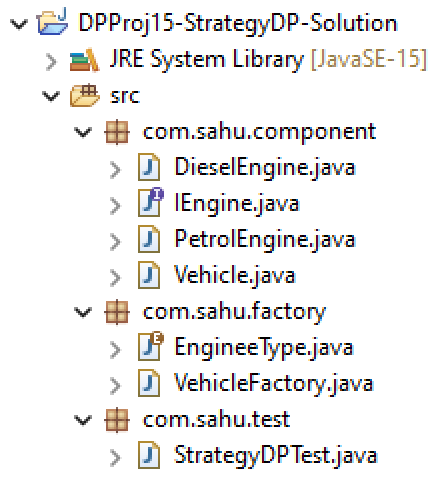
- If we follow Strategy design pattern ruleno:2 (nothing but working with Interface model assignment), our code automatically becomes open for extension because we can add more implementation class for common interfaces as new dependent classes to target class.
- While developing the classes of the application, if we take those classes as the final classes or its methods as final methods then that application and its classes become closed for modification.

- We cannot develop sub classes for final classes and we cannot override final methods of super class in sub classes.
- String class, wrapper classes and etc. classes are given as final classes for not allowing developers to develop sub classes and to override the logics.

Note:

- ✓ If you are using Strategy design pattern in the environment where underlying servers/ container/ frameworks are forced to generate InMemory sub classes for our application classes then taking our application classes as final classes is partially not possible.
- ✓ In spring programming while working with Lookup method injection, Method replacer, Spring AOP and etc. concepts
- ✓ In Hibernate programming while working with ses.load(-,-) with lazy="true" and proxy="......".

Directory Structure of DPProj15-StrategyDP-Solution:



```
DPProj15-StrategyDP-Solution
  JRE System Library [JavaSE-15]
  src
    com.sahu.component
      DieselEngine.java
      IEngine.java
      PetrolEngine.java
      Vehicle.java
    com.sahu.factory
      EngineeType.java
      VehicleFactory.java
    com.sahu.test
      StrategyDPTest.java
```

- Develop the above directory Structure package, class in src folder.
- Then use the following code with in their respective file.

IEngine.java

```
package com.sahu.component;

public interface IEngine {

	public void start();
	public void stop();

}
```

DieselEngine.java

```
package com.sahu.component;

public final class DieselEngine implements IEngine {

	@Override
	public void start() {
		System.out.println("Diesel Engine has started");
	}

	@Override
	public void stop() {
		System.out.println("Diesel Engine has stoped");
	}

}
```

PetrolEngine.java

```
package com.sahu.component;

public final class PetrolEngine implements IEngine {

	@Override
	public void start() {
		System.out.println("Petrol Engine has started");
	}

	@Override
	public void stop() {
		System.out.println("Petrol Engine has stoped");
	}

}
```

Vehicle.java

```
package com.sahu.component;

public final class Vehicle {
	private IEngine engine;    //HAS-A property

	public void setEngine(IEngine engine) {
		this.engine = engine;
	}

	public Vehicle() {
		System.out.println("Vehicle.Vehicle()");
	}

	public void drive(String sourcePlace, String destPlace) {
		System.out.println("Vehicle.drive()");
		engine.start();
		System.out.println("Driving started at "+sourcePlace);
		System.out.println("Driving is going on");
		engine.stop();
		System.out.println("Driving ended at "+destPlace);
	}

}
```

EngineeType.java

```
package com.sahu.factory;

public enum EngineeType {
	DIESEL, PETROL;
}
```

VehicleFactory.java

```
package com.sahu.factory;

import com.sahu.component.DieselEngine;
import com.sahu.component.IEngine;
import com.sahu.component.PetrolEngine;
import com.sahu.component.Vehicle;

public class VehicleFactory {

	public static Vehicle getVehicle(EngineeType engineeType) {
```

```
                IEngine engine = null;
                if (engineeType.equals(EngineeType.DIESEL))
                        engine =  new DieselEngine();
                else if (engineeType.equals(EngineeType.PETROL))
                        engine =  new PetrolEngine();

                //Create target class object
                Vehicle vehicle = new Vehicle();
                //assign dependent class object to target class object
                vehicle.setEngine(engine);

                return vehicle;
        }

}
```

StrategyDPTest.java

```
package com.sahu.test;

import com.sahu.component.Vehicle;
import com.sahu.factory.EngineeType;
import com.sahu.factory.VehicleFactory;

public class StrategyDPTest {

        public static void main(String[] args) {
                //Use factory
                Vehicle vehicle =
VehicleFactory.getVehicle(EngineeType.DIESEL);
                vehicle.drive("Hyd", "GOA");
        }

}
```

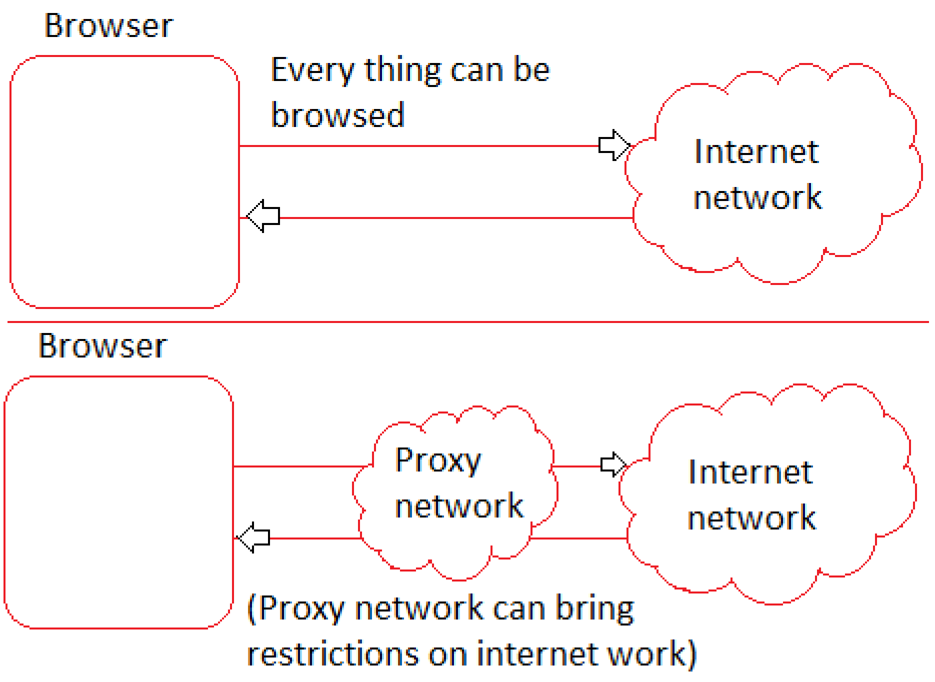
Strategy Design Pattern (recognizable by behavioral methods in an abstract/ interface type which invokes a method in an implementation of a different abstract/ interface type which has been passed-in as method argument into the strategy implementation)

- 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().

Proxy Design Pattern

- Proxy means duplicate that acts as real on temporary basis. Every proxy internally uses real, but allows to added additional functionalities on temporary basis without disturbing the real one.
- Working with Proxy gives the felling of working the real, but proxy adds new functionalities dynamically.


Browser
Every thing can be browsed
Internet network
Browser
Proxy network
Internet network
(Proxy network can bring restrictions on internet work)

- Proxy object adds additional functionalities on top of existing real object dynamically at runtime on temporary basis. Every proxy object internally uses real object after adding additional functionalities.

Note: These proxy classes can be developed manually or can be generated as the InMemory proxy classes (These are the classes generated in JVM Memory of RAM dynamically at runtime) using JDK libraries or CGLIB (Code Generation Library).

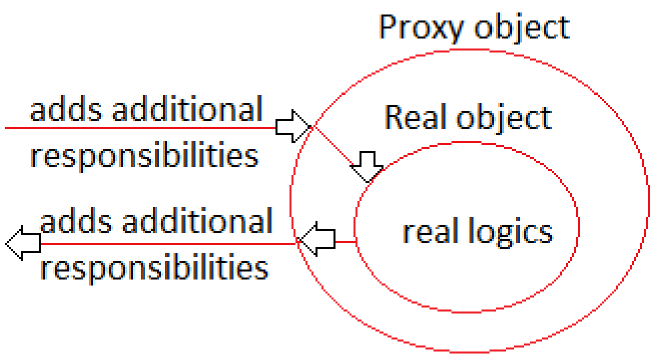
Normal classes
.java (HDD) ------> .class (HDD) ------> execution in the JVM memory of RAM

InMemory classes

Source code generation (JVM memory) ------> byte code generation (JVM memory) ------> execution of the class code (JVM Memory)

Components of Proxy design pattern implementation:

a. Component interface: contains the declaration of methods, and it is common interface for both concrete component class and proxy class.
b. Concrete component class: implements component interface and contains real logics.
c. Proxy class: implements component interface or extends concrete component class and uses concrete component class object internally to complete the task after adding the additional functionalities.
d. Factory class: returns either proxy class object or real class (concrete component) class object based on the need.


Proxy object
adds additional responsibilities
Real object
real logics
adds additional responsibilities

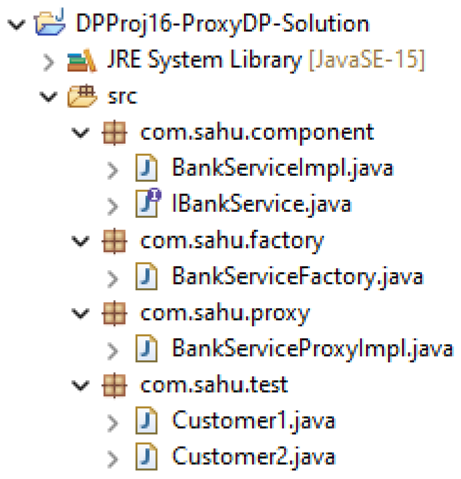
- Bank contains withdraw, deposit logics as normal logics or real logics. During demonetization restriction have come on withdraw and deposit operations (It allows to withdraw max of 4000) that to 2 months period. It is better to write these new logics in proxy class which internally takes the support real class.

  IBankService.java (component interface)
  BankServiceImpl.java (real class)
  BankServiceProxyImpl.java (proxy class)
  BankServiceFactory
  gives either real class or proxy class object based on demonetization is enabled or not
  Client app

- Both real and proxy class implement Component interfaces so, that client can refer both objects by using same component interface reference variable.

Note: If do not take Proxy interface separately then the proxy class should directly extend from real class.

Directory Structure of DPProj16-ProxyDP-Solution:



```
DPProj16-ProxyDP-Solution
  JRE System Library [JavaSE-15]
  src
    com.sahu.component
      BankServiceImpl.java
      IBankService.java
    com.sahu.factory
      BankServiceFactory.java
    com.sahu.proxy
      BankServiceProxyImpl.java
    com.sahu.test
      Customer1.java
      Customer2.java
```

- Develop the above directory Structure package, class in src folder.
- Then use the following code with in their respective file.

IBankService.java

```java
package com.sahu.component;

public interface IBankService {

	public String withdraw(long accNo, double amount);

}
```

BankServiceImpl.java

```java
package com.sahu.component;

public class BankServiceImpl implements IBankService {

	@Override
	public String withdraw(long accNo, double amount) {
		return "Withdrawing amount "+amount+" from account 
number "+accNo;
	}

}
```

BankServiceProxyImpl.java

```java
package com.sahu.proxy;

import com.sahu.component.BankServiceImpl;
import com.sahu.component.IBankService;

public class BankServiceProxyImpl implements IBankService {

	private IBankService realService;

	public BankServiceProxyImpl() {
		realService = new BankServiceImpl();
	}

	@Override
	public String withdraw(long accNo, double amount) {
		String msg = null;
		if (amount<=4000)
			msg = realService.withdraw(accNo, amount);
		else {
			msg = realService.withdraw(accNo, 4000);
			msg = msg +"\n**** Only max of 4000 can be withdraw
during transition period  ****";
		}
		return msg;
	}

}
```

BankServiceFactory.java

```java
package com.sahu.factory;

import com.sahu.component.BankServiceImpl;
import com.sahu.component.IBankService;
import com.sahu.proxy.BankServiceProxyImpl;

public class BankServiceFactory {

	public static IBankService getInstance(boolean demonetization) {
		IBankService service = null;
```

```
                if (demonetization)
                        service = new BankServiceProxyImpl();
                else
                        service = new BankServiceImpl();
                return service;
        }

}
```

Customer1.java

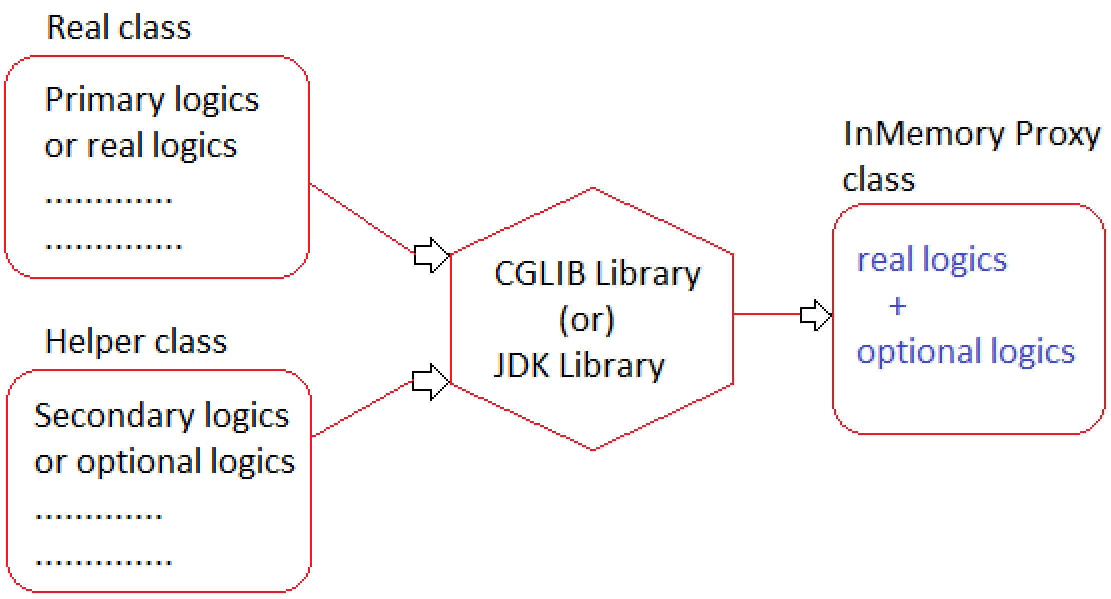
```
package com.sahu.test;

import com.sahu.component.IBankService;
import com.sahu.factory.BankServiceFactory;

public class Customer1 {

        public static void main(String[] args) {
                IBankService service = BankServiceFactory.getInstance(true);
                System.out.println(service.withdraw(100001, 300000));
                System.out.println("-----------------");
                System.out.println(service.withdraw(100001, 3000));
        }

}
```

Customer2.java

```
package com.sahu.test;

import com.sahu.component.IBankService;
import com.sahu.factory.BankServiceFactory;

public class Customer2 {

        public static void main(String[] args) {
                IBankService service = BankServiceFactory.getInstance(false);
                System.out.println(service.withdraw(100002, 300000));
                System.out.println("-----------------");
                System.out.println(service.withdraw(100002, 3000));
        }

}
```

Proxy design pattern implementation using InMemory proxy classes with the support of special libraries:

- In the manually created proxy class, all standard of Proxy design pattern implementation should be taken care by programmer explicitly.
- If we use either CGLIB libraries or JDK libraries to generate same proxy classes as InMemory classes then all standard related to Proxy design pattern will be taken internally more ever it will dynamically mix up main logics and additional logics in the InMemory classes.


Real class
Primary logics or real logics
....
....
Helper class
Secondary logics or optional logics
....
....
CGLIB Library (or) JDK Library
InMemory Proxy class
real logics + optional logics

- Spring AOP, Spring Method Replacer, Lookup Method Injection and etc. concepts are developed based Proxy pattern using InMemory proxy class generation support.
- ses.load(-,-) of Hibernate also designed based on Proxy design pattern.

Generating InMemory proxy class using CGLIB library

Step 1: Create a project with concrete component class, need not to create component interface, and add a withdraw logic as like previous application.

Step 2: Add the following two jar files

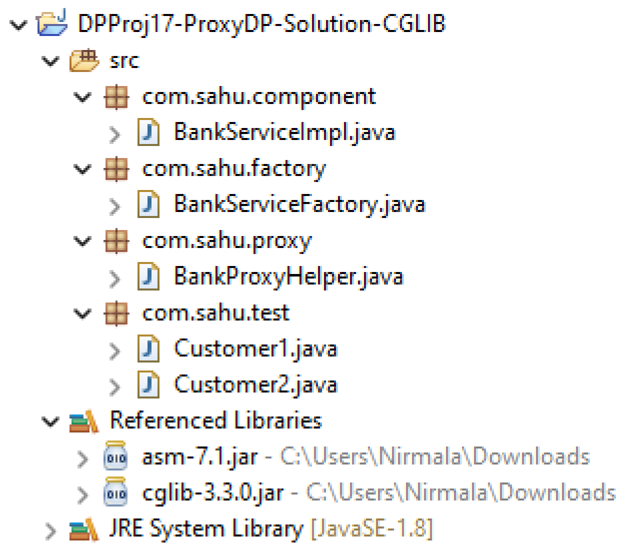
cglib-3.3.0.jar, CGLIB mvnrepository.com
asm-7.1.jar, ASM mvnrepository.com

Step 3: Develop Proxy helper class having additional logics to be used in InMemory proxy class generation.

Step 4: Improve factory class code.

Step 5: Develop the client application.

Directory Structure of DPProj17-ProxyDP-Solution-CGLIB:



```
DPProj17-ProxyDP-Solution-CGLIB
  src
    com.sahu.component
      BankServiceImpl.java
    com.sahu.factory
      BankServiceFactory.java
    com.sahu.proxy
      BankProxyHelper.java
    com.sahu.test
      Customer1.java
      Customer2.java
  Referenced Libraries
    asm-7.1.jar - C:\Users\Nirmala\Downloads
    cglib-3.3.0.jar - C:\Users\Nirmala\Downloads
  JRE System Library [JavaSE-1.8]
```

- Develop the above directory Structure package, class in src folder.
- Add the following jars to the CLASSPATH.
  - cglib-3.3.0.jar, CGLIB mvnrepository.com
  - asm-7.1.jar, ASM mvnrepository.com
- Then use the following code with in their respective file.

BankServiceImpl.java

```
package com.sahu.component;

public class BankServiceImpl {

	public String withdraw(long accNo, double amount) {
		return "Withdrawing amount "+amount+" from account
number "+accNo;
	}

}
```

BankProxyHelper.java

```
package com.sahu.proxy;

import java.lang.reflect.Method;

import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;

public class BankProxyHelper implements MethodInterceptor {

	@Override
	public Object intercept(Object realObject, Method realMethod,
Object[] args, MethodProxy proxy) throws Throwable {
		Object returnVal = null;
		if (realMethod.getName().equalsIgnoreCase("withdraw")) {
			if (((double)args[1])<=4000)
				returnVal = proxy.invokeSuper(realObject, args);
			else {
				args[1] = 4000;
				returnVal = proxy.invokeSuper(realObject, args);
				returnVal = returnVal+"\n**** Only max of 4000
can be withdraw during transition period ****";
			}
		}
		else {
			returnVal = proxy.invokeSuper(realObject, args);
		}
		return returnVal;
	}

}
```

BankServiceFactory.java

```
package com.sahu.factory;

import com.sahu.component.BankServiceImpl;
import com.sahu.proxy.BankProxyHelper;

import net.sf.cglib.proxy.Enhancer;

public class BankServiceFactory {

	public static BankServiceImpl getInstance(boolean demonetization) {
		BankServiceImpl service = null;
		if (demonetization)
			service = (BankServiceImpl)
Enhancer.create(BankServiceImpl.class, new BankProxyHelper());
                else
                        service  =  new BankServiceImpl();

                return service;
        }

}
```

Customer1.java

```
package com.sahu.test;

import com.sahu.component.BankServiceImpl;
import com.sahu.factory.BankServiceFactory;

public class Customer1 {

        public static void main(String[] args) {
                BankServiceImpl service =
BankServiceFactory.getInstance(true);

        System.out.println(service.getClass()+"...."+service.getClass().getSupe
rclass());
                System.out.println(service.withdraw(100001, 300000));
                System.out.println("------------------");
                System.out.println(service.withdraw(100001, 3000));
        }

}
```

Customer2.java

```
package com.sahu.test;

import com.sahu.component.BankServiceImpl;
import com.sahu.factory.BankServiceFactory;

public class Customer2 {

        public static void main(String[] args) {
```

```
                    BankServiceImpl service = 
BankServiceFactory.getInstance(false);

         System.out.println(service.getClass()+"...."+service.getClass().getSupe
rclass());
                    System.out.println(service.withdraw(100002, 300000));
                    System.out.println("------------------");
                    System.out.println(service.withdraw(100002, 3000));
         }

}
```

Note:

- ✓ CGLIB library always generates InMemory proxy class as the sub class of real/ target class. So, no need of taking component interface (interface implemented by real/ target class).
- ✓ If you are using Open JRE and not getting any exception then go with Open JRE, if you are getting below exception then use the installed JDK of your system.

```
Exception in thread "main" java.lang.ExceptionInInitializerError
         at com.sahu.factory.BankServiceFactory.getInstance
                                        (BankServiceFactory.java:13)
         at com.sahu.test.Customer1.main(Customer1.java:9)
Caused by: net.sf.cglib.core.CodeGenerationException:
java.lang.reflect.InaccessibleObjectException-->Unable to make
protected final java.lang.Class java.lang.ClassLoader.defineClass
(java.lang.String,byte[],int,int,java.security.ProtectionDomain) throws
java.lang.ClassFormatError accessible: module java.base does not
"opens java.lang" to unnamed module @7f690630
```

Flow execution of Proxy class:

BankServiceImpl.java

```
public class BankServiceImpl {

#17     public String withdraw(long accNo, double amount) {
                return "Withdrawing amount "+amount+" from account number
"+accNo;        #18
        }

}
```

BankServiceFactory.java

```
public class BankServiceFactory {        #2
        public static BankServiceImpl getInstance(boolean demonetization) {
                BankServiceImpl service = null;
                if (demonetization)
        #6              service = (BankServiceImpl)
Enhancer.create(BankServiceImpl.class, new BankProxyHelper()); #3
                else
                        service = new BankServiceImpl();
                return service; #7
        }
}
```

Customer1.java

```
public class Customer1 {
        public static void main(String[] args) {                #1
        #8      BankServiceImpl service = BankServiceFactory.getInstance(true);
        #23     System.out.println(service.withdraw(100001, 300000)); #9
                System.out.println("------------------");
                System.out.println(service.withdraw(100001, 3000));
        }
}
```

BankServiceImpl$$EnhanceCGLIB$$.java (Sample Proxy class given CGLIB library) #4   #5 Proxy class object creation.

```
public class BankServiceImpl$$EnhanceCGLIB$$ extnds BankServiceImpl {
#10     public String withdraw (long accNo, double amount) {
                //gathers second arg value from Enhancer.create(-,-,-,-) i.e.,
                BankProxyHelper class object (helper)
                ...............
                //gathers 1st arg value from Enhancer.create(-,-,-,-) method
                i.e., BankServiceImpl.class and creates object for it.
        #11     BankServieImpl realObject = (BankServiceIml)
                                        BankServiceIml.class.newInstance();
                //prepare method class object representing withdraw method
                Method realMethod =
                        realMethod.getDeclaredMethods("withdraw");
                //prepare Object[] having args
                Object args[] = new Object[]{accNo, amount};
                //prepare MethodProxy object (proxy)
```

```
                    ……
                    ……..
                    //invoke method
          #21       Object result =                    #12
                              helper.intercept(realObject, realMethod, args, proxy);
                    return result; #22
          }
}
```

BankProxyHelper.java

```
public class BankProxyHelper implements MethodInterceptor {
        @Override   #13
        public Object intercept(Object realObject, Method realMethod, Object[]
args, MethodProxy proxy) throws Throwable {
                Object returnVal = null;
                if (realMethod.getName().equalsIgnoreCase("withdraw")) { #14
                        if (((double)args[1])<=4000)
                                returnVal = proxy.invokeSuper(realObject, args);
                        else {
                        |       args[1] = 4000;
                #15     |       returnVal = proxy.invokeSuper(realObject, args); #16
                        |       returnVal = returnVal+"\n**** Only max of 4000 can
be withdraw during transition period ****"; #19
                        }
                }
                else {
                        returnVal = proxy.invokeSuper(realObject, args);
                }
                return returnVal; #20
        }
}
```

Note: BankProxyHelper.java is not a proxy class. It is a helper class having additional code to placed in dynamically generated InMemory proxy class.

Limitation with CGLIB Library:
Here Proxy class will come as the sub class of target/ real class, so

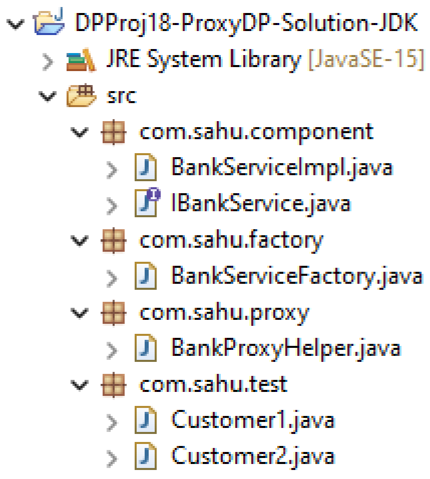
a. We cannot real class as final class (against of Strategy design pattern).
b. We cannot take real class methods as final or static methods (against of Strategy design pattern).

- To overcome these problems, use JDK Libraries to generate InMemory proxy class generation.
- JDK Library generates the InMemory proxy class as the implementation class of component interface (the interface that is implemented by concrete component class/ real class), so
  - We can take real class as the final class (full support for Strategy design pattern).
  - We can take real class methods as final methods or static methods.
  - No need of adding additional third-party libraries to CLASSATH

Generating InMemory proxy class using JDK library

Step 1: Keep basic Proxy design pattern project ready. Make sure that real/ target class is implementing component interface/ proxy interface.
Step 2: Develop the Proxy Helper class using JDK Reflection API.
Step 3: Develop the factory class.
Step 4: Develop client application.

Directory Structure of DPProj18-ProxyDP-Solution-JDK:


```
DPProj18-ProxyDP-Solution-JDK
  JRE System Library [JavaSE-15]
  src
    com.sahu.component
      BankServiceImpl.java
      IBankService.java
    com.sahu.factory
      BankServiceFactory.java
    com.sahu.proxy
      BankProxyHelper.java
    com.sahu.test
      Customer1.java
      Customer2.java
```

- Develop the above directory Structure package, class in src folder.
- Then use the following code with in their respective file.

IBankService.java

```
package com.sahu.component;

public interface IBankService {
        public String withdraw(long accNo, double amount);
}
```

BankServiceImpl.java

```
package com.sahu.component;

public class BankServiceImpl implements IBankService {

	@Override
	public String withdraw(long accNo, double amount) {
		return "Withdrawing amount "+amount+" from account 
number "+accNo;
	}

}
```

BankProxyHelper.java

```
package com.sahu.proxy;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

import com.sahu.component.BankServiceImpl;
import com.sahu.component.IBankService;

public class BankProxyHelper implements InvocationHandler {

	private IBankService realService = null;

	public BankProxyHelper() {
		System.out.println("BankPoxyHelper.BankPoxyHelper()");
		realService = new BankServiceImpl();
	}

	@Override
	public Object invoke(Object proxy, Method method, Object[] args) 
throws Throwable {
		Object retVal = null;
		if (method.getName().equalsIgnoreCase("withdraw")) {
			if (((double)args[1])<=4000)
				retVal = method.invoke(realService, args);
			else {
					args[1] = 4000;
					retVal = method.invoke(realService, args);
					retVal = retVal+"\n**** Only max of 4000 can be
withdraw during transition period ****";
				}
			}
			else {
				retVal = method.invoke(retVal, args);
			}
			return retVal;
		}

}
```

BankServiceFactory.java

```
package com.sahu.factory;

import java.lang.reflect.Proxy;

import com.sahu.component.BankServiceImpl;
import com.sahu.component.IBankService;
import com.sahu.proxy.BankProxyHelper;

public class BankServiceFactory {

	public static IBankService getInstance(boolean demonetization) {
		IBankService service = null;
		if (demonetization)
			service =(IBankService)
Proxy.newProxyInstance(IBankService.class.getClassLoader(), new Class[]
{IBankService.class}, new BankProxyHelper());
		else
			service  =  new BankServiceImpl();

		return service;
	}

}
```

Customer1.java

```
package com.sahu.test;

import java.util.Arrays;

import com.sahu.component.IBankService;
import com.sahu.factory.BankServiceFactory;

public class Customer1 {

	public static void main(String[] args) {
		IBankService service = BankServiceFactory.getInstance(true);

	System.out.println(service.getClass()+"...."+service.getClass().getSupe
rclass()+"...."+Arrays.toString(service.getClass().getInterfaces()));
		System.out.println(service.withdraw(100001, 300000));
		System.out.println("-------------------");
		System.out.println(service.withdraw(100001, 3000));
	}

}
```

Customer2.java

```
package com.sahu.test;

import java.util.Arrays;

import com.sahu.component.IBankService;
import com.sahu.factory.BankServiceFactory;

public class Customer2 {

	public static void main(String[] args) {
		IBankService service = BankServiceFactory.getInstance(false);

	System.out.println(service.getClass()+"...."+service.getClass().getSupe
rclass()+"...."+Arrays.toString(service.getClass().getInterfaces()));
		System.out.println(service.withdraw(100002, 300000));
		System.out.println("-------------------");
		System.out.println(service.withdraw(100002, 3000));
	}
}
```

Flow execution of Proxy class:

Customer1.java

```
public class Customer1 {
	public static void main(String[] args) {
	(f)	IBankService service = BankServiceFactory.getInstance(true); (a)
	(s)	System.out.println(service.withdraw(100001, 300000)); (g)
		System.out.println(service.withdraw(100001, 3000));
	}
}
```

BankServiceFactory.java

```
public class BankServiceFactory {	(b)
	public static IBankService getInstance(boolean demonetization) {
		IBankService service = null;
		if (demonetization)
			service =(IBankService)
Proxy.newProxyInstance(IBankService.class.getClassLoader(), new Class[]
{IBankService.class}, new BankProxyHelper()); (c)
		else
			service  =  new BankServiceImpl();
		return service; (e) – InMemory proxy class object
	}
}
```

BankServiceFactory.java

```
public class BankProxyHelper implements InvocationHandler {
	private IBankService realService = null;
	public BankProxyHelper() {
		System.out.println("BankPoxyHelper.BankPoxyHelper()");
		realService = new BankServiceImpl();
	}
	@Override   (k)
	public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
		Object retVal = null;
		if (method.getName().equalsIgnoreCase("withdraw")) {
			if (((double)args[1])<=4000)
				retVal = method.invoke(realService, args);
			else {
				args[1] = 4000;
```

```
                    (o)   | retVal = method.invoke(realService, args); (l)
                          | retVal = retVal+"\n**** Only max of 4000 can be
withdraw during transition period ****";
                    }
                }
                else {
                    retVal = method.invoke(retVal, args);
                }
                return retVal; (p)
        }
}
```

$Proxy0.java (InMemory proxy class)   (d)

```
public class $Proxy0 extends java.lang.reflect.Proxy implments IBankService {
(h)     public String withdraw(long accNo, double amount) {
              | //prepare Object args[] representing the arguments
              | Object args[] = new Object[]{accNo, amount};
              | //prepare Method object
              | Method method =
              |         this.getClass().getDeclaredMethod("withdraw");
(i)           | //prepare proxy object (this)
              | ..........
              | //gather BankProxyHelper class object from 3rd arg  of
              | Proxy.newProxyInstance(-,-,-) (helper object)
              | .........
        (q)     Object retVal = helper.invoke(this, method, args) (j)
                return retVal; (r)
        }
}
```

BankServiceImpl.java

```
public class BankServiceImpl implements IBankService {
        @Override           (m)
        public String withdraw(long accNo, double amount) {
        (n)     return "Withdrawing amount "+amount+" from account number
"+accNo;
        }
}
```

Proxy Design Pattern (recognizable 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)

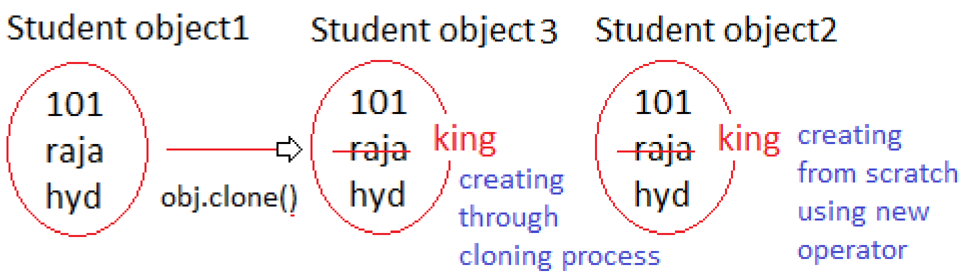
- java.lang.reflect.Proxy
- java.rmi.* (stub and Skelton generation through Proxy design pattern)
- javax.eib.EJB (@EJB because annotation InMemory proxy will be generated for EJB component class)
- javax.iniect.Iniect (@Inject)
- javax.persistence.PersistenceContext (@PersistneceContext) @WebService (Entire web service comes in the Proxy class that is generated as the sub class for current class where @WebService is placed)
- Spring Data JPA Repository Interface and Spring AOP
- Spring Core Lookup method injection, Method Replacer and etc.

Prototype Design Pattern

- The Prototype design pattern is a creational design pattern. It is required, when object creation is time consuming, and costly operation, so we create object with existing object itself. One of the best available ways to create object from existing objects are clone () method. Clone is the simplest approach to implement prototype pattern.
- In simple words we can say "Create objects based on a template of an existing object through cloning".
- Definition according to the Gang of Four: Specify the kind of objects to create using a prototypical instance, and create new objects by copying this prototype.

When to use Prototype Pattern:

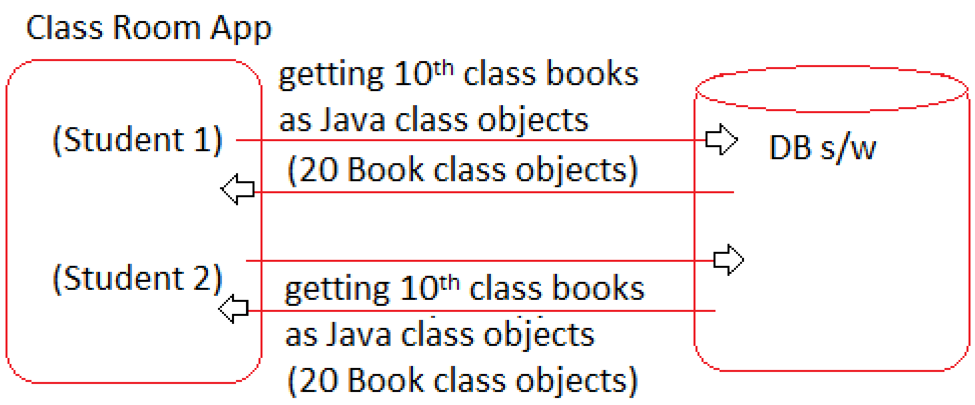
- This pattern is used when creation of object directly is costly.
- For example, an object is to be created after a costly database operation. We can cache the object, returns its clone on next request and update the database as and when needed thus reducing database calls.



Note: The object created through cloning process takes less time and uses less

CPU resources. So, prefer creating new objects of same type with same data or little modified data taking the support of cloning process.

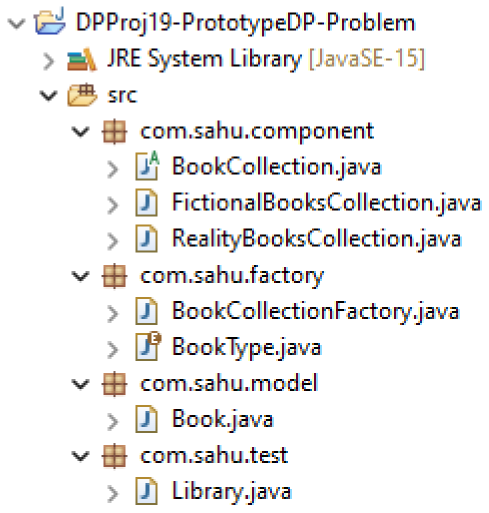
Problem: If 30 students are in class room app, we need to hit DB s/w for 30 times and we need to get same books from DB s/w and need to store in 30 sets of 20 Book class objects (600 objects).



Class Room App
(Student 1)
(Student 2)
getting 10th class books as Java class objects
(20 Book class objects)
getting 10th class books as Java class objects
(20 Book class objects)
DB s/w

Creating 30 sets of 20 Book class objects from scratch level using new operator having same data all the 30 times is bad practice.

Directory Structure of DPProj19-PrototypeDP-Problem:



```
DPProj19-PrototypeDP-Problem
  JRE System Library [JavaSE-15]
  src
    com.sahu.component
      BookCollection.java
      FictionalBooksCollection.java
      RealityBooksCollection.java
    com.sahu.factory
      BookCollectionFactory.java
      BookType.java
    com.sahu.model
      Book.java
    com.sahu.test
      Library.java
```

- Develop the above directory Structure package, class in src folder.
- Then use the following code with in their respective file.

BookType.java

```
package com.sahu.factory;
public enum BookType {
        FICTIONAL, REALITY;       }
```

Book.java

```java
package com.sahu.model;

public class Book {
	private int bookId;
	private String bookName;

	public Book(int bookId, String bookName) {
		System.out.println("Book.Book()");
		this.bookId = bookId;
		this.bookName = bookName;
	}

	public int getBookId() {
		return bookId;
	}

	public void setBookId(int bookId) {
		this.bookId = bookId;
	}

	public String getBookName() {
		return bookName;
	}

	public void setBookName(String bookName) {
		this.bookName = bookName;
	}

	@Override
	public String toString() {
		return "Book [bookId=" + bookId + ", bookName=" + bookName
+ "]";
	}

}
```

BookCollection.java

```java
package com.sahu.component;
```

```
import java.util.ArrayList;
import java.util.List;

import com.sahu.factory.BookType;
import com.sahu.model.Book;

public abstract class BookCollection {
	private BookType bookType;
	private List<Book> books = new ArrayList<>();

	public BookType getBookType() {
		return bookType;
	}

	public void setBookType(BookType bookType) {
		this.bookType = bookType;
	}

	public List<Book> getBooks() {
		return books;
	}

	public void setBooks(List<Book> books) {
		this.books = books;
	}

	public abstract void loadBooks();

	@Override
	public String toString() {
		return "BookCollection [bookType=" + bookType + ", books=" +
books + "]";
	}

}
```

FictionalBooksCollection.java

```
package com.sahu.component;

import com.sahu.factory.BookType;
import com.sahu.model.Book;

public class FictionalBooksCollection extends BookCollection {

	@Override
	public void loadBooks() {
		//write JDBC code to get books from DB s/w and load books 
collection(List Collection)
		System.out.println("hitting DB s/w to get fictional books");
		setBookType(BookType.FICTIONAL);
		Book book = null;
		for (int i = 1; i <= 20; i++) {
			book = new Book(1000+i, BookType.FICTIONAL+"-"+i);
			getBooks().add(book);
		}
		System.out.println("Books are loaded to books collection from 
DB s/w table records");
	}

}
```

RealityBooksCollection.java

```
package com.sahu.component;

import com.sahu.factory.BookType;
import com.sahu.model.Book;

public class RealityBooksCollection extends BookCollection {

	@Override
	public void loadBooks() {
		//write JDBC code to get books from DB s/w and load books 
collection(List Collection)
		System.out.println("hitting DB s/w to get fictional books");
		setBookType(BookType.REALITY);
		Book book = null;
		for (int i = 1; i <= 20; i++) {
			book = new Book(1000+i, BookType.REALITY+"-"+i);
			getBooks().add(book);
		}
```

```
                System.out.println("Books are loaded to books collection from
DB s/w table records");
        }

}
```

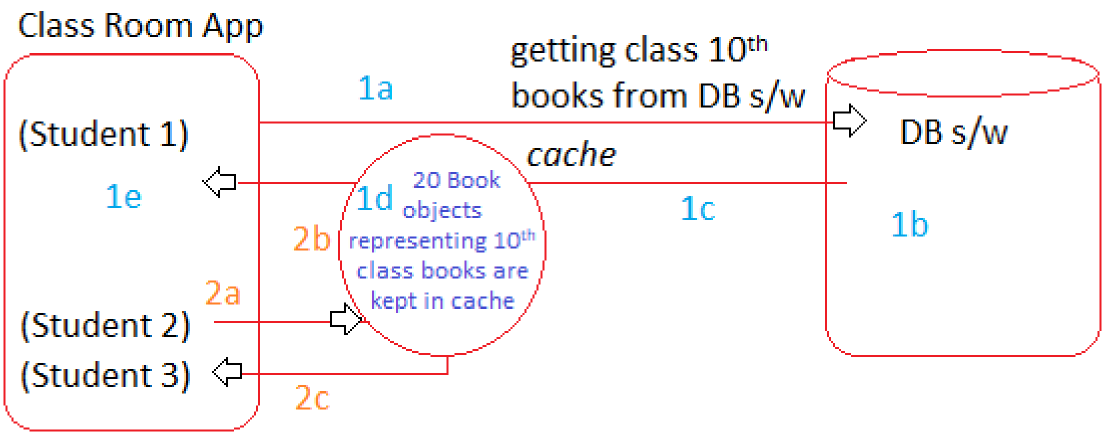
BookCollectionFactory.java

```
package com.sahu.factory;

import com.sahu.component.BookCollection;
import com.sahu.component.FictionalBooksCollection;
import com.sahu.component.RealityBooksCollection;

public class BookCollectionFactory {

        public static BookCollection getBookCollection(BookType bookType) {
                BookCollection bookCollection = null;
                if (bookType.equals(BookType.FICTIONAL))
                        bookCollection = new FictionalBooksCollection();
                else if (bookType.equals(BookType.REALITY))
                        bookCollection = new RealityBooksCollection();

                bookCollection.loadBooks();
                return bookCollection;
        }

}
```

Library.java

```
package com.sahu.test;

import com.sahu.component.BookCollection;
import com.sahu.factory.BookCollectionFactory;
import com.sahu.factory.BookType;

public class Library {

        public static void main(String[] args) {
                BookCollection fBooks =
BookCollectionFactory.getBookCollection(BookType.FICTIONAL);
                System.out.println(fBooks);
                System.out.println("-----------------");
                BookCollection rBooks =
BookCollectionFactory.getBookCollection(BookType.REALITY);
                System.out.println(rBooks);
                System.out.println("*****************");
                BookCollection fBooks1 =
BookCollectionFactory.getBookCollection(BookType.FICTIONAL);
                System.out.println(fBooks1);
                System.out.println("-----------------");
                BookCollection rBooks1 =
BookCollectionFactory.getBookCollection(BookType.REALITY);
                System.out.println(rBooks1);
        }

}
```

Solution 1: (using caching) (Not Recommended)



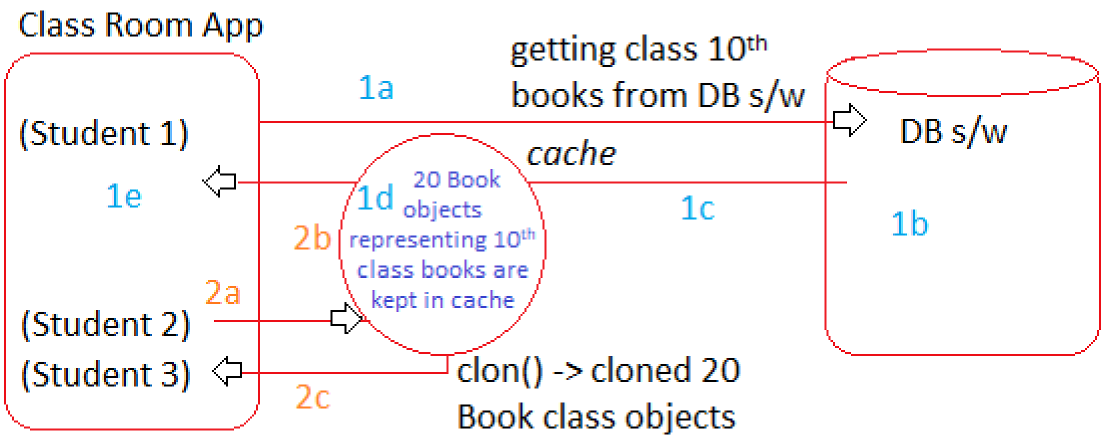
Class Room App
(Student 1)
1e
(Student 2)
(Student 3)
2a
1a
getting class 10th books from DB s/w
cache
1d
20 Book objects representing 10th class books are kept in cache
2b
2c
1c
DB s/w
1b

- By keeping 20 Book class objects representing 10th class books in the cache, we can reduce number of hits to DB s/w. But all the 30 students using same set of books collected cache is also bad practice (sharing of class room books is not practice).

Solution 2: (caching + cloning: Prototype design pattern) (Recommended)

- By keeping 20 Book class objects representing 10th class books in the cache, we can reduce number of hits to DB s/w. Since Student2 onwards every student is getting the cloned copy of 10th class books (20 Book class objects). So, we can say data inconsistency problem is solved.

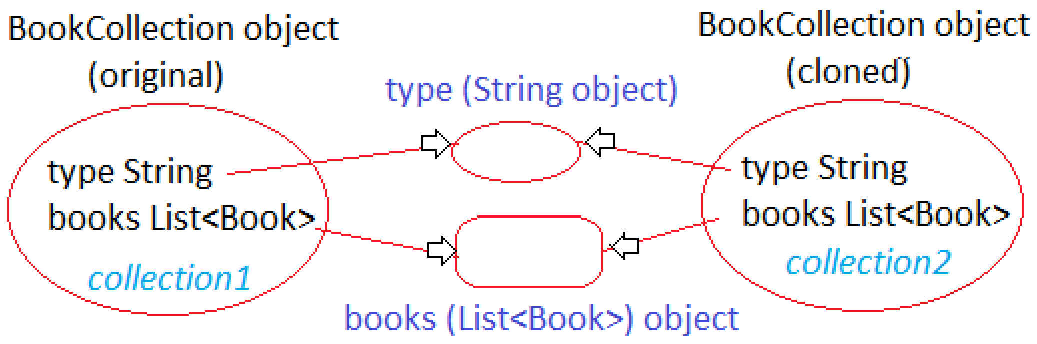
(Because every student is his own copy of 10th class books).


Class Room App
(Student 1)
1e
(Student 2)
(Student 3)
2a
1a
getting class 10th books from DB s/w
DB s/w
cache
1d
20 Book objects representing 10th class books are kept in cache
2b
1c
1b
2c
clon() -> cloned 20 Book class objects

- If we create object using new operators then the following things takes place,
    - Instantiation (object creation)
    - Object mapping (linking object with reference variable)
    - Object initialization (execution of constructor)
- If we create object using cloning process then the following things takes place,
    - Instantiation (object creation)
    - Object mapping (linking object with reference variable)
    - No object initialization (no execution of constructor) Here new object will be initialization having existing object as the initial data.

- To implement Prototype design pattern, we need to go for caching + cloning at prototype registry (nothing but the prototype factory).
- Cloning can be done in two ways
    - Shallow cloning
    - Deep cloning

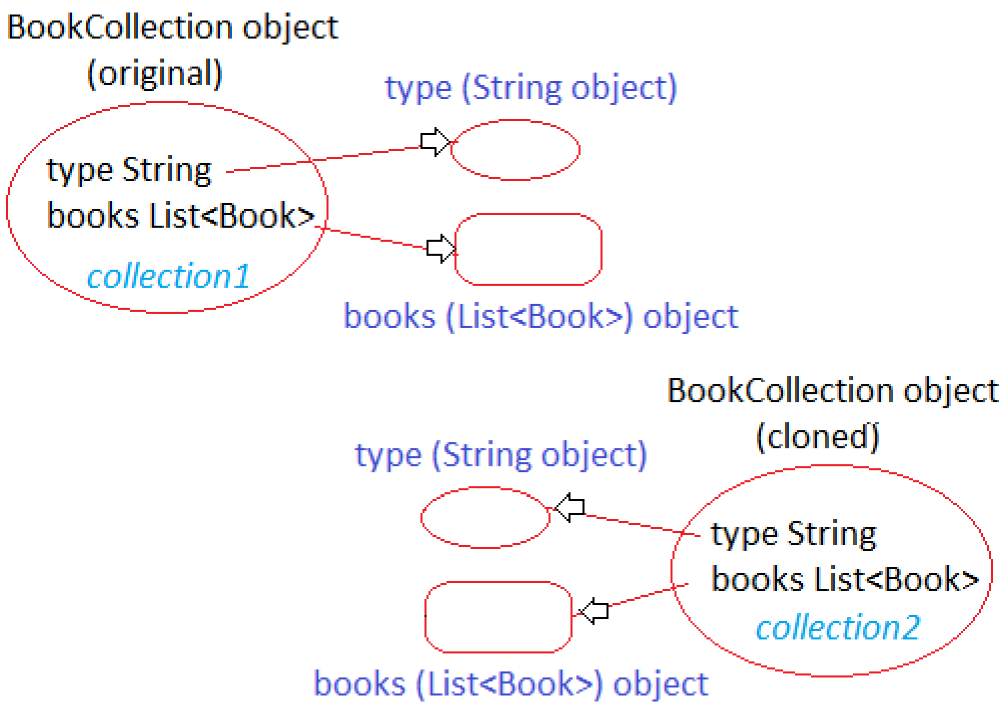
Shallow Cloning: Only outer object will be cloned not the inner object.



```
BookCollection collection2 = (BookCollection) collection1.clone();
```

- By default, clone () method will do shallow cloning and shallow cloning is not suitable for Prototype design pattern.

Deep Cloning: Both outer object and inner properties (objects) will be cloned neatly.



```
BookCollection collection2 = (BookCollection) collectionl.clone();
String otype_=collection1.getType();
String ctype = new String(otype);
List<Book> oBooks = collection1.getBooks();
List<Book> cBooks= new ArrayList();
cBooks.removeAll();
//If Book class object is not cloneable
for (Book b:oBooks) {
        Book b1=new Book(b.getBookld(), b.getBookName());
        cBooks.add(b1);
}
//If Book class object is cloneable
for (Book b:oBooks) {
        Book b1= (Book) b.clone();
        cBooks.add(b1);
}
```

```
collection2.setType(ctype);
collection2.setBooks(cBooks);
```

Directory Structure of DPProj20-PrototypeDP-Solution-ShallowCloning:

- Copy & paste the DPProj19-PrototypeDP-Problem project and rename to DPProj20-PrototypeDP-Solution-ShallowCloning.
- No extra class or package added.
- Then use the following code with in their respective file.

BookCollection.java

```
package com.sahu.component;

import java.util.ArrayList;
import java.util.List;

import com.sahu.factory.BookType;
import com.sahu.model.Book;

public abstract class BookCollection implements Cloneable {
	private BookType bookType;
	private List<Book> books = new ArrayList<>();

	public BookType getBookType() {
		return bookType;
	}

	public void setBookType(BookType bookType) {
		this.bookType = bookType;
	}

	public List<Book> getBooks() {
		return books;
	}

	public void setBooks(List<Book> books) {
		this.books = books;
	}

	public abstract void loadBooks();
	@Override
	public String toString() {
		return "BookCollection [bookType=" + bookType + ", books=" +
books + "]";
	}

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

}
```

BookCollectionFactory.java

```
package com.sahu.factory;

import java.util.HashMap;
import java.util.Map;

import com.sahu.component.BookCollection;
import com.sahu.component.FictionalBooksCollection;
import com.sahu.component.RealityBooksCollection;

public class BookCollectionFactory {

	private static Map<BookType, BookCollection> cacheMap = new
HashMap<>();

	static {
		//In app startup itself load all types of books from DB s/w and
keep them in cache
		BookCollection fBookCollection = new
FictionalBooksCollection();
		fBookCollection.loadBooks();
		BookCollection rBookCollection = new RealityBooksCollection();
		rBookCollection.loadBooks();

		//Keep BookCollection in cache
		cacheMap.put(BookType.FICTIONAL, fBookCollection);
		cacheMap.put(BookType.REALITY, rBookCollection);
	}

	public static BookCollection getBookCollection(BookType bookType)
throws CloneNotSupportedException {
		return (BookCollection) cacheMap.get(bookType).clone();
	}

}
```

Library.java

```
package com.sahu.test;

import com.sahu.component.BookCollection;
import com.sahu.factory.BookCollectionFactory;
import com.sahu.factory.BookType;

public class Library {

	public static void main(String[] args) throws
CloneNotSupportedException {
		BookCollection fBooks =
BookCollectionFactory.getBookCollection(BookType.FICTIONAL);
		System.out.println(fBooks);
		System.out.println("------------------");
		BookCollection fBooks1 =
BookCollectionFactory.getBookCollection(BookType.FICTIONAL);
		System.out.println(fBooks1);

		//remove one book using fbook1 from List collection
		// Because of shallow cloning, the change will reflect to fbooks
which is against of prototype design pattern
		fBooks1.getBooks().remove(0);
		System.out.println(fBooks.getBooks().size()+"
"+fBooks1.getBooks().size());

		System.out.println("*****************");
		BookCollection rBooks =
BookCollectionFactory.getBookCollection(BookType.REALITY);
		System.out.println(rBooks);
		System.out.println("------------------");
```

```
                    BookCollection rBooks1 =
BookCollectionFactory.getBookCollection(BookType.REALITY);
                    System.out.println(rBooks1);
          }

}
```

Note: The above is not perfect implementation of Prototype design pattern, because still multiple set of same category books referring to set of List<Book> collection because of shallow cloning.

Directory Structure of DPProj21-PrototypeDP-Solution-DeepCloning:

- Copy & paste the DPProj20-PrototypeDP-Solution-ShallowCloning project and rename to DPProj21-PrototypeDP-Solution-DeepCloning.
- No extra class or package added.
- Then use the following code with in their respective file.

Book.java

```
package com.sahu.model;

public class Book implements Cloneable {
          private int bookId;
          private String bookName;

          public Book(int bookId, String bookName) {
                    System.out.println("Book.Book()");
                    this.bookId = bookId;
                    this.bookName = bookName;
          }

          public int getBookId() {
                    return bookId;
          }

          public void setBookId(int bookId) {
                    this.bookId = bookId;
          }

          public String getBookName() {
```

```
			return bookName;
		}

		public void setBookName(String bookName) {
			this.bookName = bookName;
		}

		@Override
		public String toString() {
			return "Book [bookId=" + bookId + ", bookName=" + bookName
+ "]";
		}

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

}
```

BookCollectionFactory.java

```
package com.sahu.factory;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import com.sahu.component.BookCollection;
import com.sahu.component.FictionalBooksCollection;
import com.sahu.component.RealityBooksCollection;
import com.sahu.model.Book;

public class BookCollectionFactory {

		private static Map<BookType, BookCollection> cacheMap = new
HashMap<>();

		static {
			//In app startup itself load all types of books from DB s/w and
keep them in cache
		BookCollection fBookCollection = new
FictionalBooksCollection();
		fBookCollection.loadBooks();
		BookCollection rBookCollection = new RealityBooksCollection();
		rBookCollection.loadBooks();

		//Keep BookCollection in cache
		cacheMap.put(BookType.FICTIONAL, fBookCollection);
		cacheMap.put(BookType.REALITY, rBookCollection);
	}

	public static BookCollection getBookCollection(BookType bookType)
throws CloneNotSupportedException {
		BookCollection originalBookCollection =
cacheMap.get(bookType);
		BookCollection clonedBookCollection = (BookCollection)
originalBookCollection.clone();
		BookType originalBookType =
originalBookCollection.getBookType();
		BookType cloneBookType = originalBookType;
		List<Book> originalBooks = originalBookCollection.getBooks();
		List<Book> cloneBooks = new ArrayList<>();
		for (Book originalBook : originalBooks) {
			Book cloneBook = (Book) originalBook.clone();
			cloneBooks.add(cloneBook);
		}
		//set everything to clonedBookCollection
		clonedBookCollection.setBookType(cloneBookType);
		clonedBookCollection.setBooks(cloneBooks);

		return clonedBookCollection;
	}

}
```

Prototype Design Pattern (recognizable by creational methods returning an instance of itself with the same properties)

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

Q. Where did u use interface and where did use abstract class in your real-time project?

Ans. Let us assume TL/ PL has decided release an API for team of developers before start of coding phase on certain module or functionality.

- If TL/ PL decides to give only rules to team members then they use interfaces having declaration of methods (here the methods declared in interfaces represent as rules because the implementation classes must provide definition for methods of interfaces).
- If TL/ PL decides to give both rules and guidelines to team members then they use abstract classes having both declarations and method definitions (here method declarations represent as rules and method definitions represent as guidelines).

Note: Both interfaces and abstract classes are there supporting loose coupling and runtime polymorphism.

Popular Java pattern catalogs are:

a. GOF patterns catalog
    a. Creational patterns
    b. Structural patterns
    c. Behavioral patterns
b. JEE patterns catalog
    a. Presentation tier patterns
    b. Business tier patterns
    c. Integration tier patterns

Presentation tier patterns: This tier contains all the presentation tier logic required to service the clients that access the system. The presentation tier design patterns are,

- Intercepting Filter
- Front Controller
- Context Object
- Application Controller
- View Helper
- Composite View
- Service to worker
- Dispatcher View

Business tier patterns: This tier provides business service required by the application clients; The Business tier design patterns are,

- Business Delegate,
- Service Locator
- Session Façade
- Application Service
- Business Object
- Composite Entity
- Transfer Object
- Transfer Object Assembler
- Value List Handler

Integration tier patterns: This tier is responsible for communicating with external resources and systems such as data stores etc. The integration tier design patterns are,

- Data Access Object
- Service Activator
- Domain Store
- Web Service Broker

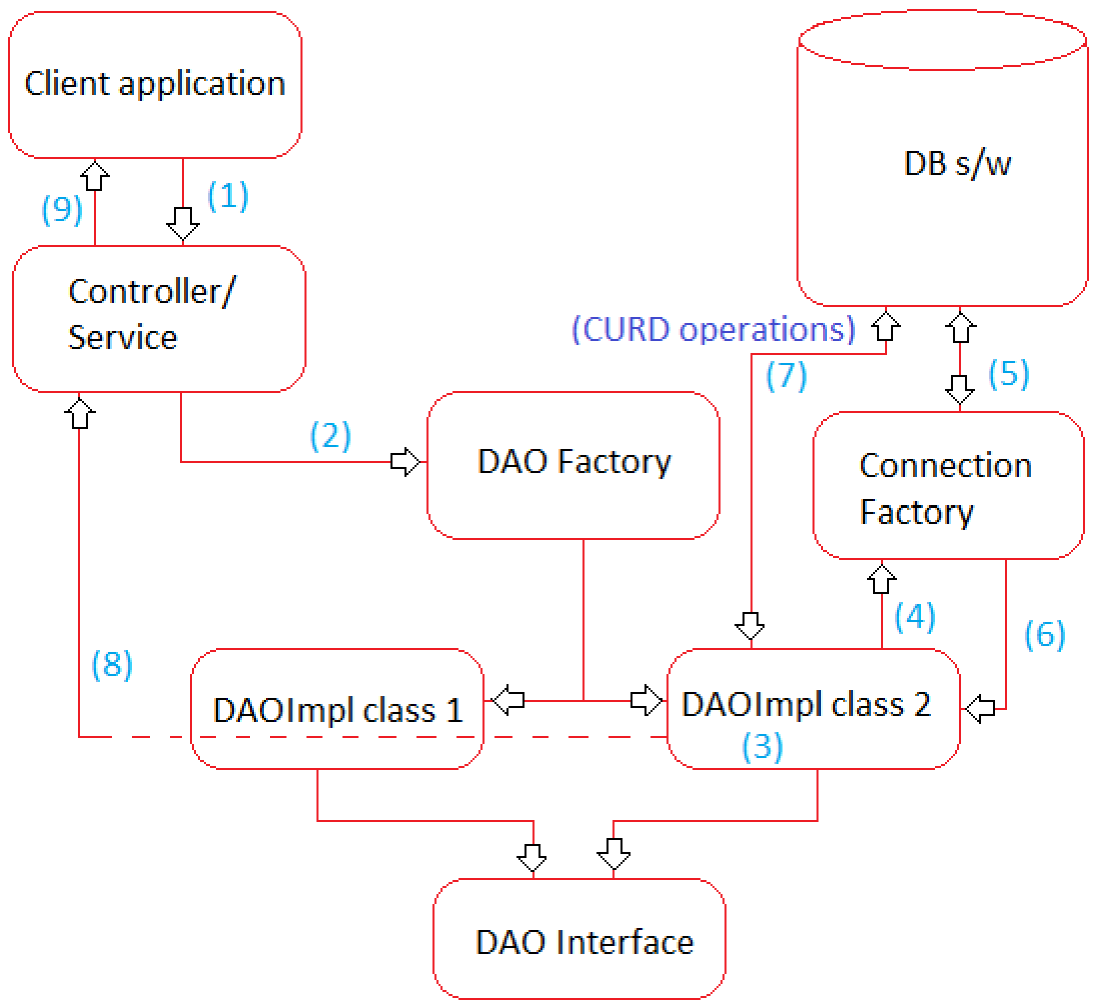
DAO (Data Access Object)

- DAO = D.A.O or DOW class (speaking wise).
- The java classes that separate persistence logic from other logics of the to make persistence logic as flexible logic to modify and reusable logics.
- DB s/w are called persistence storage technologies.
- JDBC, Hibernate, Spring JDBC, Spring ORM, JPA, Spring Data JPA and etc. are called Persistence access technologies or frameworks because using these technologies/ frameworks code we can access and modify the persistence storage technologies (DB s/w) data.
- This java class contains the code written in data access technologies/ frameworks. So, it is called DAO class.
- DAO class should contain only persistence logic not even single line of other logics (like c = a + b is also not allowed).
- DAO class contains logics to perform CURD operations on DB s/w data and it uses ConnectionFactory class as helper class to get JDBC connection and to close Connection.
- If a project is having < 100 DB tables then we take 1 DAO per 1 DB table (i.e., All CURD operations of 1 DB table will be done using 1 DAO class).
- If project is having >= 100 DB tables then we take 1 DAO per 4 or 5 related DB tables (i.e., All CURD operations of 4/ 5 DB tables will be done using 1 DAO class).
- Every DAO class contains two parts,

a. Query part
    - Declaration of all SQL queries takes place at top of the class as private static final String variables values.
    - Generally, it is recommended to take all SQL queries in upper case letters to differentiate them from Java code.
    - Better to avoid * symbol in the SQL query (specify column names).

b. Code part
    - Nothing but java method definitions of DAO class having Persistence logic.
    - Takes and uses the above declared SQL queries.
    - We can write this persistence logic using any persistence technology/ framework.
    - Can work with direct JDBC connection or pooled JDBC connection.

DAO Pattern components:



a. DAO interface (declaration of persistence methods).
b. DAO implementation class (we can have multiple implementation classes developed in multiple persistence technologies either for same DB s/w or for different DB s/w).
c. DAO Factory (returns one of the several DAO implementations classes object based on the data that is supplied)
d. ConnectionFactory (contains logic for creating and closing JDBC connection, acts as helper class to DAO class).

DAO class advantages:

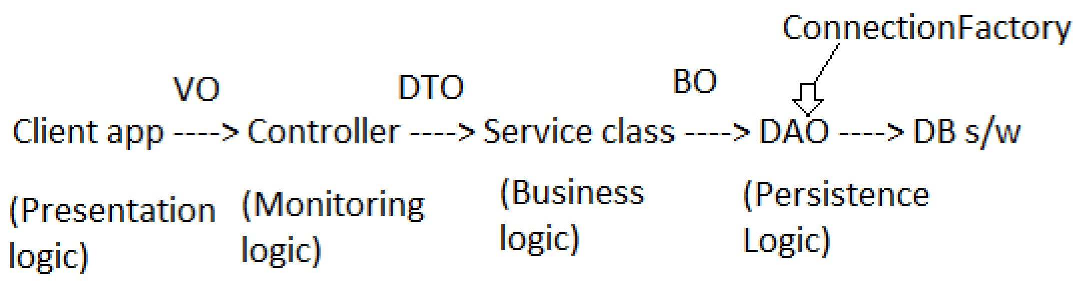
a. Makes the persistence logic as the reusable logic.
b. Separates persistence logic from other logics and makes that logic as flexible logic to modify.
c. Code maintenance becomes easy.
d. Improves the productivity.

Advantages of Layered application:

- Clean separation between logics.
- Code does not look clumsy.
- The modification done in one-layer logics does not affect other layer logics.
- Maintenance and enhancement of the project becomes easy.
- Parallel development is possible, So the productivity is good.
- It is industry standard approach.

  and etc.

Standalone layered application components:



Service class:

- Separates business logic from other logics and makes it as reusable.
- It is generally one per module.
- Gets inputs from controller as Java bean (DTO class object).
- Gives inputs to DAO class having persistable data in the form of BO class object.

Controller class:

- ○ Controller are monitors all the activities of the project.
- ○ Makes correct client app inputs are going to correct service class and vice-versa.
- ○ Gets inputs from client app as VO class object and gives to service class as DTO class object
- ○ Generally, it is 1 per project.

Client app:

- ○ Contains presentation logic to give inputs and to receive and display outputs.
- ○ Gives the inputs to controller as VO class object (generally all inputs as String values).
- ○ Gets the results from controller.

Note:

- ✓ The problems/ exceptions raised in any layer/ component/ class of the project should not be suppressed/ eaten in the same layer itself by using try-catch blocks. We should propagate the exception to next layer using exception throwing or rethrowing process across the multiple layers until it reaches to client app where end-user can receive the problem/ exception.
- ✓ For example, the exception raised in ConnectionFactory should be propagated to DAO class to Service class to Controller class to client app in order to present to end-user.

Exception propagation:

```
public EmployeeDAOImpl implements EmployeeDAO {

        public int insert (EmployeeBO bo) throws Exception {
                try {
                        …….. JDBC code that my throw exception
                        …………
                } catch (SQLException se) {
                        se.printStackTrace();
                        throw se; //exception rethrowing
                }
                catch (Exception e) {
                        e.printStacktrace();
                        throw e; //exception throwing
                }
                finally {
                        …………
                }
                return …….;
        }
}
```

Note: We replace try-catch-finally with try with resource then there is no need of writing finally block separately.

Lombok API

- Lombok API generates commonly required boiler plate code of Java class dynamically like toString(), hashCode(), setters, getters, constructor and etc.
- Lombok API Jar file - [Lombok API], lombok-1.18.24.jar

To use Lombok API in Eclipse Project:

a. Configure Lombok API with Eclipse IDE
   double click on lombok-1.18.18.jar and choose your eclipse installation location then install.
b. Add Lombok API jar file to the project build path/ class path.
c. Use the following annotations to generate boilerplate code
   - @Getter
   - @Setter
   - @Data: @Setter + @Getter +@ToString + @EqualsAndHashCode
   - @ToString
   - @EqualsAndHashCode
   - @NoArgConstructor
   - @RquiredConstructor

   and etc.

- Do not make multiple requests performing multiple persistence operations using same JDBC con object because it may raise Data inconsistency problems in multi-threaded environment.

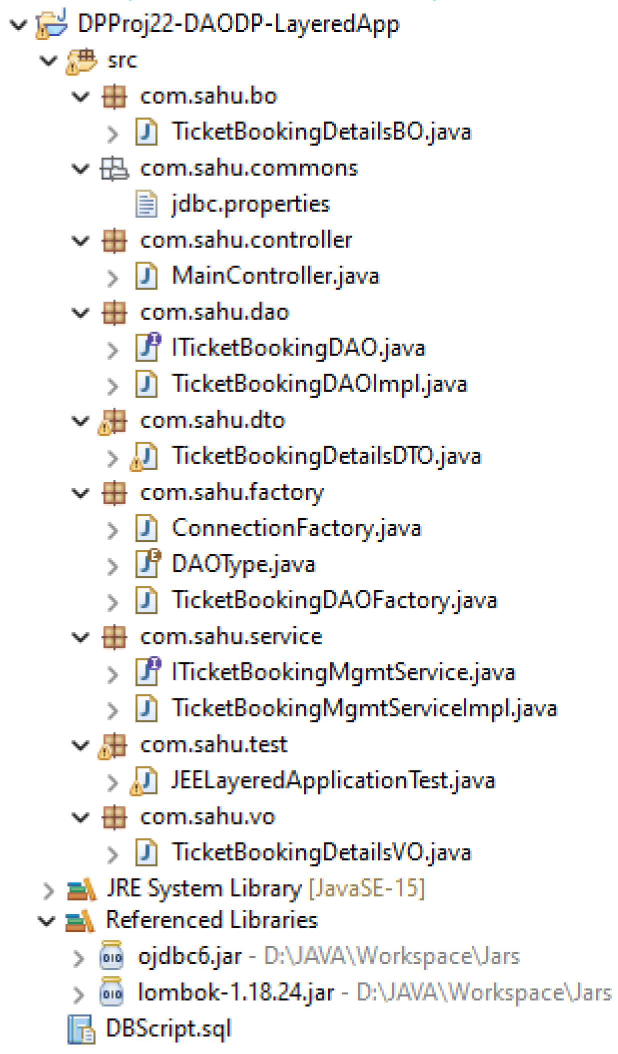
```
request1 -----> conn1 ----> insert operation
request2 -----> conn1 ----> update operation
request3 -----> conn1 ----> rollback operation
```

- If conn1.rollback() is called before committing the results request1, request2 insert, update queries execution then it would be problem.

- To solve this problem, make every request getting its own JDBC connection object to complete its persistence operation.

```
request1 -----> conn1 ----> insert operation
request2 -----> conn2 ----> update operation
request3 -----> conn3 ----> rollback operation
```

Directory Structure of DPProj22-DAODP-LayeredApp:



```
DPProj22-DAODP-LayeredApp
  src
    com.sahu.bo
      TicketBookingDetailsBO.java
    com.sahu.commons
      jdbc.properties
    com.sahu.controller
      MainController.java
    com.sahu.dao
      ITicketBookingDAO.java
      TicketBookingDAOImpl.java
    com.sahu.dto
      TicketBookingDetailsDTO.java
    com.sahu.factory
      ConnectionFactory.java
      DAOType.java
      TicketBookingDAOFactory.java
    com.sahu.service
      ITicketBookingMgmtService.java
      TicketBookingMgmtServiceImpl.java
    com.sahu.test
      JEELayeredApplicationTest.java
    com.sahu.vo
      TicketBookingDetailsVO.java
  JRE System Library [JavaSE-15]
  Referenced Libraries
    ojdbc6.jar - D:\JAVA\Workspace\Jars
    lombok-1.18.24.jar - D:\JAVA\Workspace\Jars
  DBScript.sql
```

- Develop the above directory structure package, class, properties file in src folder.
- Create the DBScript.sql file at project level and to create the table and sequence you can use SQL developer tool.
- Add the following jars in build path or class path.
    - [Lombok API], lombok-1.18.24.jar
    - ojdbc6-11.2.0.4.jar
- Then use the following code with in their respective file.

DBScript.sql

```
-- MOVIE_TICKET_BOOKING Table
CREATE TABLE "SYSTEM"."MOVIE_TICKET_BOOKING"
  ("TICKET_ID" NUMBER NOT NULL ENABLE,
        "CUST_NAME" VARCHAR2(20 BYTE),
        "TICKET_COUNT" NUMBER,
        "TYPE" VARCHAR2(20 BYTE),
        "SEAT_NO" VARCHAR2(20 BYTE),
        "BILL_AMOUNT" FLOAT(126),
        CONSTRAINT "MOVIE_TICKET_BOOKING_PK" PRIMARY KEY
("TICKET_ID"));

-- TICKET_ID_SEQ
CREATE SEQUENCE "SYSTEM"."TICKET_ID_SEQ" MINVALUE 1000
MAXVALUE 2000 INCREMENT BY 1 START WITH 1000 CACHE 20 NOORDER
NOCYCLE;
```

jdbc.properties

```
#JDBC properties
jdbc.driver=oracle.jdbc.driver.OracleDriver
jdbc.url=jdbc:oracle:thin:@localhost:1521:xe
jdbc.user=system
jdbc.password=manager
```

ConnectionFactory.java

```
package com.sahu.factory;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConnectionFactory {

        public static Connection makeConnection(String url, String userName,
String password) throws SQLException {
                Connection conn = null;
                try {
                        conn = DriverManager.getConnection(url,userName,
password);
```

```
                } catch (SQLException e) {
                        e.printStackTrace();
                        throw e;
                }

                return conn;
        }

}
```

ITicketBookingDAO.java

```
package com.sahu.dao;

import com.sahu.bo.TicketBookingDetailsBO;

public interface ITicketBookingDAO {
        public int insert(TicketBookingDetailsBO bo) throws Exception;
}
```

TicketBookingDAOImpl.java

```
package com.sahu.dao;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.Properties;

import com.sahu.bo.TicketBookingDetailsBO;
import com.sahu.factory.ConnectionFactory;

public class TicketBookingDAOImpl implements ITicketBookingDAO {

        private static final String INSERT_MOVIE_TICKERT_BOOKING_QUERY
= "INSERT INTO MOVIE_TICKET_BOOKING VALUES
(TICKET_ID_SEQ.NEXTVAL, ?, ?, ?, ?, ?)";

```

```
	private static Properties properties;

	static {
		try {
			//load properties file
			InputStream is = new
FileInputStream("src/com/sahu/commons/jdbc.properties");
			//store properties file into java.lang.Properties class
object
			properties = new Properties();
			properties.load(is);
		} catch (IOException e) {
			e.printStackTrace();
		}
	}

	@Override
	public int insert(TicketBookingDetailsBO bo) throws Exception {
		Connection conn = null;
		PreparedStatement pStatement = null;
		int count = 0;
		try {
			//get JDBC connection using connection factory
			conn =
ConnectionFactory.makeConnection(properties.getProperty("jdbc.url"),
properties.getProperty("jdbc.user"),
properties.getProperty("jdbc.password"));
			//create PreparedStatement object
			pStatement =
conn.prepareStatement(INSERT_MOVIE_TICKERT_BOOKING_QUERY);
			//set values to Query params
			pStatement.setString(1, bo.getCustName());
			pStatement.setInt(2, bo.getTicketCount());
			pStatement.setString(3, bo.getTicketType());
			pStatement.setString(4, bo.getSeatNo());
			pStatement.setFloat(5, bo.getBillAmount());

			//Execute the query
			count = pStatement.executeUpdate();
```

```
				} catch (SQLException e) {
					e.printStackTrace();
					throw e;
				}
				finally {
					try {
						if (pStatement!=null)
							pStatement.close();
					} catch (SQLException e) {
						e.printStackTrace();
						throw e;
					}
					try {
						if (conn!=null)
							conn.close();
					} catch (SQLException e) {
						e.printStackTrace();
						throw e;
					}
				}

				return count;
		}

}
```

DAOType.java

```
package com.sahu.factory;

public enum DAOType {
	JDBC;
}
```

TicketBookingDAOFactory.java

```
package com.sahu.factory;

import com.sahu.dao.ITicketBookingDAO;
import com.sahu.dao.TicketBookingDAOImpl;
```

```java
public class TicketBookingDAOFactory {

	public static ITicketBookingDAO getInstance(DAOType daoType) {
		ITicketBookingDAO dao = null;
		if (daoType.equals(DAOType.JDBC))
			dao = new TicketBookingDAOImpl();

		return dao;
	}

}
```

TicketBookingDetailsBO.java

```java
package com.sahu.bo;

import lombok.Data;

@Data
public class TicketBookingDetailsBO {
	private String custName;
	private Integer ticketCount;
	private String seatNo;
	private String ticketType;
	private Float billAmount;
}
```

TicketBookingDetailsDTO.java

```java
package com.sahu.dto;

import java.io.Serializable;

import lombok.Data;

@Data
public class TicketBookingDetailsDTO implements Serializable {
	private String custName;
	private Integer ticketCount;
	private String seatNo;
	private String ticketType;
}
```

TicketBookingDetailsVO.java

```java
package com.sahu.vo;

import lombok.Data;

@Data
public class TicketBookingDetailsVO {
	private String custName;
	private String ticketCount;
	private String seatNo;
	private String ticketType;
}
```

ITicketBookingMgmtService.java

```java
package com.sahu.service;

import com.sahu.dto.TicketBookingDetailsDTO;

public interface ITicketBookingMgmtService {
	public String bookTickets(TicketBookingDetailsDTO dto) throws 
Exception;
}
```

TicketBookingMgmtServiceImpl.java

```java
package com.sahu.service;

import com.sahu.bo.TicketBookingDetailsBO;
import com.sahu.dao.ITicketBookingDAO;
import com.sahu.dto.TicketBookingDetailsDTO;
import com.sahu.factory.DAOType;
import com.sahu.factory.TicketBookingDAOFactory;

public class TicketBookingMgmtServiceImpl implements 
ITicketBookingMgmtService {

	private ITicketBookingDAO dao = null;

	public TicketBookingMgmtServiceImpl() {
		dao = TicketBookingDAOFactory.getInstance(DAOType.JDBC);
	}

	@Override
	public String bookTickets(TicketBookingDetailsDTO dto) throws
Exception {
		//calculate bill amount
		float price = 0.0f;
		if(dto.getTicketType().equalsIgnoreCase("reclainer"))
			price = 200.0f;
		else
			price = 150.0f;

		float billAmount = dto.getTicketCount()*price;

		//prepare BO class object having persistence data
		TicketBookingDetailsBO bo = new TicketBookingDetailsBO();
		bo.setCustName(dto.getCustName());
		bo.setSeatNo(dto.getSeatNo());
		bo.setTicketCount(dto.getTicketCount());
		bo.setTicketType(dto.getTicketType());
		bo.setBillAmount(billAmount);

		//use DAO
		int count = dao.insert(bo);

		return count==0?"Tickets are not booking":"Tickets has
booked";
	}

}
```

MainController.java

```
package com.sahu.controller;

import com.sahu.dto.TicketBookingDetailsDTO;
import com.sahu.service.ITicketBookingMgmtService;
import com.sahu.service.TicketBookingMgmtServiceImpl;
import com.sahu.vo.TicketBookingDetailsVO;

public class MainController {
	private ITicketBookingMgmtService service;

	public MainController() {
		service = new TicketBookingMgmtServiceImpl();
	}

	public String processTicketBooking(TicketBookingDetailsVO vo) 
throws Exception {
		//Convert VO class object to dto class object
		TicketBookingDetailsDTO dto = new 
TicketBookingDetailsDTO();
		dto.setCustName(vo.getCustName());
		dto.setSeatNo(vo.getSeatNo());
		dto.setTicketCount(Integer.parseInt(vo.getTicketCount()));
		dto.setTicketType(vo.getTicketType());

		//user service
		return service.bookTickets(dto);
	}

}
```

JEELayeredApplicationTest.java

```
package com.sahu.test;

import java.sql.SQLException;
import java.util.Scanner;

import com.sahu.controller.MainController;
import com.sahu.vo.TicketBookingDetailsVO;

public class JEELayeredApplicationTest {

	public static void main(String[] args) {
		//read inputs and store them in VO class object
		Scanner sc = new Scanner(System.in);
		System.out.print("Enter customer name : ");
		String name = sc.next();
		System.out.print("Enter tickets count : ");
		int ticketCount = sc.nextInt();
```

```
                    System.out.print("Choose tickets type : ");
                    String ticketType = sc.next();
                    System.out.println("Choose seat nos ----- ");
                    String seatNo = "";
                    for (int i = 1; i <=ticketCount; i++) {
                              System.out.print("Enter "+i+" seat no: ");
                              seatNo = seatNo+", "+sc.next();
                    }

                    //prepare VO class object
                    TicketBookingDetailsVO detailsVO = new
TicketBookingDetailsVO();
                    detailsVO.setCustName(name);
                    detailsVO.setTicketType(ticketType);
                    detailsVO.setTicketCount(String.valueOf(ticketCount));
                    detailsVO.setSeatNo(seatNo);

                    //invoke methods by creating object from controller
                    MainController controller = new MainController();
                    try {
                              String result =
controller.processTicketBooking(detailsVO);
                              System.out.println("\n"+result);
                    } catch (SQLException e) {
                              e.printStackTrace();
                              System.err.println("Internal DB problem :
"+e.getMessage());
                    } catch (Exception e) {
                              e.printStackTrace();
                              System.err.println("Internal Problem : "+e.getMessage());
                    }
          }

}
```

Logging

- The process of keeping track of components involved in the execution of application is called Logging.
- Logging gives the classes, methods, statements that are involved in execution of code.

Note: Writing S.o.p(-) indicating which task/ job is completed is also one kind of logging.

Q. What is the difference between auditing and logging?
Ans. Logging keeps track of the components involved in the execution of the application whereas auditing is some kind of logging which keeps track of end-user activities in the application execution.

Q. what is the difference between logging and debugging?
Ans.

- Debugging gives visual representation of line-by-line execution to see the code flow and to fix the bugs. For this we need debugger which is generally part of IDE.
- Logging records flow of execution and that can be used even after execution of the of the application (after few days or months or years) to analyze the flow of the application on certain day and time.

After releasing project

```
[Offsite/ offshore location                         [Onsite/ onshore location
        offshore team]                                      onshore team]
Infosys <--------------------------------------------------> Citi bank (client organization)
(HYD)                                                       (London)
```

Limitations with S.o.p(-) for logging:

- Can write the generated log messages only console which cannot be there for long time.
- Cannot categorize log messages.
- Cannot format log messages.
- Cannot apply filters while retrieving the log messages.
- S.o.p writes the log messages to log files as single threaded process. So, if multiple log messages are generated at a time, there will be delay writing to console.

- To overcome all these problems, use different Logging APIs/ frameworks like Log4J (best), Log Back, SLF4Jand etc.

Log4J (Logging for Java)

- Solves all the problems of System.out.println(-) based logging and simplifies logging activities in Java applications.

Advantages of Log4J:

- Allows to write the generated log messages to different destinations like console, file, DB s/w, mail servers and etc.
- Allows to format the generated log messages having different layouts like HTML layout, XML layout, Custom layout (Pattern layout), Simple layout and etc.
- Allows to categorize the log messages

  DEBUG < INFO < WARN < ERROR < FATAL

  - DEBUG: indicates flow is encountered in the current like "in that start of main method", "in the end of main (-) method", "entered into business method", "exited from business method" and etc.
  - INFO: indicates some concreate operation is taken place "connection is established", "ResultSet Processed", " Connection closed".
  - WARN: deprecated code is executed in urgency of released, so marked as WARNNING to replace later.
  - ERROR: indicates some known exception raised, generally we place it in catch blocks with known exception class names.
  - FATAL: indicates some unknown problem/ exception raised, generally we place in catch blocks with Exception or Throwable class names.
- Allows to filter the log messages that are generated.

  ALL < DEBUG < INFO < WARN < ERROR < FATAL < OFF

  - If logger level to retrieve messages is DEBUG, then we get DEBUG, INFO, WARN, ERROR, FATAL log messages.
  - If logger level to retrieve messages is WARN then we get WARN, ERROR, FATAL (>=WARN and <=FATAL) log messages.
  - If logger level to retrieve messages is ALL then we get all log messages.
  - If logger level to retrieve messages is OFF then logging will be disabled.
- Log4J can write log messages in multi-threaded environment.
- It is industry standard for logging.

  and etc.

3 important objects of Log4J programming:

a. Logger object: Enables Logging on given java class and allows to render different levels of log messages.

   E.g.:

   Logger log = Logger.getLogger(<Current class>);

```
log.debug(".....");
log.info("......");
log.warn("... ");
```

Note: Logger is a singleton java class.

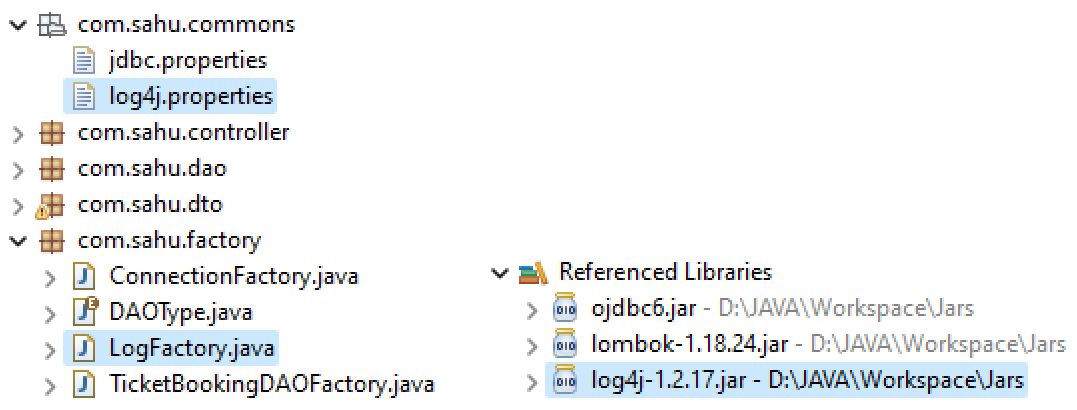
b. Appender object: Specifies the destination to record/ write the log messages.
    - ConsoleAppender
    - FileAppender
    - IMapAppender (To write as email message)
    - RollingFileAppender (a.txt, a_l.txt, a_2.txt, ...) (Creates backup log files based on the given max file size).
    - DailyingRollingFileAppender (creates the backup files on monthly basis or weekly basis, daily basis, hourly basis, minute basis and etc.).
    - JdbcAppender (to write to DB s/w).

    and etc.

c. Layout object: Specifies the format of the log messages.
    - SimpleLayout
    - HtmlLayout
    - XmlLayout
    - PatternLayout

    and etc.

Note: All these inputs can be given to Log4J either using properties file (recommended) or using XML file



```
com.sahu.commons
    jdbc.properties
    log4j.properties
com.sahu.controller
com.sahu.dao
com.sahu.dto
com.sahu.factory
    ConnectionFactory.java
    DAOType.java
    LogFactory.java
    TicketBookingDAOFactory.java
```

```
Referenced Libraries
    ojdbc6.jar - D:\JAVA\Workspace\Jars
    lombok-1.18.24.jar - D:\JAVA\Workspace\Jars
    log4j-1.2.17.jar - D:\JAVA\Workspace\Jars
```

- Add the log4j.properties in com.sahu.commons package and LogFactory.java in com.sahu.factory.

- Add the log4j to the class path or build path.
  log4j-1.2.17.jar
- Then place the following code with their respective files.

JEELayeredApplicationTest.java

```
#Log4J DailyRolling log files configuration
log4j.rootLogger=ALL, NS
log4j.appender.NS=org.apache.log4j.DailyRollingFileAppender
log4j.appender.NS.File=jee_logs.html
log4j.appender.NS.DatePattern='.'yyyy-MM-dd-HH-mm
log4j.appender.NS.layout=org.apache.log4j.HTMLLayout
```

TicketBookingDAOImpl.java

```
package com.sahu.dao;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.Properties;

import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;

import com.sahu.bo.TicketBookingDetailsBO;
import com.sahu.factory.ConnectionFactory;

public class TicketBookingDAOImpl implements ITicketBookingDAO {

	private static final String INSERT_MOVIE_TICKERT_BOOKING_QUERY
= "INSERT INTO MOVIE_TICKET_BOOKING VALUES
(TICKET_ID_SEQ.NEXTVAL, ?, ?, ?, ?, ?)";

	private static Properties properties;

	private static Logger logger =
Logger.getLogger(TicketBookingDAOImpl.class);
Logger.getLogger(TicketBookingDAOImpl.class);

	static {
		try {
			//Activate log4j by loading its log4j.properties file

	PropertyConfigurator.configure("src/com/sahu/commons/log4j.prop
erties");
			logger.debug("Log4j activated");
			//load properties file
			InputStream is = new
FileInputStream("src/com/sahu/commons/jdbc.properties");
			logger.debug("jdbc.properties file loaded");
			//store properties file into java.lang.Properties class
object
			properties = new Properties();
			properties.load(is);
			logger.debug("jdbc.properties file info is copied to
java.util.Properties class object");
		} catch (IOException e) {
			logger.error("Problem in locating jdbc.properties file");
			e.printStackTrace();
		}
	}

	@Override
	public int insert(TicketBookingDetailsBO bo) throws Exception {
		logger.debug("Inside insert() method");
		Connection conn = null;
		PreparedStatement pStatement = null;
		int count = 0;
		try {
			//get JDBC connection using connection factory
			conn =
ConnectionFactory.makeConnection(properties.getProperty("jdbc.url"),
properties.getProperty("jdbc.user"),
properties.getProperty("jdbc.password"));
			logger.info("JDBC connection is gathered from JDBC
Factory");
			//create PreparedStatement object
			pStatement = 
conn.prepareStatement(INSERT_MOVIE_TICKERT_BOOKING_QUERY);
			logger.info("PrepareStatement object is created having 
pre-compiled SQL query");
			//set values to Query params
			pStatement.setString(1, bo.getCustName());
			pStatement.setInt(2, bo.getTicketCount());
			pStatement.setString(3, bo.getTicketType());
			pStatement.setString(4, bo.getSeatNo());
			pStatement.setFloat(5, bo.getBillAmount());
			logger.debug("Values are set to pre-compiled SQL query 
parameters");

			//Execute the query
			count = pStatement.executeUpdate();
			logger.debug("Pre-compiled SQL query is executed");
		} catch (SQLException e) {
			logger.error("Problem in JDBC code execution");
			e.printStackTrace();
			throw e;
		} catch(Exception e) {
			logger.fatal("Unknown problem in JDBC code 
execution");
			e.printStackTrace();
			throw e;
		}
		finally {
			try {
				if (pStatement!=null)
					pStatement.close();
			} catch (SQLException e) {
				e.printStackTrace();
				throw e;
			}
			try {
				if (conn!=null)
					conn.close();
			} catch (SQLException e) {
				e.printStackTrace();
				throw e;
```

```
                    }
                }

                return count;
        }

}
```

Note: Instead of writing the logger logic in every class we can create a factory from there we can get the logger object.

LogFactory.java

```
package com.sahu.factory;

import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;

public class LogFactory {

        public static Logger createLoggerObject(Class<?> clazz) {
                Logger logger = Logger.getLogger(clazz);
                try {
                        //Activate log4j by loading its log4j.properties file

        PropertyConfigurator.configure("src/com/sahu/commons/log4j.prop
erties");
                        logger.debug("Log4j activated");
                } catch (Exception e) {
                        e.printStackTrace();
                }
                return logger;
        }

}
```

ConnectionFactory.java

```
package com.sahu.factory;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

import org.apache.log4j.Logger;

public class ConnectionFactory {

	private static Logger logger =
LogFactory.createLoggerObject(ConnectionFactory.class);

	public static Connection makeConnection(String url, String userName,
String password) throws SQLException {
		logger.debug("Inside makeConnection(-,-,-) method");
		Connection conn = null;
		try {
			conn = DriverManager.getConnection(url,userName,
password);
			logger.info("Cpnnection is established");
		} catch (SQLException e) {
			logger.error("Problem in connection establishment
"+e.getMessage());
			e.printStackTrace();
			throw e;
		}

		return conn;
	}

}
```

TicketBookingDAOFactory.java

```
package com.sahu.factory;

import org.apache.log4j.Logger;

import com.sahu.dao.ITicketBookingDAO;
import com.sahu.dao.TicketBookingDAOImpl;

public class TicketBookingDAOFactory {
	private static Logger logger =
LogFactory.createLoggerObject(TicketBookingDAOFactory.class);

	public static ITicketBookingDAO getInstance(DAOType daoType) {
		logger.debug("inside getInstance(-) method");
		ITicketBookingDAO dao = null;
		if (daoType.equals(DAOType.JDBC))
			dao = new TicketBookingDAOImpl();

		logger.info("Creating and returning DAO class object");
		return dao;
	}

}
```

TicketBookingMgmtServiceImpl.java

```
package com.sahu.service;

import org.apache.log4j.Logger;

import com.sahu.bo.TicketBookingDetailsBO;
import com.sahu.dao.ITicketBookingDAO;
import com.sahu.dto.TicketBookingDetailsDTO;
import com.sahu.factory.DAOType;
import com.sahu.factory.LogFactory;
import com.sahu.factory.TicketBookingDAOFactory;

public class TicketBookingMgmtServiceImpl implements
ITicketBookingMgmtService {

	private ITicketBookingDAO dao = null;

	private static Logger logger =
LogFactory.createLoggerObject(TicketBookingMgmtServiceImpl.class);

	public TicketBookingMgmtServiceImpl() {
		dao = TicketBookingDAOFactory.getInstance(DAOType.JDBC);
		logger.debug("DAO class object is gathered");
	}
	@Override
	public String bookTickets(TicketBookingDetailsDTO dto) throws
Exception {
		logger.debug("inside bookTickets(-) method");
		//calculate bill amount
		float price = 0.0f;
		if(dto.getTicketType().equalsIgnoreCase("reclainer"))
			price = 200.0f;
		else
			price = 150.0f;

		float billAmount = dto.getTicketCount()*price;
		logger.debug("Business logic executed and bill amount
calculated");
		//prepare BO class object having persistence data
		TicketBookingDetailsBO bo = new TicketBookingDetailsBO();
		bo.setCustName(dto.getCustName());
		bo.setSeatNo(dto.getSeatNo());
		bo.setTicketCount(dto.getTicketCount());
		bo.setTicketType(dto.getTicketType());
		bo.setBillAmount(billAmount);
		logger.debug("BO class object is created");

		//use DAO
		int count = dao.insert(bo);
		logger.info("DAO class insert(-) method invoked");

		return count==0?"Tickets are not booking":"Tickets has
booked";
	}

}
```

MainController.java

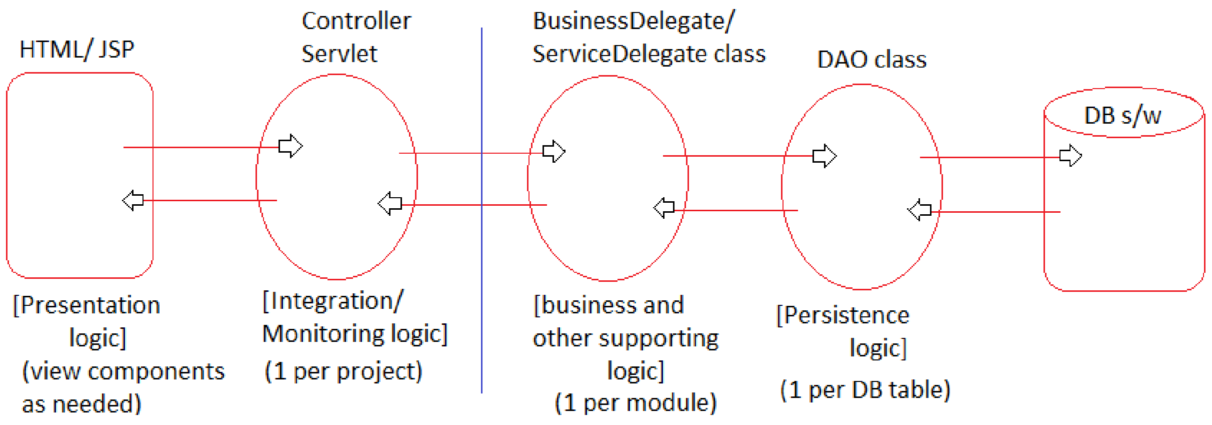
```
package com.sahu.controller;

import org.apache.log4j.Logger;

import com.sahu.dto.TicketBookingDetailsDTO;
import com.sahu.factory.LogFactory;
import com.sahu.service.ITicketBookingMgmtService;
import com.sahu.service.TicketBookingMgmtServiceImpl;
import com.sahu.vo.TicketBookingDetailsVO;

public class MainController {

	private ITicketBookingMgmtService service;

	private static Logger logger =
LogFactory.createLoggerObject(MainController.class);

	public MainController() {
		service = new TicketBookingMgmtServiceImpl();
		logger.info("service class object created");
	}

	public String processTicketBooking(TicketBookingDetailsVO vo)
throws Exception {
		logger.debug("inside processTicketBooking(-) method");
		//Convert VO class object to dto class object
		TicketBookingDetailsDTO dto = new
TicketBookingDetailsDTO();
		dto.setCustName(vo.getCustName());
		dto.setSeatNo(vo.getSeatNo());
		dto.setTicketCount(Integer.parseInt(vo.getTicketCount()));
		dto.setTicketType(vo.getTicketType());
		logger.debug("VO class object is converted to DTO class
object");

		//user service
		logger.info("service class business method is used");
		return service.bookTickets(dto);
	}

}
```

JEELayeredApplicationTest.java

```
package com.sahu.test;

import java.sql.SQLException;
import java.util.Scanner;

import org.apache.log4j.Logger;

import com.sahu.controller.MainController;
import com.sahu.factory.LogFactory;
import com.sahu.vo.TicketBookingDetailsVO;

public class JEELayeredApplicationTest {

	private static Logger logger = 
LogFactory.createLoggerObject(JEELayeredApplicationTest.class);

	public static void main(String[] args) {
		logger.debug("inside main(-) method");
		//read inputs and store them in VO class object
		Scanner sc = new Scanner(System.in);
		System.out.print("Enter customer name : ");
		String name = sc.next();
		System.out.print("Enter tickets count : ");
		int ticketCount = sc.nextInt();
		System.out.print("Choose tickets type : ");
		String ticketType = sc.next();
		System.out.println("Choose seat nos ----- ");
		String seatNo = "";
		for (int i = 1; i <=ticketCount; i++) {
			System.out.print("Enter "+i+" seat no: ");
			seatNo = seatNo+", "+sc.next();
		}
		logger.debug("inputs are read from  enduser");

		//prepare VO class object
		TicketBookingDetailsVO detailsVO = new 
TicketBookingDetailsVO();
		detailsVO.setCustName(name);
		detailsVO.setTicketType(ticketType);
		detailsVO.setTicketCount(String.valueOf(ticketCount));
		detailsVO.setSeatNo(seatNo);
		logger.debug("inputs are stored into VO class object");
		//invoke methods by creating object from controller
		MainController controller = new MainController();
		logger.info("Controller class object is created");
		try {
			String result = 
controller.processTicketBooking(detailsVO);
			logger.info("business method is invoked on controller 
class object");
			System.out.println("\n"+result);
		} catch (SQLException e) {
			e.printStackTrace();
			logger.error("Internal DB problem : "+e.getMessage());
			System.err.println("Internal DB problem : 
"+e.getMessage());
		} catch (Exception e) {
			e.printStackTrace();
			logger.fatal("Unknown Problem : "+e.getMessage());
			System.err.println("Internal Problem : "+e.getMessage());
		}
		logger.debug("end of main(-) method");
	}

}
```

Business Delegate/ Service Delegate Design Pattern

Web based layered application:


HTML/ JSP
Controller Servlet
BusinessDelegate/ ServiceDelegate class
DAO class
DB s/w
[Presentation logic] (view components as needed)
[Integration/ Monitoring logic] (1 per project)
[business and other supporting logic] (1 per module)
[Persistence logic] (1 per DB table)

[Presentation tier components]
[Because these comps are responsible to take requests and to deliver responses]

[Business tier components]
[Because they reside in the background to process requests and to generates results]

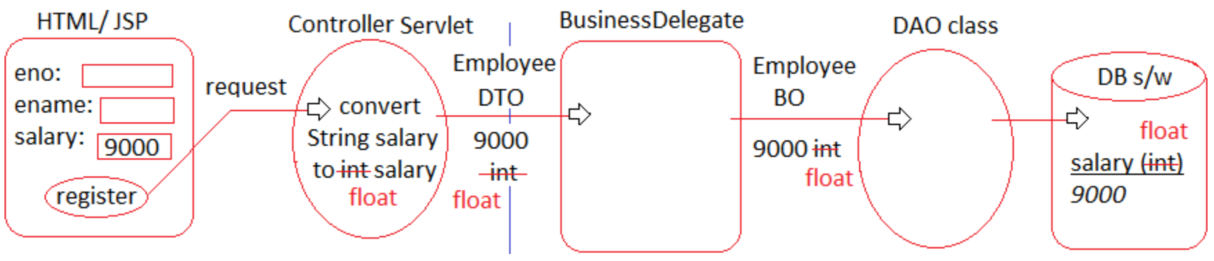
Note: We should always develop presentation-tier components and business tier components having loose coupling i.e., any change in presentation components should not disturb business tier components and vice-versa. [we can bring this effect with the support of Business delegate class]

Industry uses Business delegate design pattern in 3 angels:

a. To convert the controller component supplied VO class object that contains form data into BO class object as required for the DAO class.
b. To translate DAO raised technology specific exception to project specific user-defined exception and to propagate to Servlet component (controller).
c. To create JDBC connection and to pass to multiple DAO classes to perform Transaction Management.

To convert the controller component supplied VO class object that contains form data into BO class object as required for the DAO class

Problem:

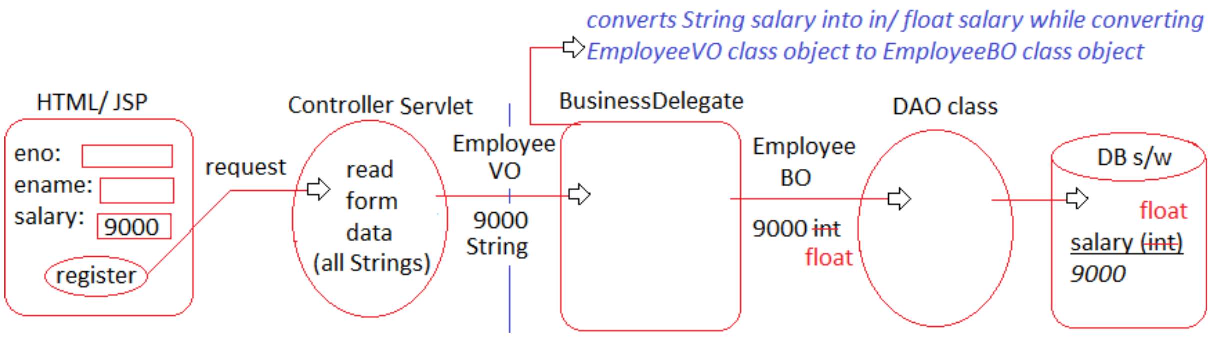


- If controller servlet component is giving form data to Business Delegate class as EmployeeDTO class object by doing necessary conversion then any change in DB table column type like changing salary column from int to float not only makes to modify the business tier DAO, BO & Business Delegate classes and it also makes us to modify to the presentation tier controller servlet component which shows tight coupling between business-tier and presentation tier components.

Solution:

- Make controller servlet component reading form data into VO class object having all String values and pass that VO class object to Business Delegate class and convert VO class object to BO class object in Business Delegate class.
- Here any change in salary column data type in DB s/w we just need to modify only Business Delegate class code which again belongs to business tier itself. i.e., there is no need doing any modification in

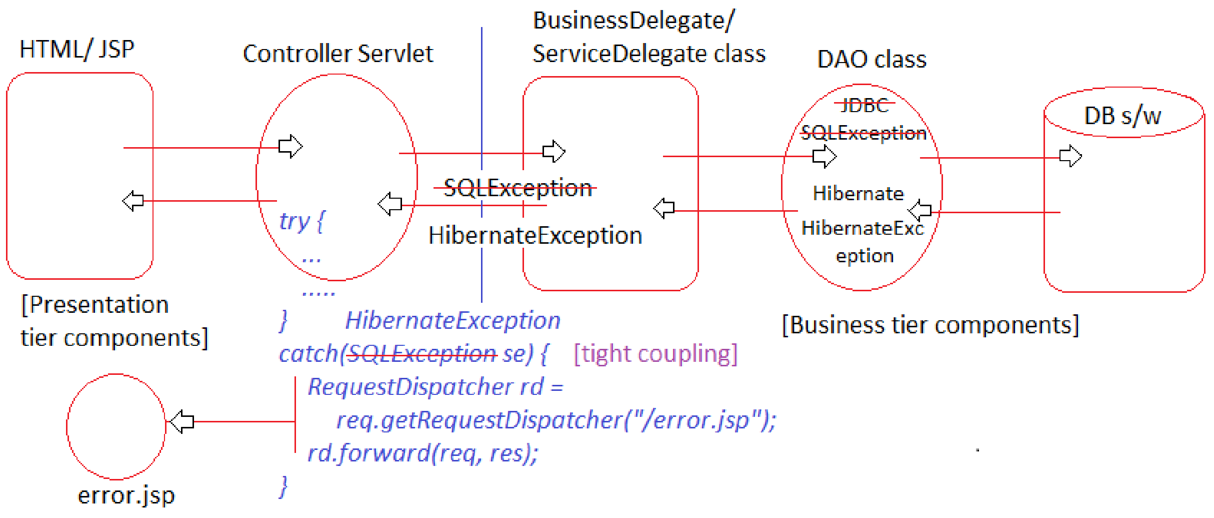
presentation tier components. This show loose coupling between presentation tier and business tier components.



Note: Since two different teams will be involved to develop presentation tier and business tier components, it is better to achieve loose coupling between teams. So, that each team should not bother about other teams' modification in their respective components.

To translate DAO raised technology specific exception to project specific user-defined exception and to propagate to Servlet component (controller)

Problem:

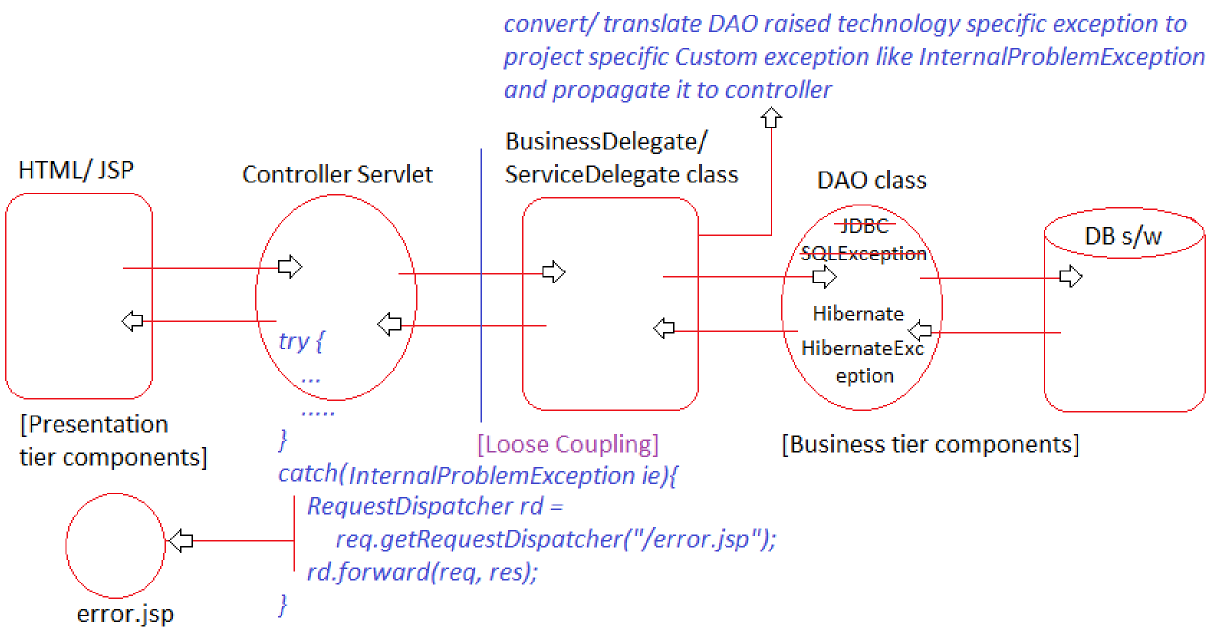


```
try {
   ...
   .....
}        HibernateException
catch(~~SQLException~~ se) {
   RequestDispatcher rd =
      req.getRequestDispatcher("/error.jsp");
   rd.forward(req, res);
}
```

Note: Since we are propagating the DAO raised technology specific exception directly to controller servlet (presentation tier) through business delegate. So, any change in DAO persistence technology we should also modify the controller servlet exception that has be handled, like SQLException (JDBC) to HibernateException (hibernate) as shown in the above diagram (This gives tight coupling between presentation tier and business tier components)

Solution:

- In Business Delegate receive the technology specific exception given by DAO class and translate or convert that exception into project specific user-defined/ custom exception and propagate that exception to controller servlet. So, that controller servlet always needs to handle project specific user-defined exception irrespective of the technology changes happens in DAO class.


convert/ translate DAO raised technology specific exception to project specific Custom exception like InternalProblemException and propagate it to controller
HTML/ JSP
Controller Servlet
BusinessDelegate/ ServiceDelegate class
DAO class
DB s/w
~~JDBC~~
~~SQLException~~
Hibernate
HibernateException
try {
...
.....
}
catch(InternalProblemException ie){
RequestDispatcher rd =
req.getRequestDispatcher("/error.jsp");
rd.forward(req, res);
}
[Presentation tier components]
[Loose Coupling]
[Business tier components]
error.jsp

To create JDBC connection and to pass to multiple DAO classes to perform Transaction Management

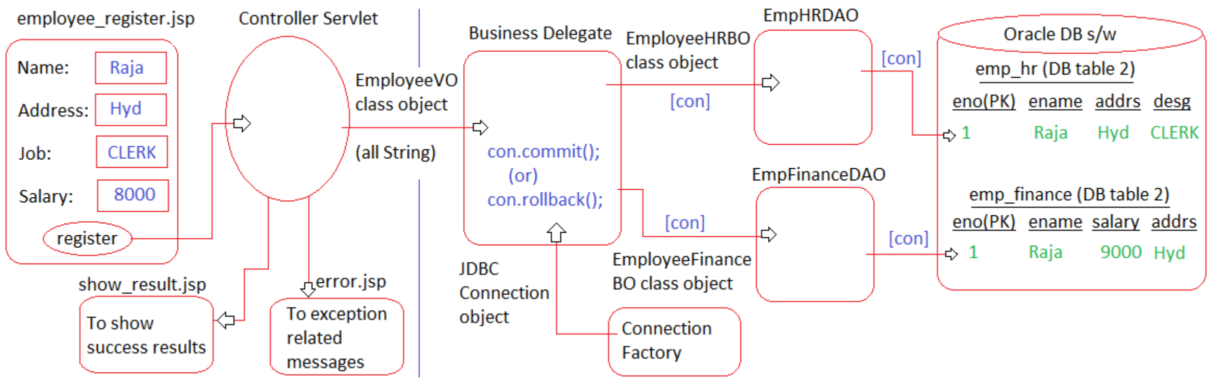
Transaction Management:

- The process of combining related operations into single unit and executing them by applying do everything or nothing principle is called Transaction Management.
- E.g.,
    - The withdraw, deposits operations of transfer money task need the support of transaction management.
    - The record insertion into HR DB table. accounts DB table towards employee registration need transaction management.

Problem: If every DAO class uses its own JDBC con object then we cannot perform transaction management in single place by checking whether both DAO classes have successfully inserted records or not.

Solution: Make Business delegate to get JDBC connection object from

Connection factory and pass same connection object to both DAO classes to perform insert persistence operations. If both persistence operations are successfully completed then go con.commit() (committing the transaction) otherwise go for con.rollback() (rollbacking the transaction).



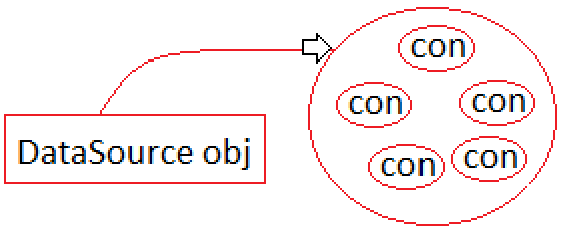
JDBC connections

a. Direct connection:

- Created and managed by the programmer using DriverManager.getConnection(-, -, -) method.

b. Pooled connection:

- It is the JDBC connection object that is gathered from JDBC connection pool.
- Here JDBC DataSource object represents JDBC connection pool and we get JDBC connection from JDBC connection pool through DataSource object.



JDBC connection pools

a. Standalone JDBC connection pool:

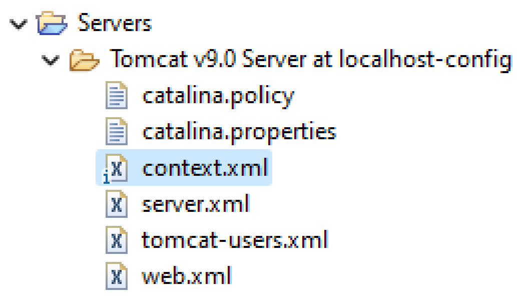
- Useful in standalone applications
- E.g., Apache DBCP, C3P0, HikariCP and etc.

b. Server Managed JDBC connection pool:

- Useful in web applications or distributed applications that are deployable in the server.

- E.g., Tomcat managed JDBC connection pool, Wildfly managed JDBC connection pool and etc.

Note: To create server managed JDBC connection pool for Oracle in Tomcat server that is configured with Eclipse IDE we need to add <Resource> under <Context> in context.xml of Tomcat server in server section of Project Explorer.



```
Servers
  Tomcat v9.0 Server at localhost-config
    catalina.policy
    catalina.properties
    context.xml
    server.xml
    tomcat-users.xml
    web.xml
```

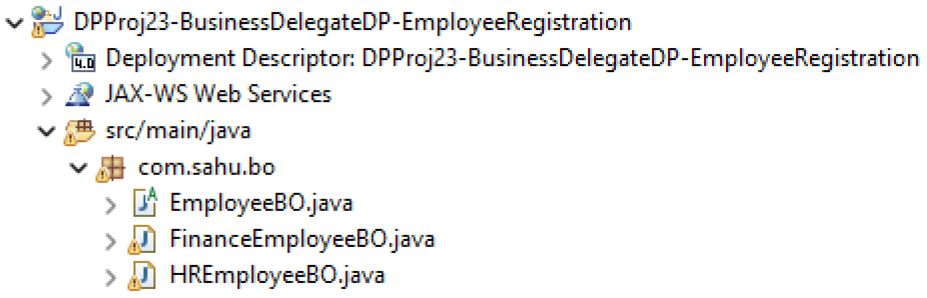
context.xml

```
<Context>
        ...............
        <Resource auth="Container"
        driverClassName="oracle.jdbc.driver.OracleDriver" maxIdle="10"
        maxTotal="20" maxWaitMillis="50000" name="DsJndi"
        password="manager" type="javax.sql.DataSource"
        url="jdbc:oracle:thin:@localhost:1521:xe" username="system"/>
</Context>
```

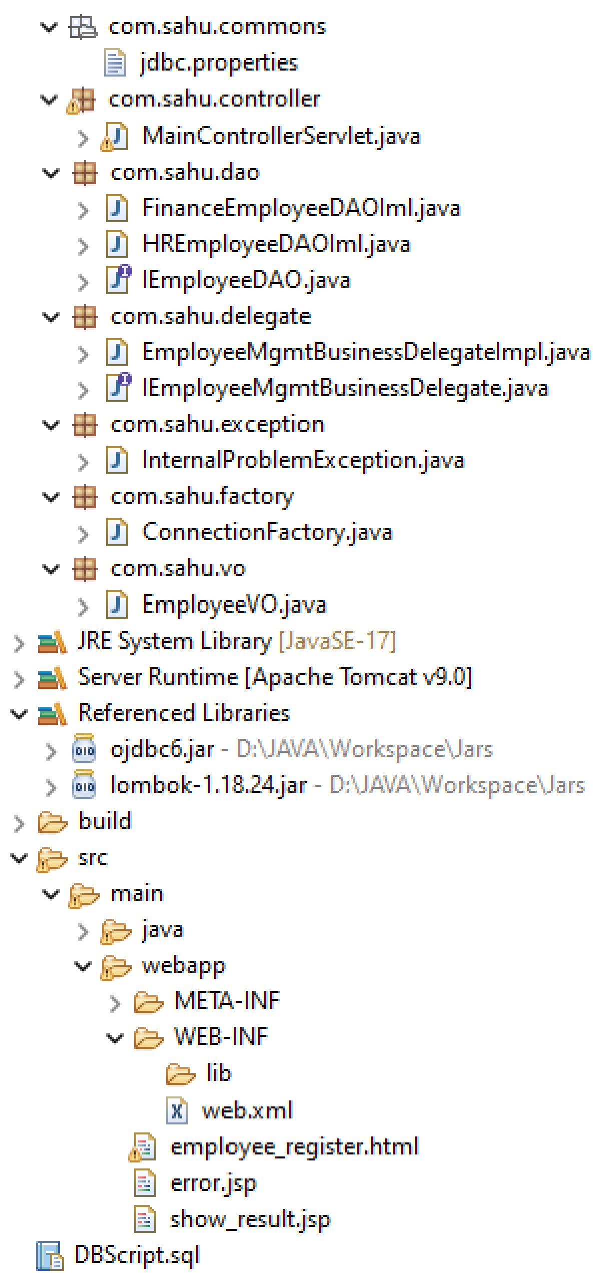
Note:

- ✓ In web application environment locating and reading properties file using IO stream and load (-) method of java.lang.Properties class is very difficult because steam can't locate WEB-INF/classes folder which is a private area of the web application.
- ✓ To overcome from this problem used java.lang.ResourceBundle class as alternative.

Directory Structure of DPProj23-BusinessDelegateDP-EmployeeRegistration:



```
DPProj23-BusinessDelegateDP-EmployeeRegistration
  > Deployment Descriptor: DPProj23-BusinessDelegateDP-EmployeeRegistration
  > JAX-WS Web Services
  src/main/java
    com.sahu.bo
      > EmployeeBO.java
      > FinanceEmployeeBO.java
      > HREmployeeBO.java
```



```
com.sahu.commons
    jdbc.properties
com.sahu.controller
    MainControllerServlet.java
com.sahu.dao
    FinanceEmployeeDAOIml.java
    HREmployeeDAOIml.java
    IEmployeeDAO.java
com.sahu.delegate
    EmployeeMgmtBusinessDelegateImpl.java
    IEmployeeMgmtBusinessDelegate.java
com.sahu.exception
    InternalProblemException.java
com.sahu.factory
    ConnectionFactory.java
com.sahu.vo
    EmployeeVO.java
JRE System Library [JavaSE-17]
Server Runtime [Apache Tomcat v9.0]
Referenced Libraries
    ojdbc6.jar - D:\JAVA\Workspace\Jars
    lombok-1.18.24.jar - D:\JAVA\Workspace\Jars
build
src
    main
        java
        webapp
            META-INF
            WEB-INF
                lib
                web.xml
            employee_register.html
            error.jsp
            show_result.jsp
DBScript.sql
```

- Develop the above directory structure as Dynamic Web Project, create all the packages, classes, files as shown in above structure.
- Add the following jar in CLASS PATH, BUILD PATH and Deployment Assembly.
    - [Lombok API], lombok-1.18.24.jar
    - ojdbc6-11.2.0.4.jar
- Then use the following code with in their respective file.

jdbc.properties

```
#Jndi name for server managed JDBC connection pool
ds.jndi.name=java:/comp/env/DsJndi
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd" id="WebApp_ID"
version="4.0">
  <display-name>DPProj23-BusinessDelegateDP-
EmployeeRegistration</display-name>
  <welcome-file-list>
    <welcome-file>employee_register.html</welcome-file>
  </welcome-file-list>
</web-app>
```

Employee_register.html

```
<h1 style="color: red; text-align: center;">Employee Registration - Business
Delegate Design Pattern</h1>
<form action="controller" method="POST">
		<table border="1" bgcolor="cyan" align="center">
			<tr>
				<td>Employee Name: </td>
				<td><input type="text" name="ename"></td>
			</tr>
	<tr>
	   <td>Employee Address: </td>
	   <td><input type="text" name="eaddress"></td>
	</tr>
	<tr>
	   <td>Employee Desg: </td>
	   <td><input type="text" name="desg"></td>
	</tr>
	<tr>
	   <td>Employee Salary: </td>
	   <td><input type="text" name="salary"></td>
	</tr>
	<tr>
	   <td><input type="submit" value="Register Employee"></td>
	</tr>
		</table>
</form>
```

show_result.jsp

```
<%@ page isELIgnored="false" %>

<h1 style="color: green; text-align: center;">Result Page</h1>

<h3>Result is: ${resultMsg}</h3>
<br>
<a href="employee_register.html">Register another Employee</a>
```

error.jsp

```
<%@ page isELIgnored="false" %>

<h1 style="color: red; text-align: center;">Error Page</h1>

<h3>Internal Problem Try again: ${errorMsg}</h3>
<br>
<a href="employee_register.html">Register another Employee</a>
```

DBScript.sql

```
-- BUSINESS_DELEGATE_HR_EMP
CREATE TABLE "SYSTEM"."BUSINESS_DELEGATE_HR_EMP"
      ("ENO" NUMBER NOT NULL ENABLE,
      "ENAME" VARCHAR2(20 BYTE),
      "DESG" VARCHAR2(20 BYTE),
      "ADDRESS" VARCHAR2(20 BYTE),
       CONSTRAINT "BUSINESS_DELEGATE_HR_EMP_PK" PRIMARY KEY ("ENO"));

-- BUSINESS_DELEGATE_FINANCE_EMP
CREATE TABLE "SYSTEM"."BUSINESS_DELEGATE_FINANCE_EMP"
      ("ENO" NUMBER NOT NULL ENABLE,
      "ENAME" VARCHAR2(20 BYTE),
      "SALARY" NUMBER,
      "ADDRESS" VARCHAR2(20 BYTE),
       CONSTRAINT "BUSINESS_DELEGATE_FINANCE__PK" PRIMARY KEY ("ENO"));

-- BD_ENO_SEQ
CREATE SEQUENCE "SYSTEM"."BD_ENO_SEQ" MINVALUE 1 MAXVALUE 10000
INCREMENT BY 1 START WITH 1 CACHE 20 NOORDER NOCYCLE;
```

EmployeeVO.java

```
package com.sahu.vo;

import lombok.Data;
```

```
@Data
public class EmployeeVO {
	private String ename;
	private String address;
	private String desg;
	private String salary;
}
```

EmployeeBO.java

```
package com.sahu.bo;

import lombok.Data;

@Data
public abstract class EmployeeBO {
	public String ename;
	public String address;
}
```

HREmployeeBO.java

```
package com.sahu.bo;

import lombok.Data;

@Data
public class HREmployeeBO extends EmployeeBO {
	private String desg;
}
```

FinanceHREmployeeBO.java

```
package com.sahu.bo;

import lombok.Data;

@Data
public class FinanceEmployeeBO extends EmployeeBO {
	private Double salary;
}
```

ConnectionFactory.java

```
package com.sahu.factory;

import java.sql.Connection;
import java.sql.SQLException;
import java.util.ResourceBundle;

import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.DataSource;

public class ConnectionFactory {

        public static Connection getPooledConnection() throws
SQLException, NamingException {
                //create ResourceBundle class object
                ResourceBundle bundle =
ResourceBundle.getBundle("com/sahu/commons/jdbc");
                //get value from properties file by giving key
                String jndi = bundle.getString("ds.jndi.name");
                //create initialContext object
                InitialContext ic = new InitialContext();
                //get DataSource object from Jndi registry through jndi lookup
operation
                DataSource ds = (DataSource) ic.lookup(jndi);
                //get pooled JDBC connection using DataSource object
                Connection con = ds.getConnection();
                return con;
        }

        public static void closeConnection(Connection con) throws
SQLException {
                con.close();
        }

}
```

InternalProblemException.java

```
package com.sahu.exception;
```

```
public class InternalProblemException extends RuntimeException {

	private static final long serialVersionUID = 1L;

	public InternalProblemException(String msg) {
		super(msg);
	}

}
```

IEmployeeDAO.java

```
package com.sahu.dao;

import java.sql.Connection;
import java.sql.SQLException;

import com.sahu.bo.EmployeeBO;

public interface IEmployeeDAO {
	public int insert(EmployeeBO bo, Connection con) throws
SQLException;
}
```

HREmployeeDAOIml.java

```
package com.sahu.dao;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

import com.sahu.bo.EmployeeBO;
import com.sahu.bo.HREmployeeBO;

public class HREmployeeDAOIml implements IEmployeeDAO {

	private static final String INSERT_HR_EMP = "INSERT INTO
BUSINESS_DELEGATE_HR_EMP VALUES (BD_ENO_SEQ.CURRVAL,?,?,?)";

	@Override
	public int insert(EmployeeBO bo, Connection con) throws
SQLException {
		int result = 0;
		PreparedStatement ps  = null;
		try {
			//get PreparedStatement object having pre-compiled
SQL query
			ps = con.prepareStatement(INSERT_HR_EMP);
			//set values to query parameters
			ps.setString(1, bo.getEname());
			ps.setString(2, ((HREmployeeBO)bo).getDesg());
			ps.setString(3, bo.getAddress());
			//execute the query
			result = ps.executeUpdate();
		}
		catch (SQLException se) {
			result = 0;
			throw se;
		}
		catch (Exception e) {
			result = 0;
			throw e;
		}
		return result;
	}

}
```

FinanceEmployeeDAOIml.java

```
package com.sahu.dao;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

import com.sahu.bo.EmployeeBO;
import com.sahu.bo.FinanceEmployeeBO;

public class FinanceEmployeeDAOIml implements IEmployeeDAO {
	private static final String INSERT_FINANCE_EMP="INSERT INTO 
BUSINESS_DELEGATE_FINANCE_EMP 
VALUES(BD_ENO_SEQ.NEXTVAL,?,?,?)";

	@Override
	public int insert(EmployeeBO bo, Connection con) throws 
SQLException {
		int result=0;
		PreparedStatement ps=null;
		try {
			//get PreparedStatement object having pre-compiled 
SQL query
			ps=con.prepareStatement(INSERT_FINANCE_EMP);
			//set values to Query parameters
			ps.setString(1, bo.getEname());
			ps.setDouble(2,((FinanceEmployeeBO)bo).getSalary());
			ps.setString(3,bo.getAddress());
			// execute the Query
			result=ps.executeUpdate();
		}
		catch(SQLException se) {
			result=0;
			throw se;
		}
		catch(Exception e) {
			result=0;
			throw e;
		}
		return result;
	}

}
```

IEmployeeMgmtBusinessDelegate.java

```
package com.sahu.delegate;

import com.sahu.exception.InternalProblemException;
import com.sahu.vo.EmployeeVO;

public interface IEmployeeMgmtBusinessDelegate {
	public String registerEmployee(EmployeeVO vo) throws
InternalProblemException;
}
```

EmployeeMgmtBusinessDelegateImpl.java

```
package com.sahu.delegate;

import java.sql.Connection;
import java.sql.SQLException;

import javax.naming.NamingException;

import com.sahu.bo.FinanceEmployeeBO;
import com.sahu.bo.HREmployeeBO;
import com.sahu.dao.FinanceEmployeeDAOIml;
import com.sahu.dao.HREmployeeDAOIml;
import com.sahu.dao.IEmployeeDAO;
import com.sahu.exception.InternalProblemException;
import com.sahu.factory.ConnectionFactory;
import com.sahu.vo.EmployeeVO;

public class EmployeeMgmtBusinessDelegateImpl implements
IEmployeeMgmtBusinessDelegate {

	private IEmployeeDAO hrDAO, financeDAO;

	public EmployeeMgmtBusinessDelegateImpl() {
		hrDAO = new HREmployeeDAOIml();
		financeDAO = new FinanceEmployeeDAOIml();
	}

	@Override
	public String registerEmployee(EmployeeVO vo) throws
InternalProblemException {
		//convert EmployeeVO class object into HREmployeeBo,
FinanceEmployeeBO class object
		HREmployeeBO hrEmpBO = new HREmployeeBO();
		hrEmpBO.setEname(vo.getEname());
		hrEmpBO.setAddress(vo.getAddress());
```

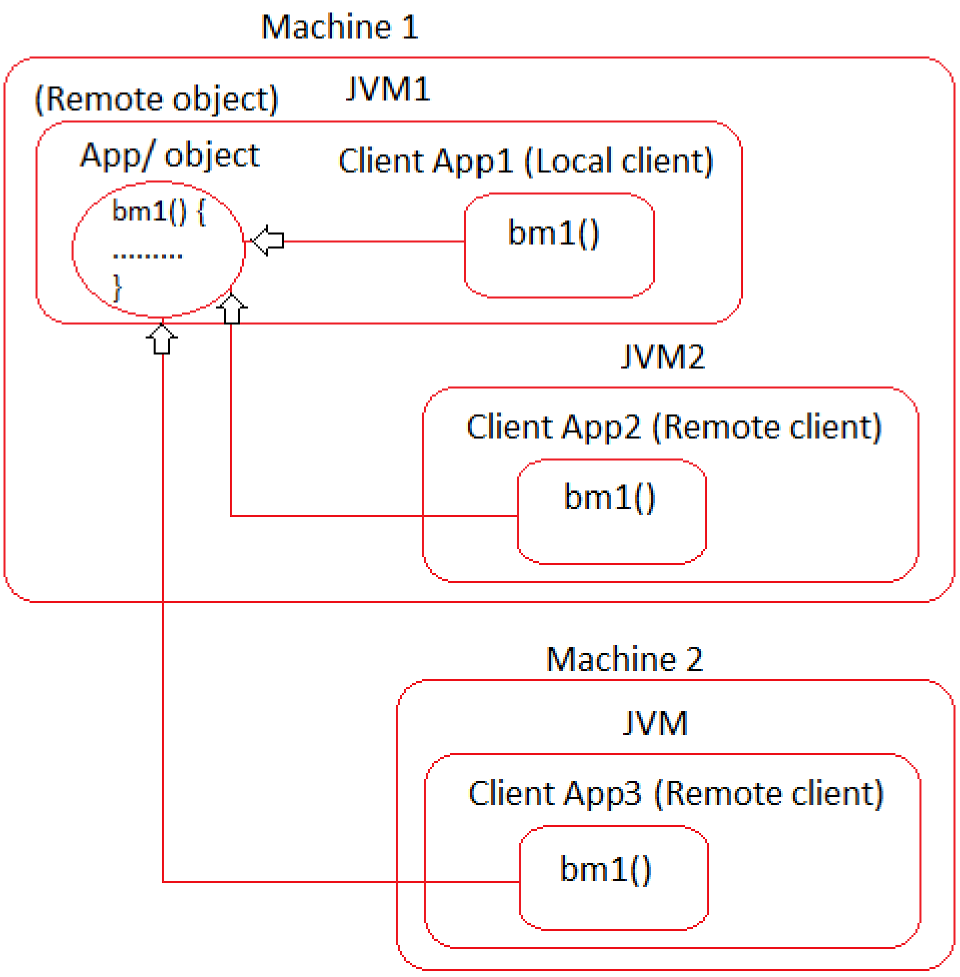
```
                    FinanceEmployeeBO financeEmpBO = new
FinanceEmployeeBO();
                    financeEmpBO.setEname(vo.getEname());
                    financeEmpBO.setAddress(vo.getAddress());
                    financeEmpBO.setSalary(Double.parseDouble(vo.getSalary()));

                    boolean flag = false;
                    Connection con = null;
                    String resultMsg = null;
                    try {
                              //get pooled JDBC connection object
                               con = ConnectionFactory.getPooledConnection();
                              //begin Tx Mgmt
                               con.setAutoCommit(false);
                              //invoke the methods both DAO class
                              int result1 = financeDAO.insert(financeEmpBO, con);
                              int result2 = hrDAO.insert(hrEmpBO, con);

                              if (result1==1 && result2==1)
                                        flag = true;
                              else
                                        flag = false;

                    }catch (SQLException se) {
                              flag = false;
                              if (se.getErrorCode()==1)
                                        throw new InternalProblemException("Duplicate
employee number is not allowed");
                              else if(se.getErrorCode()==1400)
                                        throw new InternalProblemException("Null
employee number is not allowed");
                              else if(se.getErrorCode()==12899)
                                        throw new InternalProblemException("Can't insert
more than colume size data");
                              else
                                        throw new InternalProblemException("Some DB
problem occured");
                    } catch (NamingException e) {
                              flag = false;
                              throw new InternalProblemException("Datasource
related Jndi problem");
		} catch (Exception e) {
			flag = false;
			throw new InternalProblemException("Unknown
problem has raised");
		}
		finally {
			//perform Transaction Management
			try {
				if (flag) {
					con.commit();
					resultMsg = "Employee Registration
successful";
				}
				else {
					con.rollback();
					resultMsg = "Employee Registration failed";
				}
			} catch (SQLException e) {
				e.printStackTrace();
			}

			//close connection (returning JDBC connection object
back to JDBC connection pool)
			try {
				ConnectionFactory.closeConnection(con);
			}catch (SQLException se) {
				se.printStackTrace();
			}
		}

		return resultMsg;
	}

}
```

MainController.java

```
package com.sahu.controller;

import java.io.IOException;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.sahu.delegate.EmployeeMgmtBusinessDelegateImpl;
import com.sahu.delegate.IEmployeeMgmtBusinessDelegate;
import com.sahu.exception.InternalProblemException;
import com.sahu.vo.EmployeeVO;


@WebServlet("/controller")
public class MainControllerServlet extends HttpServlet {

	private IEmployeeMgmtBusinessDelegate delegate;

	@Override
	public void init() throws ServletException {
		delegate = new EmployeeMgmtBusinessDelegateImpl();
	}

	@Override
	public void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
		//read form data and store into EmployeeVO class object
		EmployeeVO vo = new EmployeeVO();
		vo.setEname(request.getParameter("ename"));
		vo.setDesg(request.getParameter("desg"));
		vo.setAddress(request.getParameter("eaddress"));
		vo.setSalary(request.getParameter("salary"));

		//invoke business method of Business delegate
		try {
			String resultMsg = delegate.registerEmployee(vo);
			//keep result in request scope
			request.setAttribute("resultMsg", resultMsg);
			//forward request to show_result.jsp
			RequestDispatcher dispatcher =
request.getRequestDispatcher("/show_result.jsp");
			dispatcher.forward(request, response);
		}
		catch (InternalProblemException ipe) {
			request.setAttribute("errMsg", ipe.getMessage());
			//forward request to error.jsp
			RequestDispatcher dispatcher =
request.getRequestDispatcher("/error.jsp");
			dispatcher.forward(request, response);
		}
		catch (Exception e) {
			request.setAttribute("errMsg", e.getMessage());
			//forward request to error.jsp
			RequestDispatcher dispatcher =
request.getRequestDispatcher("/error.jsp");
			dispatcher.forward(request, response);
		}
	}

	@Override
	public void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
		doGet(request, response);
	}

	@Override
	public void destroy() {
		delegate = null;
	}

}
```

Service Locator Design Pattern

- The object whose methods can be called only from current JVM is called Local object.
- The object whose methods can be called from local JVM and also from remote JVMs is called Remote object.
- If app and its client app are residing in same JVM then that client app is called Local client application.



Note: Distributed application have different types of clients and it allowed both remote and local clients.

- If app and its client app are residing in two different JVMs of same machine or different machines then that client app is called Remote client application.
- These Remote objects/ Remote apps are called as Distributed application/ object and to develop such apps/ objects we need Distributed technologies or web services.
  - RMI, EJB - Java based Distributed technologies (outdated)
  - Webservices (SOAP based or Restful) - can be implemented and consumed different environment like JAVA, .NET, Python, PHP and etc.

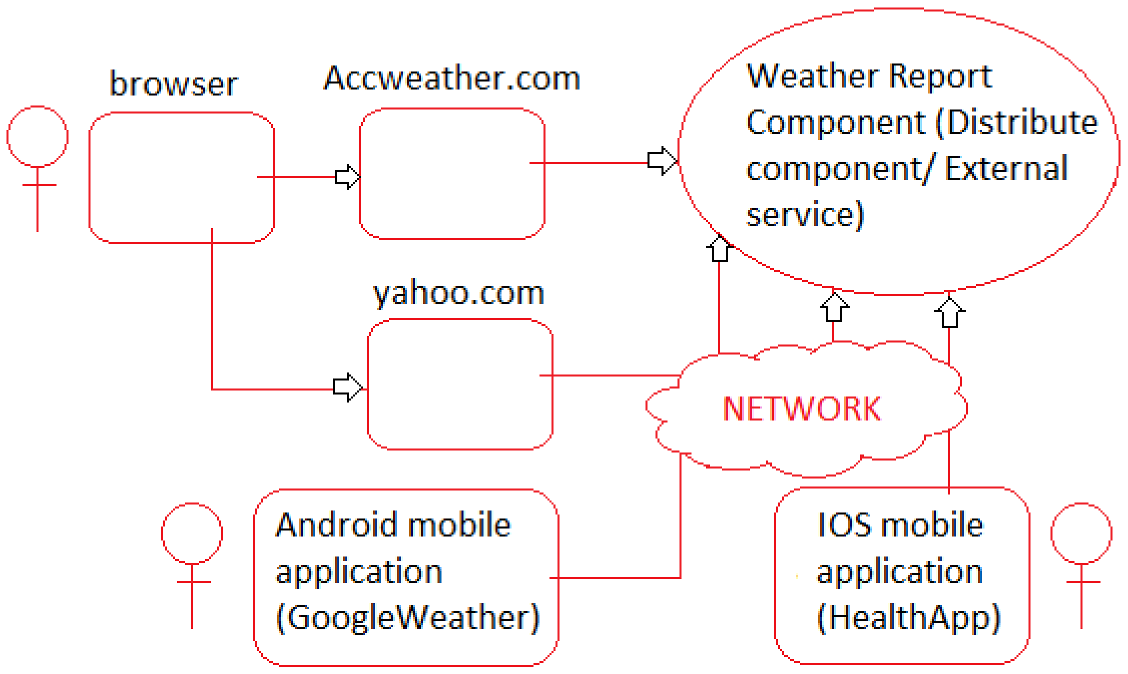
Examples for Distributed apps/ services/ objects:

- Google Pay/ PhonePe/ Amazon Pay and etc.
- PayPal, PayUMoney and etc.
- VISA/ Master/ Maestro and etc.

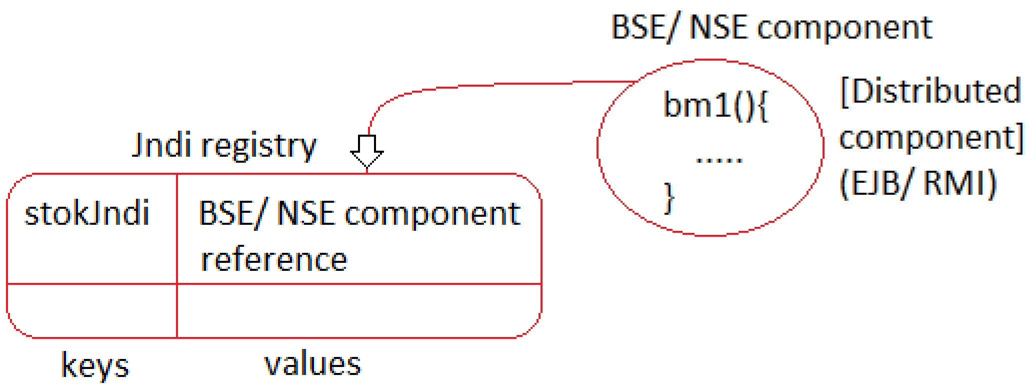
- BSE/ NSE and etc. components
- Weather report components
- ICC Cricket Score components
- COVID component

  and etc.

Note: These are also called as External services because they are outside of any project operated by end-user and can be used by multiple projects.


browser
Accweather.com
Weather Report Component (Distribute component/ External service)
yahoo.com
NETWORK
Android mobile application (GoogleWeather)
IOS mobile application (HealthApp)

- If Distributed component/ service is developed using RMI or EJB then it will be placed in JNDI registry for global visibility and accessibility and it will be identified with Jndi name

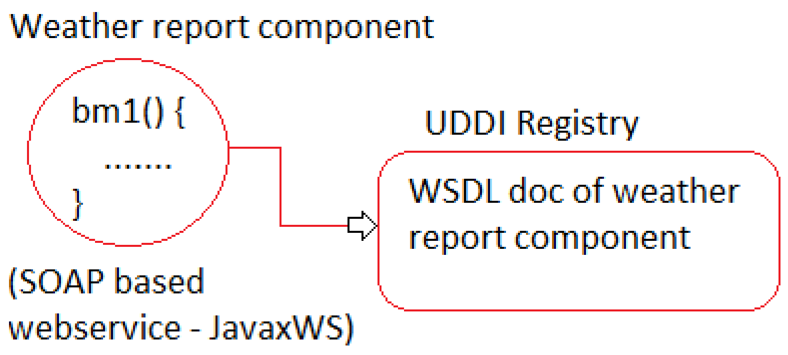

BSE/ NSE component
bm1(){
.....
}
[Distributed component] (EJB/ RMI)
Jndi registry
stokJndi
BSE/ NSE component reference
keys
values

- Here client apps get EJB/ RMI component reference from JNDI registry and invokes business methods on it.
- Local objects are specific to java local method.

- Instance objects are specific to current invoking java object.
- static objects are specific to current invoking java class.
- To provide more global visibility and accessibility to java object go for Jndi registry.

  E.g., RMI registry, COS Registry, Tomcat Jndi registry, WebLogic Jndi registry and etc.

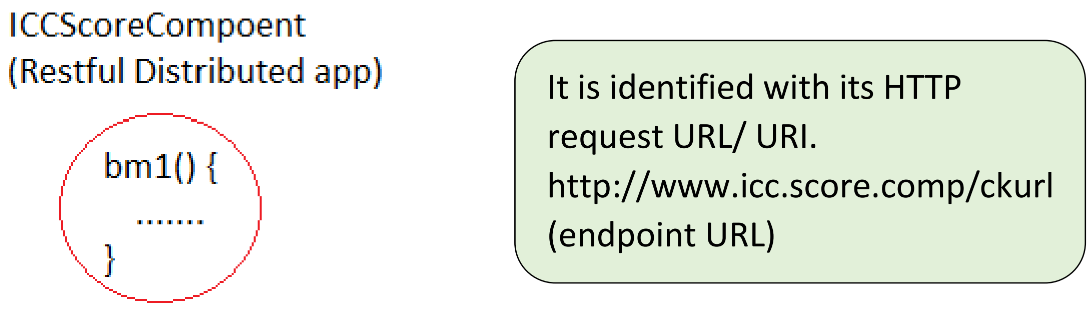
- SOAP Based web service components will be placed in UDDI Registry having WSDL Documents.
  - UDDI: Universal Description, Discovery, and Integration (registry)
  - WSDL Docs: Webservice Description Language Documents.



Weather report component
bm1() {
....
}
(SOAP based webservice - JavaxWS)
UDDI Registry
WSDL doc of weather report component

- Here client app gathers WSDL documents from UDDI registry and writes code invoke the business methods of SOAP based webservice component.

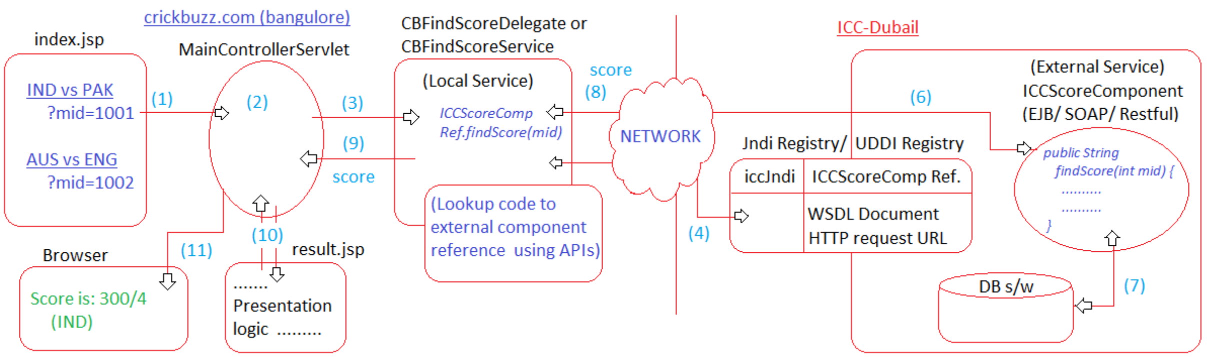
- Restful webservice components are HTTP based web component so they are identified with HTTP URLs/ URIs i.e., no need of any separate registries.



ICCScoreCompoent
(Restful Distributed app)
bm1() {
....
}
It is identified with its HTTP request URL/ URI.
http://www.icc.score.comp/ckurl
(endpoint URL)

- Here client apps use HTTP request URI/ URL of Restful webservice comp to consume webservice by invoking business methods.

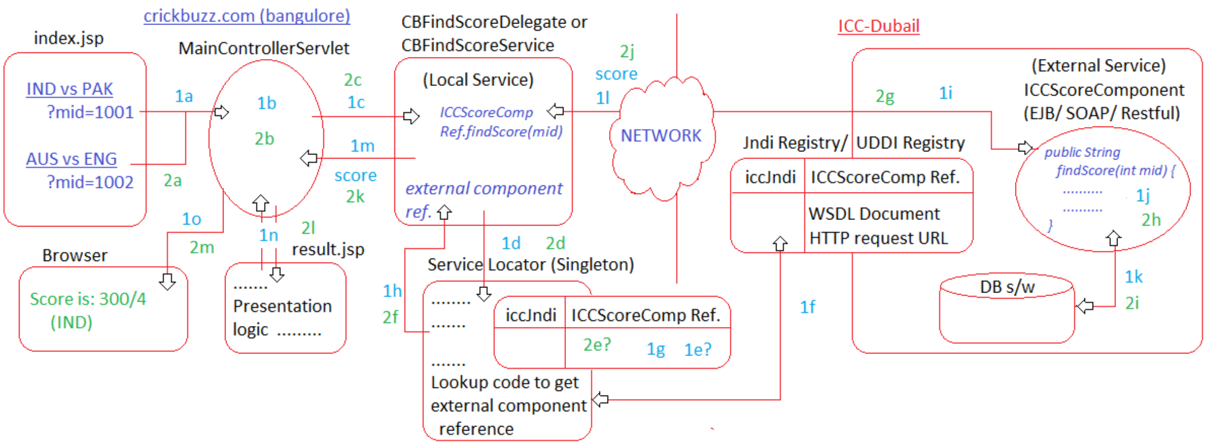
Problem:


crickbuzz.com (bangulore)
index.jsp
MainControllerServlet
IND vs PAK ?mid=1001
AUS vs ENG ?mid=1002
(1)
(2)
(3)
(9)
score
(10)
(11)
Browser
Score is: 300/4 (IND)
result.jsp
.... Presentation logic ....
CBFindScoreDelegate or CBFindScoreService
(Local Service)
ICCScoreComp Ref.findScore(mid)
(Lookup code to external component reference using APIs)
score
(8)
NETWORK
(4)
ICC-Dubail
(6)
(External Service) ICCScoreComponent (EJB/ SOAP/ Restful)
public String findScore(int mid) { .... }
Jndi Registry/ UDDI Registry
iccJndi
ICCScoreComp Ref.
WSDL Document
HTTP request URL
DB s/w
(7)

Placing lookup code to get external/ distributed component reference from different registries using our choice API (like JNDI) in Local service/ Local business delegate class is having the following limitations:

1. Lookup code to get external component reference is not reusable across the multiple Local service or business delegate classes.
2. If external component technology/ location is changed we need to modify the lookup code in Local service/ business delegate class (tight coupling).
3. If registry environment (location/ technology) is changed we need to modify the lookup code in Local service class (tight coupling).
4. Improves network rounds trips between Local service/ business delegate class and Registry service where the external component reference is registered because there is no cache maintaining external component reference across the multiple same/ similar requests.

Solution:


crickbuzz.com (bangulore)
index.jsp
MainControllerServlet
IND vs PAK ?mid=1001
AUS vs ENG ?mid=1002
1a
2a
1b
2b
2c
1c
1m
score
2k
1o
2m
1n
2l
Browser
Score is: 300/4 (IND)
result.jsp
.... Presentation logic ....
CBFindScoreDelegate or CBFindScoreService
(Local Service)
ICCScoreComp Ref.findScore(mid)
external component ref.
2j
score
1l
NETWORK
1d
2d
Service Locator (Singleton)
1h
2f
....
Lookup code to get external component reference
iccJndi
ICCScoreComp Ref.
2e?
1g
1e?
1f
ICC-Dubail
2g
1i
(External Service) ICCScoreComponent (EJB/ SOAP/ Restful)
public String findScore(int mid) { .... }
1j
2h
Jndi Registry/ UDDI Registry
iccJndi
ICCScoreComp Ref.
WSDL Document
HTTP request URL
DB s/w
1k
2i

1a to 1o – 1st request
2a to 2m – 2nd request [Both are same request]

To overcome those problems, keep lookup code to get external component reference from registries in a separate Java class having caching support, that class is called Service Locator because it contains lookup code to locate external service/ distribute component.

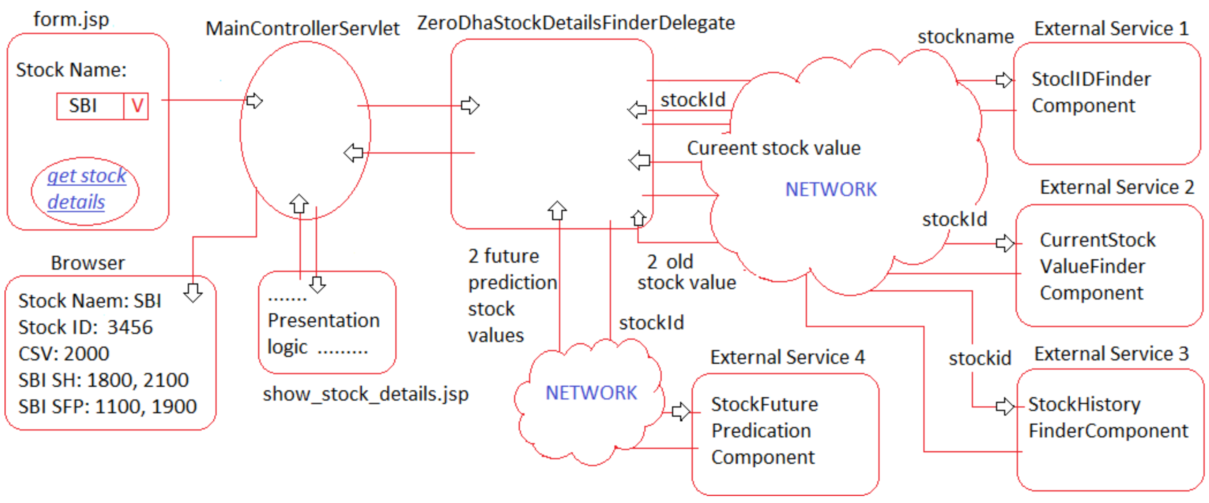
Advantages of Service Locator:

a. The lookup code placed in Service Locator becomes reusable across the multiple Local service/ business delegate classes.
b. If any change is there in external component location or technology, we just need to modify service locator rather modifying local service business delegate classes (Loose coupling).
c. If any change is there in registry environment (location or technology) we just need to modify service locator rather modifying local service business delegate classes (Loose coupling).
d. Since Service Locator is maintaining cache having external component reference the network round trips between Java app and DB s/w will be reduced drastically.

Session Facade Design Pattern

- Sometimes the local service class or business delegate class has to invoke the multiple business methods of multiple external components in order to complete the task as show below.

Problem:



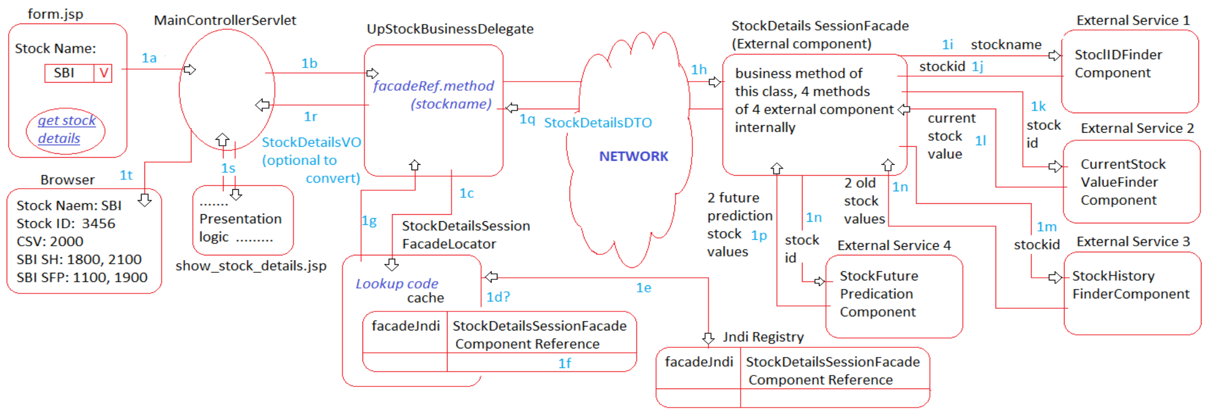
Limitations:

a. Business delegate should use multiple network round trips in order to interact with multiple external service components.

b. Business delegate should know order of invoking multiple external components.
c. The lookup codes written in Business delegate is not reusable in multiple other business delegate classes.
d. The lookup code of Business delegate is tightly coupled with external components technology and location and also tightly coupled with registry technology and location.

Solution:

- Session Facade design pattern (It is recommended to use Session Facade with Service Locator)
- Take one dummy external component where other multiple external components are available and make business delegate interacting with that dummy external component which is locally interacting with multiple external components in an order. This dummy external component is technically called as Session Facade component.

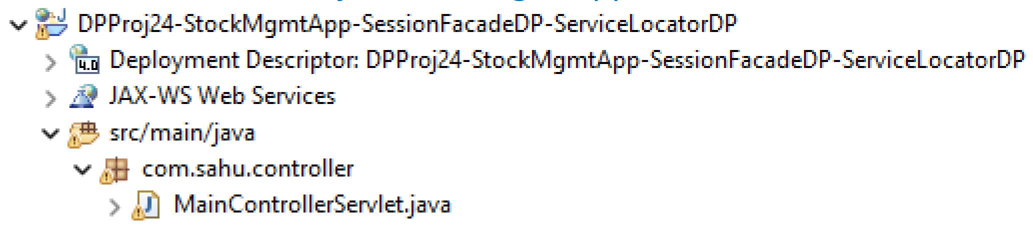


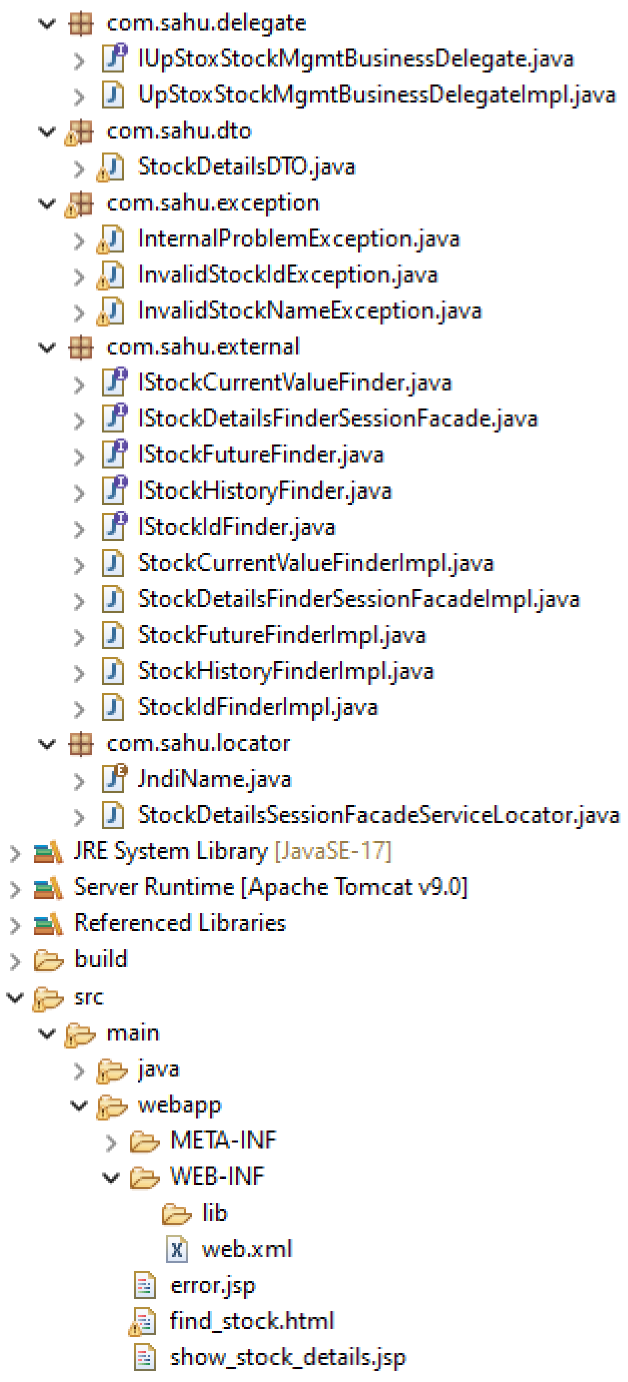
form.jsp
Stock Name:
SBI
V
get stock details
1a
MainControllerServlet
1b
1r
UpStockBusinessDelegate
facadeRef.method (stockname)
1q
StockDetailsDTO
NETWORK
1h
StockDetails SessionFacade (External component)
business method of this class, 4 methods of 4 external component internally
1i
stockname
stockid
1j
External Service 1
StocIIDFinder Component
1k
stock id
current stock value
1l
External Service 2
CurrentStock ValueFinder Component
Browser
1t
Stock Naem: SBI
Stock ID: 3456
CSV: 2000
SBI SH: 1800, 2100
SBI SFP: 1100, 1900
1s
StockDetailsVO (optional to convert)
.... Presentation logic ....
show_stock_details.jsp
1c
1g
StockDetailsSession FacadeLocator
Lookup code cache
1d?
1e
facadeJndi
StockDetailsSessionFacade Component Reference
1f
2 future prediction stock values
1p
1n
stock id
2 old stock values
1n
External Service 4
StockFuture Predication Component
1m
stockid
External Service 3
StockHistory FinderComponent
Jndi Registry
facadeJndi
StockDetailsSessionFacade Component Reference

Advantages:

a. Reduces network trips between Business delegate and external components.
b. Business delegate need not worry about order of invoking multiple external components because the Session Facade itself takes care of it.
c. All Service locator advantages are can be taken by Business delegate.

Directory Structure of DPProj24-StockMgmtApp-SessionFacadeDP-ServiceLocatorDP:


DPProj24-StockMgmtApp-SessionFacadeDP-ServiceLocatorDP
Deployment Descriptor: DPProj24-StockMgmtApp-SessionFacadeDP-ServiceLocatorDP
JAX-WS Web Services
src/main/java
com.sahu.controller
MainControllerServlet.java



```
com.sahu.delegate
    IUpStoxStockMgmtBusinessDelegate.java
    UpStoxStockMgmtBusinessDelegateImpl.java
com.sahu.dto
    StockDetailsDTO.java
com.sahu.exception
    InternalProblemException.java
    InvalidStockIdException.java
    InvalidStockNameException.java
com.sahu.external
    IStockCurrentValueFinder.java
    IStockDetailsFinderSessionFacade.java
    IStockFutureFinder.java
    IStockHistoryFinder.java
    IStockIdFinder.java
    StockCurrentValueFinderImpl.java
    StockDetailsFinderSessionFacadeImpl.java
    StockFutureFinderImpl.java
    StockHistoryFinderImpl.java
    StockIdFinderImpl.java
com.sahu.locator
    JndiName.java
    StockDetailsSessionFacadeServiceLocator.java
JRE System Library [JavaSE-17]
Server Runtime [Apache Tomcat v9.0]
Referenced Libraries
build
src
  main
    java
    webapp
      META-INF
      WEB-INF
        lib
        web.xml
      error.jsp
      find_stock.html
      show_stock_details.jsp
```

- Develop the above directory structure as Dynamic Web Project, create all the packages, classes and files as shown in above structure.
- Add the following jar in CLASS PATH, BUILD PATH and Deployment Assembly.
    - [Lombok API], lombok-1.18.24.jar
- Then use the following code with in their respective file.

Note: You can implement the external components as separate project using webservices.

IStockIdFinder.java

```java
package com.sahu.external;

import com.sahu.exception.InvalidStockNameException;

public interface IStockIdFinder {
	public long findStockIdByStockName(String stockName) throws
InvalidStockNameException;
}
```

StockIdFinderImpl.java

```java
package com.sahu.external;

import com.sahu.exception.InvalidStockNameException;

public class StockIdFinderImpl implements IStockIdFinder {

	@Override
	public long findStockIdByStockName(String stockName) throws
InvalidStockNameException {
		if (stockName.equalsIgnoreCase("SBI"))
			return 1001;
		else if (stockName.equalsIgnoreCase("ICICI"))
			return 1002;
		else if (stockName.equalsIgnoreCase("Reliance"))
			return 1003;
		else
			throw new InvalidStockNameException("Wrong stock
name - "+stockName);
	}

}
```

IStockCurrentValueFinder.java

```java
package com.sahu.external;

import com.sahu.exception.InvalidStockIdException;

public interface IStockCurrentValueFinder {
	public long findStockCurrentValue(long stockId) throws
InvalidStockIdException;
}
```

StockCurrentValueFinderImpl.java

```
package com.sahu.external;

import java.util.Random;

import com.sahu.exception.InvalidStockIdException;

public class StockCurrentValueFinderImpl implements
IStockCurrentValueFinder {

	@Override
	public long findStockCurrentValue(long stockId) throws
InvalidStockIdException {
		long stockValue = new Random(1000L).nextLong();
		if (stockId==1001)
			return stockValue+100;
		else if (stockId==1002)
			return stockValue+125;
		else if (stockId==1003)
			return stockValue+75;
		else
			throw new InvalidStockIdException("Invalid stock id -
"+stockId);
	}

}
```

InvalidStockIdException.java

```
package com.sahu.exception;

public class InvalidStockIdException extends Exception {
	public InvalidStockIdException(String msg) {
		super(msg);
	}
}
```

InvalidStockNameException.java

```
package com.sahu.exception;

public class InvalidStockNameException extends RuntimeException {
	public InvalidStockNameException(String msg) {
		super(msg);
	}
}
```

InternalProblemException.java

```
package com.sahu.exception;

public class InternalProblemException extends Exception {
	public InternalProblemException(String msg) {
		super(msg);
	}
}
```

IStockHistoryFinder.java

```
package com.sahu.external;

import java.util.List;

import com.nt.exception.InvalidStockIdException;

public interface IStockHistoryFinder {
	public List<Long> fetchStockHistory(long stockId) throws
InvalidStockIdException;
}
```

StockHistoryFinderImpl.java

```
package com.sahu.external;

import java.util.List;
import java.util.Random;

import com.nt.exception.InvalidStockIdException;

public class StockHistoryFinderImpl implements IStockHistoryFinder {
	@Override
	public List<Long> fetchStockHistory(long stockId) throws
InvalidStockIdException {
		long value1 = new Random().nextInt(1000);
		long value2 = Math.round(value1+(value1*0.1));
		long value3 = Math.round(value1-(value1*0.1));
		
		if (stockId==1001) {
			value3 = value3-10;
		}
		else if (stockId==1002) {
			value1 = value1+100;
			value2 = value1-100;
			value3 = value1-50;
		}
		else if (stockId==1003) {
			value1 = value1+100;
			value2 = value1+100;
			value3 = value1-60;
		}
		else
			throw new InvalidStockIdException("Invalid stock id -
"+stockId);
		
		return List.of(value1, value2, value3);
	}

}
```

IStockFutureFinder.java

```
package com.sahu.external;

import java.util.List;

import com.sahu.exception.InvalidStockIdException;

public interface IStockFutureFinder {
	public List<Long> fetchStockFuture(long stockId) throws
InvalidStockIdException;
}
```

StockFutureFinderImpl.java

```
package com.sahu.external;

import java.util.List;
import java.util.Random;

import com.sahu.exception.InvalidStockIdException;

public class StockFutureFinderImpl implements IStockFutureFinder {

	@Override
	public List<Long> fetchStockFuture(long stockId) throws
InvalidStockIdException {
		long value1 = new Random().nextInt(10000);
		long value2 = Math.round(value1+(value1*0.1));

		if (stockId==1001) {
			value1 = value1-10;
		}
		else if (stockId==1002) {
			value1 = value1-100;
			value2 = value1+100;
		}
		else if (stockId==1003) {
			value1 = value1+100;
			value2 = value1-200;
		}
		else
			throw new InvalidStockIdException("Invalid stock id -
"+stockId);

		return List.of(value1, value2);
	}

}
```

IStockDetailsFinderSessionFacade.java

```
package com.sahu.external;

import com.nt.dto.StockDetailsDTO;
import com.sahu.exception.InvalidStockIdException;
import com.sahu.exception.InvalidStockNameException;

public interface IStockDetailsFinderSessionFacade {
	public StockDetailsDTO findCompleteStockDetailsByName(String
stockName) throws InvalidStockNameException, InvalidStockIdException;
}
```

StockDetailsDTO.java

```
package com.sahu.dto;

import java.io.Serializable;
import java.util.List;

import lombok.Data;

@Data
public class StockDetailsDTO implements Serializable {
	private Long stockId;
	private String stockName;
	private Long currentValue;
	private List<Long> historyValue;
	private List<Long> futureValue;
}
```

StockDetailsFinderSessionFacadeImpl.java

```
package com.sahu.external;

import java.util.List;

import com.sahu.dto.StockDetailsDTO;
import com.sahu.exception.InvalidStockIdException;
import com.sahu.exception.InvalidStockNameException;

public class StockDetailsFinderSessionFacadeImpl implements
IStockDetailsFinderSessionFacade {

	@Override
	public StockDetailsDTO findCompleteStockDetailsByName(String
stockName)
                        throws InvalidStockNameException,
InvalidStockIdException {
                //Invoke other Components (here multiple Jndi lookup should
happen, but we are creating objects directly
                //because we have external components as the ordinary java
classes).
                IStockIdFinder comp1 = new StockIdFinderImpl();
                long stockId = comp1.findStockIdByStockName(stockName);

                IStockCurrentValueFinder comp2 = new
StockCurrentValueFinderImpl();
                long currentValue = comp2.findStockCurrentValue(stockId);

                IStockHistoryFinder comp3 = new StockHistoryFinderImpl();
                List<Long> historyValueList =
comp3.fetchStockHistory(stockId);

                IStockFutureFinder comp4 = new StockFutureFinderImpl();
                List<Long> futureValueList = comp4.fetchStockFuture(stockId);

                //Prepare DTO class having multiple details
                StockDetailsDTO dto = new StockDetailsDTO();
                dto.setStockId(stockId);
                dto.setStockName(stockName);
                dto.setCurrentValue(currentValue);
                dto.setHistoryValue(historyValueList);
                dto.setFutureValue(futureValueList);

                return dto;
        }

}
```

JndiName.java

```
package com.sahu.locator;

public enum JndiName {
        STOCK_JNDI;
}
```

StockDetailsSessionFacadeServiceLocator.java

```
package com.sahu.locator;

import java.util.HashMap;
import java.util.Map;

import com.sahu.external.IStockDetailsFinderSessionFacade;
import com.sahu.external.StockDetailsFinderSessionFacadeImpl;

public class StockDetailsSessionFacadeServiceLocator {

	private static StockDetailsSessionFacadeServiceLocator INSTANCE;
	private Map<JndiName, IStockDetailsFinderSessionFacade> 
cacheMap;

	private StockDetailsSessionFacadeServiceLocator() {
		cacheMap = new HashMap<>();
	}

	public static StockDetailsSessionFacadeServiceLocator getInstance() {
		if (INSTANCE==null)
			INSTANCE = new 
StockDetailsSessionFacadeServiceLocator();

		return INSTANCE;
	}

	public IStockDetailsFinderSessionFacade getFacade(JndiName 
jndiName) {
		if (!cacheMap.containsKey(jndiName)) {
			//here also we should write lookup code, since we have 
taken SessionFacade
			//as ordinary class we are creating object directly
			IStockDetailsFinderSessionFacade facade = new 
StockDetailsFinderSessionFacadeImpl();
			cacheMap.put(jndiName, facade);
		}
		return cacheMap.get(jndiName);
	}

}
```

IUpStoxStockMgmtBusinessDelegate.java

```java
package com.sahu.delegate;

import com.sahu.dto.StockDetailsDTO;
import com.sahu.exception.InternalProblemException;

public interface IUpStoxStockMgmtBusinessDelegate {
	public StockDetailsDTO fetchStockDetailsByName(String stockName)
throws InternalProblemException;
}
```

UpStoxStockMgmtBusinessDelegateImpl.java

```java
package com.sahu.delegate;

import com.sahu.dto.StockDetailsDTO;
import com.sahu.exception.InternalProblemException;
import com.sahu.exception.InvalidStockIdException;
import com.sahu.exception.InvalidStockNameException;
import com.sahu.external.IStockDetailsFinderSessionFacade;
import com.sahu.locator.JndiName;
import com.sahu.locator.StockDetailsSessionFacadeServiceLocator;

public class UpStoxStockMgmtBusinessDelegateImpl implements
IUpStoxStockMgmtBusinessDelegate {

	@Override
	public StockDetailsDTO fetchStockDetailsByName(String stockName)
throws InternalProblemException {
		//get SessionFacade component reference using ServicLocator
		try {
			StockDetailsSessionFacadeServiceLocator locator =
StockDetailsSessionFacadeServiceLocator.getInstance();
			IStockDetailsFinderSessionFacade facade =
locator.getFacade(JndiName.STOCK_JNDI);
			//Invoke business method
			StockDetailsDTO dto =
facade.findCompleteStockDetailsByName(stockName);
			return dto;
		}
		catch (InvalidStockNameException isne) {
```

```
                              throw new
InternalProblemException(isne.getMessage());
                    } catch (InvalidStockIdException isie) {
                              throw new InternalProblemException(isie.getMessage());
                    }
          }

}
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd" id="WebApp_ID"
version="4.0">
  <display-name>DPProj24-StockMgmtApp-SessionFacadeDP-
ServiceLocatorDP</display-name>
  <welcome-file-list>
    <welcome-file>find_stock.html</welcome-file>
  </welcome-file-list>
</web-app>
```

find_stock.html

```
<h1 style="color: red; text-align: center;">Session Facade - Service Locator
DP</h1>
<form action="controller" method="POST">
          <table align="center" border="1" bgcolor="cyan">
                    <tr>
                              <td>Stock Name: </td>
                              <td><input type="text" name="stockName"></td>
                    </tr>
                    <tr>
                              <td colspan="2"><input type="submit" value="Get Stock
Details"></td>
                    </tr>
          </table>
</form>
```

show_stock_details.jsp

```
<%@ page isELIgnored="false"%>

<h1 style="color: blue">StockDetils are : ${result}</h1>
<br>
<a href="find_stock.hatml">Home</a>
```

error.jsp

```
<%@ page isELIgnored="false"%>

<h1 style="color: red">Problem: ${errorMsg}</h1>
<br>
<a href="find_stock.html">Home</a>
```

MainControllerServlet.java

```
package com.sahu.controller;

import java.io.IOException;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.sahu.delegate.IUpStoxStockMgmtBusinessDelegate;
import com.sahu.delegate.UpStoxStockMgmtBusinessDelegateImpl;
import com.sahu.dto.StockDetailsDTO;
import com.sahu.exception.InternalProblemException;

@WebServlet("/controller")
public class MainControllerServlet extends HttpServlet {

	private IUpStoxStockMgmtBusinessDelegate delegate;

	@Override
	public void init() throws ServletException {
		delegate = new UpStoxStockMgmtBusinessDelegateImpl();
	}
	public void doGet(HttpServletRequest request, HttpServletResponse 
response) throws ServletException, IOException {
		//read form data
		String stockName = request.getParameter("stockName");
		//use Business delegate
		try{
			StockDetailsDTO dto = 
delegate.fetchStockDetailsByName(stockName);
			//keep result in request scope
			request.setAttribute("result", dto);
			//forward to show_stock_details.jsp
			RequestDispatcher rd = 
request.getRequestDispatcher("/show_stock_details.jsp");
			rd.forward(request, response);
		} catch (InternalProblemException ipe) {
			ipe.printStackTrace();
			request.setAttribute("errMsg", ipe.getMessage());
			RequestDispatcher rd = 
request.getRequestDispatcher("/error.jsp");
			rd.forward(request, response);
		} catch (Exception e) {
			e.printStackTrace();
			request.setAttribute("errMsg", e.getMessage());
			RequestDispatcher rd = 
request.getRequestDispatcher("/error.jsp");
			rd.forward(request, response);
		}
	}

	public void doPost(HttpServletRequest request, HttpServletResponse 
response) throws ServletException, IOException {
		doGet(request, response);
	}

	@Override
	public void destroy() {
		delegate = null;
	}

}
```

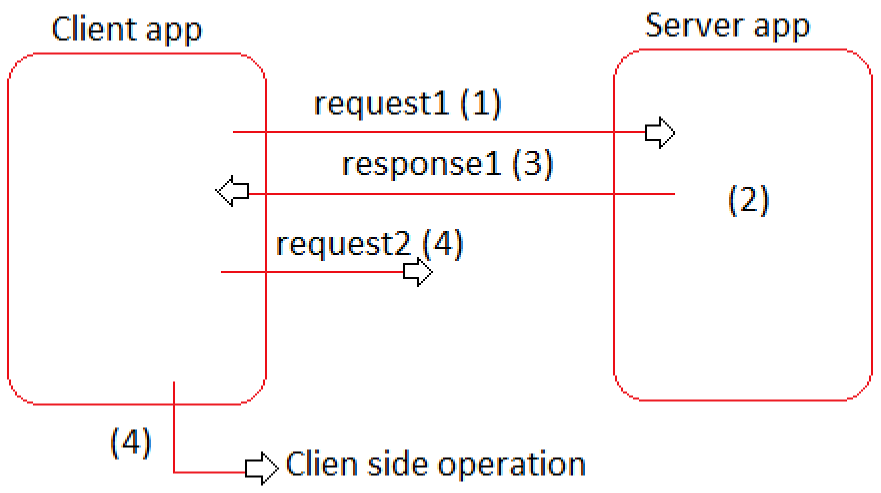
Message Facade Design Pattern

Q. What is difference between synchronous communication and Asynchronous communication?

Ans.

Synchronous communication:

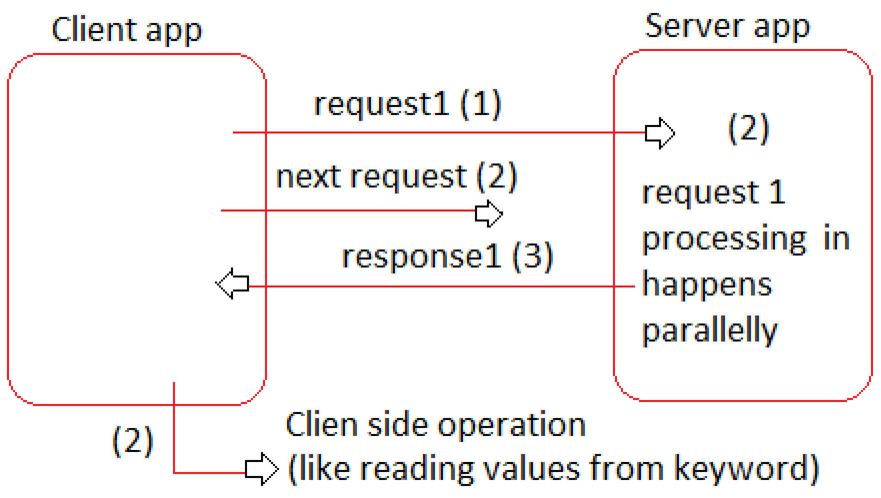
- In client-server application, if client app is waiting for given request related response to generate next request or to perform client-side operations then it is called synchronous communication.



Note: Here client app blocked (i.e., sitting idle) until given request related response comes back to client app from server app.

Asynchronous communication:

- In client-server application, if the client app is free to generate the next request or to perform client-side operations without blocking itself until given request related response comes back to client app from server app then it is called asynchronous communication.

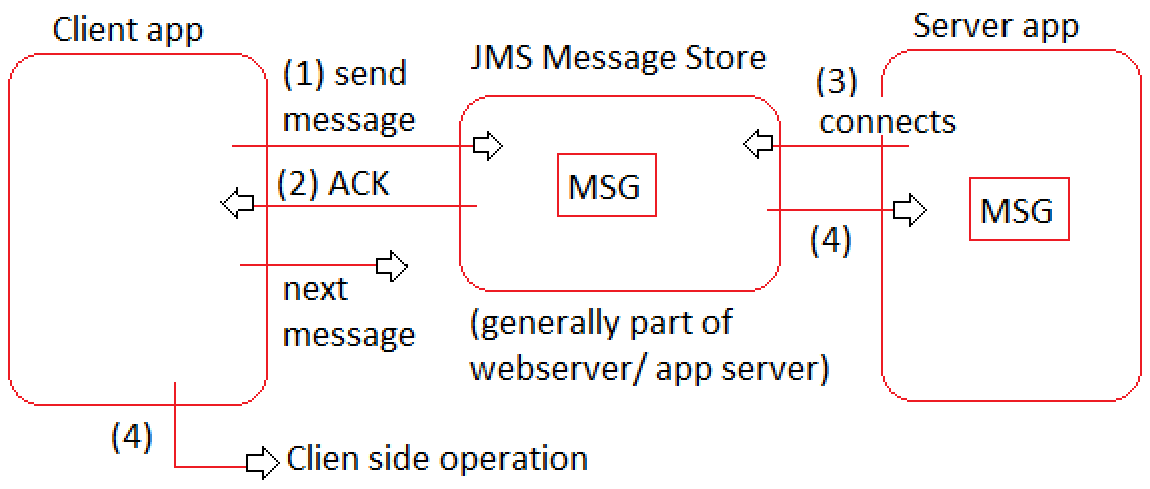


Client app
Server app
request1 (1)
(2)
next request (2)
request 1 processing in happens parallelly
response1 (3)
(2)
Clien side operation (like reading values from keyword)

Note:

- ✓ By default, java networking apps are synchronous apps.
- ✓ All web applications give synchronous communication between browser and website by default.
- ✓ All distributed applications give synchronous communication between client project (Business delegate) and distributed component (external service).

- Java web application using AJAX at client side or Servlet 4.x or Portlets at server side enables the Asynchronous commination between browser and web application (website).
- E.g.,
  - Gmail inbox page supports asynchronous communication, we can click hyperlink to open certain service though it is taking time we can continue the chatting.
  - In registration forms we check email id availability by clicking hyperlink or button though it is taking time we can fill up the remaining textboxes of the form page without getting blocked.
- To get asynchronous communication between Java apps and Java app and EJB component we can use JMS (Java Messaging Service).

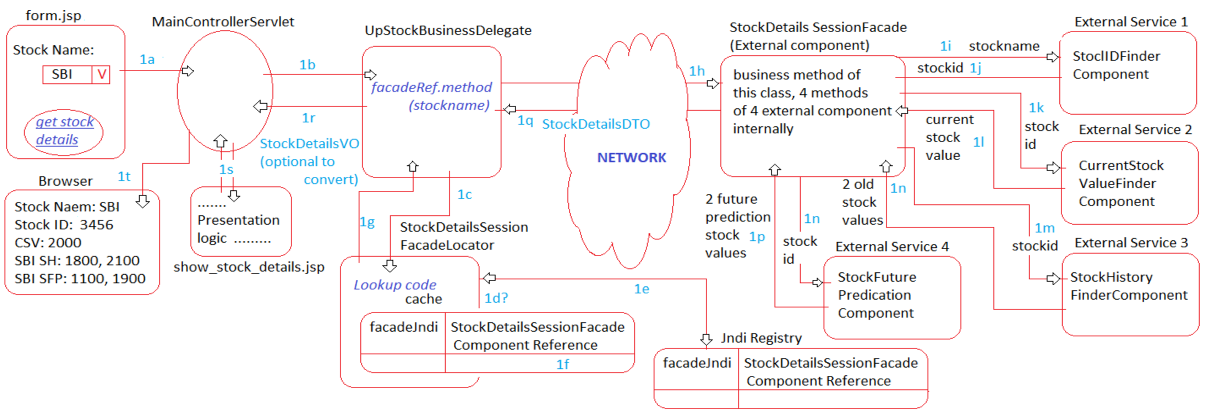


- Here both client and server need not be in active mode at a time. i.e., here app1 and app2 are using JMS support to have message-based interaction in asynchronous manner through message store.
- JMS is for interaction between two apps or components. It not for SMS sending and not person to person interaction and not for person to s/w app interaction not for WhatsApp or Hike kind of interactions. For these we can Java Mail API or Android or IOS APIs.
- MDB (Message Driven Bean) is special EJB component which can read/

keep message in JMS store as a distributed component.

- In webservice we can use polling concept to get asynchronous communication between client app and its webservice components.

Problem:

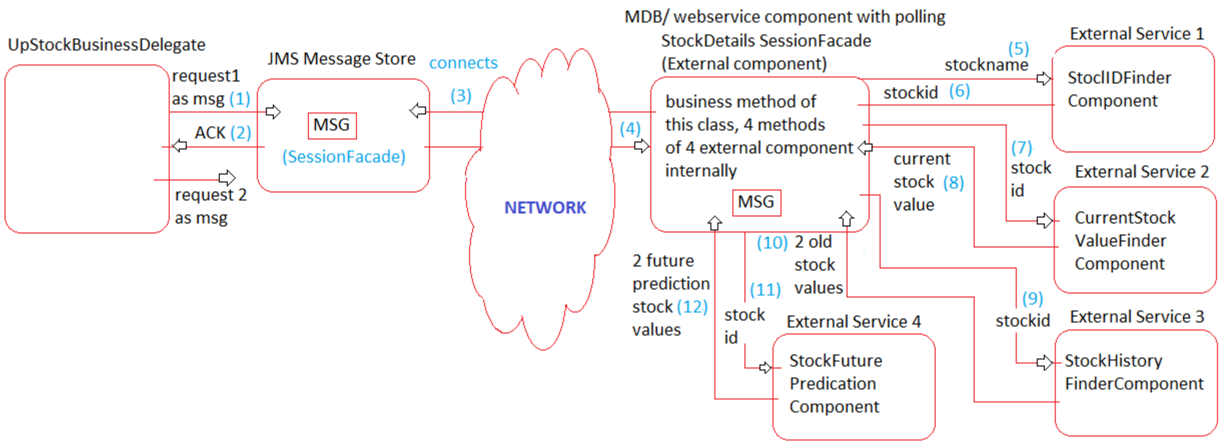


form.jsp
Stock Name:
SBI
get stock details
MainControllerServlet
UpStockBusinessDelegate
facadeRef.method (stockname)
StockDetailsVO (optional to convert)
StockDetailsDTO
NETWORK
StockDetails SessionFacade (External component)
business method of this class, 4 methods of 4 external component internally
External Service 1
StoclIDFinder Component
External Service 2
CurrentStock ValueFinder Component
External Service 3
StockHistory FinderComponent
External Service 4
StockFuture Predication Component
stockname
stockid
current stock value
stock id
2 old stock values
2 future prediction stock values
Browser
Stock Naem: SBI
Stock ID: 3456
CSV: 2000
SBI SH: 1800, 2100
SBI SFP: 1100, 1900
.... Presentation logic ....
show_stock_details.jsp
StockDetailsSession FacadeLocator
Lookup code cache
facadeJndi
StockDetailsSessionFacade Component Reference
Jndi Registry
1a 1b 1r 1s 1t 1c 1g 1d? 1e 1f 1q 1h 1i 1j 1k 1l 1m 1n 1p

- Here, Business delegate is interacting with Session Facade and its internal and external components in synchronous mode i.e., Business delegate has to wait to generate next request until given request related response comes back to Business delegate from Session Facade. This degrades the performance of client app/ project like upstox.com.

Solution:

- Use Message Facade with the support JMS and MDB/ Polling based



- Here the communication between Business delegate and Session Facade is messages based asynchronous communication. So, after putting request as message in JMS store the Business delegate is free to generate the next request as message without waiting for given request message related response.

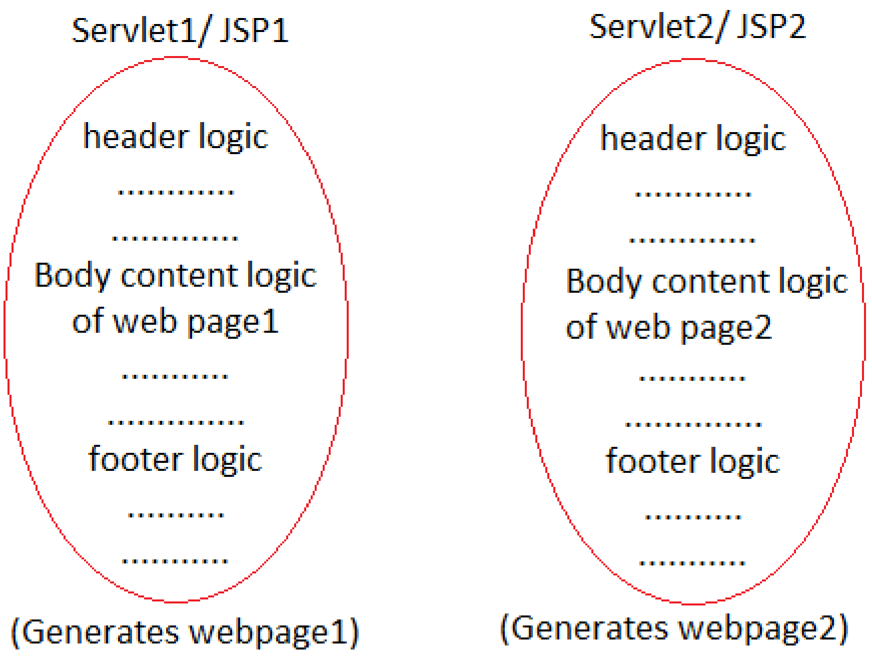
- IVR system interaction-based communication is messages based asynchronous communication.
  E.g., Call center calls, Gas Booking and etc.
- In a manufacturing company multiple departments whose working timings are different will interact with each other through messages based asynchronous communication.

Important Presentation tier Design Patterns:

a. Composite View
b. View Helper
c. Interception Filter
d. FrontController

Composite View Design Pattern

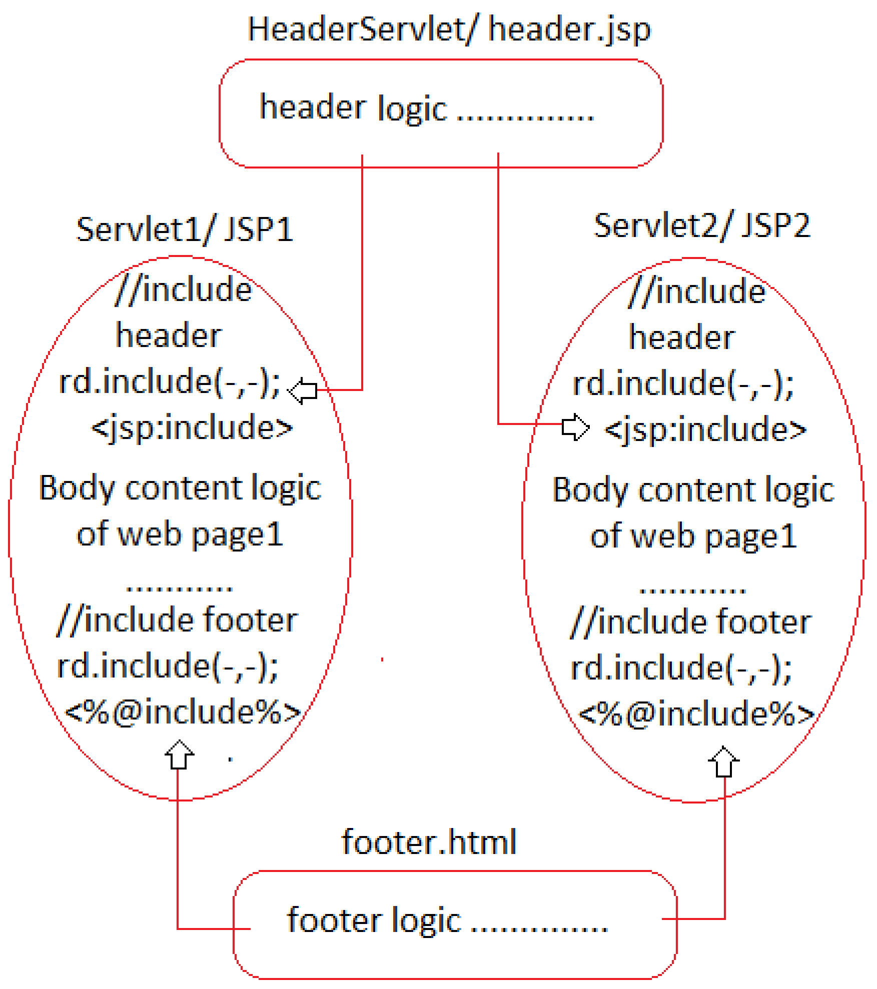
Problem:



- Webpages contain same header content, footer content and different body contents.
- Here header and footer logics are not reusable.

Solution:

- Use Composite view design pattern.
- Here each web pages comes as the outputs given by multiple web components together, so it is called composite view page.


HeaderServlet/ header.jsp
header logic ....
Servlet1/ JSP1
//include header rd.include(-,-);
<jsp:include>
Body content logic of web page1
....
//include footer rd.include(-,-);
<%@include%>
Servlet2/ JSP2
//include header rd.include(-,-);
<jsp:include>
Body content logic of web page1
....
//include footer rd.include(-,-);
<%@include%>
footer.html
footer logic ....

Note: Here header and footer logics are reusable logics.

- In Servlet programming we have to use rd.include(-,-) to include both static web components output and also the dynamic web components output.

- In JSP programming we have included tags
  - Static include/ Directive include/ Translation phase include:
    - <%@include file="...."%>
    - Given component code will be included to the JES class code (JSP Equivalent Servlet component) of source JSP page during the translation phase.

  - Dynamic include/ Action include/ Execution phase include:
    - <jsp:include page="....."%>
    - Given component output will be included to the output of source JSP page dynamically at runtime (execution phase).

View Helper Design Pattern

Problem:

- Placing Java code in JSP page is having following limitations,
    a. Kills the readability.
    b. Editing JSP code for UI developers becomes complex.
    c. It is against of JSP programming principles.
    d. The Java code placed in JSP pages is not reusable across the multiple jsp pages.
    and etc.

Solution:

- To overcome these problems, avoid or minimize Java code (scripting code) in JSP page i.e., we should use JSP pages as tag-based pages. For this we should develop JSP pages having following tags and EL.
    - HTML tags
    - JSP standard tags
    - EL
    - JSTL tags
    - Third JSP library tags
    - Custom JSP library tags
- Here the classes that are presenting the tags of JSTL, third party JSP libraries, custom JSP libraries as JSP tag handler classes are called View Helper components because they are helping view components to avoid or minimize Java code in JSP pages.

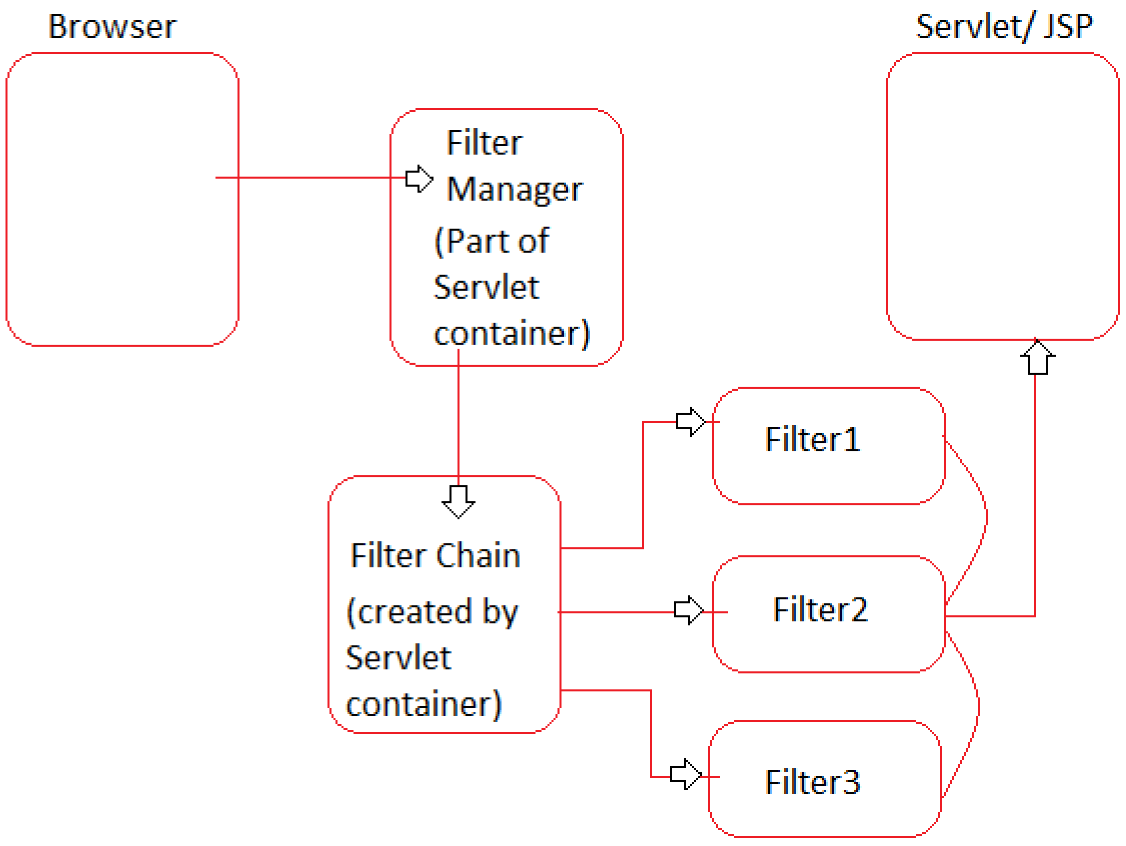
Advantages of avoiding Java code from JSP pages (implementing view helper design pattern):

- ➢ JSP pages code becomes more readable.
- ➢ Suitable for both Java developers and UI developers.
- ➢ JSP pages coding will happen according to JSP programming principle called tag-based programming.
- ➢ The Java code represented JSP tags becomes reusable across the multiple JSP tags.
- ➢ More of industry standard.

Intercepting Filter Design Pattern

- Intercepting Filter is special web component of web application which can trap request or response or both having pre-request processing logics or post response generation or both.

- Intercepting filter logics will be applied web application without disturbing other existing web components of web application having ability to enable or disable on temporary basis.
- Working with Servlet filters in Java web application is implementing intercepting filter design pattern.


Browser
Servlet/ JSP
Filter Manager (Part of Servlet container)
Filter Chain (created by Servlet container)
Filter1
Filter2
Filter3

Example Filters:

- Authentication Filters (request filter)
- Logging and Auditing Filters (request filter)
- Image conversion Filters (response filter)
- Data compression Filters (response filter)
- Encryption Filters (response filter)
- Tokenizing Filters (request filter)
- Filters that trigger resource access events (request-response filter)
- XSL/T filters (response filter)
- Mime-type chain Filter (response filter)
- Browser Check Filter (request filter)
- Timings Checking Filter (request filter)
- Double Posting Preventing Filter (Request filter)
- Session Validating Filter (request filter)
- Performances Filter (request-response filter)

and etc.

Front Controller Design Pattern

Problem:

1. Java classes cannot take HTTP request directly from browser though they request processing logic
2. Writing multiple system services like security, logging, auditing and etc. in multiple web components without reusability is bad practice.
3. Navigation management means which request should go to which web components and keeping this navigation management logic in view components (JSPs) is bad practice.

Solution:

- To overcome these problems, use Front Controller design pattern having one Servlet component managing multiple responsibilities like applying common System services, Action management, Navigation management, View management, Model management and etc.

- System services: The mandatory services that should be applied on every request by trapping the requests are called System services.
  E.g., Security, Logging, Auditing and etc.
  - Security: Performs authentication (identity verification) + Authorization (checking access permissions).
  - Logging: Keeping track of various components that are participating in flow of execution is called logging.
  - Auditing: Keeping track of user activities is called auditing.

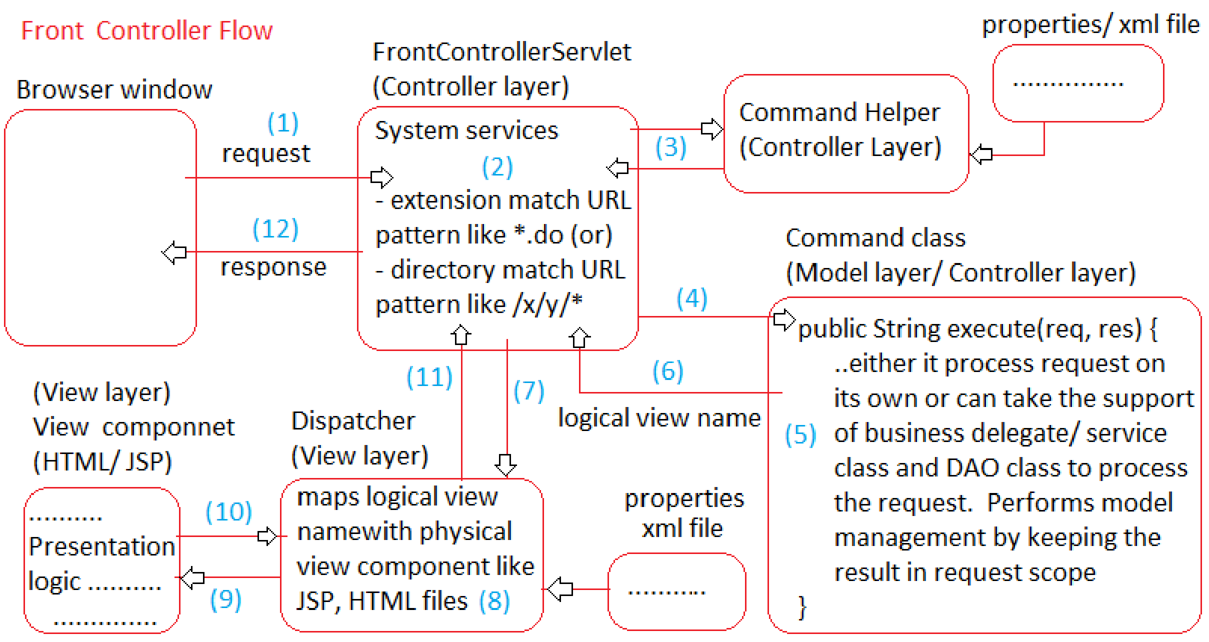
- Action management: Decides which request should go to which Java class (Action class/ command class/ controller class/ …) request processing logic is placed.

- Navigation management: Decides the end-to-end flow from request arrival to response generation.

- View management: After processing requests which Action class/ command class/ controller class result should processed and formatted by which view component will be decided by View Management.

- Model management: Keeping inputs coming to app and outputs/ results generated by the application in different scopes to provide visible and accessible in different places/ components is called Model Management.

Front Controller Flow


Browser window
(1)
request
(12)
response
FrontControllerServlet
(Controller layer)
System services
(2)
- extension match URL pattern like *.do (or)
- directory match URL pattern like /x/y/*
(3)
Command Helper
(Controller Layer)
properties/ xml file
....
Command class
(Model layer/ Controller layer)
(4)
public String execute(req, res) {
..either it process request on its own or can take the support of business delegate/ service class and DAO class to process the request. Performs model management by keeping the result in request scope
}
(5)
(6)
logical view name
(7)
(11)
Dispatcher
(View layer)
maps logical view namewith physical view component like JSP, HTML files (8)
properties xml file
....
(View layer)
View componnet
(HTML/ JSP)
.... Presentation logic ....
(10)
(9)

- ➢ The Servlet component that is configured with exact match URL pattern can take request coming from only one/ exact request URL.
- ➢ The Servlet component that is configured with extension match or directory match URL pattern can take request coming from different request URLs.

1. Browser give request.
2. Front Controller Servlet traps and takes the request and also applied common system services.
3. Front Controller Servlet hand overs the request to Command Hepler and gets command class object back, after mapping incoming request with command class.
4. Calls execute (-, -) (standard method) on command class object.
5. Process the request directly or indirectly and keeps results in request scope.
6. Execute (-, -) method command class returns logical view name to Front Controller Servlet.
7. Front Controller Servlet gives logical view name to Dispatcher component.
8. Dispatcher component map's logical view name with physical view name like JSP/ HTML components
9. Dispatcher uses rd.forward(-,-) to send the control to physical view component to process the results using presentation logics.

10. & 11. Control goes to Front Controller Servlet because of rd.forward(-,-).
12.The Front Controller Servlet writes the formatted results to browser as response.

Note: A Front Controller Servlet is entry, exit point for all requests and responses having capability to apply commons system services and to perform Action management, Navigation management and Model management and View management.

- Struts, Spring MVC, JSF and etc. are given based Front Controller design pattern and following MVC Architecture having readymade Servlet components as Front controller Servlet components
    - In Struts -----------> ActionServlet
    - In JSF ---------------> FacesServlet
    - In Spring MVC ---> DispatcherServlet

    and etc.

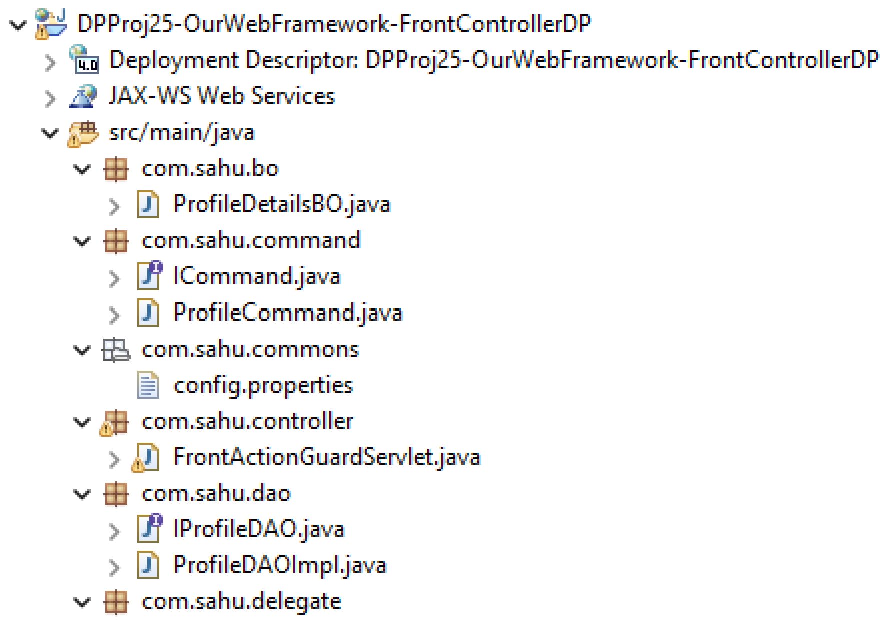
Q. When should we should use FrontController Servlet and when should we use Intercepting filters?

Ans.

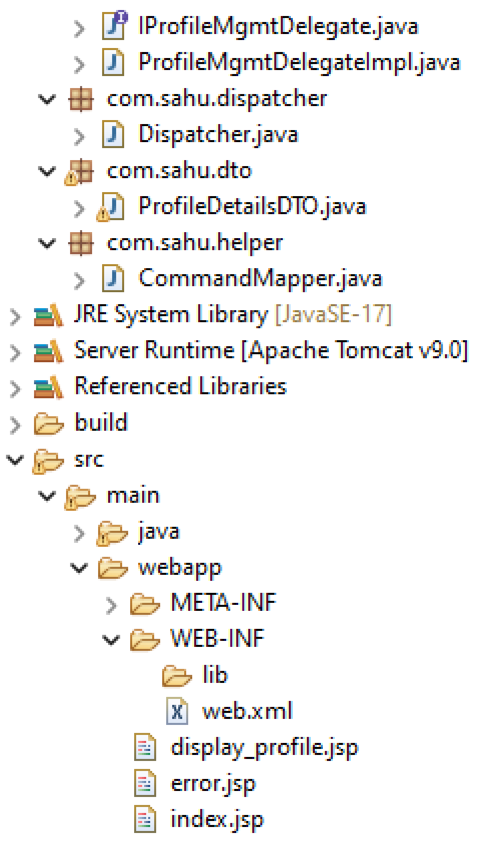
- If the services to be applied are mandatory services like System service (security, logging, auditing and etc.) then use FrontController Servlet (permanent component of web application).
- If the services to be applied are optional services like performance monitoring, browser checking and etc. which needs to be enabled or disabled based on the need then use Intercepting Filter (optional component of web application).

Note: If needed we can use both at a time.

Directory Structure of DPProj25-OurWebFramework-FrontControllerDP:



```
DPProj25-OurWebFramework-FrontControllerDP
  > Deployment Descriptor: DPProj25-OurWebFramework-FrontControllerDP
  > JAX-WS Web Services
  v src/main/java
    v com.sahu.bo
      > ProfileDetailsBO.java
    v com.sahu.command
      > ICommand.java
      > ProfileCommand.java
    v com.sahu.commons
        config.properties
    v com.sahu.controller
      > FrontActionGuardServlet.java
    v com.sahu.dao
      > IProfileDAO.java
      > ProfileDAOImpl.java
    v com.sahu.delegate
```


```
        > IProfileMgmtDelegate.java
        > ProfileMgmtDelegateImpl.java
      v com.sahu.dispatcher
        > Dispatcher.java
      v com.sahu.dto
        > ProfileDetailsDTO.java
      v com.sahu.helper
        > CommandMapper.java
  > JRE System Library [JavaSE-17]
  > Server Runtime [Apache Tomcat v9.0]
  > Referenced Libraries
  > build
  v src
    v main
      > java
      v webapp
        > META-INF
        v WEB-INF
            lib
            web.xml
          display_profile.jsp
          error.jsp
          index.jsp
```

- Develop the above directory structure as Dynamic Web Project, create all the packages, classes and files as shown in above structure.
- Add the following jar in CLASS PATH, BUILD PATH and Deployment Assembly.
  - [Lombok API], lombok-1.18.24.jar
- Then use the following code with in their respective file.

ProfileDetailsBO.java

```
package com.sahu.bo;

import java.time.LocalDateTime;

import lombok.Data;

@Data
public class ProfileDetailsBO {
	private int profileId;
	private String profileName;
	private String profileURL;
	private LocalDateTime DOB, DOJ;
}
```

IProfileDAO.java

```java
package com.sahu.dao;

import com.sahu.bo.ProfileDetailsBO;

public interface IProfileDAO {
	public ProfileDetailsBO getProfileById(int id) throws Exception;
}
```

ProfileDAOImpl.java

```java
package com.sahu.dao;

import java.time.LocalDateTime;

import com.sahu.bo.ProfileDetailsBO;

public class ProfileDAOImpl implements IProfileDAO {

	@Override
	public ProfileDetailsBO getProfileById(int id)  throws Exception {
		ProfileDetailsBO bo = new ProfileDetailsBO();
		//actually we should collect from DB s/w to put into BO class 
object
		if (id==1001) {
			bo.setProfileId(id);
			bo.setProfileName("Rajesh");
			bo.setProfileURL("https://fb.com/profiles/rajesh");
			bo.setDOB(LocalDateTime.of(1990,12,23,12,56));
			bo.setDOJ(LocalDateTime.of(2010,12,23,12,56));
		}
		else if (id==1002) {
			bo.setProfileId(id);
			bo.setProfileName("Anil");
			bo.setProfileURL("https://fb.com/profiles/anil");
			bo.setDOB(LocalDateTime.of(1980,12,23,12,56));
			bo.setDOJ(LocalDateTime.of(2011,12,23,12,56));
		}
		else
			throw new IllegalArgumentException("Invalid id - "+id);
		return bo;
	}

}
```

ProfileDetailsDTO.java

```java
package com.sahu.dto;

import java.io.Serializable;
import java.time.LocalDateTime;

import lombok.Data;

@Data
public class ProfileDetailsDTO implements Serializable {
	private int profileId;
	private String profileName;
	private String profileURL;
	private LocalDateTime DOB, DOJ;
}
```

IProfileMgmtDelegate.java

```java
package com.sahu.delegate;

import com.sahu.dto.ProfileDetailsDTO;

public interface IProfileMgmtDelegate {
	public ProfileDetailsDTO fetchProfileById(int id)  throws Exception;
}
```

ProfileMgmtDelegateImpl.java

```java
package com.sahu.delegate;

import com.sahu.bo.ProfileDetailsBO;
import com.sahu.dao.IProfileDAO;
import com.sahu.dao.ProfileDAOImpl;
import com.sahu.dto.ProfileDetailsDTO;

```

```
public class ProfileMgmtDelegateImpl implements IProfileMgmtDelegate {

	private IProfileDAO dao = null;

	public ProfileMgmtDelegateImpl() {
		dao = new ProfileDAOImpl();
	}

	@Override
	public ProfileDetailsDTO fetchProfileById(int id)  throws Exception{
		// use DAO
		ProfileDetailsBO bo = dao.getProfileById(id);
		//convert bo class object to DTO class object
		ProfileDetailsDTO dto = new ProfileDetailsDTO();
		dto.setProfileId(bo.getProfileId());
		dto.setProfileName(bo.getProfileName());
		dto.setProfileURL(bo.getProfileURL());
		dto.setDOB(bo.getDOB());
		dto.setDOJ(bo.getDOJ());

		return dto;
	}

}
```

ICommand.java

```
package com.sahu.command;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public interface ICommand {
	public String execute(HttpServletRequest req, HttpServletResponse
res) throws Exception;
}
```

ProfileCommand.java

```
package com.sahu.command;
```

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.sahu.delegate.IProfileMgmtDelegate;
import com.sahu.delegate.ProfileMgmtDelegateImpl;
import com.sahu.dto.ProfileDetailsDTO;

public class ProfileCommand implements ICommand {

	private IProfileMgmtDelegate delegate;

	public ProfileCommand() {
		delegate = new ProfileMgmtDelegateImpl();
	}

	@Override
	public String execute(HttpServletRequest req, HttpServletResponse 
res) throws Exception {
		//use delegate
		ProfileDetailsDTO dto = 
delegate.fetchProfileById(Integer.parseInt(req.getParameter("id")));
		//keep result in request scope
		req.setAttribute("profileDTO", dto);
		//return logical view name
		return "show_profile";
	}

}
```

CommandMapper.java

```
package com.sahu.helper;

import java.util.ResourceBundle;

import javax.servlet.http.HttpServletRequest;

import com.sahu.command.ICommand;

public class CommandMapper {
        private static ResourceBundle bundle;

        static {
                bundle =
ResourceBundle.getBundle("com/sahu/commons/config");
        }

        public static ICommand mapRequestToCommand(String reqUrl,
HttpServletRequest req) throws Exception {
                //get command class name from properties by giving incoming
request URL
                String className = bundle.getString(reqUrl);
                //Load and instantiate command class
                Class<?> clazz  = Class.forName(className);
                ICommand command  = (ICommand)
(clazz.getDeclaredConstructors()[0]).newInstance();
                return command;
        }

}
```

config.properties

```
#request URIs to command class mapping
/get_profile.do=com.sahu.command.ProfileCommand

#Logical view name to physical view mapping
show_profile=/display_profile.jsp
```

Dispatcher.java

```
package com.sahu.dispatcher;

import java.util.ResourceBundle;

import javax.servlet.RequestDispatcher;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class Dispatcher {

	private static ResourceBundle bundle;

	static {
		bundle = 
ResourceBundle.getBundle("com/sahu/commons/config");
	}

	public static void dispatchToView(HttpServletRequest req, 
HttpServletResponse res, String lvn) throws Exception {
		//get physical view name based on lvn
		String pvn = bundle.getString(lvn);
		//forward to physical view using rd.forward(req, res)
		RequestDispatcher dispatcher = 
req.getRequestDispatcher(pvn);
		dispatcher.forward(req, res);
	}

}
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" 
xmlns="http://xmlns.jcp.org/xml/ns/javaee" 
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee 
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd" id="WebApp_ID" 
version="4.0">
  <display-name>DPProj25-OurWebFramework-FrontControllerDP</display-
name>
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

index.jsp

```
<div style="text-align: center;">
	<a href="get_profile.do?id=1001">Get 1001 Profile</a>
	<br>
	<a href="get_profile.do?id=1002">Get 1002 Profile</a>
</div>
```

display_profile.jsp

```
<%@ page isELIgnored="false"%>
<div style="text-align: center;">
	<div style="color: blue;">Profile details: ${profileDTO}</div>
	<br>
	<a href="index.jsp">Home</a>
</div>
```

error.jsp

```
<%@ page isELIgnored="false"%>
<div style="text-align: center;">
	<div style="color: red;">Internal Problem: Try again</div>
	<br>
	<a href="index.jsp">Home</a>
</div>
```

FrontActionGuardServlet.java

```
package com.sahu.controller;

import java.io.IOException;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.sahu.command.ICommand;
import com.sahu.dispatcher.Dispatcher;
import com.sahu.helper.CommandMapper;

@WebServlet("*.do")
public class FrontActionGuardServlet extends HttpServlet {

	public void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
		try {
			//read incoming request uri
```

```
                        String reqUri = request.getServletPath(); //gives only last
part of incoming request url
                        //get command class from CommandMapper
                        ICommand command =
CommandMapper.mapRequestToCommand(reqUri, request);
                        //invoke standard method on command class object
                        String lvn = command.execute(request, response);
                        //use dispatcher to get physical view name and forward
the controller to physical view
                        Dispatcher.dispatchToView(request, response, lvn);
                }
                catch (Exception e) {
                        e.printStackTrace();
                        RequestDispatcher rd =
request.getRequestDispatcher("/error.jsp");
                        rd.forward(request, response);
                }
        }

        public void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
                doGet(request, response);
        }

}
```

Single line statements about Design Patterns

A. Singleton: In any situation it allows JVM create one object.
B. Factory: Provides abstraction on object create process.
C. Factory method: If multiple factories are creating objects for same family classes and if you want impose some rules and guidelines on that process then we need to use factory method design pattern.
D. Abstract factory: Super factory or factory of factories. When we are working with multiple factories that creates objects for different family classes, to make sure that all objects are created through same factory go for Abstract factory.
E. Adaptor: To make the class of one module compatible to work with another module class we need to use adaptor.
F. Flyweight: It makes the programmer to utilities minimum objects for

maximum utilization by identify intrinsic (sharable) state and extrinsic state.

G. Decorator: Useful to decorate the object with additional facilities without using inheritance.
H. Builder: Allows to create complex as the combination of multiple sub objects having reusability of sub objects.
I. Template method: The class that contains method automating sequence of operations to complete the task having the flexibility of customizing certain operations.
J. Business Delegate: Helper to controller Servlet to convert
   a. To convert VO to BO.
   b. To translate technology specific exceptions into application specific exception.
   c. To perform Transaction management.
K. DAO: The class that separate persistence logic from other logics is called DAO.
L. Service Locator: The class that contains lookup code to get reference of external service component and maintains cache having to get reusability.
M. Session Facade: The special server-side external component that talks to multiple other external components to complete the task.
N. Message Facade: Same as Session Facade but gives asynchronous Communication.
O. Composite View: Recommended to take common logics (header and footer logics) in separate web components and to include their outputs in main web components.
P. Intercepting Filter: Web component having optional services that can be enabled disabled dynamically.
Q. Front Controller: Entry and exit point for all requests having mandatory system services.

--- The END ---