<https://www.javatpoint.com/jee-or-j2ee-design-patterns>

Design Patterns in Java

1. Creational

1.1 Singleton

1.2 Factory

1.3 Abstract Factory

1.4 Builder

1.5 Prototype

2. Structural

2.1 Adapter

2.2 Bridge

2.3 Composite

2.4 Decorator

2.5 Facade

2.6 Flyweight

2.7 Proxy

3. Behavioural

3.1 Chain of Responsibility

3.2 Command

3.3 Interpreter

3.4 Iterator

3.5 Mediator

3.6 Memento

3.7 Observer

3.8 State

3.9 Strategy

3.10 Template Method

3.11 Visitor

4. J2EE

4.1 Presentation Layer

1. Intercepting Filter

2. Front Controller

3. View Helper

4. Composite View

5. MVC

4.2 Business Layer

1. Business Delegate

2. Service Locator

3. Session Facade

4. Transfer Object

4.3 Integration Layer

1. Data Access Object

2. Web Service Broker

1. Creational
2. One and Only one instance of the class exists in JVM
3. Singleton class should provide a global access point to get the instance

1.1 Eager Initialization

1.2 Static Block Initialization

1.3 Lazy Initialization

1.4 Thread Safe Singleton

1.5 Bill Pugh Singleton Implementation

1.6 Break Singleton Using Reflection

1.7 Enum Singleton

1.8 Break Singleton Using Serialization

1.9 Break Singleton Using Clone/Cloning

1.10 Example of Singleton within JDK

1. Singleton

1. Create a private constructor

2. Create a private static variable which is the only instance of the class

3. Create a getInstance method which returns the instance of the class.

This acts as global access point for outer world to access singleton instance

4.Examples of Singleton Classes in JDk

1. Runtime Class (in java.lang package)
2. Desktop Class (in java.awt package)
3. System Class (in java.lang package)
4. Logging
5. Caching
6. ThreadPool (single instance)
   1. Using Eager Initialization

Problem:

Singleton instance is created even though we are not using it.

No Exception Handling/try catch outside

**package** com.kkjavatutorials;

**public** **class** Singleton {

**private** **static** **final** Singleton ***INSTANCE*** = **new** Singleton();

**private** Singleton() {

}

**public** **static** Singleton getInstance() {

**return** ***INSTANCE***;

}

}

**package** com.kkjavatutorials;

**public** **class** Tester {

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

Singleton singletonInstance1 = Singleton.*getInstance*();

Singleton singletonInstance2 = Singleton.*getInstance*();

System.***out***.println(singletonInstance1.hashCode());

System.***out***.println(singletonInstance2.hashCode());

}

}

* 1. Static Block Initialization

Solution:

We can do exception handling in Static Block initialization

**package** com.kkjavatutorials;

**public** **class** Singleton {

**private** **static** Singleton *INSTANCE* = **null**;

**static** {

**try** {

**if** (*INSTANCE* == **null**) {

*INSTANCE* = **new** Singleton();

}

} **catch** (Exception e) {

e.printStackTrace();

}

}

**private** Singleton() {

}

**public** **static** Singleton getInstance() {

**return** *INSTANCE*;

}

}

**package** com.kkjavatutorials;

**public** **class** Tester {

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

Singleton singletonInstance1 = Singleton.*getInstance*();

Singleton singletonInstance2 = Singleton.*getInstance*();

System.***out***.println(singletonInstance1.hashCode());

System.***out***.println(singletonInstance2.hashCode());

}

}

* 1. Lazy Initialization

Problem:

This will not work in a Multi-Threaded environment Both threads will get different instances of singleton

**package** com.kkjavatutorials;

**public** **class** Singleton {

**private** **static** Singleton *INSTANCE* = **null**;

**private** Singleton() {

}

**public** **static** Singleton getInstance() {

**if** (*INSTANCE* == **null**) {

*INSTANCE* = **new** Singleton();

}

**return** *INSTANCE*;

}

}

**package** com.kkjavatutorials;

**public** **class** Tester {

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

Singleton singletonInstance1 = Singleton.*getInstance*();

Singleton singletonInstance2 = Singleton.*getInstance*();

System.***out***.println(singletonInstance1.hashCode());

System.***out***.println(singletonInstance2.hashCode());

}

}

1.4 Thread-Safe Singleton

Choose 2 one

1. Use synchronized keyword to make Singleton Thread Safe

(This reduces performance by using synchronized at method level)

1. Use double check locking

Solution:

When someone calls getInstance() then only Inner class gets loaded in the memory so it is Lazy Loading

**package** com.kkjavatutorials;

**public** **class** Singleton {

**private** **static** Singleton *INSTANCE* = **null**;

**private** Singleton() {

}

**public** **static** Singleton getInstance() {

**if** (*INSTANCE* == **null**) {

**synchronized** (Singleton.**class**) {

**if** (*INSTANCE* == **null**) {

**try** {

Thread.*sleep*(2000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

*INSTANCE* = **new** Singleton();

}

}

}

**return** *INSTANCE*;

}

}

**package** com.kkjavatutorials;

**public** **class** MyThread **implements** Runnable {

@Override

**public** **void** run() {

Singleton singleton = Singleton.*getInstance*();

System.***out***.println(Thread.*currentThread*().getName() + " " + singleton.hashCode());

}

}

**package** com.kkjavatutorials;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**public** **class** Tester {

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

ExecutorService executorService = **null**;

MyThread myThread = **new** MyThread();

**try** {

executorService = Executors.*newFixedThreadPool*(2);

executorService.execute(myThread);

executorService.execute(myThread);

executorService.execute(myThread);

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

**if** (executorService != **null**) {

executorService.shutdown();

}

}

}

}

1.5 Bill Pugh Singleton Implementation or Using Holder Class

Problem:

1. Prior to Java 1.5 in Double Check Locking in 1.5 too many threads will cause problem

**package** com.kkjavatutorials;

**public** **class** Singleton {

**private** Singleton() {

}

**private** **static** **class** SingletonHolder {

**private** **static** **final** Singleton ***INSTANCE*** = **new** Singleton();

}

**public** **static** Singleton getInstance() {

**return** SingletonHolder.***INSTANCE***;

}

}

**package** com.kkjavatutorials;

**public** **class** Tester {

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

Singleton singleton1 = Singleton.*getInstance*();

Singleton singleton2 = Singleton.*getInstance*();

System.***out***.println(singleton1.hashCode());

System.***out***.println(singleton2.hashCode());

}

}

* + 1. Destroy Singleton Using Reflection

Problem:

Using Reflection we can break Singleton

Solution:

Using Enum

**package** com.kkjavatutorials;

**public** **class** Singleton {

**private** **static** Singleton *INSTANCE* = **null**;

**private** Singleton() {

}

**public** **static** Singleton getInstance() {

**if** (*INSTANCE* == **null**) {

*INSTANCE* = **new** Singleton();

}

**return** *INSTANCE*;

}

}

**package** com.kkjavatutorials;

**import** java.lang.reflect.Constructor;

**import** java.lang.reflect.InvocationTargetException;

**public** **class** Tester {

**public** **static** **void** main(String[] args) **throws** InstantiationException, IllegalAccessException, IllegalArgumentException, InvocationTargetException {

Singleton singleton1 = Singleton.*getInstance*();

Singleton singleton2 = **null**;

Constructor<?>[] constructors = Singleton.**class**.getDeclaredConstructors();

**for** (Constructor<?> constructor : constructors) {

constructor.setAccessible(**true**);

Object object = constructor.newInstance();

singleton2 = (Singleton)object;

**break**;

}

System.***out***.println(singleton1.hashCode());

System.***out***.println(singleton2.hashCode());

}

}

* 1. Using Enum to implement Singleton

Problem :

Enum always use Eager Initialization

Lazy Initialization is not provided using Enum

**package** com.kkjavatutorials;

**public** **enum** Singleton {

***INSTANCE***;

**public** String welcome(String name) {

**return** "Welcome to Enum Using Singleton " + name;

}

}

**package** com.kkjavatutorials;

**import** java.lang.reflect.InvocationTargetException;

**public** **class** Tester {

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

**throws** InstantiationException, IllegalAccessException, IllegalArgumentException, InvocationTargetException {

Singleton singleton1 = Singleton.***INSTANCE***;

Singleton singleton2 = Singleton.***INSTANCE***;

System.***out***.println(singleton1.hashCode());

System.***out***.println(singleton2.hashCode());

String name = "Pushkar";

System.***out***.println(singleton1.welcome(name));

}

}

1.8 Break Singleton Using Serialization / Singleton For distributed System

Problem:

Using Serialization whenever we deserialize the instance it will create a new instance

**package** com.kkjavatutorials;

**import** java.io.Serializable;

**public** **class** Singleton **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** **static** Singleton *INSTANCE* = **null**;

**private** Singleton() {

}

**public** **static** Singleton getInstance() {

**if** (*INSTANCE* == **null**) {

*INSTANCE* = **new** Singleton();

}

**return** *INSTANCE*;

}

}

**package** com.kkjavatutorials;

**import** java.io.FileInputStream;

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**import** java.io.ObjectInputStream;

**import** java.io.ObjectOutputStream;

**public** **class** Tester {

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

Singleton singleton1 = Singleton.*getInstance*();

Singleton singleton2 = **null**;

ObjectOutputStream oos = **null**;

FileOutputStream fos = **null**;

**try** {

fos = **new** FileOutputStream("singleton.ser");

oos = **new** ObjectOutputStream(fos);

oos.writeObject(singleton1);

oos.flush();

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

**if** (oos != **null**) {

oos.close();

}

}

ObjectInputStream ois = **null**;

FileInputStream fis = **null**;

**try** {

fis = **new** FileInputStream("singleton.ser");

ois = **new** ObjectInputStream(fis);

singleton2 = (Singleton) ois.readObject();

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

**if** (ois != **null**)

**try** {

{

ois.close();

}

} **catch** (IOException e) {

e.printStackTrace();

}

}

System.***out***.println(singleton1.hashCode());

System.***out***.println(singleton2.hashCode());

}

}

Solution:

Override readResolve() method in Singleton Class.

readResolve() will be called by JVM method signature and name will be

exact can use public/private/protected for readResolve()

**public** Object readResolve() {

**return** *getInstance*();

}

**package** com.kkjavatutorials;

**import** java.io.Serializable;

**public** **class** Singleton **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** **static** Singleton *INSTANCE* = **null**;

**private** Singleton() {

}

**public** **static** Singleton getInstance() {

**if** (*INSTANCE* == **null**) {

*INSTANCE* = **new** Singleton();

}

**return** *INSTANCE*;

}

**public** Object readResolve() {

**return** *getInstance*();

}

}

**package** com.kkjavatutorials;

**import** java.io.FileInputStream;

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**import** java.io.ObjectInputStream;

**import** java.io.ObjectOutputStream;

**public** **class** Tester {

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

Singleton singleton1 = Singleton.*getInstance*();

Singleton singleton2 = **null**;

ObjectOutputStream oos = **null**;

FileOutputStream fos = **null**;

**try** {

fos = **new** FileOutputStream("singleton.ser");

oos = **new** ObjectOutputStream(fos);

oos.writeObject(singleton1);

oos.flush();

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

**if** (oos != **null**) {

oos.close();

}

}

ObjectInputStream ois = **null**;

FileInputStream fis = **null**;

**try** {

fis = **new** FileInputStream("singleton.ser");

ois = **new** ObjectInputStream(fis);

singleton2 = (Singleton) ois.readObject();

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

**if** (ois != **null**)

**try** {

{

ois.close();

}

} **catch** (IOException e) {

e.printStackTrace();

}

}

System.***out***.println(singleton1.hashCode());

System.***out***.println(singleton2.hashCode());

}

}

1.9 Break Singleton Using Clone/Cloning

Problem:

**package** com.kkjavatutorials;

**public** **class** Singleton **implements** Cloneable {

**private** **static** Singleton *INSTANCE* = **null**;

**private** Singleton() {

}

**public** **static** Singleton getInstance() {

**if** (*INSTANCE* == **null**) {

*INSTANCE* = **new** Singleton();

}

**return** *INSTANCE*;

}

// clone() from Object class

@Override

**public** Object clone() **throws** CloneNotSupportedException {

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

}

}

**package** com.kkjavatutorials;

**import** java.io.IOException;

**public** **class** Tester {

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

Singleton singleton1 = Singleton.*getInstance*();

Singleton singleton2 = (Singleton) singleton1.clone();

System.***out***.println(singleton1.hashCode());

System.***out***.println(singleton2.hashCode());

}

}

Solution:

Just throw a new CloneNotSupportedException

// clone() from Object class

@Override

**public** Object clone() **throws** CloneNotSupportedException {

//return super.clone();

**throw** **new** CloneNotSupportedException("You cannot create clone of Singleton");

}

**package** com.kkjavatutorials;

**public** **class** Singleton **implements** Cloneable {

**private** **static** Singleton *INSTANCE* = **null**;

**private** Singleton() {

}

**public** **static** Singleton getInstance() {

**if** (*INSTANCE* == **null**) {

*INSTANCE* = **new** Singleton();

}

**return** *INSTANCE*;

}

// clone() from Object class

@Override

**public** Object clone() **throws** CloneNotSupportedException {

//return super.clone();

**throw** **new** CloneNotSupportedException("You cannot create clone of Singleton");

}

}

**package** com.kkjavatutorials;

**import** java.io.IOException;

**public** **class** Tester {

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

Singleton singleton1 = Singleton.*getInstance*();

Singleton singleton2 = (Singleton) singleton1.clone();

System.***out***.println(singleton1.hashCode());

System.***out***.println(singleton2.hashCode());

}

}

Examples of Singleton Classes in JDk

1. Runtime Class (in java.lang package)
2. Desktop Class (in java.awt package)
3. System Class (in java.lang package)
4. Logging
5. Caching
6. ThreadPool (single instance)
7. Factory
8. Factory Design Pattern is used when a super class with multiple sub classes and based in input return one of the sub class object
9. Factory will create object of sub class
10. Factory design pattern : Code for interface than implementation
11. Example of factory Design Pattern
12. java.util.Calendar (getInstance method)
13. java.text.NumberFormat (getInstance method)
14. java.nio.charset.Charset (forName() method)
15. java.util.EnumSet (of() method)
16. java.lang.Class (forName() method)
17. java.sql.DriverManager (getConnection() method)
18. java.net.URL (openConnection() method)
19. java.lang.Class (newInstance() method)

**package** com.kkjavatutorials;

**public** **interface** Computer {

**public** **abstract** String ram();

**public** **abstract** String cpu();

**public** **abstract** String hdd();

**public** **boolean** isGraphicsEnabled();

**public** **boolean** isBluetoothEnabled();

}

**package** com.kkjavatutorials;

**public** **enum** ComputerType {

***PC***, ***SERVER***, ***LAPTOP***;

}

**package** com.kkjavatutorials;

**public** **class** PC **implements** Computer {

**public** String ram;

**public** String cpu;

**public** String hdd;

**public** **boolean** isGraphicsEnabled;

**public** **boolean** isBluetoothEnabled;

**public** PC() {

}

**public** PC(String ram, String cpu, String hdd, **boolean** isGraphicsEnabled, **boolean** isBluetoothEnabled) {

**super**();

**this**.ram = ram;

**this**.cpu = cpu;

**this**.hdd = hdd;

**this**.isGraphicsEnabled = isGraphicsEnabled;

**this**.isBluetoothEnabled = isBluetoothEnabled;

}

@Override

**public** String ram() {

**return** **this**.ram;

}

@Override

**public** String cpu() {

**return** **this**.cpu;

}

@Override

**public** String hdd() {

**return** hdd;

}

@Override

**public** **boolean** isGraphicsEnabled() {

**return** isGraphicsEnabled;

}

@Override

**public** **boolean** isBluetoothEnabled() {

**return** isBluetoothEnabled;

}

@Override

**public** String toString() {

**return** "PC [ram=" + ram + ", cpu=" + cpu + ", hdd=" + hdd + ", isGraphicsEnabled=" + isGraphicsEnabled

+ ", isBluetoothEnabled=" + isBluetoothEnabled + ", ram()=" + ram() + ", cpu()=" + cpu() + ", hdd()="

+ hdd() + ", isGraphicsEnabled()=" + isGraphicsEnabled() + ", isBluetoothEnabled()="

+ isBluetoothEnabled() + ", getClass()=" + getClass() + ", hashCode()=" + hashCode() + ", toString()="

+ **super**.toString() + "]";

}

}

**package** com.kkjavatutorials;

**public** **class** Server **implements** Computer {

**public** String ram;

**public** String cpu;

**public** String hdd;

**public** **boolean** isGraphicsEnabled;

**public** **boolean** isBluetoothEnabled;

**public** Server() {

}

**public** Server(String ram, String cpu, String hdd, **boolean** isGraphicsEnabled, **boolean** isBluetoothEnabled) {

**super**();

**this**.ram = ram;

**this**.cpu = cpu;

**this**.hdd = hdd;

**this**.isGraphicsEnabled = isGraphicsEnabled;

**this**.isBluetoothEnabled = isBluetoothEnabled;

}

@Override

**public** String ram() {

**return** **this**.ram;

}

@Override

**public** String cpu() {

**return** **this**.cpu;

}

@Override

**public** String hdd() {

**return** hdd;

}

@Override

**public** **boolean** isGraphicsEnabled() {

**return** isGraphicsEnabled;

}

@Override

**public** **boolean** isBluetoothEnabled() {

**return** isBluetoothEnabled;

}

@Override

**public** String toString() {

**return** "PC [ram=" + ram + ", cpu=" + cpu + ", hdd=" + hdd + ", isGraphicsEnabled=" + isGraphicsEnabled

+ ", isBluetoothEnabled=" + isBluetoothEnabled + ", ram()=" + ram() + ", cpu()=" + cpu() + ", hdd()="

+ hdd() + ", isGraphicsEnabled()=" + isGraphicsEnabled() + ", isBluetoothEnabled()="

+ isBluetoothEnabled() + ", getClass()=" + getClass() + ", hashCode()=" + hashCode() + ", toString()="

+ **super**.toString() + "]";

}

}

**package** com.kkjavatutorials;

**public** **class** Laptop **implements** Computer {

**public** String ram;

**public** String cpu;

**public** String hdd;

**public** **boolean** isGraphicsEnabled;

**public** **boolean** isBluetoothEnabled;

**public** Laptop() {

}

**public** Laptop(String ram, String cpu, String hdd, **boolean** isGraphicsEnabled, **boolean** isBluetoothEnabled) {

**super**();

**this**.ram = ram;

**this**.cpu = cpu;

**this**.hdd = hdd;

**this**.isGraphicsEnabled = isGraphicsEnabled;

**this**.isBluetoothEnabled = isBluetoothEnabled;

}

@Override

**public** String ram() {

**return** **this**.ram;

}

@Override

**public** String cpu() {

**return** **this**.cpu;

}

@Override

**public** String hdd() {

**return** hdd;

}

@Override

**public** **boolean** isGraphicsEnabled() {

**return** isGraphicsEnabled;

}

@Override

**public** **boolean** isBluetoothEnabled() {

**return** isBluetoothEnabled;

}

@Override

**public** String toString() {

**return** "PC [ram=" + ram + ", cpu=" + cpu + ", hdd=" + hdd + ", isGraphicsEnabled=" + isGraphicsEnabled

+ ", isBluetoothEnabled=" + isBluetoothEnabled + ", ram()=" + ram() + ", cpu()=" + cpu() + ", hdd()="

+ hdd() + ", isGraphicsEnabled()=" + isGraphicsEnabled() + ", isBluetoothEnabled()="

+ isBluetoothEnabled() + ", getClass()=" + getClass() + ", hashCode()=" + hashCode() + ", toString()="

+ **super**.toString() + "]";

}

}

**package** com.kkjavatutorials;

**public** **class** ComputerFactory {

**private** ComputerFactory() {

}

**public** **static** Computer getComputer(ComputerType computerType, String ram, String cpu, String hdd,

**boolean** isGraphicsEnabled, **boolean** isBluetoothEnabled) {

**switch** (computerType) {

**case** ***PC***:

**return** **new** PC(ram, cpu, hdd, isGraphicsEnabled, isBluetoothEnabled);

**case** ***SERVER***:

**return** **new** Server(ram, cpu, hdd, isGraphicsEnabled, isBluetoothEnabled);

**case** ***LAPTOP***:

**return** **new** Laptop(ram, cpu, hdd, isGraphicsEnabled, isBluetoothEnabled);

**default**:

**throw** **new** RuntimeException("Invalid Computer Type");

}

}

}

**package** com.kkjavatutorials;

**public** **class** Tester {

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

Computer pc = ComputerFactory.*getComputer*(ComputerType.***PC***, "16GB", "2.2Ghz", "2Tb", **true**, **true**);

System.***out***.println("Pc Config: " + pc);

Computer server = ComputerFactory.*getComputer*(ComputerType.***SERVER***, "32GB", "2.2Ghz", "100Tb", **true**, **true**);

System.***out***.println("Server Config: " + server);

Computer laptop = ComputerFactory.*getComputer*(ComputerType.***LAPTOP***, "8GB", "2.2Ghz", "1Tb", **true**, **true**);

System.***out***.println("Laptop Config: " + laptop);

}

}

1. Abstract Factory
2. Abstract Factory is Similar to Factory Design Pattern.

2 Factory of Factories

3 Whenever you need another level of abstraction

1. Example of AbstractFactory Design Pattern
   1. javax.xml.parsers.DocumentBuilderFactory (newInstance())
   2. javax.xml.transform.TransformerFactory (newInstance())
   3. javax.xml.xpath.XPathFactory (newInstance())

**package** com.kkjavatutorials;

**public** **interface** Computer {

**public** **abstract** String ram();

**public** **abstract** String cpu();

**public** **abstract** String hdd();

**public** **boolean** isGraphicsEnabled();

**public** **boolean** isBluetoothEnabled();

}

**package** com.kkjavatutorials;

**public** **class** PC **implements** Computer {

**public** String ram;

**public** String cpu;

**public** String hdd;

**public** **boolean** isGraphicsEnabled;

**public** **boolean** isBluetoothEnabled;

**public** PC() {

}

**public** PC(String ram, String cpu, String hdd, **boolean** isGraphicsEnabled, **boolean** isBluetoothEnabled) {

**super**();

**this**.ram = ram;

**this**.cpu = cpu;

**this**.hdd = hdd;

**this**.isGraphicsEnabled = isGraphicsEnabled;

**this**.isBluetoothEnabled = isBluetoothEnabled;

}

@Override

**public** String ram() {

**return** **this**.ram;

}

@Override

**public** String cpu() {

**return** **this**.cpu;

}

@Override

**public** String hdd() {

**return** hdd;

}

@Override

**public** **boolean** isGraphicsEnabled() {

**return** isGraphicsEnabled;

}

@Override

**public** **boolean** isBluetoothEnabled() {

**return** isBluetoothEnabled;

}

@Override

**public** String toString() {

**return** "PC [ram=" + ram + ", cpu=" + cpu + ", hdd=" + hdd + ", isGraphicsEnabled=" + isGraphicsEnabled

+ ", isBluetoothEnabled=" + isBluetoothEnabled + ", ram()=" + ram() + ", cpu()=" + cpu() + ", hdd()="

+ hdd() + ", isGraphicsEnabled()=" + isGraphicsEnabled() + ", isBluetoothEnabled()="

+ isBluetoothEnabled() + ", getClass()=" + getClass() + ", hashCode()=" + hashCode() + ", toString()="

+ **super**.toString() + "]";

}

}

**package** com.kkjavatutorials;

**public** **class** Server **implements** Computer {

**public** String ram;

**public** String cpu;

**public** String hdd;

**public** **boolean** isGraphicsEnabled;

**public** **boolean** isBluetoothEnabled;

**public** Server() {

}

**public** Server(String ram, String cpu, String hdd, **boolean** isGraphicsEnabled, **boolean** isBluetoothEnabled) {

**super**();

**this**.ram = ram;

**this**.cpu = cpu;

**this**.hdd = hdd;

**this**.isGraphicsEnabled = isGraphicsEnabled;

**this**.isBluetoothEnabled = isBluetoothEnabled;

}

@Override

**public** String ram() {

**return** **this**.ram;

}

@Override

**public** String cpu() {

**return** **this**.cpu;

}

@Override

**public** String hdd() {

**return** hdd;

}

@Override

**public** **boolean** isGraphicsEnabled() {

**return** isGraphicsEnabled;

}

@Override

**public** **boolean** isBluetoothEnabled() {

**return** isBluetoothEnabled;

}

@Override

**public** String toString() {

**return** "PC [ram=" + ram + ", cpu=" + cpu + ", hdd=" + hdd + ", isGraphicsEnabled=" + isGraphicsEnabled

+ ", isBluetoothEnabled=" + isBluetoothEnabled + ", ram()=" + ram() + ", cpu()=" + cpu() + ", hdd()="

+ hdd() + ", isGraphicsEnabled()=" + isGraphicsEnabled() + ", isBluetoothEnabled()="

+ isBluetoothEnabled() + ", getClass()=" + getClass() + ", hashCode()=" + hashCode() + ", toString()="

+ **super**.toString() + "]";

}

}

**package** com.kkjavatutorials;

**public** **class** Laptop **implements** Computer {

**public** String ram;

**public** String cpu;

**public** String hdd;

**public** **boolean** isGraphicsEnabled;

**public** **boolean** isBluetoothEnabled;

**public** Laptop() {

}

**public** Laptop(String ram, String cpu, String hdd, **boolean** isGraphicsEnabled, **boolean** isBluetoothEnabled) {

**super**();

**this**.ram = ram;

**this**.cpu = cpu;

**this**.hdd = hdd;

**this**.isGraphicsEnabled = isGraphicsEnabled;

**this**.isBluetoothEnabled = isBluetoothEnabled;

}

@Override

**public** String ram() {

**return** **this**.ram;

}

@Override

**public** String cpu() {

**return** **this**.cpu;

}

@Override

**public** String hdd() {

**return** hdd;

}

@Override

**public** **boolean** isGraphicsEnabled() {

**return** isGraphicsEnabled;

}

@Override

**public** **boolean** isBluetoothEnabled() {

**return** isBluetoothEnabled;

}

@Override

**public** String toString() {

**return** "PC [ram=" + ram + ", cpu=" + cpu + ", hdd=" + hdd + ", isGraphicsEnabled=" + isGraphicsEnabled

+ ", isBluetoothEnabled=" + isBluetoothEnabled + ", ram()=" + ram() + ", cpu()=" + cpu() + ", hdd()="

+ hdd() + ", isGraphicsEnabled()=" + isGraphicsEnabled() + ", isBluetoothEnabled()="

+ isBluetoothEnabled() + ", getClass()=" + getClass() + ", hashCode()=" + hashCode() + ", toString()="

+ **super**.toString() + "]";

}

}

**package** com.kkjavatutorials;

**public** **class** PCFactory **implements** AbstractComputerFactory {

**public** String ram;

**public** String cpu;

**public** String hdd;

**public** **boolean** isGraphicsEnabled;

**public** **boolean** isBluetoothEnabled;

**public** PCFactory(String ram, String cpu, String hdd, **boolean** isGraphicsEnabled, **boolean** isBluetoothEnabled) {

**super**();

**this**.ram = ram;

**this**.cpu = cpu;

**this**.hdd = hdd;

**this**.isGraphicsEnabled = isGraphicsEnabled;

**this**.isBluetoothEnabled = isBluetoothEnabled;

}

@Override

**public** Computer createComputer() {

**return** **new** PC(ram, cpu, hdd, isGraphicsEnabled, isBluetoothEnabled);

}

}

**package** com.kkjavatutorials;

**public** **class** ServerFactory **implements** AbstractComputerFactory {

**public** String ram;

**public** String cpu;

**public** String hdd;

**public** **boolean** isGraphicsEnabled;

**public** **boolean** isBluetoothEnabled;

**public** ServerFactory(String ram, String cpu, String hdd, **boolean** isGraphicsEnabled, **boolean** isBluetoothEnabled) {

**super**();

**this**.ram = ram;

**this**.cpu = cpu;

**this**.hdd = hdd;

**this**.isGraphicsEnabled = isGraphicsEnabled;

**this**.isBluetoothEnabled = isBluetoothEnabled;

}

@Override

**public** Computer createComputer() {

**return** **new** Server(ram, cpu, hdd, isGraphicsEnabled, isBluetoothEnabled);

}

}

**package** com.kkjavatutorials;

**public** **class** LaptopFactory **implements** AbstractComputerFactory {

**public** String ram;

**public** String cpu;

**public** String hdd;

**public** **boolean** isGraphicsEnabled;

**public** **boolean** isBluetoothEnabled;

**public** LaptopFactory(String ram, String cpu, String hdd, **boolean** isGraphicsEnabled, **boolean** isBluetoothEnabled) {

**super**();

**this**.ram = ram;

**this**.cpu = cpu;

**this**.hdd = hdd;

**this**.isGraphicsEnabled = isGraphicsEnabled;

**this**.isBluetoothEnabled = isBluetoothEnabled;

}

@Override

**public** Computer createComputer() {

**return** **new** Laptop(ram, cpu, hdd, isGraphicsEnabled, isBluetoothEnabled);

}

}

**package** com.kkjavatutorials;

**public** **interface** AbstractComputerFactory {

**public** **abstract** Computer createComputer();

}

**package** com.kkjavatutorials;

**public** **class** ComputerFactory {

**private** ComputerFactory() {

}

**public** **static** Computer getComputer(AbstractComputerFactory abstractComputerFactory) {

**return** abstractComputerFactory.createComputer();

}

}

**package** com.kkjavatutorials;

**public** **class** Tester {

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

Computer pc = ComputerFactory.*getComputer*(**new** PCFactory("16GB", "2.2Ghz", "2Tb", **true**, **true**));

System.***out***.println("PC Config: " + pc);

Computer server = ComputerFactory.*getComputer*(**new** ServerFactory("32GB", "2.2Ghz", "100Tb", **true**, **true**));

System.***out***.println("Server Config: " + server);

Computer laptop = ComputerFactory.*getComputer*(**new** LaptopFactory("8GB", "2.2Ghz", "1Tb", **true**, **true**));

System.***out***.println("Laptop Config: " + laptop);

}

}

1. Builder
2. Builder is used when object contains lot of attributes
3. We have to pass many arguments from client program
4. When object is very heavy and Object creation is complex
5. When some parameters are optional we have to pass NULL in factory
6. Example of Builder Design Pattern
7. StringBuilder append()
8. StringBuffer append()
9. java.util.stream.Stream.Builder
10. java.nio.ByteBuffer(shortBuffer, LongBuffer, FloatBuffer, DoubleBuffer, IntBuffer, CharBuffer
11. BufferedWriter, CharArrayWriter, FilterWriter

**package** com.kkjavatutorials;

**public** **class** Laptop {

**private** String ram;

**private** String cpu;

**private** String hdd;

**private** **boolean** isGraphicsEnabled;

**private** **boolean** isBluetoothEnabled;

// 1. Make constructor private and pass BUilder

**private** Laptop(LaptopBuilder builder) {

**this**.ram = builder.ram;

**this**.cpu = builder.cpu;

**this**.hdd = builder.hdd;

**this**.isGraphicsEnabled = builder.isGraphicsEnabled;

**this**.isBluetoothEnabled = builder.isBluetoothEnabled;

}

// 2. Only create getters

**public** String getRam() {

**return** ram;

}

**public** String getCpu() {

**return** cpu;

}

**public** String getHdd() {

**return** hdd;

}

**public** **boolean** isGraphicsEnabled() {

**return** isGraphicsEnabled;

}

**public** **boolean** isBluetoothEnabled() {

**return** isBluetoothEnabled;

}

@Override

**public** String toString() {

**return** "Laptop [ram=" + ram + ", cpu=" + cpu + ", hdd=" + hdd + ", isGraphicsEnabled=" + isGraphicsEnabled

+ ", isBluetoothEnabled=" + isBluetoothEnabled + "]";

}

// Create a nested public static LaptopBuilder class

**public** **static** **class** LaptopBuilder {

// 1 copy all parameters in outer class

// mandatory parameter

**private** String ram;

**private** String cpu;

**private** String hdd;

// optional parameter

**private** **boolean** isGraphicsEnabled;

**private** **boolean** isBluetoothEnabled;

// provide manadatory parameter in consructor

**public** LaptopBuilder(String ram, String cpu, String hdd) {

**super**();

**this**.ram = ram;

**this**.cpu = cpu;

**this**.hdd = hdd;

}

// provide setter for optional parameter and return LaptopBuilder Object

**public** LaptopBuilder setGraphicsEnabled(**boolean** isGraphicsEnabled) {

**this**.isGraphicsEnabled = isGraphicsEnabled;

**return** **this**;

}

**public** LaptopBuilder setBluetoothEnabled(**boolean** isBluetoothEnabled) {

**this**.isBluetoothEnabled = isBluetoothEnabled;

**return** **this**;

}

// public build method

**public** Laptop build() {

**return** **new** Laptop(**this**);

}

}

}

**package** com.kkjavatutorials;

**public** **class** Tester {

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

Laptop laptop1 = **new** Laptop.LaptopBuilder("8gb", "2.4 Hz", "1TB").build();

System.***out***.println("Laptop Config : " + laptop1);

Laptop laptop2 = **new** Laptop.LaptopBuilder("16gb", "2.6 Hz", "2TB").setBluetoothEnabled(**true**).build();

System.***out***.println("Laptop Config : " + laptop2);

Laptop laptop3 = **new** Laptop.LaptopBuilder("32gb", "2.8 Hz", "4TB").setBluetoothEnabled(**true**)

.setGraphicsEnabled(**true**).build();

System.***out***.println("Laptop Config : " + laptop3);

}

}

1. Prototype

1 Used when object creation is expensive require too much time and resources

1. Prototype design pattern uses cloning of original object
2. Example of cloning

1 Object class clone method

2 Not Loading data again from database

**package** com.kkjavatutorials;

**public** **class** Student {

**private** Integer id;

**private** String name;

**public** Integer getId() {

**return** id;

}

**public** **void** setId(Integer id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

@Override

**public** String toString() {

**return** "Student [id=" + id + ", name=" + name + "]";

}

}

**package** com.kkjavatutorials;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** StudentDAO **implements** Cloneable {

**private** **static** List<Student> *list*;

**static** {

*list* = **new** ArrayList<>();

Student student1 = **new** Student();

student1.setId(10);

student1.setName("PK");

Student student2 = **new** Student();

student2.setId(20);

student2.setName("MK");

*list*.add(student1);

*list*.add(student2);

}

**public** List<Student> getAllStudent() {

**return** *list*;

}

// override clone method and implement Cloneable marker interface

@Override

**public** List<Student> clone() **throws** CloneNotSupportedException {

List<Student> dummyStudentList = **new** ArrayList<>();

**for** (Student student : *list*) {

dummyStudentList.add(student);

}

**return** dummyStudentList;

}

}

**package** com.kkjavatutorials;

**import** java.util.List;

**public** **class** Tester {

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

StudentDAO dao = **new** StudentDAO();

List<Student> allStudents = dao.getAllStudent();

// create a clone of exiting list coming from db

List<Student> updatedList = dao.clone();

Student student = **new** Student();

student.setId(30);

student.setName("SK");

updatedList.add(student);

System.***out***.println("Original List");

allStudents.forEach(System.***out***::println);

System.***out***.println("Updated List");

updatedList.forEach(System.***out***::println);

}

}

1. Structural
2. Adapter
3. Adapter Design Pattern is used so that two unrelated interfaces can work together
4. Communication between two incompatible interfaces
5. The object which joins these unrelated interface is called Adapter
6. Example of adapter design pattern
7. Mobile charger (240V(Wall Socket or 120V(US) 3V(mobile battery))
8. Laptop Charger (120V to LaptopBattery)
9. Memory Card (memory card into memory card reader then insert into laptop)
10. java.util.Arrays asList()
11. java.io.InputStreamReader(InputStream)
12. java.io.OutputStreamWriter(OutputStream)
13. javax.xml.bind.annotation.adapters.XmlAdapter(marshall and unmarshall)

**package** com.kkjavatutorials;

**public** **class** Volt {

**private** **int** volts;

**public** Volt(**int** volts) {

**super**();

**this**.volts = volts;

}

**public** **int** getVolts() {

**return** volts;

}

**public** **void** setVolts(**int** volts) {

**this**.volts = volts;

}

@Override

**public** String toString() {

**return** "Volt [volts=" + volts + "]";

};

}

**package** com.kkjavatutorials;

**public** **interface** IMobileAdapter {

**public** Volt get3Volt();

}

**package** com.kkjavatutorials;

**public** **interface** IWallSocket {

**public** **abstract** Volt getVolts();

}

**package** com.kkjavatutorials;

**public** **class** MobileAdapterImpl **implements** IMobileAdapter {

**private** IWallSocket wallSocket;

// Composition

**public** MobileAdapterImpl(IWallSocket wallSocket) {

**super**();

**this**.wallSocket = wallSocket;

}

@Override

**public** Volt get3Volt() {

Volt v240 = wallSocket.getVolts();

**int** v3 = v240.getVolts() / 80;

**return** **new** Volt(v3);

}

}

**package** com.kkjavatutorials;

**public** **class** WallSocketImpl **implements** IWallSocket {

@Override

**public** Volt getVolts() {

**return** **new** Volt(240);

}

}

**package** com.kkjavatutorials;

**public** **class** Tester {

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

IWallSocket wallSocket = **new** WallSocketImpl();

Volt v240 = wallSocket.getVolts();

System.***out***.println(v240);

IMobileAdapter mobileAdapter = **new** MobileAdapterImpl(wallSocket);

Volt v3 = mobileAdapter.get3Volt();

System.***out***.println(v3);

}

}

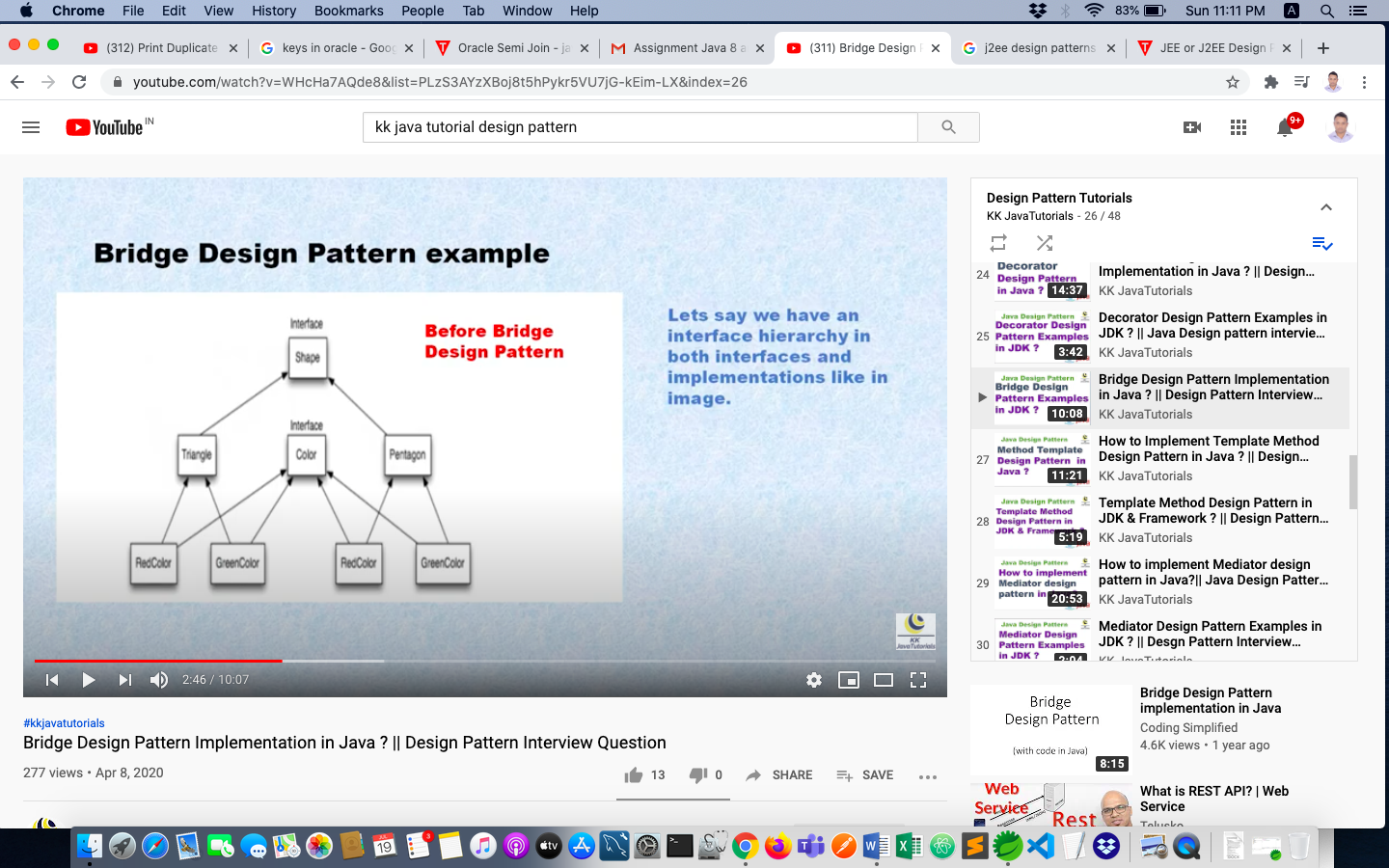
2 Bridge

1 When we have interface hierarchy in both interfaces as well as implementation then we can decouple the interfaces from implementation and hide implementation detail from client programs

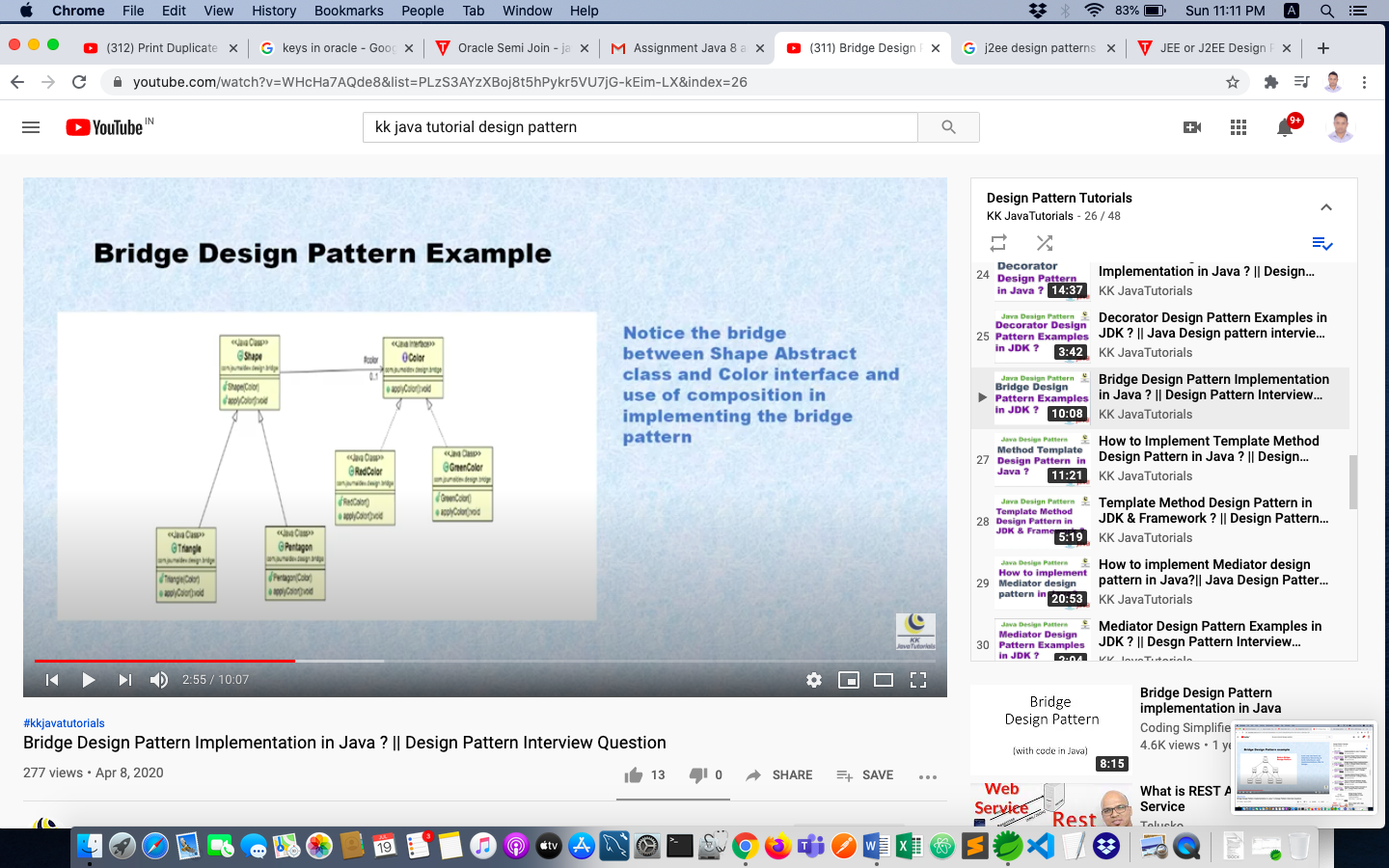
2 Decouple an abstraction and implementation

3 Composition over Inheritance

Before Bridge



After Bridge



**package** com.kkjavatutorials;

**public** **interface** Color {

**public** **abstract** String fill();

}

**package** com.kkjavatutorials;

**public** **class** Red **implements** Color {

@Override

**public** String fill() {

**return** "Color is Red";

}

}

**package** com.kkjavatutorials;

**public** **class** Blue **implements** Color {

@Override

**public** String fill() {

**return** "Color is Blue";

}

}

**package** com.kkjavatutorials;

// Shape establish bridge between Shape and Color

**public** **abstract** **class** Shape {

// 1. declare a protected reference of Color

**protected** Color color;

// 2. Initialize the reference

**public** Shape(Color color) {

**this**.color = color;

}

// 3. abstract method

**public** **abstract** String draw();

}

**package** com.kkjavatutorials;

**public** **class** Circle **extends** Shape {

**public** Circle(Color color) {

**super**(color);

}

@Override

**public** String draw() {

**return** "Circle drawn : " + color.fill();

}

}

**package** com.kkjavatutorials;

**public** **class** Square **extends** Shape {

**public** Square(Color color) {

**super**(color);

}

@Override

**public** String draw() {

**return** "Square drawn : " + color.fill();

}

}

**package** com.kkjavatutorials;

**public** **class** Tester {

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

Shape circle = **new** Circle(**new** Red());

String redCircle = circle.draw();

System.***out***.println(redCircle);

Shape square = **new** Square(**new** Blue());

String blueSquare = square.draw();

System.***out***.println(blueSquare);

}

}

1. Composite

1. Composite design pattern is used when we have to represent a part out of whole hierarchy

2. 4 parts

1. Component

2. Leaf (Leaf are independent)

3. Composite (implements Component Interface)

4. Client

3. Example of Composite

1. java.awt.Container (Swing)

2.java.faces.component.UIComponent (getChildren)

**package** com.kkjavatutorials;

// Component

**public** **interface** Service {

**public** **abstract** **void** service(String serviceType);

}

**package** com.kkjavatutorials;

// Leaf or children of Component

// EmployeeService is independent of AdminService

**public** **class** EmployeeService **implements** Service {

@Override

**public** **void** service(String serviceType) {

System.***out***.println(serviceType + " for Employee");

}

}

**package** com.kkjavatutorials;

// Leaf or children

// AdminService is independent of EmployeeService

**public** **class** AdminService **implements** Service {

@Override

**public** **void** service(String serviceType) {

System.***out***.println(serviceType + " for Admin");

}

}

**package** com.kkjavatutorials;

**import** java.util.ArrayList;

**import** java.util.List;

// Composite has to implement Component

**public** **class** ServiceProvider **implements** Service {

**private** List<Service> serviceList = **new** ArrayList<>();

@Override

**public** **void** service(String serviceType) {

**for** (Service service : serviceList) {

service.service(serviceType);

}

}

**public** **void** addService(Service service) {

serviceList.add(service);

}

**public** **void** deleteService(Service service) {

serviceList.remove(service);

}

**public** **void** clearServices() {

serviceList.clear();

}

}

**package** com.kkjavatutorials;

**public** **class** Tester {

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

Service employeeService1 = **new** EmployeeService();

Service employeeService2 = **new** EmployeeService();

Service adminService1 = **new** AdminService();

ServiceProvider serviceProvider = **new** ServiceProvider();

serviceProvider.addService(employeeService1);

serviceProvider.addService(employeeService2);

serviceProvider.addService(adminService1);

serviceProvider.service("Registration Service");

serviceProvider.deleteService(employeeService1);

Service adminService2 = **new** AdminService();

serviceProvider.addService(adminService2);

System.***out***.println("========================");

serviceProvider.service("Logout Service");

serviceProvider.clearServices();

System.***out***.println("========================");

serviceProvider.addService(employeeService1);

serviceProvider.service("Update Service");

System.***out***.println("========================");

}

}

1. Decorator
2. Decorator is used to modify the functionality of an object at runtime. At the same time other instances of the same class will not be affected by this, so individual object gets the modified behavior.
3. This pattern provides wrapper to existing class
4. Decorator design pattern uses composition
5. Decorator design pattern provides runtime modification abilities and hence more flexible, it is easy to maintain and extend when no. of choice is more.
6. Example of Decorator Design Pattern
7. In Java IO package (FileReader, BufferedReader, BufferedInputStream)
8. java.util.Collections (synchronizedMap() and unmodifiableList(), unmodifiableSet(), unmodifiableMap())
9. javax.servlet.http.HttpServletRequestWrapper and HttpServletResponseWrapper

**package** com.kkjavatutorials;

**public** **interface** Bike {

**public** **abstract** **void** assembleBike();

}

**package** com.kkjavatutorials;

**public** **class** BasicBike **implements** Bike {

@Override

**public** **void** assembleBike() {

System.***out***.println("Basic Bike...");

}

}

**package** com.kkjavatutorials;

**public** **class** DecoratorBike **implements** Bike {

**private** Bike bike;

**public** DecoratorBike(Bike bike) {

**super**();

**this**.bike = bike;

}

@Override

**public** **void** assembleBike() {

bike.assembleBike();

}

}

**package** com.kkjavatutorials;

**public** **class** SportsBike **extends** DecoratorBike {

**public** SportsBike(Bike bike) {

**super**(bike);

}

**public** **void** assembleBike() {

**super**.assembleBike();

System.***out***.println("Adding features of Sports Bike...");

};

}

**package** com.kkjavatutorials;

**public** **class** LuxuryBike **extends** DecoratorBike {

**public** LuxuryBike(Bike bike) {

**super**(bike);

}

**public** **void** assembleBike() {

**super**.assembleBike();

System.***out***.println("Adding features of Luxury Bike...");

};

}

**package** com.kkjavatutorials;

**public** **class** Tester {

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

Bike basicBike = **new** BasicBike();

basicBike.assembleBike();

System.***out***.println("=======================");

Bike sportsBike = **new** SportsBike(basicBike);

sportsBike.assembleBike();

System.***out***.println("=======================");

Bike luxuryBike = **new** LuxuryBike(basicBike);

luxuryBike.assembleBike();

System.***out***.println("=======================");

// adding features of both sprtes and luxury

Bike sportsLuxuryBike = **new** SportsBike(**new** LuxuryBike(**new** BasicBike()));

sportsLuxuryBike.assembleBike();

System.***out***.println("=======================");

}

}

5.Facade

1. Façade helps client application to easily interact with the system

2. Façade provides a unified interface to a set of interfaces in a system or subsystem.

3. Façade defines a higher-level interface that makes the subsystem easier to use

4. Façade do not hide subsystem interfaces from the client

5 Façade is used when number of interfaces grow and system gets complex

1. Subsystem interfaces are not aware of Façade and they should not have any reference of Façade interface
2. Façade is used for similar kind of interfaces
3. Façade purpose is to provide a single interface rather than multiple interfaces that does similar kind of job
4. Example of Façade design pattern

1 javax.faces.context.FacesContext

2 javax.faces.context.ExternalContext

**package** com.kkjavatutorials;

**import** java.sql.Connection;

**public** **interface** ExcelReport {

**public** **abstract** **void** generateExcelReport(Connection connection, String tableName);

}

**package** com.kkjavatutorials;

**import** java.sql.Connection;

**public** **class** ExcelReportImpl **implements** ExcelReport {

@Override

**public** **void** generateExcelReport(Connection connection, String tableName) {

System.***out***.println("EXCEL Report generation logic here...");

}

}

**package** com.kkjavatutorials;

**import** java.sql.Connection;

**public** **interface** HtmlReport {

**public** **abstract** **void** generateHtmlReport(Connection connection, String tableName);

}

**package** com.kkjavatutorials;

**import** java.sql.Connection;

**public** **class** HtmlReportImpl **implements** HtmlReport {

@Override

**public** **void** generateHtmlReport(Connection connection, String tableName) {

System.***out***.println("HTML Report generation logic here...");

}

}

**package** com.kkjavatutorials;

**import** java.sql.Connection;

**public** **interface** PdfReport {

**public** **abstract** **void** generatePdfReport(Connection connection, String tableName);

}

**package** com.kkjavatutorials;

**import** java.sql.Connection;

**public** **class** PdfReportImpl **implements** PdfReport {

@Override

**public** **void** generatePdfReport(Connection connection, String tableName) {

System.***out***.println("PDF Report generation logic here...");

}

}

**package** com.kkjavatutorials;

**import** java.sql.Connection;

**public** **class** ReportFacade {

// Code for interfaces than implementation

**private** ExcelReport excelReport;

**private** HtmlReport htmlReport;

**private** PdfReport pdfReport;

**public** ReportFacade() {

excelReport = **new** ExcelReportImpl();

htmlReport = **new** HtmlReportImpl();

pdfReport = **new** PdfReportImpl();

}

**public** **void** generateExcelReport(Connection connection, String tableName) {

excelReport.generateExcelReport(connection, tableName);

}

**public** **void** generateHtmlReport(Connection connection, String tableName) {

htmlReport.generateHtmlReport(connection, tableName);

}

**public** **void** generatePdfReport(Connection connection, String tableName) {

pdfReport.generatePdfReport(connection, tableName);

}

}

**package** com.kkjavatutorials;

**import** java.sql.Connection;

**public** **class** Tester {

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

Connection connection = **null**;

String tableName = "employee\_table";

ExcelReport excelReport = **new** ExcelReportImpl();

excelReport.generateExcelReport(connection, tableName);

HtmlReport htmlReport = **new** HtmlReportImpl();

htmlReport.generateHtmlReport(connection, tableName);

PdfReport pdfReport = **new** PdfReportImpl();

pdfReport.generatePdfReport(connection, tableName);

System.***out***.println("=====================");

// Usimg Facade Pattern to hide complexity of subsystem or system

ReportFacade reportFacade = **new** ReportFacade();

reportFacade.generateExcelReport(connection, tableName);

reportFacade.generateHtmlReport(connection, tableName);

reportFacade.generatePdfReport(connection, tableName);

System.***out***.println("=====================");

}

}

1. Flyweight

1

**package** com.kkjavatutorials;

**public** **abstract** **class** Shape {

**public** **void** draw(**int** radius, String fillColor, String lineColor) {

}

**public** **void** draw(**int** length, **int** breadth, String fillStyle) {

}

}

**package** com.kkjavatutorials;

**public** **class** Circle **extends** Shape {

**private** String label;

**public** Circle() {

label = "Circle";

}

**public** String getLabel() {

**return** label;

}

**public** **void** setLabel(String label) {

**this**.label = label;

}

**public** **void** draw(**int** radius, String fillColor, String lineColor) {

System.***out***.println("Drawing a " + label + " with radius " + radius + " fillColor " + fillColor + " lineColor " + lineColor);

}

}

**package** com.kkjavatutorials;

**public** **class** Rectangle **extends** Shape {

**private** String label;

**public** Rectangle() {

label = "Rectangle";

}

**public** String getLabel() {

**return** label;

}

**public** **void** setLabel(String label) {

**this**.label = label;

}

**public** **void** draw(**int** length, **int** breadth, String fillStyle) {

System.***out***.println(

"Drawing a " + label + " with length " + length + " breadth " + breadth + " fillStyle " + fillStyle);

}

}

**package** com.kkjavatutorials;

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** ShapeFactory {

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

**public** **static** Shape getShape(String type) {

Shape shape = **null**;

**if** (*shapes*.get(type) != **null**) {

shape = *shapes*.get(type);

} **else** {

**if** (type.equals("circle")) {

shape = **new** Circle();

} **else** **if** (type.equals("rectangle")) {

shape = **new** Rectangle();

}

}

*shapes*.put(type, shape);

**return** shape;

}

}

**package** com.kkjavatutorials;

**public** **class** PaintApp {

**public** **void** render(**int** noOfShapes) {

Shape shape = **null**;

**for**(**int** i=1;i<=noOfShapes;i++) {

**if**(i%2==0) {

shape = ShapeFactory.*getShape*("circle");

shape.draw(i, "red", "white");

}**else** {

shape = ShapeFactory.*getShape*("rectangle");

shape.draw(i, i+i, "dotted");

}

}

}

}

**package** com.kkjavatutorials;

**public** **class** Tester {

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

PaintApp paintApp = **new** PaintApp();

paintApp.render(10);

}

}

1. Proxy

1 Proxy means ‘in place of’ or ‘on behalf of’

2 Proxy allows us to create an intermediary that acts as an interface to another resource and also hiding the underlying complexity of the component

3 A heavy java object (JDBC connection or SessionFactory) that requires some initial configuration

4 We only want those objects to be initialized on demand and once they are we want to reuse them for all calls

5 When to use Proxy:

1. When we want a simplified version of a complex or heavy object
2. When the original object is present in different address space and we want to represent it locally
3. When we want to add a layer of security to the original underlying object to provide controlled access based on access rights of client (This is called Protection Proxy)

6 Example of Proxy

1 java.lang.reflect.Proxy

2 java.rmi

1. javax.ejb.EJB
2. javax.persistence.PersistenceContext

**package** com.kkjavatutorials;

**public** **interface** VeryExpensiveProcess {

// Time Consuming process

**public** **abstract** **void** process();

}

**package** com.kkjavatutorials;

//Original Impl

**public** **class** VeryExpensiveProcessImpl **implements** VeryExpensiveProcess {

**public** VeryExpensiveProcessImpl() {

heavyInitialConfigurationSetup();

}

@Override

**public** **void** process() {

System.***out***.println("Processing is done");

}

**public** **void** heavyInitialConfigurationSetup() {

System.***out***.println("Setting up initial configuration");

}

}

**package** com.kkjavatutorials;

//Proxy Impl

**public** **class** VeryExpensiveProcessProxy **implements** VeryExpensiveProcess {

**private** **static** VeryExpensiveProcess *veryExpensiveProcessObject*;

@Override

**public** **void** process() {

// Lazy Initialization

**if** (*veryExpensiveProcessObject* == **null**) {

*veryExpensiveProcessObject* = **new** VeryExpensiveProcessImpl();

}

*veryExpensiveProcessObject*.process();

}

}

**package** com.kkjavatutorials;

**public** **class** Tester {

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

// Only one time heavy object is created

VeryExpensiveProcess veryExpensiveProcess = **new** VeryExpensiveProcessProxy();

veryExpensiveProcess.process();

veryExpensiveProcess.process();

}

}

3 Behavioural

1 Chain of Responsibility

1. Chain of Responsibility is used to achieve loose coupling in software design where a request from client is passed to a chain of objects to process them. Then the chain of objects will decide themselves who will be processing the request and whether the request is required to be sent to the next object in the chain or not.
2. Example of Chain of responsibility

1 ATM machine (Amount 9800 : 2000\*4 + 500\*3 + 200\*1 + 100\*1)

2300 (Amount 2000\*1 + 200\*1 + 100**\***1)

2 Multiple catch block in Java (InstantiationException -> IllegalAccessException -> ClassNotFoundException)

3 java.util.logging.Logger (log())

4 javax.servlet.Filter (doFilter())

**package** com.kkjavatutorials;

**public** **class** Currency {

**private** **int** amount;

**public** Currency(**int** amt) {

**this**.amount = amt;

}

**public** **int** getAmount() {

**return** **this**.amount;

}

}

**package** com.kkjavatutorials;

**public** **interface** DispenseChain {

**void** setNextChain(DispenseChain nextChain);

**void** dispense(Currency cur);

}

**package** com.kkjavatutorials;

**public** **class** Rupees2000DispenseChain **implements** DispenseChain {

**private** DispenseChain chain;

@Override

**public** **void** setNextChain(DispenseChain nextChain) {

**this**.chain = nextChain;

}

@Override

**public** **void** dispense(Currency cur) {

**int** factor = 2000;

**int** amount = cur.getAmount();

**if** (amount >= factor) {

**int** num = amount / factor;

**int** remainder = amount % factor;

System.***out***.println("Dispensing " + num + " Rs " + factor + " note");

**if** (remainder != 0)

**this**.chain.dispense(**new** Currency(remainder));

} **else** {

**this**.chain.dispense(cur);

}

}

}

**package** com.kkjavatutorials;

**public** **class** Rupees500DispenseChain **implements** DispenseChain {

**private** DispenseChain chain;

@Override

**public** **void** setNextChain(DispenseChain nextChain) {

**this**.chain = nextChain;

}

@Override

**public** **void** dispense(Currency cur) {

**int** factor = 500;

**int** amount = cur.getAmount();

**if** (amount >= factor) {

**int** num = amount / factor;

**int** remainder = amount % factor;

System.***out***.println("Dispensing " + num + " Rs " + factor + " note");

**if** (remainder != 0)

**this**.chain.dispense(**new** Currency(remainder));

} **else** {

**this**.chain.dispense(cur);

}

}

}

**package** com.kkjavatutorials;

**public** **class** Rupees200DispenseChain **implements** DispenseChain {

**private** DispenseChain chain;

@Override

**public** **void** setNextChain(DispenseChain nextChain) {

**this**.chain = nextChain;

}

@Override

**public** **void** dispense(Currency cur) {

**int** factor = 200;

**int** amount = cur.getAmount();

**if** (amount >= factor) {

**int** num = amount / factor;

**int** remainder = amount % factor;

System.***out***.println("Dispensing " + num + " Rs " + factor + " note");

**if** (remainder != 0)

**this**.chain.dispense(**new** Currency(remainder));

} **else** {

**this**.chain.dispense(cur);

}

}

}

**package** com.kkjavatutorials;

**public** **class** Rupees100DispenseChain **implements** DispenseChain {

**private** DispenseChain chain;

@Override

**public** **void** setNextChain(DispenseChain nextChain) {

**this**.chain = nextChain;

}

@Override

**public** **void** dispense(Currency cur) {

**int** factor = 100;

**int** amount = cur.getAmount();

**if** (amount >= factor) {

**int** num = amount / factor;

**int** remainder = amount % factor;

System.***out***.println("Dispensing " + num + " Rs " + factor + " note");

**if** (remainder != 0)

**this**.chain.dispense(**new** Currency(remainder));

} **else** {

**this**.chain.dispense(cur);

}

}

}

**package** com.kkjavatutorials;

**public** **class** ATMMachineDispenseChain {

**private** DispenseChain moneyDispenseChain1;

**public** ATMMachineDispenseChain() {

moneyDispenseChain1 = **new** Rupees2000DispenseChain();

DispenseChain moneyDispenseChain2 = **new** Rupees500DispenseChain();

DispenseChain moneyDispenseChain3 = **new** Rupees200DispenseChain();

DispenseChain moneyDispenseChain4 = **new** Rupees100DispenseChain();

moneyDispenseChain1.setNextChain(moneyDispenseChain2);

moneyDispenseChain2.setNextChain(moneyDispenseChain3);

moneyDispenseChain3.setNextChain(moneyDispenseChain4);

}

**public** DispenseChain getMoneyDispenseChain1() {

**return** moneyDispenseChain1;

}

}

**package** com.kkjavatutorials;

**import** java.util.Scanner;

**public** **class** Tester {

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

ATMMachineDispenseChain atmMachineDispenseChain = **new** ATMMachineDispenseChain();

Scanner sc = **null**;

// int[] arr = { 100, 200, 300, 400, 500, 600, 1000, 1500, 2000, 10000, 15000,

// 21000, 23000, 24000 };

**int** amount = 0;

**while** (**true**) {

// for (int i = 0; i < arr.length; i++) {

// System.out.println();

// amount = arr[i];

**try** {

System.***out***.println("Enter amount to withdraw/dispense: ");

sc = **new** Scanner(System.***in***);

amount = sc.nextInt();

**if** (amount % 100 != 0) {

System.***out***.println("Amount must be in multiple of Rs 100");

**return**;

} **else** {

DispenseChain dispenseChain1 = atmMachineDispenseChain.getMoneyDispenseChain1();

Currency currency = **new** Currency(amount);

dispenseChain1.dispense(currency);

}

} **catch** (Exception e) {

e.printStackTrace();

}

// }

}

}

}

2 Command

1 Request object is wrapped under Command object and passed to invoker object.

2 Invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command.

3 Example of Command

1 All implementations of java.lang.Runnable

2 All implementations of javax.swing.Action

**package** com.kkjavatutorials;

// command interface

**public** **interface** Order {

**public** **abstract** **void** execute();

}

**package** com.kkjavatutorials;

**public** **class** BuyOrder **implements** Order {

**private** Stock stock;

**public** BuyOrder(Stock stock) {

**super**();

**this**.stock = stock;

}

@Override

**public** **void** execute() {

stock.buyStock();

}

}

**package** com.kkjavatutorials;

**public** **class** SellOrder **implements** Order {

**private** Stock stock;

**public** SellOrder(Stock stock) {

**super**();

**this**.stock = stock;

}

@Override

**public** **void** execute() {

stock.sellStock();

}

}

**package** com.kkjavatutorials;

**public** **class** Stock {

**private** String stockName = "Google Share";

**private** **int** stockQuantity = 20;

**public** **void** buyStock() {

System.***out***.println("Stock Name : " + stockName + " and Quantity : " + stockQuantity + " bought");

}

**public** **void** sellStock() {

System.***out***.println("Stock Name : " + stockName + " and Quantity : " + stockQuantity + " sold");

}

}

**package** com.kkjavatutorials;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** StockBroker {

**private** List<Order> orderList = **new** ArrayList<>();

**public** **void** placeOrder(Order order) {

orderList.add(order);

}

**public** **void** executeOrders() {

**for** (Order order : orderList) {

order.execute();

}

orderList.clear();

}

}

**package** com.kkjavatutorials;

**public** **class** Tester {

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

Stock stock = **new** Stock();

BuyOrder buyOrder = **new** BuyOrder(stock);

SellOrder sellOrder = **new** SellOrder(stock);

StockBroker stockBroker = **new** StockBroker();

stockBroker.placeOrder(buyOrder);

stockBroker.placeOrder(sellOrder);

stockBroker.executeOrders();

}

}

1. Interpreter

1 Interpreter design pattern is used to define a grammatical representation for a language and provide and interpreter to deal with this grammar.

2 Example of Interpreter:

1 Java Compiler (converts source code into byte code

which is understandable by JVM)

2 Google Translator (input language to output language)

3 java.util.Pattern

4 subclasses of java.text.Format like (DecimalFormat, SimpleDateFormat,

**package** com.kkjavatutorials;

**public** **interface** Expression {

**public** **abstract** **int** interpret(InterpreterEngine interpreterEngine);

}

**package** com.kkjavatutorials;

**public** **class** AdditionExpression **implements** Expression {

**private** String expression;

**public** AdditionExpression(String expression) {

**this**.expression = expression;

}

@Override

**public** **int** interpret(InterpreterEngine interpreterEngine) {

**return** interpreterEngine.addition(expression);

}

}

**package** com.kkjavatutorials;

**public** **class** SubtractionExpression **implements** Expression {

**private** String expression;

**public** SubtractionExpression(String expression) {

**this**.expression = expression;

}

@Override

**public** **int** interpret(InterpreterEngine interpreterEngine) {

**return** interpreterEngine.subtraction(expression);

}

}

**package** com.kkjavatutorials;

**public** **class** InterpreterEngine {

**public** **int** addition(String inputData) {

String[] tokens = interpret(inputData);

**int** n1 = Integer.*parseInt*(tokens[0]);

**int** n2 = Integer.*parseInt*(tokens[1]);

**return** n1 + n2;

}

**public** **int** subtraction(String inputData) {

String[] tokens = interpret(inputData);

**int** n1 = Integer.*parseInt*(tokens[0]);

**int** n2 = Integer.*parseInt*(tokens[1]);

**return** n1 - n2;

}

**public** String[] interpret(String inputData) {

String tempInputData = inputData.replaceAll("[^0-9]", " ");

tempInputData = tempInputData.replaceAll("( )+", " ").trim();

String[] inputDatatokens = tempInputData.split(" ");

**return** inputDatatokens;

}

}

**package** com.kkjavatutorials;

**public** **class** InterpreterClient {

**private** InterpreterEngine interpreterEngine;

**public** InterpreterClient(InterpreterEngine interpreterEngine) {

**this**.interpreterEngine = interpreterEngine;

}

**public** **int** interpret(String inputData) {

Expression expression = **null**;

**if** (inputData.contains("add")) {

expression = **new** AdditionExpression(inputData);

} **else** **if** (inputData.contains("subtract")) {

expression = **new** SubtractionExpression(inputData);

} **else** {

**throw** **new** RuntimeException(inputData + " is not valid expression");

}

**int** result = expression.interpret(interpreterEngine);

System.***out***.println(inputData);

**return** result;

}

}

**package** com.kkjavatutorials;

**public** **class** Tester {

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

InterpreterEngine interpreterEngine = **new** InterpreterEngine();

InterpreterClient interpreterClient = **new** InterpreterClient(interpreterEngine);

System.***out***.println("Addition of both numbers is : " + interpreterClient.interpret("add 200 and 75"));

System.***out***.println("=============================================");

System.***out***.println("Subtraction of numbers is : " + interpreterClient.interpret("subtract 30 and 100"));

}

}

1. Iterator
2. Iterator Pattern provides a standard way to traverse through a group of objects collection and hide the implementation logic from client program
3. Iterator pattern is used in Java Collection
4. Iterator interface provides methods for traversing through a collection
5. Access elements of an aggregate object without exposing its underlying representation
6. Iterator design pattern hides actual implementation of traversal through the collection and client program just use iterator methods
7. Example of Iterator :
8. All implementation of java.util.Iterator (ArrayList, LinkedList etc)
9. All implementation of java.util.Enumeration (vector,dictionary)

**package** com.kkjavatutorials;

**public** **interface** Iterator {

**public** **abstract** **boolean** hasNext();

**public** **abstract** Object next();

}

**package** com.kkjavatutorials;

**public** **interface** Collection {

**public** **abstract** Iterator getIterator();

}

**package** com.kkjavatutorials;

**public** **class** CollectionImpl **implements** Collection {

String arr[] = { "Pushkar", "KK", "PK", "Mark", "Sean" };

@Override

**public** Iterator getIterator() {

**return** **new** IteratorImpl();

}

**private** **class** IteratorImpl **implements** Iterator {

**int** index = 0;

@Override

**public** **boolean** hasNext() {

**if** (index < arr.length)

**return** **true**;

**return** **false**;

}

@Override

**public** Object next() {

**if** (**this**.hasNext())

**return** arr[index++];

**return** **null**;

}

}

}

**package** com.kkjavatutorials;

**public** **class** Tester {

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

CollectionImpl collectionImpl = **new** CollectionImpl();

Iterator iterator = collectionImpl.getIterator();

**while** (iterator.hasNext()) {

Object object = iterator.next();

String name = (String) object;

System.***out***.println("Name : " + name);

}

}

}

1. Mediator
2. Mediator is useful when communication logic between objects is complex, we can have a central point of communication that takes care of communication logic
3. Mediator is used to provide centralized communication medium between different objects in a system.
4. If objects are interacting directly with each other, the system components are tightly-coupled with each other that makes higher maintainability cost.
5. Mediator provides lose coupling between objects and acts as a mediator between objects for communication
6. An interface/abstract class that provides contract for communication and then we have concrete implementation of mediator
7. Example
8. Java.util.Timer class schedule() methods
9. Java Concurrency Executor execute() method
10. Java.lang.reflect.Method invoke() method

**package** com.kkjavatutorials;

**public** **interface** ChattingMediator {

**public** **abstract** **void** sendMessage(String message, User user);

**public** **abstract** **void** addUser(User user);

}

**package** com.kkjavatutorials;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** ChattingMediatorImpl **implements** ChattingMediator {

**private** List<User> userList;

**public** ChattingMediatorImpl() {

userList = **new** ArrayList<User>();

}

@Override

**public** **void** sendMessage(String message, User user) {

**for** (User u : userList) {

**if** (!u.getUsername().equalsIgnoreCase(user.getUsername())) {

u.receiveMessage(message);

}

}

}

@Override

**public** **void** addUser(User user) {

userList.add(user);

}

}

**package** com.kkjavatutorials;

**public** **abstract** **class** User {

**private** ChattingMediator chattingMediator;

**private** String username;

**public** User() {

}

**public** User(ChattingMediator chattingMediator, String username) {

**super**();

**this**.chattingMediator = chattingMediator;

**this**.username = username;

}

**public** String getUsername() {

**return** username;

}

**public** **void** setUsername(String username) {

**this**.username = username;

}

**public** **abstract** **void** sendMessage(String message);

**public** **abstract** **void** receiveMessage(String message);

}

**package** com.kkjavatutorials;

**public** **class** UserImpl **extends** User {

**private** ChattingMediator chattingMediator;

**private** String username;

**public** UserImpl(ChattingMediator chattingMediator, String username) {

**super**(chattingMediator, username);

**this**.chattingMediator = chattingMediator;

**this**.username = username;

}

@Override

**public** **void** sendMessage(String message) {

System.***out***.println(username + " sending message: " + message);

chattingMediator.sendMessage(message, **this**);

}

@Override

**public** **void** receiveMessage(String message) {

System.***out***.println(username + " received message: " + message);

}

}

**package** com.kkjavatutorials;

**public** **class** Tester {

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

ChattingMediator chattingMediator = **new** ChattingMediatorImpl();

User user1 = **new** UserImpl(chattingMediator, "user1");

User user2 = **new** UserImpl(chattingMediator, "user2");

User user3 = **new** UserImpl(chattingMediator, "user3");

User user4 = **new** UserImpl(chattingMediator, "user4");

User user5 = **new** UserImpl(chattingMediator, "user5");

chattingMediator.addUser(user1);

chattingMediator.addUser(user2);

chattingMediator.addUser(user3);

chattingMediator.addUser(user4);

chattingMediator.addUser(user5);

user2.sendMessage("Hi, All");

}

}

6 Memento

Example 1:

**package** com.kkjavatutorials;

**public** **class** FileWriterUtil {

**private** String fileName;

**private** StringBuilder content;

**public** FileWriterUtil(String file) {

**this**.fileName = file;

**this**.content = **new** StringBuilder();

}

@Override

**public** String toString() {

**return** **this**.content.toString();

}

**public** **void** write(String str) {

content.append(str);

}

**public** Memento save() {

**return** **new** Memento(**this**.fileName, **this**.content);

}

**public** **void** undoToLastSave(Object obj) {

Memento memento = (Memento) obj;

**this**.fileName = memento.fileName;

**this**.content = memento.content;

}

**private** **class** Memento {

**private** String fileName;

**private** StringBuilder content;

**public** Memento(String file, StringBuilder content) {

**this**.fileName = file;

// notice the deep copy so that Memento and FileWriterUtil content variables

// don't refer to same object

**this**.content = **new** StringBuilder(content);

}

}

}

**package** com.kkjavatutorials;

**public** **class** FileWriterCaretaker {

**private** Object obj;

**public** **void** save(FileWriterUtil fileWriter) {

**this**.obj = fileWriter.save();

}

**public** **void** undo(FileWriterUtil fileWriter) {

fileWriter.undoToLastSave(obj);

}

}

**package** com.kkjavatutorials;

**public** **class** Tester {

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

FileWriterCaretaker caretaker = **new** FileWriterCaretaker();

FileWriterUtil fileWriter = **new** FileWriterUtil("momento\_design\_pattern.txt");

fileWriter.write("First Set of Data\n");

System.***out***.println(fileWriter + "\n\n");

// lets save the file

caretaker.save(fileWriter);

// now write something else

fileWriter.write("Second Set of Data\n");

// checking file contents

System.***out***.println(fileWriter + "\n\n");

// lets undo to last save

caretaker.undo(fileWriter);

// checking file content again

System.***out***.println(fileWriter + "\n\n");

}

}

Example 2:

**package** com.kkjavatutorials;

**public** **class** Memento {

**private** String state;

**public** Memento(String state) {

**super**();

**this**.state = state;

}

**public** String getState() {

**return** state;

}

**public** **void** setState(String state) {

**this**.state = state;

}

}

**package** com.kkjavatutorials;

**public** **class** Originator {

**private** String state;

**public** String getState() {

**return** state;

}

**public** **void** setState(String state) {

**this**.state = state;

}

**public** Memento saveStateToMemento() {

**return** **new** Memento(state);

}

**public** **void** getStateFromMemento(Memento memento) {

state = memento.getState();

}

}

**package** com.kkjavatutorials;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** CareTaker {

**private** List<Memento> mementoList = **new** ArrayList<>();

**public** **void** add(Memento state) {

mementoList.add(state);

}

**public** Memento get(**int** index) {

**return** mementoList.get(index);

}

}

**package** com.kkjavatutorials;

**public** **class** MementoPatternClientTester {

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

Originator originator = **new** Originator();

CareTaker careTaker = **new** CareTaker();

originator.setState("State #1");

originator.setState("State #2"); // State 2 overrides State 1 hence careTaker.get(0) will give State 2

careTaker.add(originator.saveStateToMemento());

originator.setState("State #3");

careTaker.add(originator.saveStateToMemento());

originator.setState("State #4");

System.***out***.println("Current State: " + originator.getState());

originator.getStateFromMemento(careTaker.get(0));

System.***out***.println("First Saved State: " + originator.getState());

originator.getStateFromMemento(careTaker.get(1));

System.***out***.println("Second Saved State: " + originator.getState());

}

}

7 Observer

1 Example 1:

1. Observer design pattern is useful when we are interested in the state of an object and want to get notified whenever there is a change
2. Observer: Object which watches the state of another object

Subject: Object which is being watched

1. In Observer Design Pattern, we define one to many relationship between objects and whenever the state of object changes all its subscribers/observers get notified
2. Example of Observer Design Pattern:
3. JMS
4. java.util.Observer and java.util.Observable
5. javax.servlet.http.HttpSessionBindingListener
6. javax.servlet.http.HttpSessionAttributeListener

config.txt

1 www.google.com 90

2 www.facebook.com 80

**package** com.kkjavatutorials;

**public** **class** URLConfig {

**public** **static** **int** *STATUS\_ADDED* = 1;

**public** **static** **int** *STATUS\_UPDATED* = 2;

**public** **static** **int** *STATUS\_DELETED* = 3;

**private** String rowId;

**private** String url;

**private** String freq;

**private** **int** status = 0;

**public** **int** getStatus() {

**return** status;

}

**public** **void** setStatus(**int** status) {

**this**.status = status;

}

**public** String getRowId() {

**return** rowId;

}

**public** **void** setRowId(String rowId) {

**this**.rowId = rowId;

}

**public** String getUrl() {

**return** url;

}

**public** **void** setUrl(String url) {

**this**.url = url;

}

**public** String getFreq() {

**return** freq;

}

**public** **void** setFreq(String freq) {

**this**.freq = freq;

}

@Override

**public** **int** hashCode() {

**final** **int** prime = 31;

**int** result = 1;

result = prime \* result + ((freq == **null**) ? 0 : freq.hashCode());

result = prime \* result + ((rowId == **null**) ? 0 : rowId.hashCode());

result = prime \* result + ((url == **null**) ? 0 : url.hashCode());

**return** result;

}

@Override

**public** **boolean** equals(Object obj) {

**if** (**this** == obj)

**return** **true**;

**if** (obj == **null**)

**return** **false**;

**if** (getClass() != obj.getClass())

**return** **false**;

URLConfig other = (URLConfig) obj;

**if** (freq == **null**) {

**if** (other.freq != **null**)

**return** **false**;

} **else** **if** (!freq.equals(other.freq))

**return** **false**;

**if** (rowId == **null**) {

**if** (other.rowId != **null**)

**return** **false**;

} **else** **if** (!rowId.equals(other.rowId))

**return** **false**;

**if** (url == **null**) {

**if** (other.url != **null**)

**return** **false**;

} **else** **if** (!url.equals(other.url))

**return** **false**;

**return** **true**;

}

@Override

**public** String toString() {

**return** "URLConfig [rowId=" + rowId + ", url=" + url + ", freq=" + freq + "]";

}

}

**package** com.kkjavatutorials;

**public** **abstract** **class** ObserverRead {

**public** **abstract** **void** update();

}

**package** com.kkjavatutorials;

**import** java.io.File;

**import** java.io.FileNotFoundException;

**import** java.util.ArrayList;

**import** java.util.HashMap;

**import** java.util.List;

**import** java.util.Map;

**import** java.util.Map.Entry;

**import** java.util.Scanner;

**public** **class** SubjectRead {

**private** List<ObserverRead> observer = **new** ArrayList<ObserverRead>();

**public** **void** attach(ObserverRead obs) {

observer.add(obs);

}

**private** Map<String, URLConfig> cache = **new** HashMap<String, URLConfig>();

**public** Map<String, URLConfig> getCache() {

**return** cache;

}

**public** **void** setCache(Map<String, URLConfig> cache) {

**this**.cache.putAll(cache);

}

**public** Map<String, URLConfig> readfile() {

Map<String, URLConfig> map = **new** HashMap<String, URLConfig>();

**try** {

File f = **new** File("config.txt");

Scanner sc = **new** Scanner(f);

**while** (sc.hasNextLine()) {

URLConfig urlConfig = **new** URLConfig();

String line = sc.nextLine();

String[] details = line.split(" ");

String rowno = details[0];

String url = details[1];

String freq = details[2];

urlConfig.setRowId(rowno);

urlConfig.setUrl(url);

urlConfig.setFreq(freq);

map.put(url, urlConfig);

// cache.putAll(map);

}

} **catch** (IndexOutOfBoundsException e) {

e.printStackTrace();

} **catch** (FileNotFoundException e) {

e.printStackTrace();

}

**return** map;

}

**public** **void** filereader() **throws** InterruptedException {

**while** (**true**) {

Map<String, URLConfig> map = readfile();

// is cache has any data

**if** (cache.size() == 0) {

**for** (Entry<String, URLConfig> entry : map.entrySet()) {

entry.getValue().setStatus(URLConfig.*STATUS\_ADDED*);

}

setCache(map);

} **else** {

**for** (Entry<String, URLConfig> entry : map.entrySet()) {

**if** (cache.containsKey(entry.getKey())) {

URLConfig configEntry = cache.get(entry.getKey());

**if** (!configEntry.equals(entry.getValue())) {

// find newly updated rows

entry.getValue().setStatus(URLConfig.*STATUS\_UPDATED*);

}

} **else** {

// find newly added rows

entry.getValue().setStatus(URLConfig.*STATUS\_ADDED*);

}

cache.put(entry.getKey(), entry.getValue());

}

}

// for(Entry<String, List<String>> e=map.entrySet()){

/\*

\* if (map.equals(cache)) { System.out.println("file not updated till now"); }

\* else { System.out.println("file updated"); notifyAllObserver(); }

\*

\*/

notifyAllObserver();

Thread.*sleep*(5000);

}

}

// cache.putAll(map);

/\*

\* for(java.util.Map.Entry<String, List<String>> e: map.entrySet()) {

\* if(cache.equals(e.getKey())) { System.out.println("hiiii"); } else{

\* System.out.println(" byeeee"); notifyAllObserver(); }

\*/

**public** **void** notifyAllObserver() {

**for** (ObserverRead obs : observer) {

obs.update();

}

}

}

**package** com.kkjavatutorials;

**import** java.util.Date;

**import** java.util.Map;

**import** java.util.Map.Entry;

**public** **class** MyObserver **extends** ObserverRead {

SubjectRead sub;

**public** MyObserver(SubjectRead sub) {

**super**();

**this**.sub = sub;

**this**.sub.attach(**this**);

}

@Override

**public** **void** update() {

Map<String, URLConfig> map= sub.getCache();

System.***out***.println(**new** Date(System.*currentTimeMillis*()));

**for**(Entry<String, URLConfig> e: map.entrySet())

{

**if**(e.getValue().getStatus() > 0)

System.***out***.println("Key:"+e.getKey()+" "+"value:"+e.getValue().toString());

}

}

}

**package** com.kkjavatutorials;

**public** **class** Tester {

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

SubjectRead sub = **new** SubjectRead();

**new** MyObserver(sub);

// sub.readfile();

sub.filereader();

}

}

Example 2:

**package** eg00231;

// immutable class

**public** **final** **class** Message {

**private** **final** String message;

**public** Message(String message) {

**super**();

**this**.message = message;

}

**public** String getMessage() {

**return** message;

}

}

**package** eg00231;

**public** **interface** Subject {

**public** **void** register(Observer observer);

**public** **void** unregister(Observer observer);

**public** **void** notifyUpdate(Message message);

}

**package** eg00231;

**import** java.util.HashSet;

**import** java.util.Set;

**public** **class** SubjectImplMessagePublisher **implements** Subject {

**private** Set<Observer> observers = **new** HashSet<>();

@Override

**public** **void** register(Observer observer) {

observers.add(observer);

}

@Override

**public** **void** unregister(Observer observer) {

observers.remove(observer);

}

@Override

**public** **void** notifyUpdate(Message message) {

**for** (Observer observer : observers) {

observer.updateObserver(message);

}

}

}

**package** eg00231;

// Observer is also called Subscriber

**public** **interface** Observer {

**public** **void** updateObserver(Message message);

}

**package** eg00231;

// this is observer 1

**public** **class** ObserverImplFirstMessageSubscriber **implements** Observer {

@Override

**public** **void** updateObserver(Message message) {

System.***out***.println("Message for first subscriber : " + message.getMessage());

}

}

**package** eg00231;

//this is observer 2

**public** **class** ObserverImplSecondMessageSubscriber **implements** Observer {

@Override

**public** **void** updateObserver(Message message) {

System.***out***.println("Message for second subscriber : " + message.getMessage());

}

}

**package** eg00231;

// this is observer 3

**public** **class** ObserverImplThirdMessageSubscriber **implements** Observer {

@Override

**public** **void** updateObserver(Message message) {

System.***out***.println("Message for third subscriber : " + message.getMessage());

}

}

**package** eg00231;

**public** **class** Tester {

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

ObserverImplFirstMessageSubscriber observer1 = **new** ObserverImplFirstMessageSubscriber();

ObserverImplSecondMessageSubscriber observer2 = **new** ObserverImplSecondMessageSubscriber();

ObserverImplThirdMessageSubscriber observer3 = **new** ObserverImplThirdMessageSubscriber();

SubjectImplMessagePublisher subject = **new** SubjectImplMessagePublisher();

subject.register(observer1);

subject.register(observer2);

// observer1 and observer2 will receive update

Message message1 = **new** Message("This is First Message");

subject.notifyUpdate(message1);

System.out.println("=============================");

subject.unregister(observer1);

subject.register(observer3);

// observer2 and observer3 will receive update

Message message2 = **new** Message("This is Second Message");

subject.notifyUpdate(message2);

System.out.println("=============================");

subject.unregister(observer2);

// observer3 will receive update

Message message3 = **new** Message("This is Third Message");

subject.notifyUpdate(message3);

System.out.println("=============================");

}

}

1. State
2. State Design Pattern is perfect example of polymorphic behaviour
3. State Design Pattern is used based on internal state of object we are going to perform some action.
4. State Design pattern is used when an object change its behavior based on its internal state.
5. State Design Pattern implements Polymorphic Behaviour
6. The chance of error is less
7. It is very easy to add more states for additional behavior.
8. Makes code robust, easily maintainable and flexible

Problem in below code:

Client should know specific state value

When no. of state increase it will be difficult to implement

**package** eg00232;

**public** **class** ACRemoteTester {

**private** String state;

**public** **void** setState(String state) {

**this**.state = state;

}

**public** **void** doAction() {

**if**("ON".equalsIgnoreCase(state)) {

System.***out***.println("AC is turned ON");

}**else** **if**("OFF".equalsIgnoreCase(state)) {

System.***out***.println("AC is turned OFF");

}

}

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

ACRemoteTester obj = **new** ACRemoteTester();

obj.setState("ON");

obj.doAction();

obj.setState("OFF");

obj.doAction();

}

}

Solution to above problem:

**package** eg00232;

**public** **interface** State {

**public** **abstract** **void** doAction();

}

**package** eg00232;

**public** **class** ACContext **implements** State {

**private** State state;

**public** **void** setState(State state) {

**this**.state = state;

}

**public** State getState() {

**return** state;

}

@Override

**public** **void** doAction() {

state.doAction();

}

}

**package** eg00232;

**public** **class** ACStartState **implements** State {

@Override

**public** **void** doAction() {

System.out.println("AC is turned ON");

}

}

**package** eg00232;

**public** **class** ACStopState **implements** State {

@Override

**public** **void** doAction() {

System.***out***.println("AC is turned OFF");

}

}

**package** eg00232;

**public** **class** ACRemoteTesterSolution {

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

ACContext acContext = **new** ACContext();

State ACStartState = **new** ACStartState();

acContext.setState(ACStartState);

acContext.doAction();

State ACStopState = **new** ACStopState();

acContext.setState(ACStopState);

acContext.doAction();

}

}

1. Strategy
2. Example of Strategy Pattern:
3. Collections.sort()
4. javax.servlet.http.HttpServlet (doGet(), doPost(), service())
5. Strategy design pattern is used when we have multiple algorithm for a specific task and client decides the actual implementation to be used at runtime.

**package** eg00233;

**public** **class** Product {

**private** String name;

**private** String productCode;

**private** **int** price;

**public** Product() {

}

**public** Product(String name, String productCode, **int** price) {

**super**();

**this**.name = name;

**this**.productCode = productCode;

**this**.price = price;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getProductCode() {

**return** productCode;

}

**public** **void** setProductCode(String productCode) {

**this**.productCode = productCode;

}

**public** **int** getPrice() {

**return** price;

}

**public** **void** setPrice(**int** price) {

**this**.price = price;

}

}

**package** eg00233;

**public** **interface** PaymentMethodStrategy {

**public** **abstract** **void** pay(**int** amount);

}

**package** eg00233;

**public** **class** CreditCardPayment **implements** PaymentMethodStrategy {

**private** String cardHolderName;

**private** String card;

**private** String cvv;

**private** String dateOfExpiry;

**public** CreditCardPayment() {

}

**public** CreditCardPayment(String cardHolderName, String card, String cvv, String dateOfExpiry) {

**super**();

**this**.cardHolderName = cardHolderName;

**this**.card = card;

**this**.cvv = cvv;

**this**.dateOfExpiry = dateOfExpiry;

}

@Override

**public** **void** pay(**int** amount) {

System.***out***.println(amount + " is paid using credit card : " + card);

}

**public** String getCardHolderName() {

**return** cardHolderName;

}

**public** **void** setCardHolderName(String cardHolderName) {

**this**.cardHolderName = cardHolderName;

}

**public** String getCard() {

**return** card;

}

**public** **void** setCard(String card) {

**this**.card = card;

}

**public** String getCvv() {

**return** cvv;

}

**public** **void** setCvv(String cvv) {

**this**.cvv = cvv;

}

**public** String getDateOfExpiry() {

**return** dateOfExpiry;

}

**public** **void** setDateOfExpiry(String dateOfExpiry) {

**this**.dateOfExpiry = dateOfExpiry;

}

}

**package** eg00233;

**public** **class** PaypalPayment **implements** PaymentMethodStrategy {

**private** String email;

**private** String password;

**public** PaypalPayment() {

}

**public** PaypalPayment(String email, String password) {

**super**();

**this**.email = email;

**this**.password = password;

}

@Override

**public** **void** pay(**int** amount) {

System.***out***.println(amount + " is paid using paypal");

}

**public** String getEmail() {

**return** email;

}

**public** **void** setEmail(String email) {

**this**.email = email;

}

**public** String getPassword() {

**return** password;

}

**public** **void** setPassword(String password) {

**this**.password = password;

}

}

**package** eg00233;

**public** **class** PaytmPayment **implements** PaymentMethodStrategy {

**private** String mobile;

**public** PaytmPayment() {

}

**public** PaytmPayment(String mobile) {

**super**();

**this**.mobile = mobile;

}

@Override

**public** **void** pay(**int** amount) {

System.***out***.println(amount + " is paid using paytm : " + mobile);

}

**public** String getMobile() {

**return** mobile;

}

**public** **void** setMobile(String mobile) {

**this**.mobile = mobile;

}

}

**package** eg00233;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** ShoppingCart {

**private** List<Product> productList;

**public** ShoppingCart() {

productList = **new** ArrayList<>();

}

**public** **void** addProduct(Product product) {

productList.add(product);

}

**public** **void** removeProduct(Product product) {

productList.remove(product);

}

**private** **int** calculateTotalProductPrice() {

// Old way

/\*

int totalPrice = 0;

for(Product product : productList) {

totalPrice += product.getPrice();

}

return totalPrice;

\*/

// Using Java 8

**return** productList.stream().map(p->p.getPrice()).reduce(0, Integer::*sum*);

}

**public** **void** payment(PaymentMethodStrategy paymentMethodStrategy) {

paymentMethodStrategy.pay(calculateTotalProductPrice());

}

}

**package** eg00233;

**public** **class** Tester {

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

ShoppingCart shoppingCart1 = **new** ShoppingCart();

Product product1 = **new** Product("Soap", "11111", 100);

Product product2 = **new** Product("Shampoo", "22222", 200);

Product product3 = **new** Product("Biscuit", "33333", 50);

Product product4 = **new** Product("Chips", "44444", 10);

Product product5 = **new** Product("Brush", "55555", 20);

shoppingCart1.addProduct(product1);

shoppingCart1.addProduct(product2);

shoppingCart1.addProduct(product3);

shoppingCart1.addProduct(product4);

shoppingCart1.addProduct(product5);

CreditCardPayment creditCardPayment = **new** CreditCardPayment("Pushkar", "1234 5678 9101 1121", "678",

"10/12/2024");

shoppingCart1.payment(creditCardPayment);

System.***out***.println("================================");

ShoppingCart shoppingCart2 = **new** ShoppingCart();

Product product6 = **new** Product("Rice", "66666", 1000);

Product product7 = **new** Product("Wheat", "77777", 500);

Product product8 = **new** Product("Milk", "88888", 20);

Product product9 = **new** Product("Bread", "99999", 30);

Product product10 = **new** Product("Earphone", "11112", 2000);

shoppingCart2.addProduct(product6);

shoppingCart2.addProduct(product7);

shoppingCart2.addProduct(product8);

shoppingCart2.addProduct(product9);

shoppingCart2.addProduct(product10);

PaypalPayment paypalPayment = **new** PaypalPayment("Pushkar", "98263 96465");

shoppingCart2.payment(paypalPayment);

System.***out***.println("================================");

ShoppingCart shoppingCart3 = **new** ShoppingCart();

Product product11 = **new** Product("Eggs", "22224", 200);

Product product12 = **new** Product("Chicken", "22225", 400);

Product product13 = **new** Product("Fish", "22226", 500);

Product product14 = **new** Product("Dry Fruits", "22227", 1000);

Product product15 = **new** Product("Ketchup", "22228", 200);

shoppingCart3.addProduct(product11);

shoppingCart3.addProduct(product12);

shoppingCart3.addProduct(product13);

shoppingCart3.addProduct(product14);

shoppingCart3.addProduct(product15);

PaytmPayment paytmPayment = **new** PaytmPayment("9826396465");

shoppingCart3.payment(paytmPayment);

System.***out***.println("================================");

}

}

1. Template Method
2. Template Method is used to create a method stud and that differs some of the steps of implementation to the subclasses
3. An abstract template class contains a final method which provides and default implementation of the order of execution which is common of subclasses.
4. Order of execution is fixed.
5. Example of Template Method
6. All non abstract methods of java.io.InputStream , java.io.OutputStream, java.io.Reader, java.io.writer
7. All non abstract methods of java.util.AbstractList, java.util.AbstractSet, java.util.AbstractMap
8. JdbcTemplate, HibernateTemplate, JmsTemplate, RestTemplate, SimpleJdbcTemplate, TransactionTemplate

**package** eg00234;

**public** **abstract** **class** PizzaTemplate {

**public** **final** **void** preparePizza() {

selectBread();

addIngredients();

cookPizza();

addCheese();

addToppings();

}

**public** **abstract** **void** selectBread();

**public** **abstract** **void** addIngredients();

**public** **void** cookPizza() {

System.***out***.println("Cooking pizza for 30 minutes");

}

**public** **void** addCheese() {

System.***out***.println("Adding Cheese in Pizza");

}

**public** **void** addToppings() {

System.***out***.println("Adding Toppings in Pizza");

}

}

**package** eg00234;

**public** **class** VegPizza **extends** PizzaTemplate {

@Override

**public** **void** selectBread() {

System.***out***.println("Choossing bread for Veg Pizza");

}

@Override

**public** **void** addIngredients() {

System.***out***.println("Adding ingredients for Veg Pizza");

}

}

**package** eg00234;

**public** **class** NonVegPizza **extends** PizzaTemplate {

@Override

**public** **void** selectBread() {

System.***out***.println("Choossing bread for Non-Veg Pizza");

}

@Override

**public** **void** addIngredients() {

System.***out***.println("Adding ingredients for Non-Veg Pizza");

}

}

**package** eg00234;

**public** **class** Tester {

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

PizzaTemplate vegPizzaTemplate = **new** VegPizza();

vegPizzaTemplate.preparePizza();

System.***out***.println("=====================");

PizzaTemplate nonvegPizzaTemplate = **new** NonVegPizza();

nonvegPizzaTemplate.preparePizza();

System.***out***.println("=====================");

}

}

11 Visitor

1. Visitor pattern is used when we have to perform an operation on a similar kind of objects.
2. Using Visitor pattern, we move operational logic from objects to another class.
3. Benefit of Visitor Pattern, if the logic of operation changes, then we need to make changes only in the visitor implementation rather than doing it in all the classes

**package** eg00235;

**public** **interface** Item {

**public** **double** accept(ShoppingCartVisitor visitor);

}

**package** eg00235;

**public** **interface** ShoppingCartVisitor {

**public** **double** visit(Book book);

**public** **double** visit(Fruit fruit);

}

**package** eg00235;

**public** **class** Book **implements** Item {

**private** String name;

**private** String isbnNumber;

**private** **double** price;

**public** Book() {

}

**public** Book(String name, String isbnNumber, **double** price) {

**super**();

**this**.name = name;

**this**.isbnNumber = isbnNumber;

**this**.price = price;

}

@Override

**public** **double** accept(ShoppingCartVisitor visitor) {

**return** visitor.visit(**this**);

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getIsbnNumber() {

**return** isbnNumber;

}

**public** **void** setIsbnNumber(String isbnNumber) {

**this**.isbnNumber = isbnNumber;

}

**public** **double** getPrice() {

**return** price;

}

**public** **void** setPrice(**double** price) {

**this**.price = price;

}

}

**package** eg00235;

**public** **class** Fruit **implements** Item {

**private** String name;

**private** **int** weight;

**private** **double** pricePerKg;

**public** Fruit() {

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **int** getWeight() {

**return** weight;

}

**public** **void** setWeight(**int** weight) {

**this**.weight = weight;

}

**public** **double** getPricePerKg() {

**return** pricePerKg;

}

**public** **void** setPricePerKg(**double** pricePerKg) {

**this**.pricePerKg = pricePerKg;

}

**public** Fruit(String name, **int** weight, **double** pricePerKg) {

**super**();

**this**.name = name;

**this**.weight = weight;

**this**.pricePerKg = pricePerKg;

}

@Override

**public** **double** accept(ShoppingCartVisitor visitor) {

**return** visitor.visit(**this**);

}

}

**package** eg00235;

**public** **class** ShoppingCartVisitorImpl **implements** ShoppingCartVisitor {

@Override

**public** **double** visit(Book book) {

**double** finalBooksCost = 0.0;

**if** (book.getPrice() > 500) {

finalBooksCost = book.getPrice() - 100;

} **else** {

finalBooksCost = book.getPrice();

}

System.***out***.println("Book: " + book.getName() + ",book isbn : " + book.getIsbnNumber() + ",final books cost: "

+ finalBooksCost);

**return** finalBooksCost;

}

@Override

**public** **double** visit(Fruit fruit) {

**double** finalFruitsCost = fruit.getPricePerKg() \* fruit.getWeight();

System.***out***.println("Fruit: " + fruit.getName() + ",final fruits cost: " + finalFruitsCost);

**return** finalFruitsCost;

}

}

**package** eg00235;

**public** **class** Tester {

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

Tester obj = **new** Tester();

Item[] items = **new** Item[4];

items[0] = **new** Book("Core Java", "8234", 450.0);

items[1] = **new** Book("Hibernate", "8234", 700.0);

items[2] = **new** Fruit("Mango", 2, 100.0);

items[3] = **new** Fruit("Apple", 3, 200.0);

**double** totalCostOfShopping = obj.calculateTotalCost(items);

System.***out***.println("Total Shopping Cost: " + totalCostOfShopping);

}

**public** **double** calculateTotalCost(Item[] items) {

ShoppingCartVisitor shoppingCartVisitor = **new** ShoppingCartVisitorImpl();

**double** totalCost = 0.0;

**for** (Item item : items) {

totalCost += item.accept(shoppingCartVisitor);

}

**return** totalCost;

}

}