**Design Patterns**

A design pattern provides a general reusable solution for the common problems that occur in software design. The pattern typically shows relationships and interactions between classes or objects. The idea is to speed up the development process by providing well tested, proven development/design paradigm. Design patterns are programming language independent strategies for solving a common problem. That means a design pattern represents an idea, not a particular implementation. By using the design patterns you can make your code more flexible, reusable, and maintainable.

It’s not mandatory to implement design patterns in your project always. Design patterns are not meant for project development. Design patterns are meant for common problem-solving. Whenever there is a need, you have to implement a suitable pattern to avoid such problems in the future. To find out which pattern to use. You just have to try to understand the design patterns and their purposes. Only by then, you will be able to pick the right one.

So as a programmer or developer generally we documents all our best practices that we need to follow and also the common recurring problems and proven those solution to the problem. This is what patterns came into picture. A design patterns help us in identifying recurring problems and provide a ready to use proven solution.

**What is Pattern?**

The idea of design patters is came for civil engineering and the gang of four (Eric, Richard, Ralph, John) has adopted this idea and provide different design pattern in software designing these patterns are called GOF or gang of four patterns.

So in short Design Patterns provide a solution for problem in given environment.

**Why Patterns?**

Helps in identifying recurring problems

Reuse the design patterns across the project

Provide a better organized structure of the project

Provide a common design language so that other people can easily understand the each other with the help of design pattern name we are using in the project.

**How the pattern is identified or form?**

First of all if when are encountered with a problem then first of all we try to find out the solution from existing design pattern and we don’t get the solution from existing design pattern then we document the problem’s solution and use in the project initially this is called candidate pattern, let’s say this solution is now being used across the project and on other projects then this candidate pattern becomes design patterns.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Design Pattern 🡺 | GOF Pattern or Core Pattern 🡺 | Creational | Structural | Behavioral |
| J2EE Pattern | | | |

**Creational** Pattern provide guide line to instantiate a single object or a group of object i.e. it all about object creation. The **structural** pattern provides a way to define a relationship among the classes. And **Behavioral** patterns defines haw a communication happens among classes and objects.

**J2EE Pattern:** In this design pattern every java applications are organized into multiple logical layer like

DataAccesLayer (DAO) layer for Database activity

Service layer for business activity

Integration layer for integration purpose

Presentation layer for presentation purpose

**Types of Core Design Patterns**

There are mainly three types of design patterns:

**Creational**: These design patterns are all about class instantiation or object creation. Creational design patterns are the Factory Method, Abstract Factory, Builder, Singleton, Factory, Object Pool, and Prototype.

**Structural**: These design patterns are about organizing different classes and objects to form larger structures and provide new functionality. Structural design patterns are Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Private Class Data, and Proxy.

**Behavioral:** Behavioral patterns are about identifying common communication patterns between objects and realize these patterns. Behavioral patterns are Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Null Object, Observer, State, Strategy, Template method, Visitor

|  |  |  |
| --- | --- | --- |
| 1.Creational Design Pattern  1. Factory Pattern 2. Abstract Factory Pattern 3. Singleton Pattern 4. Prototype Pattern 5. Builder Pattern. | 2. Structural Design Pattern  1. Adapter Pattern 2. Bridge Pattern 3. Composite Pattern 4. Decorator Pattern 5. Facade Pattern 6. Flyweight Pattern 7. Proxy Pattern | 3. Behavioral Design Pattern  1. Chain Of Responsibility Pattern 2. Command Pattern 3. Interpreter Pattern 4. Iterator Pattern 5. Mediator Pattern 6. Memento Pattern 7. Observer Pattern 8. State Pattern 9. Strategy Pattern 10. Template Pattern 11. Visitor Pattern |

**Singleton Design Pattern:**

It is an object creational pattern which allows our application to create one and only one instance of a particular class no matter how many times class is used in our application.

For example PropertyReader(C) class that can read property from a file and it is used multiple times in our application by different classes but we are going to create only single object of the PropertyReader not multiple times and thus saving lot of memories

Another example for singleton class is **Logger class**. Using Logger class we can log different type of information (like error, debug, Info etc) from our application to log file, so that developer can read that file latter on and can analyze.

So Logger is a singleton class and same logger object can be shared across the application classes.

Another example in the JDBC, the java database connectivity **DataSource class** in the JDBC API, the DataSource class is responsible for maintaining a connection pool and giving a connection from the pool to our application classes. So here one instance of DataSource could be using by several classes in the application.

For any java class we are allowed to create only one object then such type of class is called singleton class. Example: Runtime, BusinessDeligate, ServiceLocator.

**UML Class diagram for singleton class:**

|  |  |
| --- | --- |
| DateUtil ()  -instance: DateUtil  -DateUtil ()  +getInstance (): DateUtil | 1. **DateUtil() represents the class name** 2. **-instance : DateUtil – represents the static field of class DateUtil type** 3. **-DateUtil() represents the private constructor so that no one can create the object from outside** 4. **+getInstance() : DateUtil represents the public static method of class DateUtil type and here we will return the static field instance or object** |

**Steps to create singleton class:**

1. Declare the constructor of class as private: So that no other class can create the instance of this class directly
2. Declare public static method which can be used by all other classes to create an object of singleton class
3. Declare a private static member variable of same class type in the class so that we can assign the object to this member and we can return back from the public static method

**Program to create singleton class using above steps:**

|  |  |
| --- | --- |
| **package** com.design.pattern.singleton;  **public** **class** SingletonUsingThreeSteps {  **// Static Member variable of class type**  **private** **static** SingletonUsingThreeSteps *instance* = **null**;  **// private constructor**  **private** SingletonUsingThreeSteps() {  }  **// Static method of class type**  **public** **static** SingletonUsingThreeSteps getInstance() {  **if** (*instance* == **null**) {  *instance* = **new** SingletonUsingThreeSteps();  }  **return** *instance*;  }  } | **package** com.design.pattern.singleton;  **public** **class** TestSingleton {  **public** **static** **void** main(String[] args) {  SingletonUsingThreeSteps t1 = SingletonUsingThreeSteps.*getInstance*();  SingletonUsingThreeSteps t2 = SingletonUsingThreeSteps.*getInstance*();  **if** (t1 == t2) {  System.***out***.println(" ture ");  }  }  }  Output: true |

**Eager Initialization:** Here in the above example first we are checking if instance is equal to null then only create and initialize the object and return the same. This is called lazy initialization.

But we can create and initialize the object very first and then return the same object in the getInstance () method and this is called Eager Initialization i.e. the object gets created even before calling getInstance () from outside the class. This is good when we are pretty sure that this class will be used at least once otherwise unnecessary object will be created inside the singleton class itself

|  |
| --- |
| **package** com.design.pattern.singleton;  **public** **class** SingletonUsingEagerInitilization {  // Static Member variable of class type  **private** **static** SingletonUsingEagerInitilization *instance* = **new** SingletonUsingEagerInitilization();  // private constructor  **private** SingletonUsingEagerInitilization() {  }  // Static method of class type  **public** **static** SingletonUsingEagerInitilization getInstance() {  **return** *instance*;  }  } |
| **Eagar Initialization using static block**  **package** com.design.pattern.singleton;  **public** **class** SingletonEagerUsingStaticBlock {  // Static Member variable of class type  **private** **static** SingletonEagerUsingStaticBlock *instance*;  // Create object inside static block  **static** {  *instance* = **new** SingletonEagerUsingStaticBlock();  }  // private constructor  **private** SingletonEagerUsingStaticBlock() {  }  // Static method of class type  **public** **static** SingletonEagerUsingStaticBlock getInstance() {  **return** *instance*;  }  }  **Note: Here the instance will be created when the class will be loaded into the memory** |
| **package** com.design.pattern.singleton;  **public** **class** TestSingleton {  **public** **static** **void** main(String[] args) {  SingletonUsingEagerInitilization t1 = SingletonUsingEagerInitilization. *getInstance*();  SingletonUsingEagerInitilization t2 = SingletonUsingEagerInitilization. *getInstance*();  **if** (t1 == t2) {  System.***out***.println(" ture ");  }  }  }  Output : true: |

So the eager initialization is good when we are pretty sure that our singleton class will be used at least once, otherwise it is recommended to go with lazy initialization with double check.

**Handle multiple threads in Singleton class: Or Singleton class for multithread:**

So the above program of singleton class is not thread safe, in case of multithreaded environment there may be a chance, more than one object could be created by threads. Like in case of Lazy initialization even though we have we have checked if (instance == null) then only creates the object but let’s say one thread comes and it founds the condition true and then it start executing line [**instance = new SingletonEagerUsingStaticBlock();**] for creating object and by that time let’s say another thread also got the chance and before creating new object by first thread the second thread found the condition true and thus second thread also come inside the if condition and thus both the object will create their object which will violate Singleton class rules.

So the solution to this problem is- let’s use synchronized keyword at the method level or synchronized block for the particular line which is responsible for object creation.

|  |  |
| --- | --- |
| // Static method of class type  **public** **static** **synchronized** SingletonUsingThreeSteps getInstance() {  **if** (*instance* == **null**) {  *instance* = **new** SingletonUsingThreeSteps();  }  **return** *instance*;  } | Here we have made whole method as synchronized hence whole method will be locked by one thread only at a time and other thread have to wait for getting lock |
| // Static method of class type  **public** **static** SingletonUsingThreeSteps getInstance() {  **synchronized** (SingletonUsingThreeSteps.**class**) {  **if** (*instance* == **null**) {  *instance* = **new** SingletonUsingThreeSteps();  }  **return** *instance*;  }  } | Here instead of locking whole method we have used synchronized block to lock the particular line responsible for object creation. |
| // making Static Member variable volatile of class type  **private** **static** **volatile** SingletonUsingThreeSteps *instance*;  // Static method of class type  **public** **static** SingletonUsingThreeSteps getInstance() {  **if** (*instance* == **null**) {  **synchronized** (SingletonUsingThreeSteps.**class**) {  **if** (*instance* == **null**)  *instance* = **new** SingletonUsingThreeSteps();  System.***out***.println("Object created ");  }  }  **return** *instance*;  } | This is more improved than before because using synchronized block is very costly as there could be performance issue so here first we have checked [if instance ==null)] then only enter into synchronized block and again we are checking double [if instance ==null] then create the object.  And another one is we are making static class member variable volatile to make more thread safe. |

So this way we can handle multi thread in singleton class: Remaining code for multithread singleton class:

|  |
| --- |
| **package** com.design.pattern.singleton;  **public** **class** SingletonWithMultiThread **extends** Thread {  SingletonUsingThreeSteps ms;  **public** SingletonWithMultiThread(SingletonUsingThreeSteps obj) {  **this**.ms = obj;  }  @Override  **public** **void** run() {  SingletonUsingThreeSteps t1 = SingletonUsingThreeSteps.*getInstance*();  SingletonUsingThreeSteps t2 = SingletonUsingThreeSteps.*getInstance*();  System.***out***.println((t1 == t2));  }  } |
| **package** com.design.pattern.singleton;  **public** **class** TestSingleton {  **public** **static** **void** main(String[] args) **throws** InterruptedException {  System.***out***.println("Main thread...");  SingletonUsingThreeSteps obj = SingletonUsingThreeSteps.*getInstance*();  SingletonWithMultiThread t1 = **new** SingletonWithMultiThread(obj);  SingletonWithMultiThread t2 = **new** SingletonWithMultiThread(obj);  t1.start();  // t1.sleep(2000);  t2.start();  }  }  **Output:**  Main thread...  Object created  true  true |

Volatile Keyword in Java

Volatile keyword is used to modify the value of a variable by different threads. It is also used to make classes thread safe. It means that multiple threads can use a method and instance of the classes at the same time without any problem. The volatile keyword can be used either with primitive type or objects.

The volatile keyword does not cache the value of the variable and always read the variable from the main memory. The volatile keyword cannot be used with classes or methods. However, it is used with variables. It also guarantees visibility and ordering. It prevents the compiler from the reordering of code.

The contents of the particular device register could change at any time, so you need the volatile keyword to ensure that such accesses are not optimized away by the compiler.

**Serialization and Singleton**

While working with singleton classes then we need to handle serialization problem.

|  |  |
| --- | --- |
| **package** com.design.pattern.singleton;  **import** java.io.Serializable;  **public** **class** SingletonUsingThreeSteps **implements** Serializable {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  // making Static Member variable volatile of class type  **private** **static** **volatile** SingletonUsingThreeSteps *instance*;  // private constructor  **private** SingletonUsingThreeSteps() {  }  // Static method of class type  **public** **static** SingletonUsingThreeSteps getInstance() {  **if** (*instance* == **null**) {  **synchronized** (SingletonUsingThreeSteps.**class**) {  *instance* = **new** SingletonUsingThreeSteps();  System.***out***.println("Object created ");  }  }  **return** *instance*;  }  } | Here we have implements Serializable interface.  Now in the main method let’s create object of this serialized class and then save this serialized object into file and further de-sterilize the same object and then compare both the object (serialized and desterilized object) and check still both the object are same or got changed |
| **package** com.design.pattern.singleton;  **import** java.io.File;  **import** java.io.FileInputStream;  **import** java.io.FileOutputStream;  **import** java.io.ObjectInputStream;  **import** java.io.ObjectOutputStream;  **public** **class** TestSingletonForSerilization {  **public** **static** **void** main(String[] args) **throws** Exception {  SingletonUsingThreeSteps t1 = SingletonUsingThreeSteps.*getInstance*();  **// Serialize the object and saving into file**  @SuppressWarnings("resource")  ObjectOutputStream oos = **new** ObjectOutputStream(**new** FileOutputStream(**new** File("D://FileFolder/DateUtil.ser")));  oos.writeObject(t1);  // SingletonUsingThreeSteps t2= SingletonUsingThreeSteps.getInstance();  // now read the saved file back from the same location i.e. desterilized the serialized object back  SingletonUsingThreeSteps t2 = **null**;  @SuppressWarnings("resource")  ObjectInputStream ois = **new** ObjectInputStream(**new** FileInputStream(**new** File("D://FileFolder/DateUtil.ser")));  t2 = (SingletonUsingThreeSteps) ois.readObject();  System.***out***.println((t1 == t2));  }  }  **Output : False** | |

So here we have serialized the object (t1) and saved into file and again desterilized the same object from the file and then compare the object before serialized and after serialized and found that both the object are now different which is violating the singleton concept.

**Q: Now the question comes how to solve this problem? I.e. how to hand de-serialization in singleton class?**

|  |  |
| --- | --- |
| **package** com.design.pattern.singleton;  **import** java.io.Serializable;  **public** **class** SingletonUsingThreeSteps **implements** Serializable {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  // making Static Member variable volatile of class type  **private** **static** **volatile** SingletonUsingThreeSteps *instance*;  // private constructor  **private** SingletonUsingThreeSteps() {  }  // Static method of class type  **public** **static** SingletonUsingThreeSteps getInstance() {  **if** (*instance* == **null**) {  **synchronized** (SingletonUsingThreeSteps.**class**) {  *instance* = **new** SingletonUsingThreeSteps();  System.***out***.println("Object created ");  }  }  **return** *instance*;  }  // this method will be invoked internally which returns same object (instance) during de-serialization  **private** Object readResolve() {  System.***out***.println("Inside the readResolve");  **return** *instance*;  }  } | So to solve the problem we have implemented **private** readResolve() method which returns the same object what we have created first time  **private** Object readResolve() {  **return** *instance*;  }  So :  Step-1: implements Serializable interface  Step-2: implements the readResolve()  Step-3: Return the same instance what we have created first time  This method will be internally invoked during de-serialization and thus during the de-serialization also we will get same object what we have created first time and thus we can handle de-serialization in singleton class.  Note: readResolve() is unique name and should not be changed or modified otherwise during de-serialization it will be invoked and hence we will not get expected result |
| **package** com.design.pattern.singleton;  **import** java.io.File;  **import** java.io.FileInputStream;  **import** java.io.FileOutputStream;  **import** java.io.ObjectInputStream;  **import** java.io.ObjectOutputStream;  **public** **class** TestSingletonForSerilization {  **public** **static** **void** main(String[] args) **throws** Exception {  SingletonUsingThreeSteps t1 = SingletonUsingThreeSteps.*getInstance*();  // Serialize the object and saving into file  @SuppressWarnings("resource")  ObjectOutputStream oos = **new** ObjectOutputStream(**new** FileOutputStream(**new** File("D://FileFolder/DateUtil.ser")));  oos.writeObject(t1);  System.***out***.println("Object t1 got serilized..");  // SingletonUsingThreeSteps t2= SingletonUsingThreeSteps.getInstance();  // now read the saved file back from the same location  SingletonUsingThreeSteps t2 = **null**;  @SuppressWarnings("resource")  ObjectInputStream ois = **new** ObjectInputStream(**new** FileInputStream(**new** File("D://FileFolder/DateUtil.ser")));  t2 = (SingletonUsingThreeSteps) ois.readObject();  System.***out***.println("Object t1 got de-serilized and store into t2.");  System.***out***.println((t1 == t2));  }  }  Output:  Object created  Object t1 got serilized..  Object t1 got de-serilized and store into t2.  true | |

Now both the objects are same

**Override the clone () method in singleton class:**

One last thing we need to handle working with singleton object is to ensure that the singleton object should not be Clonable; otherwise it can be modified by cloning the singleton class object.

To achieve this:

Step-1: Implements Closeable interface In the singleton class

Step-2: override clone () and return new exception [**return** **new** CloneNotSupportedException ();] so that if anyone client class try to clone our object or directly or by creating child class of singleton then CloneNotSupportedException () exception returns.

|  |
| --- |
| **package** com.design.pattern.singleton;  **import** java.io.Serializable;  **public** **class** SingletonUsingThreeSteps **implements** Serializable, Cloneable {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  // making Static Member variable volatile of class type  **private** **static** **volatile** SingletonUsingThreeSteps *instance*;  // private constructor  **private** SingletonUsingThreeSteps() {  }  // Static method of class type  **public** **static** SingletonUsingThreeSteps getInstance() {  **if** (*instance* == **null**) {  **synchronized** (SingletonUsingThreeSteps.**class**) {  *instance* = **new** SingletonUsingThreeSteps();  System.***out***.println("Object created ");  }  }  **return** *instance*;  }  // this method will be invoked internally which returns same object (instance)  // during de-serialization  **private** Object readResolve() {  **return** *instance*;  }  @Override  **protected** Object clone() **throws** CloneNotSupportedException {  // return super.clone();  **return** **new** CloneNotSupportedException();  }  } |

**Assignment**: implement a Logger class which should be singleton, this Logger class should have log (String message) method which takes string and displays that string on the console using sysout. Make sure you have handled multithreaded condition i.e. make the thread safe and also make your singleton class should not be cloned.

|  |
| --- |
| **package** com.design.pattern.singleton;  **import** java.io.Serializable;  **public** **class** LoggerSingleton **implements** Cloneable, Serializable {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  **private** **static** **volatile** LoggerSingleton *instance*;  **private** LoggerSingleton() {  }  //Safe from breaking Singleton from Multithreading  **public** **static** LoggerSingleton getInstance() {  **if** (*instance* == **null**) {  **synchronized** (LoggerSingleton.**class**) {  **if** (*instance* == **null**)  *instance* = **new** LoggerSingleton();  }  }  **return** *instance*;  }    **public** **void** log(String message) {  System.***out***.println("Log message: " + message);  }  //Safe from breaking Singleton from Serialization  **public** Object readResolve() {  **return** *instance*;  }  //Safe from breaking Singleton from Cloning  @Override  **protected** Object clone() **throws** CloneNotSupportedException {  **throw** **new** CloneNotSupportedException();  }  } |

**Error, Info, debug method in Logger singleton class:**

|  |  |
| --- | --- |
| **package** com.design.pattern.singleton;  **import** java.io.Serializable;  **public** **class** LoggerSingleton **implements** Cloneable,Serializable {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  **private** **static** **volatile** LoggerSingleton *instance*;  **private** LoggerSingleton() {  }//Safe from breaking Singleton from Multithreading  **public** **static** LoggerSingleton getInstance() {  **if** (*instance* == **null**) {  **synchronized** (LoggerSingleton.**class**) {  **if** (*instance* == **null**)  *instance* = **new** LoggerSingleton();  }  }  **return** *instance*;  }  **public** **void** log(String message) {  System.***out***.println("Log message: " + message);  }  **public** **void** info(String message) {  System.***out***.println("Info message: " + message);  }  **public** **void** error(String message) {  System.***out***.println("error message: " + message);  }  **public** **void** debug(String message) {  System.***out***.println("debug message: " + message);  }  //Safe from breaking Singleton from Serialization  **public** Object readResolve() {  **return** *instance*;  }  //Safe from breaking Singleton from Cloning  @Override  **protected** Object clone() **throws** CloneNotSupportedException {  **throw** **new** CloneNotSupportedException();  }  } | **package** com.design.pattern.singleton;  **public** **class** TestLoggerSingleton {  **public** **static** **void** main(String[] args) {  LoggerSingleton logger = LoggerSingleton.*getInstance*();  LoggerSingleton logger1 = LoggerSingleton.*getInstance*();  logger.log("Please enter your log message...");  logger.info("Please enter your info logger...");  logger.error("Please enter your error logger...");  logger.debug("Please enter your debugger logger...");  System.***out***.println(logger == logger1);  }  }  Output:  Log message: Please enter your log message...  Info message: Please enter your info logger...  error message: Please enter your error logger...  debug message: Please enter your debugger logger...  true |

Advantages:

Concept: If several people have same requirement then it is not recommended to create separate object for every requirements. Create only one object and reuse the same object for every similar requirement. This is the concept of singleton classes. And it increase performance and memory utilization. This is the central ideas of singleton classes.

For example:

|  |  |
| --- | --- |
| **Runtime r1 = Runtime.getRuntime();**  **Runtime r2 = Runtime.getRuntime();**  **.**  **.**  **Runtime r100 = Runtime.getRuntime();** | if (r1 == r2) { System.out.println("Equal");} Output : Equal  so every object will get same memory and every time we will get same object instead of creating several time |

**How to create our own singleton class: Approach 1(Eager Initialization)**

|  |  |
| --- | --- |
| **class Test {**  **// private static variable**  **private static Test t = new Test();**  **private Test() {**  **// private constructor**  **}**  **public static Test getTest() {**  **return t;**  **}**  **}** | **Test t1 = Test.getTest();**  **Test t2 = Test.getTest();**  **if (t1 == t2) {**  **System.out.println("Eqal Test Object");**  **}**  Created one private static variable of class type which returns Test class object.  Create one private constructor so that no one can call from outside of this class and create object by initialize  Create one static method and return the private static Test object (t)  Now from outside when we call this static method using class name then every time it would return the same object. |

For creating our own singleton class we have to use

Private constructor

Private static variable and

Public factory method and return the same static variable inside this class

Runtime singleton class is based on the above approach.

**Approach 2- Lazy Initialization**

|  |  |
| --- | --- |
| **class Test2{**  **private static Test2 t =null;**  **private Test2() {**  **}**  **public static Test2 getSingletonObj() {**  **if(t==null) {**  **t = new Test2();**  **}**  **// else always return t**  **return t;**  **}**  **}** | **Test2 t3 = Test2.getSingletonObj();**  **Test2 t4 = Test2.getSingletonObj();**  **if (t3 == t4) {**  **System.out.println("Same Object");**  **}**  Output: Same Object  In the first approach the object got created first only and then same object we are returning in factory method.  In the second approach object is getting created inside factory method and then same object is getting return. |

If the class is final then it is not possible to create child or sub class. Now Class is not final even though it is not possible to create child or sub class, how it is possible? I.e. Class is not final but we are not allowed to create child classes how it is possible?

Just make the entire constructor private, hence

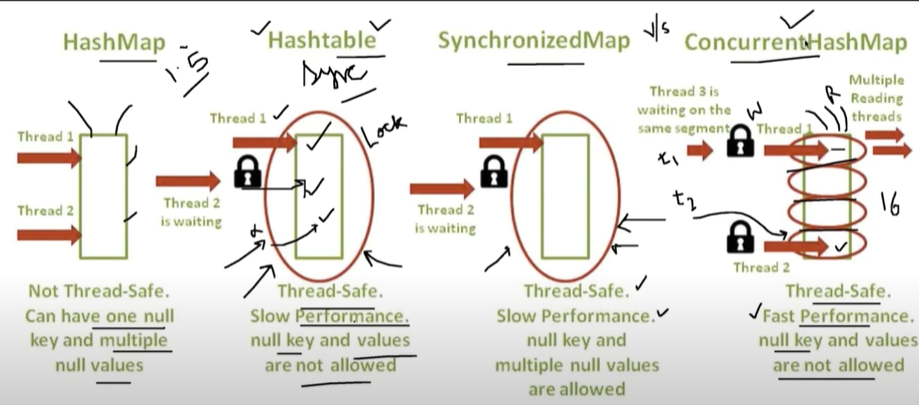
|  |  |  |
| --- | --- | --- |
| **class Test {**  **private Test() {**  **// private constructor**  **}** | **Class child extends Test{**  **Test(){**  **}**  **}** | Here we will get compile time error because private constructor cannot be accessed outside of the class and here we are trying to extend class have private constructor so by default private constructor would be available but because of private it does not allows to implement this its child class and hence we will get compile time error. |

Hence we can restrict child class creation

Q:How many locks can we have in concurrenthashmap?

32 locks

It means each thread can work on a each segment during high concurrency and atmost 32 threads can operate at max which simply maintains 32 locks to guard each bucket of the Concurrenthashmap



|  |  |
| --- | --- |
|  |  |
|  |  |

**Factory Pattern:**

1. It is a creational pattern that abstract or hide the object creation process.
2. For example [A car factory, Chocolate Factory, A Toy Factory]
3. In the above example a dealer can ask to deliver the cars , chocolate or toys and need not to be worried about its manufacturing processes. Here dealer can ask the factory to provide different types of product and its factory responsibility to deliver the product
4. Similarly Factory pattern is an object creational pattern that hides the complexity of object creation to its client
5. For example in our JDBC class when we need different types of databases like (Oracle, MySql, SqlServer). So in java we have JDBC driver called DriverManager

Use Case:

1. We will have one Pizza store
2. In the Pizza Interface we will have three methods prepare(): void , bake():void, cut():void
3. Create three classes VezPizza, CheesPizza, ChickenPizza which implements Pizza Interface
4. Now create one Factory class and inside that create one static method of Pizza interface type which returns the objects of the above classes.

Assignment:

Allows a client to create a male and female person. Create an interface Person having method wish (String wish) and this interface will be implemented by class Male and Female and provide the implementation for wish() method. The client should be able to create one of the two class object only through PersonFactory by passing male or female.

**Example to be mentioned:**

**Abstract Factory Pattern :( Factory of Factories)**

As we have just learnt factory pattern where we used to get the object of the classes without going into detail that how the object is being created, i.e. object creation process is completely hide from the client.

There is not much difference between Factory Pattern and Abstract Factory Pattern.

An Abstract Factory is a Factory of Factory. The factory patter was hiding the process of object creation while abstract factory is a factory which is hiding the process of creation of factories itself.

Use Case-1

A good example in java space is JAXP API. JAXP stands for java API for xml parsing, using this API we can read, write & update element in a XML file. The main class in this API is Document class that put the xml document in memory. To create a document class we use DocumentBuilder which is nothing but a factory class and there one more class DocumentBuilderFactory which is responsible for creating DocumentBuilder factory class itself. DocumentBuilderFactory is an Abstract Factory because it is factory of factories.

Use case-2: Another example is DAO (Data Access Object) factory. This is simply a class which can read, write, and update data. We can have different types of DAO class. Like DAO class which deals with XML data and DAO that deals with DB data. Within xml we can have employee information, department information. Similarly in the DB we can have employee information and department information.

|  |  |  |
| --- | --- | --- |
| DAOAbstractFactory(C) | DBDaoFactory(I) | XMLEmpDAO(C) |
| XMLDeptDAO(C) |
| XMLDaoFactory(I) | DBEmpDAO(C) |
| DBDeptDAO(C) |

So as per above discussion we can see that we have factories to deal with separate DAOs. We can have a DBDaoFactory(Interface) which will deals with DBEmpDao(c) & DBDepDao(c) and similarly we can have XMLDaoFactory(I) which will deals with XMLEmpDao(C) & XMLDeptDao(C).

Now these two [DBDaoFactory(I), XMLDaoFactory(I)] will extends DAOAbstractFactory(C) and we will have DAOProducer which will be responsible for creating these two factories [DBDaoFactory(I), XMLDaoFactory(I)]

Abstract Factory UML Class Diagram:

1. Create one Dao. Java Interface having one save() method prototype
2. Create four methods which implements the Dao.java interface [

DBEmpDao.java implements Dao

DBDeptDao.java implements Dao

XmlEmpDao.java implements Dao

XmlDeptDao.java implements Dao

1. Now Create one abstract class [**public** **abstract** **class** DaoAbstractFactory{] having one abstract method of Dao interface return type. **public** **abstract** Dao createDao(String type);}
2. Now create two class which will extend the above abstract class for returning object of four classes based on the type

**1-public** **class** DBDaoFactory **extends** DaoAbstractFactory returns the object of DBEmpDao and DBDeptDao java by overriding createDao() method

**2- public** **class** XmlDaoFactory **extends** DaoAbstractFactory returns the object of XmlEmpDao and XmlDeptDao java by overriding createDao() method

1. Now will create factory of factories

**public** **static** DaoAbstractFactory getDaoTypeInstance(String daoFactryType)

It will create the object of two factories (XmlDaoFactory class & DBDaoFactory class) and these two factory class in-turn create the object of above classes mentioned in the point 2

1. These two factories classes (XmlDaoFactory class & DBDaoFactory class) extends one abstract class and implement one abstract method crete

|  |
| --- |
| **Interface:**  **package** com.design.pattern.abstractfactory;  **public** **interface** Dao {  **public** **void** save();  } |

|  |  |
| --- | --- |
| **XML\_EMP\_DEPT\_DAO\_Class** | |
| **package** com.design.pattern.abstractfactory;  **public** **class** XmlDeptDao **implements** Dao {  @Override  **public** **void** save() {  System.***out***.println("Saving XmlDeptDao into XML .. ");  }  } | **package** com.design.pattern.abstractfactory;  **public** **class** XmlEmpDao **implements** Dao {  @Override  **public** **void** save() {  System.***out***.println("Saving XmlEmpDao into XML .. ");  }  } |
| **DB\_EMP\_DEPT\_DAO\_Class** | |
| **package** com.design.pattern.abstractfactory;  **public** **class** DBDeptDao **implements** Dao {  @Override  **public** **void** save() {  System.***out***.println("Saving DBDeptDao into DB ");  }  } | **package** com.design.pattern.abstractfactory;  **public** **class** DBEmpDao **implements** Dao {  @Override  **public** **void** save() {  System.***out***.println("Saving DBEmpDao into DB .. ");  }  } |

|  |
| --- |
| **Abstract Class:**  **package** com.design.pattern.abstractfactory;  public abstract class DaoAbstractFactory {  public abstract Dao createDaoForEmpOrDept(String typeOfEmpOrDept);  } |

Creating factory class which created object of about four classes by implementing abstract class method

|  |  |
| --- | --- |
| **package** com.design.pattern.abstractfactory;  **public** **class** **XmlDaoFactory** **extends** **DaoAbstractFactory** {  @Override  **public** **Dao** **createDaoForEmpOrDept**(String type) {  Dao dao = **null**;  **if** (type.equalsIgnoreCase("emp")) {  dao = **new** XmlEmpDao();  } **else** **if** (type.equalsIgnoreCase("dept")) {  dao = **new** XmlDeptDao();  }  **return** dao;  }  } | **package** com.design.pattern.abstractfactory;  **public** **class** **DBDaoFactory** **extends** DaoAbstractFactory {  @Override  **public** Dao createDaoForEmpOrDept(String type) {  Dao dao = **null**;  **if** (type.equalsIgnoreCase("emp")) {  dao = **new** DBEmpDao();  } **else** **if** (type.equalsIgnoreCase("dept")) {  dao = **new** DBDeptDao();  }  **return** dao;  }  } |
| **Create Factory of above two factories class**  **package** com.design.pattern.abstractfactory;  **public** **class** DaoFactoryProducer {    **public** **static** DaoAbstractFactory getFactoriesDaoInstance(String daoFactryType) {  DaoAbstractFactory daf = **null**;  **if** (daoFactryType.equalsIgnoreCase("XML")) {  daf = **new** **XmlDaoFactory**();  } **else** **if** (daoFactryType.equalsIgnoreCase("DB")) {  daf = **new** **DBDaoFactory**();  }  **return** daf;  }  } | |

|  |
| --- |
| **package** com.design.pattern.abstractfactory;  **public** **class** TestAbstractPattern {  **public** **static** **void** main(String[] args) {  DaoAbstractFactory dao = DaoFactoryProducer.*getFactoriesDaoInstance*("DB");  Dao obj = dao.createDaoForEmpOrDept("dept");  obj.save();  }  }  Output:  Saving DBDeptDao into DB... |
| Now as soon as we will give the different values we will get different output.  *getFactoriesDaoInstance(“****XML****”); and change the*  createDaoForEmpOrDept("emp");  **Output**:  Saving XMLEmpDao into XML... |

Benefits of Abstract Factory Pattern:

1. Abstract factory provides the abstraction of Factories(i.e. Hide the object creation of Factory of factories class) , whereas factory pattern provides abstraction for Concrete products (i.e. Hide the object creation of Factory’s product classes)
2. Better Abstraction than factory pattern
3. Loosely coupled code
4. Easy to extend

**Builder Design Pattern:**

Builder pattern was introduced to solve some of the problems or short comings with Factory and Abstract Factory design patterns when the Object contains a lot of attributes.

There are few major issues with Factory and Abstract Factory design patterns when the Object contains a lot of

**When to use Builder Pattern:**

* Whenever a class has too many variables and many of them have same data type, then it leads confusion for the client program if we use constructor or factory pattern for initializing them for some calculation. We have to be very sure passing of the parameter in the correct sequence and at the correct location.
* When some of the class variables are optional, so in this case client will have to pass Null if factory pattern is used as in the factory pattern all the parameter must be passed whether is mandatory or optional

For example

|  |  |
| --- | --- |
|  | **package** com.design.pattern.builder;  **public** **class** Phone {  **private** String os;  **private** String processor;  **private** **double** screenSize;  **private** **int** battery;  **private** **int** camera;  **public** Phone(String os, String processor,**double** screenSize , **int** battery, **int** camera) {  **super**();  **this**.os = os;  **this**.processor = processor;  **this**.screenSize = screenSize;  **this**.battery = battery;  **this**.camera = camera;  }  @Override  **public** String toString() {  **return** "Phone[OS" + os + ", Processor" + processor + ",Screen Size" + screenSize + ", Battery" + battery  + ", Camera" + camera + "]";  }  }  Test.java  **package** com.design.pattern.builder;  **public** **class** Test {  **public** **static** **void** main(String[] args) {  Phone p = **new** Phone("Android", "2.5 GZh", 17.5, 5000, 64);  System.***out***.println(p);  }  } |

Output: Phone [OSAndroid, Processor2.5 GZh, Screen Size17.5, Battery5000, Camera64]

So as here we know that in this case all the variable must be passed in the constructor whether it is mandatory or not and if the don’t pass the value in the constructor in the client (Test) class then we will get compile time error and one more thing we will have to pass the value according to sequence order define in the constructor.

So here to get the phone information we have to pass all parameters, let say someone want the phone feature just according to screen size or battery then in this case it is not possible to get the phone information.

So to solve the above problems Builder Design pattern is used. In this pattern we will create setter () for each varaible and ther we will set the value and the return he same value

|  |  |
| --- | --- |
|  | **package** com.design.pattern.builder;  **public** **class** PhoneBuilder {  **private** String os;  **private** String processor;  **private** **double** screenSize;  **private** **int** battery;  **private** **int** camera;  **public** **void** setOs(String os) {  **this**.os = os;  }  **public** **void** setProcessor(String processor) {  **this**.processor = processor;  }  **public** **void** setScreenSize(**double** screenSize) {  **this**.screenSize = screenSize;  }  **public** **void** setBattery(**int** battery) {  **this**.battery = battery;  }  **public** **void** setCamera(**int** camera) {  **this**.camera = camera;  }  @Override  **public** String toString() {  **return** "Phone[OS" + os + ", Processor" + processor + ",Screen Size" + screenSize + ", Battery" + battery  + ", Camera" + camera + "]";  }} |

Note: Here we have crated the setter() method of each variable but, note that, these setter() method are just setting the values but not returning any thing, so instead of normal setter() method we have created setter () of class type(PhoneBuilder) as shown in the LSH screen so let’s change the return type

|  |
| --- |
| **package** com.design.pattern.builder;  **public** **class** PhoneBuilder {  **private** String os;  **private** String processor;  **private** **double** screenSize;  **private** **int** battery;  **private** **int** camera;  **public** PhoneBuilder setOs(String os) {  **this**.os = os;  **return** **this**;  }  **public** PhoneBuilder setProcessor(String processor) {  **this**.processor = processor;  **return** **this**;  }  **public** PhoneBuilder setScreenSize(**double** screenSize) {  **this**.screenSize = screenSize;  **return** **this**;  }  **public** PhoneBuilder setBattery(**int** battery) {  **this**.battery = battery;  **return** **this**;  }  **public** PhoneBuilder setCamera(**int** camera) {  **this**.camera = camera;  **return** **this**;  }  @Override  **public** String toString() {  **return** "Phone[OS" + os + ", Processor" + processor + ",Screen Size" + screenSize + ", Battery" + battery  + ", Camera" + camera + "]";  }  } |

One more advantage of using class type retrun type, so that we can se the value at the runtime.

|  |  |
| --- | --- |
| **package** com.design.pattern.builder;  **public** **class** Test {  **public** **static** **void** main(String[] args) {  //Phone p = new Phone("Android", "2.5 GZh", 17.5, 5000, 64);  PhoneBuilder builder = **new** PhoneBuilder();  builder.setOs("Android");  Phone p = builder.getPhone();  System.***out***.println(p);  }  } | Here we are setting only OS value at run time are we are getting phone information, now we don’t need to set all the parameter for getting phone information |

Output:

Phone[OS=Android, Processor=null,Screen Size=0.0, Battery=0, Camera=0]

|  |
| --- |
| **package** com.design.pattern.builder;  **public** **class** Test {  **public** **static** **void** main(String[] args) {  //Phone p = new Phone("Android", "2.5 GZh", 17.5, 5000, 64);  PhoneBuilder builder = **new** PhoneBuilder();  Phone onlyOs = builder.setOs("Android").getPhone();  Phone os\_Battary = builder.setOs("Android").setBattery(5000).getPhone();  Phone os\_Battary\_camera = builder.setOs("Android").setBattery(5000).setCamera(64).getPhone();  Phone os\_Battary\_camera\_ScreenSize = builder.setOs("Android").setBattery(5000).setScreenSize(17.5).getPhone();  Phone os\_Battary\_camera\_ScreenSize\_Processor = builder.setOs("Android").setBattery(5000).setScreenSize(17.5)  .setProcessor("2.5GHz").getPhone();  System.***out***.println("onlyOs->" + onlyOs);  System.***out***.println("os\_Battary->" + os\_Battary);  System.***out***.println("os\_Battary\_camera->" + os\_Battary\_camera);  System.***out***.println("os\_Battary\_camera\_ScreenSize->" + os\_Battary\_camera\_ScreenSize);  System.***out***.println("os\_Battary\_camera\_ScreenSize\_Processor->" + os\_Battary\_camera\_ScreenSize\_Processor);  }  } |
| Output:  onlyOs->Phone[OS=Android, Processor=null,Screen Size=0.0, Battery=0, Camera=0]  os\_Battary->Phone[OS=Android, Processor=null,Screen Size=0.0, Battery=5000, Camera=0]  os\_Battary\_camera->Phone[OS=Android, Processor=null,Screen Size=0.0, Battery=5000, Camera=64]  os\_Battary\_camera\_ScreenSize->Phone[OS=Android, Processor=null,Screen Size=17.5, Battery=5000, Camera=64]  os\_Battary\_camera\_ScreenSize\_Processor->Phone[OS=Android, Processor=2.5GHz,Screen Size=17.5, Battery=5000, Camera=64] |
| So we can see that we can create different objects based on the set parameters, we can set the mandatory parameters only and need not to set optional parameter. One more important thing is that we don’t need to follow the sequence or order of the parameter to pass. |



In the above example all the variables was optional: Now let’s take an example where three fields are mandatory and two fields are optional.

|  |
| --- |
| **package** com.design.pattern.builder;  **public** **class** Computer {  // Mandatory Fields  **private** String RAM;  **private** String HDD;  **private** String CPU;  // Optional Fields  **private** **boolean** isGraphicCardEnabled;  **private** **boolean** isBluetoothEnabled;  // public Computer(Builder builder) { Step-8 written below  **private** Computer(Builder builder) {  **this**.RAM = builder.RAM;  **this**.CPU = builder.CPU;  **this**.HDD = builder.HDD;  **this**.isBluetoothEnabled = builder.isBluetoothEnabled;  **this**.isGraphicCardEnabled = builder.isGraphicCardEnabled;  }  // Step-1: Create getter() methods for concrete class variable  **public** String getRAM() {  **return** RAM;  }  **public** String getHDD() {  **return** HDD;  }  **public** String getCPU() {  **return** CPU;  }  **public** **boolean** isGraphicCardEnabled() {  **return** isGraphicCardEnabled;  }  **public** **boolean** isBluetoothEnabled() {  **return** isBluetoothEnabled;  }  **// Step-2: Create a Builder class which will be static class and it has to be**  **// nested class,**  **// so that it can use private constructor of computer class so that using this**  **// constructor we can create object of Computer class.**  **// i.e. in step-2 Create public static inner Builder class**  **public** **static** **class** Builder {  **// Step-3: Since we are using Builder class to create object of computer so all**  **// the variable of Computer should be available inside Builder**  **// i.e. copy all required variable from concrete Computer class to Builder class**  // Mandatory Fields  **private** String RAM;  **private** String HDD;  **private** String CPU;  // Optional Fields  **private** **boolean** isGraphicCardEnabled;  **private** **boolean** isBluetoothEnabled;  **// Step-4: Create builder constructor for Mandatory parameters**  **public** Builder(String rAM, String hDD, String cPU) {  **super**();  RAM = rAM;  HDD = hDD;  CPU = cPU;  }  **// Step-5: Create setter() method only for optional parameters**  **// public void setGraphicCardEnabled(boolean isGraphicCardEnabled)**  **public** Builder setGraphicCardEnabled(**boolean** isGraphicCardEnabled) {  **this**.isGraphicCardEnabled = isGraphicCardEnabled;  **return** **this**;  }  // public void setBluetoothEnabled(boolean isBluetoothEnabled)  **public** Builder setBluetoothEnabled(**boolean** isBluetoothEnabled) {  **this**.isBluetoothEnabled = isBluetoothEnabled;  **return** **this**;  }  **// Step-6: Change the setter()methods to return Builder instance so replace void**  **// from**  **// Builder**  **// Step-7: Create a method which can be used to get the instance of Computer**  **// i.e. Create build() method in Builder class to return Computer Type instance**  **public** Computer build() {  **return** **new** Computer(**this**); // Here we will get erroer will ask for private constructor  }  **// step-8: Create private constructor of Computer class with [Builder builder]**  **// argument inside Computer class not inside Builder class**  }  @Override  **public** String toString() {  // **TODO** Auto-generated method stub  **return** "Computer[ HDD=" + getHDD() + ", RAM=" + getRAM() + ", CPU=" + getCPU() + ", Graphics="  + isGraphicCardEnabled + ", Bluetooth=" + isBluetoothEnabled() + "]";  }  } |
| **package** com.design.pattern.builder;  **import** com.design.pattern.builder.Computer.Builder;  **public** **class** TestComputerBuilder {  **public** **static** **void** main(String[] args) {  Computer comp = **new** Computer.Builder("4BG", "500GB", "3.5GZh").build();  System.***out***.println(comp);  // The above line is creating object with mandatory field  // Now let's create another builder object with optional field  Computer comp1 = **new** Computer.Builder("4BG", "500GB", "3.5GZh").setBluetoothEnabled(**true**).build();  Computer comp3 = **new** Computer.Builder("4BG", "500GB", "3.5GZh").setBluetoothEnabled(**true**).setGraphicCardEnabled(**true**).build();  System.***out***.println(comp1);  System.***out***.println(comp3);  }  } |

Sequence of execution:

1. **Computer.Builder("4BG", "500GB", "3.5GZh"):** It will call the Builder class constructer and set these values after that when we put (.build()) method
2. Computer.Builder("4BG", "500GB", "3.5GZh").**build(): Once Builder class constructer get executed and set the parameter then build() method will be invoked and inside this it invoke Computer class private constructor just to set Builder parameter values to the Computer class parameter values.**
3. **Finally toString() method will be called to display the set values of Computer parameters.**
4. [**https://www.youtube.com/watch?v=D5NK5qMM14g**](https://www.youtube.com/watch?v=D5NK5qMM14g)

Conclusion:

1. Useful in creating objects with many optional parameters. If we have only one or two optional parameter then do not required to use Builder patter.
2. Drawbacks:

* Redundant Code
* No code reusability
* Any changes in the concrete class requires same changes in the Builder class as well and if it is mandatory parameter then requires same changes in client as well.

**Prototype Pattern**

This is the part of Object creational pattern. As previously we have learnt different types of design pattern for objection creational we can anyone of them, but as we know that when we create object then it occupies lot of memories and takes lot of time for creation. Let’s say if we are trying to create object with the help of database where we are loading some data from the data base and those object gets created from the loading of data then it will take some time to create an object.

Now let’s say we have to create second object then why to create in similar ways because again it will communicate with the database and will take time to create object. Can we create the second object from already created object i.e. something like can we copy or clone the first object and create the second object?

So answer is yes, we can clone the object and create the same object and this is called prototype pattern. So here instead of getting same new object using new operator, we can clone the object and create another similar object. Because if we clone the object then we don’t need to interact with database again for creating same object.

The prototype pattern is a creational design pattern. Prototype patterns is required, when object creation is time consuming, and costly operation, so we create object with existing object itself. One of the best available way to create object from existing objects are **clone () method**.

In this design pattern, an instance of actual object (i.e. prototype) is created on starting, and thereafter whenever a new instance is required, this prototype is cloned to have another instance. The main **advantage** of this pattern is to have minimal instance creation process which is much costly than cloning process.

Please ensure that you want to [deep clone or shallow clone](https://howtodoinjava.com/java/cloning/a-guide-to-object-cloning-in-java/) your prototype because both will have different behavior on runtime. If deep copy is needed, you can use a good technique given here [using in memory serialization](https://howtodoinjava.com/java/serialization/how-to-do-deep-cloning-using-in-memory-serialization-in-java/).

Let’s take an example of Online Bookshop where all the books are coming from the book where house. Now if we want to open a new similar book shops i.e. clone of this book shop virtually.

1. Create a Book class having two variable book id (bid) and book name (bName), create getter & setter and override toString() method to print book object.
2. Now create BookShop class where we will have all the books. This class will have shop name and list of books

|  |  |  |
| --- | --- | --- |
| **package com.design.pattern.prototype;**  **public class Book {**  **private int bid;**  **private String bname;**  **public int getBid() {**  **return bid;**  **}**  **public void setBid(int bid) {**  **this.bid = bid;**  **}**  **public String getBname() {**  **return bname;**  **}**  **public void setBname(String bname) {**  **this.bname = bname;**  **}**  **@Override**  **public String toString() {**  **return "Book [bid=" + bid + ", bname=" + bname + "]";**  **}**  **}** | **package com.design.pattern.prototype;**  **import java.util.ArrayList;**  **import java.util.List;**  **public class BookShop {**  **private String shopName;**  **private List<Book> books = new ArrayList<Book>();**  **public String getShopName() {**  **return shopName;**  **}**  **public void setShopName(String shopName) {**  **this.shopName = shopName;**  **}**  **public List<Book> getBooks() {**  **return books;**  **}**  **public void setBooks(List<Book> books) {**  **this.books = books;**  **}**  **public void LoadBookData() {**  **for (int i = 0; i <= 5; i++) {**  **Book b = new Book();**  **b.setBid(i);**  **b.setBname("Book-" + i);**  **// books.add(b);**  **getBooks().add(b);**  **}**  **}**  **@Override**  **public String toString() {**  **return "BookShop [shopName=" + shopName + ", books=" + books + "]";**  **}**  **}** | |
| **package** com.design.pattern.prototype;  **public** **class** TestPrototype {  **public** **static** **void** main(String[] args) {  BookShop shop1 = **new** BookShop();  Shop1.setShopName("Pustak Kendra");  Shop1.LoadBookData();  System.***out***.println(shop1);  }  } | | BookShop shop2 = **new** BookShop();  Shop2.setShopName("Novelty");  Shop2.LoadBookData();  System.***out***.println(shop2);  BookShop shop2 = shop1.clone();  Shop2.setShopName("Novelty");  Shop2.LoadBookData();  System.***out***.println(shop1); |

In the previous example now we have created one Book shop which is having 6 books. Here we have created one object of BookShop and set the book name and load the book data from database,

Now we are going to create the prototype of BookShop object (shop)

Approach-1: We can create another object of the BookShop set the ShopName and load the data again from the database, but as we have discussed earlier if we do so then it will take time in creating new object and occupy memory also.

Approach-2: So instead of creating new object we can ask shop1 object itself to provide its own copy i.e. prototype of shop1 and that can be achieved just by cloning of shop1 object instead of creating new object using new operator.

Note: BookShop shop2 = shop1.clone(); when we try to clone() then this method is not visible for shop1 because in the Clonable interface the clone() method is protected type and we will get the below message

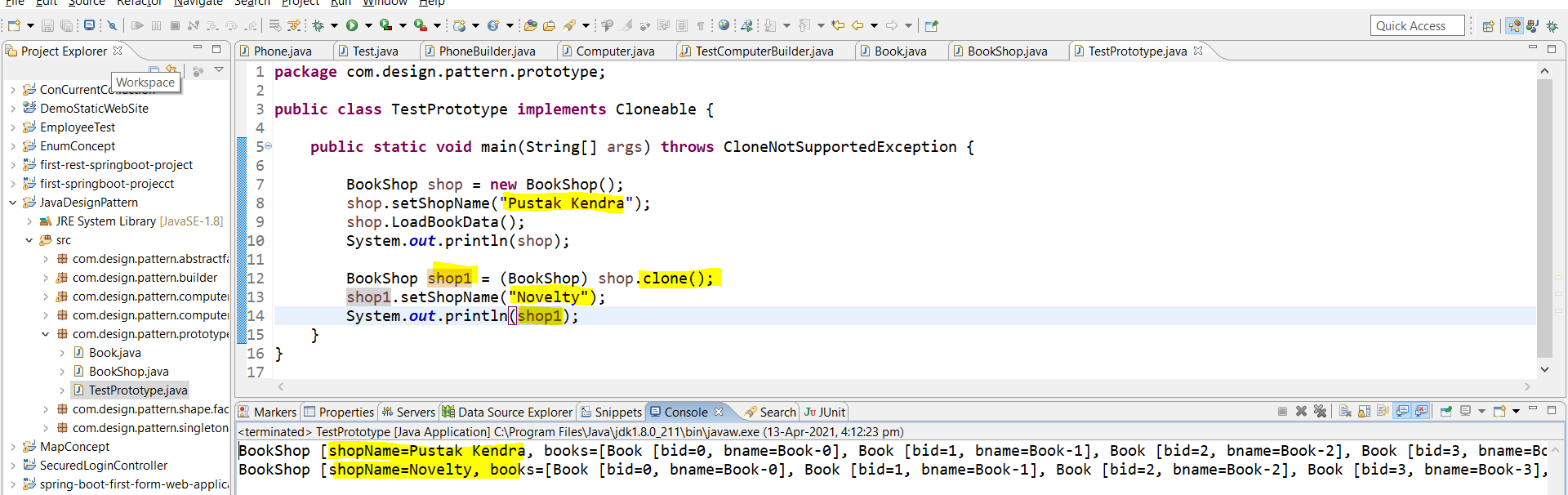
The method clone() from the type Object is not visible

So first of all if want to clone of any Class object then we have to have permission and that we can get just by implementing Clonable interface in the BookShop class. One more thing this Clonable interface does not have any method even clone() itself and it Is called marker interface. We have implements Clonable interface in the BookShop class just to take permission to clone BookShop class object.

To achieve cloning we have to override clone() from object class in the BookShop class. And once we override clone() method inside BookShop class immediately clone() method will start visible from the Object class.

|  |  |
| --- | --- |
|  |  |
| Before overiding clone method inside BookShop class | After overiding clone method inside BookShop class |

Now let’s try to print same object just only buy setting new shop name.



So Yes we have got the same book in the another book shop “Novelty” just by cloning the shop1 (Pustak Kendra)

So now we have the new object without using new operator just simply by cloning the first object shop1.

So we have made changes in BookShop class and main method

|  |  |
| --- | --- |
| **public** **class** BookShop **implements** Cloneable {  @Override  **protected** Object clone() **throws** CloneNotSupportedException {  **return** **super**.clone();  } | BookShop shop = **new** BookShop();  shop.setShopName("Pustak Kendra");  shop.LoadBookData();  shop.getBooks().remove(1);  System.***out***.println(shop);  BookShop shop1 = (BookShop) shop.clone();  shop1.setShopName("Novelty");  System.***out***.println(shop1); |

Now out put :

|  |
| --- |
| BookShop [shopName=**Pustak Kendra**, books=[Book [bid=0, bname=Book-0], Book [bid=2, bname=Book-2], Book [bid=3, bname=Book-3], Book [bid=4, bname=Book-4], Book [bid=5, bname=Book-5]]]  BookShop [shopName=**Novelty**, books=[Book [bid=0, bname=Book-0], Book [bid=2, bname=Book-2], Book [bid=3, bname=Book-3], Book [bid=4, bname=Book-4], Book [bid=5, bname=Book-5]]] |

Note: But the way we have created the clone object is called shallow cloning and if we modify the main object the clone object also will be modified accordingly so this is something we have two references of one object which is not our objective. For example we have removed one book from shop object and the same book got removed from cloned object shop1 also and Our objective is to have a complete different objet after cloning and if we do any modification in the original object then clone object should not be affected. And this process is called deep cloning.

To achieve deep cloning we have to do following modifications in the BookShop class. In deep cloning

So the changed class name and code is given:

|  |  |
| --- | --- |
| @Override  **protected** BookShop clone() **throws** CloneNotSupportedException {    BookShop cloneShop = **new** BookShop();  // Getting books data from original object shop and putting  // into cloneShop object  **for** (Book b : **this**.getBooks()) {  cloneShop.getBooks().add(b);  }  **return** cloneShop;  }  } | BookShop shop1 = shop.clone();  shop.getBooks().remove(1);  shop1.setShopName("Novelty");  System.***out***.println(shop);  System.***out***.println(shop1);  } |

So now the complete code of two changed classes are:

|  |  |
| --- | --- |
| **package** com.design.pattern.prototype;  **import** java.util.ArrayList;  **import** java.util.List;  **public** **class** BookShop **implements** Cloneable {  **private** String shopName;  **private** List<Book> books = **new** ArrayList<Book>();  **public** String getShopName() {  **return** shopName;  }  **public** **void** setShopName(String shopName) {  **this**.shopName = shopName;  }  **public** List<Book> getBooks() {  **return** books;  }  **public** **void** setBooks(List<Book> books) {  **this**.books = books;  }  **public** **void** LoadBookData() {  **for** (**int** i = 0; i <= 5; i++) {  Book b = **new** Book();  b.setBid(i);  b.setBname("Book-" + i);  // books.add(b);  getBooks().add(b);  }  }  @Override  **public** String toString() {  **return** "BookShop [shopName=" + shopName + ", books=" + books + "]";  }  // @Override  // protected Object clone() throws CloneNotSupportedException {  // **TODO** Auto-generated method stub  // return super.clone();  // }  @Override  **protected** BookShop clone() **throws** CloneNotSupportedException {    BookShop cloneShop = **new** BookShop();  // Getting books data from original object shop and putting  // into cloneShop object  **for** (Book b : **this**.getBooks()) {  cloneShop.getBooks().add(b);  }  **return** cloneShop;  }  } | **package** com.design.pattern.prototype;  **public** **class** TestPrototype {  **public** **static** **void** main(String[] args) **throws** CloneNotSupportedException {  BookShop shop = **new** BookShop();  shop.setShopName("Pustak Kendra");  shop.LoadBookData();  BookShop shop1 = shop.clone();  shop.getBooks().remove(1);  shop1.setShopName("Novelty");  System.***out***.println(shop);  System.***out***.println(shop1);  }  }  So here we have done the following changes   1. Implements Clonable interface in the BookShop class. 2. Override clone method from object class and make the return type BookShop class type. 3. In this clone method we have created one object of the BookShop class and copy all the book data from the original BookShop class object created in the main method and return same clone object having book data to the main method shop1 object . 4. Now in the main method we have got the object with list of copied book data from the original data and now here we are just setting the new shop name and then displaying the same object shop1 |

Output:

|  |
| --- |
| BookShop [shopName=Pustak Kendra, books=[Book [bid=0, bname=Book-0], Book [bid=2, bname=Book-2], Book [bid=3, bname=Book-3], Book [bid=4, bname=Book-4], Book [bid=5, bname=Book-5]]]  BookShop [shopName=Novelty, books=[Book [bid=0, bname=Book-0], Book [bid=1, bname=Book-1], Book [bid=2, bname=Book-2], Book [bid=3, bname=Book-3], Book [bid=4, bname=Book-4], Book [bid=5, bname=Book-5]]] |

So now when remove one book (book1) from shop1 then the clone object did not get affected

<https://www.youtube.com/watch?v=nZ76x13Nm8Q>

|  |  |
| --- | --- |
| **package com.design.pattern.builder;**  **import java.util.ArrayList;**  **import java.util.List;**  **public class Employees implements Cloneable{**  **private List<String> empList;**  **public Employees(){**  **empList = new ArrayList<String>();**  **}**  **public Employees(List<String> list){**  **this.empList=list;**  **}**  **public void loadData(){**  **//read all employees from database and put into the list**  **empList.add("Pankaj");**  **empList.add("Raj");**  **empList.add("David");**  **empList.add("Lisa");**  **}**  **public List<String> getEmpList() {**  **return empList;**  **}**  **@Override**  **public Object clone() throws CloneNotSupportedException{**  **List<String> temp = new ArrayList<String>();**  **for(String s : this.getEmpList()){**  **temp.add(s);**  **}**  **return new Employees(temp);**  **}**  **}** | package com.design.pattern.builder;  import java.util.List;  public class PrototypePatternTest {  public static void main(String[] args) throws CloneNotSupportedException {  Employees emps = new Employees();  emps.loadData();  //Use the clone method to get the Employee object  Employees empsNew = (Employees) emps.clone();  Employees empsNew1 = (Employees) emps.clone();  List<String> list = empsNew.getEmpList();  list.add("John");  List<String> list1 = empsNew1.getEmpList();  list1.remove("Pankaj");  System.*out*.println("Original emps List before clone: " + emps.getEmpList());  System.*out*.println("First clone empsNew List and added new element: " + list);  System.*out*.println("Second clone from empsNew1 List and remove first ele: " + list1);  }  } |

Output:

Original emps List before clone: [Pankaj, Raj, David, Lisa]

First clone empsNew List and added new element: [Pankaj, Raj, David, Lisa, John]

Second clone from empsNew1 List and remove first ele: [Raj, David, Lisa]

**Conclusion:**

Prototype design pattern is used when the Object creation is a costly affair and requires a lot of time and resources and you have a similar object already existing.

Prototype pattern provides a mechanism to copy the original object to a new object and then modify it according to our needs. Prototype design pattern uses java cloning to copy the object.

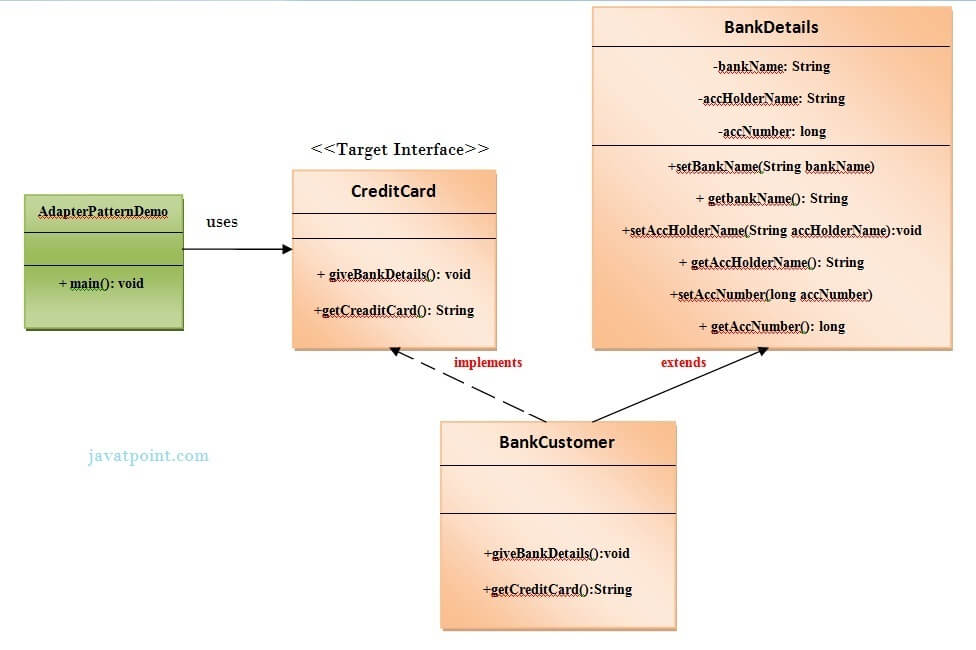
**Adapter Method Pattern**

Adapter design pattern is one of the **structural design patterns** and it’s used between two unrelated interfaces so that both can work together. The object that joins these unrelated interfaces is called an **Adapter**.

An Adapter Pattern says that just "converts the interface of a class into another interface that a client wants".

In other words, to provide the interface according to client requirement while using the services of a class with a different interface.

The Adapter Pattern is also known as **Wrapper**.



<https://www.javatpoint.com/adapter-pattern>

|  |
| --- |
| **package** com.design.pattern.adapterBank;  **public** **interface** CreditCard {  **public** **void** giveBankDetails();  **public** String getCreditCard();  } |

|  |  |
| --- | --- |
| **package** com.design.pattern.adapterBank;  // This is the adapter class.  **public** **class** BankDetails {    **private** String bankName;  **private** String accHolderName;  **private** **long** accNumber;  **public** String getBankName() {  **return** bankName;  }  **public** **void** setBankName(String bankName) {  **this**.bankName = bankName;  }  **public** String getAccHolderName() {  **return** accHolderName;  }  **public** **void** setAccHolderName(String accHolderName) {  **this**.accHolderName = accHolderName;  }  **public** **long** getAccNumber() {  **return** accNumber;  }  **public** **void** setAccNumber(**long** accNumber) {  **this**.accNumber = accNumber;  }  }// End of the BankDetails class. | **package** com.design.pattern.adapterBank;  // This is the adapter class  **import** java.io.BufferedReader;  **import** java.io.InputStreamReader;  **public** **class** BankCustomer **extends** BankDetails **implements** CreditCard {  **public** **void** giveBankDetails() {  **try** {    BufferedReader br = **new** BufferedReader(**new** InputStreamReader(System.***in***));  System.***out***.print("Enter the account holder name :");  String customername = br.readLine();  System.***out***.print("\n");  System.***out***.print("Enter the account number:");  **long** accno = Long.*parseLong*(br.readLine());  System.***out***.print("\n");  System.***out***.print("Enter the bank name :");  String bankname = br.readLine();  setAccHolderName(customername);  setAccNumber(accno);  setBankName(bankname);  } **catch** (Exception e) {  e.printStackTrace();  }  }  @Override  **public** String getCreditCard() {    **long** accno = getAccNumber();  String accholdername = getAccHolderName();  String bname = getBankName();  **return** ("The Account number " + accno + " of " + accholdername + " in " + bname  + "bank is valid and authenticated for issuing the credit card. ");  }  }// End of the BankCustomer class. |
| **package** com.design.pattern.adapterBank;  //This is the client class.  **public** **class** AdapterPatternDemo {  **public** **static** **void** main(String args[]) {  CreditCard targetInterface = **new** BankCustomer();  targetInterface.giveBankDetails();  System.***out***.print(targetInterface.getCreditCard());  }  }// End of the BankCustomer class. | |
| Enter the account holder name :Arun  Enter the account number:001234  Enter the bank name :SBI  The Account number 1234 of Arun in SBIbank is valid and authenticated for issuing the credit card. | |

**Another Example:** [**https://www.journaldev.com/1487/adapter-design-pattern-java**](https://www.journaldev.com/1487/adapter-design-pattern-java)

|  |  |
| --- | --- |
| **package** com.design.pattern.voltSocketAdapter;  **public** **class** Volt {  **private** **int** volts;    **public** Volt(**int** volts) {  **this**.volts = volts;  }  **public** **int** getVolts() {  **return** volts;  }  **public** **void** setVolts(**int** volts) {  **this**.volts = volts;  }    } | **package** com.design.pattern.voltSocketAdapter;  **public** **class** Socket {  **public** Volt getVolt(){  **return** **new** Volt(120);  }  }  **package** com.design.pattern.voltSocketAdapter;  //Using inheritance for adapter pattern  **public** **class** SocketClassAdapterImpl **extends** Socket **implements** SocketAdapter{  @Override  **public** Volt get120Volt() {  **return** getVolt();  }  @Override  **public** Volt get12Volt() {  Volt v= getVolt();  **return** convertVolt(v,10);  }  @Override  **public** Volt get3Volt() {  Volt v= getVolt();  **return** convertVolt(v,40);  }    **private** Volt convertVolt(Volt v, **int** i) {  **return** **new** Volt(v.getVolts()/i);  }  } |
| **package** com.design.pattern.voltSocketAdapter;  **public** **interface** SocketAdapter {  **public** Volt get120Volt();    **public** Volt get12Volt();    **public** Volt get3Volt();  } |
| **package** com.design.pattern.voltSocketAdapter;  **public** **class** AdapterPatternTest {  **public** **static** **void** main(String[] args) {  *testClassAdapter*();  }  **private** **static** **void** testClassAdapter() {  SocketAdapter sockAdapter = **new** SocketClassAdapterImpl();  Volt v3 = *getVolt*(sockAdapter, 3);  Volt v12 = *getVolt*(sockAdapter, 12);  Volt v120 = *getVolt*(sockAdapter, 120);  System.***out***.println("v3 volts using Class Adapter=" + v3.getVolts());  System.***out***.println("v12 volts using Class Adapter=" + v12.getVolts());  System.***out***.println("v120 volts using Class Adapter=" + v120.getVolts());  }  **private** **static** Volt getVolt(SocketAdapter sockAdapter, **int** i) {  **switch** (i) {  **case** 3:  **return** sockAdapter.get3Volt();  **case** 12:  **return** sockAdapter.get12Volt();  **case** 120:  **return** sockAdapter.get120Volt();  **default**:  **return** sockAdapter.get120Volt();  }  }  }  **output**:  v3 volts using Class Adapter=3  v12 volts using Class Adapter=12  v120 volts using Class Adapter=120 | |

**Composite Design Pattern**

It is the part of Structural Design Pattern. The word Composite means –“Consisting of different part”, So here in case of programming when an object consist of other objects in a tree structure then it is called composite design pattern. When we want to create an object and which represent a tree structure then we should go for composite design pattern.

Composite Design Pattern Compose objects into tree structures to represent whole-part hierarchies and each individual object and composite object (containing more than one object) are treated uniformly.

|  |  |
| --- | --- |
| Tree | Composite design pattern treats each node in two ways:  1) **Composite** – Composite means it can have other objects below it. 2) **Leaf** – leaf means it has no objects below it.  As you can see, there can be many children to a single parent i.e. Composite, but only one parent per child.  So composite can have composite objects and leaf object |

For example let’s assume we have to assemble one computer device, and then we need different component like [Cabinet & Peripheral devices] further inside the Cabinet we can have different component like [HDD, MotherBoard(MB)], Further MB can have component like [RAM,CPU] similarly Peripheral device can have component like [Keyboard, Mouse]

|  |  |
| --- | --- |
|  | Here each node represent an object , Computer(C ), Cabinet , Peripheral & MB represent composite object as it has other object below it  HDD, RAM, CPU, Mouse, Keyboard represent an leaf object as it has no objects below it.  All these objects represents same type i.e. all are electronic device and that is why they all follow same structure and have same type of feature. |

Note: if we can perform some operation on leaf node then the same operation needs to perform on the composite object. For example if we are able to get the price of Keyboard then we also will be able to get the price of peripheral device and in the same way we will be able to get the price of Computer also i.e. both Composite & Leaf object should share some common feature.

|  |  |
| --- | --- |
| **Component interface having shared feature for leaf and composite class**  **package** com.design.pattern.composite;  **public** **interface** Component {  **void** showPrice();  } | **Leaf.java class**  **package** com.design.pattern.composite;  **public** **class** Leaf **implements** Component {  **private** String name;  **private** **int** price;  **public** Leaf(String name, **int** price) {  **super**();  **this**.name = name;  **this**.price = price;  }  @Override  **public** **void** showPrice() {  System.***out***.println(name + ":" + price);  }  } |

|  |  |
| --- | --- |
| **Composite.java class:**  **package** com.design.pattern.composite;  **import** java.util.ArrayList;  **import** java.util.List;  **public** **class** Composite **implements** Component {  **private** String name;  List<Component> components = **new** ArrayList<Component>();  **public** Composite(String name) {  **super**();  **this**.name = name;  }  **public** **void** addComponent(Component comp) {  components.add(comp);  }  @Override  **public** **void** showPrice() {  System.***out***.println(name);  **for** (Component c : components) {  c.showPrice();  }  }  } | **package** com.design.pattern.composite;  **public** **class** TestComposite {  **public** **static** **void** main(String[] args) {  // Leaf component  Component hdd = **new** Leaf("HDD", 5000);  Component mouse = **new** Leaf("MOUSE", 500);  Component keyboard = **new** Leaf("KEYBOARD", 2000);  Component ram = **new** Leaf("RAM", 3000);  Component cpu = **new** Leaf("CPU", 10000);  Component monitor = **new** Leaf("MONITOR", 8000);  // Composite Component  Composite computer = **new** Composite("Computer");  Composite cabinate = **new** Composite("Cabinate");  Composite pheripheral = **new** Composite("Peripheral");  Composite mb = **new** Composite("MotherBoard");  // Now building tree structure by adding leaf node and composite node to  // composite node itself  // Adding Peripheral devices  pheripheral.addComponent(mouse);  pheripheral.addComponent(keyboard);  pheripheral.addComponent(monitor);  // Adding component into Mother board  mb.addComponent(ram);  mb.addComponent(cpu);  // Adding component into cabinet  cabinate.addComponent(hdd);  cabinate.addComponent(mb);  // Adding all the composite component to computer composite object  computer.addComponent(cabinate);  computer.addComponent(pheripheral);  computer.showPrice();  }  } |

Output:

|  |  |
| --- | --- |
| **Computer**  **Cabinet**  **HDD:5000**  **MotherBoard**  **RAM:3000**  **CPU:10000**  **Peripheral**  **MOUSE:500**  **KEYBOARD:2000**  **MONITOR:8000** | So here we have to composite tree structure |

Step-1: Create one Component interface having one common feature (method) shared with Leaf and Composite class

Step-2: Create one Leaf Class and implement Component Interface,

* 1. Create one constructer to assign name and price,
  2. Override price method to print Leaf node name and price

Step-3: Create one Composite class and again implement Component Interface

* 1. Create one constructor to assign Composite object name
  2. Create one addComponent method which will add all the component object created in the main method to List components
  3. Override showPrice() method from where we will display Composite component(object) name and further in the for each loop we will print all the leaf component(object) name and price

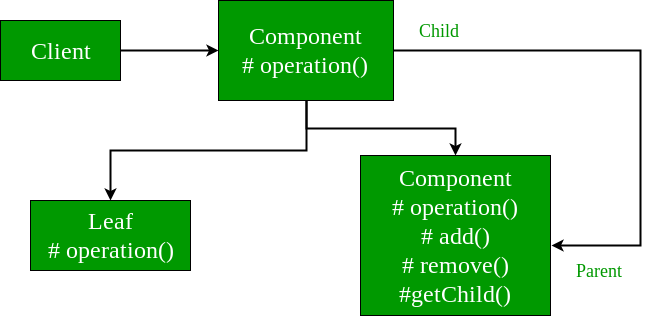
**The Composite Pattern has four participants:**

1. **Component –** Component declares the interface for objects in the composition and for accessing and managing its child components. It also implements default behavior for the interface common to all classes as appropriate.
2. **Leaf –** Leaf defines behavior for primitive objects in the composition. It represents leaf objects in the composition.
3. **Composite –** Composite stores child components and implements child related operations in the component interface.
4. **Client –** Client manipulates the objects in the composition through the component interface.

**Real Life example**

In an organization, It have general managers and under general managers, there can be managers and under managers there can be developers. Now you can set a tree structure and ask each node to perform common operation like getSalary().

|  |
| --- |
| **Step- Create one interface with common feature**  **package** com.design.pattern.composite.salary;  **public** **interface** Employee{  **public** **void** showEmployeeDetails();  } |
| **Create one leaf node Developer: implement Employee interface**  **package** com.design.pattern.composite.salary;  **public** **class** Developer **implements** Employee  {  **private** String name;  **private** **long** empId;  **private** String position;  **public** Developer(**long** empId, String name, String position)  {  **this**.empId = empId;  **this**.name = name;  **this**.position = position;  }  @Override  **public** **void** showEmployeeDetails()  {  System.***out***.println(empId + ":" + name + ":" + position);  }  } |
| **Create one composite Node: implement Employee Interface**  **package** com.design.pattern.composite.salary;  **public** **class** Manager **implements** Employee  {  **private** String name;  **private** **long** empId;  **private** String position;    **public** Manager(**long** empId, String name, String position){  **this**.empId = empId;  **this**.name = name;  **this**.position = position;  }  @Override  **public** **void** showEmployeeDetails() {  System.***out***.println(empId + ":" + name + ":" + position);  }  } |
| **Create one composite Node** CompanyDirectory**: implement Employee Interface**  **package** com.design.pattern.composite.salary;  **import** java.util.ArrayList;  **import** java.util.List;  **public** **class** CompanyDirectory **implements** Employee {  **private** List<Employee> employeeList = **new** ArrayList<Employee>();  @Override  **public** **void** showEmployeeDetails() {  **for** (Employee emp : employeeList) {  emp.showEmployeeDetails();  }  }  **public** **void** addEmployee(Employee emp) {  employeeList.add(emp);  }  **public** **void** removeEmployee(Employee emp) {  employeeList.remove(emp);  }  } |
| **package** com.design.pattern.composite.salary;  **public** **class** Company  {  **public** **static** **void** main (String[] args)  {  Developer dev1 = **new** Developer(100, "Lokesh Sharma", "Pro Developer");  Developer dev2 = **new** Developer(101, "Vinay Sharma", "Developer");  CompanyDirectory engDirectory = **new** CompanyDirectory();  engDirectory.addEmployee(dev1);  engDirectory.addEmployee(dev2);    Manager man1 = **new** Manager(200, "Kushagra Garg", "SEO Manager");  Manager man2 = **new** Manager(201, "Vikram Sharma ", "Kushagra's Manager");    CompanyDirectory accDirectory = **new** CompanyDirectory();  accDirectory.addEmployee(man1);  accDirectory.addEmployee(man2);    CompanyDirectory directory = **new** CompanyDirectory();  directory.addEmployee(engDirectory);  directory.addEmployee(accDirectory);  directory.showEmployeeDetails();  }  } |
| **Output:**  100:Lokesh Sharma:Pro Developer  101:Vinay Sharma:Developer  200:Kushagra Garg:SEO Manager  201:Vikram Sharma :Kushagra's Manager |



**Facade Method Pattern**

This is the part of Structural Design Pattern. The meaning of Façade is face of building the people walking on road can only see this glass face of the building. They do not know anything about it, the wiring, the pipes and other complexities. It hides all the complexities of the building and displays a friendly face.

Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system. This type of design pattern comes under structural pattern as this pattern adds an interface to existing system to hide its complexities.

Facade design pattern is used to help client applications to easily interact with the system.

Simply put, a facade encapsulates a complex subsystem behind a simple interface. It hides much of the complexity and makes the subsystem easy to use.

The facade pattern is appropriate when we have a complex system that we want to expose to clients in a simplified way. Its purpose is to hide internal complexity behind a single interface that appears simple from the outside.

In other words, Facade Pattern describes a higher-level interface that makes the sub-system easier to use. Practically, every Abstract Factory is a type of Facade.

Advantage of Facade Pattern

* It shields the clients from the complexities of the sub-system components.
* It promotes loose coupling between subsystems and its clients.

Usage of Facade Pattern:

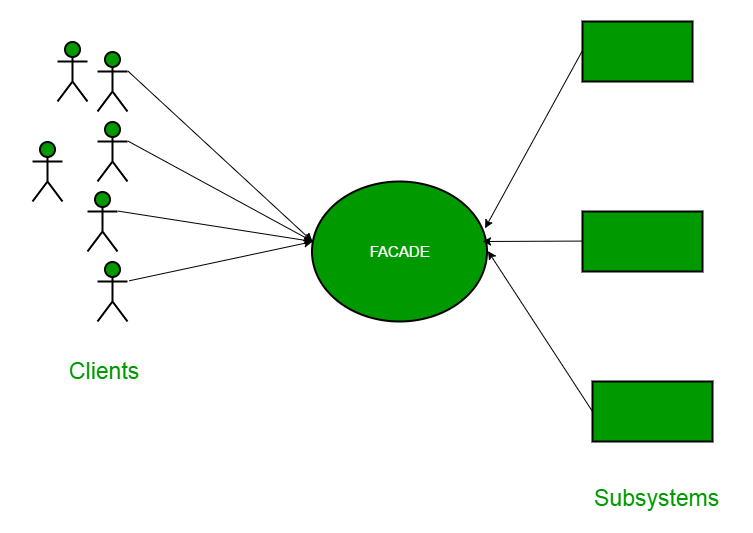
It is used:

* When you want to provide simple interface to a complex sub-system.
* When several dependencies exist between clients and the implementation classes of an abstraction.

**More examples**

In Java, the interface JDBC can be called a facade because, we as users or clients create connection using the “java.sql.Connection” interface, the implementation of which we are not concerned about. The implementation is left to the vendor of driver.

Another good example can be the startup of a computer. When a computer starts up, it involves the work of cpu, memory, hard drive, etc. To make it easy to use for users, we can add a facade which wrap the complexity of the task, and provide one simple interface instead.  
Same goes for the **Facade Design Pattern**. It hides the complexities of the system and provides an interface to the client from where the client can access the system.



**Facade Design Pattern Diagram**

Now Let’s try and understand the facade pattern better using a simple example. Let’s consider a hotel. This hotel has a hotel keeper. There are a lot of restaurants inside hotel e.g. Veg restaurants, Non-Veg restaurants and Veg/Non both restaurants.  
You, as client want access to different menus of different restaurants. You do not know what the different menus they have are. You just have access to hotel keeper who knows his hotel well. Whichever menu you want, you tell the hotel keeper and he takes it out of from the respective restaurants and hands it over to you. Here, the hotel keeper acts as the **facade**, as he hides the complexities of the system hotel.  
Let’s see how it works:

**Interface of Hotel**

|  |
| --- |
| **package** com.design.pattern.facade.hotelKeeper;  **public** **interface** Hotel{  **public** **void** getMenus();  } |

The hotel interface only returns Menus.  
Similarly, the Restaurant are of three types and can implement the hotel interface. Let’s have a look at the code for one of the Restaurants.

NonVegRestaurant.java

|  |
| --- |
| **package** com.design.pattern.facade.hotelKeeper;  **public** **class** NonVegRestaurant **implements** Hotel  {  **public** Menus getMenus()  {  NonVegMenu nv = **new** NonVegMenu();  **return** nv;  }  } |

VegRestaurant.java

|  |
| --- |
| **package** structural.facade;  **public** **class** VegRestaurant **implements** Hotel{  **public** **void** getMenus()  {  VegMenu v = **new** VegMenu();  **return** v;  }  } |

VegNonBothRestaurant.java

|  |
| --- |
| **package** com.design.pattern.facade.hotelKeeper;  **public** **class** VegNonBothRestaurant **implements** Hotel{  **public** Menus getMenus(){  Both b = **new** Both();  **return** b;  }  } |

Now let’s consider the facade,

HotelKeeper.java

|  |
| --- |
| **package** com.design.pattern.facade.hotelKeeper;  **public** **class** HotelKeeper{  **public** VegMenu getVegMenu(){  VegRestaurant v = **new** VegRestaurant();  VegMenu vegMenu = (VegMenu)v.getMenus();  **return** vegMenu;  }  **public** NonVegMenu getNonVegMenu(){  NonVegRestaurant v = **new** NonVegRestaurant();  NonVegMenu NonvegMenu = (NonVegMenu)v.getMenus();  **return** NonvegMenu;  }  **public** Both getVegNonMenu(){  VegNonBothRestaurant v = **new** VegNonBothRestaurant();  Both bothMenu = (Both)v.getMenus();  **return** bothMenu;  }  } |

|  |
| --- |
|  |

From this, It is clear that the complex implementation will be done by HotelKeeper himself. The client will just access the HotelKeeper and ask for either Veg, NonVeg or VegNon Both Restaurant menu.

**How will the client program access this façade?**

|  |
| --- |
| **package** com.design.pattern.facade.hotelKeeper;  **public** **class** Client{  **public** **static** **void** main (String[] args){  HotelKeeper keeper = **new** HotelKeeper();    VegMenu v = keeper.getVegMenu();  NonVegMenu nv = keeper.getNonVegMenu();  Both = keeper.getVegNonMenu();  }  } |

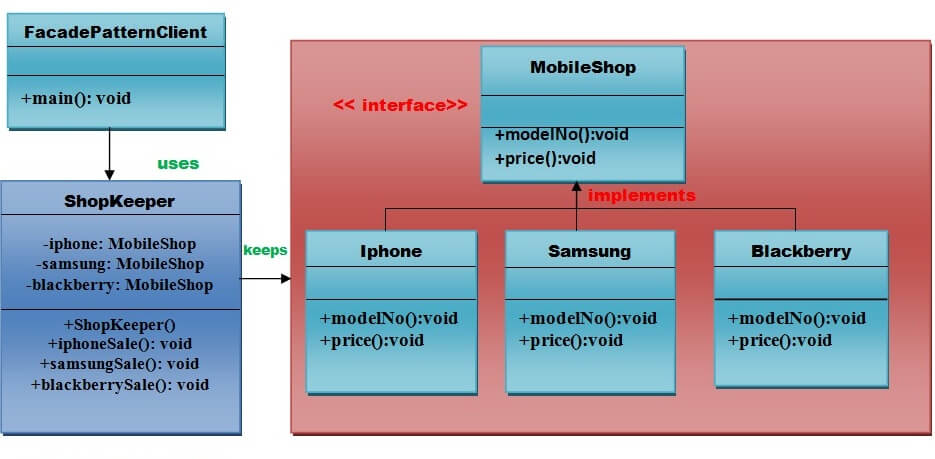
In this way the implementation is sent to the façade. The client is given just one interface and can access only that. This hides all the complexities.

**When Should this pattern be used?**

The facade pattern is appropriate when you have a complex system that you want to expose to clients in a simplified way, or you want to make an external communication layer over an existing system which is incompatible with the system. Facade deals with interfaces, not implementation. Its purpose is to hide internal complexity behind a single interface that appears simple on the outside.

Another Example : MobileShop working as façade class for customer.

|  |
| --- |
| Step 1  Create a MobileShop interface.  *File: MobileShop.java*  **package** com.design.pattern.facade.mobile;  **public** **interface** MobileShop {  **public** **void** modelNo();  **public** **void** price();  } |
| Step 2  Create a Iphone implementation class that will implement Mobileshop interface.  *File: Iphone.java*  **package** com.design.pattern.facade.mobile;  **public** **class** Iphone **implements** MobileShop {  @Override  **public** **void** modelNo() {  System.***out***.println(" Iphone 6 ");  }  @Override  **public** **void** price() {  System.***out***.println(" Rs 65000.00 ");  }  } |
| Step 3  Create a Samsung implementation class that will implement Mobileshop interface.  *File: Samsung.java*  **package** com.design.pattern.facade.mobile;  **public** **class** Samsung **implements** MobileShop {  @Override  **public** **void** modelNo() {  System.***out***.println(" Samsung galaxy tab 3 ");  }  @Override  **public** **void** price() {  System.***out***.println(" Rs 45000.00 ");  }  } |
| Step 4  Create a Blackberry implementation class that will implement Mobileshop interface .  *File: Blackberry.java*  **package** com.design.pattern.facade.mobile;  **public** **class** Blackberry **implements** MobileShop {  @Override  **public** **void** modelNo() {  System.***out***.println(" Blackberry Z10 ");  }  @Override  **public** **void** price() {  System.***out***.println(" Rs 55000.00 ");  }  } |
| Step 5  Create a ShopKeeper concrete class that will use MobileShop interface.  *File: ShopKeeper.java This class work as façade class which will create the object of all the classes and user can just call his method according to his need.*  **package** com.design.pattern.facade.mobile;  **public** **class** ShopKeeper {  **private** MobileShop iphone;  **private** MobileShop samsung;  **private** MobileShop blackberry;  **public** ShopKeeper() {  iphone = **new** Iphone();  samsung = **new** Samsung();  blackberry = **new** Blackberry();  }  **public** **void** iphoneSale() {  iphone.modelNo();  iphone.price();  }  **public** **void** samsungSale() {  samsung.modelNo();  samsung.price();  }  **public** **void** blackberrySale() {  blackberry.modelNo();  blackberry.price();  }  } |
| Step 6  **Now, Creating a client that can purchase the mobiles from MobileShop through ShopKeeper.**  ***File: FacadePatternClient.java***  **package com.design.pattern.facade.mobile;**  **import java.io.BufferedReader;**  **import java.io.IOException;**  **import java.io.InputStreamReader;**    **public class FacadePatternClient {**  **private static int choice;**  **public static void main(String args[]) throws NumberFormatException, IOException{**  **do{**  **System.out.print("========= Mobile Shop ============ \n");**  **System.out.print(" 1. IPHONE. \n");**  **System.out.print(" 2. SAMSUNG. \n");**  **System.out.print(" 3. BLACKBERRY. \n");**  **System.out.print(" 4. Exit. \n");**  **System.out.print("Enter your choice: ");**    **BufferedReader br=new BufferedReader(new InputStreamReader(System.in));**  **choice=Integer.parseInt(br.readLine());**  **ShopKeeper sk=new ShopKeeper();**    **switch (choice) {**  **case 1:**  **{**  **sk.iphoneSale();**  **}**  **break;**  **case 2:**  **{**  **sk.samsungSale();**  **}**  **break;**  **case 3:**  **{**  **sk.blackberrySale();**  **}**  **break;**  **default:**  **{**  **System.out.println("Nothing You purchased");**  **}**  **return;**  **}**    **}while(choice!=4);**  **}**  **}** |



|  |
| --- |
| **Output:**  **========= Mobile Shop ============**  **1. IPHONE.**  **2. SAMSUNG.**  **3. BLACKBERRY.**  **4. Exit.**  **Enter your choice: 1**  **Iphone 6**  **Rs 65000.00**  **========= Mobile Shop ============**  **1. IPHONE.**  **2. SAMSUNG.**  **3. BLACKBERRY.**  **4. Exit.**  **Enter your choice: 2**  **Samsung galaxy tab 3**  **Rs 45000.00**  **========= Mobile Shop ============**  **1. IPHONE.**  **2. SAMSUNG.**  **3. BLACKBERRY.**  **4. Exit.**  **Enter your choice: 3**  **Blackberry Z10**  **Rs 55000.00**  **========= Mobile Shop ============**  **1. IPHONE.**  **2. SAMSUNG.**  **3. BLACKBERRY.**  **4. Exit.**  **Enter your choice: 4**  **Nothing You purchased**  [**https://www.tutorialspoint.com/design\_pattern/facade\_pattern.htm**](https://www.tutorialspoint.com/design_pattern/facade_pattern.htm) |

**Façade system makes a complex system easy to use for client**

**Decorator Design Pattern**

It is a structural pattern that adds additional functionality to an object dynamically at run time. In other words a decorator wraps an object with additional behaviour without affecting other object of the class of same type.

Decorator design pattern is used to add additional features or behaviors to a particular instance of a class, without modifying the other instances of same class.

To extend or modify the behavior of ‘an instance’ at runtime decorator [design pattern](https://javapapers.com/design-patterns/introduction-to-design-patterns/) is used.

The classes’ inputStream -outputStream in java use the decorator pattern to read and write files.

BufferedReader br=new BufferedReader (new FileReader (file));

Using the FileReader (file) method we can read one line a file at a time but when we wrap this new FileReader (file) object inside new BufferedReader (new FileReader (file)) object then we can read a lot more data from the file at once. So this new BufferedReader (new FileReader (file)) object is nothing but act as decorator or wrapper for new FileReader (file) object.

Both these classes (BufferedReader & FileReader) implements Reader interface from the java.io

The Decorator Pattern uses composition instead of inheritance to extend the functionality of an object at runtime. The Decorator Pattern is also known as Wrapper.

**Example:**

|  |  |
| --- | --- |
| https://cdn.journaldev.com/wp-content/uploads/2013/07/inheritance-hierarchy.png | https://cdn.journaldev.com/wp-content/uploads/2013/07/decorator-pattern.png |

1. Created one Interface Car with one abstract method assemble ()
2. First of all created one Basic Car by implementing Car Interface
3. Created one decorator class CarDecorator by implementing Car Interface
4. Extends this CarDecorator class in Sports Car and Luxury Car class
5. Finally these two decorator class Sports Car and Luxury Car will be used as wrapper class and which take basic car class as input to decorate basic car into Sports car and Luxury Car

**Step-1 Component Interface** – The interface or [**abstract class**](https://www.journaldev.com/1582/abstract-class-in-java) defining the methods that will be implemented. In our case Car will be the component interface.

|  |
| --- |
| **package** com.design.pattern.decorator.basicCar;  **public** **interface** Car {  **public** **void** assemble();  } |

**Step-2 Component Implementation** – The basic implementation of the component interface. We can have BasicCar class as our component implementation.

|  |
| --- |
| **package** com.design.pattern.decorator.basicCar;  **public** **class** BasicCar **implements** Car {  @Override  **public** **void** assemble() {  System.***out***.print("Basic Car.");  }  } |

**Decorator** – Decorator class implements the component interface and it has a HAS-A relationship with the component interface. The component variable should be accessible to the child decorator classes, so we will make this variable protected.

|  |
| --- |
| **package** com.design.pattern.decorator.basicCar;  **public** **class** CarDecorator **implements** Car {  **protected** Car car;  **public** CarDecorator(Car c){  **this**.car=c;  }  @Override  **public** **void** assemble() {  **this**.car.assemble();  }  } |

**Concrete Decorators** – Extending the base decorator functionality and modifying the component behavior accordingly. We can have concrete decorator classes as LuxuryCar and SportsCar.

|  |
| --- |
| **package** com.design.pattern.decorator.basicCar;  **public** **class** LuxuryCar **extends** CarDecorator {  **public** LuxuryCar(Car c) {  **super**(c);  }  @Override  **public** **void** assemble(){  **super**.assemble();  System.***out***.print(" Adding features of Luxury Car.");  }  } |
| **package** com.design.pattern.decorator.basicCar;  **public** **class** SportsCar **extends** CarDecorator {  **public** SportsCar(Car c) {  **super**(c);  }  @Override  **public** **void** assemble(){  **super**.assemble();  System.***out***.print(" Adding features of Sports Car.");  }  } |

Theory to be added

Decorator Design Pattern Test Program

|  |
| --- |
| **package** com.design.pattern.decorator.basicCar;  **public** **class** DecoratorPatternTest {  **public** **static** **void** main(String[] args) {  Car sportsCar = **new** SportsCar(**new** BasicCar());  sportsCar.assemble();  System.***out***.println("\n\*\*\*\*\*");    Car sportsLuxuryCar = **new** SportsCar(**new** LuxuryCar(**new** BasicCar()));  sportsLuxuryCar.assemble();  }  }  Output:  Basic Car. Adding features of Sports Car.  \*\*\*\*\*  Basic Car. Adding features of Luxury Car. Adding features of Sports Car. |

**Car sportsCar = new SportsCar (new BasicCar ());**

The above line first call the SportsCar () constructor with parameter BasicCar object and inside this we are calling super (BasicCar). So first it will call BasicCar assemble () method and will display “Basic Car”

After that it will call to SportCar assemble method and display “Adding feature of sports car”

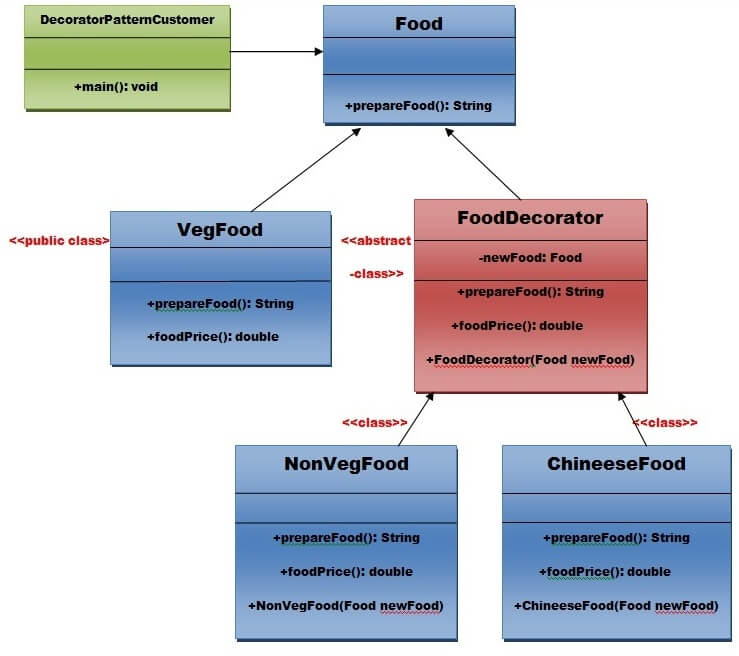
**Note: Car sportsLuxuryCar = new SportsCar (new LuxuryCar (new BasicCar ()));**

Here first we are wrapping Basic car into Luxury Car and further we Luxury car being wrapped into Sports care

### Decorator Design Pattern – Important Points

* Decorator design pattern is helpful in providing runtime modification abilities and hence more flexible. Its easy to maintain and extend when the number of choices are more.
* The disadvantage of decorator design pattern is that it uses a lot of similar kind of objects (decorators).
* Decorator pattern is used a lot in [Java IO](https://www.journaldev.com/942/java-io-tutorial) classes, such as [FileReader, BufferedReader](https://www.journaldev.com/867/java-read-text-file) etc.

**Another Example:** [**https://www.javatpoint.com/decorator-pattern**](https://www.javatpoint.com/decorator-pattern)



**As per the UML diagram.**

* 1. **Created an Interface Food which contains one method prepareFood(): String.**
  2. **Created one class VegFood which implements Food interface and prepare veg food inside prepareFood()**
  3. **Created one class FoodDecorator which will also extends Food Interface and apart from overriding all method present inside Food Interface, created constructor of FoodDecorator class and passed Food as parameter**
  4. **Now this FoodDecorator class will be used as decorator**

**Flyweight Design Pattern**

[**https://www.tutorialspoint.com/design\_pattern/design\_pattern\_interview\_questions.htm**](https://www.tutorialspoint.com/design_pattern/design_pattern_interview_questions.htm)

**Section 7: Template Method Pattern**

<https://www.geeksforgeeks.org/template-method-design-pattern/>