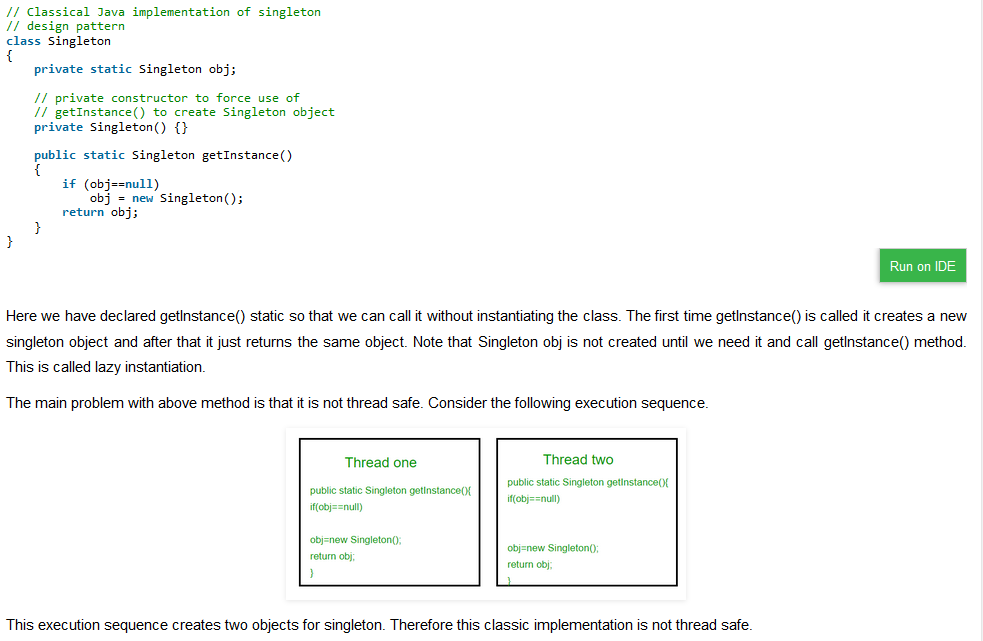
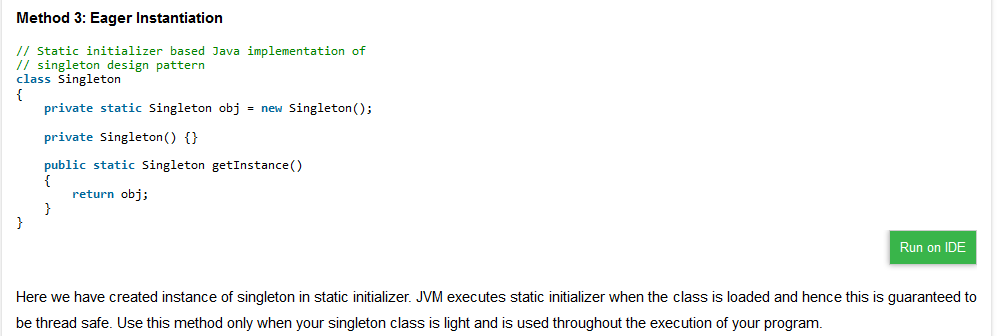
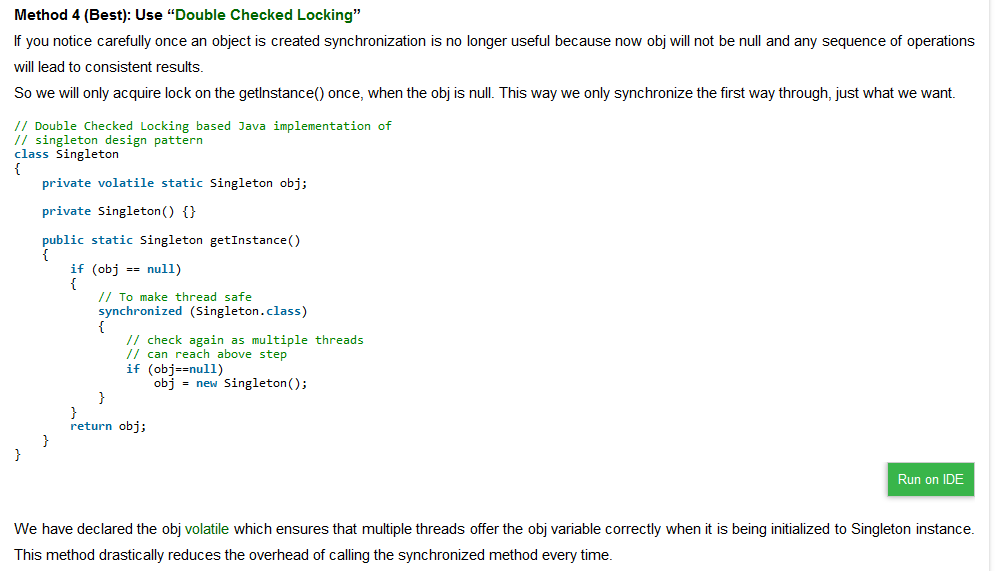


Classic Implementation









Interview Question:

### Which classes are candidates of Singleton? Which kind of class do you make Singleton in Java?

Ans: Any class which you want to be available to whole application and only one instance is viable is candidate of becoming Singleton. One example of this is Runtime class, since on whole java application only one runtime environment can be possible making Runtime Singleton is right decision. Another example is a utility classes like Popup in GUI application, if you want to show popup with message you can have one Popup class on whole GUI application and anytime just get its instance, and call show () with message.

### Is it better to make whole getInstance () method synchronized or just critical section is enough? Which one you will prefer?

Ans: Synchronization is costly and when you apply this on whole method than call to getInstance() will be synchronized and contented. Since synchronization is only needed during initialization on singleton instance, to prevent creating another instance of Singleton, It’s better to only synchronize critical section and not whole method

### What is lazy and early loading of Singleton and how will you implement it?

Ans: There are many ways to implement Singleton like using double checked locking or Singleton class with [static](http://javarevisited.blogspot.sg/2011/11/static-keyword-method-variable-java.html) [final](http://javarevisited.blogspot.sg/2011/12/final-variable-method-class-java.html) instance initialized during class loading. Former is called lazy loading because Singleton instance is created only when client calls getInstance() method while later is called early loading because Singleton instance is created when class is loaded into memory.

### Give me some examples of Singleton pattern from Java Development Kit?

Ans: There are many classes in Java Development Kit which is written using singleton pattern, here are few of them:

1. Java.lang.Runtime with getRuntime() method
2. Java.awt.Toolkit with getDefaultToolkit()
3. Java.awt.Desktop with getDesktop()

### What is double checked locking in Singleton?

Ans: Double checked locking is a technique to prevent creating another instance of Singleton when call to getInstance() method is made in multi-threading environment. In Double checked locking pattern as shown in below example, singleton instance is checked two times before initialization.

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

**private** **static** **volatile** Singleton \_instance; //volatile variable

public static **Singleton** getInstance(){

**if**(**\_INSTANCE** == **null**){

synchronized(**Singleton**.class){

//double checked locking - because second check of Singleton instance with lock

**if**(**\_INSTANCE** == **null**){

**\_INSTANCE** **=** **new** **Singleton**();

}

}

}

**return** **\_INSTANCE**;

}

Without volatile modifier it's possible for another thread in Java to see half initialized state of \_instance variable, but with volatile variable guaranteeing happens-before relationship, all the write will happen on volatile \_instance before any read of \_instance variable.

### How do you prevent for creating another instance of Singleton using clone () method?

Answer: Preferred way is not to implement Cloneable interface as why should one wants to create clone () of Singleton and if you do just throw Exception from clone () method as “Cannot create clone of Singleton class”

**Break by Cloning and fix it**

If a Singleton class implements java.lang.Cloneable interface then invoking clone() method on its single instance creates a duplicate object.

|  |
| --- |
| **public** **class** MyClass **implements** Cloneable{  **private** **static** MyClass instance = **new** MyClass();  **private** **static** MyClass getInstance(){  **return** instance;  }  **public** **static** **void** main(String[] args) **throws**  CloneNotSupportedException {  MyClass obj = MyClass.getInstance();  System.out.println(obj);  System.out.println(obj.clone());  }  } |

**Output**:

com.MyClass@544a5ab2  
com.MyClass@5d888759

Two distinct objects. Singleton gone away.

**Solution:**

Don’t make the class which should be Singleton implement java.lang.Cloneable. If it extends a class which implements Cloneable, then override clone method and throw CloneNotSupportedException from it. This will prevent clone creation.

**public** **class** MyClass **implements** Cloneable{

**private** **static** MyClass instance = **new** MyClass();

**private** **static** MyClass getInstance(){

**return** instance;

}

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

**throw** **new** CloneNotSupportedException();

}

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

CloneNotSupportedException {

MyClass obj = MyClass.getInstance();

System.out.println(obj);

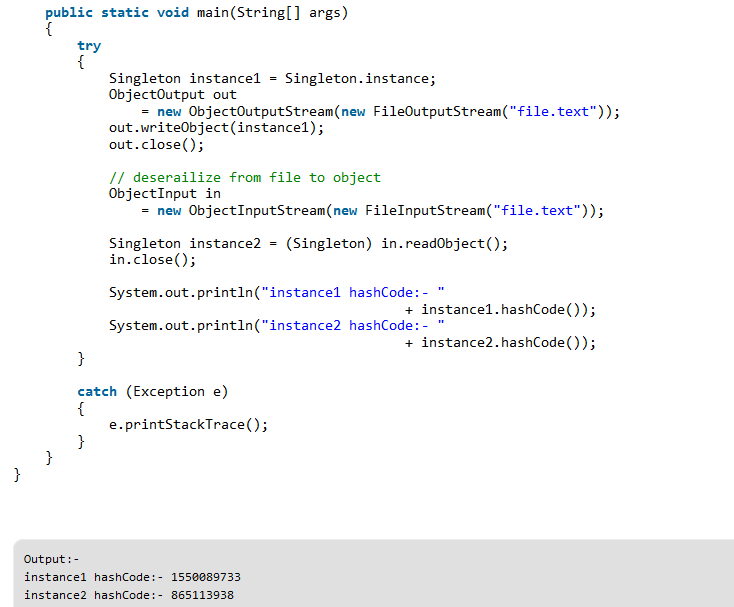
System.out.println(obj.clone()); } }

### 7. How do you prevent for creating another instance of Singleton during serialization?

You can prevent this by using readResolve() method, since during serialization readObject() is used to create instance and it return new instance every time but by using readResolve you can replace it with original Singleton instance. I have shared code on how to do it in my post Enum as Singleton in Java. This is also one of the reason I have said that use Enum to create Singleton because serialization of enum is taken care by JVM and it provides guaranteed of that.

**Deserialization breaks Singleton**

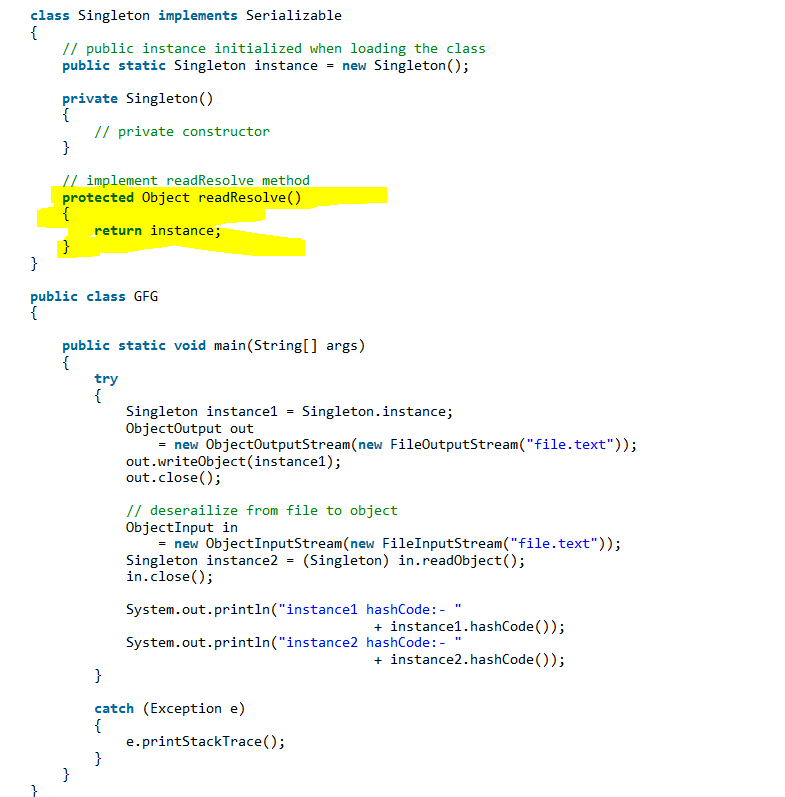
**Deserialization** means bringing a saved object back to life. When a class is deserialized, a **fresh instance** of the class is created and its instance variables are then set to the values which were serialized.

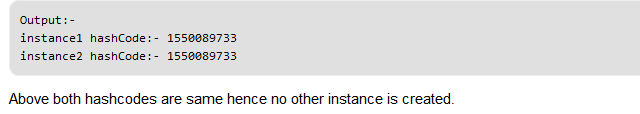


hashCode of both instances is different, hence there are 2 objects of a singleton class. Thus, the class is no more singleton.

**Solution**:

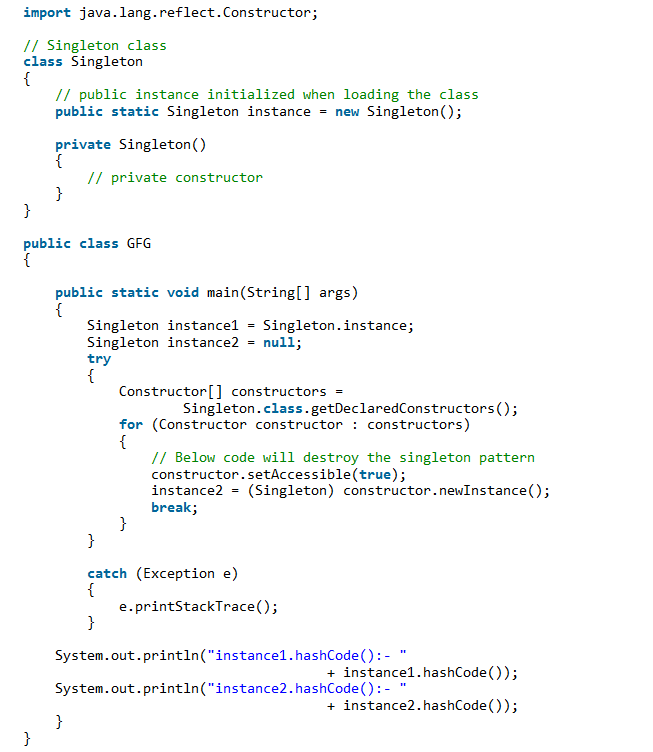
To overcome this issue, we have to implement method readResolve() method.

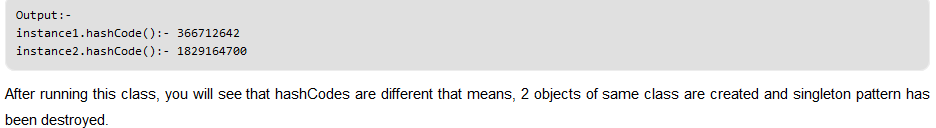


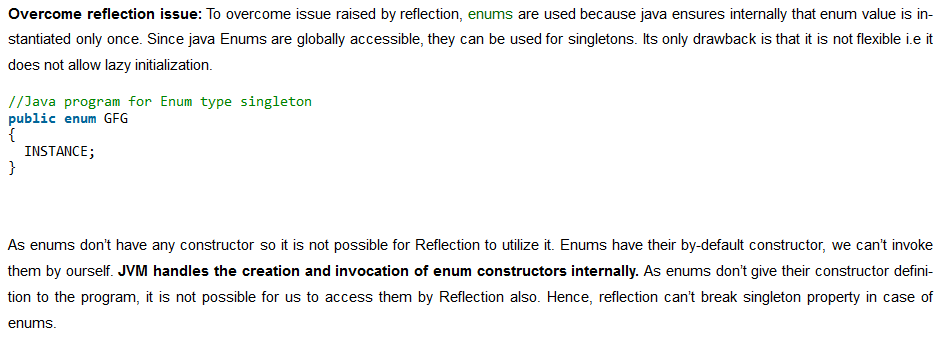


**Reflection can instantiate a Singleton multiple times**

Using java reflection api, we can tweak into a class by getting details like its fields, constructor, invoking its methods etc. Reflection can also be used to create new instance of a class. Obviously, if the class is Singleton, then creating a new instance again breaks its Singleton nature.



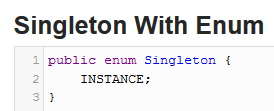


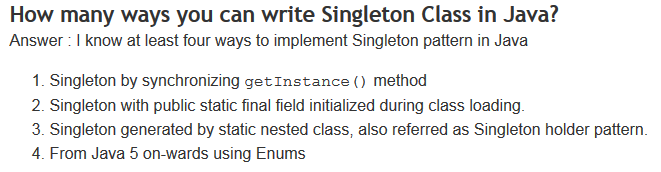


### What are the drawbacks of using singleton design pattern?

**Answer:**  
The major drawbacks of using singleton design pattern are:  
a)Singleton causes code to be tightly coupled. The singleton object is exposed globally and is available to a whole application. Thus, classes using this object become tightly coupled; any change in the global object will impact all other classes using it.  
b)They hide dependencies instead of exposing them.  
c)Singleton Pattern does not support inheritance.  
d)Singleton principle can be violated by techniques such as cloning. If an application is running on multiple JVM’s, then, in this case, Singleton might be broken.

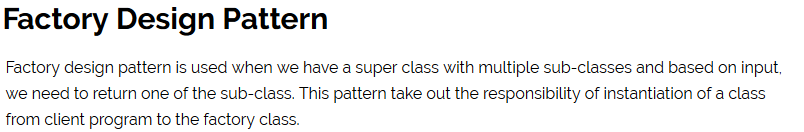
## Making Singletons with Enum in Java

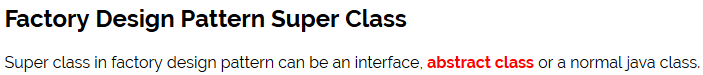




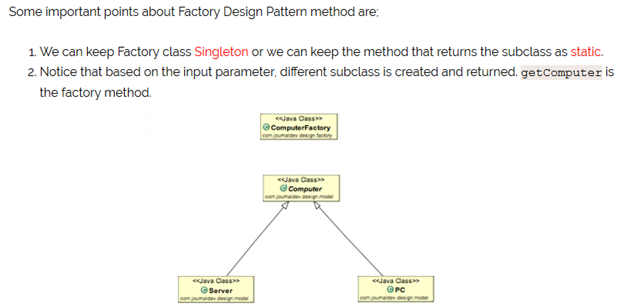
**Factory Pattern:**

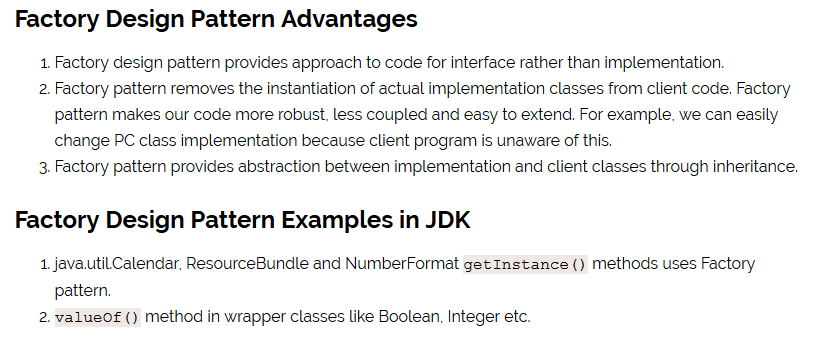
Factory pattern is one of most used design pattern in Java. This type of design pattern comes under **Creational Design Pattern** as this pattern provides one of the best ways to create an object.  
  
          In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common [interface](https://adnjavainterview.blogspot.in/2016/01/interface-and-abstract-related.html). & this pattern provides loose coupling.

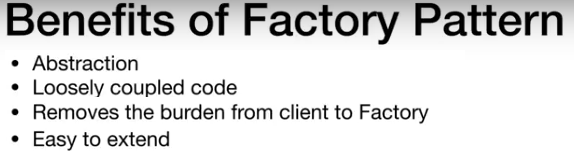










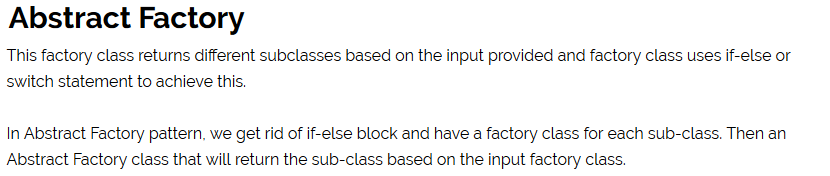


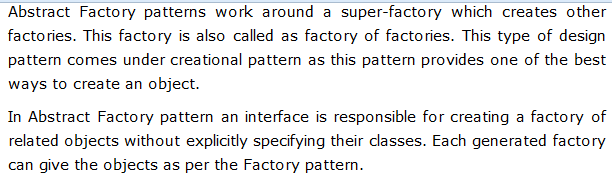
Factory pattern is most suitable where there are some complex object creation steps are involved. To ensure that these steps are centralized and not exposed to composing classes, factory pattern should be used. We can see many examples of factory pattern in JDK itself e.g.

* [java.sql.DriverManager#getConnection()](http://docs.oracle.com/javase/6/docs/api/java/sql/DriverManager.html#getConnection%28java.lang.String%29)
* [java.net.URL#openConnection()](http://docs.oracle.com/javase/6/docs/api/java/net/URL.html#openConnection%28%29)
* [java.lang.Class#newInstance()](http://docs.oracle.com/javase/6/docs/api/java/lang/Class.html#newInstance%28%29)
* [java.lang.Class#forName()](http://docs.oracle.com/javase/6/docs/api/java/lang/Class.html#forName%28java.lang.String%29)

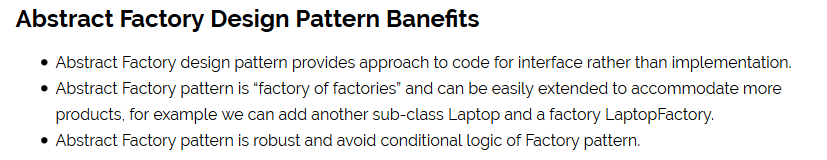
**Abstract Factory Pattern:**

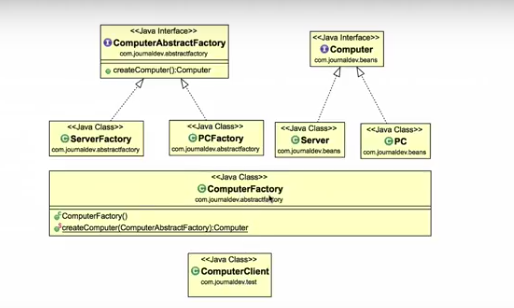
In abstract factory for every concrete class we create a factory class.



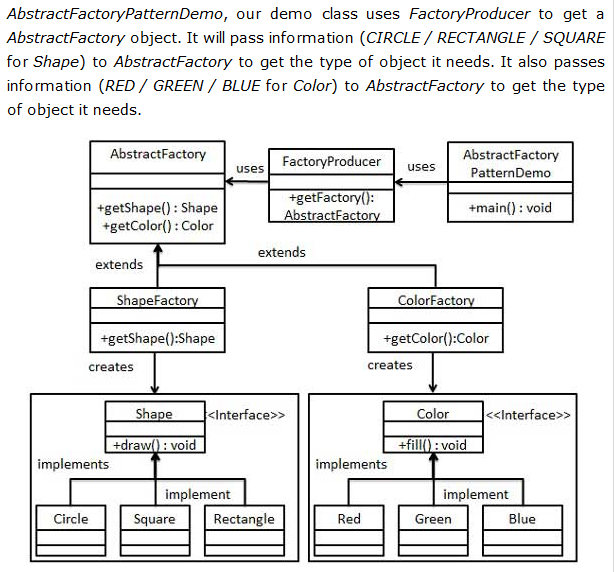




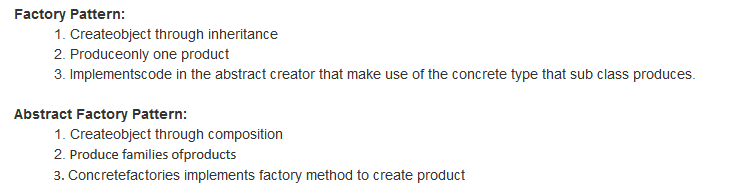
**Ex1:**

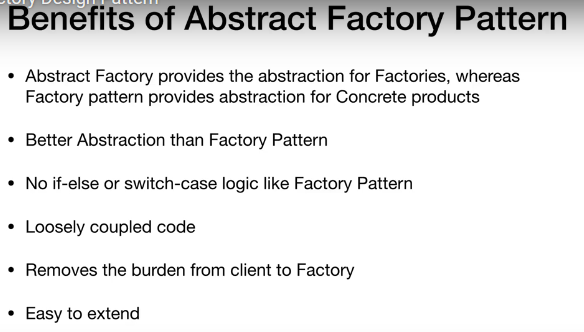


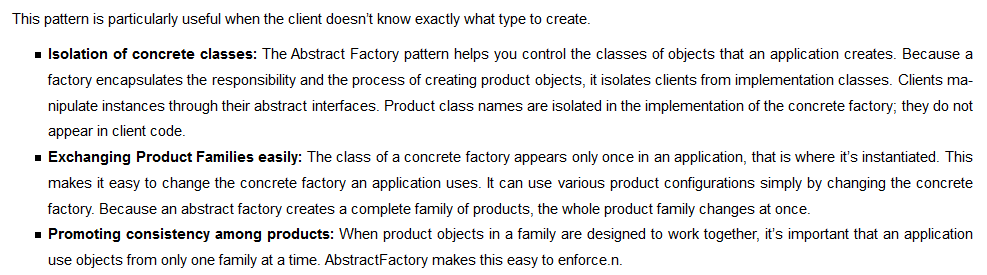
**Ex2:**

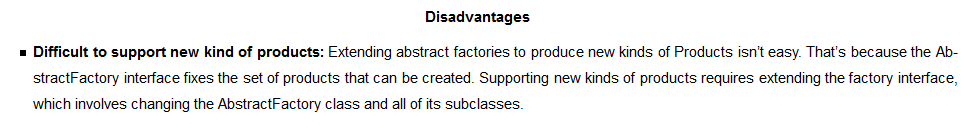




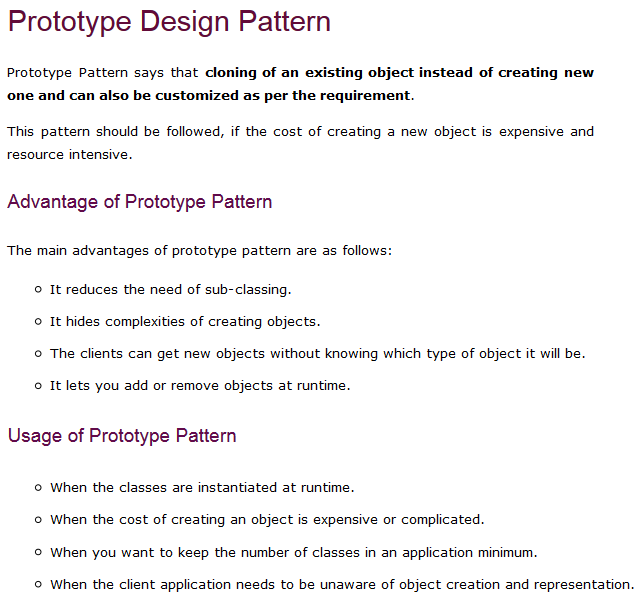


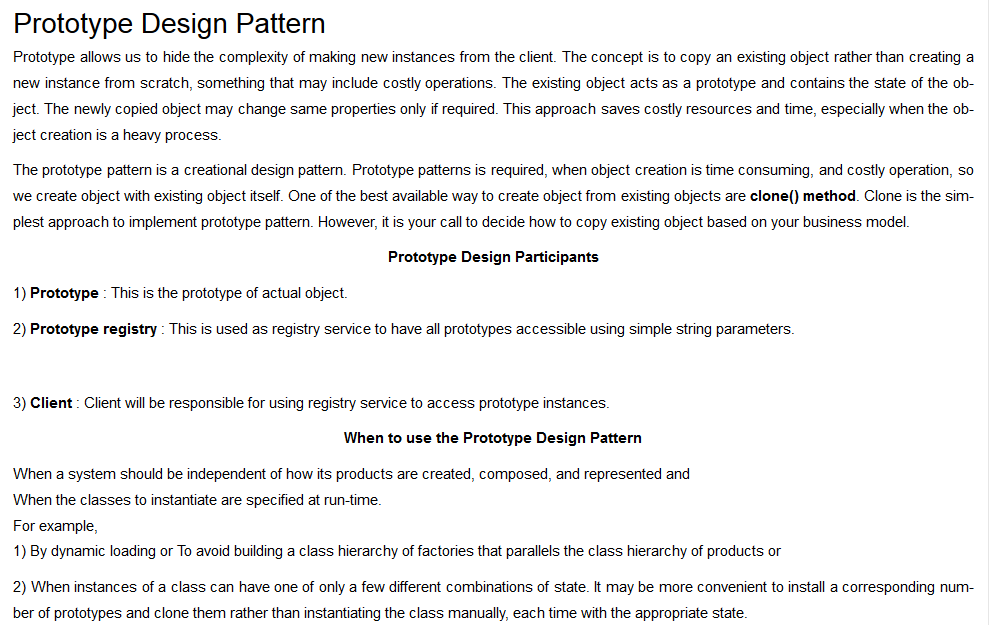






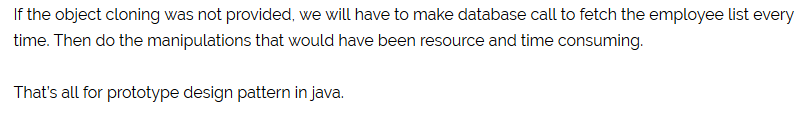
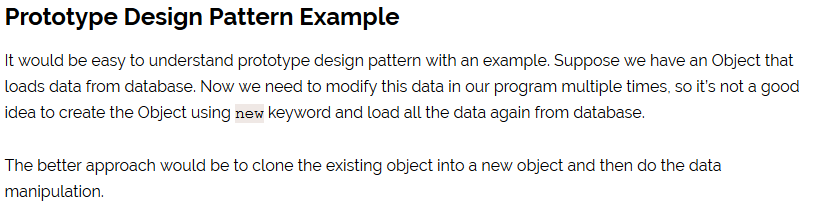
**Prototype design pattern:**

****

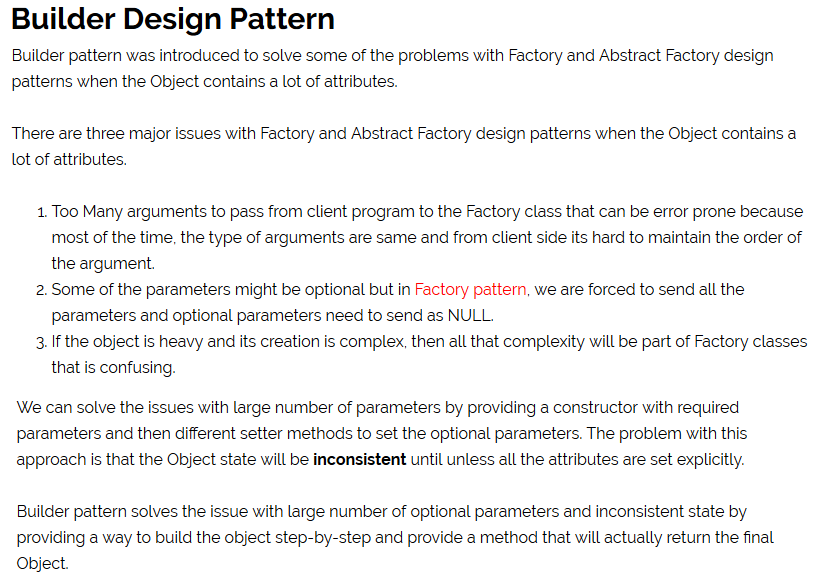


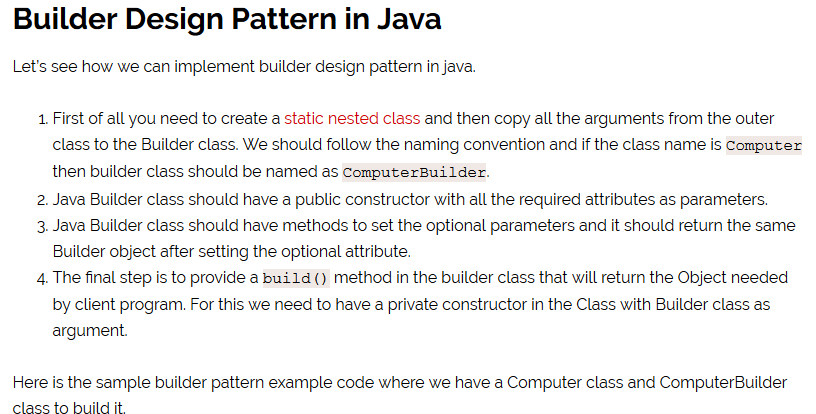


**Example:**



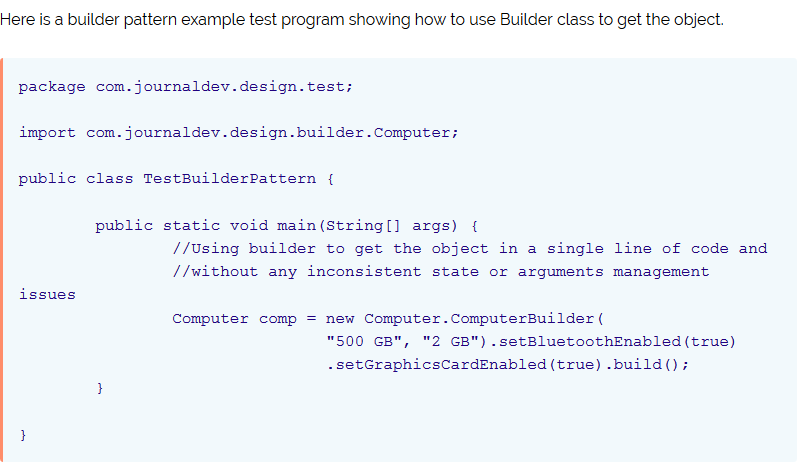
**Builder Design Pattern:**

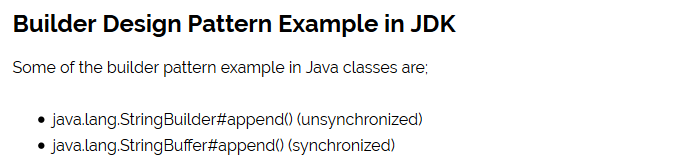


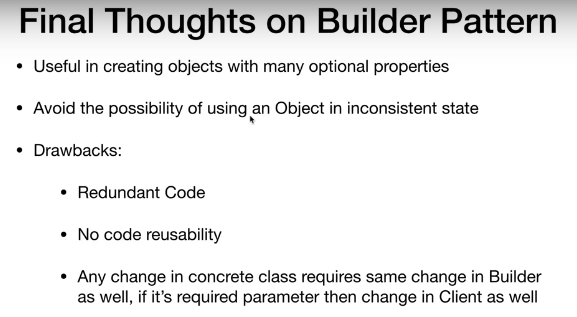




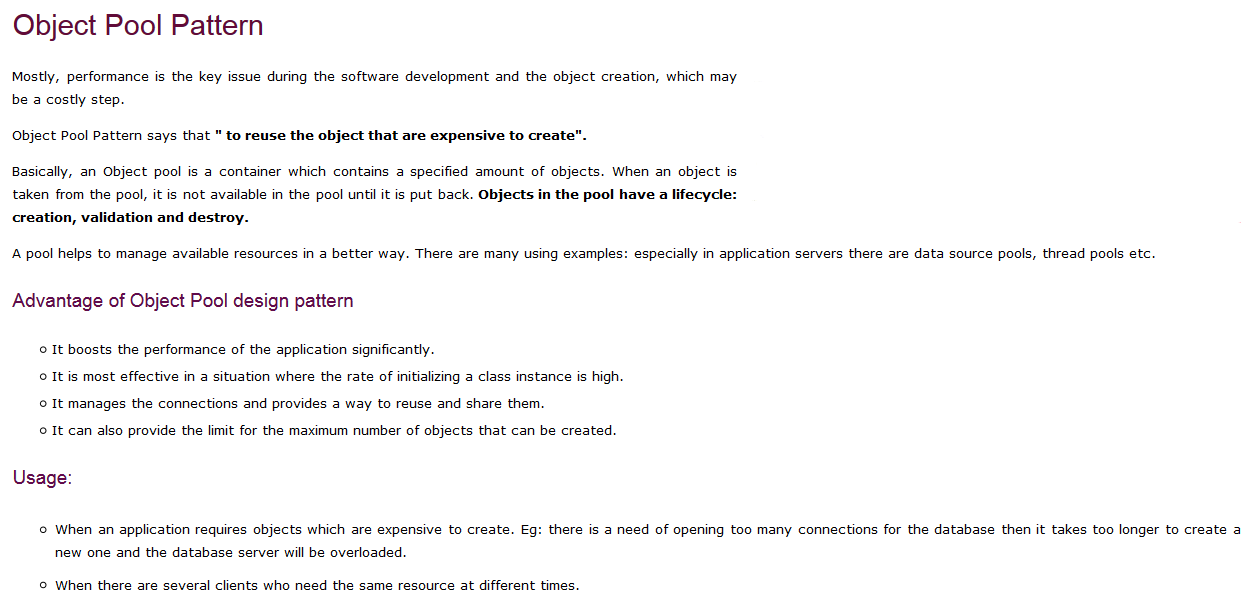
Notice that Computer class has only getter methods and no public constructor. So the only way to get a Computer object is through the ComputerBuilder class.

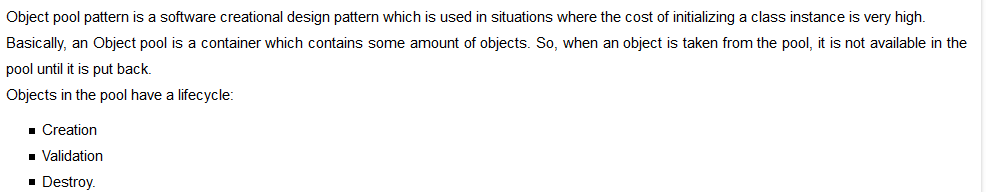


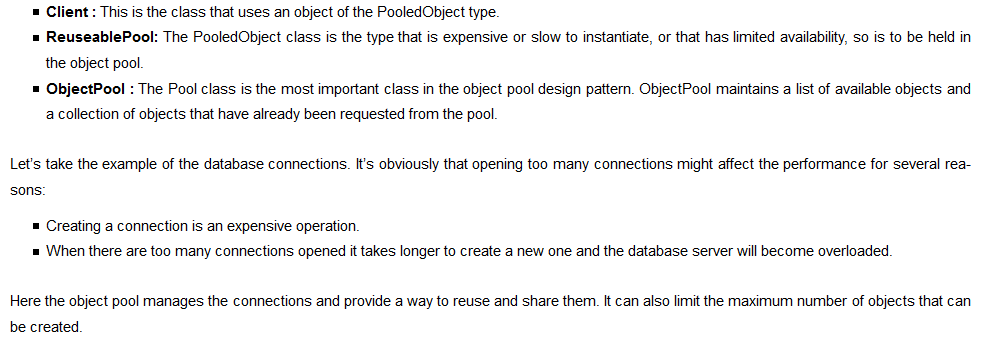


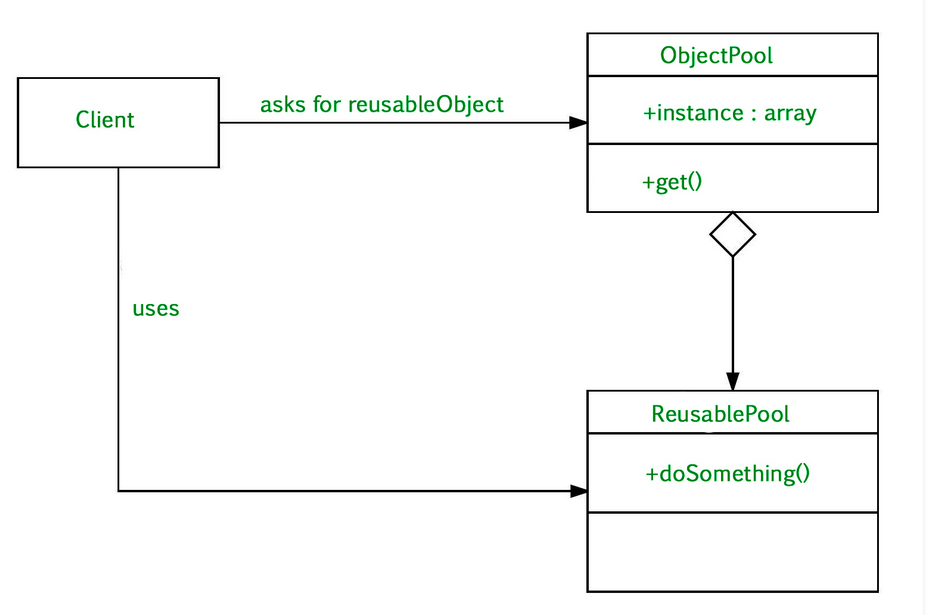


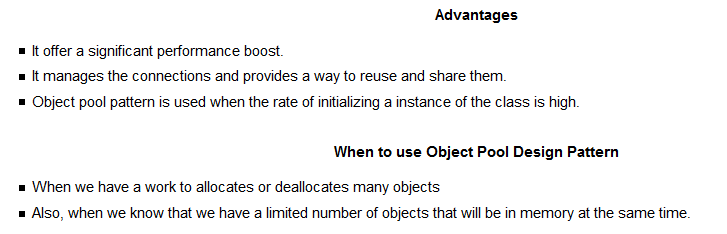
**Object Pool Pattern**

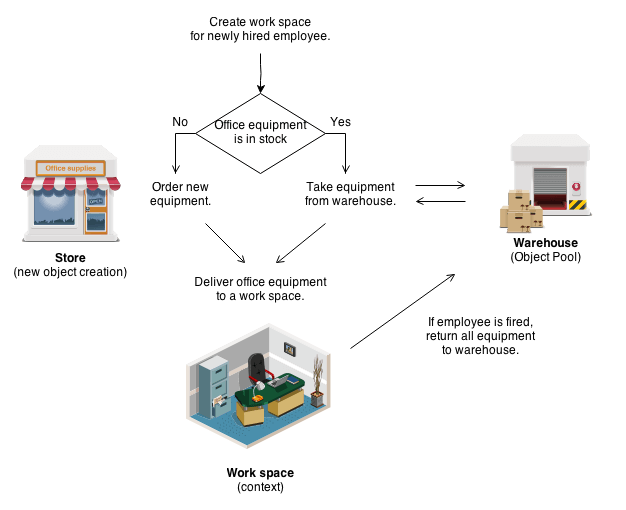






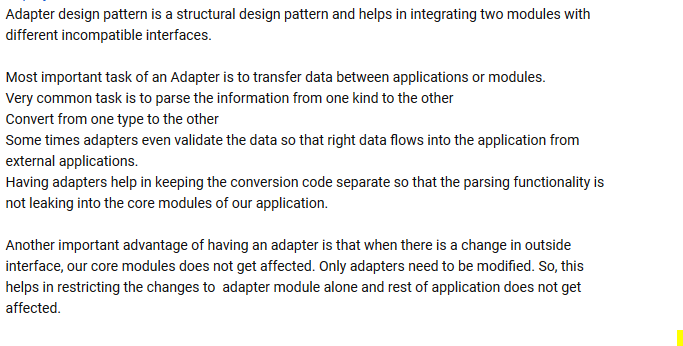


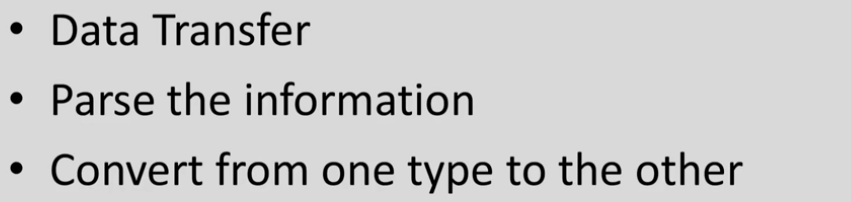
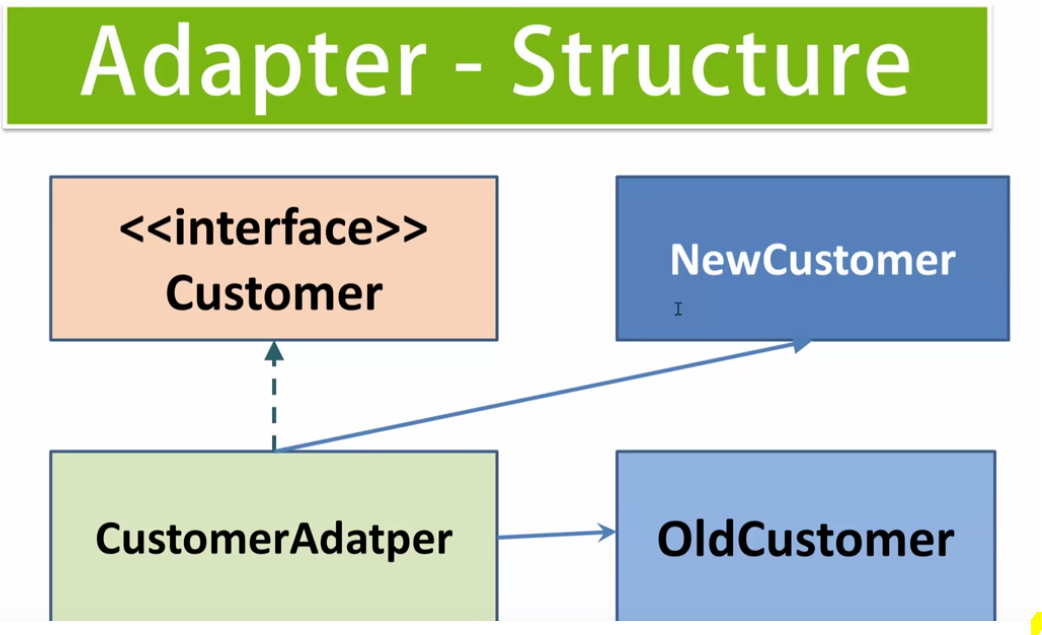
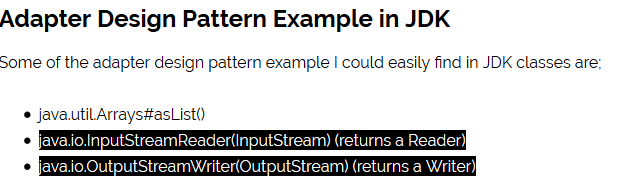


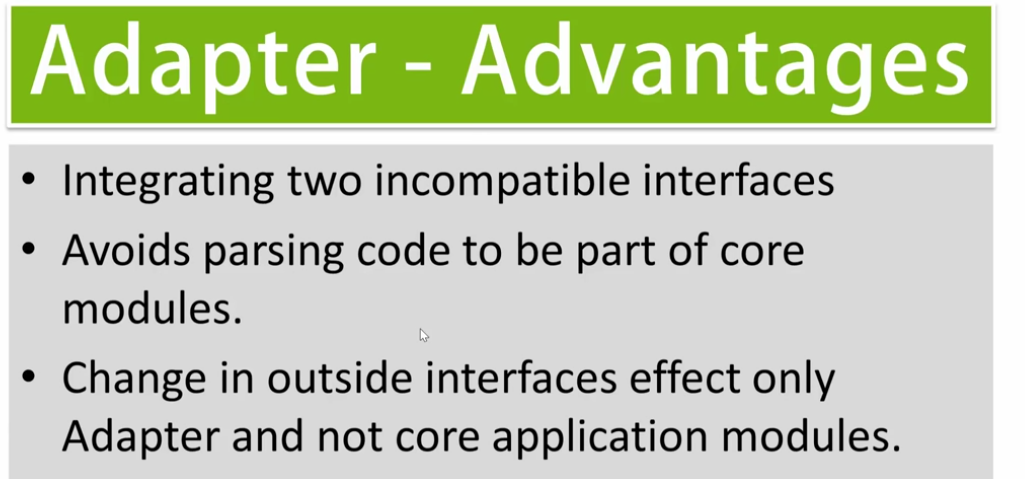
Ex: 

**Structural Design Pattern:**

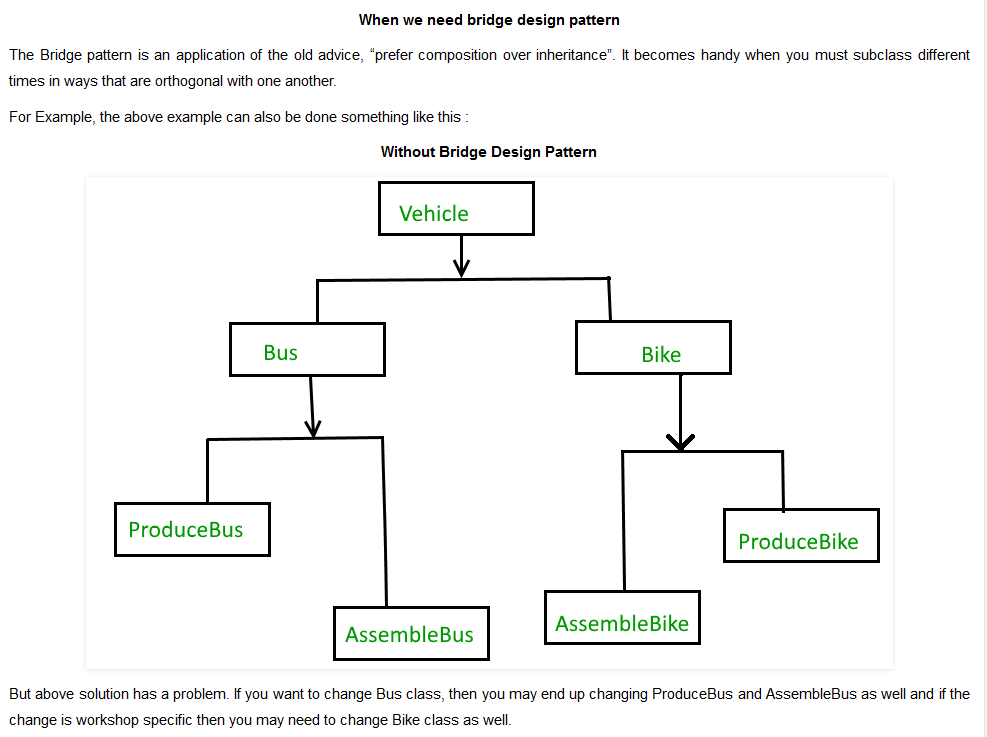
**Adapters Pattern:**

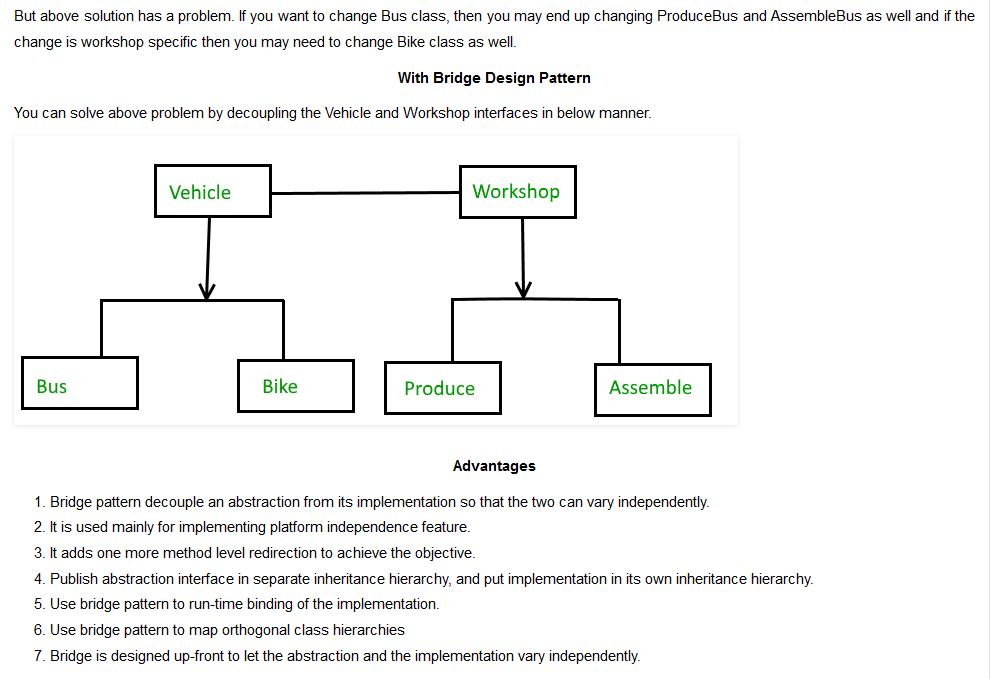




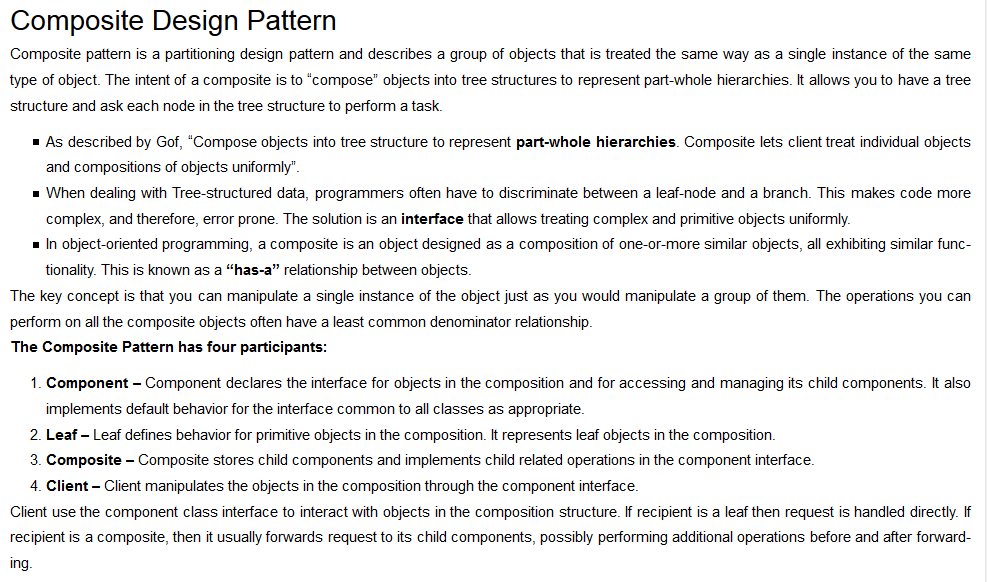


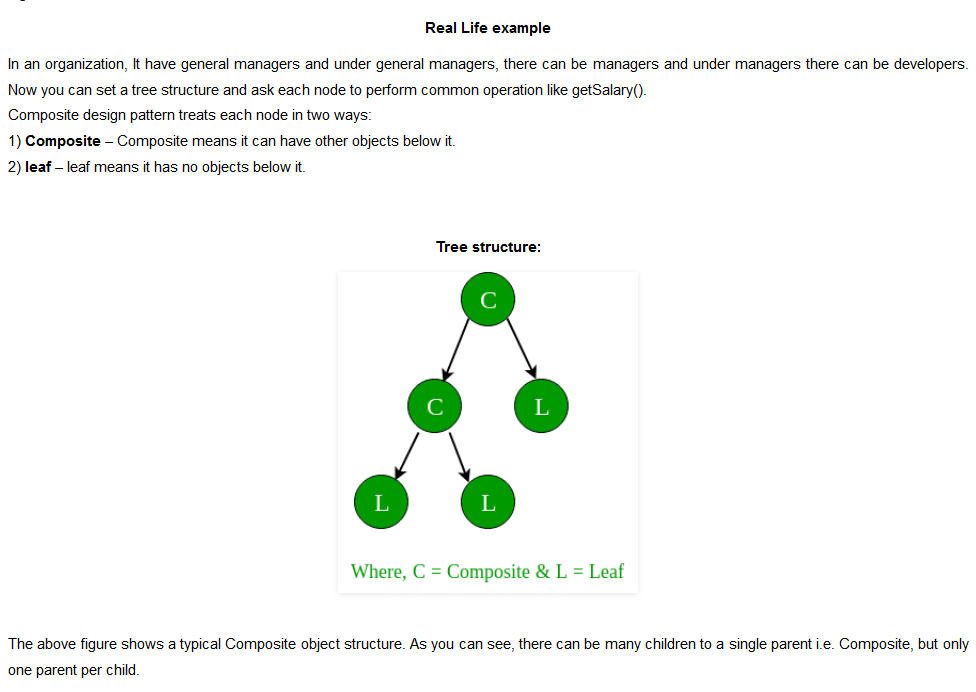
**Bridge Design pattern:**

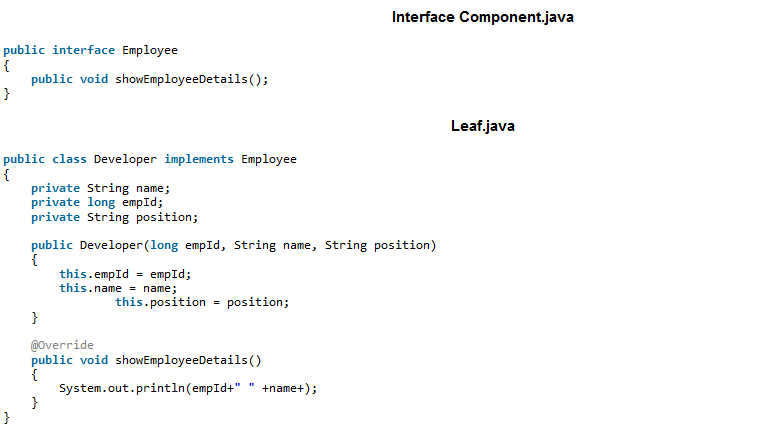


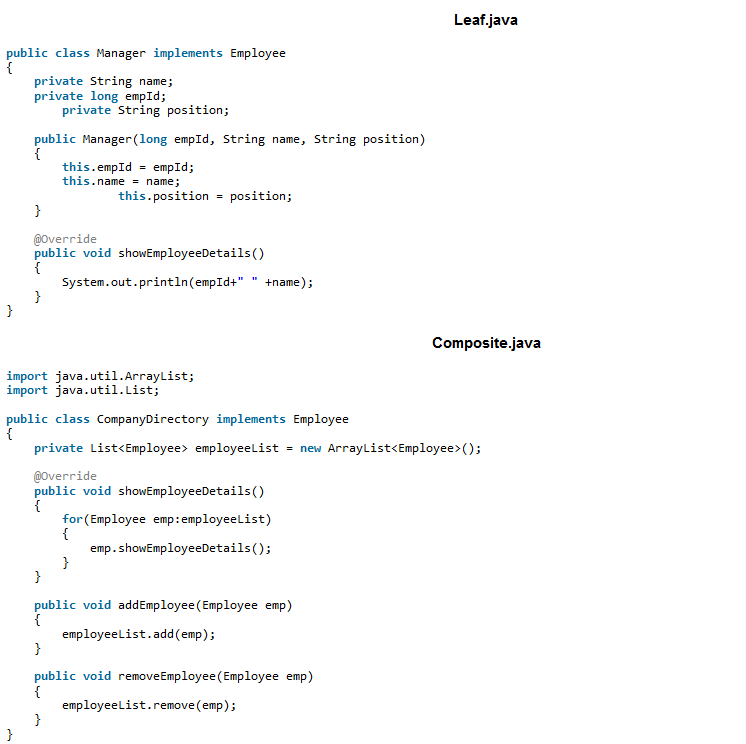


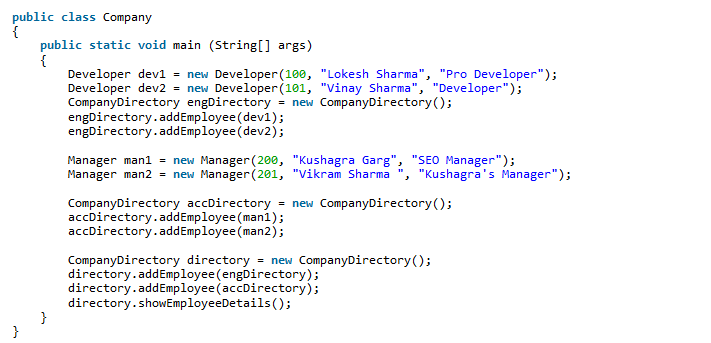
**Composite Design Pattern:**

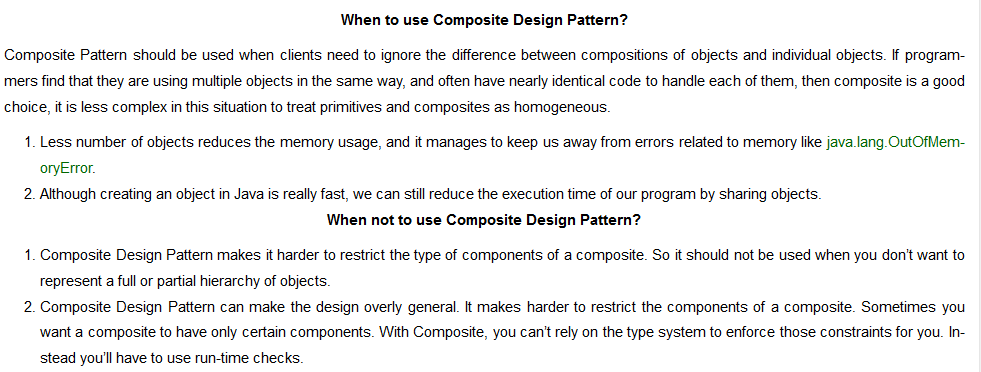


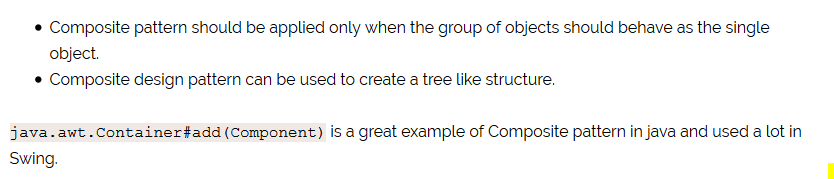




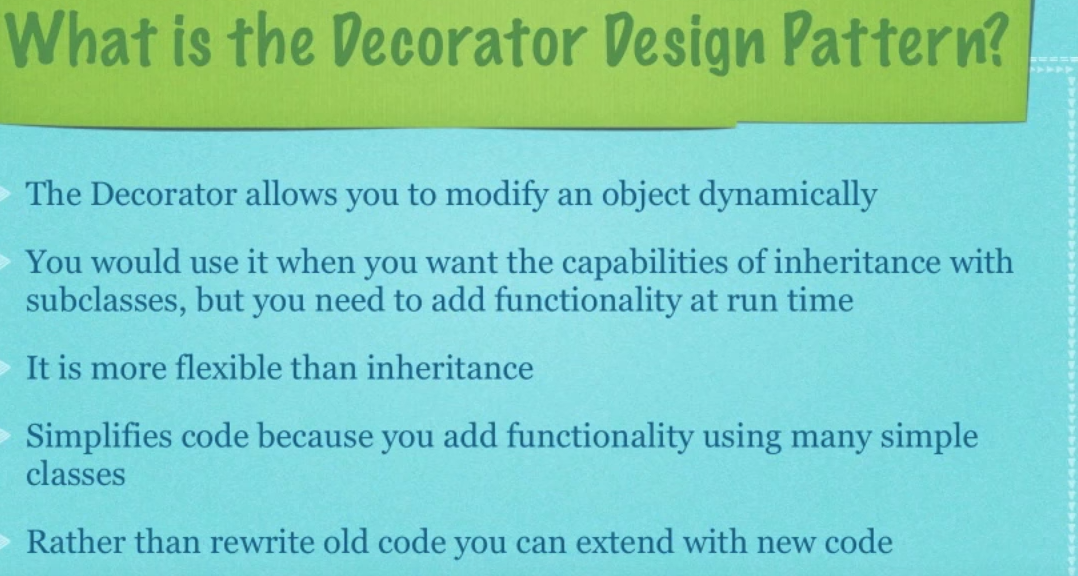


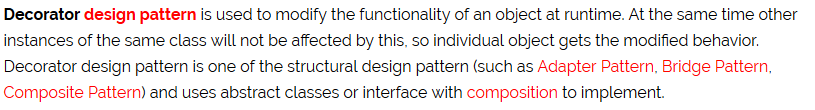


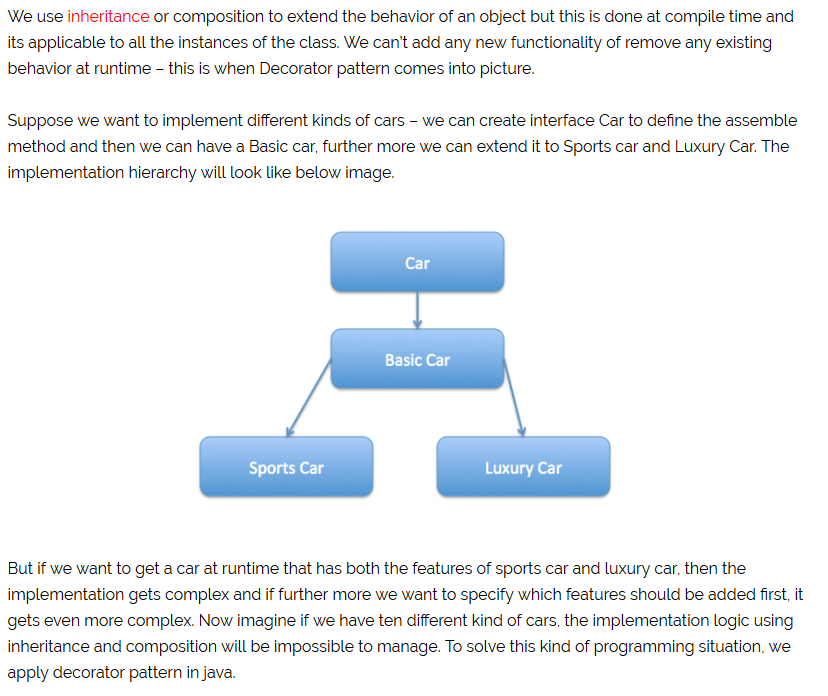


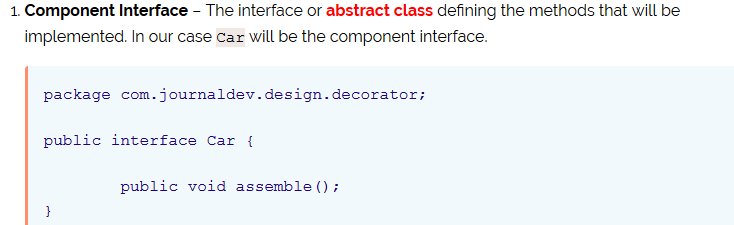


**Decorator Design Pattern:**

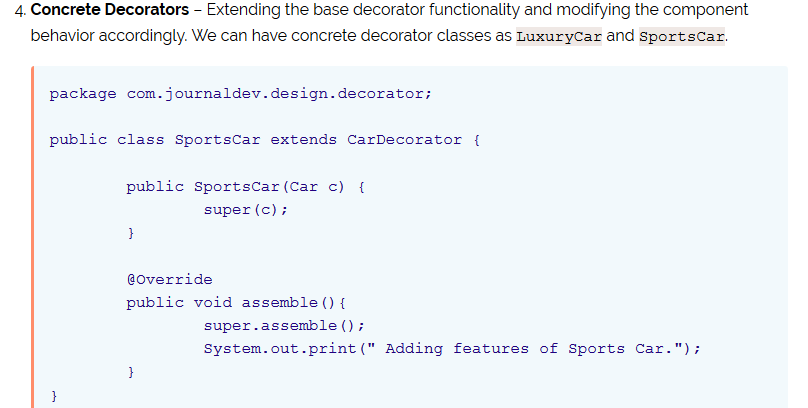




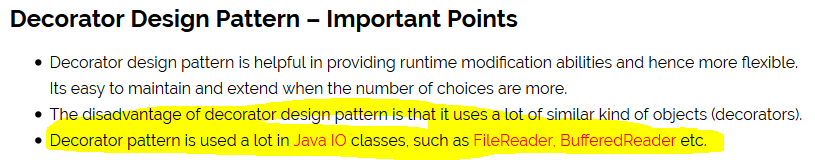




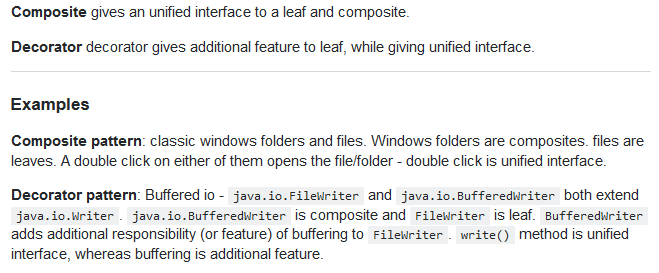




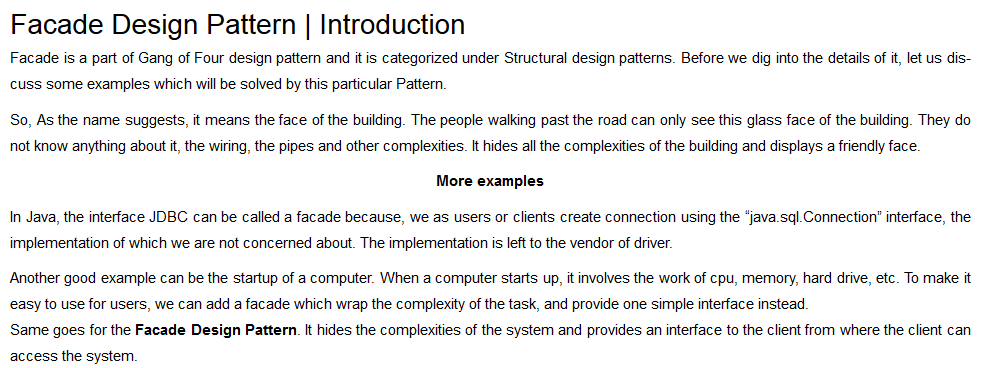


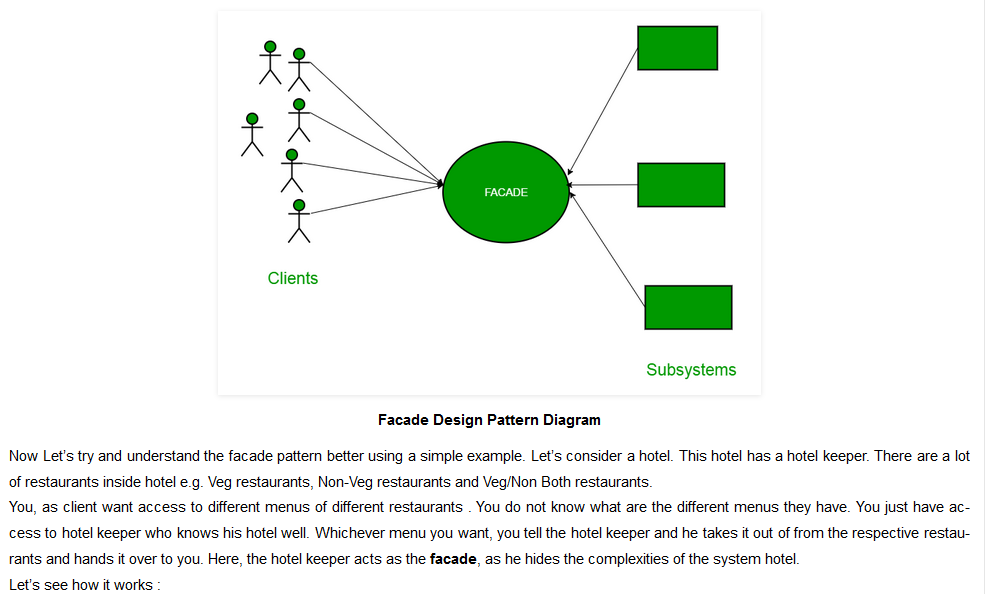


**Difference between Composite and Decorator Design Pattern:**

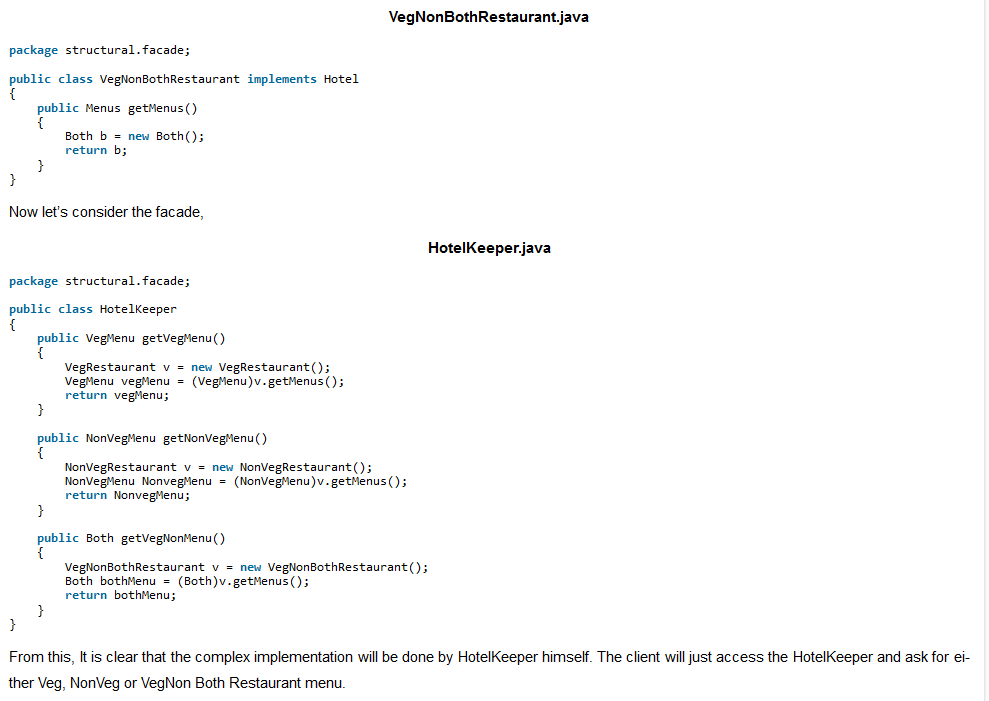


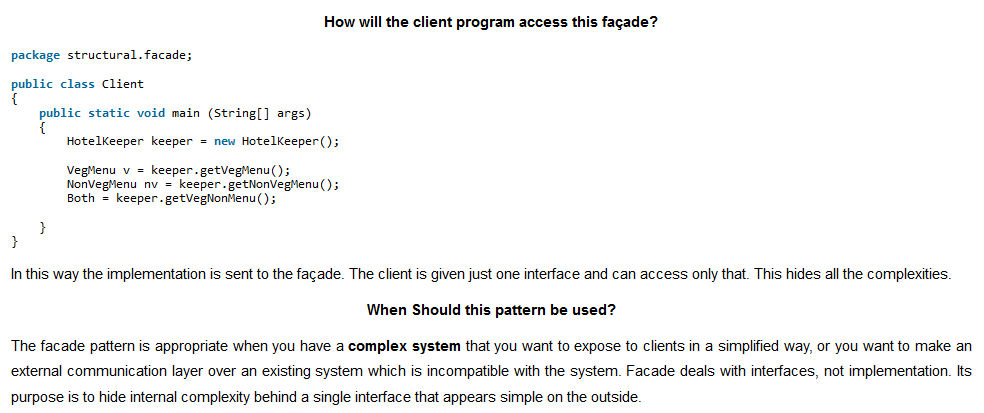
**Façade Design Pattern:**







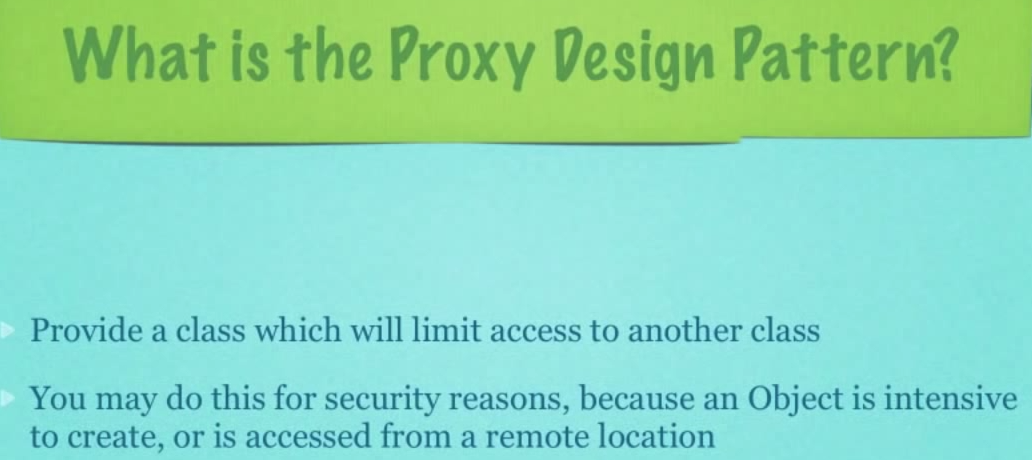




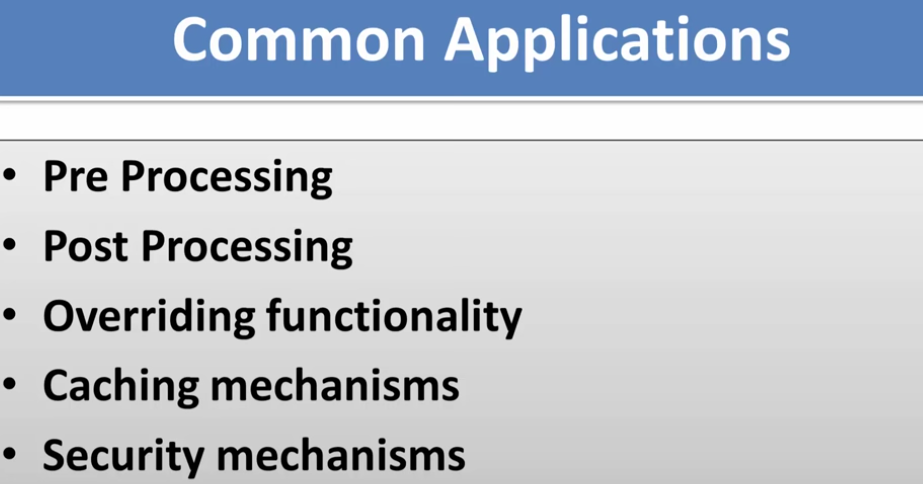
**Flyweight Design Pattern:**

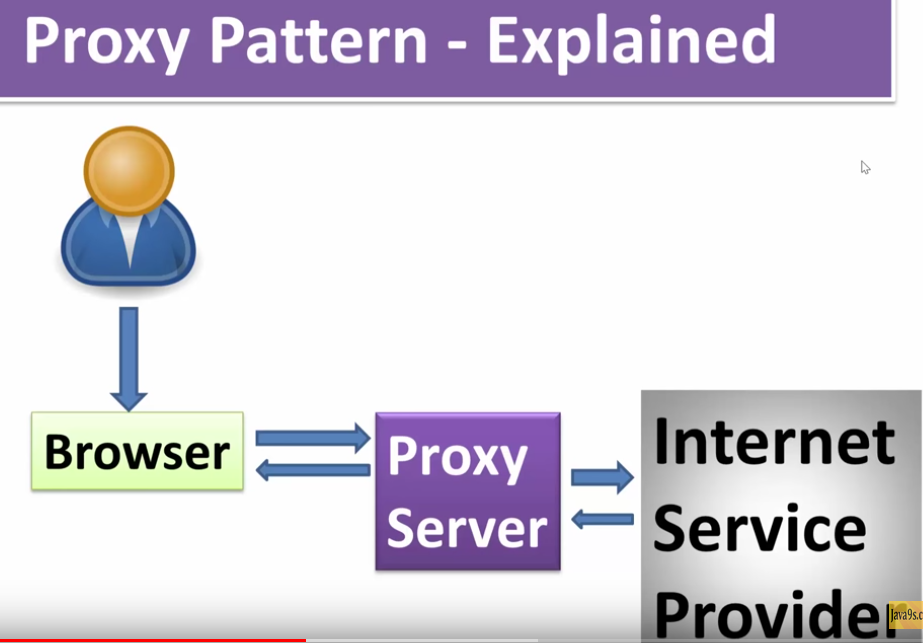
****

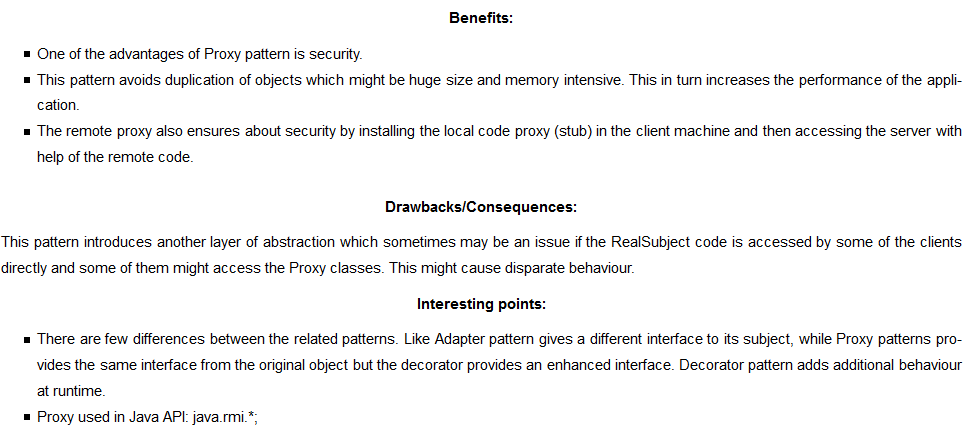
**Proxy Design Pattern:**



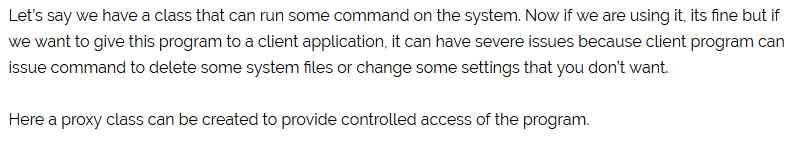


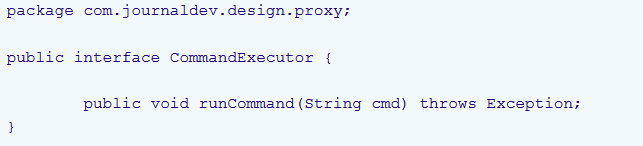


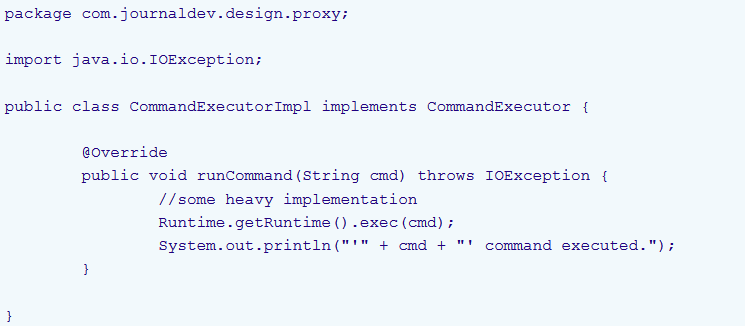


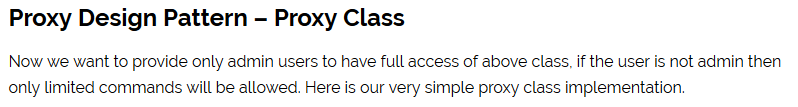


Example:













# Behavioral patterns

# Command Pattern

# 

# 

# 

# 

# 

# 

# 

# Interpreter Design Pattern:

# 

# 

# 

