1) Categories Java Design patterns?

Based on problem analysis, we can categorize design patterns into the following categories.

**Creational patterns**

* Factory method/Template
* Abstract Factory
* Builder
* Prototype
* Singleton

**Structural patterns**

* Adapter
* Bridge
* Filter
* Composite
* Decorator
* Facade
* Flyweight
* Proxy

**Behavioural patterns**

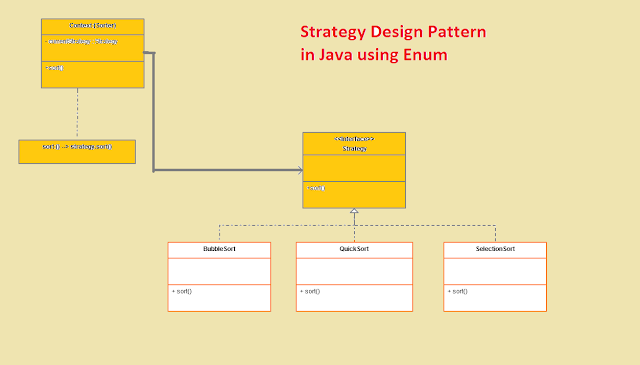
* Interpreter
* Template method/ pattern
* Chain of responsibility
* Command pattern
* Iterator pattern
* Strategy pattern
* Visitor pattern
* Command
* Memento
* Observer

**J2EE patterns**

* MVC Pattern
* Data Access Object pattern
* Front controller pattern
* Intercepting filter pattern
* Transfer object pattern

**Q. What is Strategy Design Pattern (behavioural design pattern). How can we create it in Java using Enum and interface?**

In this article, we will learn a new way to use Enum, for implementing the Strategy design pattern. Strategy pattern is one of the famous pattern, which takes advantage of polymorphism, to remove switch cases and strive for the open-closed design principle.  
  
Formally it encapsulates related algorithm, known as strategy and make them interchangeable. So your **Client**, also known as **Context, can use a different algorithm or strategy, without any modification.**  
  
**One of the key advantages of Strategy pattern is it's extensibility, like introducing a new Strategy is as easy as writing a new class and implementing Strategy interface**, **with Enum, instead of creating a separate class, you create a separate Enum instance, which means less number of classes and full benefit if Strategy pattern.  
  
UML diagram of the Strategy Design Pattern**

In this diagram, I have explained the strategy design pattern using sorting algorithms **like Bubble Sort, QuickSort, Insertion Sort, and Merge Sort.** For our code example, you can replace Strategy interface with Match, and sorting strategy to T20, OneDay, and Test Matches.  
The UML diagram clearly highlights the relationship between classes and the Strategy interface. You can see the Strategy interface has many implementations and they are swapped on Context class depending upon which kind of sorting you need.  
  


How to implement a Strategy pattern using Enum in Java

In Cricket, there are three popular formats T20 (20 over match), One day(50 over match), and Test(5 days match). If a player, plays all three formats, he needs to adjust its batting strategy to be effective.  
  
For example, T20 is all about scoring quickly, while One-day international games give a bit of time for setting and then scoring. In contrast to the shorter version, the Test match is all about grinding i.e. occupying the crease and making sure your opponent gets tired, before scoring briskly.  
  
In our Strategy pattern example, we have used an Enum to define different batting strategy. **We have defined a default play**() method and since Enum can override methods, **we are overriding it on every instance like on T20, ONE\_DAY, and TEST.**  
  
Our context class, which is named as Player also has a play() method, which delegates to play() method of configurable Strategy.

## Strategy Pattern Example using Enum

**The Strategy pattern is a good example of the Open Closed Design** Principle of **SOLID object-oriented design principles** coined by Uncle Bob in his classic Clean Codebook.

**import** **org.slf4j.Logger**;

**import** **org.slf4j.LoggerFactory**;

/\*\*

\* Java program to demonstrate that Enum can be used to implement Strategy

\* Pattern in Java.

\*

\* @author Javin

\*/

**public** **class** **Match** {

**private** **static** **final** Logger logger = LoggerFactory.getLogger(Match.class);

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

Player ctx = **new** Player(Strategy.T20);

ctx.play();

ctx.setStrategy(Strategy.ONE\_DAY);

ctx.play();

ctx.setStrategy(Strategy.TEST);

ctx.play();

}

}

/\*

\* Player class, which uses different Strategy implementation.

\*/

**class** **Player**{

**private** Strategy battingStrategy;

**public** **Player**(Strategy battingStrategy){

**this**.battingStrategy = battingStrategy;

}

**public** **void** **setStrategy**(Strategy newStrategy){

**this**.battingStrategy = newStrategy;

}

**public** **void** **play**(){

battingStrategy.play();

}

}

/\*

\* An Enum to implement Strategy design pattern in Java. Different instances of

\* Enum represent different batting strategy, based upon type of game e.g. T20,

\* One day international or Test match.

\*/

**enum** Strategy {

/\* Make sure to score quickly on T20 games \*/

T20 {

**@Override**

**public** **void** **play**() {

System.out.printf("In %s, If it's in the V, make sure it goes to tree %n",

name());

}

},

/\* Make a balance between attach and defence in One day \*/

ONE\_DAY {

**@Override**

**public** **void** **play**() {

System.out.printf("In %s, Push it for Single %n", name());

}

},

/\* Test match is all about occupying the crease and grinding opposition \*/

TEST {

**@Override**

**public** **void** **play**() {

System.out.printf("In %s, Grind them hard %n", name());

}

};

**public** **void** **play**() {

System.out.printf("In Cricket, Play as per Merit of Ball %n");

}

}

**Output:**

In T20, If it's in the V, make sure it goes to tree

In ONE\_DAY, Push it **for** Single

In TEST, Grind them hard

That's all on **How to Implement the Strategy design pattern in Java using Enum**. You can see that it's a lot easier to implement Strategy pattern with Enum, **you not only reduced the number of classes but also achieves extensibility and flexibility of Strategy pattern**. **On the downside**, Yes, you need to change your tried and tested Enum every time, though by inserting only new code, **it still violates Open Closed principle a bit,** which advocates that **new functionality should be added by writing new code, rather than modifying existing code.**

1. Strategy pattern is also known as **Policy Pattern.**
2. One of the best example of strategy pattern is Collections.sort() method that takes Comparator parameter. Based on the different implementations of Comparator interfaces, the Objects are getting sorted in different ways.

**Strategy pattern using interface Example:**

we will try to implement a simple Shopping Cart where we have two payment strategies – using Credit Card or using PayPal.

First of all we will create the interface for our strategy pattern example, in our case to pay the amount passed as argument.

public interface PaymentStrategy { public void pay(int amount); }

PaymentStrategy.java

package com.journaldev.design.strategy;

public interface PaymentStrategy {

public void pay(int amount);

}

Now we will have to create concrete implementation of algorithms for payment using credit/debit card or through paypal.

CreditCardStrategy.java

package com.journaldev.design.strategy;

public class CreditCardStrategy implements PaymentStrategy {

private String name;

private String cardNumber;

private String cvv;

private String dateOfExpiry;

public CreditCardStrategy(String nm, String ccNum, String cvv, String expiryDate){

this.name=nm;

this.cardNumber=ccNum;

this.cvv=cvv;

this.dateOfExpiry=expiryDate;

}

@Override

public void pay(int amount) {

System.out.println(amount +" paid with credit/debit card");

}

}

PaypalStrategy.java

package com.journaldev.design.strategy;

public class PaypalStrategy implements PaymentStrategy {

private String emailId;

private String password;

public PaypalStrategy(String email, String pwd){

this.emailId=email;

this.password=pwd;

}

@Override

public void pay(int amount) {

System.out.println(amount + " paid using Paypal.");

}

}

Now our strategy pattern example algorithms are ready. We can implement Shopping Cart and payment method will require input as Payment strategy.

Item.java

package com.journaldev.design.strategy;

public class Item {

private String upcCode;

private int price;

public Item(String upc, int cost){

this.upcCode=upc;

this.price=cost;

}

public String getUpcCode() {

return upcCode;

}

public int getPrice() {

return price;

}

}

ShoppingCart.java

package com.journaldev.design.strategy;

import java.text.DecimalFormat;

import java.util.ArrayList;

import java.util.List;

public class ShoppingCart {

//List of items

List<Item> items;

public ShoppingCart(){

this.items=new ArrayList<Item>();

}

public void addItem(Item item){

this.items.add(item);

}

public void removeItem(Item item){

this.items.remove(item);

}

public int calculateTotal(){

int sum = 0;

for(Item item : items){

sum += item.getPrice();

}

return sum;

}

public void pay(PaymentStrategy paymentMethod){

int amount = calculateTotal();

paymentMethod.pay(amount);

}

}

Notice that payment method of shopping cart requires payment algorithm as argument and doesn’t store it anywhere as instance variable.

Let’s test our strategy pattern example setup with a simple program.

ShoppingCartTest.java

package com.journaldev.design.strategy;

public class ShoppingCartTest {

public static void main(String[] args) {

ShoppingCart cart = new ShoppingCart();

Item item1 = new Item("1234",10);

Item item2 = new Item("5678",40);

cart.addItem(item1);

cart.addItem(item2);

//pay by paypal

cart.pay(new PaypalStrategy("myemail@example.com", "mypwd"));

//pay by credit card

cart.pay(new CreditCardStrategy("Pankaj Kumar", "1234567890123456", "786", "12/15"));

}

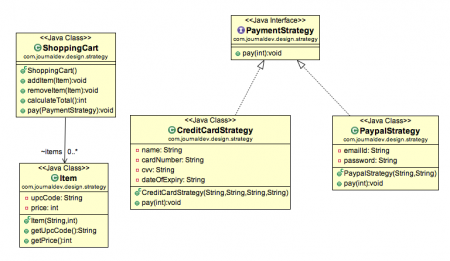
}

Output of above program is:

50 paid using Paypal.

50 paid with credit/debit card

Strategy Design **Pattern Class Diagram**

**[](https://cdn.journaldev.com/wp-content/uploads/2013/07/Strategy-Pattern.png)**

**Strategy Design Pattern Important Points**

* We could have used composition to create instance variable for strategies but we should avoid that as we want the specific strategy to be applied for a particular task. **Same is followed in Collections.sort() and Arrays.sort() method that take comparator as argument.**
* **Strategy Pattern** is very similar to State Pattern. **One of the difference is that Context contains state as instance variable and there can be multiple tasks whose implementation can be dependent on the state** whereas **in strategy pattern strategy is passed as argument to the method and context object doesn’t have any variable to store it.**
* Strategy pattern is useful **when we have multiple algorithms for specific task** and **we want our application to be flexible to chose any** of the algorithm at runtime for specific task.

**Q. What is Command Pattern?**

**Ans.** Command pattern help you **write flexible, loosely coupled code** for implementing actions and events in your application. In simple words, the command design pattern is **used to separate a request for action from the object which actually performs the action.** This decoupling between **Invoker** and **Receiver** objects provides a uniform way to perform different types of actions. **This decoupling is achieved using a Command object, which is usually an interface with methods like execute().**   
  
The Requestor or Invoker only knows about the Command object and doesn't care about the actual object which processes the request, which can be different. This transparency leads to cleaner code on the Invoker side and also enables the opportunity to do several smart things on the Command side.  
  
Your Command object can be as dumb as simply delegating the request to Receiver and can be as smart as recording the last command for performing UNDO and REDO functionality.  
  
The Command pattern ensures that your code is compliant with open closed design principles, **the O** of SOLID design principles, which means adding a new command would be very easy, creating a new implementation of the Command interface and doesn't affect the Invoker code.  
 **The command pattern is particularly popular in GUI applications**, where we use lots of commands like **Open**, **Close**, **Save**, **Cut**, **Copy**, **Paste**, and corresponding **UNDO** and **REDO** operations.

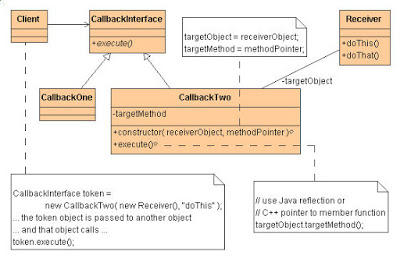
## 1. Command Pattern Terminology

Before going further and implementing a command pattern in Java, let's get familiar with the terminology used in this pattern.

* **Client**- Creates Concrete Command Object and configure with Receiver
* **Invoker**- Who hold command and calls execute() method on Command object
* **Receiver**- Actual object, which processes the request
* **Command**- Interface, which takes a request from Invoker and delegates to Receiver
* **ConcreteCommand**- implementation of Command interface for doing a particular task

### UML Diagram of the Command Design Pattern

Here is the UML diagram of the Command Design Pattern, which will make things more clear.

[](https://click.linksynergy.com/fs-bin/click?id=JVFxdTr9V80&subid=0&offerid=323058.1&type=10&tmpid=14538&RD_PARM1=https%3A%2F%2Fwww.udemy.com%2Fjava-design-patterns-the-complete-masterclass%2F)

You can see that Client has a reference of a CallbackInterface which has a generic method execute(). The individual commands implement this interface and provide an implementation which is nothing but delegating to the actual object for doing things.  
  
The important thing is that the client doesn't know about the Actual object which is performing an action on behalf of those commands. This decoupling results in flexible code and makes it easy to add new commands without impacting client code.  
  
This is possible because of the **open-closed design** principles which make **the code open for extension but closed for modification**.

## 2. Command Design Pattern in Java - Example

Here is our sample program to demonstrate how to use the command pattern in Java.  
  
This class represent Client of Command pattern  
  
**Client.java**

**import** java.util.HashMap;

**import** java.util.Map;

**import** org.slf4j.Logger;

**import** org.slf4j.LoggerFactory;

/\*\*

\* Java program to implement Command design pattern with example.

\*

\* **@author** Javin Paul

\*/

public class Client {

private static final Logger logger **=** LoggerFactory**.**getLogger(Client**.**class);

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

// Client creates Invoker object, command object and configure them

Menu menu **=** **new** Menu();

menu**.**setCommand("Create", **new** CreateCommand());

menu**.**setCommand("Delete", **new** DeleteCommand());

//Invoker invokes command

menu**.**runCommand("Create");

menu**.**runCommand("Delete");

}

}

Output**:**

Creating file

Deleting file

This class represents a command  
  
**Menu.java**

/\*

\* Invoker class, which holds command object and invokes method

\*/

public class Menu{

Map menuItems **=** **new** HashMap();

public void setCommand(String **operation**, Command **cmd**){

menuItems**.**put(operation, cmd);

}

public void runCommand(String **operation**){

menuItems**.**get(operation)**.**execute();

}

}

An interface to represent commands :  
  
**Command.java**

/\*

\* Command interface for implementing concrete command

\*/

interface Command{

public void execute();

}

Some implementation of command interface, this command will delete files  
  
**DeleteCommand.java**

/\*

\* Concrete command to delete files

\*/

public class DeleteCommand implements Command{

@Override

public void execute() {

System**.**out**.**println("Deleting file");

}

}

Another implementation of the command interface to create files :  
  
**CreateCommand.java**

/\*

\* Concrete command to create file

\*/

public class CreateCommand implements Command{

@Override

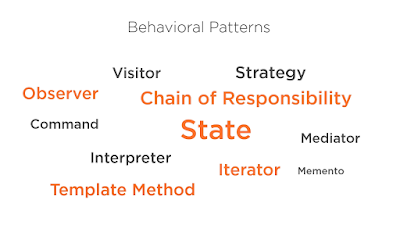
public void execute() {

System**.**out**.**println("Creating file");

}

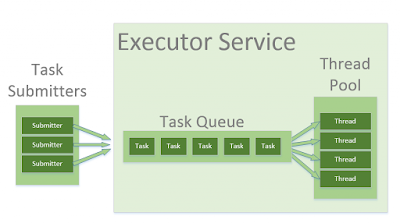
}

You can see we have **created two commands Create and Delete** by using Command pattern. It's one of the GOF patterns as well.  
  
**You can see Command is an interface and** both **CreateCommand and DeleteCommand implement that interface** and **implements execute()** method to put the logic of what command is supposed to do.



## 3. Real-world Examples of Command Pattern in Java

Two of the Command pattern examples from JDK **are java.lang.Runnable and javax.swing.Action** interface. If you look closely you will find that Thread Pool executors are invoker of Runnable commands.



## 4. Benefits of Command Design Pattern

As we have seen in this article, the main benefit of using the Command pattern in Java is decoupling the Invoker of a command from the object, which does the actual processing. By decoupling them and using introducing **Command object**, we create a design which can keep track of every state changed via Command objects.  
  
This knowledge can be used for useful purpose e.g. implementing UNDO and REDO operations, recording or queuing of request, etc. In short command design pattern provides the following advantages :  
  
**1) Encapsulate request processing.**  
2) **Decouples clients from an object which actually processes the request and provides a uniform way to perform a similar task.**  
3) **Since Command pattern encapsulates request, It can be used to record state, implement Undo and redo functionality, Queuing a request, etc.**  
4) **Command Pattern makes it easy to add new commands. It also follows the Open Closed design principle**, where new functionality is added by creating a new code. Since request processing is transparent to Invoker, adding a new command doesn't require a change on their side.

## 5. Cons of the Command Pattern

Like any other design decision, the Command pattern also involves tradeoffs. If you have a lot of commands, just like in a major GUI application like popular Java IDEs like Eclipse or IntelliJIDEA, you might end up with lots of small command classes.  
  
Though similar functionality can be grouped into a couple of classes with each method performing a task for a particular command.  
  
 By the way, using the Command pattern result in readable, maintainable, and extensible code, so this cost is worth paying.  
  
Observer design Pattern in Java with Real world code Example

**What is Observer design pattern** **in Java?**

**Ans. Observer design is a fundamental core Java pattern where Observe watch for any change in state or property of Subject**. For Example Company updates all its shareholders for any decision they make here Company is Subject and Shareholders are Observers, any change in policy of company and Company notifies all its Shareholders or Observer. This was simple real world explanation of Observer pattern. In this article we will in detail *what is Observer Design pattern*, what is *benefit of Observer design Pattern*, Example or Observer pattern in Java and few other points. Just like Decorator design Pattern and Factory Pattern in Java, Observer pattern is also used in JDK.

## Observer design Pattern Java Code Example

Now, let's deep dive into Observer design Pattern and how to use it in Java programming language:  
  
**1. What is Observer design Pattern?**

*Observer design pattern in Java* is very important pattern and as name suggest it’s used to observe things. Suppose you want to notify for change in a particular object than you observer that object and changes are notified to you. Object which is being observed is refereed as Subject and classes which observe subject are called Observer.   
  
This is beautiful pattern and **used heavily along with Model View Controller Design pattern** where change in model is propagated to view so that it can render it with modified information.

### 2. The problem which is solved by Observer Pattern:

In real world if try to find example see when we subscribe for New Phone connection whenever customer is registered with that company all other departments are notified accordingly and then depending upon the state the do their jobs like do the verification of their address then if customer state is verified then dispatch the welcome kit etc.

## How the Observer Design Pattern is implemented in Java;

For implementation of this pattern java makes our task very easy, developer need not to do so much for implementing this pattern .In **java.util**package we can find interfaces ,classes and methods for implementing this pattern.

**Public Interface Observer:**

Any class who implements this interface must be notified when subject or observable object change its status.

**Update (Observable Ob, Object arg):** This method is called when subject is changed.

**Class Observable:**

It’s a subject to whom observer wants to observe.

**Some Important Method:**

**addObserver(Observer o):** add Observers in the set of observers for this subject or observalbel object.

**deleteObserver(Observer o):**  delete Observers in the set of observers .

**hasChanged():** check if object has changed.

**clearChanged():** this method will indicate that subject has no changes or all the observers has been notified when changes is made.

**notifyObservers():** notify all the observers if object has changed .

## Code Example of Observer design pattern in Java:

**The observer Design pattern** is generic than how it is implemented in Java. You are free to choose java.util.Observable or java.util.Observer or *writing your own Subject and Observer interface*. I prefer having my own Subject and Observer interface and this is how I am going to write my Code Example of Observer design Pattern in java.   
  
My Example is very Simple you have a **Loan on which interest rate is subject to change and when it changes, Loan notifies to Newspaper or Internet media to display a new loan interest rate.**To implement this we have a **Subject** interface that contains methods for adding, removing and notifying Observers and an **Observer** interface which contains update(int interest) method which will be called by the Subject implementation when the interest rate changes.

**import** java.util.ArrayList;

**interface** Observer {

**public** **void** update(**float** interest);

}

**interface** Subject {

**public** **void** registerObserver(Observer observer);

**public** **void** removeObserver(Observer observer);

**public** **void** notifyObservers();

}

**class** Loan **implements** Subject {

**private** ArrayList<Observer> observers = **new** ArrayList<Observer>();

**private** String type;

**private** **float** interest;

**private** String bank;

**public** Loan(String type, **float** interest, String bank) {

**this**.type = type;

**this**.interest = interest;

**this**.bank = bank;

       }

**public** **float** getInterest() {

**return** interest;

       }

**public** **void** setInterest(**float** interest) {

**this**.interest = interest;

              notifyObservers();

       }

**public** String getBank() {

**return** **this**.bank;

       }

**public** String getType() {

**return** **this**.type;

       }

       @Override

**public** **void** registerObserver(Observer observer) {

              observers.add(observer);

       }

       @Override

**public** **void** removeObserver(Observer observer) {

              observers.remove(observer);

       }

       @Override

**public** **void** notifyObservers() {

**for** (Observer ob : observers) {

                     System.*out*

                                  .println("Notifying Observers on change in Loan interest rate");

                     ob.update(**this**.interest);

              }

       }

}

**class** Newspaper **implements** Observer {

       @Override

**public** **void** update(**float** interest) {

              System.*out*.println("Newspaper: Interest Rate updated, new Rate is: "

                           + interest);

       }

}

**class** Internet **implements** Observer {

       @Override

**public** **void** update(**float** interest) {

              System.*out*.println("Internet: Interest Rate updated, new Rate is: "

                           + interest);

       }

}

**public** **class** ObserverTest {

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

              // this will maintain all loans information

              Newspaper printMedia = **new** Newspaper();

              Internet onlineMedia = **new** Internet();

              Loan personalLoan = **new** Loan("Personal Loan", 12.5f,

                           "Standard Charterd");

              personalLoan.registerObserver(printMedia);

              personalLoan.registerObserver(onlineMedia);

              personalLoan.setInterest(3.5f);

       }

}

### Advantage of Observer Design Pattern in Java:

The main advantage is **loose coupling** between objects called observer and observable. The subject only knows the list of observers it don’t care about how they have their implementation. All the observers are notified by subject in a single event call as **Broadcast communication**

### Disadvantage of Observer Design Pattern in Java:

·          The disadvantage is that the sometime if any problem comes, debugging becomes very difficult because flow of control is implicitly between **observers** and **observable** we can predict that now observer is going to fire and if there is chain between observers then **debugging become more complex.**

·          Another issue **is Memory management because subject will hold all the reference of all the observers if we not unregister the object it can create the memory issue.**

**Q. What is Factory method Design Pattern in Java with Example -**

Factory design pattern in Java one of the core design pattern which is used heavily not only in JDK but also in various Open Source framework such as Spring, Struts and Apache along with decorator design pattern in Java.

**Factory Design pattern is based on Encapsulation object oriented concept**. Factory method is used to create different object from factory often refereed as Item and it encapsulate the creation code**. So instead of having object creation code on client side we encapsulate inside Factory method in Java.** One of the best examples of factory pattern in Java is BorderFactory Class of Swing API.  
  
**What is static factory method or factory design pattern**

Factory design pattern is used to create objects or Class in Java and it provides loose coupling and high cohesion. Factory pattern encapsulate object creation logic which makes it easy to change it later when you change how object gets created or you can even introduce new object with just change in one class.

**Factory should be an interface** and **clients first either creates factory or get factory which later used to create objects.**

## Example of a static factory method in JDK

**Best Example of Factory method design pattern is valueOf() method which is there in String and wrapper classes like Integer and Boolean and used for type conversion** i.e. from converting String to Integer or String to double in java..

Some more examples of factory method design pattern from JDK is :

**valueOf() method** which returns object created by factory equivalent to value of parameter passed.

**getInstance() method which creates instance of Singleton class.**

newInstance() method which is used to create and return new instance from factory method every time called.

**getType() and newType() equivalent of getInstance() and newInstance() factory method** but used when factory method resides in separate class.

## **The problem which is solved by Factory method Pattern in Java**

Whenever we talk about **object oriented language** it will based upon some concept like abstraction, polymorphism etc and on that encapsulation and delegation are important concept any design will be called good if task are delegated to different object and some kind of encapsulation is there.

Some time our application or framework will not know that what kind of object it has to create at run-time it knows only the interface or abstract class and as we know we can not create object of interface or abstract class so main problem is frame work knows **when** it has to create but don’t know **what kind** of object.

**Whenever we create object using new() we violate principle of programming for interface rather than implementation** which eventually result in inflexible code and difficult to change in maintenance. **By using Factory design pattern in Java we get rid of this problem.**

Another problem we can face is class needs to contain objects of other classes or class hierarchies within it; **this can be very easily achieved by just using the new keyword and the class constructor.** **The problem with this approach is that it is a very hard coded approach to create objects as this creates dependency between the two classes.**

So **factory pattern** solve this problem very easily by model an interface for creating an object which at creation time can let its subclasses decide which class to instantiate, Factory Pattern promotes loose coupling by eliminating the need to bind application-specific classes into the code. The **factory methods** are typically implemented as virtual methods, so this pattern is also referred to as the “**Virtual Constructor**”. These methods create the objects of the products or target classes.

## When to use the Factory design pattern in Java

* Static Factory methods are common in frameworks where library code needs to create objects of types which may be sub classed by applications using the framework.
* Some or all concrete products can be created in multiple ways, or we want to leave open the option that in the future there may be new ways to create the concrete product.
* Factory method is used when Products don't need to know how they are created.
* We  can use factory pattern where we have to create an object of any one of sub-classes depending on the data provided

### Code Example of Factory Design Pattern in Java:

Let’s see an example of how factory pattern is implemented in Code.We have requirement to create multiple currency e.g. INR, SGD, USD and code should be extensible to accommodate new Currency as well. Here we have made Currency as interface and all currency would be concrete implementation of Currency interface. Factory Class will create Currency based upon country and return concrete implementation which will be stored in interface type. This makes code dynamic and extensible.

Here is complete **code example of Factory pattern in Java**:

**interface** Currency {

       String getSymbol();

}

// Concrete Rupee Class code

**class** Rupee **implements** Currency {

       @Override

**public** String getSymbol() {

**return** "Rs";

       }

}

// Concrete SGD class Code

**class** SGDDollar **implements** Currency {

       @Override

**public** String getSymbol() {

**return** "SGD";

       }

}

// Concrete US Dollar code

**class** USDollar **implements** Currency {

       @Override

**public** String getSymbol() {

**return** "USD";

       }

}

// Factroy Class code

**class** CurrencyFactory {

**public** **static** Currency createCurrency (String country) {

**if** (country. equalsIgnoreCase ("India")){

**return** **new** Rupee();

       }**else** **if**(country. equalsIgnoreCase ("Singapore")){

**return** **new** SGDDollar();

       }**else** **if**(country. equalsIgnoreCase ("US")){

**return** **new** USDollar();

        }

**throw** **new** IllegalArgumentException("No such currency");

       }

}

// Factory client code

**public** **class** Factory {

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

              String country = args[0];

              Currency rupee = CurrencyFactory.*createCurrency*(country);

              System.*out*.println(rupee.getSymbol());

       }

}

### Advantage of Factory method Pattern in Java:

**Factory pattern in Java** is heavily used everywhere including JDK, open source library and other frameworks.In following are main advantages of using Factory pattern in Java:

**1*) Factory method design pattern*** decouples the calling class from the target class, which result in less coupled and highly cohesive code?

**E.g.: JDBC is a good example for this pattern;** application code doesn't need to know what database it will be used with, so it doesn't know what database-specific driver classes it should use. Instead, it uses factory methods to get Connections, Statements, and other objects to work with. Which gives you flexibility to change your back-end database without changing your DAO layer in case you are using ANSI SQL features and not coded on DBMS specific feature?

2) Factory pattern in Java enables the subclasses to provide extended version of an object, because creating an object inside factory is more flexible than creating an object directly in the client. Since client is working on interface level any time you can enhance the implementation and return from Factory.

3) Another benefit of using *Factory design pattern in Java* is that it encourages consistency in Code since every time object is created using Factory rather than using different constructor at different client side.

4) Code written using Factory design pattern in Java is also easy to debug and troubleshoot because you have a centralized method for object creation and every client is getting object from same place.

**Some more advantages of factory method design pattern is:**

1. **Static factory method** used in factory design pattern **enforces use of Interface** than implementation which itself a good practice. for example:

**Map** synchronizedMap = **Collections**.synchronizedMap(**new** **HashMap**());

2. **Since static factory method have return type as Interface, it allows you to replace implementation with better performance version in newer release.**

3. Another advantage of static factory method pattern is that they can cache frequently used object **and eliminate duplicate object creation**. Boolean.valueOf() method is good example which caches true and false boolean value.

4. Factory method pattern is also recommended by Joshua Bloch in Effective Java.

5 Factory method pattern offers alternative way of creating object.

6. Factory pattern can also be used to hide information related to creation of object.

### Q. What is decorator design pattern in Java?

**Ans.**

**Decorator design pattern** is used to modify the functionality of an object at runtime. At the same time other instances of the same class will not be affected by this, so individual object gets the modified behavior. **Decorator design pattern is one of the structural design pattern** (**such as Adapter Pattern, Bridge Pattern, Composite Pattern**) and uses abstract classes or interface with composition to implement.

**Decorator Design Pattern**

We use inheritance or composition to extend the behaviour of an object but this is done at compile time and its applicable to all the instances of the class. We can’t add any new functionality of remove any existing behavior at runtime – this is when Decorator pattern comes into picture.

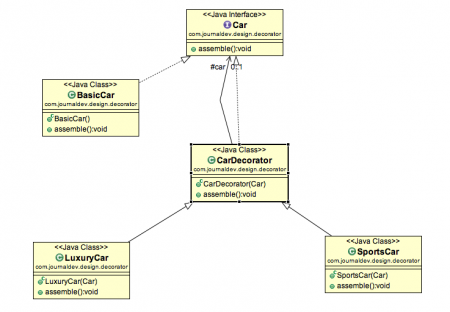
Suppose we want to implement different kinds of cars – we can create interface Car to define the assemble method and then we can have a **Basic car,** further more we can extend it to **Sports car** and **Luxury Car**.

But if we want to get a car at runtime that has both the features of sports car and luxury car, then the implementation gets complex and if further more we want to specify which features should be added first, it gets even more complex. Now imagine if we have ten different kind of cars, the implementation logic using inheritance and composition will be impossible to manage. To solve this kind of programming situation, we apply decorator pattern in java.

We need to have following types to implement decorator design pattern.

1. **Component Interface** – The interface or **abstract class** defining the methods that will be implemented. In our case Car will be the component interface.
2. package com.journaldev.design.decorator;
3. public interface Car {
4. public void assemble();
5. }
6. **Component Implementation** – The basic implementation of the component interface. We can have BasicCar class as our component implementation.
7. package com.journaldev.design.decorator;
8. public class BasicCar implements Car {
9. @Override
10. public void assemble() {
11. System.out.print("Basic Car.");
12. }
13. }
14. **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.
15. package com.journaldev.design.decorator;
16. public class CarDecorator implements Car {
17. protected Car car;
19. public CarDecorator(Car c){
20. this.car=c;
21. }
23. @Override
24. public void assemble() {
25. this.car.assemble();
26. }
27. }
28. **Concrete Decorators** – Extending the base decorator functionality and modifying the component behavior accordingly. We can have concrete decorator classes as LuxuryCar and SportsCar.
29. package com.journaldev.design.decorator;
30. public class SportsCar extends CarDecorator {
31. public SportsCar(Car c) {
32. super(c);
33. }
34. @Override
35. public void assemble(){
36. super.assemble();
37. System.out.print(" Adding features of Sports Car.");
38. }
39. }
40. package com.journaldev.design.decorator;
41. public class LuxuryCar extends CarDecorator {
42. public LuxuryCar(Car c) {
43. super(c);
44. }
46. @Override
47. public void assemble(){
48. super.assemble();
49. System.out.print(" Adding features of Luxury Car.");
50. }
51. }

**Decorator Design Pattern – Class Diagram**

[](https://cdn.journaldev.com/wp-content/uploads/2013/07/decorator-pattern.png)

**Decorator Design Pattern Test Program**

package com.journaldev.design.test;

import com.journaldev.design.decorator.BasicCar;

import com.journaldev.design.decorator.Car;

import com.journaldev.design.decorator.LuxuryCar;

import com.journaldev.design.decorator.SportsCar;

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();

}

}

Notice that client program can create different kinds of Object at runtime and they can specify the order of execution too.

Output of above test program is:

Basic Car. Adding features of Sports Car.

\*\*\*\*\*

Basic Car. Adding features of Luxury Car. Adding features of Sports Car.

**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 classes, such **as FileReader, BufferedReader**

**Q.How to create thread safe Singleton in Java - Java Singleton Example**

Ans. Thread safe Singleton means a Singleton class which returns exactly same instance even if exposed to multiple threads. Singleton in Java has been a classical design pattern like Factory method pattern or Decorator design pattern and has been used a lot even inside JDK like java.lang.Runtime is an example of Singleton class. Singleton pattern ensures that exactly one instance of class will remain in Java program at any time.   
**Prior to Java 5 double checked locking mechanism is used to create *thread-safe singleton* in Java which breaks if one Thread doesn't see instance created by other thread at same time and eventually you will end up with more than one instance of Singleton class.**  
**From Java 5 onwards volatile variable guarantee can be used to write thread safe singleton by using double checked locking pattern.**

1. **using static field to initialize Singleton instance** or
2. **using Enum as Singleton in Java**.

Java Singleton Example – Thread safe Singleton in Java using Enum

**Using Enum to create Singleton is by far most simple and effective way to create thread-safe Singleton in Java, as thread-safety guarantee is provided by Java programming language itself.** **Since Enum instances are by default final in Java**, it also provides safety against multiple instance due to serialization.   
we are talking about thread-safety during instance creation of Singleton class and not when we call any method of Singleton class. 

**public** **enum** Singleton{  
    INSTANCE;  
   
    **public** **void** show(){  
        **System**.out.println("Singleton using Enum in Java");  
    }  
}

//You can access this Singleton as Singleton.INSTANCE and call any method like below

Singleton.INSTANCE.show();

If this suits your need than this is the most easy way of writing thread-safe Singleton in Java. Using Enum as Singleton also provide couple of more benefits which you can find out on Why Enum Singleton are better in Java.

**Java Singleton Example - Thread Safe Singleton using Static field Initialization**

You can also create thread safe Singleton in Java by creating Singleton instance during class loading. **static fields are initialized during class loading and Classloader will guarantee that instance will not be visible until its fully created**. Here is example of creating thread safe singleton in **Java using static factory method**. Only **disadvantage** of this implementing Singleton patter using static field is that **this is *not a lazy initialization***and Singleton is initialized even before any clients call there getInstance() method.

**public** **class** Singleton{  
    **private** **static** **final** Singleton INSTANCE = **new** Singleton();  
   
    **private** Singleton(){ }  
  
    **public** **static** Singleton getInstance(){  
        **return** INSTANCE;  
    }  
    **public** **void** show(){  
        **System**.out.println("Singleon using static initialization in Java");  
    }  
}

//Here is how to access this Singleton class

Singleton.getInstance().show();

**here we are not creating Singleton instance inside getInstance() method instead it will be created by ClassLoader.** Also, private constructor makes impossible to create another instance, except one case. You can still access private constructor by reflection and calling **setAccessible(true).** By the way You can still prevent creating another instance of Singleton by this way by throwing Exception from constructor.

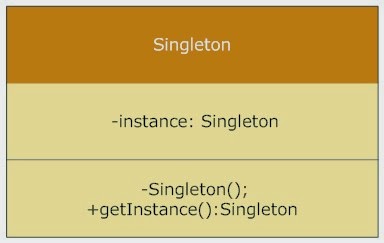
### What is Singleton class? Have you used Singleton before?

Singleton is a class which has only one instance in whole application and provides a getInstance() method to access the singleton instance. There are many classes in JDK which is implemented using Singleton pattern like **java.lang.Runtime** which provides **getRuntime()** method to get access of it and used to get free memory and total memory in Java.

### Which classes are candidates of Singleton? Which kind of class do you make Singleton in Java? Answer : Any class which you want to be available to whole application and whole 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.

### Can you write code for getInstance() method of a Singleton class in Java?

**Answer** : Until asked don’t write code using double checked locking as it is more complex and chances of errors are more but if you have deep knowledge of **double checked locking**, **volatile variable** and **lazy loading** than this is your chance to shine. I have shared code examples of writing singleton classes using enum, using static factory and with double checked locking in my recent post Why Enum Singletons are better in Java, please see there.

[](https://2.bp.blogspot.com/-vIw-2sGSrko/VJmIXk6xfnI/AAAAAAAACP4/clWZsrAY3Ro/s1600/Singleton%2BDesign%2BPattern%2Bin%2BJava%2B2.jpg)

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

This is really nice question and I mostly asked to just quickly check whether candidate is aware of performance trade off of unnecessary locking or not. Since locking only make sense when **we need to create instance and rest of the time its just read only access so locking of critical section is always better option**.

**Answer :** This is again related to double checked locking pattern, well 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. Singleton pattern is also closely related to factory design pattern where getInstance() serves as static factory method.

**What is lazy and early loading of Singleton and how will you implement it?**  
**Answer :** As there are many ways to implement Singleton like **using double checked locking or Singleton class with static final 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?**

This is open question to all, please share which classes are Singleton in JDK. Answer to this question is **java.lang.Runtime**

Answer : 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?

**Answer** : 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.** See here to learn more about double-checked-locking in Java.

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**;

}

**Double checked locking should only be used when you have requirement for lazy initialization** otherwise **use Enum to implement singleton or simple static final variable**.  
  
**Q. 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 “Can not create clone of Singleton class”.

### How do you prevent for creating another instance of Singleton using reflection?

**Answer**: This is similar to previous interview question. Since constructor of Singleton class is supposed to be private it prevents creating instance of Singleton from outside but Reflection can access private fields and methods, which opens a threat of another instance**. This can be avoided by throwing Exception from constructor as “Singleton already initialized”**

**How do you prevent for creating another instance of Singleton during serialization?**  
**Answer**: 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.**

### Why you should avoid the singleton anti-pattern at all and replace it with DI?

Answer : Singleton Dependency Injection: every class that needs access to a singleton gets the object through its constructors or with a DI-container.

### Why Singleton is Anti pattern

With more and more classes calling getInstance() the code gets more and more tightly coupled, monolithic, not testable and hard to change and hard to reuse because of not configurable, hidden dependencies. Also, there would be no need for this clumsy double checked locking if you call getInstance less often (i.e. once).

### How many ways you can write Singleton Class in Java?

**Answer :** I know at **least four ways** to implement Singleton pattern in Java

1. Singleton by **synchronizing getInstance()** method
2. Singleton with **public static final field initialized during class loading**.
3. Singleton generated by **static nested class**, also referred as Singleton holder pattern.
4. From **Java 5 on-wards using Enums**

### Q. How to write thread-safe Singleton in Java?

**Answer** : Thread safe Singleton usually refers to write thread safe code which creates one and only one instance of Singleton if called by multiple thread at same time. There are many ways to achieve this like **by using double checked locking technique** as shown above and by **using Enum** or **Singleton initialized by class loader.**

# Q. Difference between Singleton Pattern vs Static Class in Java

**When to use Static Class in place of Singleton in Java**

Indeed there are some situations, where **static classes makes sense** than Singleton. Prime example of this **is java.lang.Math** which is not Singleton, instead a class with all static methods. Here are few situation where I think using static class over Singleton pattern make sense:

1) **If your Singleton is not maintaining any state, and just providing global access to methods, than consider using static class,** as static methods are much faster than Singleton, because of static binding during compile time. But remember its not advised to maintain state inside static class, especially in concurrent environment, where it could lead subtle race conditions when modified parallel by multiple threads without adequate synchronization.

You can also choose to use static method, if you need to combine bunch of utility method together. Anything else, which requires singles access to some resource, should use Singleton design pattern.

## Difference between Singleton vs Static in Java

1) **Static class provides better performance than Singleton pattern**, because static methods are bonded on compile time.

2) One more difference between Singleton and static is, ability to override. **Since static methods in Java cannot be overridden**, they leads to inflexibility. On the other hand, **you can override methods defined in Singleton class by extending it.**

3) **Static classes are hard to mock and consequently hard to test than Singletons**, which are pretty easy to mock and thus easy to test. It’s easier to write JUnit test for Singleton than static classes, because you can pass mock object whenever Singleton is expected, e.g. into constructor or as method arguments.

4) If your requirements needs to maintain state than Singleton pattern is better choice than static class, because

maintaining  state in later case is nightmare and leads to subtle bugs.

5) **Singleton classes can be lazy loaded** if its an heavy object**, but static class doesn't have such advantages and always eagerly loaded.**

6) **Many Dependency Injection framework manages Singleton quite well e.g. Spring, which makes using them very easy.**

### Advantage of Singleton Pattern over Static Class in Java

Main advantage of Singleton over static is that the **former is more object-oriented** than later. **With Singleton, you can use Inheritance and Polymorphism to extend a base class, implement an interface and capable of providing different implementations.**   
If we talk about **java.lang.Runtime**, which is a Singleton in Java, call to **getRuntime() method return different implementations based on different JVM**, but guarantees only one instance per JVM, had java.lang.Runtime a static class, it’s not possible to return different implementation for different JVM.