Need?

* The main idea behind the design patterns is to help you improve your codebases.
  + code readability
  + performance
  + clean code (**easy to write, read and maintain**)
  + redundant code(not useful code)
  + scalability

Creational patterns

* Creational design patterns provide solution to instantiate a object in the best possible way for specific situations.
  + Singleton pattern
  + Factory pattern
  + Prototype pattern

Singleton pattern

* Singleton enables an application to have one and only one instance of a class per JVM.
* Singleton pattern is used for logging, drivers objects, caching and thread pool.
* //This code demonstrates how the Singleton pattern can be used to create a counter to  
  //provide unique sequential numbers, such as might be required for use as primary keys in a  
  //database  
  class Sequence {  
   static int *counter*;  
   static Sequence *obj* = null;  
   private Sequence() {  
   *counter* = 0;  
   }  
   static synchronized Sequence getInstance() {  
   if (*obj* == null) {  
   *obj* = new Sequence();  
   }  
   return *obj*;  
   }  
   static synchronized int getCount(){  
   *counter*++;  
   return *counter*;  
   }  
  }  
  public class DemoSingle {  
   public static void main(String[] args) {  
   Sequence s=Sequence.*getInstance*();  
    
   }  
  }

Factory pattern

* The factory design pattern is used when we have a superclass with multiple sub-classes and based on input, we need to return one of the sub-class.
* This pattern takes out the responsibility of the instantiation of a class from the client program to the factory class.
* abstract class Account {  
   void getCustInfo() {  
   System.*out*.println("Get Cust information");  
   }  
   abstract void calculateInterest();  
  }  
  class Saving extends Account {  
   @Override  
   void calculateInterest() {  
   System.*out*.println("Calculate Interest of Saving");  
   }  
  }  
  class Loan extends Account {  
   @Override  
   void calculateInterest() {  
   System.*out*.println("Calculate Interest of Loan");  
   }  
  }  
  class AccountFactory {  
   public static Account getAccount(String type) {  
   Account obj = null;  
   if (type.equalsIgnoreCase("saving")) {  
   obj = new Saving();  
    
   } else {  
   obj = new Loan();  
   }  
   return obj;  
   }  
  }  
    
  public class DemoFactory {  
   public static void main(String[] args) {  
    
   Account ob1 = AccountFactory.*getAccount*("Loan");  
   ob1.calculateInterest();  
   ob1 = AccountFactory.*getAccount*("saving");  
   ob1.calculateInterest();  
    
   }  
  }

Prototype pattern

* The prototype 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. So this pattern provides a mechanism to copy the original object to a new object and then modify it according to our needs. This pattern uses java cloning to copy the object.
* import java.util.ArrayList;  
    
  class Student implements Cloneable {  
   ArrayList<String> students;  
   Student(){  
   students=new ArrayList<>();  
   }  
   Student( ArrayList<String> students){  
   this.students=students;  
   }  
   void loadData(){  
   students.add("A");  
   students.add("B");  
   students.add("C");  
   students.add("D");  
   }  
   void showData(){  
   for (String s:students){  
   System.*out*.println(s);  
   }  
   }  
   @Override  
   protected Object clone() throws CloneNotSupportedException {  
   ArrayList<String> temp = new ArrayList<String>();  
   for(String s : this.students){  
   temp.add(s);  
   }  
   return new Student(temp);  
    
   }  
  }  
  public class DemoPattern {  
   public static void main(String[] args) throws CloneNotSupportedException {  
   Student st1=new Student();  
   st1.loadData();  
   Student st2=(Student)st1.clone();  
   st1.showData() ;  
   st2.showData();  
   }  
  }

Structural patterns

Structural patterns prescribe the organization of classes and objects. These patterns are  
concerned with how classes inherit from each other or how they are composed from other  
classes

* + Decorator pattern
  + Façade pattern

Decorator pattern

* **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
* We use inheritance or composition to extend the behavior 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.
* interface Car {  
   public void assemble();  
  }  
  class BasicCar implements Car {  
   @Override  
   public void assemble() {  
   System.*out*.print("Basic Car.");  
   }  
  }  
   class CarDecorator implements Car {  
   protected Car car;  
   public CarDecorator(Car c){  
   this.car=c;  
   }  
   @Override  
   public void assemble() {  
   this.car.assemble();  
   }  
  }  
   class SportsCar extends CarDecorator {  
   public SportsCar(Car c) {  
   super(c);  
   }  
   @Override  
   public void assemble(){  
   super.assemble();  
   System.*out*.println(" Adding features of Sports Car.");  
   }  
  }  
   class LuxuryCar extends CarDecorator {  
   public LuxuryCar(Car c) {  
   super(c);  
   }  
   @Override  
   public void assemble(){  
   super.assemble();  
   System.*out*.println(" Adding features of Luxury Car.");  
   }  
  }  
  public class DemoDecorator {  
   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();  
   }  
  }

Facade Pattern

* Facade Pattern is used to help client applications to easily interact with the system. Suppose we have an application with a set of interfaces to use MySql/Oracle database and to generate different types of reports, such as HTML report, PDF report, etc. So we will have a different set of interfaces to work with different types of databases. Now a client application can use these interfaces to get the required database connection and generate reports. But when the complexity increases or the interface behavior names are confusing, the client application will find it difficult to manage it. So we can apply Facade pattern here and provide a wrapper interface on top of the existing interface to help client application.
* So we can apply Facade design pattern here and provide a wrapper interface on top of the existing interface to help client application.
* import java.sql.Connection;  
  class MySQLHelper{  
   public static Connection getMySqlDBConnection(){  
   //get MySql DB connection using connection parameters  
   return null;  
   }  
  }  
  class OracleHelper {  
    
   public static Connection getOracleDBConnection() {  
   //get Oracle DB connection using connection parameters  
   return null;  
   }  
  }  
  class HelperFacade{  
   static Connection getConnection(String type){  
   Connection con=null;  
   if(type.equals("oracle")){  
   con=OracleHelper.*getOracleDBConnection*();  
   }else if(type.equals("mysql")){  
   con=MySQLHelper.*getMySqlDBConnection*();  
   }  
   return con;  
   }  
  }  
  public class DemoFacade {  
   public static void main(String[] args) {  
   Connection con;  
   con=HelperFacade.*getConnection*("oracle");  
   }  
  }

Be**havioral Design Patterns**

* Behavioral patterns provide solution for the better interaction between objects and how to provide lose coupling and flexibility to extend easily.

Chain of Responsibility

* In writing an application of any kind, it often happens that the event generated by one object needs to be handled by another one.
* The chain of responsibility pattern is used to achieve loose coupling in software design where a request from the client is passed to a chain of objects to process them. Then the object in the chain will decide who will be processing the request and whether the request is required to be sent to the next object in the chain or not.
* class Request {  
   private String value;  
   private String description;  
   public Request(String description, String value)  
   {  
   this.description = description;  
   this.value = value;  
   }  
    
   public String getValue()  
   {  
   return value;  
   }  
    
   public String getDescription()  
   {  
   return description;  
   }  
  }  
   abstract class Handler  
  {  
   protected Handler m\_successor;  
   public void setSuccessor(Handler successor)  
   {  
   m\_successor = successor;  
   }  
    
   public abstract void handleRequest(Request request);  
  }  
  class ETC extends Handler  
  {  
   public void handleRequest(Request request)  
   {  
   if (request.getValue().equals("ETC"))  
   { //if request is eligible handle it  
   System.*out*.println(" handled by ETC:");  
   System.*out*.println("\tRequest : " + request.getDescription()  
   + request.getValue());  
   }  
   else  
   {  
   m\_successor.handleRequest(request);  
   }  
   }  
  }  
  class CSE extends Handler  
  {  
   public void handleRequest(Request request)  
   {  
   if (request.getValue().equals("CSE"))  
   { //if request is eligible handle it  
   System.*out*.println(" handled by CSE:");  
   System.*out*.println("\tRequest : " + request.getDescription()  
   + request.getValue());  
   }  
   else  
   {  
   m\_successor.handleRequest(request);  
   }  
   }  
  }  
  public class DemoChain {  
   public static void main(String[] args) {  
   Request r1=new Request("For CSE DEPT","CSE");  
   Request r2=new Request("For ETC DEPT","ETC");  
   Handler h1=new ETC();  
   Handler h2=new CSE();  
   h1.setSuccessor(h2);  
   h1.handleRequest(r1);  
   h1.handleRequest(r2);  
   }  
  }