**Java Design Patterns**

A design pattern is a well-described solution to a common software problem.

Design patterns are already defined and provide an industry-standard approach to solving a recurring problem, so it saves time if we sensibly use the design pattern. There are many Java design patterns that we can use in our Java-based projects.

Using design patterns promotes reusability that leads to more robust and highly maintainable code. It helps in reducing the total cost of ownership (TCO) of the software product.

Since design patterns are already defined, it makes our code easy to understand and debug. It leads to faster development and new members of the team understand it easily.

Java design patterns are divided into three categories –

**Creational**

Creational design patterns provide solutions to instantiate an Object in the best possible way for specific situations.

Singleton Pattern:

--- Ensures that a class has only one instance and provides a global point of access to it.

-- Useful for managaing the shared resources or maintaining single configuration.

**Singleton.java**

package JavaDesignPatterns;  
  
public class Singleton  
{  
 private static Singleton *obj*;  
 private Singleton() // private Constructor  
 {  
 }  
 public static Singleton getInstance()  
 {  
 if(*obj* == null)  
 {  
 *obj* = new Singleton();  
 }  
 return *obj*;  
 }  
 public static void main(String args[])  
 {  
 Singleton instance = Singleton.*getInstance*();  
 System.*out*.println("Singleton instance "+instance);  
 // same object will get created , As it ensures that only one object is created.  
 Singleton in = Singleton.*getInstance*();  
 System.*out*.println("Singleton instance "+in);  
 }  
}

**Factory Desing Pattern**

A Factory Pattern or Factory Method Pattern says that just define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate. In other words, subclasses are responsible to create the instance of the class.

Advantage of Factory Design Pattern

* Factory Method Pattern allows the sub-classes to choose the type of objects to create.
* It promotes the **loose-coupling** by eliminating the need to bind application-specific classes into the code. That means the code interacts solely with the resultant interface or abstract class, so that it will work with any classes that implement that interface or that extends that abstract class.

Usage of Factory Design Pattern

* When a class doesn't know what sub-classes will be required to create
* When a class wants that its sub-classes specify the objects to be created.
* When the parent classes choose the creation of objects to its sub-classes.

**FactoryPattern.java**

package JavaDesignPatterns;  
  
interface Fruit  
{  
 void properties();  
}  
  
class Apple implements Fruit  
{  
 public void properties()  
 {  
 System.*out*.println("Red Colour + Sweet Taste");  
 }  
}  
  
class Orange implements Fruit  
{  
 public void properties()  
 {  
 System.*out*.println("Orange Colour + Tangy Taste");  
 }  
}  
  
class Mango implements Fruit  
{  
 public void properties()  
 {  
 System.*out*.println("Yellow Colour + Sweet Taste");  
 }  
}  
  
public class FactoryPattern  
{  
 public Fruit createFruit(String type)  
 {  
 if(type.equals("Apple"))  
 return new Apple();  
 else if (type.equals("Orange"))  
 return new Orange();  
 else if (type.equals("Mango"))  
 return new Mango();  
 else  
 return (Fruit) new IllegalAccessException("Unknown Fruit Type: "+type);  
 }  
  
 public static void main(String args[])  
 {  
 FactoryPattern obj = new FactoryPattern();  
 Fruit apple = obj.createFruit("Apple");  
 apple.properties();  
 Fruit orange = obj.createFruit("Orange");  
 orange.properties();  
 Fruit mango = obj.createFruit("Mango");  
 mango.properties();  
 }  
}

**Structural Design Pattern (Code reusability and maintainability)**

**Structural design patterns** are concerned with how classes and objects can be composed, to form larger structures.

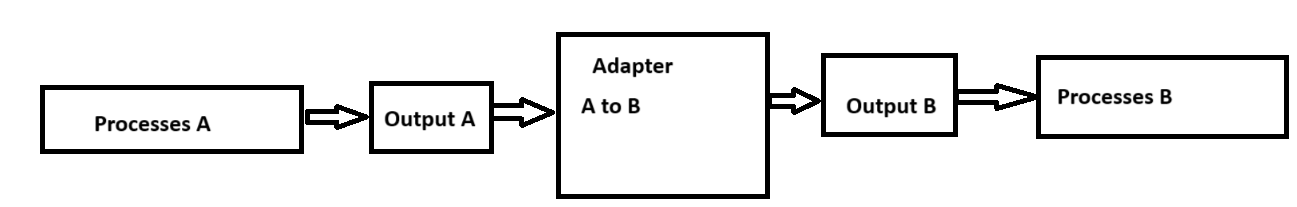
The structural design patterns **simplify the structure by identifying the relationships**.

These patterns focus on, how the classes inherit from each other and how they are composed from other classes.

**Adapter Pattern**

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.



**Two Way Adapter Pattern**

While implementing Adapter pattern, there are two approaches - class adapter and object adapter - however both these approaches produce same result.

**Class Adapter** - This form uses java inheritance and extends the source interface, in our case Socket class.

**Objects Adapter** - This form uses Java Composition and adapter contains the source object.

**Adapter Design Pattern - Class Adapter**

**Volt.java**

package JavaDesignPatterns;  
  
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;  
 }  
}

**Socket.java**

package JavaDesignPatterns;  
  
public class Socket {  
 public Volt getVolt(){  
 return new Volt(120);  
 }  
}

**SocketAdapter.java**

package JavaDesignPatterns;  
  
public interface SocketAdapter {  
  
 public Volt get120Volt();  
  
 public Volt get12Volt();  
  
 public Volt get3Volt();  
}

**SocketImplAdapter**

package JavaDesignPatterns;  
  
public class SocketImplAdapter extends Socket implements SocketAdapter  
{  
 public Volt get120Volt()  
 {  
 return getVolt(); // calling the getVolt() of Socket class and hence it returns Volt object of 120 volts  
  
 }  
  
 public Volt get12Volt()  
 {  
 Volt v = getVolt(); // calling the getVolt() of Socket class and hence it returns Volt object of 120 volts  
 return convert(v,10);// calling covert method to get colt of 12  
 }  
  
 public Volt get3Volt()  
 {  
 Volt v = getVolt();// calling the getVolt() of Socket class and hence it returns Volt object of 120 volts  
 return convert(v,40); // calling covert method to get colt of 3  
 }  
 public Volt convert(Volt v , int i)  
 {  
 return new Volt(v.getVolts()/i);  
 }  
}