**Design patterns are solutions to recurring problems.**

1) Categories Java Design patterns?

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

**Creational patterns**

* Singleton
* Factory
* Abstract Factory
* Prototype
* Builder

**Structural patterns**

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

**Behavioral patterns**Skip Ad

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

**J2EE patterns**

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

**Q) Why do we need Design pattern?**

As Design Patterns are well documented and understood by software architects, designers and developers, then their application within a specific solution will likewise be well understood.

Patterns give a software developer an array of tried and tested solutions to common problems, thus reducing the technical risk to the project by not having to employ a new and untested design, thus saving time and effort during the implementation stage of the software development lifecycle.

**Q) What are the Creational Patterns?**

Creational design patterns are related to the way of creating objects.

This pattern is used to define and describe how objects are created at class instantiation time

**Q) What Is Factory Pattern?**

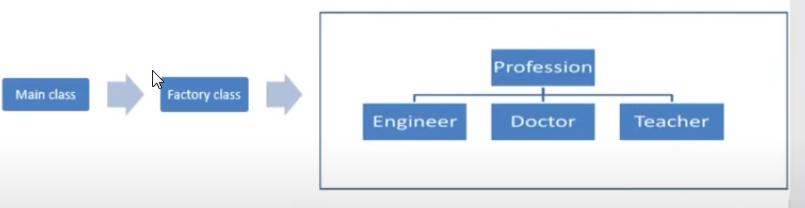
In the Factory pattern, we don't expose the creation logic to the client and refer the created object using a standard interface.

The Factory Pattern is also known as Virtual Constructor.(Because you don’t know how the object is created ).

Steps:

1) Create main class which call factory class.

2) Factory class returns required class instance



**public** **interface** Profession {

**void** print();

}

**public** **class** Doctor **implements** Profession {

@Override

**public** **void** print() {

System.***out***.println("In Print of Doctor class");

}

**public** **class** Engineer **implements** Profession {

@Override

**public** **void** print() {

System.***out***.println("In Print of Engineer class");

}

}

**public** **class** Teacher **implements** Profession {

@Override

**public** **void** print() {

System.***out***.println("In Print of Teacher class");

}

}

**public** **class** ProfessionFactory {

**public** Profession getProfession(String typeOfProfession) {

**if** (typeOfProfession == **null**) {

**return** **null**;

}

**if** (typeOfProfession.equalsIgnoreCase("Doctor")) {

**return** **new** Doctor();

} **else** **if** (typeOfProfession.equalsIgnoreCase("Engineer")) {

**return** **new** Engineer();

} **else** **if** (typeOfProfession.equalsIgnoreCase("Teacher")) {

**return** **new** Teacher();

}

**return** **null**;

}

}

**public** **class** FactoryPatternMainClass {

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

ProfessionFactory professionFactory = **new** ProfessionFactory();

Profession doc = professionFactory.getProfession("Doctor");

doc.print();

}

}

**Q) What Is Abstract Factory Pattern?**

This factory is also called as factory of factories.

Abstract Factory lets a class returns a factory of classes.

So, this is the reason that Abstract Factory Pattern is one level higher than the Factory Pattern.

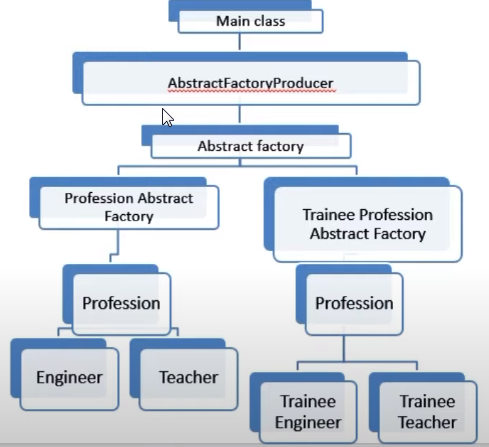
An Abstract Factory Pattern is also known as Kit.

**Steps:**

1) Create main class which call Factory Producer class.

2) Factory Producer creates instance of factory class.

3) Factory class returns required class instance



**public** **interface** Profession {

**void** print();

}

**public** **class** Teacher **implements** Profession{

@Override

**public** **void** print() {

System.***out***.println("In Print of Teacher class");

}

}

**public** **class** Engineer **implements** Profession{

@Override

**public** **void** print() {

System.***out***.println("In Print of Engineer class");

}

}

**public** **class** TraineeTeacher **implements** Profession{

@Override

**public** **void** print() {

System.***out***.println("In Print of Trainee Teacher class");

}

}

**public** **class** TraineeEngineer **implements** Profession{

@Override

**public** **void** print() {

System.***out***.println("In Print of Trainee Engineer class");

}

}

**public** **abstract** **class** AbstractFactory {

**abstract** Profession getProfession(String typeOfProfession);

}

**public** **class** ProfessionAbstractFactory **extends** AbstractFactory{

@Override

**public** Profession getProfession(String typeOfProfession){

**if**(typeOfProfession == **null**){

**return** **null**;

}

**else** **if**(typeOfProfession.equalsIgnoreCase("Engineer")){

**return** **new** Engineer();

} **else** **if**(typeOfProfession.equalsIgnoreCase("Teacher")){

**return** **new** Teacher();

}

**return** **null**;

}

}

**public** **class** TraineeProfessionAbstractFactory **extends** AbstractFactory{

@Override

**public** Profession getProfession(String typeOfProfession){

**if**(typeOfProfession == **null**){

**return** **null**;

}

**else** **if**(typeOfProfession.equalsIgnoreCase("Engineer")){

**return** **new** TraineeEngineer();

} **else** **if**(typeOfProfession.equalsIgnoreCase("Teacher")){

**return** **new** TraineeTeacher();

}

**return** **null**;

}

}

// this is factory of factory , gives u factory instance which wil in turn give u required class object

**public** **class** AbstractFactoryProducer {

**public** **static** AbstractFactory getProfession(**boolean** isTrainee) {

**if**(isTrainee) {

**return** **new** TraineeProfessionAbstractFactory();

}

**else** {

**return** **new** ProfessionAbstractFactory();

}

}

}

**public** **class** FactoryPatternMainClass {

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

AbstractFactory abstractFactory = AbstractFactoryProducer.*getProfession*(**true**);

Profession engg = abstractFactory.getProfession("Engineer");

engg.print();

}

}

If we pass true

O/P: - In Print of **Trainee Engineer** class

If we pass false

O/P: In Print of **Engineer** class

**Q) What Is Singleton Design Pattern?**

Singleton pattern is one of the simplest design patterns in Java.

This pattern involves a single class which is responsible to create an object while making sure that only single object gets created.

This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

Its create only once instance of class in memory .

**public** **class** SingeltonClass {

**private** **static** SingeltonClass *singletonInstance* = **null**;

**private** SingeltonClass() {

}

// Get the only object available

**public** **static** SingeltonClass getInstance() {

**if** (**null** == *singletonInstance*) {

*singletonInstance* = **new** SingeltonClass();

}

**return** *singletonInstance*;

}

**public** **void** simpleMethod() {

System.***out***.println("hashcode of singelton object" + *singletonInstance*);

}

}

**public** **class** MainClass {

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

SingeltonClass singeltonObject = SingeltonClass.*getInstance*();

singeltonObject.simpleMethod();

SingeltonClass singeltonObject2 = SingeltonClass.*getInstance*();

singeltonObject2.simpleMethod();

Teacher teacher = **new** Teacher();

System.***out***.println(teacher);

Teacher teacher2 = **new** Teacher();

System.***out***.println(teacher2);

}

}

O/P:

hashcode of singelton objectsingelton.design.pattern.SingeltonClass@**15db9742**

hashcode of singelton objectsingelton.design.pattern.SingeltonClass@**15db9742**

factory.pattern.Teacher@6d06d69c

factory.pattern.Teacher@7852e922

If we notice here singleton class having same hash code where no singleton class have different hash code.

**Prototype Design Pattern**

Prototype Pattern says that cloning of an existing object instead of creating new one and can also be customized as per the requirement.

This pattern should be followed, if the cost of creating a new object is expensive and resource intensive. For example it required data base call or required to o much of processing that will take a lot of memory.

**class** Sheep **implements** Cloneable {

**protected** String name;

**protected** String category;

**public** Sheep(String name, String category) {

**this**.name = name;

**this**.category = category;

}

**public** Sheep(String name) {

**this**.name = name;

**this**.category = "Mountain Sheep";

}

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

**return** **super**.clone();

}

}

**public** **class** PrototypePattern {

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

Sheep original = **new** Sheep("Jolly");

System.***out***.println(original.getName()); // Jolly

System.***out***.println(original.getCategory()); // Mountain Sheep

// Clone and modify what is required

Sheep cloned;

**try** {

cloned = (Sheep) original.clone();

cloned.setName("Dolly");

System.***out***.println(cloned.getName()); // Dolly

System.***out***.println(cloned.getCategory()); // Mountain sheep

} **catch** (CloneNotSupportedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

O/P:

Jolly

Mountain Sheep

Dolly

Mountain Sheep

Q) What Is Builder Design Pattern?

Builder Pattern says that "construct a complex object from simple objects using step-by-step approach" .

**class** Burger {

**protected** **int** size;

**protected** **boolean** cheese = **false**;

**protected** **boolean** pepperoni = **false**;

**protected** **boolean** lettuce = **false**;

**protected** **boolean** tomato = **false**;

**public** Burger(BurgerBuilder builder) {

**this**.size = builder.size;

**this**.cheese = builder.cheese;

**this**.pepperoni = builder.pepperoni;

**this**.lettuce = builder.lettuce;

**this**.tomato = builder.tomato;

}

}

**class** BurgerBuilder {

**public** **int** size;

**public** **boolean** cheese = **false**;

**public** **boolean** pepperoni = **false**;

**public** **boolean** lettuce = **false**;

**public** **boolean** tomato = **false**;

**public** BurgerBuilder(**int** size) {

**this**.size = size;

}

**public** BurgerBuilder addPepperoni() {

**this**.pepperoni = **true**;

**return** **this**;

}

**public** BurgerBuilder addLettuce() {

**this**.lettuce = **true**;

**return** **this**;

}

**public** BurgerBuilder addCheese() {

**this**.cheese = **true**;

**return** **this**;

}

**public** BurgerBuilder addTomato() {

**this**.tomato = **true**;

**return** **this**;

}

**public** Burger build() {

**return** **new** Burger(**this**);

}

}

**public** **class** BuilderPattern {

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

Burger burger = (**new** BurgerBuilder(14))

.addPepperoni()

.addLettuce()

.addTomato()

.addCheese()

.build();

System.***out***.println(burger);

}

}

O/P

Burger [size=14, cheese=true, pepperoni=true, lettuce=true, tomato=true]