**Design Pattern Interview Questions**

Table of Contents

[**1.** **What are the design principles in java [OO Design Principles]?** 2](#_Toc483430687)

[**2.** **Builder Design Pattern in Java?** 7](#_Toc483430688)

[**Example** 8](#_Toc483430689)

[**Advantages:** 10](#_Toc483430690)

[**Disadvantages:** 10](#_Toc483430691)

[**When to use Builder Design pattern in Java** 10](#_Toc483430692)

[**3.** **Factory Design Pattern in Java?** 11](#_Toc483430693)

[**Example of static factory method in JDK** 11](#_Toc483430694)

[**Problem which is solved by Factory method Pattern in Java** 12](#_Toc483430695)

[**Example** 12](#_Toc483430696)

[**Advantages** 14](#_Toc483430697)

[**4.** **Abstract Factory Design Pattern in Java?** 16](#_Toc483430698)

[**Package Diagram** 17](#_Toc483430699)

[**Sequence Diagram** 18](#_Toc483430700)

[**Implementation in code** 18](#_Toc483430701)

[**5.** **Prototype Design Pattern in Java?** 26](#_Toc483430702)

[**Problem Statement** 26](#_Toc483430703)

[**Implementation of Prototype Design Pattern** 27](#_Toc483430704)

# **What are the design principles in java [OO Design Principles]?**

DRY (Don't repeat yourself) **–** avoids duplication in code

* + Don't write duplicate code, instead use Abstraction to abstract common things in one place.
  + If you have block of code in more than two place consider making it a separate method, or if you use a hard-coded value more than one time make them public final constant.
  + Benefit of this Object oriented design principle is in maintenance.
  + It's important not to abuse it, duplication is not for code, but for functionality.
  + It means, if you used common code to validate OrderID and SSN it doesn’t mean they are same or they will remain same in future.
  + By using common code for two different functionality or thing you closely couple them forever and when your OrderID changes its format, your SSN validation code will break.
  + So beware of such coupling and just don’t combine anything which uses similar code but are not related.

Encapsulate What Changes **–** hides implementation details, helps in maintenance

* + Only one thing is constant in software field and that is "Change", so encapsulate the code you expect or suspect to be changed in future.
  + Benefit of this OOPS Design principle is that It’s easy to test and maintain proper encapsulated code.
  + If you are coding in Java then follow principle of making variable and methods private by default and increasing access step by step e.g. from private to protected and not public.
  + Several of design pattern in Java uses Encapsulation, Factory design pattern is one example of Encapsulation which encapsulate object creation code and provides flexibility to introduce new product later with no impact on existing code.

Open Closed Design Principle **–** open for extension, closed for modification

* + Classes, methods or functions should be Open for extension (new functionality) and Closed for modification.
  + This is another beautiful SOLID design principle, which prevents some-one from changing already tried and tested code.
  + Ideally if you are adding new functionality only than your code should be tested and that's the goal of Open Closed Design principle.
  + By the way, Open Closed principle is "O" from SOLID acronym.

Single Responsibility Principle (SRP) **–** one class should do one thing and do it well

* + Single Responsibility Principle is another SOLID design principle, and represent “S" on SOLID acronym.
  + As per SRP, there should not be more than one reason for a class to change, or a class should always handle single functionality.
  + If you put more than one functionality in one Class in Java it introduce coupling between two functionality and even if you change one functionality there is chance you broke coupled functionality, which require another round of testing to avoid any surprise on production environment.

Dependency Injection or Inversion principle **–** don’t ask, let framework give to you

* + Don't ask for dependency it will be provided to you by framework.
  + This has been very well implemented in spring framework, beauty of this design principle is that any class which is injected by DI framework is easy to test with mock object and easier to maintain because object creation code is centralized in framework and client code is not littered with that.
  + There are multiple ways to implemented Dependency injection like using byte code instrumentation which some AOP (Aspect Oriented programming) framework like AspectJ does or by using proxies just like used in spring.
  + It represent "D" on SOLID acronym.

Favor Composition over Inheritance **–** Code reuse without cost of inflexibility

* + Always favor composition over inheritance, if possible.
  + Some of you may argue this, but I found that Composition is lot more flexible than Inheritance.
  + Composition allows to change behavior of a class at run-time by setting property during run-time and by using Interfaces to compose a class we use polymorphism which provides flexibility of to replace with better implementation any time.

Liskov Substitution Principle (LSP) **–** Subtype must be substitutable for super type

* + According to Liskov Substitution Principle, Subtypes must be substitutable for super type i.e. methods or functions which uses super class type must be able to work with object of sub class without any issue".
  + LSP is closely related to Single responsibility principle and Interface Segregation Principle.
  + If a class has more functionality than subclass might not support some of the functionality, and does violated LSP.
  + In order to follow LSP SOLID design principle, derived class or sub class must enhance functionality, but not reduce them. LSP represent “L" on SOLID acronym.

Interface Segregation principle (ISP) **–** avoid monolithic interface, reduce pain on client side

* + Interface Segregation Principle stats that, a client should not implement an interface, if it doesn't use that.
  + This happens mostly when one interface contains more than one functionality, and client only need one functionality and not other.
  + Interface design is tricky job because once you release your interface you cannot change it without breaking all implementation.
  + Another benefit of this design principle in Java is, interface has disadvantage to implement all method before any class can use it so having single functionality means less method to implement.

Programming for Interface not implementation **–** helps in maintenance, improves flexibility

* + Always program for interface and not for implementation this will lead to flexible code which can work with any new implementation of interface.
  + So use interface type on variables, return types of method or argument type of methods in Java.

Delegation principle –don’t do all things by yourself, delegate it

* + Don't do all stuff by yourself, delegate it to respective class.
  + Classical example of delegation design principle is equals() and hashCode() method in Java.
  + In order to compare two object for equality we ask class itself to do comparison instead of Client class doing that check.
  + Benefit of this design principle is no duplication of code and pretty easy to modify behavior.

# **Builder Design Pattern in Java?**

* + Builder pattern is a creational design pattern it means its solves problem related to object creation.
  + Constructors in Java are used to create object and can take parameters required to create object.
  + Problem starts when an Object can be created with lot of parameters, some of them may be mandatory and others may be optional.
  + Consider a class which is used to create Cake, now you need number of item like egg, milk, flour to create cake. Many of them are mandatory and some of them are optional like cherry, fruits etc.
  + If we are going to have overloaded constructor for different kind of cake then there will be many constructor and even worst they will accept many parameter.
  + Problems**:**
    - Too many constructors to maintain.
    - error prone because many fields has same type e.g. sugar and and butter are in cups so instead of 2 cup sugar if you pass 2 cup butter, your compiler will not complain but will get a buttery cake with almost no sugar with high cost of wasting butter.
  + You can partially solve this problem by creating Cake and then adding ingredients but that will impose another problem of leaving Object on inconsistent state during building, ideally cake should not be available until it’s created.
  + Both of these problem can be solved by using Builder design pattern in Java.
  + Builder design pattern not only improves readability but also reduces chance of error by adding ingredients explicitly and making object available once fully constructed.

## **Example**

* + Make a static nested class called Builder inside the class whose object will be build by Builder. In this example it’s Cake.
  + Builder class will have exactly same set of fields as original class.
  + Builder class will expose method for adding ingredients e.g. sugar() in this example. each method will return same Builder object. Builder will be enriched with each method call.
  + Builder.build() method will copy all builder field values into actual class and return object of Item class.
  + Item class (class for which we are creating Builder) should have private constructor to create its object from build() method and prevent outsider to access its constructor.

**public** **class** BuilderPatternExample {    
    **public** **static** **void** main(**String** args[]) {        
        *//Creating object using Builder pattern in java*  
        Cake whiteCake = **new**Cake.Builder().sugar(1).butter(0.5).  eggs(2).vanila(2).flour(1.5). bakingpowder(0.75).milk(0.5).build();  
        
        *//Cake is ready to eat :)*  
        **System**.out.println(whiteCake);  
    }  
 }  
 **class** Cake {  
    **private** **final** **double** sugar;   *//cup*  
    **private** **final** **double** butter;  *//cup*  
    **private** **final** **int** eggs;  
    **private** **final** **int** vanila;     *//spoon*  
    **private** **final** **double** flour;   *//cup*  
    **private** **final** **double** bakingpowder; *//spoon*  
    **private** **final** **double** milk;  *//cup*  
    **private** **final** **int** cherry;  
  
    **public** **static** **class** Builder {  
  
        **private** **double** sugar;   *//cup*  
        **private** **double** butter;  *//cup*  
        **private** **int** eggs;  
        **private** **int** vanila;     *//spoon*  
        **private** **double** flour;   *//cup*  
        **private** **double** bakingpowder; *//spoon*  
        **private** **double** milk;  *//cup*  
        **private** **int** cherry;  
  
        *//builder methods for setting property*  
        **public** Builder sugar(**double** cup){**this**.sugar = cup; **return** **this**; }  
        **public** Builder butter(**double** cup){**this**.butter = cup; **return** **this**; }  
        **public** Builder eggs(**int** number){**this**.eggs = number; **return** **this**; }  
        **public** Builder vanila(**int** spoon){**this**.vanila = spoon; **return** **this**; }  
        **public** Builder flour(**double** cup){**this**.flour = cup; **return** **this**; }  
        **public** Builder bakingpowder(**double** spoon){**this**.sugar = spoon; **return** **this**; }  
        **public** Builder milk(**double** cup){**this**.milk = cup; **return** **this**; }  
        **public** Builder cherry(**int** number){**this**.cherry = number; **return** **this**; }        
        
        *//return fully build object*  
        **public** Cake build() {  
            **return** **new** Cake(**this**);  
        }  
    }  
  
    *//private constructor to enforce object creation through builder*  
    **private** Cake(Builder builder) {  
        **this**.sugar = builder.sugar;  
        **this**.butter = builder.butter;  
        **this**.eggs = builder.eggs;  
        **this**.vanila = builder.vanila;  
        **this**.flour = builder.flour;  
        **this**.bakingpowder = builder.bakingpowder;  
        **this**.milk = builder.milk;  
        **this**.cherry = builder.cherry;         
    }

    @**Override**  
    **public** **String** toString() {  
        **return** "Cake{" + "sugar=" + sugar + ", butter=" + butter + ", eggs=" + eggs + ", vanila=" + vanila + ", flour=" + flour + ", bakingpowder=" + bakingpowder + ", milk=" + milk + ", cherry=" + cherry + '}';  
    }     
}  
  
**Output:**  
Cake{sugar=0.75, butter=0.5, eggs=2, vanila=2, flour=1.5, bakingpowder=0.0, milk=0.5, cherry=0}

## **Advantages:**

* + More maintainable if number of fields required to create object is more than 4 or 5.
  + Less error-prone as user will know what they are passing because of explicit method call.
  + More robust as only fully constructed object will be available to client.

## **Disadvantages:**

* + Verbose and code duplication as Builder needs to copy all fields from Original or Item class.

## **When to use Builder Design pattern in Java**

* + Builder Design pattern is a creational pattern and should be used when number of parameter required in constructor is more than manageable usually 4 or at most 5.
  + Don't confuse with Builder and Factory pattern there is an obvious difference between Builder and Factory pattern, as Factory can be used to create different implementation of same interface but Builder is tied up with its Container class and only returns object of Outer class.

# **Factory Design Pattern in Java?**

* + 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
  + 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.
  + In GOF pattern list Factory pattern is listed as Creation design pattern.
  + Factory should be an interface and clients first either creates factory or get factory which later used to create objects.

## **Example of static factory method in JDK**

* + 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.

## **Problem which is solved by Factory method Pattern in Java**

* + 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 cannot 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.

## **Example**

* + 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.

**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());

       }

}

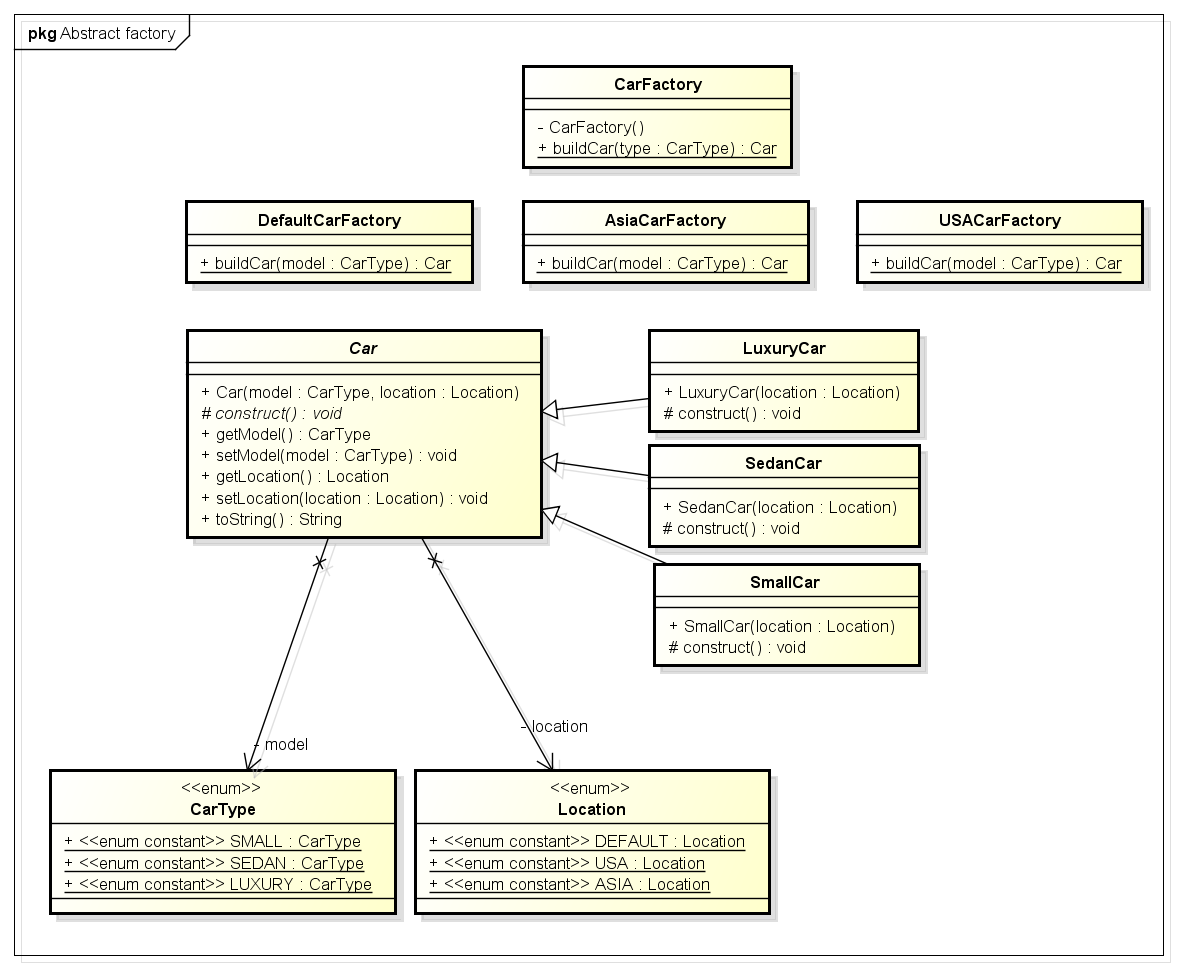
## **Advantages**

* + Factory method design pattern decouples the calling class from the target class, which result in less coupled and highly cohesive code?
  + 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.
  + 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.
  + 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.
  + 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());
  + Since static factory method have return type as Interface, it allows you to replace implementation with better performance version in newer release.
  + 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.
  + Factory method pattern is also recommended by Joshua Bloch in Effective Java.
  + Factory method pattern offers alternative way of creating object.
  + Factory pattern can also be used to hide information related to creation of object.

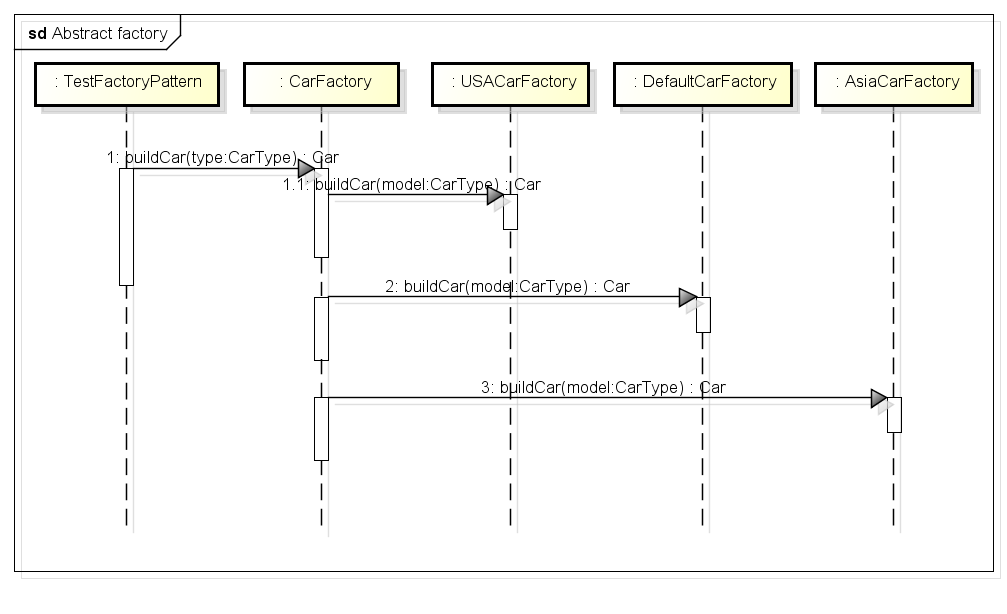
# **Abstract Factory Design Pattern in Java?**

* + Abstract factory pattern is yet another creational design pattern and is considered as another layer of abstraction over factory pattern
  + Imagine if our car maker decide to go global. Becoming global will require to enhance the system to support different car making styles for different countries. For example we will consider USA, Asia and default for all other countries.
  + First of all, we need car factories in each location specified in problem statement. i.e. USACarFactory, AsiaCarFactory and DefaultCarFactory.
  + Now, our application should be smart enough to identify the location where is being used, so we should be able to use appropriate car factory without even knowing which car factory implementation will be used internally. This also saves us from someone calling wrong factory for a particular location.
  + So basically, we need another layer of abstraction which will identify the location and internally use correct car factory implementation without even giving a single hint to user. This is exactly the problem, which abstract factory pattern is used to solve.
  + Let’s see how abstract factory solve above issue in form of design:

## **Package Diagram**



## **Sequence Diagram**



## **Implementation in code**

* + Lets write all separate car factories for different locations. Begin with Car.java class

Car.java

public abstract class Car {

public Car(CarType model, Location location){

this.model = model;

this.location = location;

}

protected abstract void construct();

private CarType model = null;

private Location location = null;

public CarType getModel() {

return model;

}

public void setModel(CarType model) {

this.model = model;

}

public Location getLocation() {

return location;

}

public void setLocation(Location location) {

this.location = location;

}

@Override

public String toString() {

return ”Model- “ + model + “ built in “ +location;

}

}

* + This adds extra work of creating another enum for storing different locations.

Location.java

public enum Location {

DEFAULT, USA, ASIA

}

* + All car types will also have additional location property. I am writing only for luxury car. Same follows for small and sedan also.

LuxuryCar.java

public class LuxuryCar extends Car

{

public LuxuryCar(Location location)

{

super(CarType.LUXURY, location);

construct();

}

@Override

protected void construct() {

System.out.println(“luxury car”);

//add accessories

}

}

* + So for we have created basic classes. Now lets have different car factories.

AsiaCarFactory.java

public class AsiaCarFactory

{

public static Car buildCar(CarType model)

{

Car car = null;

switch (model)

{

case SMALL:

car = new SmallCar(Location.ASIA);

break;

case SEDAN:

car = new SedanCar(Location.ASIA);

break;

case LUXURY:

car = new LuxuryCar(Location.ASIA);

break;

default:

//throw some exception

break;

}

return car;

}

}

DefaultCarFactory.java

public class DefaultCarFactory

{

public static Car buildCar(CarType model)

{

Car car = null;

switch (model)

{

case SMALL:

car = new SmallCar(Location.DEFAULT);

break;

case SEDAN:

car = new SedanCar(Location.DEFAULT);

break;

case LUXURY:

car = new LuxuryCar(Location.DEFAULT);

break;

default:

//throw some exception

break;

}

return car;

}

}

USACarFactory.java

public class USACarFactory

{

public static Car buildCar(CarType model)

{

Car car = null;

switch (model)

{

case SMALL:

car = new SmallCar(Location.USA);

break;

case SEDAN:

car = new SedanCar(Location.USA);

break;

case LUXURY:

car = new LuxuryCar(Location.USA);

break;

default:

//throw some exception

break;

}

return car;

}

}

* + Well, now we have all 3 different Car factories. Now, we have to abstract the way these factories are accessed. Lets see how?

public class CarFactory

{

private CarFactory() {

//Prevent instantiation

}

public static Car buildCar(CarType type)

{

Car car = null;

Location location = Location.ASIA; //Read location property somewhere from configuration

//Use location specific car factory

switch(location)

{

case USA:

car = USACarFactory.buildCar(type);

break;

case ASIA:

car = AsiaCarFactory.buildCar(type);

break;

default:

car = DefaultCarFactory.buildCar(type);

}

return car;

}

}

* + We are done with writing code. Now lets test what we have written till now.

public class TestFactoryPattern

{

public static void main(String[] args)

{

System.out.println(CarFactory.buildCar(CarType.SMALL));

System.out.println(CarFactory.buildCar(CarType.SEDAN));

System.out.println(CarFactory.buildCar(CarType.LUXURY));

}

}

Output: (Default location is Asia)

Building small car

Model- SMALL built in ASIA

Building sedan car

Model- SEDAN built in ASIA

Building luxury car

Model- LUXURY built in ASIA

# **Prototype Design Pattern in Java?**

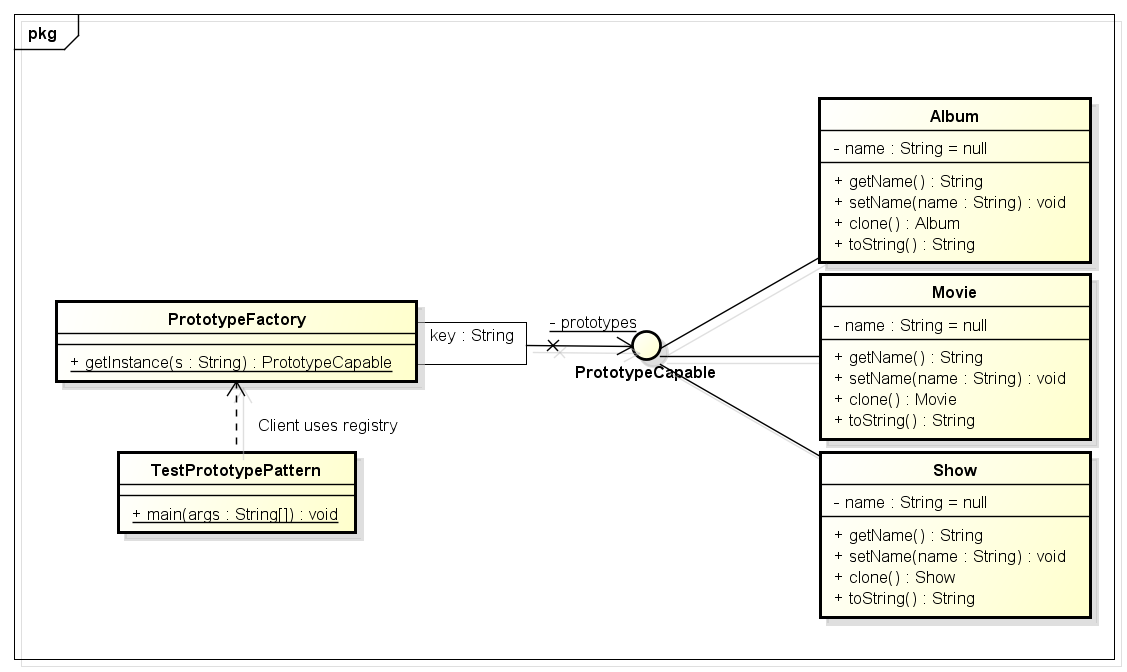
A prototype is a template of any object before the actual object is constructed. In java also, it holds the same meaning. Prototype design pattern is used in scenarios where application needs to create a number of instances of a class, which has almost same state or differs very little.

In this design pattern, an instance of actual object (i.e. prototype) is created on starting, and thereafter whenever a new instance is required, this prototype is cloned to have another instance. The main advantage of this pattern is to have minimal instance creation process which is much costly than cloning process.

## **Problem Statement**

Let’s understand this pattern using an example. I am creating an entertainment application that will require instances of Movie, Album and Show classes very frequently. I do not want to create their instances every time as it is costly. So, I will create their prototype instances, and every time when i will need a new instance, I will just clone the prototype.

## **Implementation of Prototype Design Pattern**

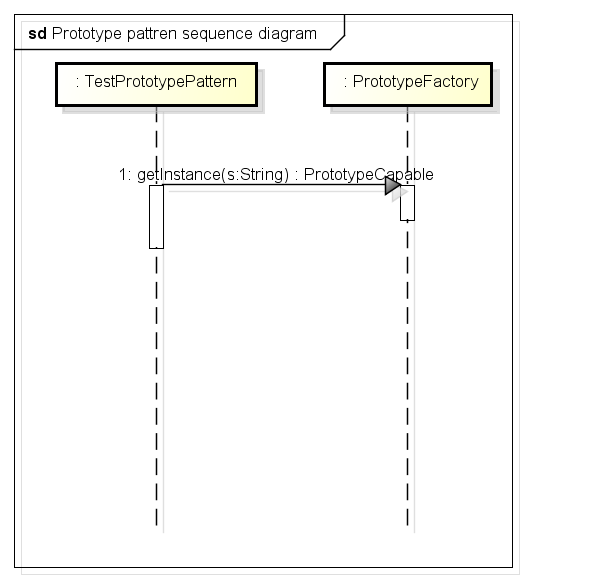


Above class diagram explains the necessary classes and their relationship.

Only one interface, “PrototypeCapable” is new addition in solution. The reason to use this interface is broken behavior of Cloneable interface. This interface helps in achieving following goals:

1. Ability to clone prototypes without knowing their actual types
2. Provides a type reference to be used in registry

Their workflow will look like this:



**PrototypeCapable.java**

|  |
| --- |
| package com.howtodoinjava.prototypeDemo.contract;    public interface PrototypeCapable extends Cloneable  {      public PrototypeCapable clone() throws CloneNotSupportedException;  } |

**Movie.java, Album.java and Show.java**

|  |
| --- |
| package com.howtodoinjava.prototypeDemo.model;    import com.howtodoinjava.prototypeDemo.contract.PrototypeCapable;    public class Movie implements PrototypeCapable  {      private String name = null;      public String getName() {          return name;      }      public void setName(String name) {          this.name = name;      }      @Override      public Movie clone() throws CloneNotSupportedException {          System.out.println("Cloning Movie object..");          return (Movie) super.clone();      }      @Override      public String toString() {          return "Movie";      }  }    public class Album implements PrototypeCapable  {      private String name = null;      public String getName() {          return name;      }      public void setName(String name) {          this.name = name;      }      @Override      public Album clone() throws CloneNotSupportedException {          System.out.println("Cloning Album object..");          return (Album) super.clone();      }      @Override      public String toString() {          return "Album";      }  }    public class Show implements PrototypeCapable  {      private String name = null;      public String getName() {          return name;      }      public void setName(String name) {          this.name = name;      }      @Override      public Show clone() throws CloneNotSupportedException {          System.out.println("Cloning Show object..");          return (Show) super.clone();      }      @Override      public String toString() {          return "Show";      }  } |

**PrototypeFactory.java**

|  |
| --- |
| package com.howtodoinjava.prototypeDemo.factory;    import com.howtodoinjava.prototypeDemo.contract.PrototypeCapable;  import com.howtodoinjava.prototypeDemo.model.Album;  import com.howtodoinjava.prototypeDemo.model.Movie;  import com.howtodoinjava.prototypeDemo.model.Show;    public class PrototypeFactory  {      public static class ModelType      {          public static final String MOVIE = "movie";          public static final String ALBUM = "album";          public static final String SHOW = "show";      }      private static java.util.Map<String , PrototypeCapable> prototypes = new java.util.HashMap<String , PrototypeCapable>();        static      {          prototypes.put(ModelType.MOVIE, new Movie());          prototypes.put(ModelType.ALBUM, new Album());          prototypes.put(ModelType.SHOW, new Show());      }        public static PrototypeCapable getInstance(final String s) throws CloneNotSupportedException {          return ((PrototypeCapable) prototypes.get(s)).clone();      }  } |

**TestPrototypePattern**

|  |
| --- |
| package com.howtodoinjava.prototypeDemo.client;    import com.howtodoinjava.prototypeDemo.factory.PrototypeFactory;  import com.howtodoinjava.prototypeDemo.factory.PrototypeFactory.ModelType;    public class TestPrototypePattern  {      public static void main(String[] args)      {          try          {              String moviePrototype  = PrototypeFactory.getInstance(ModelType.MOVIE).toString();              System.out.println(moviePrototype);                String albumPrototype  = PrototypeFactory.getInstance(ModelType.ALBUM).toString();              System.out.println(albumPrototype);                String showPrototype  = PrototypeFactory.getInstance(ModelType.SHOW).toString();              System.out.println(showPrototype);            }          catch (CloneNotSupportedException e)          {              e.printStackTrace();          }      }  } |

When you run the client code, following is the output.

Cloning Movie object..

Movie

Cloning Album object..

Album

Cloning Show object..

Show