HowToDoInJava

Prototype design pattern in Java

Last Updated: August 22, 2021 By: Lokesh Gupta Creational Patterns Design Patterns

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.

Table of Contents

> Please ensure that you want to deep clone or shallow clone your prototype because both will have different behavior on runtime. If deep copy is needed, you can use a good technique given here using in memory serialization.

Prototype pattern – Participants

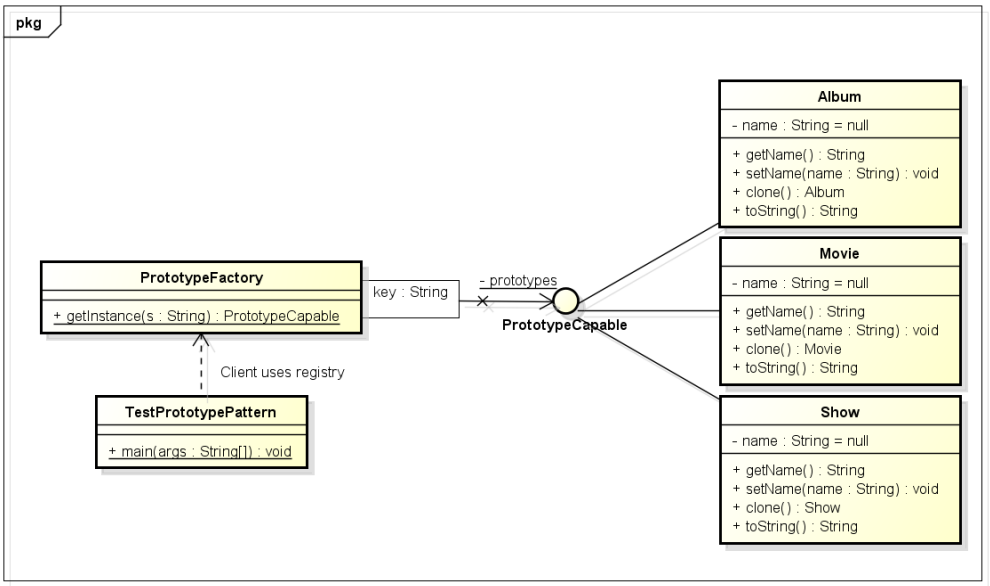
- **Prototype** : This is the prototype of actual object as discussed above.
- **Prototype registry** : This is used as registry service to have all prototypes accessible using simple string parameters.
- **Client** : Client will be responsible for using registry service to access prototype instances.

Problem Statement

Lets 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 everytime as it is costly. So, I will create their prototype instances, and everytime when i will need a new instance, I will just clone the prototype.

Prototype pattern example – Implementation

Lets start by creating class diagram.



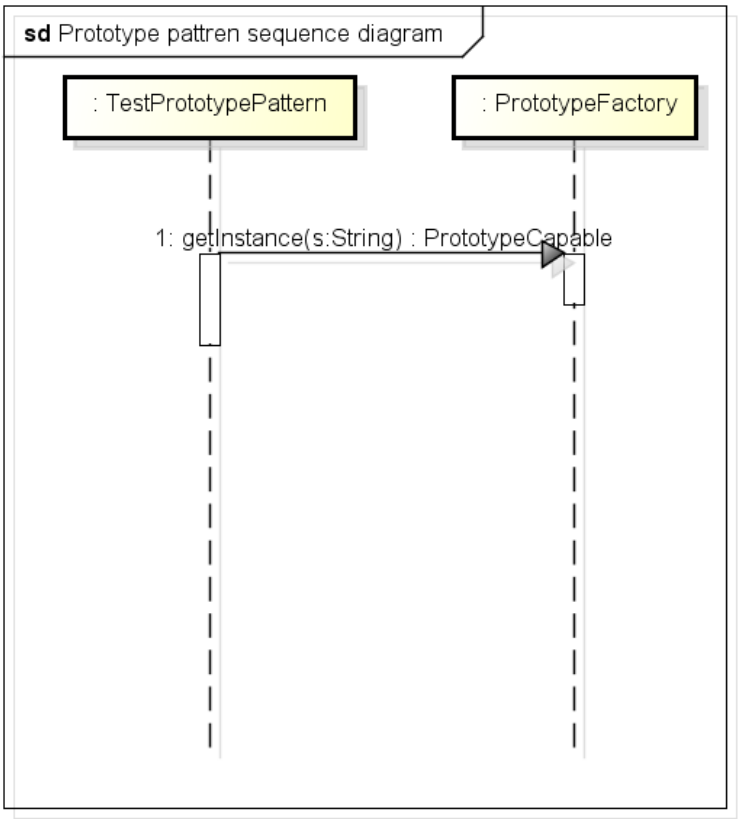
Above class diagram explains the necessary classes and their relationship.

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:

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

Their workflow will look like this:



Lets hit the keyboard and compose these classes.

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.ut

  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 CloneNotSuppor
    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 movieProtoype  = PrototypeFactory.getInstance(ModelType.MOVIE).toStr
      System.out.println(movieProtoype);

      String albumPrototype  = PrototypeFactory.getInstance(ModelType.ALBUM).toStr
      System.out.println(albumPrototype);

      String showPrototype  = PrototypeFactory.getInstance(ModelType.SHOW).toStrin
      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
```

I hope, you liked this post on **Java prototype pattern example**. If any question, drop a comment.

Download source code

Happy Learning !!

Recommended Reading:

1. Builder Design Pattern
2. Decorator Design Pattern in Java
3. Adapter Design Pattern in Java
4. Chain of Responsibility Design Pattern
5. Template Method Design Pattern
6. Strategy Design Pattern
7. Facade Design Pattern
8. Flyweight Design Pattern
9. Mediator Design Pattern
0. State Design Pattern

29 thoughts on “Prototype design pattern in Java”

Yovel

March 1, 2020 at 7:11 pm

Why do we have to create new 3 objects (movie, album, show) when loading PrototypeFactory?
btw everything else is great.

Reply

Batman

March 3, 2020 at 1:15 pm

because movie, album, and show are different classes and when asked they should return their own instance. what can be done here to make sense is that instead of creating the movie, album and show with a default constructor, A parameterized constructor can be used with some default values.

Reply

Vikram

July 14, 2019 at 6:22 pm

Why we are returning returned object's clone , instead of the object itself?

Reply

Yovel

March 4, 2020 at 12:43 am

Because the idea of the pattern is to create serval instances of objects using the cloning technique instead of using an instance creation. We want serval objects of these prototype templates: Movie, Album, and Show.

Reply

Madhu

July 15, 2018 at 3:25 pm

Thank you so much, your design pattern tutorials helped me a lot in understanding design patters.

Reply

Kris blue

September 8, 2016 at 7:49 pm

```
public interface PrototypeCapable extends Cloneable
```

extends or implements?

Reply

Lokesh Gupta

September 8, 2016 at 10:00 pm

> An interface extends another interface… not implements it. You probably missed the “interface” declaration.
>
> Reply

Naren

September 30, 2015 at 5:42 am

> Your main method should be demonstrating something like
>
> ```
> PrototypeFactory myCopyMaker = new PrototypeFactory();
>
> Movie myMovie = new Movie();
> myMovie.setName("Red Planet");
>
> Movie clonedMovie = myCopyMaker.getInstance(myMovie);
>
> //Verify myMovie and clonedMovie have the same names but they are different
> ```
>
> You need to change your PrototypeFactory class to spit out the copy of given PrototypeCapable object type.
>
> Your example doesn’t demonstrate the Prototype pattern because it doesn’t copy/clone the object we want to copy/clone. The ‘prototype’ part of the pattern is missing 🙂
>
> Imagine a scenario where the object’s state evolves over time and it is significantly different from the creation time. This pattern is handy when you

want to make one or several copies of this evolved object.

Reply

hanuma

September 21, 2015 at 10:43 am

Hi,
I didn't get what is the difference between cloning an object by implementing Clonable(I) normally,and by using prototype pattern.what is the advantages in prototype design pattern over normal cloning process….

Thanks in advance….

Reply

Lokesh Gupta

September 25, 2015 at 12:54 pm

Prototype is kind factory pattern wrapped around cloning feature.

Reply

Suraj

November 27, 2014 at 4:22 pm

> Thank you ! Article helped me a lot ! :)

Reply

Raghav

October 27, 2014 at 6:50 am

Hi lokesh,
Can i solve this problem by Movie, Album and Show as a singleton classes(singleton pattern in java)

Thanks

Reply

Lokesh Gupta

October 27, 2014 at 7:14 am

Singleton 'Movie' means ONLY ONE MOVIE. How you will represent two movies with single instance?

Reply

Raghav

October 27, 2014 at 8:30 am

Thanks

Reply

aniket

February 10, 2014 at 8:21 am

Nice article!!!!
what actually this clone method do??? why creating instance is more costly than using clone???

Reply

Lokesh Gupta

February 10, 2014 at 8:26 am

You can read more here: https://howtodoinjava.com/java/cloning/a-guide-to-object-cloning-in-java/

Reply

Kunal

November 17, 2014 at 3:18 pm

Hi Lokesh,

Can you please let me know why creating instance is more costly than cloning? In your cloning session you have not provided the explanation. Please reply.

Thanks,
Kunal

Reply

Lokesh Gupta
November 17, 2014 at 4:50 pm

Well creating instances is not expensive all the times. It is only when object creation process involve other expensive processes such as opening DB connection, Opening network connection or doing some IO operations in file system etc. It can be expensive too when it involves creating many other instances as well inform of associated properties.

In above cases, creating instances is mostly expensive process in comparison to cloning an existing object.

Reply

ulaga
August 15, 2013 at 2:58 pm

Very good article,easy to understand the design patterns with real examples and also with perfect class diagram.
Really useful

Reply

Anonymous

June 13, 2013 at 3:10 pm

I didn't see much of value of introducing PrototypeCapable interface to enable clone() method, which is build-in in Object. It is not the contract obligation broken on Cloneable in java, it is the implementation was build-in for a purpose to make sure the Clone mechanism is working even across inheritance hierarchy, otherwise, the type inheritance will be mismatch. That means, there is no free way to correctly implement "clone" behavior defined in PrototypeCapable interface other than call "super.clone()" which binding to Object.clone build-in mechanism. What's the point?

Reply

Lokesh Gupta

June 13, 2013 at 4:36 pm

In java, to make an object cloning supported, it takes two steps: 1) Implement Cloneable 2) Override clone()

Lets say, a new developer joined a team which knows only basics in java. He is assigned a task in which he has to add one more object in prototype registry(factory) i.e. "Ticket".

By which design, you can force that developer to perform above both steps??

It is only possible when you tie both steps together otherwise behavior will be surprising.

Reply

Saurabh

November 9, 2014 at 11:45 pm

Lokesh,
So, in other words what you mean to say is the only purpose of this prototype pattern is to introduce an interface which actually has the method clone( ), that somehow Cloneable interface missed in it, so that the rule of overriding the clone( ) method is followed in future.

That's the only only only thing in this whole jargon I guess. That's it. Nothing else. I wonder why we create so many classes and confuse things when all we are doing here is just creating one interface! This whole thing is confusing and not worth the effort.

By the way, that little java programmer has to learn stuff, we cannot design software components keeping worries of junior programmers in focus. Someone who can spend 1 or 2 hours to understand how java is handling the concept of cloning its objects can easily understand how to override clone and why it's important to call the clone of super class.

Frankly, I think this is a design pattern is confusing and not worth studying and remembering, rather programmers should spend time on understanding the basics of cloning in java, that is enough.

Your thoughts?

Thanks,
Saurabh

Reply

Lokesh Gupta

November 11, 2014 at 6:52 am

Hi Saurabh, Thanks for sharing your thoughts. Much appreciated.

I believe that design patterns will feel like a burden if someone will try to apply them forcefully. They are solution of some specific problem. Until you encounter the problem in your design/implementation, you shouldn't inject patterns just to make code fancy. I agree with you that simplicity is the best thing you can do with your code.

Regarding prototype design pattern, I do not feel that it is not worth studying. Yes, proper use of clone() will also achieve the target, but you need some code to wrap the actual logic somewhere centralized. So when in future, a junior programmer find a tiny fault in cloning process, he does not have to change it multiple places.

Remember that if you have to change you code in more than place, to solve one specific problem, then your code need redesign/refactoring.

Reply

MarioOrcus

June 23, 2015 at 11:26 am

First of all, design patterns are not designed for java only. There is clone() in java but this is not the entire universe of programming.
Second, clone() works his own specific way, sometimes we want to achive other results and have to get rid of default clone() behaviour.
Third, I do not agree that 'keep it simple' always do the trick. Usually, you

> ‘keep it simple’ on one level and ‘make it very hard’ on the other: only global look and careful design let you avoid critical difficulties. Applaying design patterns makes code ready to develop and change; avoid problems before even noticed. My expirience tells me, that it really works.

Reply

Lokesh Gupta

June 23, 2015 at 11:34 am

> Hi Mario, Thanks for sharing your thoughts and experience.

Nick Risaro (@NickRisaro)

January 8, 2013 at 9:15 pm

> Using an enum for the ModelType is way better, plus strongly typed.
>
> As a matter of fact, you can do all of this with enums. I love enums 😉

Reply

Lokesh Gupta

January 8, 2013 at 10:00 pm

> +1, Agree that enum will be most suitable for the purpose.

Reply

Rajendra Joshi

January 4, 2013 at 10:28 pm

Dear Lokesh
ModelType is static subclass of PropertyFactory class ,so Following correction is required in running class TestPropertytypePattern

String moviePrototype = PrototypeFactory.getInstance(PrototypeFactory.ModelType.MOVIE).toString();

rest is fine ,I have tested it

Reply

Lokesh Gupta

January 4, 2013 at 10:44 pm

It can run both way… but ok… your way is more popular.

Reply

Leave a Comment

Name *

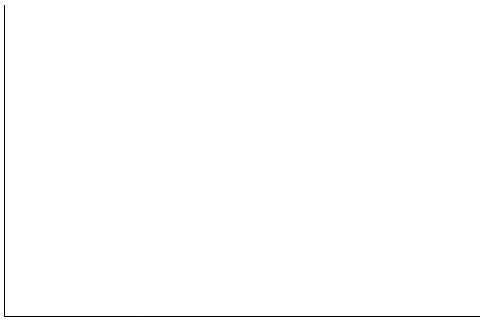
Email *

Website

Add me to your newsletter and keep me updated whenever you publish new blog posts

Post Comment

Search …



Privacy policy

Advertise

Guest Posts

Blogs

REST API Tutorial