HowToDoInJava

Java Factory Pattern Explained

Last Updated: January 25, 2022 By: Lokesh Gupta Creational Patterns Design Patterns

What is the most usual method of creating an instance of a class in java? Most people will answer this question: "*using new keyword*". Well, it is considered old-fashioned now. Let's see how??

If object creation code is spread in the whole application, and if you need to change the process of object creation then you need to go in each and every place to make necessary changes. After finishing this article, while writing your application, consider using the **Java factory pattern**.

In my previous post, "**Singleton design pattern in java**", we discussed various ways to create an instance of a class such that there can not exist another instance of same class in same JVM.

In this post, I will demonstrate another creational pattern, i.e. Factory pattern, for creating instances for your classes. Factory, as the name suggests, is a place to create some different products which are somehow similar in features yet divided into categories.

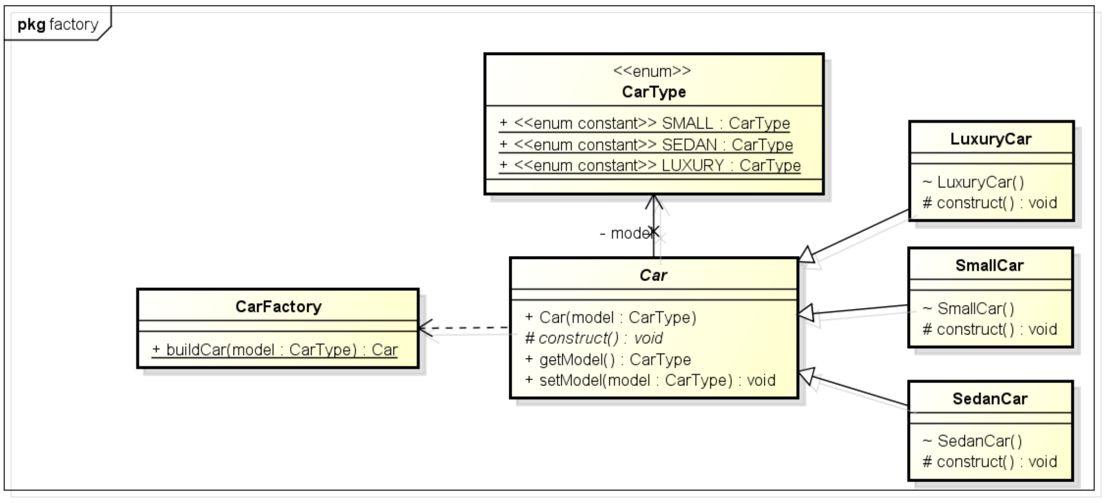
In Java, factory pattern is used to create instances of different classes of the same type.

Table of Contents

1. When to use factory pattern?

Factory pattern introduces **loose coupling between classes** which is the most important principle one should consider and apply while designing the application architecture. Loose coupling can be introduced in application architecture by programming against abstract entities rather than concrete implementations. This not only makes our architecture more flexible but also less fragile.

A picture is worth a thousand words. Let's see how a factory implementation will look like.



Above class-diagram depicts a common scenario using an example of a car factory which is able to build 3 types of cars i.e. small, sedan and luxury. Building a car requires many steps from allocating accessories to final makeup. These steps can be written in programming as methods and should be called while creating an instance of a specific car type.

If we are unfortunate then we will create instances of car types (e.g. `SmallCar`) in our application classes and thus we will expose the car building logic to the outside world and this is certainly not good. It also prevents us in making changes to car making process because the code is not centralized, and making changes in all composing classes seems not feasible.

2. Java Factory Pattern Example

So far we have design the classes need to be designed for making a **CarFactory**. Let's create them now.

2.1. Object types

`CarType` will hold the types of car and will provide car types to all other classes.

```
CarType.java

package designPatterns.creational.factory;

public enum CarType {
    SMALL, SEDAN, LUXURY
}
```

2.2. Object implementations

`Car` is parent class of all car instances and it will also contain the common logic applicable in car making of all types.

```
Car.java

package designPatterns.creational.factory;

public abstract class Car {

  public Car(CarType model) {
    this.model = model;
    arrangeParts();
  }

  private void arrangeParts() {
    // Do one time processing here
  }

  // Do subclass level processing in this method
  protected abstract void construct();

  private CarType model = null;

  public CarType getModel() {
    return model;
  }

  public void setModel(CarType model) {
    this.model = model;
  }
}
```

`LuxuryCar` is concrete implementation of car type `LUXURY`.

```
LuxuryCar.java

package designPatterns.creational.factory;

public class LuxuryCar extends Car {

  LuxuryCar() {
    super(CarType.LUXURY);
    construct();
  }

  @Override
  protected void construct() {
    System.out.println(&quot;Building luxury car&quot;);
    // add accessories
  }
}
```

`SmallCar` is concrete implementation of car type `SMALL`.

```
SmallCar.java

package designPatterns.creational.factory;

public class SmallCar extends Car {

  SmallCar() {
    super(CarType.SMALL);
    construct();
  }

  @Override
  protected void construct() {
    System.out.println(&quot;Building small car&quot;);
    // add accessories
  }
}
```

`SedanCar` is concrete implementation of car type `SEDAN`.

```
SedanCar.java

package designPatterns.creational.factory;

public class SedanCar extends Car {

  SedanCar() {
    super(CarType.SEDAN);
    construct();
  }

  @Override
  protected void construct() {
    System.out.println(&quot;Building sedan car&quot;);
    // add accessories
  }
}
```

2.3. Factory to create objects

`CarFactory.java` is our main class implemented using factory pattern. It instantiates a car instance only after determining its type.

```
CarFactory.java

package designPatterns.creational.factory;

public class CarFactory {
  public static Car buildCar(CarType model) {
    Car car = null;
    switch (model) {
    case SMALL:
      car = new SmallCar();
      break;

    case SEDAN:
      car = new SedanCar();
      break;

    case LUXURY:
      car = new LuxuryCar();
      break;

    default:
      // throw some exception
      break;
    }
    return car;
  }
}
```

2.4. Test factory pattern

In `TestFactoryPattern`, we will test our factory code. Lets run this class.

```
CarFactory.java

package designPatterns.creational.factory;

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));
    }
  }
```

Program Output.

```
Console

Building small car
designPatterns.creational.factory.SmallCar@7c230be4
Building sedan car
designPatterns.creational.factory.SedanCar@60e1e567
Building luxury car
designPatterns.creational.factory.LuxuryCar@e9bfee2
```

As you can see, the factory is able to return any type of car instance it is requested for. It will help us in making any kind of changes in car making process without even touching the composing classes i.e. classes using `CarFactory`.

3. Benefits of factory pattern

By now, you should be able to count the main advantages of using the factory pattern. Let's note down:

1. The creation of an object precludes its reuse without significant duplication of code.
2. The creation of an object requires access to information or resources that should not be contained within the composing class.
3. The lifetime management of the generated objects must be centralized to ensure a consistent behavior within the application.

4. Final notes

Factory pattern is most suitable where there is some complex object creation steps are involved. To ensure that these steps are centralized and not exposed to composing classes, factory pattern should be used. We can see many realtime examples of factory pattern in JDK itself e.g.

- java.sql.DriverManager#getConnection()
- java.net.URL#openConnection()
- java.lang.Class#newInstance()
- java.lang.Class#forName()

I hope, I have included enough information in this **Java factory pattern example** to make this post informative.

If you still have some doubt on abstract factory design pattern in Java, please leave a comment. I will be happy to discuss with you.

Happy Learning !!

Was this post helpful?

Let us know if you liked the post. That's the only way we can improve.

Yes

No

Recommended Reading:

1. Abstract Factory Pattern Explained
2. Java Singleton Pattern Explained
3. Immutable Collections with Factory Methods in Java 9
4. TestNG @Factory
5. TestNG – @Factory vs @DataProvider
6. Spring static factory-method example
7. Prototype design pattern in Java
8. Builder Design Pattern
9. Decorator Design Pattern in Java
0. Adapter Design Pattern in Java

Join 7000+ Awesome Developers

Get the latest updates from industry, awesome resources, blog updates and much more.

Email Address

Subscribe

** We do not spam !!*

51 thoughts on “Java Factory Pattern Explained”

Aarti

June 7, 2018 at 6:09 pm

in factory pattern.
If we have 1 laks sub classes ,then how to create factory.

if else condition,we cant put in factory class for 1 laks subclassess.
Please explain.

Reply

Lokesh Gupta

June 7, 2018 at 6:47 pm

🙂 Very hypothetical usecase. I will pass it to others.

Reply

atul m

April 29, 2019 at 2:07 pm

Can you put here name for those 1 L classes?

Reply

Lajpat Bishnoi

July 1, 2019 at 2:59 pm

We can use Reflection but it is costly and usecase specified by you doesn't make sense either. Anyway below is the partial code snipet.

1. Modify CarType enum to include classname as well.
Like for SMALL –> SmallCar etc.

```
public enum CarType{
    SMALL("SmallCar");
    private String className;
    public String getClassName(){return className;}
    public void setClassName(String className){this.className = className;}
    private CarType(String className){this.className = className;}
}
2. You can iterate over CarType enum.

public static Car buildCar(CarType model) {
        Car car = null;
       for(CarType carType : CarType.values) {
           if(model.equals(carType)) {
               car = Class.forName(carType.getClassName()).getConstructor().newInstance();
               break;
           }
       }
    if(null == car){
         // throw some exception
   }
  return car;
}
```

Reply

Priyak

February 12, 2020 at 10:28 pm

If you have 1 lakh subclasses, I think the next step should be to sit down and rethink your design and application domain model !

Reply

Nitin

February 2, 2018 at 10:45 am

What if we need to add another car type as SUV then we need to change CarFactory which is not good practice as it voilates Open close principle.am i correct?

Reply

Shivam Garg

July 8, 2019 at 2:39 pm

Adding a new CarType should not violate Open Close principle as we are not modifying any functionality of factory class by adding logic to create instance for SUV.
It works as it was working earlier.

Reply

Laxmikant

October 17, 2019 at 11:59 am

Hi Shivan, Yes you are correct, but CarFactory class changes if we add new car type, So CarFactory class violate OCP,
Instead of if else or for loop, We can create separate factory for each car type and let client decide to choose.
Please share your though on this. Thanks

Reply

Priyak Dey

February 12, 2020 at 10:35 pm

"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."

In case of you adding another if -else/ switch case, you are sort of extending the functionality of providing with another type of object but somewhat of the same type (thanks to abstract class/interface).But you are not modifying it.

I build a factory today to produce different kinds of chips, package and sell them. I start with potato chips and then EXTEND to produce banana chips. I am extending my functionality, not modifying the behaviour, I AM STILL A CHIPS FACTORY.

In case you have a library which today provides you with 3 Types of objects, – CAR, if you want to modify it and also provide with a ELECTRIC CAR, you have to recompile and provide the new upgraded jars. But you are still providing cars.

If you completely change/modify the behaviour it will be a problem. I think this is what OCP means. PLease pitch in in case it is wrong. Thanks

Reply

Niteen Dhule

November 14, 2016 at 10:56 am

Really nice post for Design Pattern begginers

Reply

Fabian

August 12, 2015 at 8:15 pm

Definitely, if it's "modern" or "up-to-date" to instantiate classes in Java within a switch case statement, you need to change your programming language right now. Consider moving your skills to a professional programming language. You are working very hard in Java. A real software development don't need to be very hard. Regards.

Reply

Lokesh Gupta

August 13, 2015 at 2:22 am

Hi Fabian, Thanks for the suggestion but perhaps it will be unnecessary. I really looked up in outer world to find if I am THE alien, but found plenty of guys like this.

I will not say that you are incorrect, but will appreciate if you can express your thoughts with facts/reasons. OR you can suggest your edit in wikipedia page because there I could see the same thing.

Reply

tom

August 11, 2015 at 3:37 am

Hi Lokesh,
How do you avoid casting after getting back an instance?

LuxuryCar lc= (LuxyryCar) FactoryManager.buildCar(CarType.LUXURY);

Reply

Michal

October 17, 2015 at 12:40 pm

He overrides the "construct" method, so he doesn't need casting ;-).

Informative post.

Regards,
Michał

Reply

Vlad

April 29, 2015 at 6:59 pm

Thanks a lot!

Reply

dev

March 27, 2015 at 5:00 am

Hello Lokesh Gupta,

can you tell me how the factory method is use to save the memory in java?

Please tell me about this.

Reply

Teja

September 26, 2014 at 12:25 pm

I might sound silly , why the concrete classes (Sedan,etc) are public. Since we are exposing a factory class to create us instance , it doesn't make sense to have them public right , any one can create new instance of these classes directly from anywhere, ain't they ??

Reply

Yovel

February 27, 2020 at 7:07 pm

Just what i was thinking, thanks for leaving a comment.

Reply

janakan kanaganayagam

July 22, 2014 at 11:32 pm

Hi in the original GOF factory pattern, there is a concept of having a creator and a concrete creator. I find this missing in your example. The CarFactory class is of course the concrete creator , but it does not extend from an abstract class / interface (creator)

So is it OK to call it a factory pattern without having the creator / concrete creator

Reply

Evaldas

July 20, 2014 at 3:11 pm

It's impossible to read this article on Samsung Galaxy 10.1 – the black advert strip jumps to the center of the webpage and covers the article text.

Reply

Lokesh Gupta

July 20, 2014 at 4:58 pm

I am trying to understand the problem. In the mean time, if possible then can you please share a screenshot of the problem.

Reply

Praveen Das S

July 2, 2014 at 11:49 am

How to reuse the same object? i.e If we call – CarFactory.buildCar(CarType.SMALL) multiple times, it returns a new object for every call.

Regards,
Praveen Das

Reply

Lokesh Gupta

July 2, 2014 at 1:03 pm

Hi Praveen, another question in response to your question. Why would a factory will return the same instance on each request? What's use of this approach? I believe you will have a good reason for it, so please share with us. It will expand our thought process as well.

To answer your question, you can apply singleton pattern on different Car objects and return from construct() method.

On another thought, you should better using prototype pattern, if you want to save some CPU cycles in car construction process. Singleton simply doesn't make sense to me.

https://howtodoinjava.com/design-patterns/creational/prototype-design-pattern-in-java/

Reply

Praveen Das S

July 7, 2014 at 9:28 am

Hi Lokesh, I'm looking at implementing a mailbox which fetches messages at a specific time interval.

Each messages fall under different types of validations, so depending on the NotificationType i call the ValidationFactory.

since there are for ex:22 notifications, and 100 messages, the notifications can be of same type. in this case i should be able to reuse the already created object for that particular NotificationType.

NotificationType would be my enum class.

ValidationService would be the parent class for all validation instances.

ValidationFactory would get me the respective object for the notificationType as input.

Regards,
Praveen

Reply

rze

October 10, 2014 at 12:40 pm

I don't think that Factory is the pattern you are looking for. However if you really want to use Factory pattern as your approach you should combine it with a caching adapter for the actual Factory which returns the cached instances.

Reply

Student

June 5, 2014 at 3:46 pm

Thanks for this wonderful example. My doubt is why do I get a warning in my IDE that the method "construct()" shouldn't be called in side the construtor because it's an overridable method?

I read that it has something to do with inheritance and if other classes extend this class things COULD go wrong down the line if someone overrides that method in another place.

Did I understand correctly the potencial danger here?

Reply

iles

June 30, 2014 at 12:07 pm

There is a danger in doing this and you should not do it (add overridable methods in the constructor)

Effective Java 2nd Edition, Item 17: Design and document for inheritance or else prohibit it

"There are a few more restrictions that a class must obey to allow inheritance. Constructors must not invoke overridable methods, directly or indirectly. If you violate this rule, program failure will result. The superclass constructor runs before the subclass constructor, so the overriding method in the subclass will get invoked before the subclass constructor has run. If the overriding method depends on any initialization performed by the subclass constructor, the method will not behave as expected."

Reply

Lokesh Gupta

June 30, 2014 at 12:34 pm

Pretty valid point.

Reply

Ryan Chesla

April 23, 2014 at 5:43 pm

Agreed!

Reply

mahantesh hiremath

April 7, 2014 at 5:59 am

Thanks Lokesh Gupta ji.

Reply

Roman Pereginiak

February 7, 2014 at 3:41 pm

I guess there is one small mistake (inaccuracy) in your example.
You have to make method constcut() abstract and put it into constructor instead of method arrangeParts():
public Car(CarType model) {
this.model = model;
//arrangeParts(); // remove this. You cold leave it but it just misslead you, it has nothing to design pattern.
construct(); // abstrac method! will be called implementation from interited class.
}
Then constructor LuxuryCar will look like:
LuxuryCar() {
super(CarType.LUXURY);
//construct(); // remove this!
}

Reply

Ryan Chesla

April 23, 2014 at 5:43 pm

Agreed!

Reply

Sagar Kumar Tak

May 14, 2014 at 6:44 pm

dear roman , pls explain i am new to java you told that use construct method in Car class constructor if we do so than how it will display information, because constructor will use this method definition from its own class and in Car class method construct() have empty definition that it is abstract.

Reply

Lokesh Gupta

May 14, 2014 at 7:06 pm

Sagar, Car is an abstract class you cannot create instance of it. You can create instance of classes which “extend Car” e.g. SmallCar. So actually in runtime construct method will be called either from SmallCar, SedanCar or LuxuryCar only. At this stage I would like to suggest you to first brush up the basic java concepts. That will help you in understanding more complex topics. Good Luck !!

Reply

Sagar Kumar Tak

May 15, 2014 at 4:59 pm

thnx
Can you pls suggest me learning link with exercise.

Reply

Lokesh Gupta

May 15, 2014 at 5:54 pm

https://docs.oracle.com/javase/tutorial/java/index.html
https://docs.oracle.com/javase/tutorial/collections/index.html
And Then

https://howtodoinjava.com/java/

Sagar Kumar Tak

May 16, 2014 at 7:45 pm

thnx boss 🙂

Gogo

June 1, 2014 at 11:51 pm

I do not agree with you Roman. The example is great.
The consruct() method is for logical code in the inherited Sub-Classes,
while method arrangeParts() is calling in the constructor, if in some cases has needs for that.
Regards

Reply

Mirek
December 8, 2013 at 2:50 am

Good post, thank you. Could you make it a bit more complex (as part2) in the way, lets say: Car is composed of underframe, engine and body and each of this component is created by factory depending of type of a car. What I'd like to see is how those factories will be used together, where to put it, how to call it. Thanks

Reply

Lokesh Gupta
December 8, 2013 at 11:03 am

A similar study you will find at: https://howtodoinjava.com/design-patterns/creational/abstract-factory-pattern-in-java/

Reply

Mirek
December 8, 2013 at 3:18 pm

Ok, thanks. What I'm trying to achieve is how to junittest class which have nested classes which have nested classes. Lets say: Service class uses nested Util class and Util class uses nested library class. And that library class is created by hardcoded new LibClass() but I'd like to use factory to easily switch instance to LibTestClass(). That is because LibClass connects to the internet but I'd like to mock it. By factory. Not by EasyMock or so even if it is possible. Lets say in another production environment I'd like to use LibClassDatabase() instead of LibClassHttp()

Reply

Søren Tryde Andersen

November 28, 2013 at 3:31 pm

Awesome stuff! Thanks.

Reply

srikanth

November 9, 2013 at 11:52 am

Read jus now before interview…really worth it!

Reply

Roshan

November 6, 2013 at 1:38 am

Thank You. That was useful. 🙂

Reply

alupis

September 24, 2013 at 11:40 pm

This was helpful. Thanks! 🙂

Reply

kushi

September 23, 2013 at 10:23 pm

You did not mention where CarFactory.buildCar() should be called… You have just written the testing part but I suppose the integration part is missed…

Reply

Lokesh Gupta
September 23, 2013 at 10:51 pm

Whenever you want a instance of car, use this method. Integration part.. I don't fully understand what you are referring to?? Where you want to integrate this example??

Reply

kushi
September 24, 2013 at 8:38 am

Thanks for ur prompt response… I got it… 1 more ques… In the UML diagram, am not able to understand the relationship between Car and CarFactory…

Reply

zikko
October 3, 2013 at 8:13 pm

yeah the arrow shud be in opposite direction.

Reply

sreenath
July 15, 2014 at 7:12 am

i do agree , the arrow should be in opposte direction
Ref : see uml diagram in : https://en.wikipedia.org/wiki/Factory_method_pattern

satnam singh

May 17, 2013 at 8:54 am

thanks dear

Reply

Leave a Comment

Name *

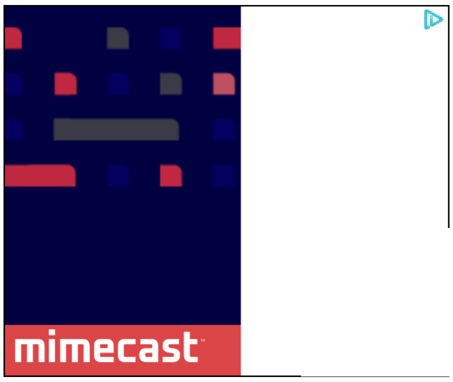
Email *

Website

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

Post Comment

Search …


mimecast