Factory Design Pattern in Java

MAY 3, 2018 BY PANKAJ — 30 COMMENTS

Welcome to the Factory Design Pattern in Java tutorial. **Factory Pattern** is one of the **Creational Design pattern** and it's widely used in JDK as well as frameworks like Spring and Struts.

Table of Contents [hide]

Factory Design Pattern

Factory design pattern is used when we have a super class with multiple sub-classes and based on input, we need to return one of the sub-class. This pattern take out the responsibility of instantiation of a class from client program to the factory class.

Let's first learn how to implement factory design pattern in java and then we will look into factory pattern advantages. We will see some of factory design pattern usage in JDK. Note that this pattern is also known as **Factory Method Design Pattern**.

Factory Design Pattern Super Class

Super class in factory design pattern can be an interface, **abstract class** or a normal java class. For our factory design pattern example, we have abstract super class with overridden `toString()` method for testing purpose.

```
package com.journaldev.design.model;

public abstract class Computer {

	public abstract String getRAM();
	public abstract String getHDD();
	public abstract String getCPU();

	@Override
	public String toString(){
		return "RAM= "+this.getRAM()+", HDD="+this.getHDD()+",
CPU="+this.getCPU();
	}
}
```

Factory Design Pattern Sub Classes

Let's say we have two sub-classes PC and Server with below implementation.

```
package com.journaldev.design.model;

public class PC extends Computer {

	private String ram;
	private String hdd;
	private String cpu;

	public PC(String ram, String hdd, String cpu){
		this.ram=ram;
		this.hdd=hdd;
		this.cpu=cpu;
	}
	@Override
	public String getRAM() {
		return this.ram;
	}

	@Override
	public String getHDD() {
		return this.hdd;
	}
```

Notice that both the classes are extending `Computer` super class.

```
package com.journaldev.design.model;

public class Server extends Computer {

	private String ram;
	private String hdd;
	private String cpu;

	public Server(String ram, String hdd, String cpu){
		this.ram=ram;
		this.hdd=hdd;
		this.cpu=cpu;
	}
	@Override
	public String getRAM() {
		return this.ram;
```

```
        }

        @Override
        public String getHDD() {
                return this.hdd;
        }
```

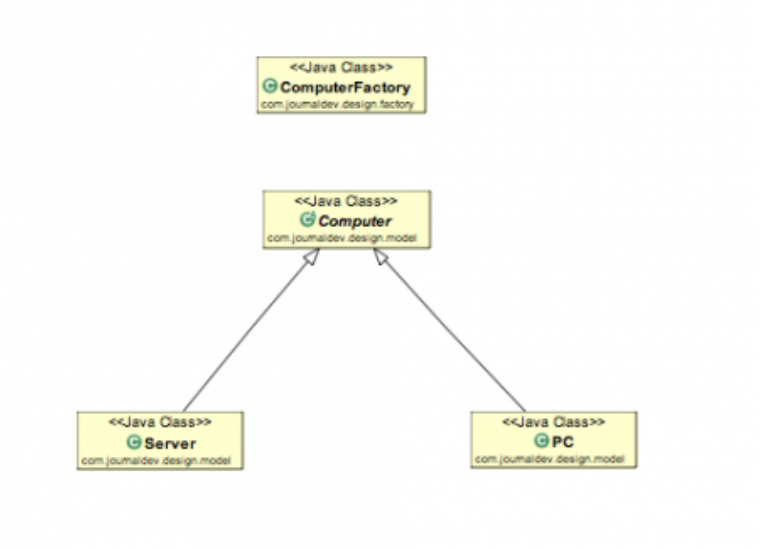
Factory Class

Now that we have super classes and sub-classes ready, we can write our factory class. Here is the basic implementation.

```
package com.journaldev.design.factory;

import com.journaldev.design.model.Computer;
import com.journaldev.design.model.PC;
import com.journaldev.design.model.Server;

public class ComputerFactory {

        public static Computer getComputer(String type, String ram, String hdd, String
cpu){
                if("PC".equalsIgnoreCase(type)) return new PC(ram, hdd, cpu);
                else if("Server".equalsIgnoreCase(type)) return new Server(ram, hdd,
cpu);

                return null;
        }
}
```

Some important points about Factory Design Pattern method are;

1. We can keep Factory class Singleton or we can keep the method that returns the subclass as static.
2. Notice that based on the input parameter, different subclass is created and returned. `getComputer` is the factory method.



Here is a simple test client program that uses above factory design pattern implementation.

```
package com.journaldev.design.test;

import com.journaldev.design.factory.ComputerFactory;
import com.journaldev.design.model.Computer;

public class TestFactory {

	public static void main(String[] args) {
		Computer pc = ComputerFactory.getComputer("pc","2 GB","500 GB","2.4
GHz");
		Computer server = ComputerFactory.getComputer("server","16 GB","1
TB","2.9 GHz");
		System.out.println("Factory PC Config::"+pc);
		System.out.println("Factory Server Config::"+server);
	}

}
```

Output of above program is:

```
Factory PC Config::RAM= 2 GB, HDD=500 GB, CPU=2.4 GHz
Factory Server Config::RAM= 16 GB, HDD=1 TB, CPU=2.9 GHz
```

Factory Design Pattern Advantages

1. Factory design pattern provides approach to code for interface rather than implementation.
2. Factory pattern removes the instantiation of actual implementation classes from client code. Factory pattern makes our code more robust, less coupled and easy to extend. For example, we can easily change PC class implementation because client program is unaware of this.
3. Factory pattern provides abstraction between implementation and client classes through inheritance.

Factory Design Pattern Examples in JDK

1. java.util.Calendar, ResourceBundle and NumberFormat `getInstance()` methods uses Factory pattern.
2. `valueOf()` method in wrapper classes like Boolean, Integer etc.

Factory Design Pattern YouTube Video Tutorial

I recently uploaded a video on YouTube for Factory Design pattern, please check it out. Please like and share the video and subscribe to my YouTube channel.

You can download the example code from my GitHub Project.

« PREVIOUS
Java Singleton Design Pattern Best Practices with Examples

NEXT »
Abstract Factory Design Pattern in Java

About Pankaj

If you have come this far, it means that you liked what you are reading. Why not reach little more and connect with me directly on **Google Plus**, **Facebook** or **Twitter**. I would love to hear your thoughts and opinions on my articles directly.

Recently I started creating video tutorials too, so do check out my videos on **Youtube**.

FILED UNDER: DESIGN PATTERNS

Comments

Jaleel says
JULY 24, 2018 AT 2:30 PM

Very weel explained! Thank you!
Reply

wade says
JUNE 3, 2018 AT 4:43 AM
very nice tutorial, easy to understand!
Reply

Nirav Khandhedia says
APRIL 4, 2018 AT 9:59 AM
I understand that there's always a benefit to get the instance of a class using the provided factory implementation.
How can I force users to compulsorily use the factory implementation to get the instance, instead of getting away creating an instance directly using new operator of the actual class?
i.e. How can I force people to use ServerFactory.getInstance() instead of new Server()?
Reply

Geeks says
APRIL 10, 2018 AT 2:31 AM
You can make Server Class as an abstract class.
Reply

abhi says
APRIL 19, 2018 AT 3:47 AM
But in that case Your Factory wont be able to create the instance itself.
Reply

Muthu Vignesh says
MAY 5, 2018 AT 11:56 PM
If you can make the constructor private as per singleton, then i assume that you cannot create the instance using new Server(); and only enforces the user to access the object only through getInstance(). Also as an additional rule getInstance() method can be enforced as a rule to be implemented in the sub-classes extending computer. i.e. an abstract method of getInstance(), be present in Computer class which is already an abstract class. Hope m right, others correct me if m wrong
Reply

Srinivasa..... says
MARCH 30, 2018 AT 2:58 AM
It's Nice Explanation Learnt new things more it's best practice and understanding
Reply

Chris says
FEBRUARY 21, 2018 AT 3:13 AM
Quick and easy tutorial, thanks.
Reply

Daniel says
FEBRUARY 16, 2018 AT 7:35 AM
Learned some new stuff with very detailed information.
Reply

BkOfc says
JANUARY 23, 2018 AT 5:39 PM
Where below are implemented
import com.journaldev.design.abstractfactory.PCFactory;
import com.journaldev.design.abstractfactory.ServerFactory;
Reply

Pankaj says
JANUARY 23, 2018 AT 8:33 PM
Sorry, that came out while copying the code from my Eclipse editor by mistake. Those imports are useless, I have removed them from above code.
Reply

Prashanth says
NOVEMBER 15, 2017 AT 8:05 PM
It's a very good tutorial. But I have a doubt,
You mentioned Calendar#getInstance() as factory pattern implementation. But in this there is a small difference right?
There is no separate factory class. The super class Calendar itself is acting as the factory class.
Does an implementation like this have any advantage or disadvantage?
Reply

Luis Cunha says
SEPTEMBER 8, 2017 AT 8:02 AM
Hi, is there a place in which I can download all the source code for the several design pattern examples. These are great examples, but I have to copy-paste each single text box into Intellij, and it is very cumbersome.
Thank you very much, and congratulations for such good material.
Reply

Vishal says
AUGUST 23, 2017 AT 1:47 AM
Yes. Its very nice article about simple factory covers basic concepts.
Reply

Stephen Ubogu says
MAY 7, 2017 AT 4:09 AM
I am relatively new to design patterns but I need to ask this question. What if we need to add another subclass of computer say Laptop to the application.? Does this mean we will have to modify the computer factory class? This looks like violating the OO principle which says classes should be closed to modification but open to extension.
Reply

JocelynL says
OCTOBER 6, 2016 AT 3:28 AM
It seems to me that you're showing what is called a simple factory with ComputerFactory; It is not the Factory Method Pattern.
The client TestFactory delegates the creation to another class which it is composed with.

If you want to implement the Factory Method Pattern,:
1. ComputerFactory should define an asbtract method getComputer(String ram, String hdd, String cpu)
2. ComputerFactory should have two subclasses PCFactory and ServerFactory.which implements the superclass abstract method to return either a PC or a server
3. The client should be given one of the two concrete factories and call the factory method to get PC or servers, depending which one was instanciated
Reply

catherine says
FEBRUARY 25, 2017 AT 6:24 PM
yes , i agree. the article is about simple factory not factory .
But still a nice article.
Reply

Vijay Kambala says
MAY 3, 2018 AT 5:18 AM
Could you please reply back with actual factory pattern example
in your own explanatroy words…
Reply

Vinod Kumar says
MARCH 23, 2017 AT 4:52 AM
yes absolutely you are right. It is not factory method pattern.
Reply

ravi says
JUNE 15, 2017 AT 1:18 AM
This is a factory (factory method) pattern, if you make factory abstract then it becomes abstract factory pattern
Reply

Gani Victory says
AUGUST 11, 2016 AT 3:37 AM

Nice article !!!!!!!!
Reply

panky031 says
JUNE 3, 2016 AT 7:27 AM
Now i found the perfect article for Design pattern.
Thanks Pankaj
Reply

vamshi says
FEBRUARY 23, 2015 AT 6:20 PM
Thanks for the clear explanation.
I have one doubt here in Factory pattern. We have two concrete classes implementing the interface/Abstract class whose instances are created inside Factory class.But, instead of below line
Computer pc = ComputerFactory.getComputer("pc","2 GB","500 GB","2.4 GHz");
we can also use
Computer pc=new PC("pc","2 GB","500 GB","2.4 GHz"); to get new instance of PC. Then what is the advantage of using ComputerFactory.getComputer() method on the client side directly.?
Reply

Ofer Yuval says
APRIL 22, 2015 AT 12:03 AM
See here
http://stackoverflow.com/questions/14575457/factory-classes
Reply

Ajay says
APRIL 5, 2016 AT 11:29 AM
That's why it's called an creation all design pattern cause then we don't have to dirty our code keeping the instance creation all logic here and there cause we have implanted a factory out there which is just doing it for us you just name it..name the object and it's ready for you....
Reply

Avinash Nayak says
SEPTEMBER 14, 2017 AT 10:31 AM

Thats because u will be bound to the object, ie if you create Computer pc=new PC("pc","2 GB","500 GB","2.4 GHz") u will always get the instance of PC and it would be hardcoding, So if you use factory you wil not worry of the implementation u will always get the object of reference Computer.
Reply

Siva says
OCTOBER 10, 2014 AT 2:26 PM
Some body asked me that Why do we have to implement singleton pattern when we have static.
And here in your post you say that either we can use static method or implement it as Singleton. Can you please detail on this?
Reply

JavaBee says
MAY 16, 2016 AT 7:00 AM
May be its a late reply, but worth share thought here.
Factory classes(In general design patterns) are meant for maintainability and re-usability. Factory pattern states that, the objective of this pattern is to decouple the object instantiation from the client program. And this can be achieved either by static method or singleton factory class.
Why we use singleton pattern when we have static?
we use "static" when a piece of code/data same across all instances of a class OR important piece of code that needs to be executed even when class is not instantiated OR instantiation is not required.
Question here is, how to make outer class itself static? the answer is "Singleton Pattern".
The "singleton pattern" is a mechanism, which gives the flexibility to reuse the same instance of a class(avoid multiple instantiation of a class when it is not required OR execute mandatory functionality only once) OR have the single instance to achieve the expected functionality.
This can be achieved by using static + additional checks on the pre-existence of the instance(of self).
Ex: Thread pool.
Reply

robothy says
OCTOBER 11, 2016 AT 2:26 AM
Good answer!!!
Reply

RazorEdge says
FEBRUARY 26, 2017 AT 8:14 AM

Very good answer… Thank You

Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Save my name, email, and website in this browser for the next time I comment.

POST COMMENT

Search for tutorials...

DOWNLOAD ANDROID APP

DESIGN PATTERNS TUTORIAL

Java Design Patterns

Creational Design Patterns

- Singleton
- Factory
- Abstract Factory
- Builder
- Prototype

Structural Design Patterns

- Adapter
- Composite
- Proxy
- Flyweight
- Facade
- Bridge
- Decorator

Behavioral Design Patterns

- Template Method
- Mediator
- Chain of Responsibility
- Observer
- Strategy
- Command
- State
- Visitor
- Interpreter
- Iterator
- Memento

Miscellaneous Design Patterns

- Dependency Injection
- Thread Safety in Java Singleton

RECOMMENDED TUTORIALS

Java Tutorials

- Java IO
- Java Regular Expressions
- Multithreading in Java
- Java Logging
- Java Annotations
- Java XML
- Collections in Java
- Java Generics

- Exception Handling in Java
- Java Reflection
- Java Design Patterns
- JDBC Tutorial

Java EE Tutorials

- Servlet JSP Tutorial
- Struts2 Tutorial
- Spring Tutorial
- Hibernate Tutorial
- Primefaces Tutorial
- Apache Axis 2
- JAX-RS
- Memcached Tutorial