**Factory Design Pattern**

[](https://cdn.journaldev.com/wp-content/uploads/2013/05/factory-design-pattern.jpg)

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

Let’s first learn how to implement a factory design pattern in java and then we will look into factory pattern advantages. We will see some of the 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**](https://www.journaldev.com/1582/abstract-class-in-java) or a normal java class. For our factory design pattern example, we have abstract super class with [overridden](https://www.journaldev.com/817/java-override-method-overriding) toString() method for testing purpose.

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.

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;

}

@Override

public String getCPU() {

return this.cpu;

}

}

Notice that both the classes are extending Computer super class.

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;

}

@Override

public String getCPU() {

return this.cpu;

}

}

**Factory Class**

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

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);

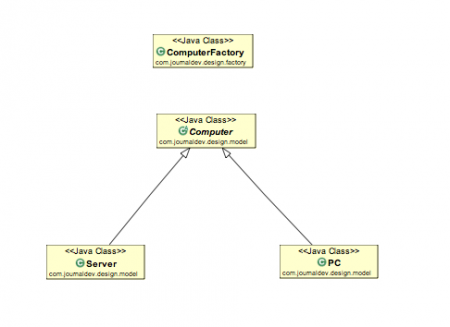
return null;

}

}

Some important points about Factory Design Pattern method are;

1. We can keep Factory class [Singleton](https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples) or we can keep the method that returns the subclass as [static](https://www.journaldev.com/1365/static-keyword-in-java).
2. Notice that based on the input parameter, different subclass is created and returned. getComputer is the factory method.

[](https://cdn.journaldev.com/wp-content/uploads/2013/05/factory-pattern-java.png)

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

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.