**Abstract Factory**

[](https://cdn.journaldev.com/wp-content/uploads/2013/06/abstract-factory-design-pattern.jpg)

If you are familiar with [**factory design pattern in java**](https://www.journaldev.com/1392/factory-design-pattern-in-java), you will notice that we have a single Factory class. This factory class returns different subclasses based on the input provided and factory class uses if-else or switch statement to achieve this.

In the Abstract Factory pattern, we get rid of if-else block and have a factory class for each sub-class. Then an Abstract Factory class that will return the sub-class based on the input factory class. At first, it seems confusing but once you see the implementation, it’s really easy to grasp and understand the minor difference between Factory and Abstract Factory pattern.

Like our factory pattern post, we will use the same superclass and sub-classes.

**Abstract Factory Design Pattern Super Class and Subclasses**

Computer.java

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();

}

}

PC.java

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;

}

}

Server.java

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 for Each subclass**

First of all we need to create a Abstract Factory interface or [**abstract class**](https://www.journaldev.com/1582/abstract-class-in-java).

ComputerAbstractFactory.java

package com.journaldev.design.abstractfactory;

import com.journaldev.design.model.Computer;

public interface ComputerAbstractFactory {

public Computer createComputer();

}

Notice that createComputer() method is returning an instance of super class Computer. Now our factory classes will implement this interface and return their respective sub-class.

PCFactory.java

public class PCFactory implements ComputerAbstractFactory {

private String ram;

private String hdd;

private String cpu;

public PCFactory(String ram, String hdd, String cpu){

this.ram=ram;

this.hdd=hdd;

this.cpu=cpu;

}

@Override

public Computer createComputer() {

return new PC(ram,hdd,cpu);

}

}

Similarly we will have a factory class for Server subclass.

ServerFactory.java

public class ServerFactory implements ComputerAbstractFactory {

private String ram;

private String hdd;

private String cpu;

public ServerFactory(String ram, String hdd, String cpu){

this.ram=ram;

this.hdd=hdd;

this.cpu=cpu;

}

@Override

public Computer createComputer() {

return new Server(ram,hdd,cpu);

}

}

Now we will create a consumer class that will provide the entry point for the client classes to create sub-classes.

ComputerFactory.java

public class ComputerFactory {

public static Computer getComputer(ComputerAbstractFactory factory){

return factory.createComputer();

}

}

Notice that its a simple class and getComputer method is accepting ComputerAbstractFactory argument and returning Computer object. At this point the implementation must be getting clear.

Let’s write a simple test method and see how to use the abstract factory to get the instance of sub-classes.

TestDesignPatterns.java

public class TestDesignPatterns {

public static void main(String[] args) {

testAbstractFactory();

}

private static void testAbstractFactory() {

Computer pc = com.journaldev.design.abstractfactory.ComputerFactory.getComputer(new PCFactory("2 GB","500 GB","2.4 GHz"));

Computer server = com.journaldev.design.abstractfactory.ComputerFactory.getComputer(new ServerFactory("16 GB","1 TB","2.9 GHz"));

System.out.println("AbstractFactory PC Config::"+pc);

System.out.println("AbstractFactory Server Config::"+server);

}

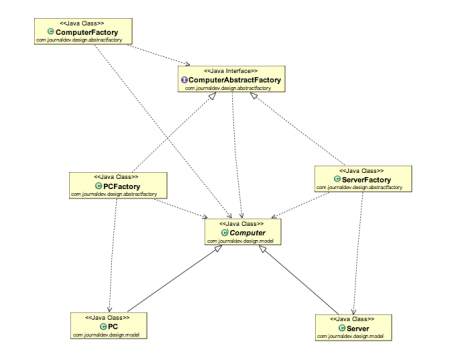
}

Output of the above program will be:

AbstractFactory PC Config::RAM= 2 GB, HDD=500 GB, CPU=2.4 GHz

AbstractFactory Server Config::RAM= 16 GB, HDD=1 TB, CPU=2.9 GHz

Here is the class diagram of abstract factory design pattern implementation.

[](https://cdn.journaldev.com/wp-content/uploads/2013/06/Abstract-Factory-Pattern.png)

**Abstract Factory Design Pattern Benefits**

* Abstract Factory design pattern provides approach to code for interface rather than implementation.
* Abstract Factory pattern is “factory of factories” and can be easily extended to accommodate more products, for example we can add another sub-class Laptop and a factory LaptopFactory.
* Abstract Factory pattern is robust and avoid conditional logic of Factory pattern.

**Abstract Factory Design Pattern Examples in JDK**

* javax.xml.parsers.DocumentBuilderFactory#newInstance()
* javax.xml.transform.TransformerFactory#newInstance()
* javax.xml.xpath.XPathFactory#newInstance()