# Abstract Factory Design Pattern in Java

Pankaj Kumar

Abstract Factory is one of the **Creational pattern** and almost similar to **Factory Pattern** except the fact that its more like factory of factories.

If you are familiar with **factory design pattern in java**, you will notice that we have a single Factory class that returns the different sub-classes based on the input provided and factory class uses if-else or switch statement to achieve this.

In Abstract Factory pattern, we get rid of if-else block and have a factory class for each sub-class and 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, its 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 super class and sub-classes.

## Super Class and Sub-Classes

|  |
| --- |
| **package** org.xman.dp.creational.abstractfactory;  **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();  }  } |

|  |
| --- |
| **package** org.xman.dp.creational.abstractfactory;  **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;  }  } |

|  |
| --- |
| **package** org.xman.dp.creational.abstractfactory;  **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 Classes for Each sub-class

First of all we need to create a Abstract Factory interface or **abstract class**.

|  |
| --- |
| **package** org.xman.dp.creational.abstractfactory;  **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.

|  |
| --- |
| **package** org.xman.dp.creational.abstractfactory;  **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 sub-class.

|  |
| --- |
| **package** org.xman.dp.creational.abstractfactory;  **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.

|  |
| --- |
| **package** org.xman.dp.creational.abstractfactory;  **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.

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

|  |
| --- |
| **package** org.xman.dp.creational.abstractfactory;  **public** **class** TestDesignPatterns {  **public** **static** **void** **main**(String[] args) {  *testAbstractFactory*();  }  **private** **static** **void** **testAbstractFactory**() {  Computer pc = ComputerFactory.*getComputer*(**new** PCFactory("2 GB", "500 GB", "2.4 GHz"));  Computer server = 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 implementation.



## Benefits of Abstract Factory Pattern

* Abstract Factory 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 Pattern Examples in JDK

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