Factory Design Pattern in Java

[Pankaj](http://www.journaldev.com/author/Pankaj) May 22, 2013 [Design Patterns](http://www.journaldev.com/dev/java/design-patterns), [Java](http://www.journaldev.com/dev/java)

Pankaj Kumar

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

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 pattern in java and then we will learn it’s benefits and we will see its usage in JDK.

Super Class

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

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

Sub Classes

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

|  |
| --- |
| **package** org.xman.dp.creational.factory;  **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 class.

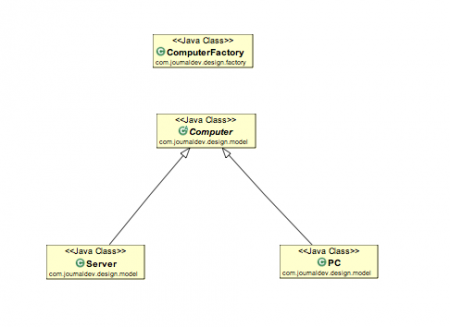
|  |
| --- |
| **package** org.xman.dp.creational.factory;  **public** **class** Server **extends** Computer {  **private** String ram;  **private** String hdd;  **private** String cpu;  **public** **Server**(String ram, String hdd, String cpu) {  **this**.ram = ram;  **this**.hdd = hdd;  **this**.cpu = 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.

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

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.

[](http://cdn2.journaldev.com/wp-content/uploads/2013/05/factory-pattern-java.png)

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

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

Benefits of Factory Pattern

1. Factory pattern provides approach to code for interface rather than implementation.
2. Factory pattern removes the instantiation of actual implementation classes from client code, making it 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 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.