FACTORY PATTERN:

**Factory design pattern in Java** one of the core design pattern which is used heavily not only in JDK but also in various Open Source framework such as Spring, Struts and Apache along with decorator design pattern in Java. Factory Design pattern is based on [**Encapsulation**](http://javarevisited.blogspot.com/2012/03/what-is-encapsulation-in-java-and-oops.html)object oriented concept. Factory method is used to create different object from factory often refereed as Item and it encapsulate the creation code**. So instead of having object creation code on client side we encapsulate inside Factory method in Java**. One of the best examples of factory pattern in Java is BorderFactory Class of Swing API. In this Design pattern tutorial we will see **What is Factory method design pattern in Java**, What are main *advantages of factory pattern in Java* , Code example of Factory design pattern and What problem **Factory pattern** solves in Java or when to use Factory design pattern.

### When to use Factory design pattern in Java

* Static Factory methods are common in frameworks where library code needs to create objects of types which may be sub classed by applications using the framework.
* Some or all concrete products can be created in multiple ways, or we want to leave open the option that in the future there may be new ways to create the concrete product.
* **Factory method is used when Products don't need to know how they are created.**
* **We  can use factory pattern where we have to create an object of any one of sub-classes depending on the data provided**

### Code Example of Factory Design Pattern in Java:

Let’s see an example of how factory pattern is implemented in Code.We have requirement to create multiple currency e.g. INR, SGD, USD and code should be extensible to accommodate new Currency as well. Here we have made Currency as interface and all currency would be concrete implementation of Currency interface. Factory Class will create Currency based upon country and return concrete implementation which will be stored in interface type. This makes code dynamic and extensible.

Here is complete **code example of Factory pattern in Java**:

**interface** Currency {

       String getSymbol();

}

// Concrete Rupee Class code

**class** Rupee **implements** Currency {

       @Override

**public** String getSymbol() {

**return** "Rs";

       }

}

// Concrete SGD class Code

**class** SGDDollar **implements** Currency {

       @Override

**public** String getSymbol() {

**return** "SGD";

       }

}

// Concrete US Dollar code

**class** USDollar **implements** Currency {

       @Override

**public** String getSymbol() {

**return** "USD";

       }

}

// Factroy Class code

**class** CurrencyFactory {

**public** **static** Currency createCurrency (String country) {

**if** (country. equalsIgnoreCase ("India")){

**return** **new** Rupee();

       }**else** **if**(country. equalsIgnoreCase ("Singapore")){

**return** **new** SGDDollar();

       }**else** **if**(country. equalsIgnoreCase ("US")){

**return** **new** USDollar();

        }

**throw** **new** IllegalArgumentException("No such currency");

       }

}

// Factory client code

**public** **class** Factory {

**public** **static** **void** main(String args[]) {

              String country = args[0];

              Currency rupee = CurrencyFactory.*createCurrency*(country);

              System.*out*.println(rupee.getSymbol());

       }

}

### Advantage of Factory method Pattern in Java:

**Factory pattern in Java** is heavily used everywhere including JDK, open source library and other frameworks. In following are main advantages of using Factory pattern in Java:

1*) Factory method design pattern* decouples the calling class from the target class, which result in less coupled and highly cohesive code?

E.g.: **JDBC is a good example for this pattern; application code doesn't need to know what database it will be used with, so it doesn't know what database-specific driver classes it should use.** Instead, it uses factory methods to get Connections, Statements, and other objects to work with. Which gives you flexibility to change your back-end database without changing your DAO layer in case you are using ANSI SQL features and not coded on DBMS specific feature?

**Java.util.Collection#Iterator is a good example of a Factory Method. Depending on the concrete subclass of Collection you use, it will create an Iterator implementation.**

Because both the Factory superclass (Collection) and the Iterator created are interfaces,

* [**java.util.Calendar.getInstance()**](http://docs.oracle.com/javase/6/docs/api/java/util/Calendar.html#getInstance%28%29)
* [**java.util.ResourceBundle.getBundle()**](http://docs.oracle.com/javase/6/docs/api/java/util/ResourceBundle.html#getBundle%28java.lang.String%29)
* [**java.text.NumberFormat.getInstance()**](http://docs.oracle.com/javase/6/docs/api/java/text/NumberFormat.html#getInstance%28%29)
* [**java.nio.charset.Charset.forName()**](http://docs.oracle.com/javase/6/docs/api/java/nio/charset/Charset.html#forName%28java.lang.String%29)
* [**java.net.URLStreamHandlerFactory.createURLStreamHandler(String)**](http://docs.oracle.com/javase/6/docs/api/java/net/URLStreamHandlerFactory.html) (Returns singleton object per protocol)

2) Factory pattern in Java enables the subclasses to provide extended version of an object, **because creating an object inside factory is more flexible than creating an object directly in the client.** Since client is working on interface level any time you can enhance the implementation and return from Factory.

3) Another benefit of using *Factory design pattern in Java* is that it encourages [consistency in Code](http://javarevisited.blogspot.com/2011/09/code-review-checklist-best-practice.html) since every time object is created using Factory rather than using different constructor at different client side.

4) Code written using Factory design pattern in Java is also [easy to debug](http://javarevisited.blogspot.com/2011/07/java-debugging-tutorial-example-tips.html) and troubleshoot because you have a centralized method for object creation and every client is getting object from same place.

Some more advantages of factory method design pattern is:

1. **Static factory method** used in factory design pattern enforces use of Interface than implementation which itself a good practice. for example:

**Map** synchronizedMap = **Collections**.synchronizedMap(**new** **HashMap**());

2. Since static factory method have return type as Interface, it allows you to replace implementation with better performance version in newer release.

3. Another advantage of static factory method pattern is that they can cache frequently used object and eliminate duplicate object creation**. Boolean.valueOf()** method is good example which caches true and false boolean value.

4. Factory method pattern is also recommended by Joshua Bloch in Effective Java.

5 Factory method pattern offers alternative way of creating object.

6. Factory pattern can also be used to hide information related to creation of object.

That’s all on **Factory design patten in Java** for now. This is one of the most used patterns in Java library and different Java frameworks. Summary is try to use **Factory pattern** whenever you see an opportunity to encapsulate object creation code and see chance of creating different object in near future.