**Factory method Design 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](http://javarevisited.blogspot.com/2011/11/decorator-design-pattern-java-example.html).

## 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 Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface. What is static factory method or factory design pattern

[factory design patter in java tutorial with example](http://2.bp.blogspot.com/-wrzDeQGAe1I/TWu8pLuLr4I/AAAAAAAAADE/V017G-6Q61w/s1600/java_logo_50_50.jpg)Factory design pattern is used to create objects or [Class in Java](http://javarevisited.blogspot.com/2011/10/class-in-java-programming-general.html) and it provides loose coupling and high cohesion. Factory pattern encapsulate object creation logic which makes it easy to change it later when you change how object gets created or you can even introduce new object with just change in one class. In GOF pattern list Factory pattern is listed as Creation design pattern. Factory should be an interface and clients first either creates factory or get factory which later used to create objects.

## Example of static factory method in JDK

Best Example of Factory method design pattern is valueOf() method which is there in String and wrapper classes like Integer and Boolean and used for type conversion i.e. from converting String to Integer or String to double in java..

Some more examples of factory method design pattern from JDK is :

**valueOf()** method which returns object created by factory equivalent to value of parameter passed.

**getInstance()** method which creates instance of Singleton class.

**newInstance()** method which is used to create and return new instance from factory method every time called.

**getType()** and **newType()** equivalent of **getInstance()** and **newInstance()** factory method but used when factory method resides in separate class.

### Problem which is solved by Factory method Pattern in Java

Whenever we talk about **object oriented language** it will based upon some concept like [abstraction](http://javarevisited.blogspot.com/2010/10/abstraction-in-java.html), [polymorphism](http://javarevisited.blogspot.com/2011/08/what-is-polymorphism-in-java-example.html) etc and on that [encapsulation](http://javarevisited.blogspot.com/2012/03/what-is-encapsulation-in-java-and-oops.html)and delegation are important concept any design will be called good if task are delegated to different object and some kind of encapsulation is there.

Sometime our application or framework will not know that what kind of object it has to create at run-time it knows only the interface or abstract class and as we know we cannot create object of interface or abstract class so main problem is frame work knows **when** it has to create but don’t know **what kind** of object.

Whenever we create object using new() we violate **principle of programming for interface rather than implementation** which eventually result in inflexible code and difficult to change in maintenance. By using Factory design pattern in Java we get rid of this problem.

Another problem we can face is class needs to contain objects of other classes or class hierarchies within it; this can be very easily achieved by just using the new keyword and the class constructor. The problem with this approach is that it is a very hard coded approach to create objects as this creates dependency between the two classes.

So **factory pattern** solve this problem very easily by model an interface for creating an object which at creation time can let its subclasses decide which class to instantiate, Factory Pattern promotes loose coupling by eliminating the need to bind application-specific classes into the code. The **factory methods** are typically implemented as virtual methods, so this pattern is also referred to as the “**Virtual Constructor**”. These methods create the objects of the products or target classes.

### 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?

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**](http://java.sun.com/j2se/1.5.0/docs/api/java/util/Collections.html).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.](http://www.amazon.com/dp/0321356683/?tag=javamysqlanta-20)

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.

## Builder Design pattern

## Builder design pattern in Java is a creational pattern i.e. used to create objects, similar to [factory method design pattern](http://javarevisited.blogspot.com/2011/12/factory-design-pattern-java-example.html) which is also creational design pattern.

## What problem Builder pattern solves in Java

[What is builder design pattern in Java with example](http://2.bp.blogspot.com/-wrzDeQGAe1I/TWu8pLuLr4I/AAAAAAAAADE/V017G-6Q61w/s1600/java_logo_50_50.jpg)As I said earlier *Builder pattern* is a creational design pattern it means its solves problem related to object creation. Constructors in Java are used to create object and can take parameters required to create object. Problem starts when an Object can be created with **lot of parameters**, some of them may be **mandatory** and others may be **optional**. Consider a class which is used to create Cake, now you need number of item like egg, milk, flour to create cake. many of them are mandatory and some  of them are optional like cherry, fruits etc. If we are going to have [overloaded constructor](http://javarevisited.blogspot.sg/2012/01/what-is-constructor-overloading-in-java.html) for different kind of cake then there will be many constructor and even worst they will accept many parameter.

**Problems:**

1) too many constructors to maintain.

2) error prone because many fields has same type e.g. sugar and and butter are in cups so instead of 2 cup sugar if you pass 2 cup butter, your compiler will not complain but will get a buttery cake with almost no sugar with high cost of wasting butter.

You can partially solve this problem by creating Cake and then adding ingredients but that will impose another problem of **leaving Object on inconsistent state during building**, ideally cake should not be available until its created. Both of these problem can be solved by using Builder design pattern in Java. Builder design pattern not only improves readability but also reduces chance of error by adding ingredients explicitly and making object available once fully constructed.

By the way there are many design pattern tutorial already there in Javarevisited like [Decorator pattern tutorial](http://javarevisited.blogspot.com/2011/11/decorator-design-pattern-java-example.html) and  [Observer pattern in Java](http://javarevisited.blogspot.com/2011/12/observer-design-pattern-java-example.html). If you haven’t read them already then its worth looking.

## Example of Builder Design pattern in Java

We will use same example of creating Cake using Builder design pattern in Java. here we have [static nested builder class](http://javarevisited.blogspot.com/2011/11/static-keyword-method-variable-java.html) inside Cake which is used to create object.

**Guidelines for Builder design pattern in Java**

1) Make a static nested class called Builder inside the class whose object will be build by Builder. In this example its Cake.

2) Builder class will have exactly same set of fields as original class.

3) Builder class will expose method for adding ingredients e.g. sugar() in this example. each method will return same Builder object. Builder will be enriched with each method call.

4) Builder.build() method will copy all builder field values into actual class and return object of Item class.

5) Item class (class for which we are creating Builder) should have [private constructor](http://javarevisited.blogspot.sg/2012/03/private-in-java-why-should-you-always.html) to create its object from build() method and prevent outsider to access its constructor.

**public** **class** BuilderPatternExample {  
    
    **public** **static** **void** main(**String** args[]) {  
        
        *//Creating object using Builder pattern in java*  
        Cake whiteCake = **new**Cake.Builder().sugar(1).butter(0.5).  eggs(2).vanila(2).flour(1.5). bakingpowder(0.75).milk(0.5).build();  
        
        *//Cake is ready to eat :)*  
        **System**.out.println(whiteCake);  
    }  
}  
  
**class** Cake {  
  
    **private** **final** **double** sugar;   *//cup*  
    **private** **final** **double** butter;  *//cup*  
    **private** **final** **int** eggs;  
    **private** **final** **int** vanila;     *//spoon*  
    **private** **final** **double** flour;   *//cup*  
    **private** **final** **double** bakingpowder; *//spoon*  
    **private** **final** **double** milk;  *//cup*  
    **private** **final** **int** cherry;  
  
    **public** **static** **class** Builder {  
  
        **private** **double** sugar;   *//cup*  
        **private** **double** butter;  *//cup*  
        **private** **int** eggs;  
        **private** **int** vanila;     *//spoon*  
        **private** **double** flour;   *//cup*  
        **private** **double** bakingpowder; *//spoon*  
        **private** **double** milk;  *//cup*  
        **private** **int** cherry;  
  
        *//builder methods for setting property*  
        **public** Builder sugar(**double** cup){**this**.sugar = cup; **return** **this**; }  
        **public** Builder butter(**double** cup){**this**.butter = cup; **return** **this**; }  
        **public** Builder eggs(**int** number){**this**.eggs = number; **return** **this**; }  
        **public** Builder vanila(**int** spoon){**this**.vanila = spoon; **return** **this**; }  
        **public** Builder flour(**double** cup){**this**.flour = cup; **return** **this**; }  
        **public** Builder bakingpowder(**double** spoon){**this**.sugar = spoon; **return** **this**; }  
        **public** Builder milk(**double** cup){**this**.milk = cup; **return** **this**; }  
        **public** Builder cherry(**int** number){**this**.cherry = number; **return** **this**; }  
        
        
        *//return fully build object*  
        **public** Cake build() {  
            **return** **new** Cake(**this**);  
        }  
    }  
  
    *//private constructor to enforce object creation through builder*  
    **private** Cake(Builder builder) {  
        **this**.sugar = builder.sugar;  
        **this**.butter = builder.butter;  
        **this**.eggs = builder.eggs;  
        **this**.vanila = builder.vanila;  
        **this**.flour = builder.flour;  
        **this**.bakingpowder = builder.bakingpowder;  
        **this**.milk = builder.milk;  
        **this**.cherry = builder.cherry;         
    }  
  
    @**Override**  
    **public** **String** toString() {  
        **return** "Cake{" + "sugar=" + sugar + ", butter=" + butter + ", eggs=" + eggs + ", vanila=" + vanila + ", flour=" + flour + ", bakingpowder=" + bakingpowder + ", milk=" + milk + ", cherry=" + cherry + '}';  
  
    }   
    
}  
  
**Output:**  
Cake{sugar=0.75, butter=0.5, eggs=2, vanila=2, flour=1.5, bakingpowder=0.0, milk=0.5, cherry=0}

## Builder design pattern in Java – Pros and Cons

Live everything Builder pattern also has some disadvantages, but if you look at below, advantages clearly outnumber disadvantages of Builder design pattern. Any way here are few advantages and disadvantage of Builder design pattern for creating objects in Java.

**Advantages:**

1) More maintainable if number of fields required to create object is more than 4 or 5.

2) less error-prone as user will know what they are passing because of explicit method call.

3) More robust as only fully constructed object will be available to client.

**Disadvantages:**

1) Verbose (means long-winded) and code duplication as Builder needs to copy all fields from Original or Item class.

### When to use Builder Design pattern in Java

Builder Design pattern is a creational pattern and should be used when number of parameter required in constructor is more than manageable usually 4 or at most 5. Don't confuse with **Builder and Factory pattern** there is an obvious difference between Builder and Factory pattern, as Factory can be used to create different implementation of same interface but Builder is tied up with its Container class and only returns object of Outer class.