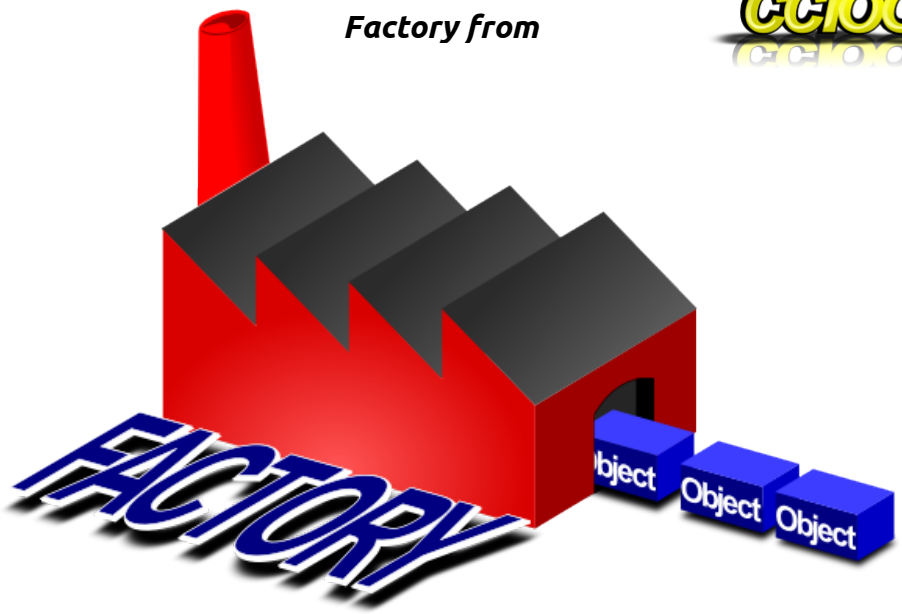


Factory tool

Introduction

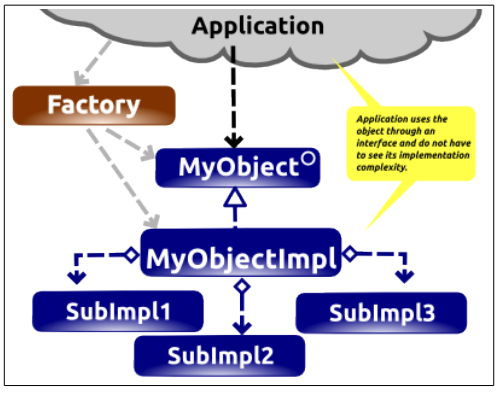
This article describes the different Factory functions provided by **ccTools**. It also goes beyond by providing cases on which this pattern has to be used and also the one on which not.

First let be clear, **ccTools** does not provide the basic Factory pattern but more precisely an abstract factory one. For an easier use of the library and reading of this article, the shorter term Factory is used instead of abstract factory.

ccTools is a library providing utilities classes extending Java API. It is not domain specific and provide features on generalist computing issues. It contains generic factory, Corba servant, asynchronous junit extension, etc..

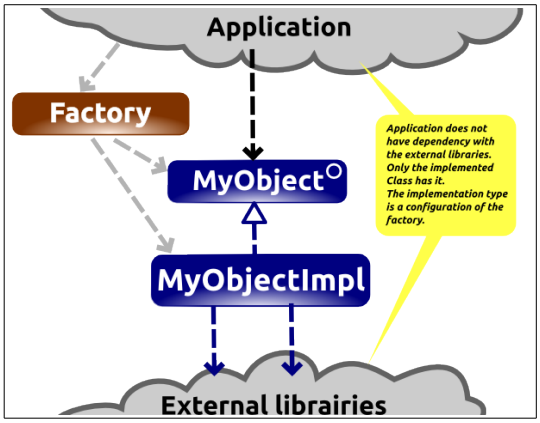
Factory pattern description

This pattern main purpose is to allocate new object instances and using it via a given interface. This way the implementation is hidden, on purpose, in order :

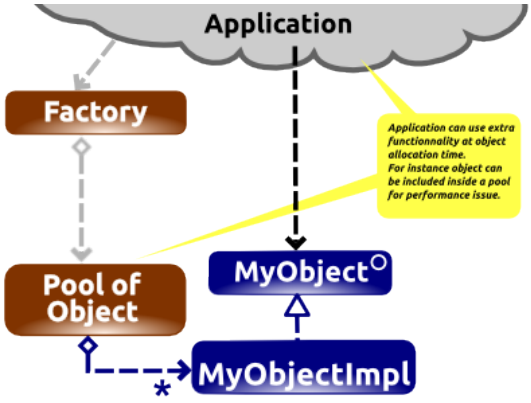


- ***to hide an implementation complexity***. For instance an object can provide very simple function to their users and in a same time it can have a very complex implementation. With class, there is not split between what can be used and how it is done. The user is in contact of the complexity where he has only to use few methods. This kind of classes can have several aggregations, complex dependency etc.. As the user does not

need to know about it, having an Interface and an implementation class is a great solution to avoid unnecessary complexity. These way your domain source can be split from the technical one.



- ***to simply manage dependency*** by not spreading external library types. For instance let's say that you have an application which has to display map. Let's say that you have a very nice and very fast map rendering library provided by a very talented commercial company but unfortunately their licence is very expense, which worth it in regard of their product quality. As your development team is around 60 persons, buying 60 developpers licence will cost you a fortune. A simple solution is to use an Interface with an implementation managed by a factory. This way, you can create another map implementation using a free open source map library which is far enough for most your developers and thus save lots of money. The choice of which implementation will be set can easily be done by a parameter.



- ***to add or perform specific behaviour*** at allocation time. Let's say that you have to work on a Near-Real-time application with very large number of object at a high rate. With this kind of application the garbage collection can be in trouble to scope a such high number of object creation. A solution can be to use recycled object. Where you allocate at start-up the maximum number of object that your system will use. When your application needs an object, the factory can give it an already existing object. When it does not need it any more, just keep the instance until another object is needed, without removing reference. This way the garbage collector will not consume much cpu on it.

Example for both Factory Type

For the two factory types descriptions, the following interface and implementation type are used :

```
package org.capcaval.cctools.factory._sample.generic;
```

```
public interface Greeter {
 String sayHello(String name);
}
```

Greeter interface

```
package org.capcaval.cctools.factory._sample.generic;

public class GreeterImpl implements Greeter{

 @Override
 public String sayHello(String name) {
  return "Hello " + name;
 }
}
```

Greeter implementation

This a simple interface implementation couple which is able to greet any input inside the console.

Generic Factory

This is the easiest and the fastest factory to be used. You have only to specify the interface and implementation type of your object and this is all. The interface type is set as the generic type of the factory and the implementation is set as a parameter to the "***newGenericFactory***" method call. To do so inside your Object interface just add the following line :

```
public static GenericFactory<Greeter> factory =
FactoryTools.newGenericFactory(GreeterImpl.class);
```

This way it can used as simply as :

```
package org.capcaval.cctools.factory._sample.generic;


public class GreeterMain {

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

```
  // Create an instance of Greeter
  Greeter g = Greeter.factory.newInstance();
  // Use it
  System.out.println(g.sayHello("world"));
 }
}
```

Use of generic factory sample

The console output of this sample is **"Hello world"**.

Still being generic, you are able to put as many parameters as you want, for object creation. To do so, your object implementation has to have the identical constructor signature. If not a run-time error will be raised.

Specific Factory

This one let you write the factory the way you want it. You can write your own custom factory interface without restriction. This way you can pass to the factory parameters to customize object creation. See the Greeter interface with a custom factory interface :

```
package org.capcaval.cctools.factory._sample.specific;

import org.capcaval.cctools.factory.FactoryTools;
import
org.capcaval.cctools.factory._sample.specific.EnglishGreete
rImpl.EnglishGreeterFactoryImpl;

public interface Greteer{

// associate the factory to the object type with static
member
 GreeterFactory factory = FactoryTools.newFactory(
       GreeterFactory.class, // interface of the factory
       new EnglishGreeterFactoryImpl()); // default factory
implementation instance

 String sayHelloTo(String name);

 // a specific factory with parameters
 public interface GreeterFactory{
  public Greteer newGreteer(String postMessage);
 }
}
```

interface with a custom factory

Inside the implementation you can allocate and customize object the way you want.

```
package org.capcaval.cctools.factory._sample.specific;

public class EnglishGreeterImpl implements Greteer{

 private String postMesage;
```

```
 public EnglishGreeterImpl(String postMessage) {
  this.postMesage = postMessage;
 }

 @Override
 public String sayHelloTo(String name) {
  return "Hello " + name + this.postMesage;
 }

 static public class EnglishGreeterFactoryImpl implements
GreeterFactory{

  @Override
  public Greteer newGreteer(String postMessage) {
       return new EnglishGreeterImpl(postMessage);
  }
 }
}
```

Now the customs factory interface can be used :

```
package org.capcaval.cctools.factory._sample.specific;

import
org.capcaval.cctools.factory._sample.specific.Greeter;

public class SpecificFactoryMain {

 public static void main(String[] args) {
  // create a greeter instance with a custom end
  Greeter greeter = Greeter.factory.newGreteer("my old
Chap!");
  // just greet
```

```
    System.out.println(greeter.sayHelloTo("Roger"));

  }
}
```

The console output of this sample is **"Hello Roger my old Chap!"**.

Mutable Factory

Both factories are Mutable that means that you are able to change dynamically it. For instance from the previous example if you prefer to have an greeting like **"G'day Roger"**, just use the class "**FactoryTools**" to change the default factory English implementation with a new Aussie one, as the following example.

```
FactoryTools.setFactoryImplementationInstance(
      Greeter.GreeterFactory.class,
      new AussieGreeterImpl.AussieGreeterFactoryImpl());

// create a new greeter instance from a different factory
Greeter ozGreeter = Greeter.factory.newGreteer();

System.out.println(ozGreeter.sayHelloTo("Roger"));
```

FactoryTools example

For the Generic Factory you can specify directly the Object implementation that you want to be created. To do so, still in "**FactoryTools**", just use the method "**setObjectImplementationType**".

Recycling Object

Generic factory provides you the ability to recycle object. This is a good feature when you create and delete lots of object at a high rate. This way all object instances are created once, and reduce Garbage Collector activity.

```
// create a pool with 1000 instance in it
MyObject.factory.setObjectPoolSize(1000);

// get one of the 1000 instance
 MyObject object = MyObject.factory.newInstance();

….
// When finish with do not forget to release instance
```

```
// Otherwise you are creating a memory leak
 MyObject.factory.releaseInstance(greeter);
}
```

Conclusion

The following table resume the differences between the two types of factory. Choose from it depending to your needs.

	Generic Factory	Specific factory
Easy to use	++	+
Mutable factory	√	√
Recycling Object	√	x
Specific factory interface	+	++

From my experience, all the time that I have learnt a new feature, I have tends to over use it. For factories, I did it for, what I call, Domain Object. Domain Objects are object type used at the highest functional domain inside your software. For instance for a car rental software a “vehicle” class can be a domain object or for a Bank software an “Account” or a “Client” can also be domain objects. The problem is that kind of object does not match any of the three main criteria for factory usage : hide a complex implementation, manage library dependency and perform specific function at allocation time. Basically this kind of object shall not have dependency with any library, shall not have several implementation and it shall not have an implementation complexity different to their interface complexity.
Writing an implementation class with its interface and with its Factory is not an insignificant effort, that why for such need I recommend to use POJO(Plain Old Java Object) for Domain object.
Another kind of object where there is a better abstraction to be used as structural Object. The ones which are corresponding to your application core. For this ones it is better to have an abstraction which provides communication features. A

To finish, these two kinds of factory have been created to help you coding faster by not rewriting the factory technical code and to be able to focus on your domain code. The main purpose it to do it as simple and type safe as possible.