# Atelier 1

# Améliorer la qualité de son application Java

## Java Singleton Design Pattern Best Practices with Examples

Java Singleton Pattern is one of the Gangs of Four Design patterns and comes in the Creational Design Pattern category. From the definition, it seems to be a very simple design pattern but when it comes to implementation, it comes with a lot of implementation concerns. The implementation of Java Singleton pattern has always been a controversial topic among developers. Here we will learn about Singleton design pattern principles, different ways to implement Singleton design pattern and some of the best practices for it's usage.

Singleton pattern restricts the instantiation of a class and ensures that only one instance of the class exists in the java virtual machine. The singleton class must provide a global access point to get the instance of the class. Singleton pattern is used for logging, drivers objects, caching and thread pool.

Singleton design pattern is also used in other design patterns like Abstract Factory, Builder, Prototype, Facade etc. Singleton design pattern is used in core java classes also, for example java.lang.Runtime, java.awt.Desktop.

To implement Singleton pattern, we have different approaches but all of them have following common concepts.

- Private constructor to restrict instantiation of the class from other classes.
- Private static variable of the same class that is the only instance of the class.
- Public static method that returns the instance of the class, this is the global access point for outer world to get the instance of the singleton class.

In further sections, we will learn different approaches of Singleton pattern implementation and design concerns with the implementation.

1. Eager initialization
2. Static block initialization
3. Lazy Initialization
4. Thread Safe Singleton
5. Bill Pugh Singleton Implementation
6. Using Reflection to destroy Singleton Pattern
7. Enum Singleton
8. Serialization and Singleton

# Eager initialization

In eager initialization, the instance of Singleton Class is created at the time of class loading, this is the easiest method to create a singleton class but it has a drawback that instance is created even though client application might not be using it.

Here is the implementation of static initialization singleton class.

```
package com.journaldev.singleton;
public class EagerInitializedSingleton {
    private static final EagerInitializedSingleton instance =
                                new EagerInitializedSingleton();
```

```
    //private constructor to avoid client applications to use constructor
    private EagerInitializedSingleton(){}
    public static EagerInitializedSingleton getInstance(){
        return instance;
    }
}
```

If your singleton class is not using a lot of resources, this is the approach to use. But in most of the scenarios, Singleton classes are created for resources such as File System, Database connections etc and we should avoid the instantiation until unless client calls the **getInstance** method. Also this method doesn't provide any options for exception handling.

## Static block initialization

**Static block** initialization implementation is similar to eager initialization, except that instance of class is created in the static block that provides option for **exception handling**.

```
package com.journaldev.singleton;
public class StaticBlockSingleton {
    private static StaticBlockSingleton instance;
    private StaticBlockSingleton(){}
    //static block initialization for exception handling
    static{
        try{
            instance = new StaticBlockSingleton();
        }catch(Exception e){
            throw new RuntimeException("Exception occured in
creating singleton instance");
        }
    }
    public static StaticBlockSingleton getInstance(){
        return instance;
    }
}
```

Both eager initialization and static block initialization creates the instance even before it's being used and that is not the best practice to use. So in further sections, we will learn how to create Singleton class that supports lazy initialization.

## Lazy Initialization

Lazy initialization method to implement Singleton pattern creates the instance in the global access method. Here is the sample code for creating Singleton class with this approach.

```
package com.journaldev.singleton;
public class LazyInitializedSingleton {
    private static LazyInitializedSingleton instance;
    private LazyInitializedSingleton(){}
    public static LazyInitializedSingleton getInstance(){
        if(instance == null){
            instance = new LazyInitializedSingleton();
        }
        return instance;
    }
}
```

The above implementation works fine incase of single threaded environment but when it comes to multithreaded systems, it can cause issues if multiple threads are inside the if loop at the same time. It will destroy the singleton pattern and both threads will get the different instances of singleton class. In next section, we will see different ways to create a **thread-safe** singleton class.

## Thread Safe Singleton

The easier way to create a thread-safe singleton class is to make the global access method **synchronized**, so that only one thread can execute this method at a time. General implementation of this approach is like the below class.

```
package com.journaldev.singleton;
public class ThreadSafeSingleton {
    private static ThreadSafeSingleton instance;
    private ThreadSafeSingleton(){}
    public static synchronized ThreadSafeSingleton getInstance(){
        if(instance == null){
            instance = new ThreadSafeSingleton();
        }
        return instance;
    }
}
```

Above implementation works fine and provides thread-safety but it reduces the performance because of cost associated with the synchronized method, although we need it only for the first few threads who might create the separate instances. To avoid this extra overhead every time, **double checked locking** principle is used. In this approach, the synchronized block is used inside the if condition with an additional check to ensure that only one instance of singleton class is created.

Below code snippet provides the double checked locking implementation.

```
public static ThreadSafeSingleton getInstanceUsingDoubleLocking(){
    if(instance == null){
        synchronized (ThreadSafeSingleton.class) {
            if(instance == null){
                instance = new ThreadSafeSingleton();
            }
        }
    }
    return instance;
}
```

## Bill Pugh Singleton Implementation

Prior to Java 5, java memory model had a lot of issues and above approaches used to fail in certain scenarios where too many threads try to get the instance of the Singleton class simultaneously. So Bill Pugh came up with a different approach to create the Singleton class using a inner static helper class. The Bill Pugh Singleton implementation goes like this;

```
package com.journaldev.singleton;
public class BillPughSingleton {
    private BillPughSingleton(){}
    private static class SingletonHelper{
        private static final BillPughSingleton INSTANCE = new
                                                BillPughSingleton();
    }
    public static BillPughSingleton getInstance(){
        return SingletonHelper.INSTANCE;
    }
}
```

Notice the **private inner static class** that contains the instance of the singleton class. When the singleton class is loaded, SingletonHelper class is not loaded into memory and only when someone calls the getInstance method, this class gets loaded and creates the Singleton class instance.

This is the most widely used approach for Singleton class as it doesn't require synchronization.

## Dependency Injection avec Spring Core

Pour des besoins de démonstration nous définissons le domaine d'application « gestion des ouvertures de comptes » (*Customer Accounts Management*) avec deux classes qui représentent le client (*Customer*) et le compte (*Account*) comme montré dans le code 2.1.

Code 2.1

```
package net.formation.springav.domain;
...
public class Customer {

 private long internalId;
 private String name;
 private Set<Account> accounts;
 ...}
```

```
package net.formation.springav.domain;

public class Account {

 private long internalId;
 private BigDecimal balance;
 private Customer customer;
```

Nous définissons par la suite deux interfaces (DAO – Data Access Objects) pour les accès aux données et leurs implémentations mémoire (à base de Collections Java) respectives.

Code 2.2

```
public interface IAccountsDao {
 public abstract Set<Account> loadAll();
 public abstract Account find(long id);
 public abstract int insert(Account account);
 public abstract int update(Account account);
 public abstract int delete(Account account);
}
public interface ICustomersDao {
 public abstract Set<Customer> loadAll();
 public abstract Customer find(long id);
 public abstract int insert(Customer customer);
 public abstract int update(Customer customer);
 public abstract int delete(Customer customer);

 public abstract int register(Customer customer, Account account);
}
```

```
public class MemoryCustomersDao implements ICustomersDao {
 private Set<Customer> customersRegistry;
 private Sets sets;
 ...}
public class MemoryAccountsDao implements IAccountsDao {
 private Set<Customer> customersRegistry;
 private Sets sets;
```

Nous proposons deux programmes faisant office de clients pour notre interface DAO. Uniquement le second (*Code 2.4*) fait appel aux fonctions d'inversion de contrôle apportées par Spring Framework.

Dans le code 2.3 nous retrouvons le scénario habituel où un objet est instancié à partir de sa définition de classe et aligné, de préférence, sur une interface préalablement établie. Nous montrons le schéma d'une portion des dépendances résultantes dans la figure 2.1.

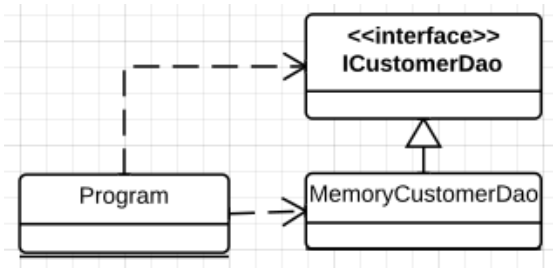


**Figure 2.1 - Dépendances sans Spring**

Code 2.3

```
public class Program {
private IAccountsDao accountsDao;
private ICustomersDao customersDao;


      public Program() {
...
this.accountsDao = new MemoryAccountsDao();
((MemoryAccountsDao)this.accountsDao).setCustomersRegistry(this.customersRegistry);
((MemoryAccountsDao)this.accountsDao).setSets(this.sets);

this.customersDao = new MemoryCustomersDao();

               ((MemoryCustomersDao)this.customersDao).setCustomersRegistry(this.customersRegistry);
((MemoryCustomersDao)this.customersDao).setSets(this.sets);
}
     public static void main(String[] args) {
Program p = new Program();

p.customersDao.insert(new Customer(1, "Frantz Kafka", null));
p.customersDao.insert(new Customer(2, "Martin Heidegger", null));

p.customersDao.register(new Customer(1, null, null), new Account(1,new BigDecimal(300), null));
p.customersDao.register(new Customer(3, "Eugene Ionesco", null), new Account(2,new BigDecimal(500), null));
p.accountsDao.insert(new Account(3,new BigDecimal(600), new Customer(1,null, null)));

for(Customer c : p.customersRegistry)
{
System.out.println(c.getInternalId()+" - "+c.getName()+" ("+c.getAccounts().size()+")");
for(Account a : c.getAccounts())
 System.out.println("--- " +a.getInternalId()+" - "+a.getBalance());
}

p.accountsDao.delete(new Account(1, null, new Customer(1, null, null)));
}
```

Code 2.4

```
public class Program {

private static final String APPLICATION_CONTEXT_XML = "applicationContext.xml";
private IAccountsDao accountsDao;
private ICustomersDao customersDao;

public static void main(String[] args) {

Resource res = new ClassPathResource(APPLICATION_CONTEXT_XML);
BeanFactory factory = new XmlBeanFactory(res);
               Program p = ac.getBean("program",Program.class);
for(Customer c : p.customersRegistry)
{
System.out.println(c.getInternalId()+" - "+c.getName()+" ("+c.getAccounts().size()+")");
for(Account a : c.getAccounts())
 System.out.println("--- " +a.getInternalId()+" - "+a.getBalance());
}
p.accountsDao.delete(new Account(1, null, new Customer(1, null, null)));
}
```

Le code 2.4 démontre le recours à deux interfaces Spring qui sont **org.springframework .core.io. Resource** et **org.springframework.beans.factory.BeanFactory** pour récupérer l'objet de type **MemoryCustomerDao.** Le schéma de dépendances prend la forme illustrée par la figure 2.2. Nous notons la suppression de lien d'instanciation entre l'application (Program) et l'objet de dépendance (MemoryCustomerDao). La gestion de ce lien a été déléguée au Framework (BeanFactory). Ceci permettra d'agir directement et uniquement sur Spring dans le cadre de mesures de réutilisation (*La classe Program n'a plus besoin d'être re-factorisée ou reprogrammée pour changer ou agir sur ses dépendances*).

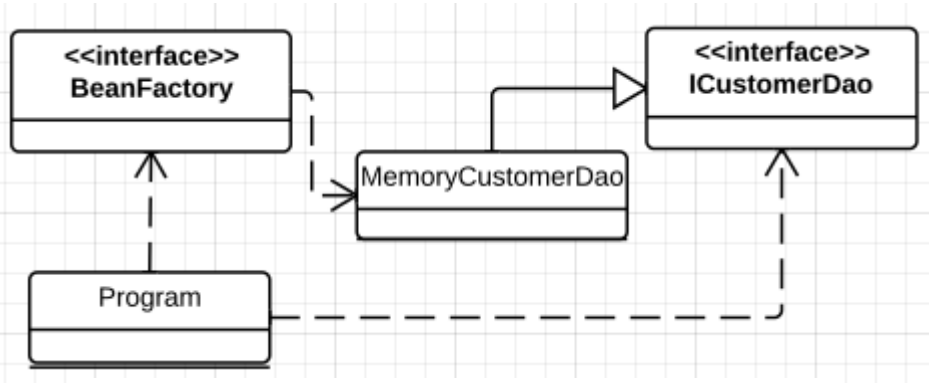


**Figure 2.2 - Dépendances avec Spring**

L'interface **BeanFactory** du Framework Spring est l'élément central pour les opérations d'inversion de contrôle. Elle permet d'instancier et de maintenir les composants applicatifs que l'on nomme Beans (*en emprunt au standard des Java Beans*). Dans notre exemple le Bean que nous utilisons est celui répondant au contrat du programme principal Program et portant le nom d'identification « program » au niveau du Framework Spring (`Program p = ac.getBean("program",Program.class);`)

Pour agir sur le modèle de gestion, d'instanciation et de maintien des différents Beans applicatifs nous recourons à des implémentations de la BeanFactory dont notamment la **XMLBeanFactory** que nous invoquons dans l'exemple illustratif.

XMLBeanFactory charge les Beans applicatifs à travers un modèle XML traduit par une ressource déclarée (*ClassPathResource*). Pour créer cette ressource nous procédons comme suit :

1-Transformer le projet Java en un projet Spring : **Clic Droit > Spring Tools > Add Spring Project Nature**

2-**Clic Droit sur le projet > New > Spring Beans Configuration File**

3-Préciser le répertoire du fichier : **src/main/resources** et son nom : **applicationContext.xml**

4-Préciser la version du schéma XSD Beans : **version 3.0**

5-Sur le répertoire **src/main/resources :** Clic Droit > Build Path > Define as a source folder

6-Editer le fichier comme montré dans le code 2.5

Code 2.5

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd" ...>

<bean id="accountsDao" class="net.formation.springav.dao.MemoryAccountsDao">
 <property ref="setUtils" name="sets"></property>
 <property name="customersRegistry" ref="customersRegistry"></property>
</bean>
        ...
<bean id="program" class="net.formation.springav.cli.Program">
 <property name="accountsDao" ref="accountsDao"></property>
 <property name="customersDao" ref="customersDao"></property>
 <property name="customersRegistry" ref="customersRegistry"></property>
 <property name="sets" ref="setUtils"></property>
</bean>
</beans>
```

A ce niveau notre exemple est fin prêt et complètement fonctionnel. Le Bean Program est défini dans la ressource XML et chargé avec ses dépendances dans le programme principal à travers XMLBeansFactory en utilisant son identifiant Spring « program ». Il est possible de charger le Bean à travers son nom de classe (S'il n'existe pas d'ambiguïté) comme suit :
*Program program = (Program) factory.getBean(Program.**class**);*

Cette méthode est évidemment à éviter puisqu'elle n'encourage pas la réutilisation et fait réapparaitre les dépendances dont l'objectif est de les faire disparaitre. D'autres méthodes sont prévues par la BeanFactory dont celles qui renseignent sur le type et la nature du Bean, son existence, etc.

## 1. Notions détaillées sur l'IoC avec Spring

### 1.1. Gestion des scopes

Dans le Framework Spring la notion des « scopes » est introduite comme étant le modèle d'instanciation observé par le Bean applicatif. Les deux principaux scopes sont le « singleton » et le « prototype » et à ces derniers d'autres viennent s'ajouter dans le contexte du multithreading et de la gestion des sessions http.

Le scope « singleton » est celui utilisé par défaut dans Spring et garantit, comme son nom l'indique, qu'une seule instance d'un Bean applicatif soit créée lors d'une utilisation du conteneur. Le scope « prototype » quant à lui fait qu'à chaque sollicitation du Bean en question, une nouvelle instance soit créée.

Dans le code 3.1, nous récupérons 3 références d'objet (Bean) de type customersDao et affichons leurs adresses JVM (*méthode toString qui renvoie vers le hashcode par défaut qui n'est autre que l'emplacement mémoire au niveau de la JVM de l'objet considéré*) pour vérifier s'il s'agit du même objet ou pas.

Code 3.1

```
Resource res = new ClassPathResource(APPLICATION_CONTEXT_XML);
BeanFactory factory = new XmlBeanFactory(res);
ICustomersDao cdao1 = factory.getBean("customersDao",ICustomersDao.class);
ICustomersDao cdao2 = factory.getBean("customersDao",ICustomersDao.class);
ICustomersDao cdao3 = factory.getBean("customersDao",ICustomersDao.class);

System.out.println(cdao1);
System.out.println(cdao2);
System.out.println(cdao3);
```

Dans une première exécution nous récupérons l'affichage suivant qui valide qu'il s'agit bel et bien d'un objet unique :

```
net.formation.springav.dao.MemoryCustomersDao@16fe0f4
net.formation.springav.dao.MemoryCustomersDao@16fe0f4
net.formation.springav.dao.MemoryCustomersDao@16fe0f4
```

En second lieu nous agissons sur la configuration applicationContext.xml pour attribuer la propriété « prototype » à notre Bean « customersDao» :

```
<bean id="customersDao" class="net.formation.springav.dao.MemoryCustomersDao" scope="prototype">
```

Lors de la réexécution nous obtenons un affichage différent montrant que chacun des objets récupérés est différent de l'autre :

```
net.formation.springav.dao.MemoryCustomersDao@16fe0f4
net.formation.springav.dao.MemoryCustomersDao@19d0a1
net.formation.springav.dao.MemoryCustomersDao@d02b51
```

### 1.2. Méthodes d'instanciation

Par défaut, le Framework Spring utilise réflexivement le constructeur d'une classe définie sous forme de Bean pour l'instanciation. Le constructeur sans arguments est invoqué si aucune précision contraires n'est fournie (*injection de dépendances via constructeurs avec arguments - cf. 3.3*).

Deux autres approches d'instanciation sont prévues. La première concerne l'utilisation d'une méthode statique de type fabrique (Static Factory Method) et la seconde concerne l'utilisation d'une méthode sur un objet de type fabrique (Instance Factory Method)

Pour la première méthode nous définissons un bean **customersDaoStaticFactory** conformément à la configuration ci-après :

```
<bean id="customersDaoStaticFactory" class="net.formation.springav.dao.StaticMemoryCustomersDao"
                                     init-method="getMemoryAccountsDao">
<property name="sets" ref="setUtils"></property>
<property name="customersRegistry" ref="customersRegistry"></property>
</bean>
```

La classe du Bean **customersDaoStaticFactory** est définie comme montré dans le code 3.2. La deuxième partie du code représente un programme instanciant 4 références du Bean en question. En mode prototype l'instanciation est observée 4 fois de suite (*contre 1 fois si le Bean est configuré en mode singleton*)

Code 3.2

```
public class StaticMemoryCustomersDao extends MemoryCustomersDao {
public static MemoryAccountsDao getMemoryAccountsDao(){
System.out.println("Static Instanciation");
return new MemoryAccountsDao();
}
}
```

```
BeanFactory factory = new XmlBeanFactory(res);
ICustomersDao cdao1 = factory.getBean("customersDaoStaticFactory",ICustomersDao.class);
ICustomersDao cdao2 = factory.getBean("customersDaoStaticFactory",ICustomersDao.class);
ICustomersDao cdao3 = factory.getBean("customersDaoStaticFactory",ICustomersDao.class);
ICustomersDao cdao4 = factory.getBean("customersDaoStaticFactory",ICustomersDao.class);IBooksDAO
```

Pour la seconde méthode (Instance Factory Method), nous définissons un Bean d'instanciation (*customersDaoFactory*) et y faisons référence dans les Bean applicatif (*customersDaoPooled*) comme suit :

```
<bean id="customersDaoPooled" class="net.formation.springav.dao.MemoryCustomersDao"
                              factory-bean="customersDaoFactory" factory-method="getICustomersDao">
<property name="sets" ref="setUtils"></property>
<property name="customersRegistry" ref="customersRegistry"></property>
</bean>

<bean id="customersDaoFactory" class="net.formation.springav.dao.customersDaoFactory"/>
```

Code 3.3

L'idée est de maintenir un Pool d'objets de type ICustomersDao à travers la bibliothèque Apache Commons Pool (cf. Code 3.3)

```
public class customersDaoFactory {
/**
 * Main Object Pool for ICustomersDao
 */
private ObjectPool<ICustomersDao> customersDaoPool ;

public customersDaoFactory()  {
  this.customersDaoPool = new GenericObjectPool<ICustomersDao>(
 new BasePoolableObjectFactory<ICustomersDao>() {
 @Override
 public ICustomersDao makeObject() {
 return new MemoryCustomersDao();}
 });

 ((GenericObjectPool<ICustomersDao>)this.customersDaoPool).setMaxActive(2);
}
/**
 * Factory Method for Borrowing ICustomersDao Object from the main Pool
 */
public  ICustomersDao getICustomersDao () throws ...{
System.out.println("Borrowed");
return this.customersDaoPool.borrowObject();
}
}
```

### 1.3. Méthodes d'injection de dépendance

Les dépendances injectées (Collaborateurs) et les valeurs initialisées au niveau du conteneur sont, jusqu'à présent, composées à travers les méthodes **setter**. Il s'agit là de l'une des deux méthodes d'injection connues pour le Framework Spring. La deuxième méthode, que nous étudions dans cette section, est celle qui recourt aux constructeurs et à leurs arguments.

Nous prenons l'exemple d'un Bean de type CustomersDao et le constructeur correspondant :

```
public MemoryCustomersDao(Set<Customer> customersRegistry, Sets sets) {
super();
      System.out.println("Using Constructor");
this.customersRegistry = customersRegistry;
this.sets = sets;
}
<bean id="constCustomersDao" class="net.formation.springav.dao.MemoryCustomersDao">
<constructor-arg index="0"  ref="customersRegistry"/>
<constructor-arg index="1"  ref="setUtils"/>
</bean>
```

Il arrive qu'un Bean définisse plusieurs constructeurs à un nombre égal d'arguments mais de types variant. Pour résoudre cette ambiguïté, Spring prévoit de préciser le type d'argument dans le fichier contexte :

```
<bean id="constCustomersDao" class="net.formation.springav.dao.MemoryCustomersDao" >
<constructor-arg index="0" type="java.util.Set" ref="customersRegistry"/>
<constructor-arg index="1" type="net.formation.springav.utils.Sets" ref="setUtils"/>
</bean>
```

Cette méthode d'injection de dépendance n'est pas préconisée car elle impose un lien d'ordre entre le Bean et sa configuration. Les noms d'argument ne sont pas conservés lors de la compilation du code Java vers du ByteCode et cela rend impossible leur référencement en *Runtime*. Néanmoins, dans sa version 3.0, Spring a introduit l'utilisation de l'annotation @ConstructorProperties (JDK 1.6) pour remédier à ce problème (*Notons que de par sa nature intrusive, nous évitons également cette méthode*) :

```
@ConstructorProperties({"customersRegistry","sets"})
public MemoryCustomersDao(Set<Customer> customersRegistry, Sets sets){…}
…
<bean id="constCustomersDaoWithProps" class="net.formation.springav.dao.MemoryCustomersDao">
<constructor-arg name="sets" type="net.formation.springav.utils.Sets" ref="setUtils"/>
<constructor-arg name="customersRegistry" ref="customersRegistry"/>
</bean>
```

### 1.4. L'interface ApplicationContext

L'interface **ApplicationContext** étend la **BeanFactory** pour y rajouter un nombre de fonctionnalités. Il est recommandé d'utiliser directement l'interface ApplicationContext (*« à moins qu'une bonne raison vous dissuade de le faire » pour reprendre l'expression de la documentation officielle de Spring*). Dans cette section nous étudions deux fonctionnalités de l'ApplicationContext qui sont l'internationalisation et la gestion d'évènements.

#### 1.4.1. Internationalisation

L'une des fonctionnalités d'ApplicationContext est la gestion des messages internationalisés à travers l'interface **MessageSource.**

L'idée est d'initialiser un Bean dont l'identifiant correspond à messageSource comme suit :

```
<bean name="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">
<property name="basename"  value="messages"/>
</bean>
```

La propriété *basename* (qui *pourrait être multiple*) précise l'identifiant du fichier de ressources résidant en classpath. Le nom de fichier est déduit à partir de cet identifiant (*exemple : messages.properties, messages_fr_FR.properties, …)*

Nous créons deux fichiers messages dont le contenu est :
messages.properties :
greetings=Hello Word \!\!

messages_fr_FR.properties :
greetings=Bonjour le Monde \!\!

Au niveau du programme principal nous initialisons l'objet ApplicationContext et récupérons le message « greeting » d'une manière localisée :

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("applicationContext-2.xml");
System.out.println(ctx.getMessage("greetings", null,"Default Greating",Locale.FRENCH));
```

Sur un autre plan, il est possible de rendre les Beans applicatifs sensibles à l'ApplicationContext en implémentant l'interface *org.springframework .context .Application ContextAware*. Ceci permet de récupérer l'ApplicationContext courant au niveau du Bean et de procéder, entre autre, à la récupération des messages internationalisés (*attention à la dépendance entre Bean et Framework Spring*)

Code 3.4

```
public class ACAwareGreetingsBean implements ApplicationContextAware {

private static final String DEFAULT_GREATING = "Default Greating";
private ApplicationContext ctx;
@Override
public void setApplicationContext(ApplicationContext applicationContext)
                            throws BeansException {
this.ctx = applicationContext;
}
public void sayHello(Locale l){
System.out.println(ctx.getMessage("greetings", null,DEFAULT_GREATING,l));
}
}
```

### 1.4.2.Gestion des évènements

Au niveau de l'interface ApplicationContext, il est possible d'adhérer à, de configurer et de personnaliser un mécanisme de gestion d'évènements conforme au Design Pattern Observer.

Pour se mettre à l'écoute d'un évènement il suffit d'implémenter l'interface *org. springframework. context. ApplicationListener< E extends ApplicationEvent >* et d'enregistrer l'objet sous forme de Bean.

Code 3.5

```
public class StandardApplicationListener implements ApplicationListener<ApplicationEvent> {

@Override
public void onApplicationEvent(ApplicationEvent e) {
        System.out.println(e.getSource());

if(e instanceof ContextRefreshedEvent)
System.out.println("Context Started with "
                +((ContextRefreshedEvent)e).getApplicationContext().getBeanDefinitionCount()
 +" Beans");
  }
}
```

Il est aussi possible d'enregistrer ses propres événements. Nous imaginons, par rapport à notre application exemple, que nous voudrions publier une notification à chaque fois qu'un compte est ouvert.

Nous commençons par concevoir l'évènement d'ouverture de compte :

Code 3.6

```
public class AccountRegistredEvent extends ApplicationEvent {
private static final long serialVersionUID = 10004L;

private Account accout;

public AccountRegistredEvent(Object source, Account added) {
super(source);
this.accout = added;
}
public Account getAccout() {
return accout;
}
}
```

Nous implémentons l'interface *org.springframework.context.ApplicationEventPublisherAware* au niveau du Bean EventBasedCustomersDao pour récupérer l'objet de publication d'évènement et l'utiliser à chaque opération d'insertion de nouveau compte :

Code 3.7

```
public class EventBasedCustomersDao extends MemoryCustomersDao
                                    implements ApplicationEventPublisherAware  {
private ApplicationEventPublisher aep;
@Override
public int register(Customer customer,Account account) {
aep.publishEvent(new AccountRegistredEvent(this, account));
return super.register(customer, account);
};
@Override
public void setApplicationEventPublisher(ApplicationEventPublisher applicationEventPublisher) {
this.aep = applicationEventPublisher;
}
}
```

## 2. La programmation par aspects : Spring AOP

La programmation par aspects complémente la programmation objet en œuvrant au niveau de la propriété de cohésion des composants formant un applicatif. Les méthodes de programmation par aspect permettent de structurer les opérations transversales en une unité modulaire parfaitement découplée des composantes sur lesquelles elle intervient. Un exemple typique est celui des opérations de journalisation. Pour implémenter un comportement de Log, nous utilisons un Framework, à l'instar de Log4J, de la manière suivante :

```
public class MaClasse {
private static final Logger logger = Logger.getLogger(MaClasse.class);

public boolean uneMethodeMetier() {
logger.log(Level.INFO, "Un Log Quelconque");
                ...}
        }
```

Notons que « MaClasse » avec son traitement purement métier dépend, néanmoins, du Framework Log4J et que ses méthodes sont contraintes à compromettre leur propriété de forte cohésion pour effectuer les traitements relatifs au Log. La programmation par aspects agit au niveau de ces structures pour rétablir la forte cohésion.

Pour ce faire, un nombre de notions sont introduites :

1-**Join Point** : Un point identifié par rapport à la progression du flot de contrôle d'un programme (*Généralement l'exécution d'une méthode*)

2-**PointCut** : Un ensemble de Join Point identifié par un prédicat (*Spring utilise le langage AspectJ Expression Language du Framework AOP AspectJ*)

3-**Advice** : Un comportement qui prend place au niveau d'un Join Point (*Exemple traitement de journalisation*)

4-**Target Object** : Objet subissant la conséquence d'un traitement par aspect (*Exemple : Une classe métier*)

5-**AOP Proxy** : Objet résultant de la composition entre Target Object et Aspects s'y rapportant

6-**Weaving (Tissage) :** Opération de composition entre Target Object et Aspects s'y rapportant

7-**Introduction :** Modification dans le contrat ou dans les éléments maintenus dans un Target Object

Au niveau du Framework Spring, un volet AOP élémentaire (*Mais puissant et suffisant pour une grande majorité des cas*) est présent depuis les premières versions. Les développeurs préféraient, historiquement, l'intégration du très connu Framework AOP AspectJ. Cette tendance n'est pas uniquement motivée par les possibilités additionnelles offertes par AspectJ (Compile Time Weaving, Join Points autres que l'exécution d'une méthode, …) mais aussi par la simplicité syntaxique (En XML ou en Annotations) comparée à celle des dispositifs Spring AOP.

Actuellement, et depuis la version 2.0 du Framework Spring, une intégration de la syntaxe par annotations d'AspectJ est fournie en plus d'une syntaxe XML nouvelle. Le moteur d'exécution reste celui de Spring AOP mais la méthode de configuration est reprise du Framework AspectJ. Dès lors, les raisons pour intégrer AspectJ deviennent moindres et spécifiques à des besoins pointus.

Dans cette section, nous étudions l'utilisation de la programmation par Aspects dans Spring et en continuité avec l'application Accounts Management pour implémenter le volet journalisation.

### Etape 1 – Déclaration des dépendances

Nous commençons par rajouter les dépendances nécessaires pour effectuer la configuration à la manière d'AspectJ (*Les dépendances de Spring-Aop sont rapportées avec Spring-Context*) :

```
<dependency>
<groupId>aspectj</groupId>
<artifactId>aspectjrt</artifactId>
<version>1.5.4</version>
```

```
</dependency>

<dependency>
<groupId>aspectj</groupId>
<artifactId>aspectjweaver</artifactId>
<version>1.5.4</version>
</dependency>
```

### Etape 2 – Création de l'aspect

Code 7.1

```
@Aspect
public class LogAspect {
private static final Logger logger = Logger.getLogger(LogAspect.class);


@AfterReturning("execution(* net.formation.springav.dao.HibernateCustomersDao.register(..)) " +
 "&& args(customer,account) && target(target)")
public void accountRegistering(Customer customer, Account account, Object target){
logger.log(Level.INFO, target.toString()+" - Registering Account Number : "
                                                        +account.getInternalId());
}
@AfterThrowing(pointcut="execution(* net.formation.springav.dao.HibernateCustomersDao.register(..))  "
                                        +"&& args(account)",
 throwing="exception")
public void batchUpdateException(RuntimeException exception, Account account){
logger.log(Level.ERROR, " - Error on Registering Account");
logger.log(Level.ERROR, exception.toString());
for(StackTraceElement ste : exception.getStackTrace()) logger.log(Level.ERROR, ste.toString());
}
}
```

### Etape 3 – Configuration des Aspects

Code 7.2

```
<aop:aspectj-autoproxy/>
<bean name="logAspect" class="net.formation.springav.aspects.LogAspect"/>
```