Spring Core : inversion de contrôle (IOC) et injection de dépendance

Mohammed OUANAN

m.ouanan@umi.ac.ma

Plan

Spring

Framework

- Plusieurs traductions
  - cadriciel
  - environnement de développement
  - cadre d'applications
  - ...
- { composants logiciels, conventions, structures... }
- Facilitant la réalisation d'une (partie de l') application
- Imposant une certaine structure, logique, syntaxe...

Plusieurs types de frameworks

- Frameworks applicatifs pour le développement d'applications web :
  - **Angular** pour **JavaScript**
  - **Spring** pour **Java**
  - **Symfony** pour **PHP**
  - ...
- Frameworks de présentation de contenu web :
  - **Bootstrap** pour **CSS**
  - ...
- Frameworks de persistance (**ORM**) comme **Hibernate**
- Frameworks de logging comme **Log4j**
- Frameworks de test comme **JUnit**
- ...

Spring

Framework : avantages

- Gain de temps
- Meilleure organisation de projet
- Respect de bonnes pratiques et utilisation de design patterns
- Meilleure efficacité
- Faciliter le travail d'équipe
- ...

Spring

Framework : inconvénients

- Complexité
- Apprentissage
- ...

Spring

Question?

Librairie et Framework, désignent-ils la même chose ?

Spring

Question

Librairie et Framework, désignent-ils la même chose ?

Réponse

Non

Spring

Librairie?

- { fonctions, classes, modules } réutilisables et indépendants
- Destiné à accomplir des tâches spécifiques dans un programme

Spring

Librairie

- { fonctions, classes, modules } réutilisables et indépendants
- Destiné à accomplir des tâches spécifiques dans un programme

Mais une librairie

- n'impose aucune structure
- ne génère pas de code
- ...

Spring

Spring Framework

- Framework, initialement conçu, pour le langage **Java**.
- Open-source.
- Créé par **Roderick JOHNSON** (appelé aussi **Rod JOHNSON**) en 2003 puis **SpringSource** et enfin **VMware**.
- Écrit en **Java**, **Kotlin** et **Groovy**
- Basé sur le concept : **inversion de contrôle** par la recherche et **l'injection de dépendance**.
- Objectif : proposer une solution plus simple et plus légère que celle de **JEE**.

Spring

Autres frameworks Java

- **JSF** (**J**ava **S**erver **F**aces) : technologie **JEE** standard créée par **Sun Microsystems** puis **Oracle Corporation**
- **Apache Struts** : projet open-source de la fondation **Apache** créé par **Craig R. McClanahan**
- **Grails** : projet open-source créé par **SpringSource**, connu maintenant sous le nom **Pivotal Software** et appartenant à **VMware**
- **Vert.x** : projet open-source soutenu par **Eclipse Foundation** et créé par **Tim Fox**
- ...

Spring

Principales versions de Spring Framework

- **Spring 1.0** (2003)
- **Spring 2.0** (2006) : intégration de **AspectJ**.
- **Spring 2.5** (2007) : intégration des annotations de configuration.
- **Spring 3.0** (2009) : intégration de **Java config**.
- **Spring 4.0** (2013) : intégration de **JPA 2.1**. (utilisation de **Java 7**)
- **Spring 5.0** (2016) : intégration de **Kotlin**. (utilisation de **Java 8** et **JUnit 5 Jupiter**).
- **Spring 6.0** (2022) : utilisation de **Java 17** et **Jakarta EE 9**

Spring

Spring vs JEE

- **Spring** ne respecte pas les spécifications **JEE**
- Mais, il utilise plusieurs **API JEE**
  - **Servlet**
  - **JSP**
  - **JMS**
  - **WebSocket**
  - ...

Spring

Spring s'appuie sur

- **IoC** (Inversion de contrôle) : assurée par la **DI** (Injection de dépendances
- **AOP** (Programmation Orientée Aspect)
- Abstraction

Spring

Architecture de Spring Framework

- **Spring Core**
  - Un conteneur de beans (**IoC Container**)
  - Une fabrique de beans (**Bean Factory**)
  - Un contexte (**Spring Context**)
  - ...
- **AOP** : Aspect Oriented Programming
- **Web** : **Servlet**, **WebSocket**...
- **Data** : **JDBC**, **ORM**...
- **Test**
- ...

Spring

Bean?

- Objet **Java** géré (création et configuration) par le conteneur de **Spring**.
- Configurable dans un fichier **XML** ou une classe **Java**
- Pouvant représenter différentes parties de l'application, telles que les services, les connexions à la base de données...
- **Spring** garantit que les dépendances d'un bean sont injectées correctement.

Spring

Dépendance entre objets?

Les objets de la classe C_1 dépendent des objets de la classe C_2 si :

- C_1 a un attribut objet de la classe C_2
- C_1 hérite de la classe C_2
- C_1 dépend d'un autre objet de type C_3 qui dépend d'un objet de type C_2
- Une méthode de C_1 appelle une méthode de C_2

Spring

En programmation objet classique

Le développeur :

- Instancie les objets nécessaires pour le fonctionnement de son application (avec l'opérateur `new`)
- Prépare les paramètres nécessaires pour instancier ses objets
- Définit les liens entre les objets

Spring

Avec Spring

- Couplage faible
- Injection de dépendance

Spring

Quand une classe A est lié à une classe B, on dit que la classe A est **fortement couplée** à la classe B. La classe A ne peut fonctionner qu'en présence de la classe B.
Si une nouvelle version de la classe B (soit B2), est crée, on est obligé de modifier dans la classe A.
Modifier une classe implique:

- Il faut disposer du code source.
- Il faut recompiler, déployer et distribuer la nouvelle application aux clients.
- Ce qui engendre un cauchemar au niveau de la maintenance de l'application

Spring

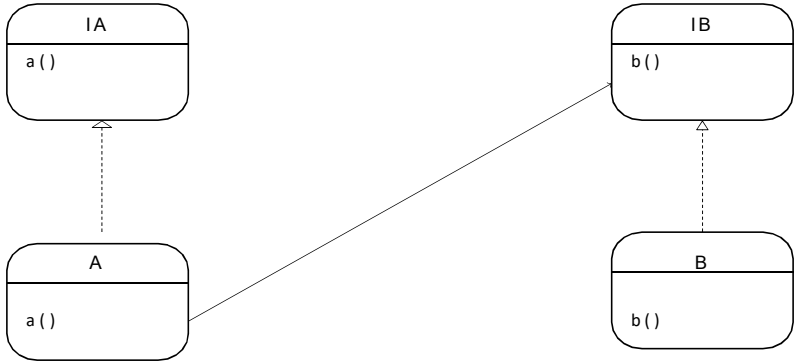
Contenu de la méthode `main`

```
main () {
  A obj = new A ();
  obj.a();
}
```

Contenu de la classe `A`

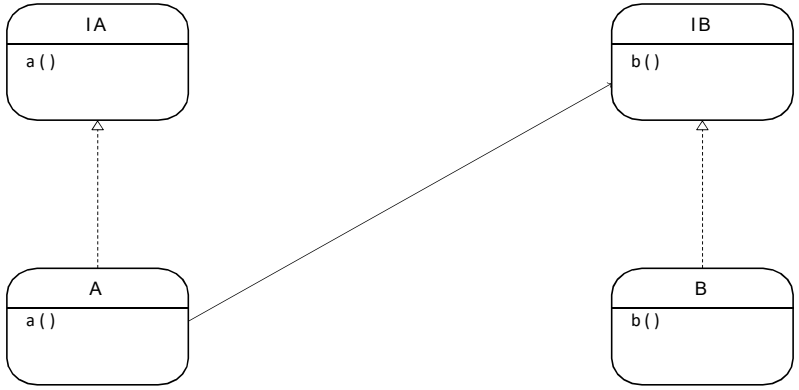
```
B obj = new B ();
a () {
  obj.b();
}
```

couplage faible



Pour utiliser le **couplage faible**, nous devons utiliser les interfaces.

- Considérons une classe A qui implémente une interface IA, et une classe B qui implémente une interface IB.
- Si la classe A est liée à l'interface IB par une association, on dit que le classe A et la classe B sont liées par un couplage faible.
- Cela signifie que la classe B peut fonctionner avec n'importe quelle classe qui implémente l'interface IA.
- En effet la classe B ne connait que l'interface IA. De ce fait n'importe quelle classe implémentant cette interface peut être associée à la classe B, sans qu'il soit nécéssaire de modifier quoi que se soit dans la classe B.
- Avec le couplage faible, nous pourrons créer des application fermée à la modification et ouvertes à l'extension.



Contenu de la méthode `main`

```
main () {
  IA ia;
  IB ib;
  ia.obj = ib;
  ia.a();
}
```

Contenu de la classe `A`

```
IB obj;
a () {
  obj.b();
}
```

Spring

Remarques

- On ne peut instancier les interfaces `IA` et `IB`
- Mais, on peut injecter les classes qui les implémentent pour créer les deux objets `ia` et `ib`

Spring

L'injection de dépendances est souvent la base de tout programme moderne.

• L'idée en résumé est de déporter la responsabilité de la liaison des composants du programme dans un framework afin de pouvoir facilement changer ces composants ou leur comportement.

• Parmi les leaders du marché Java, il y a Spring IoC, Guice, Dagger ou encore le standard « Java EE » CDI qui existe depuis Java EE 6.

• Spring IOC commence par lire un fichier XML qui déclare quelles sont différentes classes à instancier et d'assurer les dépendances entre les différentes instances.

• Quand on a besoin d'intégrer une nouvelle implémentation à une application, il suffirait de la déclarer dans le fichier xml de beans spring.

Spring

Inversion de contrôle par injection de dépendance

- Patron de conception:couramment utilisé en programmation pour découpler les composants et permettre une meilleure flexibilité et testabilité du code. Ce principe consiste à déléguer la responsabilité de la création et de l'injection des dépendances d'une classe à un conteneur externe ou à une infrastructure de gestion des dépendances.
- Permettant de dynamiser la gestion de dépendance entre objets
- Facilitant l'utilisation des composants
- Minimisant l'instanciation statique d'objets (avec l'opérateur `new`)

Spring

Spring est un conteneur léger (Lightweight Container)

- Instanciation d'objets définis dans un fichier de configuration **XML** ou dans une classe **Java**
- Pas besoin d'implémenter une quelconque interface pour être prises en charge par le framework

Spring

Spring est un conteneur léger (Lightweight Container)

- Instanciation d'objets définis dans un fichier de configuration **XML** ou dans une classe **Java**
- Pas besoin d'implémenter une quelconque interface pour être prises en charge par le framework

Remarque

JEE est un conteneur lourd (instancie seulement les classes qui implémentent certaines interfaces):Il permet l'injection de dépendances pour certains types de classes (ex : EJB, servlets, etc.)

Spring

3 manières différentes pour l'injection de dépendance avec Spring

- injection de dépendance via le constructeur
- injection de dépendance via les setters
- injection de dépendance via les interfaces et les annotations

Spring

3 manières différentes pour l'injection de dépendance avec Spring

- injection de dépendance via le constructeur
- injection de dépendance via les setters
- injection de dépendance via les interfaces et les annotations

Deux modes de configuration avec **Spring**

- en utilisant les XML (XML Config), ou
- en utilisant les classes de configuration (Java Config)

Spring

Intégrer Spring sous Eclipse

- Dans le menu `Help`, choisir `Eclipse Marketplace`
- Dans la zone de saisie `Find`, saisir `Spring tools` et attendre la fin de chargement
- Sélectionner `Spring Tools 4(aka Spring Tool Suite 4)`
- Puis cliquer sur `Install`
- Enfin attendre la fin d'installation et redémarrer **Eclipse**

Spring

Créons un projet Maven

- Aller `File > New > Maven Project`
- Cliquer sur `Next`
- Choisir `maven.archetype.quickstart`
- Remplir les champs
  - `Group Id` avec `org.eclipse`
  - `Artifact Id` avec `spring-ioc`
  - `Package` avec `org.eclipse.main`
- Valider

Spring

Ajoutons la dépendance `spring-context-support` **dans** `pom.xml`

```
<!-- https://mvnrepository.com/artifact/org.
   springframework/spring-context-support -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context-support</artifactId>
    <version>6.0.3</version>
</dependency>
```

Spring-context-support est un module de Spring qui fournie des fonctionnalités supplémentaires pour l'intégration avec des bibliothèques et services tiers (services externes fournis par des entreprises ou des organisations autres que celle qui développe une application), en étendant le module de base spring-context.
Il est particulièrement utile pour les applications qui nécessitent des fonctionnalités avancées de gestion des dépendances et d'intégration.

Spring

Créons une classe `Personne` **dans** `org.eclipse.model`

```
package org.eclipse.model;

public class Personne {
  private int id;
  private String nom;

  public Personne(int id, String nom) {
    this.id = id;
    this.nom = nom;
  }

  public void afficher() {
    System.out.println(id + " " + nom);
  }
}
```

Spring

Créons un fichier `applicationContext` **dans** `src/main/java` **avec le contenu suivant**

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

  <bean id="per" class="org.eclipse.model.Personne">
    <constructor-arg value="1" type="int"></constructor-arg>
    <constructor-arg value="wick"></constructor-arg>
  </bean>
</beans>
```

applicationContext.xml :un fichier de configuration XML dans Spring Framework qui sert à configurer les beans de l'application, définir les dépendances, et configurer les services fournis par Spring, comme la gestion des transactions, la sécurité, la planification des tâches, etc

L'élément **<bean>** dans Spring est utilisé pour définir et configurer un objet dans le conteneur Spring. Dans votre exemple, le bean **per** représente une instance de la classe Personne (située dans le package org.eclipse.model), qui sera créée par le conteneur Spring avec des arguments passés au constructeur.

<bean id="per" class="org.eclipse.model.Personne"> :

id="per" : L'attribut id sert à identifier le bean. Ce nom unique permet de référencer ce bean dans d'autres parties de l'application Spring. class="org.eclipse.model.Personne" : L'attribut class indique la classe que Spring doit instancier pour ce bean. Ici, Spring va créer une instance de la classe Personne du package org.eclipse.model.

<constructor-arg> :sont utilisés pour passer des arguments au constructeur de la classe Personne. Spring injectera ces valeurs lors de la création de l'instance du bean per.

Premier <constructor-arg> :

value="1" : Cela signifie que le premier argument passé au constructeur de Personne sera la valeur 1. type="int" : Indique que l'argument est de type int. Cela permet de spécifier explicitement le type si le constructeur a plusieurs surcharges.

Deuxième <constructor-arg> :

value="wick" : Le deuxième argument passé au constructeur de Personne sera la chaîne "wick". Ici, le type n'est pas spécifié. Spring peut le déduire en fonction de l'ordre des paramètres et du type attendu dans le constructeur.

Spring

Créons un fichier `applicationContext` **dans** `src/main/java` **avec le contenu suivant**

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

  <bean id="per" class="org.eclipse.model.Personne">
    <constructor-arg value="1" type="int"></constructor-arg>
    <constructor-arg value="wick"></constructor-arg>
  </bean>

</beans>
```

Par défaut, le type d'attribut est `String` (le `nom` par exemple)

Spring

Pour récupérer le bean `per`

```
package org.eclipse.main;

import org.eclipse.model.Personne;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App {

  public static void main(String[] args) {
    ApplicationContext context = new
    ClassPathXmlApplicationContext("applicationContext.xml");


    Personne p = context.getBean("per", Personne.class);
    p.afficher();
  }
}
```

Spring

Pour récupérer le bean `per`

```
package org.eclipse.main;

import org.eclipse.model.Personne;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App {

  public static void main(String[] args) {
    ApplicationContext context = new
    ClassPathXmlApplicationContext("applicationContext.xml");
    Personne p = context.getBean("per", Personne.class);
    p.afficher();
  }
}
```

Pas d'opérateur `new` ici.

Spring

Pour récupérer le bean `per`

```
package org.eclipse.main;

import org.eclipse.model.Personne;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App {

  public static void main(String[] args) {
    ApplicationContext context = new
    ClassPathXmlApplicationContext("applicationContext.xml");
    Personne p = context.getBean("per", Personne.class);
    p.afficher();
  }
}
```

Pas d'opérateur `new` ici.

Et si la classe `Personne` dépendait d'une autre classe ou d'une collection?

Spring

Commençons par créer une classe `Adresse`

```
package org.eclipse.model;

public class Adresse {
  private String rue;
  private String codeP;
  private String ville;

  public Adresse(String rue, String codeP, String ville) {
    this.rue = rue;
    this.codeP = codeP;
    this.ville = ville;
  }

  public String toString() {
    return "Adresse [rue=" + rue + ", codeP=" + codeP + ",
       ville=" + ville + "]";
  }
}
```

Spring

Modifions la classe `Personne`

```
public class Personne {
  private int id;
  private String nom;
  private Adresse adresse;

  public Personne(int id, String nom, Adresse adresse) {
    this.id = id;
    this.nom = nom;
    this.adresse = adresse;
  }

  public void afficher() {
    System.out.println(id + " " + nom + " " + adresse);
  }
}
```

Le fichier `applicationContext.xml`

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

  <bean id="adresse" class="org.eclipse.model.Adresse">
    <constructor-arg value="paradis"></constructor-arg>
    <constructor-arg value="13015"></constructor-arg>
    <constructor-arg value="Marseille"></constructor-arg>
  </bean>

  <bean id="per" class="org.eclipse.model.Personne">
    <constructor-arg value="1" type="int"></constructor-arg>
    <constructor-arg value="wick"></constructor-arg>
    <constructor-arg>
          <ref bean="adresse"/>
    </constructor-arg>
  </bean>
</beans>
```

Ce fichier de configuration Spring initialise un objet Personne en utilisant l'injection de dépendances. Le bean adresse est créé d'abord, puis référencé dans le bean per (Personne) grâce à <ref bean="adresse"/>. Cela permet à Spring de gérer la création et l'injection des dépendances automatiquement.

Spring

On ne modifie pas le main

```
package org.eclipse.main;

import org.eclipse.model.Personne;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App  {

  public static void main(String[] args) {
    ApplicationContext context = new
    ClassPathXmlApplicationContext("applicationContext.xml");
    Personne p = context.getBean("per", Personne.class);
    p.afficher();
  }
}
```

Spring

On ne modifie pas le main

```
package org.eclipse.main;

import org.eclipse.model.Personne;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App  {

  public static void main(String[] args) {
    ApplicationContext context = new
    ClassPathXmlApplicationContext("applicationContext.xml"); Personne p =
    context.getBean("per", Personne.class);
    p.afficher();
  }
}
```

Toujours sans l'opérateur `new`.

Spring

Et pour les collections ?

```
package org.eclipse.model;

import java.util.List;

public class Personne {
  private int id;
  private String nom;
  private List<String> sports;

  public Personne(int id, String nom, List<String> sports) {
    this.id = id;
    this.nom = nom;
    this.sports = sports;
  }

  public void afficher(){
    System.out.println(id + " " + nom);
    System.out.println("Mes sports : ");
    sports.forEach(System.out::println);
  }
}
```

Spring

Le fichier `applicationContext.xml`

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

  <bean id="per" class="org.eclipse.model.Personne">
    <constructor-arg value="1" type="int"></constructor-arg>
    <constructor-arg value="wick"></constructor-arg>
    <constructor-arg>
      <list>
        <value>foot</value>
        <value>hand</value>
        <value>basket</value>
      </list>
    </constructor-arg>
  </bean>
</beans>
```

Spring

On ne modifie toujours rien dans `main`

```
package org.eclipse.main;

import org.eclipse.model.Personne;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App {

  public static void main(String[] args) {
    ApplicationContext context = new
    ClassPathXmlApplicationContext("applicationContext.xml"); Personne p =
    context.getBean("per", Personne.class);
    p.afficher();
  }
}
```

Spring

On ne modifie toujours rien dans `main`

```
package org.eclipse.main;

import org.eclipse.model.Personne;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App {

  public static void main(String[] args) {
    ApplicationContext context = new
    ClassPathXmlApplicationContext("applicationContext.xml"); Personne p =
    context.getBean("per", Personne.class);
    p.afficher();
  }
}
```

Toujours sans l'opérateur `new`.

Spring

Remarque

- Dans les exemples précédents, on a fait une injection de dépendance en utilisant le constructeur.
- Mais, on peut aussi faire une injection de dépendance en utilisant le setter.

Spring

Supprimons les différents constructeurs de l'exemple précédent et ajoutons les getters/setters

```
public class Personne { private
  int id;  private String nom;

  public int getId() { return
    id;
  }
  public void setId(int id) { this.id
    = id;
  }
  public String getNom() { return
    nom;
  }
  public void setNom(String nom) { this.nom
    = nom;
  }
  public void afficher() {
    System.out.println(id + " " + nom);
  }
}
```

Spring

Le fichier `applicationContext.xml`

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

   <bean id="per" class="org.eclipse.model.Personne">
     <property name="id">
       <value>1</value>
     </property>
     <property name="nom">
       <value>Wick</value>
     </property>
   </bean>

</beans>
```

Spring

On ne modifie toujours rien dans `main`

```
package org.eclipse.main;

import org.eclipse.model.Personne;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App {

  public static void main(String[] args) {
    ApplicationContext context = new
    ClassPathXmlApplicationContext("applicationContext.xml"); Personne p =
    context.getBean("per", Personne.class);
    p.afficher();
  }
}
```

Spring

On ne modifie toujours rien dans `main`

```
package org.eclipse.main;

import org.eclipse.model.Personne;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App {

  public static void main(String[] args) {
    ApplicationContext context = new
    ClassPathXmlApplicationContext("applicationContext.xml");
     Personne p = context.getBean("per", Personne.class);
    p.afficher();
  }
}
```

Toujours sans l'opérateur `new`.

Spring

Et si on a un objet de type `Adresse` **dans** `Personne`

```
public class Adresse {

  private String rue;
  private String codeP;
  private String ville;

  public Adresse(String rue, String codeP, String ville) {
    this.rue = rue;
    this.codeP = codeP;
    this.ville = ville;
  }

  public String toString() {
    return "Adresse [rue=" + rue + ", codeP=" + codeP + ", ville=" +
       ville + "]";
  }

}
```

Modifions la classe `Personne`

```
public class Personne {
  private int id;
  private String nom;
  private Adresse adresse;

  public int getId() {
    return id;
  }
  public void setId(int id) {
    this.id = id;
  }

  public String getNom() {
    return nom;
  }
  public void setNom(String nom) {
    this.nom = nom;
  }

  public Adresse getAdresse() {
    return adresse;
  }
  public void setAdresse(Adresse adresse) {
    this.adresse = adresse;
  }

  public void afficher(){
    System.out.println(id + " " + nom + " " + adresse);
  }
}
```

Le fichier `applicationContext.xml`

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

  <bean id="adr" class="org.eclipse.model.Adresse">
    <constructor-arg value="paradis"></constructor-arg>
    <constructor-arg value="13015"></constructor-arg>
    <constructor-arg value="Marseille"></constructor-arg>
  </bean>
  <bean id="per" class="org.eclipse.model.Personne">
    <property name="id" value="1"></property>
    <property name="nom">
      <value>Wick</value>
    </property>
    <property name="adresse" ref="adr">
    </property>
  </bean>
</beans>
```

Spring

On ne modifie toujours rien dans `main`

```
package org.eclipse.main;

import org.eclipse.model.Personne;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App {

  public static void main(String[] args) {
    ApplicationContext context = new
    ClassPathXmlApplicationContext("applicationContext.xml"); Personne p =
    context.getBean("per", Personne.class);
    p.afficher();
  }
}
```

Spring

On ne modifie toujours rien dans `main`

```
package org.eclipse.main;

import org.eclipse.model.Personne;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App {

  public static void main(String[] args) {
    ApplicationContext context = new
    ClassPathXmlApplicationContext("applicationContext.xml"); Personne p =
    context.getBean("per", Personne.class);
    p.afficher();
  }
}
```

Toujours sans l'opérateur `new`.

Spring

On peut aussi utiliser l'usine de bean dans le main

```
package org.eclipse.main;

import org.eclipse.model.Personne;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App {

  public static void main(String[] args) {
    ApplicationContext context = new
    ClassPathXmlApplicationContext("applicationContext.xml");/*crée une instance de
    ApplicationContext en chargeant la configuration Spring définie dans le fichier applicationContext.xml,
    */
     BeanFactory factory = (BeanFactory) context;
    Personne p = (Personne)factory.getBean("per");
    p.afficher();
  }
}
```

BeanFatory est une interface fondamentale dans le framework Spring. Elle fait partie du **Spring IoC (Inversion of Control) Container** et est responsable de la gestion des objets (ou beans) dans une application Spring. En d'autres termes, BeanFactory est un conteneur léger utilisé pour instancier, configurer et gérer les objets de l'application de manière centralisée

Spring

On peut aussi utiliser l'usine de bean dans le main

```
package org.eclipse.main;

import org.eclipse.model.Personne;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App {

  public static void main(String[] args) {
    ApplicationContext context = new
    ClassPathXmlApplicationContext("applicationContext.xml"); BeanFactory factory =
    (BeanFactory) context;
    Personne p = (Personne)factory.getBean("per");
    p.afficher();
  }
}
```

Toujours sans l'opérateur `new`.

Spring

Remarque

Pour faire l'injection de dépendance, on peut également utiliser des annotations telles que

- `@Autowired`
- `@Component`
- `@Service`
- `@Repository`
- ...

Spring

Dans le package `org.eclipse.nation`**, créons une interface** `European`

```
public interface European {
        public void saluer();
}
```

Spring

Dans le même package, créons une classe `French` qui implémente `European`

```
public class French implements European {
        public void saluer() {
                System.out.println("Bonjour");
        }
}
```

Spring

Dans le même package, créons une classe `French` qui implémente `European`

```
public class French implements European {
        public void saluer() {
                System.out.println("Bonjour");
        }
}
```

Et une classe `English` qui implémente aussi `European`

```
public class English implements European {
        public void saluer() {
                System.out.println("Hello");
        }
}
```

Spring

Préparons le `main`

```
public class App  {
  public static void main(String[] args) {
    ApplicationContext context = new
    ClassPathXmlApplicationContext("applicationContext.xml"); European e = (European)
    context.getBean("european");
    e.saluer();
  }
}
```

Spring

Préparons le `main`

```
public class App  {
  public static void main(String[] args) {
    ApplicationContext context = new
    ClassPathXmlApplicationContext("applicationContext.xml"); European e = (European)
    context.getBean("european");
    e.saluer();
  }
}
```

Et le fichier de configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.0.xsd">

  <context:component-scan base-package="org.eclipse.nation" ></context:component-scan>

 </beans>
```

Spring

Remarques

- La balise <`context:component-scan base-package="org.eclipse.nation" >` </`context:component-scan`> permet d'indiquer l'emplacement de beans
- Pour déclarer un bean, il faut annoter la classe par `@Component`
- `European` est une interface, donc on ne peut instancier.

Spring

Ajoutons l'annotation `@Component` **à la classe** `French`

```
@Component
public class French implements European {
        public void saluer() {
                System.out.println("Bonjour");
        }
}
```

Et `English`

```
@Component
public class English implements European {
        public void saluer() {
                System.out.println("Hello");
        }
}
```

Spring

Modifions le `main`

```
public class App {
  public static void main(String[] args) {
    ApplicationContext context = new
    ClassPathXmlApplicationContext("applicationContext.xml");
    European e = (European) context.getBean("french");
    e.saluer();
  }
}
```

Spring

Remarques

- En exécutant, un `Bonjour` s'affiche. Si on remplace `french` par `english` dans `getBean()`, un `Hello` sera affiché.
- Le **CamelCase** est important pour la reconnaissance des beans.
- Toutefois, nous pouvons aussi attribuer des noms à nos composants pour pouvoir les utiliser plus tard.

Spring

Exemple

```
@Component("eng")
public class English implements European{
        public void saluer() {
                System.out.println("Hello");
        }
}
```

Spring

Exemple

```
@Component("eng")
public class English implements European{
        public void saluer() {
                System.out.println("Hello");
        }
}
```

Modifions le `main`

```
public class App  {
  public static void main(String[] args) {
    ApplicationContext context = new
    ClassPathXmlApplicationContext("applicationContext.xml"); European e = (European)
    context.getBean("eng");
    e.saluer();
  }
}
```

Spring

Exemple

```
@Component("eng")
public class English implements European{
        public void saluer() {
                System.out.println("Hello");
        }
}
```

Modifions le `main`

```
public class App  {
  public static void main(String[] args) {
    ApplicationContext context = new
    ClassPathXmlApplicationContext("applicationContext.xml");
    European e = (European) context.getBean("eng");
    e.saluer();
  }
}
```

On ne peut plus appeler le bean par le nom `english`

Exemple 2 : considérons la classe `Car` suivante

```
@Component
public class Car {
  public void start() {
    System.out.println("Voiture démarrée et prête à rouler");
  }
}
```

Et la classe `Person` qui dépend de `Car`

```
@Component
public class Person {

  Car c;

  public void drive() {
    System.out.println("Je suis prêt à conduire");
    c.start();
  }
}
```

Spring

Contenu de la méthode `main`

```
public class App {
  public static void main(String[] args) {
    ApplicationContext context = new
    ClassPathXmlApplicationContext("applicationContext.xml"); Person p = (Person)
    context.getBean("person");
    p.drive();
  }
}
```

Spring

Contenu de la méthode `main`

```
public class App {
  public static void main(String[] args) {
    ApplicationContext context = new
    ClassPathXmlApplicationContext("applicationContext.xml"); Person p = (Person)
    context.getBean("person");
    p.drive();
  }
}
```

En exécutant ce code, une erreur sera affichée car on ne peut appeler la méthode `start()` si la classe `car` n'a pas été instanciée.

Spring

Solution : utiliser une annotation pour créer l'objet `c` après instanciation de la classe `Person`

```
@Component
public class Person {
  @Autowired
  Car c;
  public void drive() {
    System.out.println("Je suis prˆet `aconduire");
    c.start();
  }
}
```

Spring

Remarque

- `@Autowired` cherche les beans selon le type.
- Pour chercher un bean selon le nom, il faut utiliser l'annotation `@Qualifier`

Spring

```
@Component("c1")
public class Car {
  public void start() {
    System.out.println("Voiture d´emarr´ee et pr^ete `a rouler");
  }
}
```

```
@Component
public class Person {

  @Autowired
  @Qualifier("c1")
  Car c;

  public void drive() {
    System.out.println("Je suis pr^et `aconduire");
    c.start();
  }
}
```

Spring

Dans un package `org.eclipse.language`**, créons une interface** `Salutation`

```
package org.eclipse.language;

public interface Salutation {
        void sayHello();
}
```

Dans le même package, créons une classe `SalutationFr` qui implémente `Salutation`

```
@Component
public class SalutationFr implements Salutation {

    public void sayHello() {
        System.out.println("En franc¸ais, on dit bonjour.");
    }

}
```

Dans le même package, créons une classe `SalutationFr` qui implémente `Salutation`

```
@Component
public class SalutationFr implements Salutation {

    public void sayHello() {
        System.out.println("En franc¸ais, on dit bonjour.");
    }

}
```

Et une classe `Francais` qui injecte l'interface `Salutation`

```
@Component
public class Francais {
    @Autowired
    private Salutation salutation;

    public void direBonjour() {
        salutation.sayHello();
    }
}
```

Spring

Scannons notre nouveau package et préparons le `main`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.0.xsd">

  <context:component-scan base-package="org.eclipse.nation, org.eclipse.language" >
  </context:component-scan>

 </beans>
```

Spring

Scannons notre nouveau package et préparons le `main`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.0.xsd">

  <context:component-scan base-package="org.eclipse.nation, org.eclipse.language" >
  </context:component-scan>

 </beans>
```

```
public class App  {
  public static void main(String[] args) {
    ApplicationContext context = new
    ClassPathXmlApplicationContext("applicationContext.xml"); Francais f = (Francais)
    context.getBean("francais");
    f.direBonjour();
    // affiche En franc¸ais, on dit bonjour.
  }
}
```

Spring

Et si on définit dans le même package une nouvelle classe `SalutationEn` **qui implémente aussi** `Salutation`

```
@Component
public class SalutationEn implements Salutation {

        public void sayHello() {
                System.out.println("In english we say hello");
        }

}
```

Spring

Et si on définit dans le même package une nouvelle classe `SalutationEn` **qui implémente aussi** `Salutation`

```
@Component
public class SalutationEn implements Salutation {

        public void sayHello() {
                System.out.println("In english we say hello");
        }

}
```

Le `main` **précédent génère une erreur**

```
public class App  {
  public static void main(String[] args) {
    ApplicationContext context = new
    ClassPathXmlApplicationContext("applicationContext.xml"); Francais f = (Francais)
    context.getBean("francais");
    f.direBonjour();
    // erreur
  }
}
```

Spring

Pour charger la version anglaise, on utilise l'annotation `@Qualifier`

```
@Component
public class Francais {
    @Autowired
    @Qualifier("salutationEn")
    private Salutation salutation;

    public void direBonjour() {
        salutation.sayHello();
    }
}
```

Spring

Pour charger la version anglaise, on utilise l'annotation `@Qualifier`

```
@Component
public class Francais {
    @Autowired
    @Qualifier("salutationEn")
    private Salutation salutation;

    public void direBonjour() {
        salutation.sayHello();
    }
}
```

Le `main` **précédent affiche**

```
public class App  {
  public static void main(String[] args) {
    ApplicationContext context = new
    ClassPathXmlApplicationContext("applicationContext.xml"); Francais f = (Francais)
    context.getBean("francais");
    f.direBonjour();
    // In english we say hello
  }
}
```

Spring

Remarque

- Il existe plusieurs autres variations de `@Component` qui s'utilise pour une sémantique particulière comme :
  - `@Controller` : pour désigner le contrôleur du modèle **MVC** (couche présentation)
  - `@Service` : pour désigner un élément de la couche métier
  - `@Repository` : pour désigner un élément **DAO** (couche persistence de données)
  - ...
- `@Service`, `@Controller`, `@Repository` = { `@Component` + quelques autres fonctionnalités }

Spring

Un aperçu du code-source de quelques annotations Spring

```
@Component
public @interface Service {
    ...
}

@Component
public @interface Repository {
    ...
}

@Component
public @interface Controller {
    ...
}
```

Spring

Pour Java Config

- ignorons le fichier `applicationContext.xml` (pas besoin de le supprimer)
- créons une classe `ApplicationConfig` dans `org.eclipse.configuration`

Spring

Considérons la classe `Personne` suivante

```
package org.eclipse.model;

public class Personne {
        private int id;
        private String nom;

        public Personne(int id, String nom) {
                this.id = id;
                this.nom = nom;
        }

        public void afficher() {
                System.out.println(id + " " + nom);
        }
}
```

Spring

Commençons par créer la classe `ApplicationConfig`

```
package org.eclipse.configuration;

public class ApplicationConfig {

}
```

Spring

Pour définir `ApplicationConfig` comme classe de configuration, on ajoute l'annotation `@Configuration`

```
package org.eclipse.configuration;

import org.springframework.context.annotation.Configuration;

@Configuration
public class ApplicationConfig {

}
```

Spring

Définissons notre premier bean dans `ApplicationConfig` en utilisant l'annotation `@Bean`

```
package org.eclipse.configuration;

import org.eclipse.model.Personne;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class ApplicationConfig {

        @Bean
        public Personne per() {
                Personne personne = new Personne(1, "wick");
                return personne;
        }
}
```

Spring

Pour utiliser cet objet

```
package org.eclipse.main;

import org.eclipse.configuration.ApplicationConfig;
import org.eclipse.model.Personne;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class App {
  public static void main( String[] args ) {
    ApplicationContext context =
      new AnnotationConfigApplicationContext(ApplicationConfig.class);
    Personne p = context.getBean("per", Personne.class);
    p.afficher();
  }
}
```

Spring

Pour utiliser cet objet

```
package org.eclipse.main;

import org.eclipse.configuration.ApplicationConfig;
import org.eclipse.model.Personne;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class App {
  public static void main( String[] args ) {
    ApplicationContext context =
      new AnnotationConfigApplicationContext(ApplicationConfig.class);
    Personne p = context.getBean("per", Personne.class);
    p.afficher();
  }
}
```

Et si la classe `Personne` dépendait d'une autre classe ?

Spring

Considéons `Adresse`

```
package org.eclipse.model;

public class Adresse {
  private String rue;
  private String codeP;
  private String ville;

  public Adresse(String rue, String codeP, String ville) {
    this.rue = rue;
    this.codeP = codeP;
    this.ville = ville;
  }

  public String toString() {
    return "Adresse [rue=" + rue + ", codeP=" + codeP + ",
       ville=" + ville + "]";
  }
}
```

Spring

Ajoutons un attribut de type `Adresse` dans la classe `Personne`

```
public class Personne {
  private int id;
  private String nom;
  private Adresse adresse;

  public Personne(int id, String nom, Adresse adresse) {
    this.id = id;
    this.nom = nom;
    this.adresse = adresse;
  }

  public void afficher() {
    System.out.println(id + " " + nom + " " + adresse);
  }
}
```

Définissons un bean `Adresse` et utilisons le dans le bean `Personne`

```
package org.eclipse.configuration;

import org.eclipse.model.Adresse;
import org.eclipse.model.Personne;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class ApplicationConfig {

    @Bean
    public Adresse adr() {
        Adresse adresse = new Adresse("paradis", "13000", "Marseille");
        return adresse;
    }

    @Bean
    public Personne per(Adresse adr) {
        Personne personne = new Personne(1, "wick", adr);
        return personne;
    }
}
```

Spring

On ne modifie pas le main

```
package org.eclipse.main;

import org.eclipse.configuration.ApplicationConfig;
import org.eclipse.model.Personne;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.
  AnnotationConfigApplicationContext;

public class App
{
    public static void main( String[] args )
    {
        ApplicationContext context = new
          AnnotationConfigApplicationContext(ApplicationConfig.class);
        Personne p = context.getBean("per", Personne.class);
        p.afficher();
        // affiche 1 wick Adresse [rue=paradis, codeP=13000, ville=
          Marseille]
    }
}
```

Spring

Considérons l'interface `European` **et les deux classes précédentes** `French` **et** `English` **qui l'implémentent**

```
public interface European {
        public void saluer();
}
```

```
@Component
public class French implements European{
        public void saluer() {
                System.out.println("Bonjour");
        }
}
```

```
@Component
public class English implements European{
        public void saluer() {
                System.out.println("Hello");
        }
```

Indiquons ànotre classe `ApplicationConfig` l'emplacement de nos composants

```
package org.eclipse.configuration;

import org.eclipse.model.Adresse;
import org.eclipse.model.Personne;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan("org.eclipse.nation")
public class ApplicationConfig {
    @Bean
    public Adresse adr() {
        Adresse adresse = new Adresse("paradis", "13000", "Marseille");
        return adresse;
    }

    @Bean
    public Personne per(Adresse adr) {
        Personne personne = new Personne(1, "wick", adr);
        return personne;
    }
}
```

Spring

Pour tester

```
package org.eclipse.main;

import org.eclipse.configuration.ApplicationConfig;
import org.eclipse.nation.European;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.
  AnnotationConfigApplicationContext;

public class App
{
    public static void main( String[] args )
    {
        ApplicationContext context = new
          AnnotationConfigApplicationContext(ApplicationConfig.class);
        European e = (European) context.getBean("english");
        e.saluer();
        // affiche hello
    }
}
```

Spring

Ajoutons le constructeur suivant dans la classe `French`

```
package org.eclipse.nation;

import org.springframework.stereotype.Component;

@Component
public class French implements European {

        public French() {
                System.out.println("Constructeur");
        }

        public void saluer() {
                System.out.println("Bonjour");
        }

}
```

Spring

Dans le `main`, demandons deux beans `french` et appelons la méthode `saluer`

```
public class App
{
    public static void main( String[] args )
    {
        ApplicationContext context = new
           AnnotationConfigApplicationContext(ApplicationConfig.class);
        European e1 = (European) context.getBean("french");
        e1.saluer();
        European e2 = (European) context.getBean("french");
        e2.saluer();
    }
}
```

Spring

Dans le `main`, demandons deux beans `french` et appelons la méthode `saluer`

```
public class App
{
    public static void main( String[] args )
    {
        ApplicationContext context = new
           AnnotationConfigApplicationContext(ApplicationConfig.class);
        European e1 = (European) context.getBean("french");
        e1.saluer();
        European e2 = (European) context.getBean("french");
        e2.saluer();
    }
}
```

Le résultat

```
Constructeur
Bonjour
Bonjour
```

Spring

Et si on ajoute dans le `main` le code suivant

```
System.out.println("e1 == e2 : " + (e1 == e2));
System.out.println("@e1 : " + e1);
System.out.println("@e2 : " + e2);
```

Spring

Et si on ajoute dans le `main` le code suivant

```
System.out.println("e1 == e2 : " + (e1 == e2));
System.out.println("@e1 : " + e1);
System.out.println("@e2 : " + e2);
```

Le résultat sera

```
e1 == e2 : true
@e1 : org.eclipse.nation.French@47eaca72
@e2 : org.eclipse.nation.French@47eaca72
```

Spring

Explication : cinq portées (scope) possibles pour les beans Spring

- `singleton` (par défaut)
- `prototype`
- `request`
- `session`
- `application`

Spring

Ajoutons l'annotation `@Scope` et précisons la portée `prototype` dans `French`

```
package org.eclipse.nation;

import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Component
@Scope("prototype")
public class French implements European {

        public French() {
                System.out.println("Constructeur");
        }

        public void saluer() {
                System.out.println("Bonjour");
        }

}
```

Spring

Testons le `main` précédent

```
public class App
{
    public static void main( String[] args )
    {
        ApplicationContext context = new
           AnnotationConfigApplicationContext(ApplicationConfig.class);
        European e1 = (European) context.getBean("french");
        e1.saluer();
        European e2 = (European) context.getBean("french");
        e2.saluer();
    }
}
```

Spring

Testons le `main` précédent

```
public class App
{
    public static void main( String[] args )
    {
        ApplicationContext context = new
           AnnotationConfigApplicationContext(ApplicationConfig.class);
        European e1 = (European) context.getBean("french");
        e1.saluer();
        European e2 = (European) context.getBean("french");
        e2.saluer();
    }
}
```

Le résultat

```
Constructeur
Bonjour
Constructeur
Bonjour
```

Spring

Problématique

- Si nous disposons d'un bean de portée `singleton` retournant un bean de portée `prototype`.
- Chaque fois qu'on demande un bean `prototype`, le bean de portée `singleton` nous retournera toujours la même instance.

Dans un package `org.eclipse.scope`**, créons la classe** `PrototypeBean` **avec la portée** `prototype`

```
@Component
@Scope("prototype")
public class PrototypeBean {
        public PrototypeBean() {
                System.out.println("Prototype instance created");
        }
}
```

Dans un package `org.eclipse.scope`**, créons la classe** `PrototypeBean` **avec la portée** `prototype`

```
@Component
@Scope("prototype")
public class PrototypeBean {
        public PrototypeBean() {
                System.out.println("Prototype instance created");
        }
}
```

et la classe `SingletonBean` **avec la portée** `singleton`

```
@Component
public class SingletonBean {
    @Autowired
    private PrototypeBean prototypeBean;

    public SingletonBean() {
        System.out.println("Singleton instance created");
    }
    public PrototypeBean getPrototypeBean() {
        return prototypeBean;
    }
}
```

Spring

Le `main` pour tester (N'oublions pas de scanner le package `org.eclipse.scope`)

```
public class App
{
    public static void main( String[] args )
    {
        ApplicationContext context = new
           AnnotationConfigApplicationContext(ApplicationConfig.class);
        SingletonBean instance1 = context.getBean(SingletonBean.class);
        PrototypeBean prototype1 = instance1.getPrototypeBean();
        PrototypeBean prototype2 = instance1.getPrototypeBean();
    }
}
```

Spring

Le `main` pour tester (N'oublions pas de scanner le package `org.eclipse.scope`)

```
public class App
{
    public static void main( String[] args )
    {
        ApplicationContext context = new
           AnnotationConfigApplicationContext(ApplicationConfig.class);
        SingletonBean instance1 = context.getBean(SingletonBean.class);
        PrototypeBean prototype1 = instance1.getPrototypeBean();
        PrototypeBean prototype2 = instance1.getPrototypeBean();
    }
}
```

Le résultat

```
Singleton instance created
Prototype instance created
```

Spring

Pour résoudre le problème précédent, on utilise l'annotation `@Lookup`

```
@Component
public class SingletonBean {

    @Autowired
    private PrototypeBean prototypeBean;

    public SingletonBean() {
        System.out.println("Singleton instance created");
    }

    @Lookup
    public PrototypeBean getPrototypeBean() {
        return prototypeBean;
    }
}
```

Spring

Testons le `main` précédent

```
public class App
{
    public static void main( String[] args )
    {
        ApplicationContext context = new
           AnnotationConfigApplicationContext(ApplicationConfig.class);
        SingletonBean instance1 = context.getBean(SingletonBean.class);
        PrototypeBean prototype1 = instance1.getPrototypeBean();
        PrototypeBean prototype2 = instance1.getPrototypeBean();
    }
}
```

Spring

Testons le `main` précédent

```
public class App
{
    public static void main( String[] args )
    {
        ApplicationContext context = new
           AnnotationConfigApplicationContext(ApplicationConfig.class);
        SingletonBean instance1 = context.getBean(SingletonBean.class);
        PrototypeBean prototype1 = instance1.getPrototypeBean();
        PrototypeBean prototype2 = instance1.getPrototypeBean();
    }
}
```

Le résultat

```
Singleton instance created
Prototype instance created
Prototype instance created
Prototype instance created
```

Spring

Problématique

- La classe `SingletonBean` avait une méthode qui retournait une instance différente de `PrototypeBean` à chaque appel.
- Et si nous voulions qu'une méthode de notre classe singleton utilisait une instance différente de notre classe prototype à chaque appel.

Spring

Ajouter un constructeur dans la classe `Car` et indiquons la portée `prototype`

```
@Component
@Scope(value="prototype")
public class Car {
    public Car() {
        System.out.println("Car constructor");
    }
    public void start() {
        System.out.println("Voiture démarrée et prête à rouler");
    }
}
```

Spring

Ajouter un constructeur dans la classe `Car` et indiquons la portée `prototype`

```
@Component
@Scope(value="prototype")
public class Car {
    public Car() {
        System.out.println("Car constructor");
    }
    public void start() {
        System.out.println("Voiture démarrée et prête à rouler");
    }
}
```

et un constructeur dans la classe `Person`

```
@Component
public class Person {
    @Autowired
    Car c;
    public Person() {
        System.out.println("Person constructor");
    }
    public void drive() {
        System.out.println("Je suis prêt à conduire");
        c.start();
    }
}
```

Testons le `main` précédent

```
public class App
{
    public static void main( String[] args )
    {
        ApplicationContext context = new
           AnnotationConfigApplicationContext(ApplicationConfig.class);
        Person p = (Person) context.getBean("person");
        p.drive();
        p.drive();
    }
}
```

Testons le `main` précédent

```
public class App
{
    public static void main( String[] args )
    {
        ApplicationContext context = new
           AnnotationConfigApplicationContext(ApplicationConfig.class);
        Person p = (Person) context.getBean("person");
        p.drive();
        p.drive();
    }
}
```

Le résultat

```
Person constructor
Car constructor
Je suis prêt à conduire
Voiture démarrée et prête à rouler
Je suis prêt à conduire
Voiture démarrée et prête à rouler
```

Spring

Pour résoudre le problème précèdent (une personne conduit un véhicule diffèrent à chaque appel de la méthode `drive`), on modifie l'annotation `@Scope` en ajoutant un procureur qui retournera une nouvelle instance de `Car`

```
@Component
@Scope(value="prototype", proxyMode=ScopedProxyMode.TARGET_CLASS)
public class Car {

    public Car() {
        System.out.println("Car constructor");
    }

    public void start() {
        System.out.println("Voiture démarrée et prête à rouler");
    }

}
```

Testons le `main` précédent

```
public class App
{
    public static void main( String[] args )
    {
        ApplicationContext context = new
           AnnotationConfigApplicationContext(ApplicationConfig.class);
        Person p = (Person) context.getBean("person");
        p.drive();
        p.drive();
    }
}
```

Testons le `main` précédent

```
public class App
{
    public static void main( String[] args )
    {
        ApplicationContext context = new
           AnnotationConfigApplicationContext(ApplicationConfig.class);
        Person p = (Person) context.getBean("person");
        p.drive();
        p.drive();
    }
}
```

Le résultat

```
Person constructor
Je suis prêt à conduire
Car constructor
Voiture démarrée et prête à rouler
Je suis prêt à conduire
Car constructor
Voiture démarrée et prête à rouler
```