Support Formation JAVA Spring Core

Pour	Formation
Date	13/11/2024
Objet	Support Formation JAVA Spring Core

I) Glossaire 3
II) Présentation du Framework Spring 4
III) Quelques rappels sur la réflexivité 5
IV) Design Pattern Injection de Dépendance. 12
V) Injection de Dépendance, Inversion de contrôle, couplage faible, couplage forte. 16
VI) SpEL (Spring Expression Language) 39
1. Introduction 39
2. Operateur 40
a) Les opérateurs arithmétiques 41
b) Opérateurs relationnels et logiques 42
c) Opérateurs logiques 43
d) Utilisation de Regex dans SpEL 44
e) Accéder aux objets d'une Liste et d'un Map 45
3. Utiliser le SpEl dans Spring configuration 46
VII) Spring profiles 47
1. Utiliser @Profile sur un Bean 48
2. Declaration Profiles en XML 49
3. Définir des profils 50
a) Par programmation via l'interface WebApplicationInitializer 50
b) Par programmation via ConfigurableEnvironment 51
c) Context Parameter sur web.xml 52
d) Par paramètres système de la JVM 53
e) Par profil Maven 54
f) @ActiveProfile dans les tests 55
VIII) Spring Bean scopes 56
IX) Les modes d'Autowire avec Spring 57

I) Glossaire

- **API** : Signifie Application Programming Interface. Ce qui veut dire que c'est un ensemble de bibliothèques et librairies dédié pour implémenter une fonctionnalité donnée.

- **ORM**: Object-Relational Mapping (**MOR** : Mapping Objet-Relationnel en français) est une technique de programmation informatique qui crée l'illusion d'une base de données orientée objet à partir d'une base de données relationnelle en définissant des correspondances entre cette base de données et les objets du langage utilisé.

- **JPA** : Java Persistence API (abrégée en JPA), est une interface de programmation Java permettant aux développeurs d'organiser des données relationnelles dans des applications utilisant la plateforme Java.

- **JPQL** : Le langage JPQL (**Java Persistence Query Language**) est un langage de requête orienté objet, similaire à SQL, mais au lieu d'opérer sur les tables et colonnes, JPQL travaille avec des objets persistants et de leurs propriétés. Il est très proche du langage SQL dont il s'inspire fortement mais offre une approche objet. La grammaire de ce langage est définie par la spécification J.P.A.

- **HQL** : **Hibernate Query Language** est aussi un langage de requête orienté objet au même titre que JPQL. La principale différence avec le langage JQL est que le en HQL le « **Select** » sur l'objet n'est pas nécessaire. En fin de compte pour le JPQL on aura : **Select person from Personne person** alors que pour le HQL on aura : **from Personne**.

- **Bean** : le « **Bean** » (ou haricot en français) est une technologie de composants logiciels écrits en langage Java. Les **Beans** sont utilisés pour encapsuler plusieurs objets dans un seul objet. Le « **Bean** » regroupe alors tous les attributs des objets encapsulés. Ainsi, il représente une entité plus globale que les objets encapsulés de manière à répondre à un besoin métier.

- **Pattern IoC** : L'inversion de contrôle (inversion of control, IoC) est un patron d'architecture commun à tous les Frameworks (ou cadre de développement et d'exécution). Il fonctionne selon le principe que le flot d'exécution d'un logiciel n'est plus sous le contrôle direct de l'application elle-même mais du Framework ou de la couche logicielle sous-jacente. En effet selon un problème, il existe différentes formes, ou représentation d'IoC, le plus connu étant l'injection de dépendances (dependency injection) qui est un patron de conception permettant, en programmation orientée objet, de découpler les dépendances entre objets.

- **Pattern AOP** : L'AOP (Aspect Oriented Programming) ou POA (Programmation Orientée Aspect) est un paradigme de programmation ayant pour but de compléter la programmation orientée objet et permettre d'implémenter de façon plus propre les problématiques transverses à l'application. En effet, elle permet de factoriser du code dans des greffons et de les injecter en divers endroits sans pour autant modifier le code source des endroits en question.

II) Présentation du Framework Spring

Le framework Spring a été initialement développé par Rod Johnson et Juergen Holler dans sa version **Spring 1.0** en mars 2004. Spring est un framework libre qui permet de construire et de définir l'infrastructure d'une application java, dont il facilite le développement et les tests. Il fournit de nombreuses fonctionnalités qui peuvent être utilisées de plusieurs manières ce qui laisse le choix au développeur d'utiliser la solution qui répond à ses besoins.Il est l'un des frameworks les plus répandus dans le monde Java, sa popularité a grandie à cause de la complexité des serveurs d'application Java EE.

Le framework a un coeur reposant sur un conteneur de type IoC assure la gestion du cycle de vies des beans et l'injection des dépendances et un autre conteneur dite AOP qui elle permet de factoriser du code dans des greffons et de les injecter en divers endroits sans pour autant modifier le code source des endroits en question.

Spring était un framework applicatif qui permet de faciliter l'intégration avec des projets open source ou API de Java EE. Sa principale interêt est sa grande flexibilité ses fonctionnalités.
Spring est souvent appelé conteneur léger (lightweight container) par opposition aux conteneurs lourds que sont les serveurs d'applications Java EE.

III) Quelques rappels sur la réflexivité

La réflexivité ou encore l'introspection consiste à découvrir de manière dynamique des informations propres à une classe Java ou à un objet. Ce mécanisme est notamment utilisé au niveau de la machine virtuelle Java lors de l'exécution de votre programme.

Le paquetage de l'API qui gère la réflexivité est « java.lang.reflect ».
La réflexivité permet notamment l'introspection et rend possible l'accès aux classes, à leurs champs, méthodes, constructeurs et à toutes les informations les caractérisant, même celles qu'on pensait inaccessibles Elle est également très utile pour instancier des classes de manière dynamique, dans le processus de sérialisation d'un Bean Java. Elle est aussi utilisée dans la génération de code (Exemple ORM tel que Hibernate).

Avec la réflexivité il est possible :

1) D'identifier les classes parents d'une classe donnée

C'est-à-dire connaître l'ensemble des classes dont hérite une classe. Considérons l'exemple précédemment de la classe « Vehicule » cf. « Les Objets Java » nous avions la classe « Voiture » qui hérite de la classe « Vehicule » et une classe « Voiture4X4 » qui hérite de « Voiture ».

Affichons dans le programme ci-dessous tous les parents de la classe « Voiture4X4 » :

```
24
25      public static void displaySuperclass() {
26          String methodName = "displaySuperclass";
27          Class theClass = null;
28          try {
29              theClass = Class.forName("javaapplicationreflect.Voiture4X4");
30          } catch (ClassNotFoundException ex) {
31              System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + ex);
32          }
33          while ((theClass = theClass.getSuperclass()) != null) {
34              System.out.println("Nom de la classe Mère: " + theClass.getSimpleName());
35              System.out.println("Nom Complet de la classe Mère: " + theClass.getName()+"\n");
36          }
37      }
```

On obtient donc à l'affichage :

```
Output - JavaApplicationReflect (run) ×   Search Results   Test Results
run:
Nom de la classe Mère: Voiture
Nom Complet de la classe Mère: javaapplicationreflect.Voiture

Nom de la classe Mère: Vehicule
Nom Complet de la classe Mère: javaapplicationreflect.Vehicule

Nom de la classe Mère: Object
Nom Complet de la classe Mère: java.lang.Object

BUILD SUCCESSFUL (total time: 0 seconds)
```

L'affichage nous montre bien que la classe « Voiture4X4 » hérite des classes « Voiture », « Vehicule » et bien entendu la classe « Object » qui est par définition la mère de toute les classe Java.

2) Connaitre toutes les interfaces implémentées par une classe donnée

Considérons l'exemple d'une interface « IAnimale » avec sa classe d'implémentation « Animal ».
« Animal » :

```java
13  public class Animal implements IAnimal, Serializable {
14
15      public String nom;
16      public double poids;
17      public String couleur;
18
19      @Override
20      public void crie() {
21          System.out.println("L'animal crie");
22      }
23
24      @Override
25      public void marcher() {
26          System.out.println("L'animal marche");
27      }
28
29      public String getNom() {
30          return nom;
31      }
32
33      public void setNom(String nom) {
34          this.nom = nom;
35      }
36
37      public double getPoids() {
38          return poids;
39      }
40
41      public void setPoids(double poids) {
42          this.poids = poids;
43      }
44
45      public String getCouleur() {
46          return couleur;
47      }
48
49      public void setCouleur(String couleur) {
50          this.couleur = couleur;
51      }
52  }
53
```

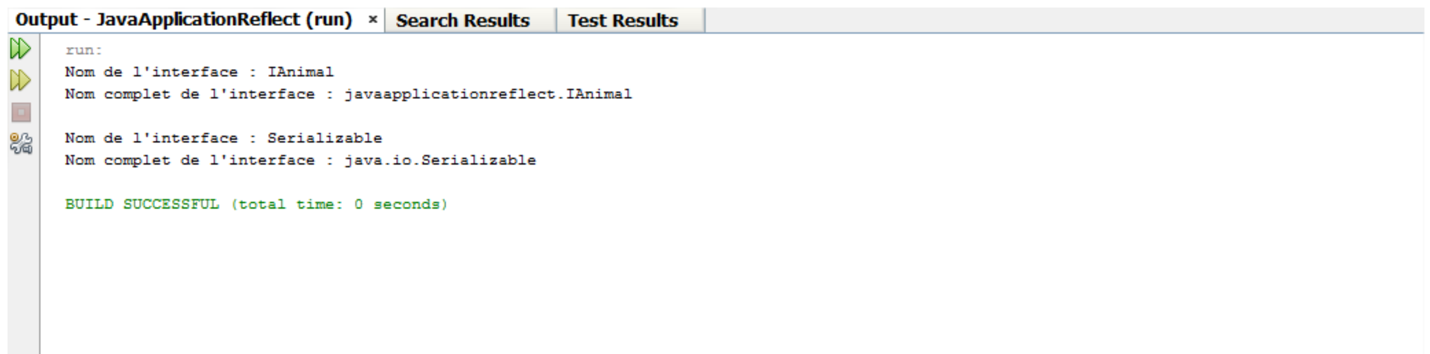
« IAnimale » :

```java
10   */
11  public interface IAnimal {
12
13      void crie();
14
15      void marcher();
16  }
17
18
```

Considérons le programme Java ci-dessous :

```
39
40     public static void displayInterfaces() {
41         String methodName = "displayInterfaces";
42         Class theClass = null;
43         try {
44             theClass = Class.forName("javaapplicationreflect.Animal");
45             Class[] interfaces = theClass.getInterfaces();
46             for (Class theInterface : interfaces) {
47                 System.out.println("Nom de l'interface : " + theInterface.getSimpleName());
48                 System.out.println("Nom complet de l'interface : " + theInterface.getName() + "\n");
49             }
50         } catch (ClassNotFoundException ex) {
51             System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + ex);
52         }
53     }
54
```

On obtient à l'affichage :

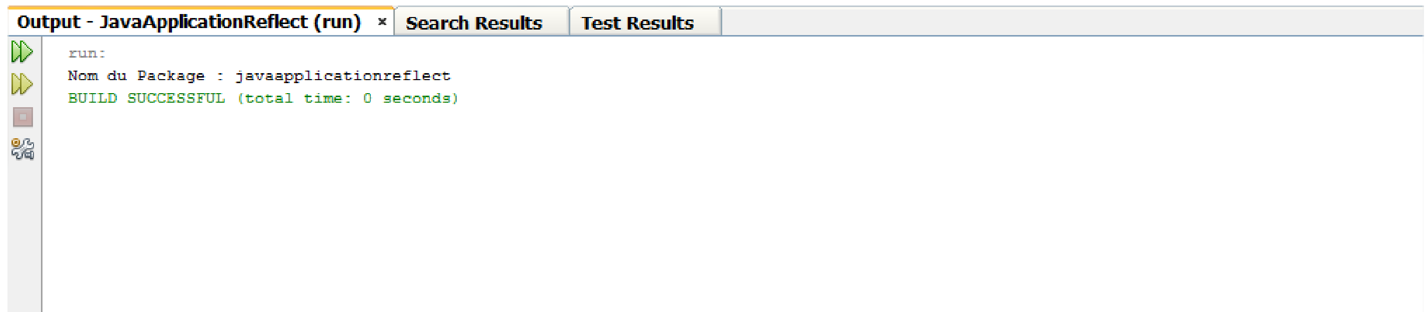


3) Connaitre le nom du paquetage d'une classe donnée

Toujours avec la classe « Animal » nous pouvons récupérer son nom de paquetage.

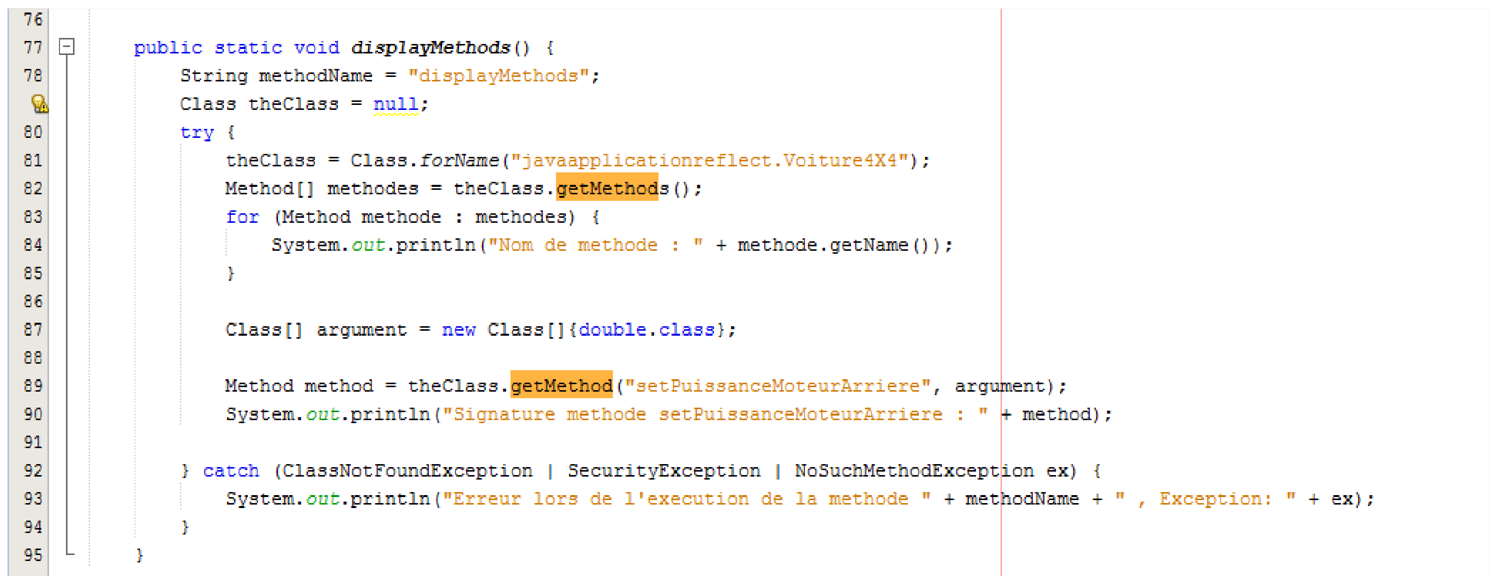
```
61
62     public static void displayPackage() {
63         String methodName = "displayPackage";
64         Class theClass = null;
65         try {
66             theClass = Class.forName("javaapplicationreflect.Animal");
67             Package pack = theClass.getPackage();
68             System.out.println("Nom du Package : " + pack.getName());
69         } catch (ClassNotFoundException ex) {
70             System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + ex);
71         }
72     }
73
```

On obtient à l'affichage :



4) Récupérer la liste des méthodes d'une classe.

Nous pouvons récupérer la liste des méthodes de la classe la classe « Voiture4X4 ».



Ligne 83 : On récupère la liste des méthodes de la classe « Voiture4X4 ».
Ligne 89 : On récupère la signature de la méthode « setPuissanceMoteurArriere » en lui passant comme argument « new Class [] {double.class} » car en effet la méthode « setPuissanceMoteurArriere » a comme signature dans la classe « Voiture4X4 » :

```
public void setPuissanceMoteurArriere(double puissanceMoteurArriere) {
    this.puissanceMoteurArriere = puissanceMoteurArriere;
}
```

5) Récupérer la liste des attributs d'une classe.

Il est aussi possible de récupérer la liste des attributs d'une classe.

```
98
99      public static void displayAllFields() {
100         String methodName = "displayAllFields";
            Class theClass = null;
102         try {
103             theClass = Class.forName("javaapplicationreflect.Vehicule");
104             java.lang.reflect.Field[] fields = theClass.getDeclaredFields();
105             for (Field field : fields) {
106                 System.out.println("Nom du champ : " + field);
107             }
108         } catch (ClassNotFoundException | SecurityException ex) {
109             System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + ex);
110         }
111     }
```

```
Output - JavaApplicationReflect (run)   Search Results   Test Results
run:
Nom du champ : protected java.lang.String javaapplicationreflect.Vehicule.marque
Nom du champ : protected java.lang.String javaapplicationreflect.Vehicule.couleur
Nom du champ : protected int javaapplicationreflect.Vehicule.annee
Nom du champ : protected double javaapplicationreflect.Vehicule.taille
Nom du champ : protected boolean javaapplicationreflect.Vehicule.estAssure
BUILD SUCCESSFUL (total time: 0 seconds)
```

6) Récupérer la liste des constructeurs d'une classe.

On aussi récupérer la liste de constructeurs.

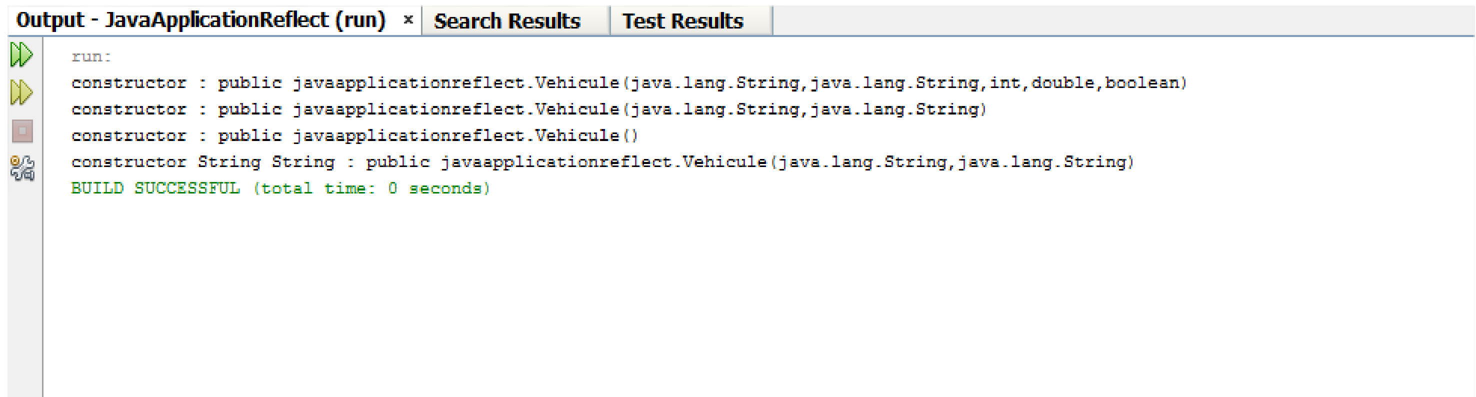
```
131
132     public static void displayConstructors() {
133         String methodName = "displayConstructors";
            Class theClass = null;
135         try {
136             theClass = Class.forName("javaapplicationreflect.Vehicule");
137             Constructor[] constructors = theClass.getConstructors();
138             for (Constructor constructor : constructors) {
139                 System.out.println("constructor : " + constructor);
140             }
141             Class[] arguments = new Class[]{String.class, String.class};
142             Constructor constructor = theClass.getConstructor(arguments);
143             System.out.println("constructor String String : " + constructor);
144         } catch (ClassNotFoundException | SecurityException | NoSuchMethodException ex) {
145             System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + ex);
146         }
147     }
148
```

Ligne 137 : On récupère la liste des constructeurs de la classe « Vehicule ».
Ligne 89 : On récupère la signature de constructeur « Vehicule(String marque, String couleur) » en lui passant comme argument « new Class [] {String.class, String.class } ».

Nous rappelons que le code du constructeur est le suivant :

```
public Vehicule(String marque, String couleur) {
	this.marque = marque;
	this.couleur = couleur;
	System.out.println("Contructeur Vehicule(string,string)");
}
```

On obtient le résultat suivant :



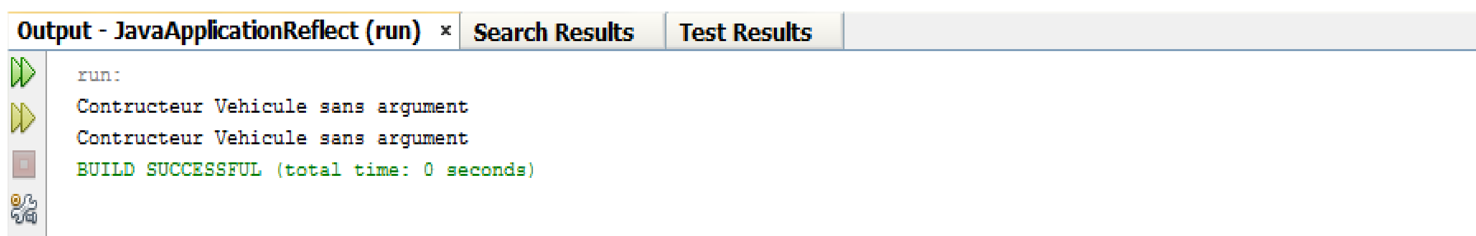
```
Output - JavaApplicationReflect (run) ×   Search Results   Test Results
run:
constructor : public javaapplicationreflect.Vehicule(java.lang.String,java.lang.String,int,double,boolean)
constructor : public javaapplicationreflect.Vehicule(java.lang.String,java.lang.String)
constructor : public javaapplicationreflect.Vehicule()
constructor String String : public javaapplicationreflect.Vehicule(java.lang.String,java.lang.String)
BUILD SUCCESSFUL (total time: 0 seconds)
```

7) Instancier une classe de manière dynamique.

Dans l’exemple ci-dessous nous allons fabriquer une instance de « Vehicule » de manière inhabituelle en utilisant la retro inspection.

```
150
151     public static void instanciateVehicule() {
152         String methodName = "instanciateVehicule";
153         try {
154             Vehicule renault = new Vehicule();
155             // equivalent à
156             Class theClass = Class.forName("javaapplicationreflect.Vehicule");
157             Object objectCitroen = theClass.newInstance();
158         } catch (ClassNotFoundException | SecurityException | IllegalAccessException | InstantiationException ex) {
159             System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + ex);
160         }
161     }
```

Ligne 154 et 157 : Ces deux lignes sont totalement équivalentes. D’ailleurs même la console affiche :



```
Output - JavaApplicationReflect (run) ×   Search Results   Test Results
run:
Contructeur Vehicule sans argument
Contructeur Vehicule sans argument
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ce qui correspont exactement à l’instanciation de deux object de type « Vehicule ».

On peut caster « objectCitroen » en « Vehicule ».

```
150
151     public static void instanciateVehicule() {
152         String methodName = "instanciateVehicule";
153         try {
154             Vehicule citroen = null;
155             Vehicule renault = new Vehicule();
156             // equivalent à
157             Class theClass = Class.forName("javaapplicationreflect.Vehicule");
158             Object objectCitroen = theClass.newInstance();
159             if (objectCitroen instanceof Vehicule) {
160                 citroen = (Vehicule) objectCitroen;
161             }
162         } catch (ClassNotFoundException | SecurityException | IllegalAccessException | InstantiationException ex) {
163             System.out.println("Erreur lors de l'execution de la methode " + methodName + " , Exception: " + ex);
164         }
165     }
```

IV) Design Pattern Injection de Dépendance.

Considérons une entreprise avec des employés dont chacun a une adresse et un rôle précis dans l'entreprise. Les entités mis en jeu dans cet example son **Employe**, **Role** et **Adresse** dont les signatures sont :

```
1    public class Role {
2        private int idRole;
3        private String codeRole;
4        private String descriptionRole;
5        public Role() {}
6    }
7    public class Adresse {
8        private int idAdresse;
9        private String nomRue;
10       private String codePostal;
11       private String ville;
12       private String pays;
13       public Adresse() {}
14   }
15   public class Employe {
16       private int idEmploye;
17       private Role role;
18       private Adresse adresse;
19       public Employe() {
20           this.role = new Role();
21           this.adresse = new Adresse();
22       }
23   }
```

En version copiable:

```
public class Role {
   private int idRole;
   private String codeRole;
   private String descriptionRole;
   public Role() {}
}
public class Adresse {
   private int idAdresse;
   private String nomRue;
   private String codePostal;
   private String ville;
   private String pays;
   public Adresse() {}
}
public class Employe {
   private int idEmploye;
   private Role role;
   private Adresse adresse;
   public Employe() {
      this.role = new Role();
      this.adresse = new Adresse();
   }
}
```

Ce code ci-dessus permet de voir la relation de dépendance entre la classe **Employe** et les classes **Role** et **Adresse**. Mais le code comporte quand même quelques soucis :

- Lorsqu'on instancie un objet de type **Employe** alors les objets **Role et Adresse** sont tout temps les même.
- Lorsqu'on change la signature des classes **Role** ou **Adresse** (constructeur par exemple) alors on doit aussi changer le constructeur de la classe **Employe** aussi sous peine d'avoir des erreurs de compilation.

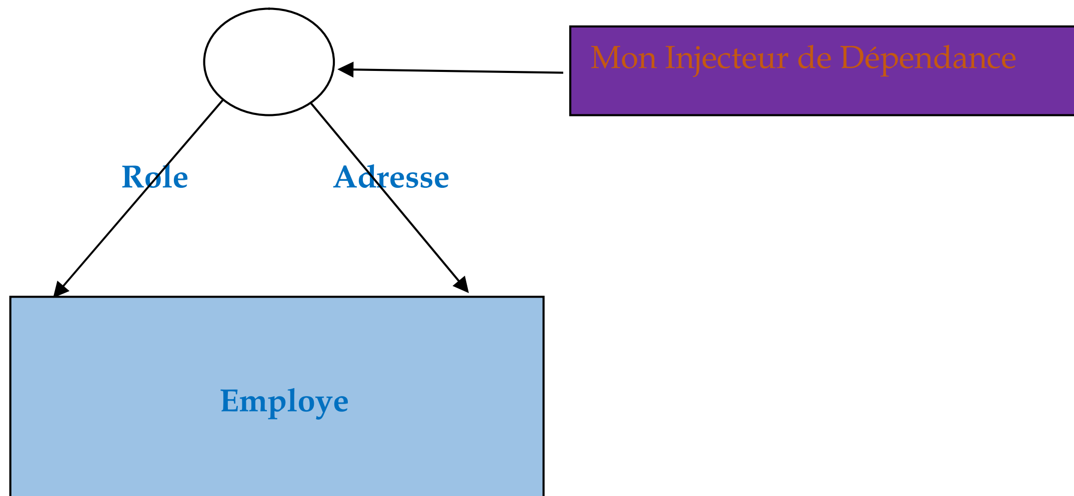
Le premier problème peut être résolu facilement en redéfinissant les constructeurs **Role** et **Adresse**. Ceci va nous obliger à changer les classes **Employe**, **Role** et **Adresse**.

```
1   public class Role {
2       private int idRole;
3       private String codeRole;
4       private String descriptionRole;
5
6       public Role(String codeRole, String descriptionRole) {
7           this.codeRole = codeRole;
8           this.descriptionRole = descriptionRole;
9       }
10  }
11
12  public class Adresse {
13      private int idAdresse;
14      private String nomRue;
15      private String codePostal;
16      private String ville;
17      private String pays;
18
19      public Adresse(String nomRue, String codePostal, String ville, String pays) {
20          this.nomRue = nomRue;
21          this.codePostal = codePostal;
22          this.ville = ville;
23          this.pays = pays;
24      }
25  }
26
27  public class Employe {
28      private int idEmploye;
29      private Role role;
30      private Adresse adresse;
31
32      public Employe(Role role,Adresse adresse) {
33          this.role = role;
34          this.adresse = adresse;
35      }
36  }
37  Role role = new Role("Administrateur","Je suis Admin");
38  Address addresse = new Address("river","5555","New York","Etats Unis");
39  Employe employe = new Employe(role,addresse);
40
```

En version copiable:

```
public class Role {
    private int idRole;
    private String codeRole;
    private String descriptionRole;

    public Role(String codeRole, String descriptionRole) {
        this.codeRole = codeRole;
        this.descriptionRole = descriptionRole;
    }
}

public class Adresse {
    private int idAdresse;
    private String nomRue;
    private String codePostal;
    private String ville;
    private String pays;

    public Adresse(String nomRue, String codePostal, String ville, String pays) {
        this.nomRue = nomRue;
        this.codePostal = codePostal;
        this.ville = ville;
        this.pays = pays;
    }
}

public class Employe {
    private int idEmploye;
    private Role role;
    private Adresse adresse;

    public Employe(Role role,Adresse adresse) {
        this.role = role;
        this.adresse = adresse;
    }
}
Role role = new Role("Administrateur","Je suis Admin");
Address addresse = new Address("river","5555","New York","Etats Unis");
Employe employe = new Employe(role,addresse);
```

Lignes 37 ,38, 39 : Pour avoir une instance de **Employe** il faut aussi avoir une instance de **Role** et **Adresse** ce qui cause aussi problème.
Ce qui serait trop cool c'est de pouvoir avoir une instance de **Employe** sans se soucier des instances qui servent à construire un objet de type **Employe** (ici **Role** et **Adresse**).

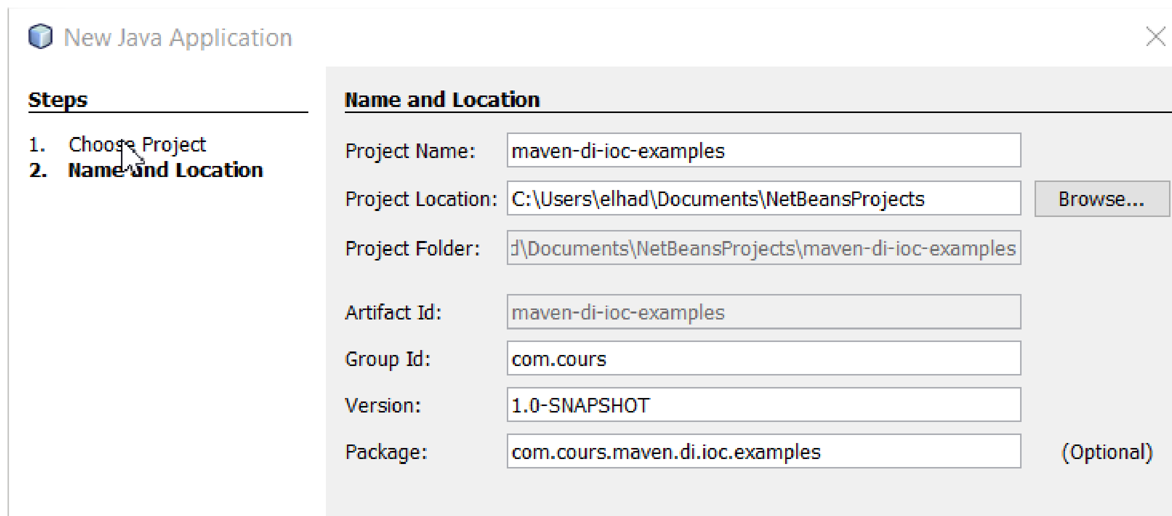


Dans l'exemple ci dessus c'est le framework « **Mon injecteur de dépendance** » qui se charge d'injecter les instances necessaire à la contruction d'un objet de type **Employe** permettant ainsi de renvoyer des instances d'**Employe** sans se soucier de **Role** et **Adresse**.
En POO (programmation orienté objet) c'est ce processus que l'on appelle Injection de dépendance.
Dans les langages tel que **Java** et **.Net** ce processus d'injection est géré par un Framework portant le nom de **Spring**.

V) Injection de Dépendance, Inversion de contrôle, couplage faible, couplage forte.

Considérons une application qui calcule une moyenne de poids d'une liste de personnes.
Nous allons de ce fait avoir besoin de différentes versions de classe **DaoImpl** avec une méthode **getMoyenne** qui implémente une interface IDao avec aussi une méthode **getMoyenne**.
Nous allons aussi créer la classe **MetierImpl** et son interface **IMetier**.
Nous lancerons l'application dans sa version 1 avec la classe **com.cours.couplage.forte.MainCouplageForte**, puis dans la version 2 avec la classe **com.cours.couplage.faible.reflexive.MainReflexive**, puis dans la version 3 avec la classe **com.cours.couplage.faible.spring.xml.MainSpringXml** , puis dans la version 4 avec la classe **com.cours.couplage.faible.spring.annotation.MainSpringAnnotation** et enfin version 5 avec la classe **com.cours.couplage.faible.spring.xml.annotation.MainCouplageFaibleSpringXmlAnnotation**.

Créer le projet **maven-di-ioc-examples** (New Project → Maven→Java Application).



1) **Version 1** : Application avec couplage ou dependance forte.

l'interface **IDao** peut avoir comme contenu :

```
package com.cours.dao;

/**
 *
 * @author elhad
 */
public interface IDao {

    public Double getMoyenne();
}
```

En version copiable :

```
package com.cours.dao;

/**
 *
 * @author elhad
 */
public interface IDao {

    public Double getMoyenne();
}
```

La classe **DaoImpl** peut avoir comme contenu :

```
package com.cours.couplage.forte;

import com.cours.dao.IDao;


public class DaoImpl implements IDao {

    private final Double[] poids = new Double[]{62.0, 83.0, 85.0, 55.0, 70.0, 99.0, 110.0, 56.0,
    68.0, 96.0, 53.0, 79.5, 112.0};

    @Override
    public Double getMoyenne() {
        Double moyenne = 0.0;
        for (Double onePoids : poids) {
            moyenne += onePoids;
        }
        moyenne = moyenne / poids.length;
        return moyenne;
    }
}
```

En version copiable :

```
package com.cours.couplage.forte;

import com.cours.dao.IDao;


public class DaoImpl implements IDao {

   private final Double[] poids = new Double[]{62.0, 83.0, 85.0, 55.0, 70.0, 99.0, 110.0, 56.0,
   68.0, 96.0, 53.0, 79.5, 112.0};

   @Override
   public Double getMoyenne() {
      Double moyenne = 0.0;
      for (Double onePoids : poids) {
         moyenne += onePoids;
      }
      moyenne = moyenne / poids.length;
      return moyenne;
   }
}
```

l'interface **IMetier** peut avoir comme contenu :

```
1   package com.cours.metier;
2
3   import com.cours.dao.IDao;
4
5
6   public interface IMetier {
7
8       public Double getStatistic();
9       public IDao getDao();
10  }
11
```

En version copiable :

```
package com.cours.metier;

import com.cours.dao.IDao;


public interface IMetier {

   public Double getStatistic();
   public IDao getDao();
}
```

La classe **MetierImpl** peut avoir comme contenu :

```
1
2   package com.cours.couplage.forte.metier.v1;
3
4   import com.cours.couplage.forte.*;
5   import com.cours.metier.IMetier;
6
7   /**
8    *
9    * @author elhad
10   */
11  public class MetierImpl implements IMetier {
12
13      private DaoImpl dao = new DaoImpl();
14
15      @Override
16      public Double getStatistic() {
17          return dao.getMoyenne();
18      }
19
20      @Override
21      public DaoImpl getDao() {
22          return dao;
23      }
24  }
```

En version copiable :

```
package com.cours.couplage.forte.metier.v1;

import com.cours.couplage.forte.*;
import com.cours.metier.IMetier;

/**
 *
 * @author elhad
 */
public class MetierImpl implements IMetier {

    private DaoImpl dao = new DaoImpl();

    @Override
    public Double getStatistic() {
        return dao.getMoyenne();
    }

    @Override
    public DaoImpl getDao() {
        return dao;
    }
}
```

La classe de démarage **MainCouplageForte** peut avoir comme contenu :

```
1   package com.cours.couplage.forte;
2
3   /**
4    *
5    * @author elhad
6    */
7   public class MainCouplageForte {
8
9       /**
10       * @param args the command line arguments
11       */
12      public static void main(String[] args) {
13          // TODO code application logic here
14           com.cours.couplage.forte.metier.v1.MetierImpl metier = new com.cours.couplage.forte.metier.v1.MetierImpl();
15          System.out.println("statistic : " + metier.getStatistic());
16      }
17  }
```

En version copiable :

```
package com.cours.couplage.forte;

/**
 *
 * @author elhad
 */
public class MainCouplageForte {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
         com.cours.couplage.forte.metier.v1.MetierImpl metier = new
com.cours.couplage.forte.metier.v1.MetierImpl();
        System.out.println("statistic : " + metier.getStatistic());
    }
}
```

Le lancement de l'application donne :

```
------------------------------------------------------------------------
Building maven-di-ioc-examples 1.0-SNAPSHOT
------------------------------------------------------------------------

--- exec-maven-plugin:1.2.1:exec (default-cli) @ maven-di-ioc-examples ---
statistic : 79.11538461538461
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time: 0.770s
```

Ce code ci-dessus fait sont travail mais comporte un defaut, si on décide de changer la classe d'implementation de **IDao** alors on sera obligé de changer la classe MetierImpl. On appelle cela une dépendance forte. Il est possible de reduire cette dépendance par quelques ajustements dans le code.

La classe **MetierImpl** peut donc devenir :

```java
package com.cours.couplage.forte.metier.v2;

import com.cours.dao.IDao;
import com.cours.metier.IMetier;

/**
 *
 * @author elhad
 */
public class MetierImpl implements IMetier {

    private IDao dao;

    @Override
    public Double getStatistic() {
        return dao.getMoyenne();
    }

    @Override
    public IDao getDao() {
        return dao;
    }

    public void setDao(IDao dao) {
        this.dao = dao;
    }
}
```

En version copiable :

```
package com.cours.couplage.forte.metier.v2;

import com.cours.dao.IDao;
import com.cours.metier.IMetier;

/**
 *
 * @author elhad
 */
public class MetierImpl implements IMetier {

    private IDao dao;

    @Override
    public Double getStatistic() {
        return dao.getMoyenne();
    }

    @Override
    public IDao getDao() {
        return dao;
    }

    public void setDao(IDao dao) {
        this.dao = dao;
    }
}
```

La classe de démarage **MainCouplageForte** devient :

```
1  package com.cours.couplage.forte;
2
3  /**
4   *
5   * @author elhad
6   */
7  public class MainCouplageForte {
8
9      /**
10      * @param args the command line arguments
11      */
12     public static void main(String[] args) {
13         // TODO code application logic here
14         com.cours.couplage.forte.metier.v2.MetierImpl metier = new com.cours.couplage.forte.metier.v2.MetierImpl();
15         DaoImpl dao = new DaoImpl();
16         metier.setDao(dao);
17         System.out.println("statistic : " + metier.getStatistic());
18     }
19 }
```

En version copiable :

```
package com.cours.couplage.forte;

/**
 *
 * @author elhad
 */
public class MainCouplageForte {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        com.cours.couplage.forte.metier.v2.MetierImpl metier = new
com.cours.couplage.forte.metier.v2.MetierImpl();
        DaoImpl dao = new DaoImpl();
        metier.setDao(dao);
        System.out.println("statistic : " + metier.getStatistic());
    }
}
```

Avec le code ci-dessus on obtient le même résultat. Il est claire qu’on a réduit la dependance mais on peut faire mieux.

2) **Version 2** : Application avec couplage faible avec la reflexivité.

Nous allons dans cette version créer un fichier **config.txt** dans lequel nous allons mettre les noms des classes d'implementation de **IDao** et **IMetier**.
Le contenu du fichier sera :

```
com.cours.couplage.faible.reflexive.DaoImpl
com.cours.couplage.faible.reflexive.MetierImpl
```

La nouvelle classe de démarage **MainCouplageFaibleReflexive** devient :

```java
package com.cours.couplage.faible.reflexive;

import com.cours.dao.IDao;
import com.cours.metier.IMetier;
import java.io.File;
import java.lang.reflect.Method;
import java.util.Scanner;

public class MainCouplageFaibleReflexive {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        String methodName = "main";
        try {
            Scanner scanner = new Scanner(new File("src/main/java/config.txt"));
            String daoClassName = scanner.nextLine();
            System.out.println("daoClassName : " + daoClassName);
            Class cDao = Class.forName(daoClassName);
            IDao dao = (IDao) cDao.newInstance();
            System.out.println("moyenne : " + dao.getMoyenne());

            String metierClassName = scanner.nextLine();
            System.out.println("metierClassName : " + metierClassName);
            Class cMetier = Class.forName(metierClassName);
            IMetier metier = (IMetier) cMetier.newInstance();
            Method method = cMetier.getMethod("setDao", IDao.class);
            method.invoke(metier, dao);
            System.out.println("statistic : " + metier.getStatistic());

        } catch (Exception e) {
            System.out.println(methodName + ", Erreur lors de l'execution de la methode, Exception : " + e);
        }
    }
}
```

En version copiable :

```
package com.cours.couplage.faible.reflexive;

import com.cours.dao.IDao;
import com.cours.metier.IMetier;
import java.io.File;
import java.lang.reflect.Method;
import java.util.Scanner;

public class MainCouplageFaibleReflexive {

   /**
    * @param args the command line arguments
    */
   public static void main(String[] args) {
      // TODO code application logic here
      String methodName = "main";
      try {
         Scanner scanner = new Scanner(new File("src/main/java/config.txt"));
         String daoClassName = scanner.nextLine();
         System.out.println("daoClassName : " + daoClassName);
         Class cDao = Class.forName(daoClassName);
         IDao dao = (IDao) cDao.newInstance();
         System.out.println("moyenne : " + dao.getMoyenne());

         String metierClassName = scanner.nextLine();
         System.out.println("metierClassName : " + metierClassName);
         Class cMetier = Class.forName(metierClassName);
         IMetier metier = (IMetier) cMetier.newInstance();
         Method method = cMetier.getMethod("setDao", IDao.class);
         method.invoke(metier, dao);
         System.out.println("statistic : " + metier.getStatistic());

      } catch (Exception e) {
         System.out.println(methodName + ", Erreur lors de l'execution de la methode, Exception : " + e);
      }
   }
}
```

Ligne 18 : recupération et chargement du fichier « **config.txt** ».
Ligne 21 : recupération de la classe qui correspond à **DaoImpl**.
Ligne 22 : création d'une instance de type **DaoImpl**.
Ligne 29 : initialisation de la méthode **setDao** de la classe **MetierImpl**.
Ligne 30 : appel dynamique de la méthode **setDao** de la classe **MetierImpl**.

Au lancement de **MainCouplageFaibleReflexive** on obtient :

```
------------------------------------------------------------------------
Building maven-di-ioc-examples 1.0-SNAPSHOT
------------------------------------------------------------------------

--- exec-maven-plugin:1.2.1:exec (default-cli) @ maven-di-ioc-examples ---
daoClassName : com.cours.couplage.faible.reflexive.DaoImpl
moyenne : 79.11538461538461
metierClassName : com.cours.couplage.faible.reflexive.MetierImpl
statistic : 79.11538461538461
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
```

3) **Version 3** : Application avec couplage faible avec Spring par XML

.

Nous allons tout d'abord modifier notre fichier **pom.xml** afin de rajouter les dépendance de Spring. Le bloc dependencies du **pom.xml** devient :

```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>4.2.1.RELEASE</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>4.2.1.RELEASE</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>4.2.1.RELEASE</version>
  </dependency>
</dependencies>
```

Nous allons aussi créer le fichier **applicationContextXml.xml** dans le dossier **resources**. Son contenu sera :

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <beans xmlns="http://www.springframework.org/schema/beans"
3          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4          xsi:schemaLocation="http://www.springframework.org/schema/beans
5     http://www.springframework.org/schema/beans/spring-beans-4.2.xsd">
6     <bean class="com.cours.couplage.faible.spring.xml.DaoImpl" id="myDao"></bean>
7     <bean class="com.cours.couplage.faible.spring.xml.MetierImpl" id="metier">
8       <property name="dao" ref="myDao"></property>
9     </bean>
10  </beans>
11
```

En version copiable :

```
<?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-4.2.xsd">
  <bean class="com.cours.couplage.faible.spring.xml.DaoImpl" id="myDao"></bean>
  <bean class="com.cours.couplage.faible.spring.xml.MetierImpl" id="metier">
    <property name="dao" ref="myDao"></property>
  </bean>
</beans>
```

Ligne 6 : création d'un bean d'identifiant **myDao** de type **DaoImpl**.
Ligne 8 : injection du bean d'identifiant **myDao** de type **DaoImpl** dans le bean d'identifiant **metier** de type **MetierImpl**.

La nouvelle classe de démarrage **MainCouplageFaibleSpringXml** devient :

```
1  package com.cours.couplage.faible.spring.xml;
2
3  import com.cours.metier.IMetier;
4  import org.springframework.context.ApplicationContext;
5  import org.springframework.context.support.ClassPathXmlApplicationContext;
6
7
8  public class MainCouplageFaibleSpringXml {
9
10     /**
11      * @param args the command line arguments
12      */
13     public static void main(String[] args) {
14         // TODO code application logic here
15         ApplicationContext context = new ClassPathXmlApplicationContext("applicationContextXml.xml");
16         IMetier metier = (IMetier) context.getBean("metier");
17         System.out.println("statistic : " + metier.getStatistic());
18     }
19 }
```

En version copiable :

```
package com.cours.couplage.faible.spring.xml;

import com.cours.metier.IMetier;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;


public class MainCouplageFaibleSpringXml {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationContextXml.xml");
        IMetier metier = (IMetier) context.getBean("metier");
        System.out.println("statistic : " + metier.getStatistic());
    }
}
```

Ligne 11 : chargement du fichier **applicationContextXml.xml** dans le contextde Spring.
Ligne 12 : recupération du Bean d'identifiant **metier** de type **MetierImpl**.

Le lancement de la classe **MainCouplageFaibleSpringXml** de l'application donne comme resultat :

```
------------------------------------------------------------------------
Building maven-di-ioc-examples 1.0-SNAPSHOT
------------------------------------------------------------------------

--- exec-maven-plugin:1.2.1:exec (default-cli) @ maven-di-ioc-examples ---
mai 13, 2019 4:00:48 PM org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh
INFOS: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@5d099f62: startup date [Mon May 13 16:00:48 CEST 2019];
mai 13, 2019 4:00:48 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
INFOS: Loading XML bean definitions from class path resource [applicationContextXml.xml]
statistic : 79.11538461538461
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time: 1.264s
```

4) **Version 4** : Application avec couplage faible avec Spring par annotation

.

Nous allons cette fois-ci instancier les beans de spring par annotation et non plus par XML.

Créer la classe **com.cours.couplage.faible.spring.annotation.DaoImpl** avec le contenu :

```
1   package com.cours.couplage.faible.spring.annotation;
2
3   import com.cours.dao.IDao;
4   import org.springframework.stereotype.Component;
5
6   /**
7    *
8    * @author elhad
9    */
10  @Component
11  public class DaoImpl implements IDao {
12
13      private final Double[] poids = new Double[]{62.0, 83.0, 85.0, 55.0, 70.0, 99.0, 110.0, 56.0, 68.0, 96.0, 53.0, 79.5, 112.0};
14
15      @Override
16      public Double getMoyenne() {
17          Double moyenne = 0.0;
18          for (Double onePoids : poids) {
19              moyenne += onePoids;
20          }
21          moyenne = moyenne / poids.length;
22          return moyenne;
23      }
24  }
```

En version copiable :

```
package com.cours.couplage.faible.spring.annotation;

import com.cours.dao.IDao;
import org.springframework.stereotype.Component;

/**
 *
 * @author elhad
 */
@Component
public class DaoImpl implements IDao {

    private final Double[] poids = new Double[]{62.0, 83.0, 85.0, 55.0, 70.0, 99.0, 110.0, 56.0, 68.0, 96.0, 53.0, 79.5,
112.0};

    @Override
    public Double getMoyenne() {
        Double moyenne = 0.0;
        for (Double onePoids : poids) {
            moyenne += onePoids;
        }
        moyenne = moyenne / poids.length;
        return moyenne;
    }
}
```

Ligne 5 : Mise en place de l'annotation @Component pour que Spring puisse charger cette classe dans son context.

La classe **MetierImpl** devient :

```
1  package com.cours.couplage.faible.spring.annotation;
2
3  import com.cours.dao.IDao;
4  import com.cours.metier.IMetier;
5  import org.springframework.beans.factory.annotation.Autowired;
6  import org.springframework.stereotype.Component;
7
8  /**
9   *
10  * @author elhad
11  */
12 @Component
13 public class MetierImpl implements IMetier {
14
15     @Autowired
16     private IDao dao;
17
18     @Override
19     public Double getStatistic() {
20         return dao.getMoyenne();
21     }
22
23     @Override
24     public IDao getDao() {
25         return dao;
26     }
27
28     public void setDao(IDao dao) {
29         this.dao = dao;
30     }
31 }
```

En version copiable :

```
package com.cours.couplage.faible.spring.annotation;

import com.cours.dao.IDao;
import com.cours.metier.IMetier;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

/**
 *
 * @author elhad
 */
@Component
public class MetierImpl implements IMetier {

    @Autowired
    private IDao dao;

    @Override
    public Double getStatistic() {
        return dao.getMoyenne();
    }
```

```
  @Override
  public IDao getDao() {
    return dao;
  }

  public void setDao(IDao dao) {
    this.dao = dao;
  }
}
```

Ligne 8 : Mise en place de l'annotation @Component pour que Spring puisse charger cette classe dans son context.
Ligne 11 : Mise en place de l'annotation @Autowired pour que Spring puisse injecter l'implementation de l'interface IDao qu'il trouvera dans son context.

La nouvelle classe de démarage **MainCouplageFaibleSpringAnnotation** devient :

```
1   package com.cours.couplage.faible.spring.annotation;
2
3   import com.cours.metier.IMetier;
4   import org.springframework.context.ApplicationContext;
5   import org.springframework.context.annotation.AnnotationConfigApplicationContext;
6
7   /**
8    *
9    * @author elhad
10   */
11  public class MainCouplageFaibleSpringAnnotation {
12
13      /**
14       * @param args the command line arguments
15       */
16      public static void main(String[] args) {
17          // TODO code application logic here
18          ApplicationContext context = new AnnotationConfigApplicationContext("com.cours.couplage.faible.spring.annotation");
19          IMetier metier = context.getBean(IMetier.class);
20          System.out.println("statistic : " + metier.getStatistic());
21      }
22  }
```

En version copiable :

```
package com.cours.couplage.faible.spring.annotation;

import com.cours.metier.IMetier;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class MainCouplageFaibleSpringAnnotation {

  public static void main(String[] args) {
    // TODO code application logic here
    ApplicationContext context = new
AnnotationConfigApplicationContext("com.cours.couplage.faible.spring.annotation");
    IMetier metier = context.getBean(IMetier.class);
    System.out.println("statistic : " + metier.getStatistic());
  }
}
```

Ligne 11 : chargement du package **com.cours.couplage.faible.spring.annotation** (il est possible de charger plusieurs packages en mettant les nom de package séparés par des virgules)si **Spring** trouve dans ces classes les annotations **@Component, @Repository et @Service**.
Ligne 12 : recupération du Bean **Metier** qui implémente l'interface **IMetier**.

Par convention les DAO porteront l'annotation **@Repository** et les facades metier porteront l'annotation **@Service.**

Le lancement de la classe **MainCouplageFaibleSpringAnnotation** donne comme resultat :

```
------------------------------------------------------------------------
Building maven-di-ioc-examples 1.0-SNAPSHOT
------------------------------------------------------------------------

--- exec-maven-plugin:1.2.1:exec (default-cli) @ maven-di-ioc-examples ---
mai 13, 2019 6:55:23 PM org.springframework.context.annotation.AnnotationConfigApplicationContext prepareRefresh
INFOS: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@300ffa5d: startup date [Mon May 13
statistic : 79.11538461538461
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time: 1.163s
```

5) **Version 5** : Application avec couplage faible avec Spring par XML/ Annotation

Il est tout a fait possible de melanger des configuration de Spring par annotation et par XML.

La classe **com.cours.couplage.faible.spring.xml.annotation.DaoImpl** devient :

```java
1   package com.cours.couplage.faible.spring.xml.annotation;
2
3   import com.cours.dao.*;
4   import org.springframework.stereotype.Repository;
5
6   /**
7    *
8    * @author elhad
9    */
10  @Repository("myDao")
11  public class DaoImpl implements IDao {
12
13      private final Double[] poids = new Double[]{62.0, 83.0, 85.0, 55.0, 70.0, 99.0, 110.0, 56.0, 68.0, 96.0, 53.0, 79.5, 112.0};
14
15      @Override
16      public Double getMoyenne() {
17          Double moyenne = 0.0;
18
19          for (Double onePoids : poids) {
20              moyenne += onePoids;
21          }
22          moyenne = moyenne / poids.length;
23          return moyenne;
24      }
25
26  }
27
```

En version copiable :

```java
package com.cours.couplage.faible.spring.xml.annotation;

import com.cours.dao.*;
import org.springframework.stereotype.Repository;

@Repository("myDao")
public class DaoImpl implements IDao {

   private final Double[] poids = new Double[]{62.0, 83.0, 85.0, 55.0, 70.0, 99.0, 110.0, 56.0, 68.0, 96.0, 53.0, 79.5,
112.0};

   @Override
   public Double getMoyenne() {
      Double moyenne = 0.0;

      for (Double onePoids : poids) {
         moyenne += onePoids;
      }
      moyenne = moyenne / poids.length;
      return moyenne;
   }
}
```

Ligne 5 : Mise en place de l'annotation @ Repository pour que Spring puisse charger cette classe dans son context.

La classe **MetierImpl** devient :

```
package com.cours.couplage.faible.spring.xml.annotation;

import com.cours.metier.*;
import com.cours.dao.IDao;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service("metier")
public class MetierImpl implements IMetier {

    @Autowired
    private IDao dao;

    @Override
    public Double getStatistic() {
        return dao.getMoyenne();
    }

    @Override
    public IDao getDao() {
        return dao;
    }

    public void setDao(IDao dao) {
        this.dao = dao;
    }
}
```

En version copiable :

```
package com.cours.couplage.faible.spring.xml.annotation;

import com.cours.metier.*;
import com.cours.dao.IDao;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service("metier")
public class MetierImpl implements IMetier {

    @Autowired
    private IDao dao;

    @Override
    public Double getStatistic() {
        return dao.getMoyenne();
    }

    @Override
    public IDao getDao() {
        return dao;
    }
```

```
    public void setDao(IDao dao) {
        this.dao = dao;
    }
}
```

Ligne 8 : Mise en place de l'annotation @ Service pour que Spring puisse charger cette classe dans son context.
Ligne 11 : Mise en place de l'annotation @Autowired pour que Spring puisse injecter l'implementation de l'interface IDao qu'il trouvera dans son context.

Le fichier **applicationContextXmlAnnotation.xml** devient :

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xmlns:context="http://www.springframework.org/schema/context"
5         xsi:schemaLocation="http://www.springframework.org/schema/beans
6         http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
7         http://www.springframework.org/schema/context
8         http://www.springframework.org/schema/context/spring-context-4.2.xsd">
9
10     <context:component-scan base-package="com.cours.couplage.faible.spring.xml.annotation" />
11 </beans>
```

En version copiable :

```
<?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"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.2.xsd">

  <context:component-scan base-package="com.cours.couplage.faible.spring.xml.annotation" />
</beans>
```

Ligne 10 : Utilisation de la fonction scan de Spring afn de trouver dans les packages **com.cours.dao et com.cours.metier** les annotations **@Component, @Repository et @Service**.

La nouvelle classe de démarage **MainCouplageFaibleSpringXmlAnnotation** sera :

```
1  package com.cours.couplage.faible.spring.xml.annotation;
2
3  import com.cours.dao.IDao;
4  import com.cours.metier.IMetier;
5  import org.springframework.context.ApplicationContext;
6  import org.springframework.context.support.ClassPathXmlApplicationContext;
7
8  public class MainCouplageFaibleSpringXmlAnnotation {
9
10     public static void main(String[] args) {
11         // TODO code application logic here
12         ApplicationContext context = new ClassPathXmlApplicationContext("applicationContextXmlAnnotation.xml");
13         IDao dao = (IDao) context.getBean("myDao");
14         IMetier metier = (IMetier) context.getBean("metier");
15         System.out.println("moyenne : " + dao.getMoyenne());
16         System.out.println("statistic : " + metier.getStatistic());
17     }
18 }
```

En version copiable :

```
package com.cours.couplage.faible.spring.xml.annotation;

import com.cours.dao.IDao;
import com.cours.metier.IMetier;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MainCouplageFaibleSpringXmlAnnotation {

    public static void main(String[] args) {
        // TODO code application logic here
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContextXmlAnnotation.xml");
        IDao dao = (IDao) context.getBean("myDao");
        IMetier metier = (IMetier) context.getBean("metier");
        System.out.println("moyenne : " + dao.getMoyenne());
        System.out.println("statistic : " + metier.getStatistic());
    }
}
```

Le lancement de la classe **MainCouplageFaibleSpringXmlAnnotation** de l'application donne comme resultat :

```
------------------------------------------------------------------------
Building maven-di-ioc-examples 1.0-SNAPSHOT
------------------------------------------------------------------------

--- exec-maven-plugin:1.2.1:exec (default-cli) @ maven-di-ioc-examples ---
mai 13, 2019 7:29:11 PM org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh
INFOS: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@5d099f62: startup date [Mon May 13 19:29:11
mai 13, 2019 7:29:11 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
INFOS: Loading XML bean definitions from class path resource [applicationContextXmlAnnotation.xml]
moyenne : 79.11538461538461
statistic : 79.11538461538461
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time: 1.423s
```

VI) SpEL (Spring Expression Language)

1. Introduction

Le langage Spring Expression Language (SpEL) est un langage d'expression puissant qui prend en charge l'interrogation et la manipulation d'un graphe d'objets lors de l'exécution. Nous pouvons l'utiliser avec des configurations Spring basées sur XML ou des annotations.

Plusieurs opérateurs sont disponibles dans le langage :

Type	**Operators**
Arithmetic	+, -, *, /, %, ^, div, mod
Relational	<, >, ==, !=, <=, >=, lt, gt, eq, ne, le, ge
Logical	and, or, not, &&, ||, !
Conditional	?:
Regex	matches

2. *Operateur*

Pour ces exemples, nous utiliserons une configuration basée sur des annotations.

Les expressions SpEL commencent par le symbole # et sont entourées d'accolades : #{expression}.

Les propriétés peuvent être référencées de manière similaire, en commençant par un symbole $ et en les entourant d'accolades : ${property.name}.

Les espaces réservés aux propriétés ne peuvent pas contenir d'expressions SpEL, mais les expressions peuvent contenir des références de propriétés :

#{${someProperty} + 2}

Dans l'exemple ci-dessus, supposons que someProperty a la valeur 2, donc l'expression résultante serait 2 + 2, qui serait évaluée à 4.

a) Les opérateurs arithmétiques

```
@Value("#{19 + 1}") // 20
private double add;

@Value("#{'String1 ' + 'string2'}") // "String1 string2"
private String addString;

@Value("#{20 - 1}") // 19
private double subtract;

@Value("#{10 * 2}") // 20
private double multiply;

@Value("#{36 / 2}") // 19
private double divide;

@Value("#{36 div 2}") // 18, the same as for / operator
private double divideAlphabetic;

@Value("#{37 % 10}") // 7
private double modulo;

@Value("#{37 mod 10}") // 7, the same as for % operator
private double moduloAlphabetic;

@Value("#{2 ^ 9}") // 512
private double powerOf;

@Value("#{(2 + 2) * 2 + 9}") // 17
private double brackets;
```

b) Opérateurs relationnels et logiques

SpEL prend également en charge toutes les opérations relationnelles et logiques de base :

```
@Value("#{19 + 1}") // 20
private double add;

@Value("#{'String1 ' + 'string2'}") // "String1 string2"
private String addString;

@Value("#{20 - 1}") // 19
private double subtract;

@Value("#{10 * 2}") // 20
private double multiply;

@Value("#{36 / 2}") // 19
private double divide;

@Value("#{36 div 2}") // 18, the same as for / operator
private double divideAlphabetic;

@Value("#{37 % 10}") // 7
private double modulo;

@Value("#{37 mod 10}") // 7, the same as for % operator
private double moduloAlphabetic;

@Value("#{2 ^ 9}") // 512
private double powerOf;

@Value("#{(2 + 2) * 2 + 9}") // 17
private double brackets;
```

Tous les opérateurs relationnels ont également des alias alphabétiques. Par exemple, dans les configurations basées sur XML, nous ne pouvons pas utiliser d'opérateurs contenant des crochets angulaires (<, <=, >, >=). À la place, nous pouvons utiliser lt (inférieur à), le (inférieur ou égal à), gt (supérieur à) ou ge (supérieur ou égal à).

c) Opérateurs logiques

SpEL prend également en charge toutes les opérations logiques de base :

```
@Value("#{250 > 200 && 200 < 4000}") // true
private boolean and;

@Value("#{250 > 200 and 200 < 4000}") // true
private boolean andAlphabetic;

@Value("#{400 > 300 || 150 < 100}") // true
private boolean or;

@Value("#{400 > 300 or 150 < 100}") // true
private boolean orAlphabetic;

@Value("#{!true}") // false
private boolean not;

@Value("#{not true}") // false
private boolean notAlphabetic;
```

d) Utilisation de Regex dans SpEL

Nous pouvons utiliser l'opérateur matches pour vérifier si une chaîne correspond ou non à une expression régulière donnée :

```
@Value("#{'100' matches '\\d+' }") // true
private boolean validNumericStringResult;

@Value("#{'100fghdjf' matches '\\d+' }") // false
private boolean invalidNumericStringResult;

@Value("#{'valid alphabetic string' matches '[a-zA-Z\\s]+' }") // true
private boolean validAlphabeticStringResult;

@Value("#{'invalid alphabetic string #$1' matches '[a-zA-Z\\s]+' }") // false
private boolean invalidAlphabeticStringResult;

@Value("#{someBean.someValue matches '\d+'}") // true if someValue contains only digits
private boolean validNumericValue;
```

e) Accéder aux objets d'une Liste et d'un Map

Avec l'aide de SpEL, nous pouvons accéder au contenu de n'importe quelle Map ou liste dans le contexte.

Nous allons créer un nouveau bean **carPark** qui stockera des informations sur certaines voitures et leurs conducteurs dans une liste et une carte :

```
@Component("carPark")
public class CarPark {
    private List<Car> cars = new ArrayList<>();
    private Map<String, Car> carsByDriver = new HashMap<>();
    public CarPark() {
        Car model1 = new Car();
        model1.setMake("Good company");
        model1.setModel("Model1");
        model1.setYearOfProduction(1998);
        Car model2 = new Car();
        model2.setMake("Good company");
        model2.setModel("Model2");
        model2.setYearOfProduction(2005);
        cars.add(model1);
        cars.add(model2);
        carsByDriver.put("Driver1", model1);
        carsByDriver.put("Driver2", model2);
    }
    //Getters and setters
}
```

Nous pouvons maintenant accéder aux valeurs des collections en utilisant SpEL :

```
@Value("#{carPark.carsByDriver['Driver1']}") // Model1
private Car driver1Car;
@Value("#{carPark.carsByDriver['Driver2']}") // Model2
private Car driver2Car;
@Value("#{carPark.cars[0]}") // Model1
private Car firstCarInPark;
@Value("#{carPark.cars.size()}") // Model2
private Integer numberOfCarsInPark;
```

3. *Utiliser le SpEl dans Spring configuration*

Dans cet exemple, nous verrons comment utiliser SpEL dans une configuration basée sur XML. Nous pouvons utiliser des expressions pour référencer des beans ou des champs/méthodes de beans.

Par exemple, supposons que nous ayons les classes suivantes :

```
public class Engine {
  private int capacity;
  private int horsePower;
  private int numberOfCylinders;

 // Getters and setters
}

public class Car {
  private String make;
  private int model;
  private Engine engine;
  private int horsePower;

 // Getters and setters
}
```

Nous créons maintenant un contexte d'application dans lequel des expressions sont utilisées pour injecter des valeurs :

```
<bean id="engine" class="com.training.Engine">
   <property name="capacity" value="3200"/>
   <property name="horsePower" value="250"/>
   <property name="numberOfCylinders" value="6"/>
</bean>
<bean id="someCar" class="com.training.Car">
   <property name="make" value="Some make"/>
   <property name="model" value="Some model"/>
   <property name="engine" value="#{engine}"/>
   <property name="horsePower" value="#{engine.horsePower}"/>
</bean>
```

VII) Spring profiles

Les profils sont une fonctionnalité essentielle du framework, nous permettant de mapper nos beans à différents profils, par exemple, dev, test et prod.

Nous pouvons ensuite activer différents profils dans différents environnements pour amorcer uniquement les beans dont nous avons besoin.

1. *Utiliser @Profile sur un Bean*

Commençons simplement et voyons comment nous pouvons faire en sorte qu'un bean appartienne à un profil particulier. Nous utilisons l'annotation @Profile : nous mappons le bean à ce profil particulier ; l'annotation prend simplement les noms d'un (ou de plusieurs) profils.

Prenons un scénario de base : nous avons un bean qui ne doit être actif que pendant le développement mais pas déployé en production.

Nous annotons ce bean avec un profil dev, et il ne sera présent dans le conteneur que pendant le développement. En production, le dev ne sera tout simplement pas actif :

```
@Component
@Profile("dev")
public class DevDatasourceConfig
```

Remarque rapide : les noms de profil peuvent également être préfixés par un opérateur NOT, par exemple !dev, pour les exclure d'un profil.

Dans l'exemple, le composant n'est activé que si le profil dev n'est pas actif :

```
@Component
@Profile("!dev")
public class DevDatasourceConfig
```

2. *Declaration Profiles en XML*

Les profils peuvent également être configurés en XML. La balise <beans> possède un attribut profile, qui prend des valeurs séparées par des virgules des profils applicables :

```
<beans profile="dev">
  <bean id="devDatasourceConfig"
    class="com.training.profiles.DevDatasourceConfig" />
</beans>
```

3. *Définir des profils*

L'étape suivante consiste à activer et à définir les profils afin que les beans respectifs soient enregistrés dans le conteneur.

a) Par programmation via l'interface WebApplicationInitializer

Dans les applications Web, WebApplicationInitializer peut être utilisé pour configurer le ServletContext par programmation.

C'est également un endroit très pratique pour définir nos profils actifs par programmation :

```
@Configuration
public class MyWebApplicationInitializer
  implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {

        servletContext.setInitParameter(
          "spring.profiles.active", "dev");
    }
}
```

b) Par programmation via ConfigurableEnvironment

Nous pouvons également définir des profils directement sur l'environnement :

```
@Autowired
private ConfigurableEnvironment env;
...
env.setActiveProfiles("someProfile");
```

c) Context Parameter sur web.xml

De même, nous pouvons définir les profils actifs dans le fichier web.xml de l'application Web à l'aide d'un paramètre de contexte :

```
<context-param>
   <param-name>contextConfigLocation</param-name>
   <param-value>/WEB-INF/app-config.xml</param-value>
</context-param>
<context-param>
   <param-name>spring.profiles.active</param-name>
   <param-value>dev</param-value>
</context-param>
```

d) Par paramètres système de la JVM

Les noms de profil peuvent également être transmis via un paramètre système JVM. Ces profils seront activés au démarrage de l'application :

-Dspring.profiles.active=dev

e) Par profil Maven

Les profils Spring peuvent également être activés via les profils Maven en spécifiant la propriété de configuration **spring.profiles.active**.

Dans chaque profil Maven, nous pouvons définir une propriété **spring.profiles.active** :

```
<profiles>
  <profile>
    <id>dev</id>
    <activation>
      <activeByDefault>true</activeByDefault>
    </activation>
    <properties>
      <spring.profiles.active>dev</spring.profiles.active>
    </properties>
  </profile>
  <profile>
    <id>prod</id>
    <properties>
      <spring.profiles.active>prod</spring.profiles.active>
    </properties>
  </profile>
</profiles>
```

Sa valeur sera utilisée pour remplacer l'espace réservé @spring.profiles.active@ dans application.properties :

spring.profiles.active=@spring.profiles.active@

Nous devons maintenant activer le filtrage des ressources dans **pom.xml** :

```
<build>
  <resources>
    <resource>
      <directory>src/main/resources</directory>
      <filtering>true</filtering>
    </resource>
  </resources>
  ...
</build>
```

et ajoutez un paramètre -P pour changer le profil Maven qui sera appliqué :

mvn clean package -Pprod

Cette commande empaquetera l'application pour le profil prod. Elle applique également la valeur **spring.profiles.active** prod pour cette application lorsqu'elle est en cours d'exécution.

f) @ActiveProfile dans les tests

Les tests permettent de spécifier très facilement quels profils sont actifs à l'aide de l'annotation **@ActiveProfile** pour activer des profils spécifiques :

```
@ActiveProfiles("dev")
```

VIII) Spring Bean scopes

Lorsque vous créez une définition de bean, vous créez une recette pour créer des instances réelles de la classe définie par cette définition de bean. L'idée qu'une définition de bean est une recette est importante, car cela signifie que, comme avec une classe, vous pouvez créer de nombreuses instances d'objet à partir d'une seule recette.

Vous pouvez contrôler non seulement les différentes dépendances et valeurs de configuration qui doivent être connectées à un objet créé à partir d'une définition de bean particulière, mais également contrôler la portée des objets créés à partir d'une définition de bean particulière. Cette approche est puissante et flexible, car vous pouvez choisir la portée des objets que vous créez via la configuration au lieu de devoir intégrer la portée d'un objet au niveau de la classe Java. Les beans peuvent être définis pour être déployés dans l'une des nombreuses portées. Spring Framework prend en charge six portées, dont quatre ne sont disponibles que si vous utilisez un ApplicationContext compatible Web. Vous pouvez également créer une portée personnalisée.

Le tableau suivant décrit les portées prises en charge :

Scope	Description
singleton	(Par défaut) Scope une définition de bean unique à une instance d'objet unique pour chaque conteneur Spring IoC.
prototype	Scope une définition de bean unique à n'importe quel nombre d'instances d'objet.
request	Scope une définition de bean unique au cycle de vie d'une seule requête HTTP. Autrement dit, chaque requête HTTP possède sa propre instance d'un bean créé à partir d'une définition de bean unique. Valide uniquement dans le contexte d'un Spring ApplicationContext compatible Web.
session	Scope une définition de bean unique au cycle de vie d'une session HTTP. Valide uniquement dans le contexte d'un Spring ApplicationContext compatible Web.
application	Scope une définition de bean unique au cycle de vie d'un ServletContext. Valide uniquement dans le contexte d'un Spring ApplicationContext compatible Web.
websocket	Scope une définition de bean unique au cycle de vie d'un WebSocket. Valide uniquement dans le contexte d'un Spring ApplicationContext compatible Web.

IX) Les modes d’Autowire avec Spring

Le conteneur Spring peut établir automatiquement des relations entre les beans collaboratifs (beans dépendants). Jusqu'à l'injection de dépendances explicite (comme dans la configuration XML), Spring résout automatiquement les collaborateurs en inspectant le contenu de l'ApplicationContext, c'est-à-dire les beans enregistrés avec celui-ci.

Le tableau suivant répertorie les 4 différents modes de câblage automatique des haricots dans Spring :

Mode Autowire	Description
no	Il n'y a pas de magie de Autowire automatique en arrière-plan. Les références de bean doivent être définies explicitement, soit à l'aide d'une configuration XML, soit d'une annotation @Qualifier.
byName	Autowire automatique en faisant correspondre le nom de la propriété avec le nom du bean, si le mode est défini sur « byName ». Ceci est utile s'il existe plusieurs beans du même type mais avec des noms différents.
byType	Autowire automatique en faisant correspondre le type de propriété avec le type de bean, s'il existe exactement un bean de ce type dans le contexte de l'application. Lorsque l’Autowire automatique « byType » est utilisé et qu'il existe plusieurs beans de ce type, une exception fatale est levée.
constructor	Similaire à byType mais s'applique aux arguments du constructeur.