Université de Montpellier
Faculté des Sciences
Département Informatique

Filière : Master Informatique **Intervenant :** Chouki TIBERMACINE

TP 4
Introduction à la DI dans le framework Spring

Mise en place de l'environnement de travail.

- Récupérer l'archive spring-base.zip qui se trouve sur Moodle
- Créer un nouveau projet Java sous Eclipse
- Créer un répertoire nommé lib/ dans ce projet
- Décompresser l'archive sur votre disque (dans le répertoire lib/ de votre projet Eclipse)
- Ajouter les JARs de l'archive décompressée dans le Build Path du projet
- Vous êtes maintenant prêts à utiliser Spring dans votre projet Java

Il est possible également de réaliser vos projets Spring sur l'outil : *Spring Tool Suite* (STS) fourni par *VMWare*. Cet outil est disponible sur le site : http://spring.io/tools/sts

Exercice 1. Une application jouet

Écrire une classe Helloer qui fournit une méthode sayHello(), et une classe Printer fournissant une méthode printHello(). La première classe dispose d'une propriété référençant un Printer. Ce dernier affiche simplement le message dans la console.

Utiliser la solution XML pure proposée par le framework Spring pour déclarer vos classes comme beans et les connecter.

Écrire une classe (`Application.java`) contenant un `main(...)` qui démarre la DI.

N.-B. : Chaque classe doit implémenter et/ou requérir une ou plusieurs interfaces ; une classe ne doit pas directement référencer une autre classe.

Étendre l'application pour que le Printer affiche :

- « Bonjour tout le monde » sur la console : ajouter une deuxième propriété au bean Helloer qui stocke le message à afficher (« Hello World », « Bonjour tout le monde », « Hola, mundo », ...). Celle-ci remplace l'argument des méthodes.
- « Hello World » sur une fenêtre graphique. Dans ce cas, écrire un nouveau bean (GraphicalPrinter) puis reconfigurer l'application (ne pas toucher au bean Helloer). Utiliser ici : JOptionPane.showMessageDialog(null, message);

Tester ensuite la solution basée sur les annotations : @Autowired, @Value et @Component

Exercice 2. Votre propre injecteur de dépendances

En se basant sur Java Reflect, les annotations et un chargeur de classe personnalisé (nous allons nous séparer de Spring ici), nous allons créer un injecteur de dépendances simple (classe Injecteur avec un main) permettant de créer un objet d'une classe B, et injecter sa référence comme valeur de l'attribut de type B d'un objet de la classe A.
Voici le code de ces classes :

```
public class A {
    public B b;
    public void m() { System.out.println("Je suis m de A."); b.n(); }
}
public class B {
    public void n() { System.out.println("Je suis n de B."); }
}
```

Question 1. Sans chargeur de classes
Pour commencer, nous allons supposer que les classes A et B sont accessibles depuis le CLASSPATH (les créer par exemple dans le même projet que votre programme d'injection de dépendances).

En utilisant Java Reflect :

- obtenir l'objet qui réifie la classe A en utilisant une des trois instructions (préférence pour la première, où le nom de la classe peut être reçu en paramètre) :
  - `Class<?> classe = Class.forName("A");`
  - `Class<?> classe = new A().getClass();`
  - `Class<?> classe = A.class;`
- instancier A : `Object o = classe.newInstance()`
- rechercher les attributs de l'instance de A qui contiennent null et leur affecter comme valeur une instance de la classe correspondant à leur type (pour notre exemple, instance de B pour l'attribut b). Utiliser les méthodes :
  - `Field[] getFields()` de `Class`
  - `Object get(Object o)` de `Field`
  - `void set(Object o, Object o1)` de `Field`
- appeler la méthode m sur l'instance de A : `classe.getMethod("m").invoke(o)`

A l'exécution, si vous avez une exception de type NullPointerException, votre injecteur ne fonctionne pas (l'attribut b n'est toujours pas initialisé, il contient null, alors qu'il y a une invocation de méthode n() sur celui-ci). Si vous voyez les messages suivants s'afficher, votre injecteur fonctionne :

```
Je suis m de A.
Je suis n de B.
```

Noter ici, que nous simplifions au maximum l'injection de dépendances. Nous supposons :

1. que les types des attributs sont toujours des classes, alors que ça peut être un type primitif, un tableau ou une interface
2. qu'il y a toujours un constructeur sans paramètres défini dans les classes qu'on instancie
3. que les attributs qu'on injecte sont publiques

Question 2. Utilisation d'un chargeur de classes
Nous allons maintenant recevoir les noms des classes des objets à connecter à l'exécution (tableau args de main, par exemple). Ces classes ne sont pas nécessairement chargées déjà par la JVM (ne sont pas référencées dans le CLASSPATH ou ne sont pas des classes de la librairie standard Java).

Dans les versions actuelles du SDK Java, il est impossible de modifier le CLASSPATH dynamiquement. Il faudra donc charger les classes programmatiquement en utilisant un chargeur de classes personnalisé. Celui-ci va spécialiser le chargeur de classe de la JVM pour lire le byte-code des classes (en analysant leurs fichiers .class) à partir d'un répertoire du disque (ça peut être aussi, à partir d'un URL) et les charger dans la JVM.
Pour aller vite dans ce TP, je fournis un chargeur de classes simple qui répond à ce besoin. Je l'ai mis dans la page Moodle du cours.

Lire attentivement le code de ce chargeur pour comprendre le fonctionnement d'un chargeur de classe. Ensuite, le télécharger et l'ajouter à votre projet Java.

Vous pouvez utiliser ce chargeur de la façon suivante. Ce code remplacera l'instruction permettant l'obtention de l'objet Class qui représente la classe A.

```
BasicClassLoader loader = new BasicClassLoader();
Class<?> classe = loader.loadClassInDirectory("A","<votre répertoire>");
```

Déplacer les classes A et B dans un répertoire de votre choix que vous indiquerez comme argument dans l'instruction ci-dessus. Le reste de votre code est inchangé. L'exécuter.

Noter que le chargeur de classe s'occupe du chargement (à la volée) des autres classes (types) dont dépend la classe que nous avons explicitement chargé ci-dessus (la classe A). Vous avez remarqué que nous n'avons pas demandé le chargement de la classe B. Celle-ci a été chargée au moment de l'introspection des attributs de A (avec getFields()). Pour récupérer les informations sur ces attributs, et donc le type de l'attribut b, la JVM lance le chargement de B en utilisant le même chargeur de classe (BasicClassLoader).

Question 3. Avec des annotations
Créer votre propre annotation que vous nommerez @ConnectMe pour marquer les attributs qu'il faudra injecter. Le code de cette annotation est donné ci-dessous :

```
import java.lang.annotation.*;
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface ConnectMe { }
```

Je rappelle ici le sens des deux (méta-)annotations appliquées à l'annotation ConnectMe :

- `Target` désigne la cible sur laquelle l'annotation peut être appliquée. Ici, l'annotation ne peut être appliquée qu'aux attributs

- `Retention` désigne la portée de l'annotation :
  - SOURCE : l'annotation est supprimée lors de la compilation (n'existe que dans le code source)
  - CLASS : l'annotation est Insérée dans le byte-code, mais elle est ignorée par la JVM à l'exécution (non disponible lors de l'introspection)
  - RUNTIME (ce qui est le cas ici) : l'annotation doit être maintenue à l'exécution

pour que l'on puisse faire de l'introspection sur les objets et voir si leurs champs (attributs) sont annotés

Modifier la classe A pour apposer l'annotation ConnectMe sur l'attribut b :

```
@ConnectMe
public B b;
```

Modifier l'injecteur de dépendances pour vérifier d'abord si l'attribut est annoté `@ConnectMe` avant d'injecter une référence dans celui-ci. Il faudra invoquer la méthode suivante de l'objet Field :

```
boolean isAnnotationPresent(ConnectMe.class)
```

Vous pouvez ajouter d'autres attributs annotés ou pas, de différents types (String, JFrame, ...), leur mettre différentes visibilités (public, private, ...) pour tester votre injecteur de dépendances.

Chouki TIBERMACINE