SPRING – Injection de Dépendances

esprit
Se former autrement
HONORIS UNITED UNIVERSITIES

UP ASI : Bureau E204
(Architectures des Systèmes d'Information)

Plan du Cours

- Couplage fort
- Couplage faible
- Inversion de Contrôle **IoC**
- Injection de Dépendances ID
- **Bean** : Configuration, Stéréotypes, Portées
- Implémentation de **l'ID**

Dépendance entre objets ?

- Les objets de la classe C1 dépendent des objets de la classe C2 si :
- C1 a un attribut objet de la classe C2
- C1 hérite de la classe C2
- C1 dépend d'un autre objet de type C3 qui dépend d'un objet de type C2
- Une méthode de C1 appelle une méthode de C2

Toute application Java est une composition d'objets qui collaborent pour rendre le service attendu

- **Les objets dépendent les uns des autres**

Dépendance entre objets ?

Avec la programmation orientée 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
- Utilise Couplage fort

Avec Spring, nous nous basons sur :

- Le couplage faible
- L'Injection de dépendance

Couplage : rappel

- Le **couplage** est une métrique indiquant le niveau d'interaction entre deux ou plusieurs composants logiciels (Class, Module…)

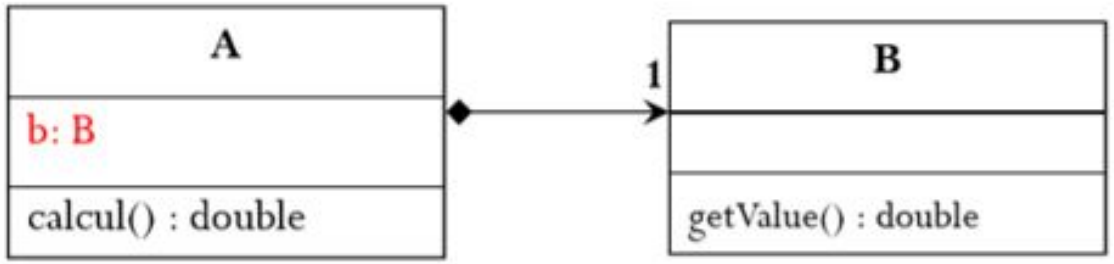
Couplage : degré de dépendance entre objets

- On parle de **couplage fort**
  si les composants échangent beaucoup d'informations.

- On parle de **couplage faible**
  si les composants échangent peu d'informations.

Couplage fort

- La classe A ne peut fonctionner qu'à la présence de la classe B !!

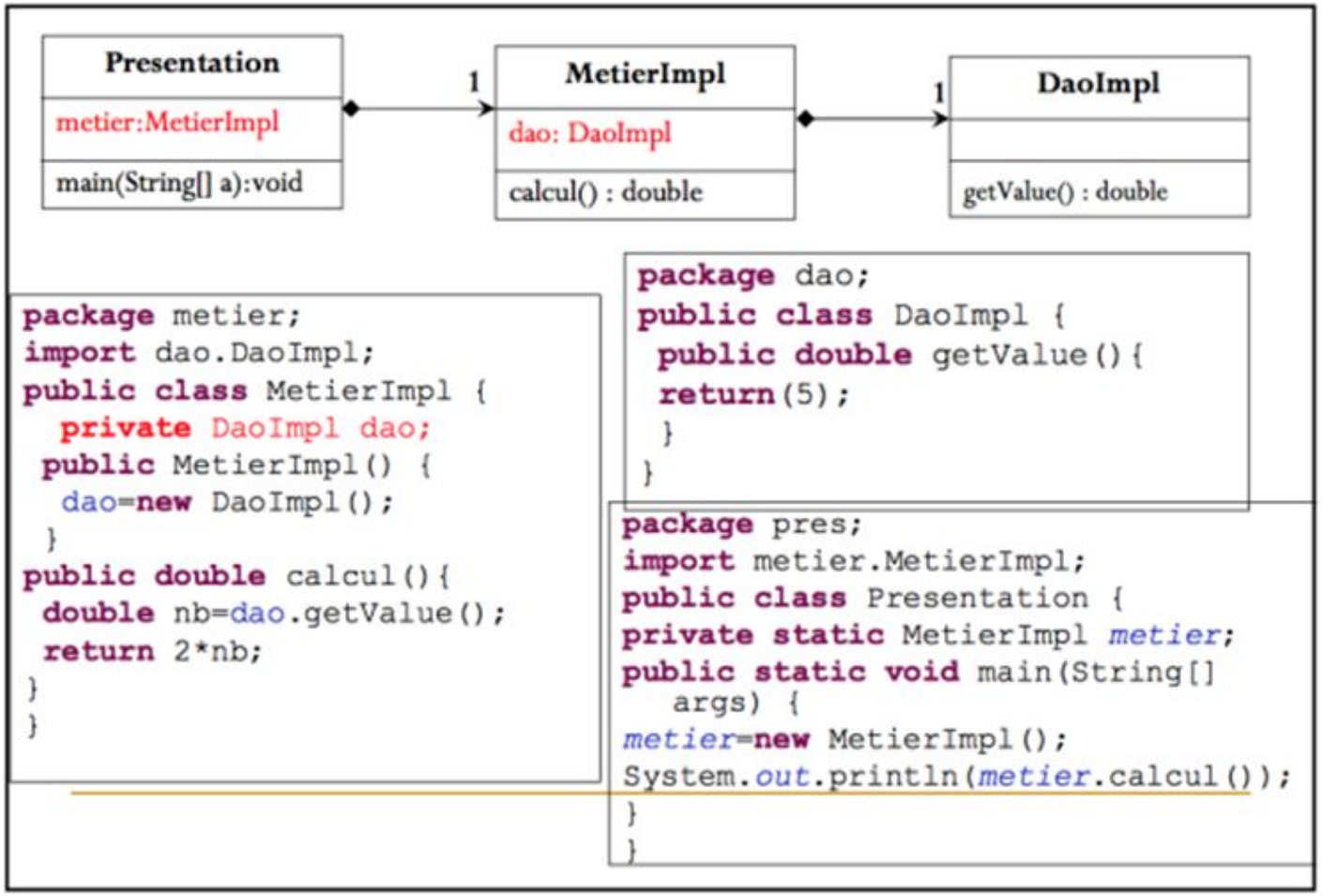


- Si une nouvelle version de la classe B (soit B2), est créée, on est obligé de modifier dans la classe A.

Modifier une classe implique:

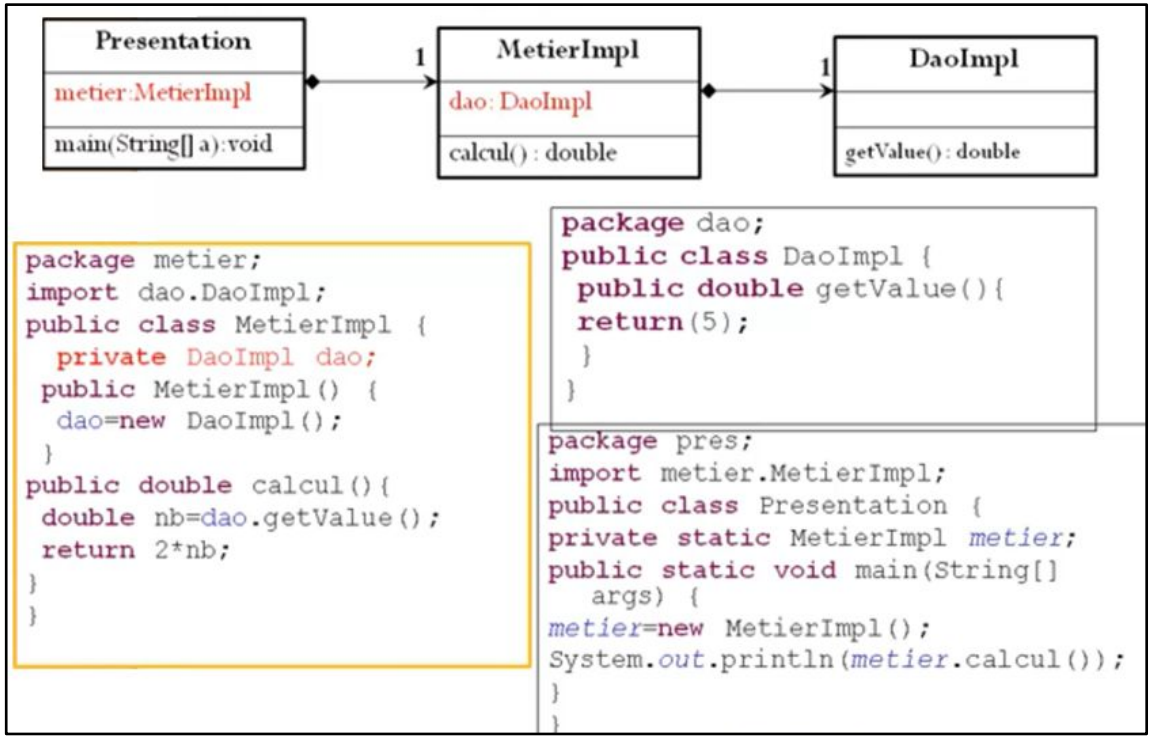
- Il faut disposer du code source.
- Il faut recompiler, déployer et distribuer la nouvelle application aux clients. Ce qui engendre des problèmes liés à la maintenance de l'application

Couplage fort : exemple



- De ce fait nous avons violé un principe SOLID **« une application doit être fermée à la modification et ouverte à l'extension »**

Couplage fort : Exemple




```
package metier;
import dao.DaoImpl;
public class MetierImpl {
  private DaoImpl dao;
 public MetierImpl() {
  dao=new DaoImpl();
 }
public double calcul(){
 double nb=dao.getValue();
 return 2*nb;
}
}
```

```
package dao;
public class DaoImpl {
 public double getValue(){
 return(5);
 }
}
```

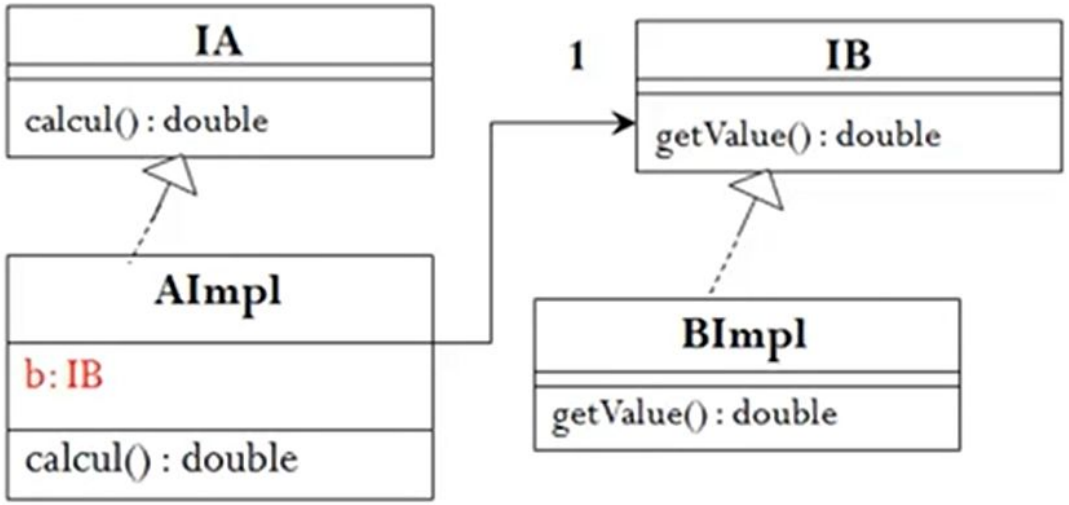
```
package pres;
import metier.MetierImpl;
public class Presentation {
private static MetierImpl metier;
public static void main(String[]
   args) {
metier=new MetierImpl();
System.out.println(metier.calcul());
}
}
```

Une nouvelle classe à la place de la classe DAOImpl

```
package dao;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class DaoImpl2 {

    public double getValue2() {
        Scanner scan;
        double value = 0;
        File file = new File("data.txt");
        try {
            scan = new Scanner(file);
            value = scan.nextDouble();

        } catch (FileNotFoundException e1) {
            e1.printStackTrace();
        }
        return value;
    }

}
```

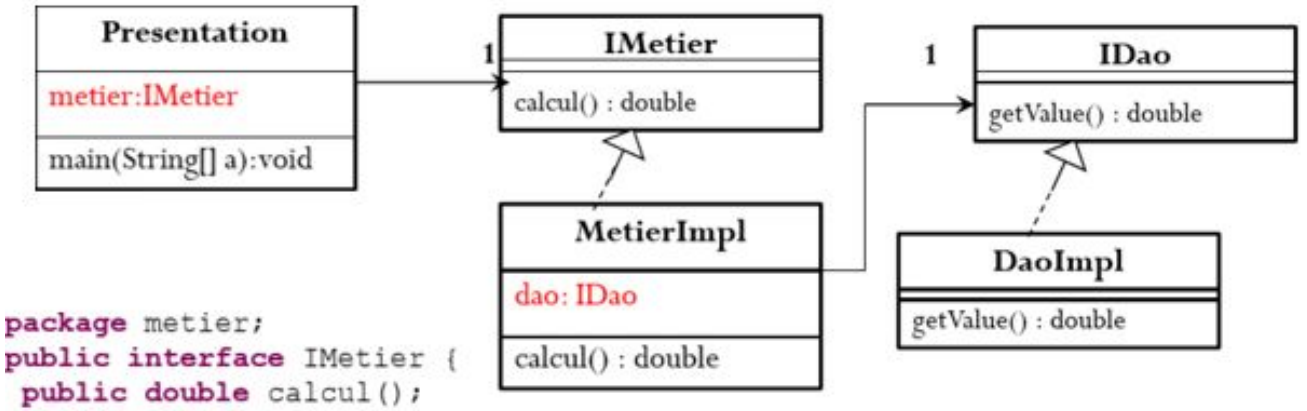
Couplage faible

- Le couplage faible permet de réduire les interdépendances entre les composants d'un système dans le but de réduire le risque que les changements dans un composant nécessitent des changements dans tout autre composant.



- Pour Utiliser le couplage faible, nous devons utiliser les interfaces…

Couplage faible




```
package metier;
public interface IMetier {
 public double calcul();
}
```

```
package metier;
import dao.IDao;
public class MetierImpl
   implements IMetier {
 private IDao dao;
public double calcul() {
double nb=dao.getValue();
 return 2*nb;
}
// Getters et Setters
```

```
package dao;
public interface IDao {
public double getValue();
}
```

```
package dao;
public class DaoImpl implements IDao {
public double getValue() {
 return 5;
}
}
```

- Avec le couplage faible, nous pourrons créer **des applications fermées à la modification et ouvertes a l'extension.**

Inversion de Contrôle

- L'IOC (**Inversion Of Control**) est un Patron d'architecture
- C'est le conteneur qui prend le contrôle du flot d'exécution et gère notamment la résolution des dépendances
- L'IOC permet de dynamiser la gestion de dépendance entre objets
- L'IOC facilite l'utilisation des composants
- L'IOC minimise l'instanciation statique d'objets (avec l'opérateur **new**)

Inversion de Contrôle

- L'exécution de l'application n'est plus sous le contrôle direct de l'application elle-même mais du Framework sous-jacent.
- Raccourci qui désigne un conteneur basé sur l'injection de dépendances.
- Dispose d'un noyau d'inversion de contrôle qui instancie et assemble les composants par injection de dépendances.

Inversion de Contrôle

- L'inversion de contrôle est une façon de concevoir l'architecture d'une application en se basant sur le mécanisme objet de l'injection de dépendance.

NB : L'IoC est liée étroitement au principe d'inversion de dépendance. Le principe d'inversion de dépendance est un des cinq principes fondamentaux de la conception objet identifiés par Robert C. Martin et résumés par l'acronyme SOLID. Le D de SOLID signifie Dependency inversion principle.

Injection de Dépendances

Remarque : L'IoC est en fait le principe qui est mis en application par la plupart des frameworks. On peut même dire que c'est principalement ce qui distingue un framework d'une bibliothèque.

- Dans notre cas, nos applications web seront contrôlées par le Framework Spring. Comment? :

- **L'injection de dépendances** est une mise en œuvre de l'IoC : les instances des dépendances vont être injectées automatiquement par le conteneur selon la configuration lors de l'instanciation d'un objet.

Injection de Dépendances

- **Principe** : un objet extérieur est chargé de composer l'application

– il crée les instances

– il les injecte dans les classes qui les utilisent

Définition 1

- **L'Injection de Dépendance** (ID) est un mécanisme simple à mettre en œuvre dans le cadre de la programmation objet et qui permet de diminuer le couplage entre deux ou plusieurs objets.

Définition 2

- **L'injection de dépendances** est un motif de conception qui propose un mécanisme pour fournir à un composant les dépendances dont il a besoin. C'est une forme particulière d'inversion de contrôle.

Bean – Définition et Portée

- C'est un composant Java spécifique avec un identifiant unique.
- Un bean est un simple objet (**POJO**: Plain Old Java Object).
- Plusieurs types de Bean (**scope**):

  - **singleton** : une instance par conteneur **(défaut)**
  - **prototype** : une instance par récupération du bean
  - **request** : une instance par requête http
  - **session** : une instance par session http
  - **global** : une instance par session http globale

Bean – Définition et Portée

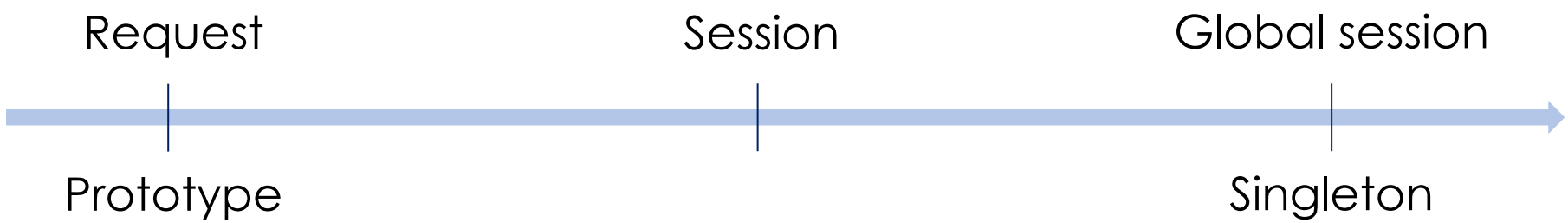
- Request
- Session
- Global session

} web

- Prototype
- Singleton

} Back -end

Ordre ⇧ selon la durée de vie du bean :



Implémentation de l'injection des dépendances

- **L'implémentation de l'injection de dépendances** proposée par Spring peut se faire de deux façons :

- **Injection par le constructeur :** les instances des dépendances sont fournies par le conteneur lorsque celui-ci invoque le constructeur de la classe pour créer une nouvelle instance

- **Injection par un setter :** le conteneur invoque le constructeur puis invoque les setters de l'instance pour fournir chaque instance des dépendances

- **Annotation (Autowiring)**

Injection des dépendances

- par **constructeur**

```
public class Movie {

    private Director director;
    private String name;
    private short year;

    public Movie(String name, Director director, short year) {
        this.director = director;
        this.name = name;
        this.year = year;
    }
```

```
public class Director {

    private String name;

    public Director(String name) {
        this.name = name;
    }
```

Résolution des arguments

- sans ambiguïté
- grâce au type
- selon l'index

Passage

- par valeur
- par référence

```
<bean id="director" class="sample.Director">
        <constructor-arg value="Allen"/>
</bean>

<bean id="movie" class="sample.Movie">
        <constructor-arg type="java.lang.String" value="Annie Hall"/>
        <constructor-arg ref="director"/>
        <constructor-arg index="2" value="1977"/>
</bean>
```

Injection des dépendances

- par **mutateur**

```xml
<bean id="director" class="sample.Director">
    <constructor-arg value="Allen"/>
</bean>

<bean id="movie" class="sample.Movie">
    <property name="name" value="Annie Hall"/>
    <property name="director" ref="director"/>
</bean>
```

```java
public class Movie {

    private Director director;
    private String name;

    public Movie() {
    }

    public void setDirector(Director director) {
        this.director = director;
    }

    public void setName(String name) {
        this.name = name;
    }
```

Injection des dépendances

- ***Le choix entre injection par constructeur ou par setter***

- L'avantage d'utiliser l'injection par constructeur est que l'instance obtenue est complètement initialisée suite à son instanciation.

- Si le nombre de dépendances est important ou si certaines sont optionnelles, le choix d'utiliser l'injection par constructeur n'est peut être pas judicieux.

- L'injection par constructeur peut aussi induire une dépendance circulaire : par exemple une classe A a une dépendance avec une instance de la classe B et la classe B a une dépendance avec une instance de la classe A. Dans ce cas, le conteneur lève une exception.

Injection des dépendances

- L'injection par setter peut permettre de modifier l'instance de la dépendance : cela n'est pas toujours souhaitable mais peut être utile dans certaines circonstances.
- Le conteneur Spring permet aussi de mixer ces deux modes d'injection : une partie par constructeur et une autre par setter.
- Le choix doit donc tenir compte du contexte d'utilisation.

Injection des dépendances - Annotations

- Pas du fichier ou de classe de configuration.
- Pour définir les beans, il faut utiliser les annotations.
- **@Component** c'est une annotation générique pour définir un bean.

Injection des dépendances - Annotations

- On peut utiliser les annotations ci-dessous pour définir les beans:

  ✔ **@Controller et @RestController** pour les beans de la couche controller.

  ✔ **@Service** pour les beans de la couche Service.

  ✔ **@Repository** pour les beans de la couche DAO/Persistence.

  @Controller,@RestController, @Service et @Repository héritent de la classe @Component

- On utilise **@Scope** pour définir la durée de vie du bean.

Injection des dépendances - Autowiring

- On utilise **@Autowired** pour injecter un bean dans un autre bean.
- L'autowiring laisse le conteneur déterminer automatiquement l'objet géré par le conteneur qui sera injecté comme dépendance d'un autre objet.
- L'utilisation de l'autowiring permet de simplifier grandement le fichier de configuration car il devient superflu de définir les valeurs des paramètres fournis au setter ou au constructeur pour injecter les dépendances.

@Autowired / @Inject

- Injection automatique des dépendances
- 2 annotations pour la même fonction

– @Autowired : annotation Spring

– @Inject : annotation JSR-330

@Autowired / @Inject

- Spring supporte les 2 annotations

– @Autowired dispose d'une propriété (required) que n'a pas l'annotation @Inject. Ceci est utile pour indiquer le caractère facultatif d'une injection.

- Résolution par type

– Se base sur le type de l'objet pour retrouver la dépendance à injecter.

@Autowired sur un attribut

- **Utilisation de @Autowired pour injecter helloWorld**

```
import org.springframework.beans.factory.annotation.Autowired;
public class HelloWorldCaller {

   @Autowired
   // Le nom de l'attribut importe peu, seul son type compte
   private HelloWorld helloWorldService;

   public void callHelloWorld(){
          helloWorldService.sayHello();
   }
}
```

@Autowired sur un mutateur

- **Utilisation de @Autowired pour injecter helloWorld**

```
import org.springframework.beans.factory.annotation.Autowired;
public class HelloWorldCaller {
    private HelloWorld helloWorldService;

    @Autowired
    public void setHelloWorldService(HelloWorld helloWorldService){
            // Le nom de la méthode setter importe peu. Seul le
        // type du paramètre compte
            this.helloWorldService=helloWorldService;
    }
    public void callHelloWorld(){
            helloWorldService.sayHello();
    }
}
```

Utilisation de @Autowired

- **Position de l'annotation**
  - devant un attribut
  - devant une méthode (« init-method », constructeur, setters)
- **Vérification de dépendance**
  - Par défaut @Autowired vérifie les dépendances
  - Une exception est générée si aucun bean n'est trouvé

Injection des dépendances - Annotations

- Dans la classe Java de configuration des beans. Chaque bean est annoté (Annotation décentralisée)

```
@Component
public class DependencyController {
   @Autowired
    private IDependencyService myService;
…
@Component
public class DependencyService implements IDependencyService {
   @Autowired
    private IDependencyRepository myRepository;
…
 @Configuration // Cherche les beans Spring dans le package :
    tn.esprit.esponline
 @ComponentScan("tn.esprit.esponline")
 public class Main {}
```

TP Injection des dépendances - Avec les Annotations

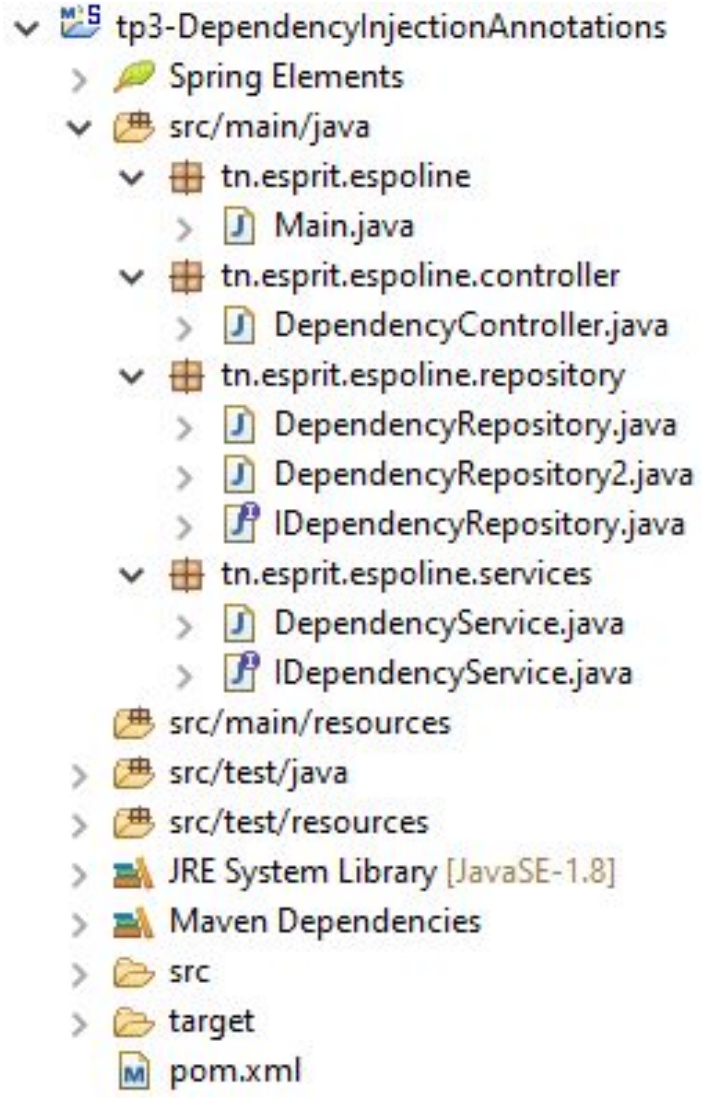
- Nous allons utiliser les annotations (@Controller, @Service, @Repository)

- Injecter l'interface Service dans le Controller **(@Autowired**)
- Injecter l'interface Repository dans le Service **(@Autowired**)

- Ajouter les annotations suivantes dans la classe Main :

```
@Configuration
@ComponentScan("tn.esprit.espoline")
public class Main {}
```

TP Injection des dépendances - Avec les Annotations



TP Injection des dépendances - Avec les Annotations

```java
package tn.esprit.espoline.services;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import tn.esprit.espoline.repository.IDependencyRepository;
@Service
public class DependencyService implements IDependencyService {
    @Autowired
    private IDependencyRepository myRepository;

    public IDependencyRepository getMyRepository() {
     return myRepository;
    }
    public void setMyRepository(IDependencyRepository myRepository) {
     this.myRepository = myRepository;
    }
    public List<String> getCoursesList() {
     return myRepository.getCoursesList();
    }
}
```

@Qualifier, @Primary

- Si on a deux classes qui implémente la même interface.

```java
public interface UserService {
    public void test();
}
```

```java
@Service("UserServiceImpl")
public class UserServiceImpl implements UserService {

    @Autowired
    UserDAO userDAO;

    @Override
    public void test() {
        System.out.println("test from UserServiceImpl");

    }

}
```

```java
@Service("userServiceImpl2")
public class UserServiceImpl2 implements UserService {

    @Override
    public void test() {
        System.out.println("test from UserServiceImpl2");

    }
}
```

```java
@Controller
public class UserControllerImpl {

    @Autowired
    UserService userService;

}
```

Erreur : No qualifying bean of type 'tn.esprit.esponline.service.UserService' available: expected single matching bean but found 2: userServiceImpl,userServiceImpl2

@Qualifier, @Primary

- On utilise l'annotation @Qualifier pour éliminer le problème du bean à injecter.

```java
public interface UserService {
    public void test();
}
```

```java
@Service("UserServiceImpl")
public class UserServiceImpl implements UserService {

    @Autowired
    UserDAO userDAO;

    @Override
    public void test() {
        System.out.println("test from UserServiceImpl");

    }

}
```

```java
@Service("userServiceImpl2")
public class UserServiceImpl2 implements UserService {

    @Override
    public void test() {
        System.out.println("test from UserServiceImpl2");

    }
}
```

```java
@Controller
public class UserControllerImpl {

    @Autowired
    @Qualifier("UserServiceImpl")
    UserService userService;

}
```

@Qualifier, @Primary

- Aussi, en utilisant l'annotation @Primary, on peut éliminer le problème du bean à injecter.

```java
public interface UserService {
    public void test();
}
```

```java
@Service("UserServiceImpl")
public class UserServiceImpl implements UserService {

    @Autowired
    UserDAO userDAO;

    @Override
    public void test() {
        System.out.println("test from UserServiceImpl");

    }

}
```

```java
@Primary
@Service("userServiceImpl2")
public class UserServiceImpl2 implements UserService {

    @Override
    public void test() {
        System.out.println("test from UserServiceImpl2");

    }
}
```

```java
@Controller
public class UserControllerImpl {

    @Autowired
    UserService userService;

}
```

Les Trois Configurations

	Java	Annotation	XML
Avantages	- Conf centralisée dans un seul endroit. - Typage fort.	- Bien pour un développement rapide. - La classe contient déjà sa configuration	
Inconvénients	- Par rapport au annotation, un code de configuration supplémentaire à écrire.	- Configuration éparpillée. - Conf et code dans un même endroit.	Obsolète

Si vous avez des questions, n'hésitez pas à nous contacter :

Département Informatique
UP ASI

(Architectures des Systèmes d'Information)

Bureau E204

SPRING – Injection de Dépendances