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 ?

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

- 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

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…)

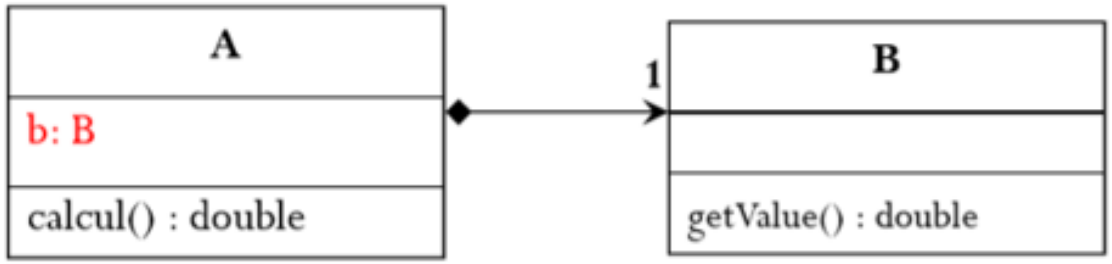
 Couplage : degré de dépendance entre objets

  - 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 !!

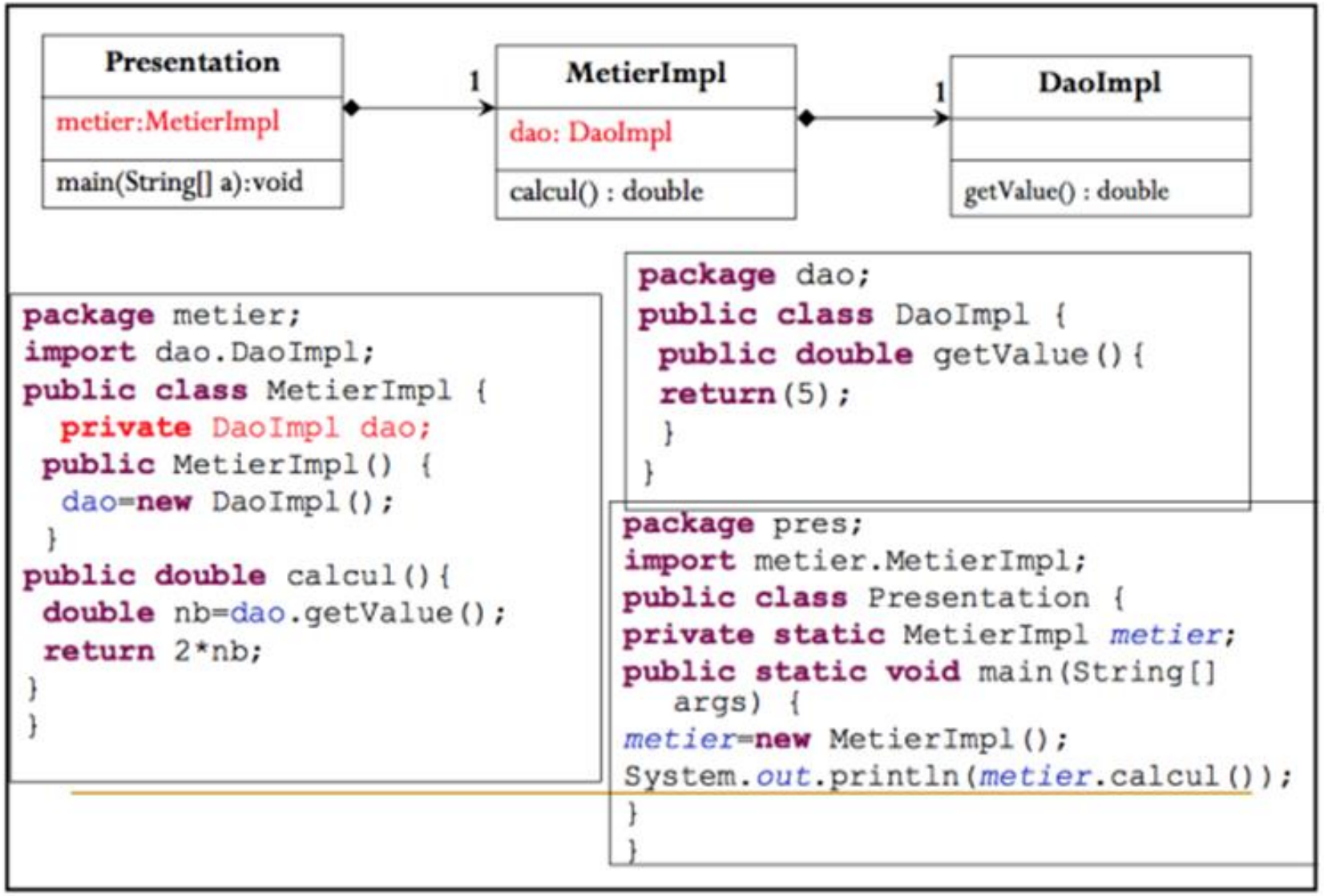


- Si une nouvelle version de la classe B (soit B2) est créée, on est obligé de modifier dans la classe A.

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

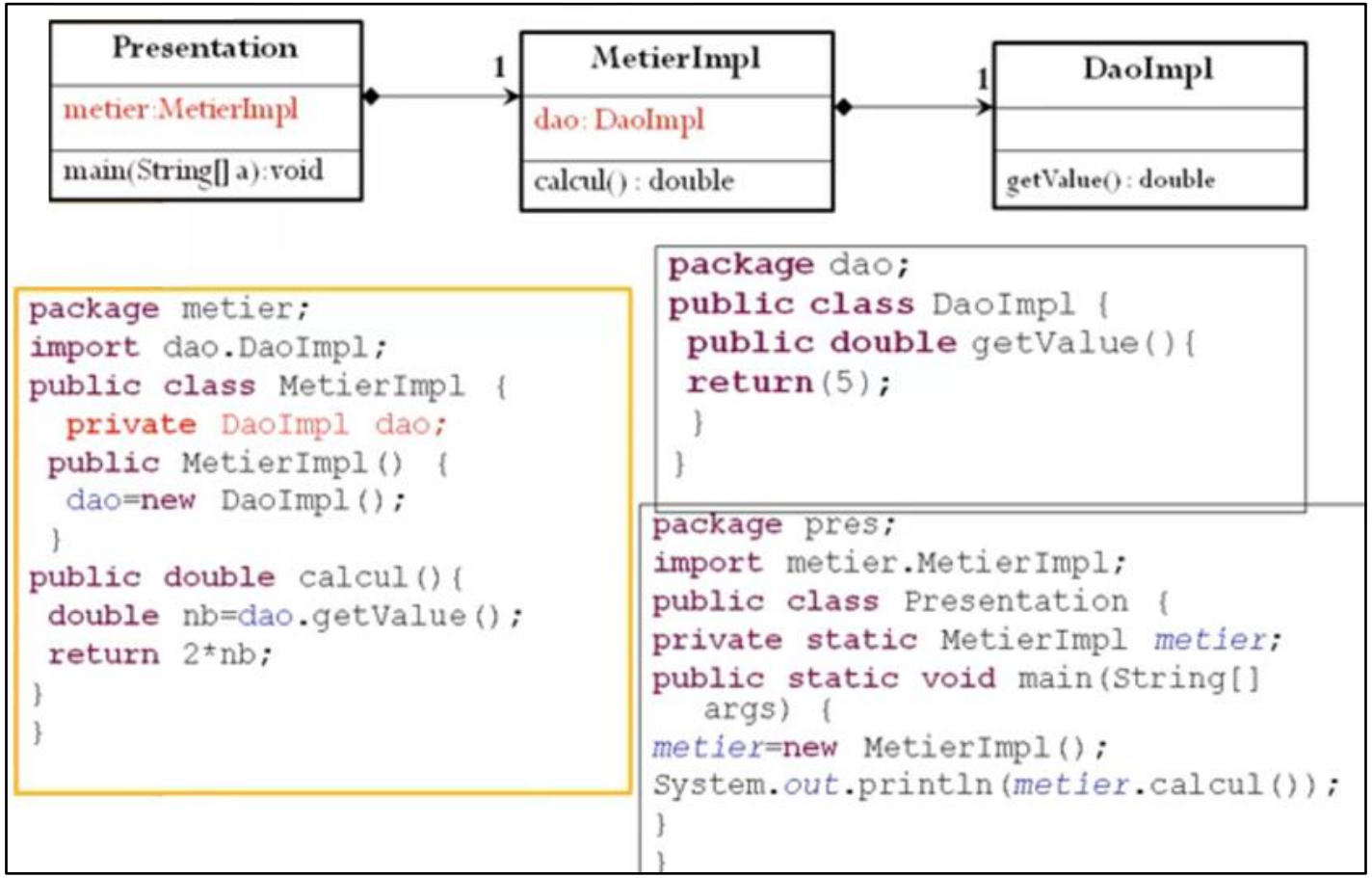



```
package metier;
import dao.DaoImpl;
public class MetierImpl {
  private DaoImpl dao;
 public MetierImpl() {
  dao=new DaoImpl();
 }
public double calcul(){
 double nb=dao.getValue();
 return 2*nb;
}
}
```

```
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());
}
}
```

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());
}
}
```

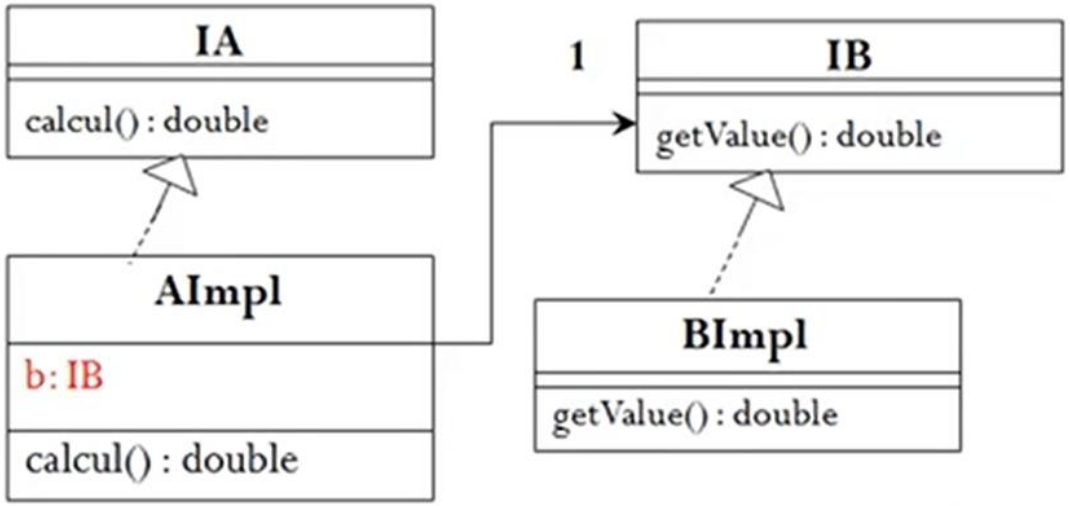
Une nouvelle classe à la place de la classe DAOImpl

```
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;
    }

}
```

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

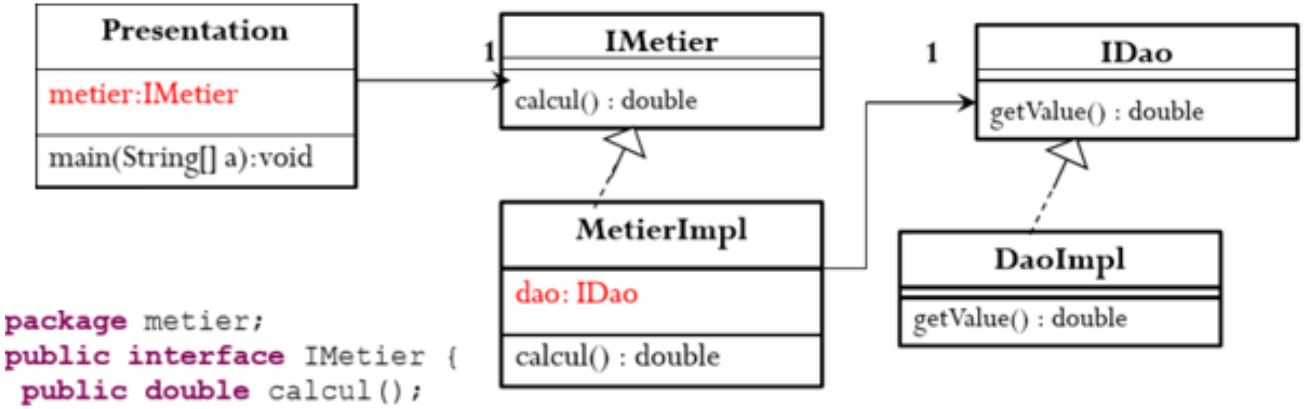
Couplage faible

• 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 à l'extension.**

Inversion de Contrôle

L'IOC (**Inversion Of Control**) est un patron d'architecture :

- Permettant de dynamiser la gestion de dépendance entre objets
- Facilitant l'utilisation des composants
- Minimisant 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.

- Si l'application s'exécute dans un Framework, celui est représenté par un conteneur à l'exécution :
  - celui-ci prend le contrôle du flot d'exécution et gère notamment la résolution des dépendances
  - l'application s'insère dans le cadre qu'il fixe
  - Il y a donc **inversion de contrôle**

Injection de Dépendances

- 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 (Inversion of Control) :
- Les instances des dépendances vont être injectées automatiquement par le conteneur selon la configuration lors de l'instanciation d'un objet.

Injection de Dépendances

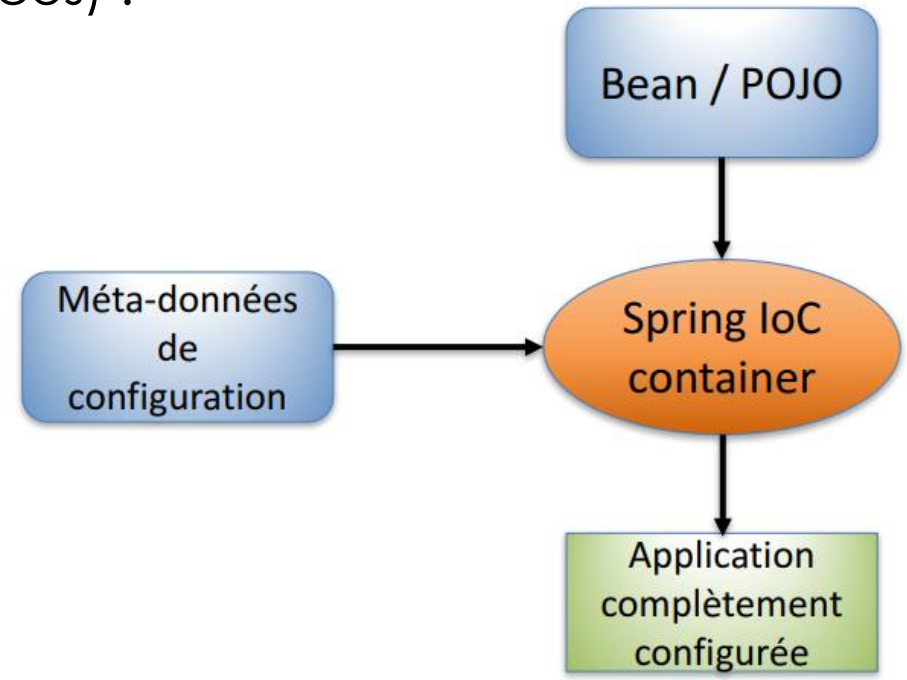
- L'**Injection de Dépendance** (ID) est une méthode pour instancier des objets et créer les dépendances nécessaires entre elles, sans avoir besoin de coder cela nous même.

- Ceci permet de réduire le code relatif aux dépendances. Ce qui permet une lisibilité du code, et permet de gagner en temps de développement.

Bean – Définition

- C'est un composant Java spécifique avec un identifiant unique.
- Un bean est un simple objet (**POJO**: Plain Old Java Object).
- Spring se base sur **trois façons** différentes pour configurer **l'injection de dépendances** et la création des beans (instances) :
  - ✓ **XML** (obsolète) et **Java** (obsolète)
  - ✓ **Annotations** (le plus utilisé)

  Les méta-données permettent à Spring
  - d'instancier les beans
  - de les configurer
  - de les assembler



Bean – Définition par Annotation

- Pas de fichier ou de classe de configuration.
- Pour définir les beans, il faut utiliser les annotations.

@Component est un stéréotype générique de Spring Framework puisqu'il indique simplement que cette classe doit être utilisée pour instancier un bean.

Bean – Définition par Annotations

- Les autres stéréotypes fournis par le Spring Framework sont :

  @Service Un composant qui remplit une fonctionnalité centrale dans l'architecture d'une application.

  @Repository Un composant qui représente un mécanisme permettant de stocker et de rechercher une collection d'objets (DAO/Persistance).

  @Controller et @RestController Un composant qui joue le rôle d'un contrôleur dans une architecture MVC pour une application Web.

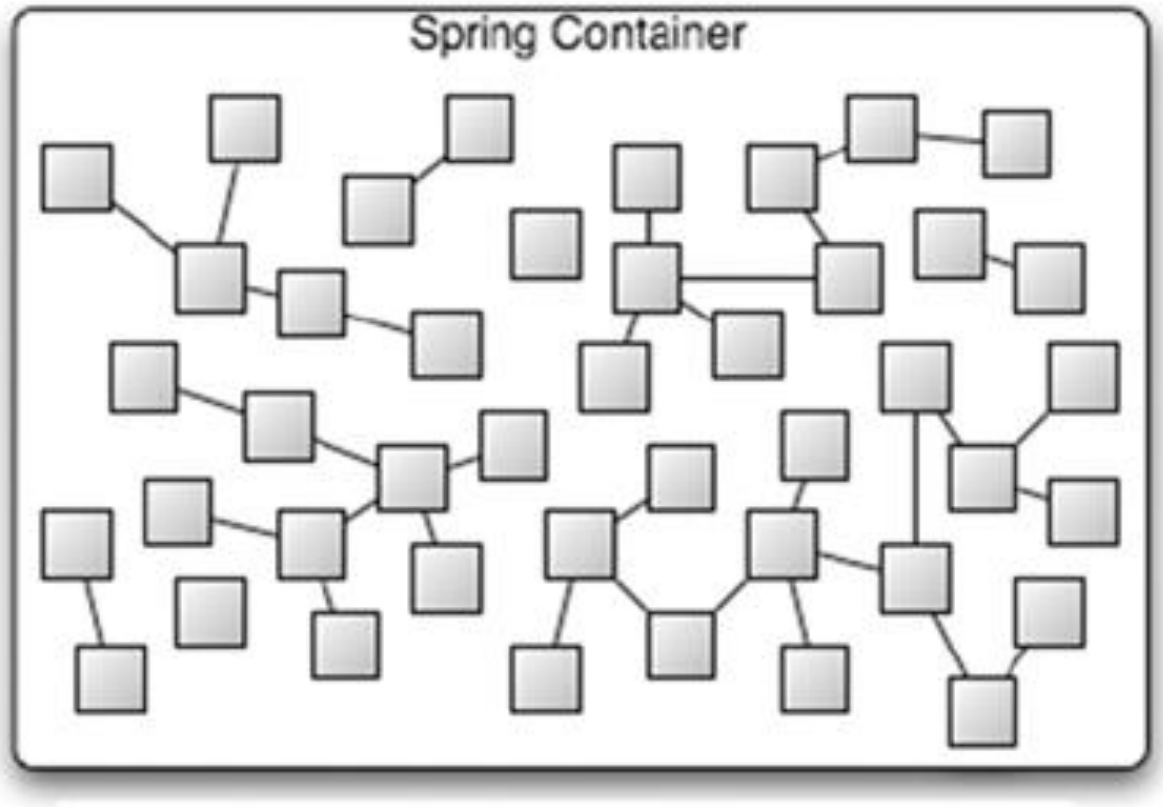
  - @Repository, @Service , @Controller et @RestController héritent de @Component

Bean –Portée

- On utilise **@Scope** pour définir la durée de vie du bean.
- Plusieurs portées de Bean (**scope**):
  - **singleton** : Une seule instance du Bean crée pour chaque Conteneur IoC Spring, et référencée à chaque invocation), **(scope par défaut)**
  - **prototype** : une nouvelle instance créée à chaque appel du Bean
  - **request** : une instance par requête http (Contexte Web uniquement)
  - **session** : une instance par session http
  - **global** : une instance par session http globale (valable pour les portlets : ensemble d'applications déployées sur un même conteneur web)

Spring IOC Container

Dans une application Spring, les objets sont crées, liés ensembles et communiquent dans le Spring IOC Container.



Injection des dépendances - Autowiring

- Il est possible de procéder à une injection de dépendances lorsqu'on utilise le mécanisme d'annotations sous Spring. Pour cela, Spring passe par le concept d'**autowiring**.

- L'autowiring est utilisée par Spring pour injecter les dépendances où il le faut à l'aide des annotations :

  - ✓ **@Autowired** : annotation Spring
  - ✓ **@Inject** : annotation JSR-330

Injection des dépendances - Autowiring

- Il existe **3 façons** pour effectuer de l'autowiring sous Spring :

  - ✓ Une injection par constructeur
  - ✓ Une injection par mutateur (setter)
  - ✓ Une injection par attribut / propriété

Injection des dépendances - par attribut / propriété

```java
@Service
public class StudentServicesImp implements IStudentServices {

    @Autowired
    private IStudentRepository studentRepository;

    public StudentServicesImp() {
    }
}
```

Injection des dépendances - par mutateur

```java
@Service
public class StudentServicesImp implements IStudentServices {

    private IStudentRepository studentRepository;

    public StudentServicesImp() {
    }

    @Autowired
    public void setStudentRepository(IStudentRepository studentRepository) {
        this.studentRepository = studentRepository;
    }
}
```

Injection des dépendances - par constructeur

```java
@Service
public class StudentServicesImp implements IStudentServices {

    private final IStudentRepository studentRepository;

    @Autowired
    public StudentServicesImp(IStudentRepository studentRepository) {
        this.studentRepository = studentRepository;
    }
}
```

- Remarque à partir de Spring 4.3 : l'ajout de @Autowired est facultatif pour faire une injection depuis le constructeur

Injection des dépendances - par constructeur & Lombok

Avec Lombok, il est possible de générer un constructeur pour :

- ✓ tous les champs de la classe (avec @AllArgsConstructor)
- ✓ ou tous les champs de type final (avec @RequiredArgsConstructor).
- ✓ Si on a besoin d'un constructeur vide, on peut ajouter l'annotation @NoArgsConstructor

```
@Service
@RequiredArgsConstructor
public class StudentServicesImp implements IStudentServices {

    private final IStudentRepository studentRepository;

    @Override
    public Student addOrUpdateStudent(Student student) {
        return studentRepository.save(student);
    }
}
```

Cas des implémentations multiples

- Supposons que UserService possède **plusieurs implémentations possibles** :

```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;

}
```

- Spring ne saura plus la bonne implémentation à injecter, et nous donnera une exception de type `NoUniqueBeanDefinitionException`

Cas des implémentations multiples

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

- Pour préciser une **spécificité dans l'injection**, il faut utiliser l'annotation **@Qualifier**, tout en précisant l'identifiant du bean :

```
@Controller
public class UserControllerImpl {

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

}
```

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

    @Autowired
    UserDAO userDAO;

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

    }
}
```

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

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

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

    }
}
```

Cas des implémentations multiples – Injection par constructeur

- L'utilisation de **@Qualifier** est particulière pour l'injection par constructeur, dans le sens où l'appel de l'annotation doit se faire **à l'intérieur des arguments** de la méthode :

```
@RestController
public class StudentRestController {

    private final IStudentServices studentServices;

    public StudentRestController(@Qualifier("studentServicesImp") IStudentServices studentServices) {
        this.studentServices = studentServices;
    }
}
```

Cas des implémentations multiples : @Primary

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

```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;

}
```

TP Injection des dépendances - Avec les Annotations

- Nous allons appliquer les annotations (@RestController, @Service, @Repository) dans l'étude de cas Kaddem.

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

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