SPRING – Injection de Dépendances

Bureau E204
UP ASI

Plan du Cours

- Dépendance entre objets ?
- C'est quoi le couplage ?
- Couplage fort
- Couplage faible
- Inversion de Control **IoC**
- Injection de Dépendances **ID**
- **Bean** : Configuration, Stéréotypes, Portées
- Implémentation de l'injection des dépendances
- @Qualifier et @Primary
- TP 4 : Courses (UML et Spring)

Dépendance entre objets ?

Comment on peut dire que 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

  ➔ Le couplage fort

- Avec **Spring**, nous nous basons sur:
  - Le couplage faible
  - Injection de dépendance

C'est quoi le couplage ?

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

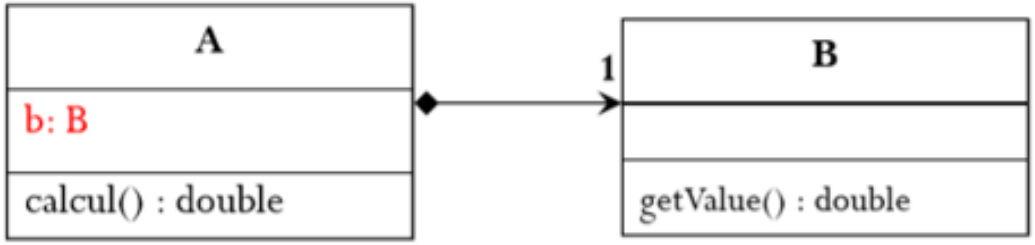
→ Degré de dépendance entre objets

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

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

Couplage fort

- La classe A ne peut fonctionner que à la présence de la classe B.



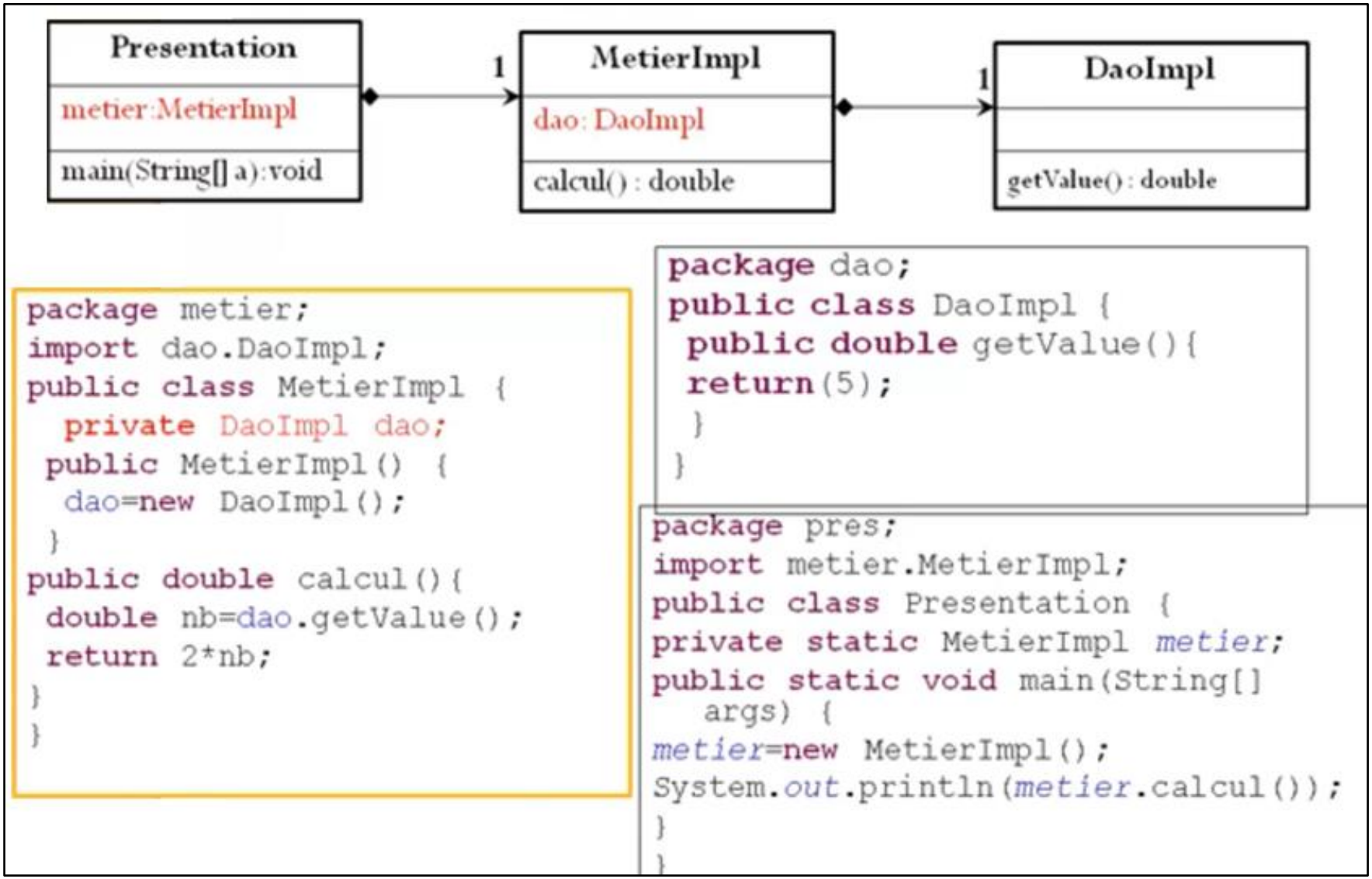
- Si une nouvelle version de la classe B (soit B2), est créée, on est obligé

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 de la maintenance de l'application

➔ Avec le **couplage fort**, nous pourrons créer des applications ouvertes à la modification et ouverte à l'extension

Couplage fort : exemple

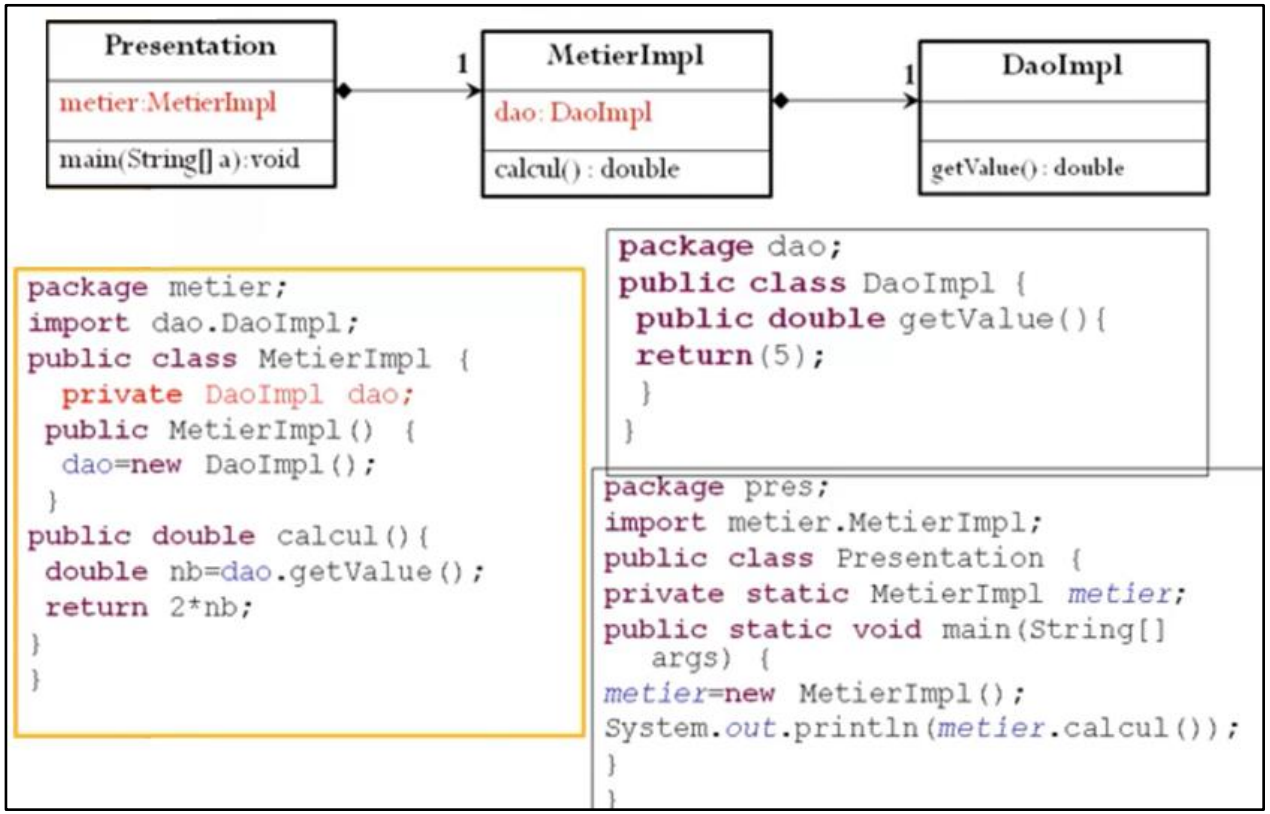



```
package metier;
import dao.DaoImpl;
public class MetierImpl {
  private DaoImpl dao;
 public MetierImpl() {
  dao=new DaoImpl();
 }
public double calcul(){
 double nb=dao.getValue();
 return 2*nb;
}
}
```

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

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);
 }
}
```

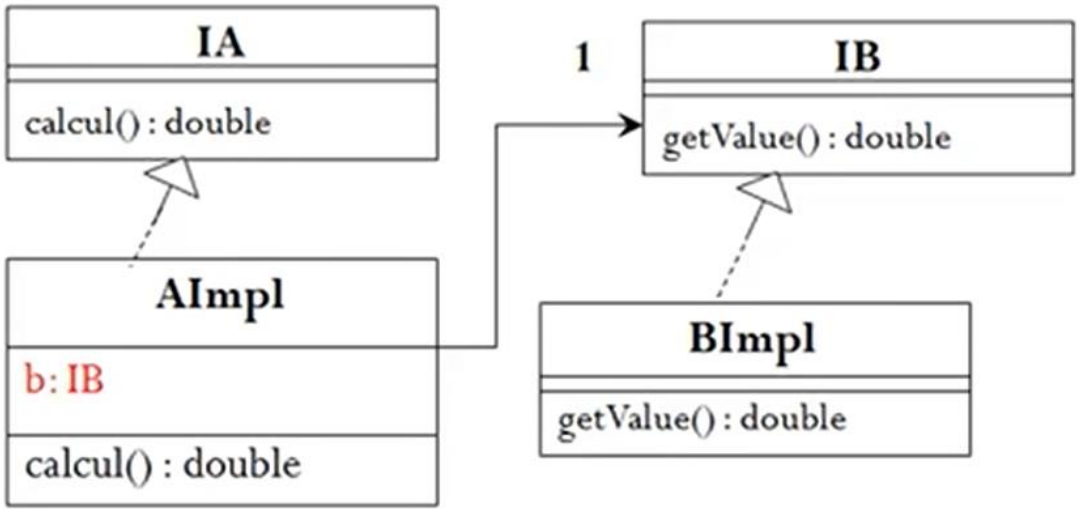
```
package pres;
import metier.MetierImpl;
public class Presentation {
private static MetierImpl metier;
public static void main(String[]
   args) {
metier=new MetierImpl();
System.out.println(metier.calcul());
}
}
```

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;
    }

}
```

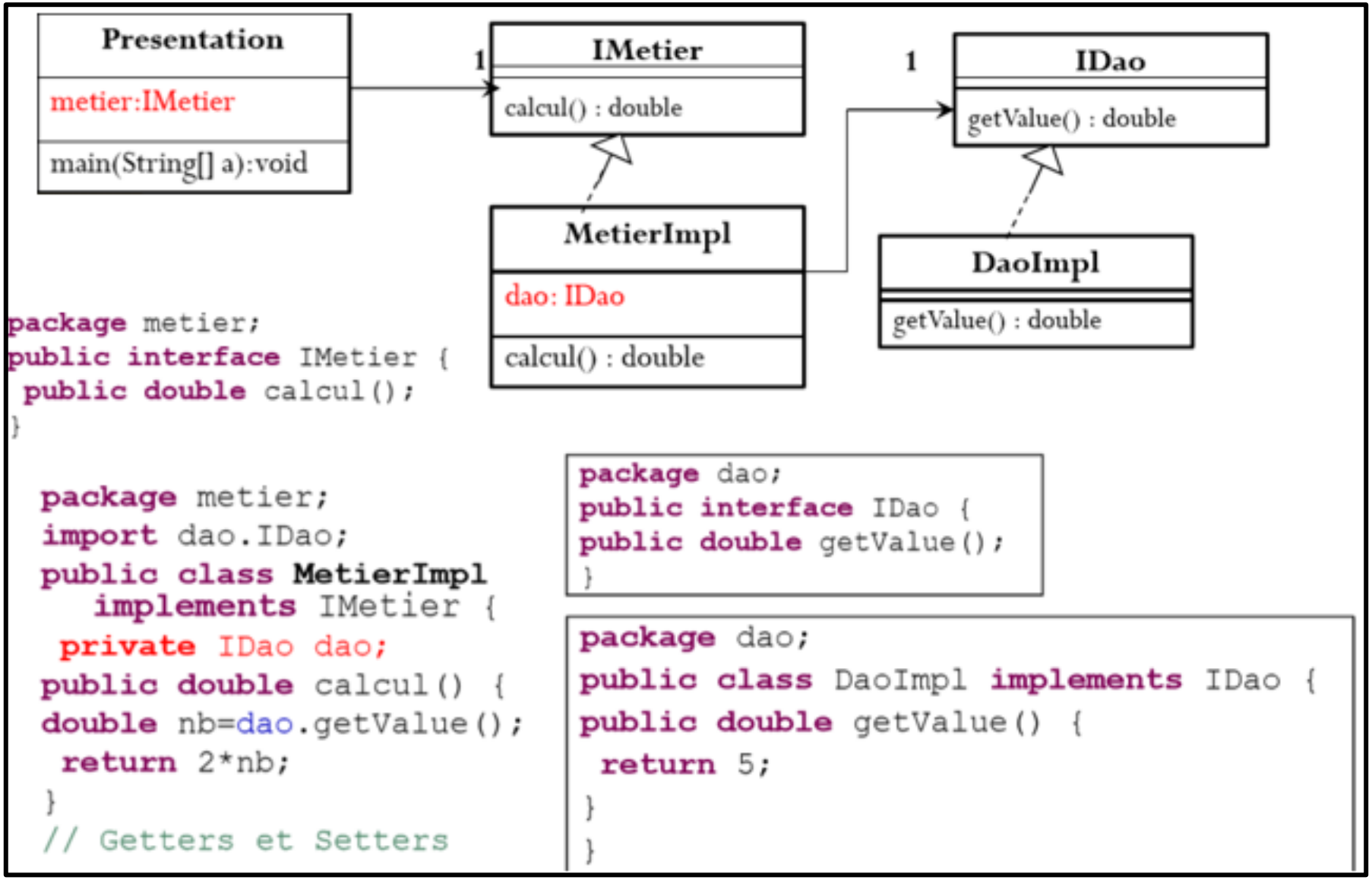
Couplage faible

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

Couplage faible : Exemple




```
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;
}
}
```

Inversion de Contrôle

- L'IOC (**Inversion Of Control**) est un **Patron de conception** qui :
  - Permet de dynamiser la gestion de dépendance entre objets.
  - Facilite l'utilisation des composants.
  - Minimalise l'instanciation statique d'objets (avec l'opérateur `new`).
- L'IOC fonctionne selon le principe que l'exécution de l'application n'est plus sous le contrôle direct de l'application elle-même mais du Framework sous-jacent.
- Dans notre cas, nos applications web seront contrôlées par le Framework Spring par l'**Injection de Dépendance** qui est la forme principale de l'IoC.
- Comment Spring va prendre le contrôle de l'application ?

  ➔ Il va créer les instances de tous les beans Spring en mémoire.

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 (Instanciation).

  ➔Avoir une lisibilité facile du code.

  ➔Gagner en temps de développement.

  ➔Faciliter les tests.
- Elle utilise les **beans** pour faciliter la mise en place des dépendances par l'injection automatique des objets.

Bean – Définition et Portée

- C'est un composant Java spécifique avec un identifiant unique c'est-à-dire c'est une classe Java avec des getters et des setters.
- Plusieurs types de Bean (scope):
  - **Singleton** : Une seule instance du Bean crée pour chaque Conteneur IoC Spring, et référencée à chaque invocation), **scope par défaut**.
  - **Prototype** : Nouvelle instance créée à chaque appel du Bean
  - **Request** : (Contexte Web uniquement) Nouvelle instance créée pour chaque requête HTTP.
  - **Session** : une instance du bean par session HTTP.
  - **Global-session** : une instance du bean par session globale (valable pour les portlets : ensemble d'applications déployées sur un même conteneur web).
- On utilise **@Scope** pour définir la durée de vie du bean.

Bean – Configuration

- Pour définir les beans, il faut utiliser **@Component:** C'est l'annotation générique.
- On peut utiliser les annotations ci-dessous pour personnaliser la définition des 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

Implémentation de l'injection des dépendances

- **L'implémentation de l'injection de dépendances** proposée par Spring peut se faire de trois 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 le mutateur (Setter) :** Le conteneur invoque le constructeur puis invoque les setters de l'instance pour fournir chaque instance des dépendances.
  - ✓ **Injection par les annotations**

Implémentation de l'ID - Injection par le constructeur

- La version la plus courte et optimisée.
- Créer un constructeur qui prend en paramètres toutes les dépendances requises.

```
@Service
public class HelloService {

    private HelloRepository repository;

    public HelloService(HelloRepository repository) {
        this.repository = repository;
    }

}
```

→ En arrière-plan, quand Spring démarrera, il découvrira le HelloService grâce à l'annotation **@Service** et saura qu'il doit être initialisé. Il découvrira ensuite le constructeur dont il devra fournir les paramètres demandés. Il les cherchera dans son contexte.

Implémentation de l'ID - Injection par le setter

- C'est presque la même configuration de l'injection par constructeur

```
@Service
public class HelloService {

    private HelloRepository repository;

    public void setRepository(HelloRepository repository)
    {
        this.repository=repository
    }

}
```

Implémentation de l'ID - Injection par les annotations

- Pour faire l'injection d'un bean dans un autre bean avec les annotations, il faut mettre **@Autowired** ou **@Inject** au dessus de l'instanciation.

```
@Service
public class HelloService {

    @Autowired
    // ou @Inject
    private HelloRepository repository;

}
```

- L'annotation **@Inject** fait partie de Java CDI qui a été introduit dans Java 6, tandis que l'annotation **@Autowired** fait partie du cadre Spring. Les deux annotations remplissent le même objectif, par conséquent, nous pouvons utiliser n'importe laquelle d'entre elles dans notre application.

Implémentation de l'ID - Injection par les annotations: @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

Implémentation de l'ID - Injection par les annotations: @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;

}
```

Implémentation de l'ID - Injection par les annotations: @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;

}
```

Implémentation de l'ID - Injection par les annotations: @Qualifier, @Primary

- On peut spécifier le bean à injecter sans utiliser les annotations. Il suffit juste de nommer l'instance du bean par le bean souhaité.

```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 UserServiceImpl;

}
```

TP 4: Courses (UML et Spring)

- Importer le projet tpDependencyInjectionTODO depuis le classroom.
- Ajouter les annotations adéquates pour charger les classes nécessaires au bon fonctionnement de notre logiciel dans l'IOC Container.
- Paramétrer les classes de façon à afficher la liste des cours liées à la matière UML.
- Paramétrer les classes de façon à afficher la liste des cours liées à la matière Spring.
- Utiliser la notion de @Primary pour donner la préférence à l'affichage de la liste des cours de la matière Spring.
- Utiliser l'injection de dépendances par le nom pour donner la préférence à l'affichage de la liste des cours de la matière Spring.

SPRING – Injection de Dépendances

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

Département Informatique
UP ASI
Bureau E204

SPRING – Injection de Dépendances