Inversion de contrôle ou Injection de dépendances

IOC ou DI

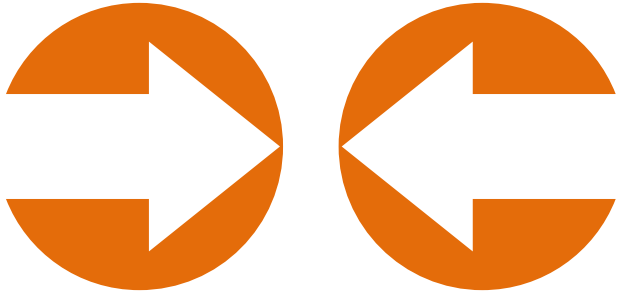
Pr. Sara Belattar

Inversion de contrôle ou Injection de dépendances

Rappels de quelque principes de conception

1
Une application qui n’évolue pas meurt.

2
Une application doit être fermée à la modification et ouverte à l’extension.



3
Une application doit s’adapter aux changements

4
Efforcez-vous à coupler faiblement vos classes.

5
programmer une interface et non une implémentation

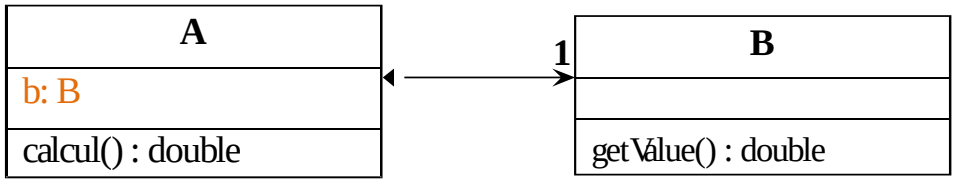
6
Etc…

Couplage Fort et Couplage Faible

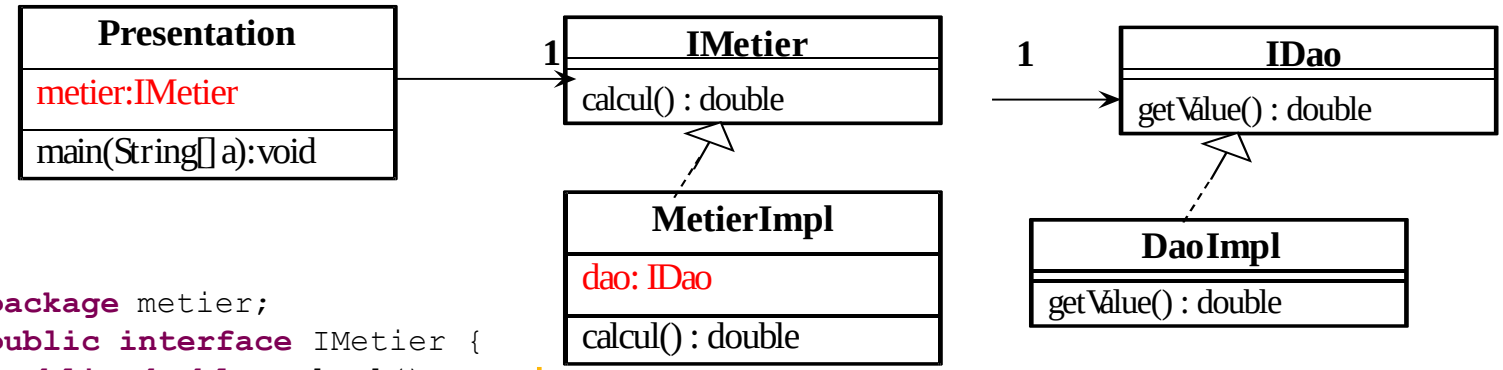
CF, CF

Couplage fort

- Quand une classe A est lié à une classe B, on dit que la classe A est fortement couplée à la classe B.
- La classe A ne peut fonctionner qu'en présence de la classe B.
- Si une nouvelle version de la classe B (soit B2), est crée, 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 un cauchemar au niveau de la maintenance de l'application



Exemple de coupage 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;
}
```

Injection des dépendances

- Injection par instanciation statique :

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

Spring IOC

L'objectif principal du **conteneur IOC (Inversion of Control)** de **Spring**, souvent appelé conteneur Spring, est de gérer la création, la configuration et la gestion des composants d'une application. Il met en œuvre le principe de l'inversion de contrôle, qui consiste à inverser la responsabilité de la gestion des dépendances et de la création d'objets à partir du développeur vers le conteneur Spring.

Voici les principaux objectifs de Spring IOC :

- **Gestion des dépendances** : Spring IOC permet de gérer les dépendances entre les composants de votre application. Vous définissez les dépendances dans un fichier de configuration **(XML ou annotations)** ou via des classes Java, et le conteneur Spring se charge de les injecter correctement.
- **Configuration externe** : L'IOC permet de configurer l'application de manière externe, ce qui signifie que vous pouvez changer le comportement de l'application sans avoir à modifier le code source.

Spring IOC

- **Injection de dépendances** : Spring IOC prend en charge l'injection de dépendances, ce qui signifie qu'il injecte automatiquement les dépendances requises dans les composants de l'application. Cela réduit la nécessité pour le développeur de créer manuellement et de gérer des dépendances.

- **Amélioration de la flexibilité** : L'IOC rend l'application plus flexible en permettant de configurer facilement les composants, de remplacer les implémentations de dépendances, et d'ajouter ou de supprimer des fonctionnalités sans modifier le code source existant.

- **Gestion du cycle de vie des objets** : Le conteneur Spring gère le cycle de vie des objets, tels que la création, l'initialisation, et la destruction, ce qui garantit que les ressources sont correctement libérées.

TP1

IOC, DI, Crud, Postman

Les annotations utilisés dans le TP 1

- **@RestController :** en **Spring Framework** est utilisée pour créer des contrôleurs spécifiquement destinés à la création de services **Web RESTful.** Les contrôleurs annotés avec **@RestController** sont principaleme nt responsables de la gestion des **requêtes HTTP** en vue de fournir des données structurées (par exemple, **JSON ou XML**) en réponse aux requêtes.

  Le principal rôle de **@RestController** est de simplifier la création de services W**eb RESTful** en **facilitant la génération de réponses au format de données structurées.**

- **@RequestMapping** :en Spring Framework est utilisée pour mapper des méthodes de contrôleur (contrôleu r Spring MVC**) à des URL spécifiques ou à des méthodes HTTP**, définissant ainsi comment les requêtes HTTP sont gérées par votre application.

Les annotations utilisés dans le TP 1

- **@Service :** est une annotation plus spécifique que **@Component**. Elle est généralement utilisée pour anno ter les classes qui fournissent des services métier au sein de l'application.
- **@Override :** est utilisée en Java pour indiquer qu'une méthode dans une classe dérive d'une classe parent e ou implémente une méthode définie dans une interface, conformément au mécanisme de l'héritage et de la polymorphie
- **@Autowired :** est utilisée en Spring Framework pour **effectuer l'injection de dépendances automatique**. Son rôle principal est de **simplifier le câblage des composants (beans)** dans une application Spring en permettant à Spring de rechercher et d'injecter automatiquement les dépendances requises dans une classe ou un composant
- **@GetMapping :** en Spring Framework est utilisée pour définir un point de terminaison (endpoint) de contrôleur qui **gère les requêtes HTTP de type GET vers une URL spécifique.**
- **@PathVariable :** permettre la récupération de données dynamiques à partir de l'URL de la requête, ce qui est couramment utilisé dans les applications web pour traiter des URL contenant des paramètres variables.

Les annotations utilisés dans le TP 1

```
@GetMapping(value = "/products/{id}")
public Product getProductById(@PathVariable(value = "id") Long productId) {
        return service.getById(productId);
}
```

L'annotation **@PathVariable("id")** est utilisée pour extraire la valeur du segment d'URL "id" et l'associer au paramètre productId. Ainsi, lorsque vous accédez à "/products/123.

- **@PostMapping :** en Spring Framework est utilisée pour définir un point de terminaison (endpoint) de contr ôleur qui gère les requêtes HTTP de type POST vers une URL spécifique. **Cela signifie que la méthode**
- **est utilisée pour créer ou mettre à jour des données sur le serveur, généralement en envoyant des d onnées dans le corps de la requête.**

Les annotations utilisés dans le TP 1

- **@RequestBody :** Son rôle principal est de permettre la conversion automatique des données JSON, XML ou d'autres formats de données du corps de la requête en un objet Java, ce qui facilite la manipulation des données envoyées avec la requête.

- **@PutMapping :** en Spring Framework est utilisée pour définir un point de terminaison (endpoint) de contrôl eur qui gère les requêtes HTTP de type PUT vers une URL spécifique. Les requêtes **HTTP PUT** sont coura mment utilisées pour mettre à jour ou remplacer complètement des ressources existantes sur le serveur.

- **@DeleteMapping :** Les requêtes HTTP DELETE sont couramment utilisées pour supprimer des ressource s ou des données existantes sur le serveur.

- **@Valid** est utilisée pour annoter des paramètres de méthode dans un contrôleur. **Les données reçues avec la requête HTTP sont automatiquement validées** en fonction des contraintes de validation définies dans les classes de modèle associées.

.

Les annotations utilisés dans le TP 1

- **@Valid** :cette annotation est d'appliquer des contraintes de validation aux données reçues pour s'assurer qu'elles respectent les règles définies dans les classes de modèle (entités) de votre application. Elle est couramment utilisée pour garantir l'intégrité des données et la sécurité des applications.

.

Spring IOC

- **Spring IOC ou Spring Core est le noyau du Framework Spring :**

  - **Est basé sur le Design Pattern IOC;**
  - **Se charge de l'instanciation de tous les objets de l'application et de la résolution des dépendan ces entre eux.**

- **Deux API très importantes dans ce module :**

  - **org.springframework.beans;**
  - **org.springframework.context.**

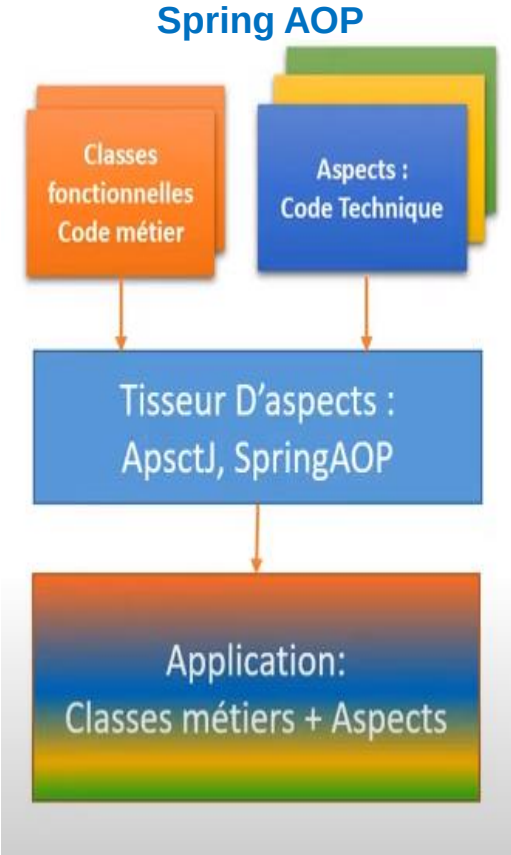
Spring IOC

- **Offrent les bases pour le Design Pattern IOC.**

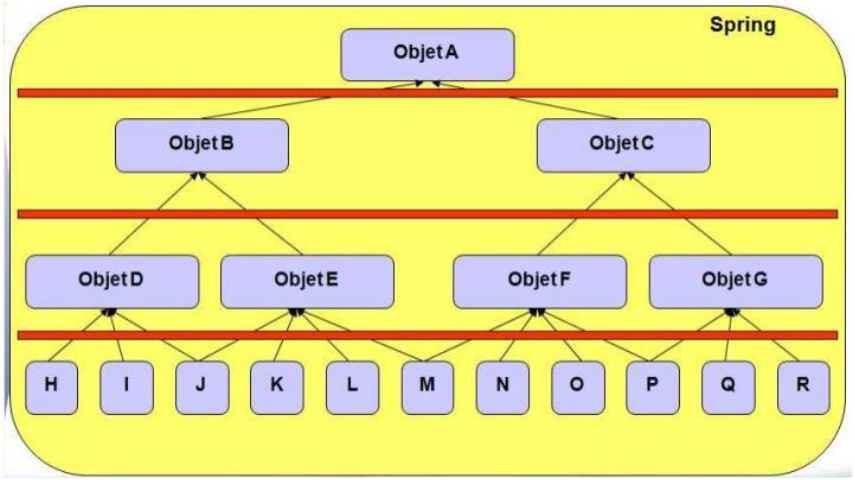
- **BeanFatory (org.springframework.beans.factory.BeanFactory) permet de configurer les Beans (dans un fichier XML ou via les annotations) et les gérer (instanciation, gestion de la dépendance).**

- **ApplicationContext (org.springframework.context.ApplicationContext) ajoute des fonctionnalités avancées au BeanFactory (facilite l'intégration avec Spring AOP (**Aspect- Oriented Programming**), …).**

Spring AOP est un outil puissant pour gérer des préoccupations **transversales** (**sécurité**, **transaction**,…) dans une application de manière modulaire. Il améliore la maintenance du code en facilitant l'ajout, la modification ou la suppression de comportements transversaux **sans affecter le code métier principal.**



Spring IOC



Spring va permettre à chaque couche de s'abstraire de sa ou ses couches inférieures (injection de dépendance) :

- **Le code de l'application est beaucoup plus lisible.**
- **Le maintien de l'application est facilité.**
- **Les tests unitaires sont simplifiés.**
- **Spring gère les dépendances entre les beans dans un fichier XML ou via les annotations**

IOC

- **IOC (Inversion Of Control), est une Design Pattern qui corrige le problème de couplage fort.**
- **IOC permet de ne pas utiliser les classes concrètes.**
- **Dans l'IOC, la relation entre les classes se fait par interfaces.**
- **L'IOC permet d'injecter les objets moyennant trois méthodes**
  - **Injection par le modificateur;**
  - **Injection par constructeur;**
  - **Injection par Factory.**

IOC

❖ **Injection par le modificateur :** Pour utiliser l'injection par Modificateur (par le setter), ajouter la méthode ***set Dao(IDao dao)*** suivante au niveau de la classe ***ServiceImpl***:

```
// Injection par modificateur
@Autowired
public void setDao(IDao dao) {
   this.dao = dao;
}
```

❖ L'annotation **@Autowired** appliquée sur le setter permet d'injecter le Bean implémentant l'interface **IDao.** Dans ce cas, Spring IOC passe une instance de la classe implémentant l'interface **IDao** en paramètre de la méthode **setDao(IDao dao)** : c'est l'injection par Modificateur.

IOC

❖ **Injection par Constructeur:**

```
//@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
@Service
public class ServiceImpl implements IService {
  private final IDao dao;

  public ServiceImpl(@Qualifier("dao1") IDao dao) {
    this.dao = dao;
  }
```

- Remarquez que nous n'avons pas annoté le constructeur par ***@Autowired***. Car, comme expliqué ci-dessus, puisque la classe ***ServiceIml*** contient un seul constructeur, l'annotation ***@Autowired*** **devient facultative.**
- Remarquez que Spring injectera le Bean moyennant cette surcharge du constructeur. Il s'agit ici de **l'injection par Constructeur**.
- Nous avons utilisé **@Qualifier** puisque dans cet exemple, nous avons deux Bean qui implémentent la même interface **IDao.**

IOC

❖ **Injection par factory:**

```
@Service
public class ServiceImpl implements IService {
    4 usages
    @Autowired
    @Qualifier("dao1")
    private IDao dao;
```

```
@Configuration
@ComponentScan(basePackages = "ma.formations.ioc")
public class MainApplication {
  // Injection par factory
  @Bean
  @Qualifier("dao1")
   public IDao getDao() {
      return new DaoImpl1();
   }
}
```

- L'annotation **@Bean** s'applique sur les **méthodes**.
- **L'instance** retournée par la méthode annotée par **@Bean** est ajoutée au niveau du conteneur de **Spring**.
- Remarquez que la méthode **getDao()** ci-dessus joue le rôle d'une fabrique. C'est le principe de l'injection par **Factory**.

Les annotations utilisés dans le TP 2

- **@Configuration:** L'annotation ***@Configuration*** est une classe Java Config. Elle est équivalente à un Fichier XML. Cette classe devrait contenir les Bean à utiliser dans l'application.

- ***@ComponentScan*****:** Permet à Spring de gérer toutes les classes se trouvant au niveau du package ***(ma.formations.ioc)*** et également au niveau des sous packages de ce dernier.

❖ **Junit** est un Framework *open source* de la communauté Jakarta Apache.
❖ **Junit** permet de réaliser les tests unitaires très facilement.
❖ **Junit** offre plusieurs annotations : **@Test, @TestBeforeAll, @TestAfterAll**, etc.…

- la méthode ***Assertions.assertAll*** qui permet d'exécuter plusieurs vérifications.