# REST / Spring Boot

## -L’inversion de contrôle-


technocité
Centre de compétence



WAVENET
Intelligent IT Solutions

# Objectifs du chapitre

- Les objectifs de ce chapitre sont:
- Comprendre les besoins à l'origine de l'inversion de contrôle.
- Prise de connaissance de Java EE.
- Exemple basique de l'injection des dépendances avec Java EE.

# Prérequis - Classes de bas et de haut niveau

- Une classe est dite de bas niveau quand son cycle de vie et ses features sont gérées par une autre classe, ainsi qualifiée de haut niveau.
- Une classe de haut niveau gère / organise plusieurs classes de bas niveaux entre elles.
- Une classe de bas niveau ne dépend pas forcément d'une de haut niveau.

```
public class Maestro {
    private List<Musician> musicians;

    public void startMusic() {
        musicians.forEach(musician -> musician.playInstrument());
    }
}
```

```
public class Musician {
    public void playInstrument(){
        //...
    }
}
```

technocité
Centre de compétence



WAVENET
Intelligent IT Solutions

# Prérequis – Les principes SOLID

- **S**: Single Responsibility Principle
- **O**: Open Closed ( Les logiciels doivent être ouverts à l'extension mais fermés à la modification )
- **L**: Liskov's Substitution Principle ( Les sous classes doivent complètement se substituer à leur classes mères)
- **I**: Interface Segregation( le client ne doit pas se voir contraint d'implémenter des méthodes dont il n'a pas besoin)
- **D**: Dependency Inversion Principle (Les modules de haut niveau ne doivent pas dépendre des modules de plus bas niveau. Les deux doivent dépendre d'abstractions.)

# Cheminement de l'IoC

- Nous avons une classe qui permet de lire des livres. La source peut être différente en fonction de l'environnement dans lequel le logiciel est exécuté.
- Problèmes: cela ne respecte pas le S, O et D du principe SOLID.

```
public class Bookshelf {

    private XMLBookImporter xmlBookImporter;

    public Bookshelf(String source) {
        this.xmlBookImporter = new XMLBookImporter(source);
    }

    public List<Disk> readDisks() {
        // Lire les livres
    }
}
```

technocité
Centre de compétence



WAVENET
Intelligent IT Solutions

# Cheminement de l'IoC

- **S**: Notre étagère s'occupe d'importer des livres mais lit aussi des disques. Alors qu'elle ne devrait gérer qu'un type d'élément, des livres.
- **O**: Le programme n'est pas permissif à une nouvelle source de livre à l'avenir. Il faudrait pour ce faire créer une interface IBookImporter héritée par XMLBookImporter et ScannerBookImporter. Ainsi la classe pourra se faire injecter soit l'un soit l'autre.
- **D**: Le fonctionnement de la classe de haut niveau *BookShelf* dépend de l'import de XMLBookImporter qui est une classe de bas niveau.

# Cheminement de l'IoC

- **Si on applique les bons principes au niveau humain, voici ce que cela donne:**
- Un cuisinier qui entre au matin dans sa cuisine, ne doit pas se soucier de construire sa casserole. Ni demander à quelqu'un de la lui construire chaque matin. Il faut que la casserole soit déjà là avant d'entrer dans la cuisine. Il se moque de savoir si aujourd'hui c'est une casserole de la marque X au de lieu de Y; elle peut être plus avancée tant qu'elle cuit les aliments pareillement.
- **S**: Single Responsibility Principle
- **O**: Open Closed ( Les logiciels doivent être ouverts à l'extension mais fermés à la modification )
- **D**: Dependency Inversion Principle (Les modules de haut niveau ne doivent pas dépendre des modules de plus bas niveau. Les deux doivent dépendre d'abstractions.)

technocité
Centre de compétence



WAVENET
Intelligent IT Solutions

# L'IoC
## Autre exemple

- Autre point de vue pour comprendre le S,O,D:

- Notre cuisinier entre dans sa cuisine le matin et malheureusement ses pommes de terres sont encore
  dans le terreau et ses casseroles sont encore sous l'état de tas de ferraille.

- Premièrement Le cuisinier a donc plusieurs responsabilités (S):
  - Cuisiner
  - Cultiver des pommes de terre
  - Construire la casserole.

- Il ne doit avoir qu'un job, cuisiner.

technocité
Centre de compétence



WAVENET
Intelligent IT Solutions

# L’IoC

## Autre exemple suite

- Deuxièmement un bon cuisinier doit pouvoir s'améliorer d'après les critiques de ses maîtres tout en étant capable de conserver ses propres acquis (O).
- Troisièmement il a du matériel de moindre importance que lui (donc bas niveau), mais il ne devrait jamais avoir à se soucier si sa casserole et ses PDT existent quand il arrive sur son lieu de travail (D).

# Cheminement de l'IoC

- Revenons à l'exemple codé et voici une proposition de solution: ajout d'abstraction via une interface:

```
public class Bookshelf {

    private IBookImporter importer;

    public Bookshelf(String source) {
        this.importer = new XMLBookImporter(source);
    }
}
```

- Le O est résolu mais pas le S, on s'occupe encore d'instancier l'Importer.

# Cheminement de l'IoC

- Autre proposition de solution avec le pattern Singleton:

```
public class Bookshelf {

    private IBookImporter importer;

    public Bookshelf(String source) {
        this.importer = DefaultBookImporter.instance();
    }
}

public class DefaultBookImporter {
    private static IBookImporter importer = new XMLBookImporter();

    public static IBookImport instance() {
        return importer;
    }
}
```

- Les principes S et O sont résolus. Néanmoins le pattern Singleton a pour but de ne permettre qu'une instanciation. Cela peut poser des problèmes si l'application permet de créer plusieurs bibliothèques provenant de sources différentes.

# Cheminement de l'IoC

- Notre classe *BookShelf* est maintenant bien découplée du *XMLImporter*. Le S et le O des principes SOLID sont respectés.  Mais toujours pas le D.
- Suggestion d'amélioration: le pattern Factory afin d'encapsuler le *XMLImporter* au lieu de l'importer.
- ...Comme ci-dessous?

```
public class Bookshelf {

    private IBookImporter importer;
    private BookImporterFactory factory = new BookImporterFactory();

    public Bookshelf(String source) {
        this.importer = factory.getImporter(sourceName);
    }
}
```

technocité
Centre de compétence



WAVENET
Intelligent IT Solutions

# Cheminement de l'IoC

- Surtout pas. On retomberait sur le même problème qu'au début avec le non respect des principes, car on instancie maintenant une Factory.
- Implémentation correcte via une méthode static:

```
public class Bookshelf {

    private IBookImporter importer;

    public Bookshelf(String source) {
        this.importer = BookImporterFactory.getImporter(sourceName);
    }
}
public class BookImporterFactory {
  public static IBookImporter getImporter(String sourceName) {
    IBookImporter importer;
    if (sourceName.endsWith(".pdf")) {
      importer = new PDFBookImporter(sourceName);
    } else {
      importer = new XMLBookImporter(sourceName);
    }
    return importer;
  }
}
```

technocité
Centre de compétence



WAVENET
Intelligent IT Solutions

# Cheminement de l'IoC

- La fabrique impose au développeur de gérer lui-même toutes les dépendances entre les objets.
- Donc il faudrait que la classe *BookShelf* n'aie plus à appeler une classe intermédiaire à chaque fois qu'elle a besoin d'un *IBookImporter.* De sorte à ce qu'il soit déjà là quand le *BookShelf* est créé.
- Toutefois il est recommandé de favoriser le pattern Factory pour les POJO. (Plain Old Java Object)
- Quelle solution ultime nous réserve donc Java ?...

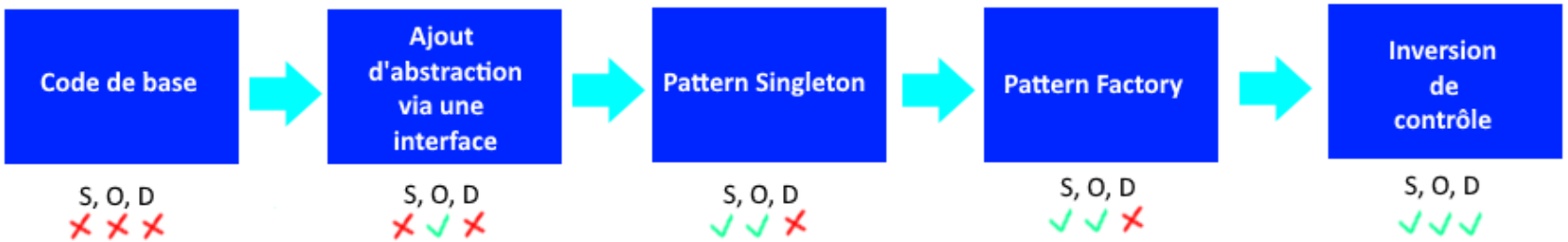
# Cheminement de l'IoC


technocité
Centre de compétence



WAVENET
Intelligent IT Solutions

# Cheminement de l'IoC




Raisons de non respect des principes SOLID:

**S:** Instancier une classe de bas niveau c'est une responsabilité supplémentaire

**O:** Pas d'extensibilité

**D:** Pour exister notre classe de haut niveau se soucie de l'existance de notre classe de bas niveau

technocité
Centre de compétence



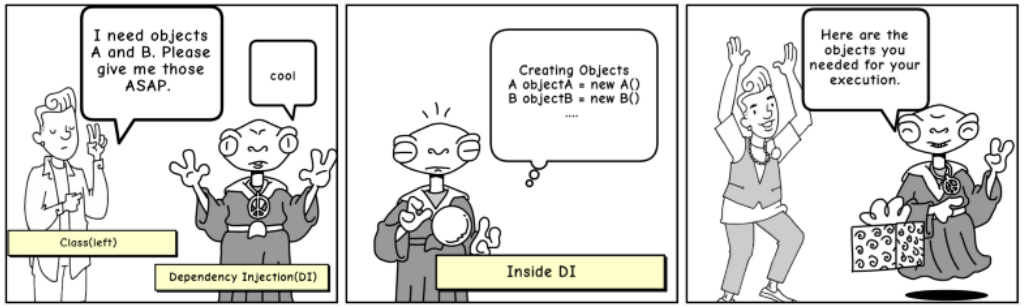
WAVENET
Intelligent IT Solutions

# L'IoC

- L'Inversion de contrôle est un contrat.
- Exemple: Pour passer l'aspirateur, la femme de ménage sait qu'elle doit jouer sur le type de brosse en fonction de la texture de la surface. Si elle change d'aspirateur, elle veut toujours qu'il possède une brosse pour parquet et une pour tapis. Peu importe la marque de l'aspirateur et sa technologie employée pour aspirer les saletés.
- Le contrat ici est avec son patron: il doit y avoir 2 brosses, une pour parquet et une pour tapis.
- La manière de travailler de notre femme de ménage ne dépend ainsi plus de l'aspirateur, quelque soit celui qu'on lui fournit, elle sait qu'il y aura deux brosses car le patron l'a écrit sur le contrat.

# L'IoC

- L'inversion de contrôle (IoC) permet donc de respecter le principe d'inversion des dépendances (DIP).
- Les deux manières de faire de l'IoC sont :
  - Le pattern Factory (notamment pour les POJO)
  - L'injection des dépendances (DI) pour les services




- Pour ce faire Java Standard Edition ne suffit plus, il faut passer à Java Enterprise Edition.

technocité
Centre de compétence



WAVENET
Intelligent IT Solutions

# Java EE

- Java Enterprise Edition (1999) est dédiée à la réalisation d'applications pour entreprises. J2EE est basé sur J2SE (Java 2 Standard Edition) qui contient les API de base de Java. Depuis sa version 5, J2EE est renommée Java EE (Enterprise Edition).
- J2EE est une plate-forme fortement orientée serveur pour le développement et l'exécution d'applications distribuées.
- Typiquement une application distribuée est constituée d’un client, un serveur d’application et un serveur de base de données.

# Java EE

- Depuis 2018 Java EE est devenu Jakarta EE !


- Pour faire de l'injection des dépendances en Java la librairie javax.inject est utilisée, intégrée à la spécification Java EE 6, sa version 1.0 est sortie en décembre 2009.
- Utiliser la librairie javax.inject seule n'a pas de sens, il est mieux de passer complètement à JavaEE et sa suite de librairies.
- Voyons ce que donnerait l'injection des dépendances avec notre exemple *BookShelf.*

# Injection des dépendances – Avec Java EE

```java
@Named
public class XMLBookImporter implements IBookImporter {

    @Override
    public void imports() {
        //...
    }
}
```

```java
public class Bookshelf {

    @Inject
    public Bookshelf(IBookImporter importer) {
        importer.imports();
    }

    public List<Book> readBooks() {
        //lire les livres
    }
}
```

- C'est ça l'injection de dépendances. Chaque bean reçoit ce dont il a besoin, sans forcément connaître l'implémentation qui correspond, mais comme il a ce qu'il attend – le contrat est respecté – tout se passe bien.
- **Named** : Déclaration d'un managed bean.
- **Inject**: on précise que que les arguments du constructeur soient injectés par Java.

# Injection des dépendances – Avec Java EE

- Les annotations nous amènent à une **approche déclarative** au lieu d'une approche séquentielle d'instanciations et d'injections.
- Chaque classe entrant en jeu dans l'injection de dépendances se déclare dans **l'IoC container** via les annotations qui sont scannées *at runtime.* Le container crée les objets, les lie entre eux, les configure et les suit dans leur cycle de vie.
- Pour chaque objet, on indique les dépendances à injecter (ex.: *IBookImporter)*
- L'IoC container se charge d'instancier et d'injecter les dépendances afin de produire la grappe d'objets pleinement configurée.

# Injection des dépendances – Avec Java EE

- Anciennement les beans étaient gérés via XML.

```xml
<beans>
    <bean id="bookShelf" class="org.example.demo.BookShelf">
        <property name="bookImporter" ref="bookImporter" />
    </bean>

    <bean id="bookImporter" class="org.example.demo.XMLBookImporter" />
</beans>
```

technocité
Centre de compétence



WAVENET
Intelligent IT Solutions

# Injection des dépendances – Avec Java EE

- De nouveaux mots à ne pas confondre font leur entrées :
- **POJO**: Plain Old Java Object: une classe qui ne déclare que des variables et mais aucune méthode (excepté pour accéder à ces variables).
- **Managed Bean**: c'est le nom donné aux classes qui sont injectées ou se font injecter d'autres classes. Simplement nommés Bean chez le framework Spring.
- **Java Bean**: Souvent confondu avec le POJO. Un Java Bean est un POJO avec des règlementations de code nommées JavaBeans Conventions. Par exemple un Java Bean doit :
  - Implémenter l'interface Serializable
  - Avoir un constructeur par défaut
  - Toutes les propriétés doivent avoir un getter / setter

# Injection des dépendances – Avec Java EE

- Un Java Bean est un POJO mais dire l'inverse est faux.

- **Enterprise Java Bean (EJB):** Un Managed Bean pour développer des applications distribuées robustes et sécurisées. C'est une specification fournie originellement par Sun Microsystems Un EJB se configure par annotations et exploite l'injection des dépendances pour définir les rôles aux classes.

- Exemple :

```
@Stateless(mappedName="st1")
public class AdderImpl implements AdderImplRemote {
  public int add(int a,int b){
      return a+b;
  }
}
```

technocité
Centre de compétence



WAVENET
Intelligent IT Solutions