

MODULE Spring

Injection de dépendances

Injection de dépendances

Plan du module Spring - Injection de dépendances

- La problématique
- IoC container, beans et métadonnées
- Instanciation de beans
- Configuration de beans
- Portée d'un bean
- Customisation du cycle de vie des beans
- Utilisation des annotations
- Configuration « java-based »
- Et caetera

Injection de dépendances

La problématique

Injection de dépendances

Un objet logiciel utilise généralement d'autres objets :

- un système de pilotage de voiture contrôle ses éléments : un châssis, un ou plusieurs moteur(s), un volant, des roues, etc.
- une boutique propose des produits, gère des clients et des fournisseurs, etc.
- un établissement d'enseignement regroupe des étudiants, des enseignants et du personnel administratif, dispose de bâtiments, propose des modules d'enseignement, etc.

Injection de dépendances

Exemple

```
class PiloteVoiture {  
    private Moteur moteur;  
    private Chassis chassis;  
    public PiloteVoiture() {  
        ...  
    }  
    ...  
}
```

Injection de dépendances

La création des objets utilisés peut être réalisée par l'objet englobant.

```
class PiloteVoiture {  
    private Moteur moteur;  
    private Chassis chassis;  
    public PiloteVoiture() {  
        moteur = new Moteur(...);  
        chassis = new Chassis(...);  
    }  
    ...  
}
```

Injection de dépendances

Problème : comment l'objet englobant sait-il **comment** construire un objet particulier (i.e. quels sont les paramètres de construction de cet objet) ?

Est-ce que cela ne contrevient pas au **premier principe SOLID** (**S**ingle responsibility) ?

Et si ces paramètres de construction changent, qu'est-ce que ça veut dire pour l'objet englobant ?

Injection de dépendances

Si l'objet englobant sait comment construire un objet qu'il utilise (avec tous les détails nécessaires)

⇒ **Complexification** de l'objet

Si on modifie la manière de construire l'un des objets contenus dans l'objet englobant, alors l'objet englobant doit être modifié

⇒ **Impact** sur l'objet englobant

On dit que l'objet englobant **dépend** de l'objet qu'il utilise, ou que l'objet utilisé est une **dépendance** de l'objet englobant.

Injection de dépendances

L'une des buts principaux d'une bonne conception logicielle est de **limiter les dépendances** au maximum.

Sinon, toute modification sur un objet utilisé impliquera potentiellement des **modifications** sur l'objet qui l'utilise

- On parle de l'**impact** d'une modification, et on cherche toujours à **minimiser les impacts** !

Alors comment faire pour que l'objet englobant n'ait pas à savoir comment créer les objets qu'il utilise ?

Injection de dépendances

Première possibilité : utiliser le patron de conception ***Fabrique*** (design pattern ***Factory***).

Dans cette approche, une classe est dédiée à la fourniture (instanciation) des objets utilisés.

Injection de dépendances

Exemple :

```
class FabriqueVoiture {  
    public Moteur creeMoteur() {  
        ...  
    }  
    public Chassis creeChassis() {  
        ...  
    }  
}
```

Injection de dépendances

```
class PiloteVoiture {  
    private Moteur moteur;  
    private Chassis chassis;  
    private FabriqueVoiture fabriqueVoiture;  
    public PiloteVoiture() {  
        moteur = fabriqueVoiture.creeMoteur();  
        chassis = fabriqueVoiture.creeChassis();  
    }  
    ...  
}
```

Injection de dépendances

Question : comment l'objet englobant crée t'il l'instance de Fabrique ?



Injection de dépendances

On évite d'instancier la fabrique dans l'objet englobant, car on retomberait sur le même problème.



Injection de dépendances

Et si on faisait de la Fabrique un ***Singleton*** ?

```
public final class FabriqueVoiture {  
    private static final FabriqueVoiture _instance = new  
        FabriqueVoiture();  
  
    public static FabriqueVoiture getInstance() {  
        return FabriqueVoiture._instance ;  
    }  
  
    public Moteur creeMoteur() {  
        ...  
    }  
}
```

Injection de dépendances

```
class PiloteVoiture {  
    private Moteur moteur;  
    private Chassis chassis;  
    private FabriqueVoiture fabriqueVoiture =  
        FabriqueVoiture.getInstance();  
    public PiloteVoiture() {  
        moteur = fabriqueVoiture.creeMoteur();  
        chassis = fabriqueVoiture.creeChassis();  
    }  
    ...  
}
```




Injection de dépendances

Mais cette solution n'est pas idéale, car le `PiloteVoiture` dépend de l'implémentation de la `Fabrique` : si on veut changer de fabrique, alors il faut modifier `PiloteVoiture...`

⇒ Ce cas de figure se produira typiquement lorsque l'on voudra **tester** le `PiloteVoiture` dans un **environnement limité**, c'est-à-dire sans disposer d'une **vraie** `FabriqueVoiture`.

Injection de dépendances

Les solutions seraient alors :

- Soit d'indiquer à la `FabriqueVoiture` qu'on est en mode "test", par passage de paramètre par exemple 
- Ou bien de changer le code du `PiloteVoiture` pour utiliser une autre `FabriqueVoiture` plus simple (**stub** ou **mock**) 

Autrement dit, la solution de la `Fabrique` en tant que singleton n'est pas idéale.

Injection de dépendances

Ce qui serait bien, ce serait qu'un autre composant, externe au `PiloteVoiture`, lui fournisse une `FabriqueVoiture` toute prête

⇒ lui **injecte** une `FabriqueVoiture`

Voilà ! L'injection de dépendances, c'est principalement cela : récupérer automatiquement une instance d'objet utilisé dans un objet englobant.

Injection de dépendances

Et tant qu'à faire, pourquoi ne pas se passer de la `FabriqueVoiture`, et directement injecter le `Moteur` et le `Chassis` dans le `PiloteVoiture` ?

⇒ **injecter** les dépendances du `PiloteVoiture`

Injection de dépendances

```
class PiloteVoiture {  
    private Moteur moteur;  
    private Chassis chassis;  
    public void demarrerVoiture() {  
        moteur.demarrer();  
    }  
    public void arreterVoiture() {  
        moteur.arreter();  
    }  
    ...  
}
```

Injection de dépendances

IoC container, beans et métadonnées

Injection de dépendances

C'est ce que la partie Core du framework Spring permet de faire (entre autres).

Plus précisément, c'est ce qu'on appelle le « **Spring IoC container** » qui est en charge d'injecter les dépendances.

[IoC signifie « **Inversion of Control** ». C'est, plus ou moins, l'équivalent du terme « injection de dépendances ». L'idée derrière l'expression IoC, c'est que ce n'est plus l'application qui contrôle des librairies, c'est un framework (des librairies, représentant le Spring IoC container) qui contrôle l'application.]

Injection de dépendances

L'IoC container est un conteneur, comme un conteneur de servlets (exemple : Tomcat) ou un conteneur d'EJB, mais spécialisé dans la gestion des graphes d'objets.

Ce conteneur est représenté par l'interface `org.springframework.context.ApplicationContext`.

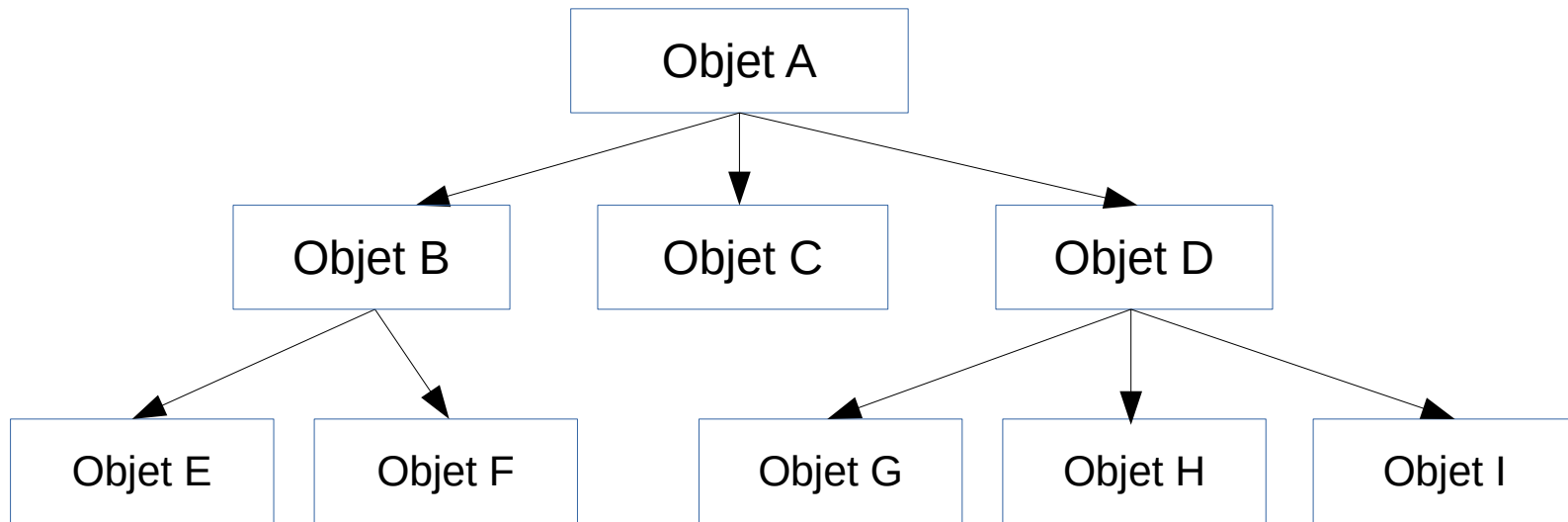
Cette interface étend l'interface `BeanFactory` du package `org.springframework.beans.factory`, qui implémente les fonctions du noyau (core) du framework Spring.

Injection de dépendances

L'IoC container est en charge d'**instancier** les objets (les créer), de les **configurer** (donner une valeur à leurs propriétés) et de les **assembler** (établir les dépendances entre objets).

Injection de dépendances

Un exemple simple de graphe d'objets : l'objet A dépend des objets B, C et D ; B dépend de E et F ; D dépend de G, H et I.



Injection de dépendances

Comment indique t'on à l'IoC container quel est le graphe d'objets à construire et quelles sont les caractéristiques de ces objets (paramètres de construction) ?

⇒ au travers de **métadonnées**

Ces métadonnées sont définies dans des **fichiers XML**, des **annotations Java** ou dans du **code Java** extérieur à l'application.

Dans la terminologie de l'IoC container Spring, un objet est appelé un **bean**.

Injection de dépendances

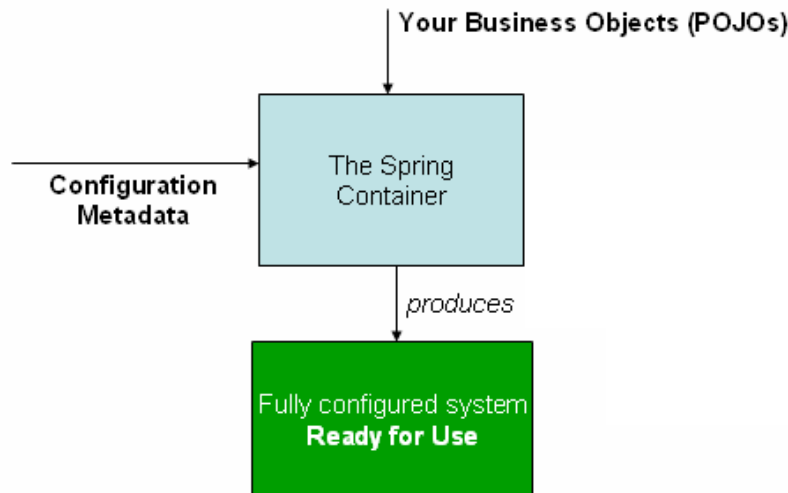


Schéma extrait de la documentation de Spring

POJO signifie « Plain Old Java Object » : un objet Java standard

Injection de dépendances

Exemple de fichier XML de métadonnées

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           https://www.springframework.org/schema/beans/spring-beans.xsd">

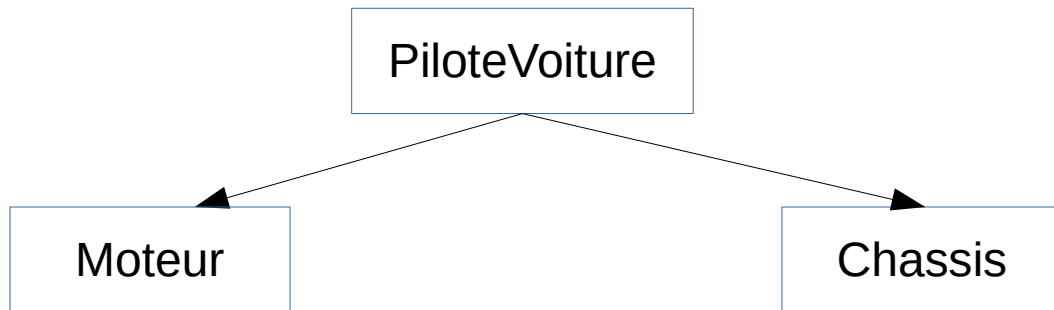
  <bean id="..." class="...">
    <!-- informations sur la configuration et les propriétés de ce bean
    -->
  </bean>

</beans>
```

Injection de dépendances

Reprenons l'exemple précédent :

- Un `PiloteVoiture` utilise un `Moteur`
- Un `PiloteVoiture` utilise un `Chassis`



Injection de dépendances

Fichier XML de métadonnées correspondant à cet exemple

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<beans>
```

```
  <bean id="moteur" class="exemples.Moteur"/>
```

```
  <bean id="chassis" class="exemples.Chassis"/>
```

```
  <bean id="piloteVoiture" class="exemples.PiloteVoiture">
```

```
    <property name="leMoteur" ref="moteur"/>
```

```
    <property name="leChassis" ref="chassis"/>
```

```
  </bean>
```

```
</beans>
```

Injection de dépendances

Le graphe d'objets peut être déclaré de deux manières :

- Au moyen d'un **fichier XML de métadonnées** (ou d'un **fichier Groovy de métadonnées**)
- Ou au moyen de classes Java dédiées (on parle alors de « **Java-based container configuration** »).

On verra qu'il est aussi possible de **décrire** les beans et leurs relations au moyen d'annotations Java.

Selon les cas, l'IoC container lui-même est **instancié** différemment.

Injection de dépendances

Cas d'un IoC container permettant l'utilisation des annotations :

```
ApplicationContext context = new  
AnnotationConfigApplicationContext("mon.package.a.moi");
```

Ce type d'IoC container autorise aussi la « Java-based configuration ».

Injection de dépendances

Cas d'un fichier XML :

```
ApplicationContext context = new  
ClassPathXmlApplicationContext("fichier1.xml",  
"fichier2.xml", ...);
```

Cas d'un fichier Groovy :

```
ApplicationContext context = new  
GenericGroovyApplicationContext("fichier1.groovy",  
"fichier2.groovy", ...);
```

Injection de dépendances

La récupération d'une instance de bean est effectuée ainsi.

```
// Récupération du conteneur
```

```
ApplicationContext contexte = ...;
```

```
// Récupération d'un bean
```

```
ClasseDuBean leBean = contexte.getBean("<id du bean>",  
ClasseDuBean);
```

```
// Utilisation du bean
```

```
leBean.<méthode>(...);
```

Injection de dépendances

Instanciación de beans

Injection de dépendances

L'IoC container peut créer une instance de bean de deux manières différentes :

- Par utilisation d'un constructeur du bean
 - Avec ou sans paramètres
- Par utilisation d'une factory
 - Soit avec appel d'une méthode statique
 - Soit avec appel d'une méthode d'instance

Injection de dépendances

Spring permet d'instancier un bean par appel d'un constructeur :

- Sans paramètre
 - appelé “constructeur par défaut”, celui fourni obligatoirement par une classe dite “JavaBean”
- Ou avec paramètres

Spring est donc **ouvert**, dans le sens qu'il accepte tous les types de beans (véritables JavaBeans ou appartenant à d'autres types de classe).

Injection de dépendances

La classe du bean à créer doit être spécifiée.

Lorsque les métadonnées sont définies dans un fichier XML, l'attribut **class** de la balise `<bean>` permet d'indiquer cette classe :

```
<bean id="moteur" class="exemples.Moteur" />
```

Le nom complet de la classe (i.e. avec les noms de package) doit être indiqué.

Injection de dépendances

Spring permet aussi d'instancier un bean par appel d'une méthode **statique** de fabrication.

Dans ce cas, l'attribut **class** de la définition du bean permet d'indiquer la classe comportant la méthode statique de fabrication.

Un attribut **factory-method** permet d'indiquer le nom de cette méthode statique.

```
<bean id="moteur"  
      class="exemples.FabriqueVoiture"  
      factory-method="creeMoteur" />
```


Injection de dépendances

Exemple adapté à la définition de bean précédente :

```
package exemples;

class FabriqueVoiture {
    public static Moteur creeMoteur() {
        return new Moteur();
    }
    ...
}
```

Injection de dépendances

Spring permet également d'instancier un bean par appel d'une méthode **d'instance** de fabrication.

Dans ce cas de figure, l'attribut **class** de la définition du bean n'est pas utilisé.

Par contre, l'attribut **factory-bean** est utilisé pour indiquer l'identifiant du bean portant la méthode de fabrication.

Là aussi, le nom de la méthode de fabrication est indiqué par l'attribut **factory-method**.

```
<bean id="chassis"  
      factory-bean="exemples.PiloteVoiture" factory-method="creeChassis"/>
```

Injection de dépendances

Exemple adapté à la définition de bean précédente :

```
package exemples;

class PiloteVoiture {
    public Chassis creeChassis() {
        return new Chassis();
    }
    ...
}
```

Injection de dépendances

Comme vu précédemment, les dépendances d'un bean sont les objets qu'il utilise, qui sont ses propriétés:

```
class PiloteVoiture {  
    private Moteur moteur;  
    private Chassis chassis;  
    ...  
}
```

Le moteur et le chassis sont des **dépendances** du bean `PiloteVoiture`.

Injection de dépendances

Configuration de beans

Injection de dépendances

Spécifier les dépendances d'un bean peut être réalisé de trois manières différentes :

- En précisant des **arguments** lors de l'appel du **constructeur** du bean
 - Ces arguments étant bien sûr des dépendances du bean
- En précisant des **arguments** lors de l'appel d'une **méthode de fabrication** (statique ou pas)
 - Ces arguments étant bien sûr des dépendances du bean
- En utilisant des **setters** du bean
 - Ces setters permettant de positionner les dépendances

Injection de dépendances

L'IoC container sera alors en charge d'injecter les dépendances du bean ainsi défini :

- Soit lors de l'appel du constructeur du bean
- Soit lors de l'appel de la méthode de fabrication du bean
- Soit en appelant les setters du bean nouvellement créé

Injection de dépendances

L'injection de dépendances **par arguments de constructeur** correspond à l'exemple Java suivant :

```
class PiloteVoiture {  
    private Moteur moteur;  
    private Chassis chassis;  
    public PiloteVoiture (Moteur moteur, Chassis chassis) {  
        this.moteur = moteur;  
        this.chassis = chassis;  
    }  
    ...  
}
```


Injection de dépendances

Pour un fichier XML de métadonnées, cette injection de dépendances est portée par la balise **constructor-arg** et l'attribut **ref** de cette balise.

```
<bean id ="aBean" class="ABeanClass">  
  <bean id ="anotherBean" class="AnotherBeanClass">  
    <constructor-arg ref="aBean">  
  </bean>  
</bean>
```

Injection de dépendances

L'injection de dépendances de l'exemple serait alors indiquée ainsi :

```
<beans>

  <bean id="piloteVoiture" class="exemples.PiloteVoiture">
    <constructor-arg ref="moteur"/>
    <constructor-arg ref="chassis"/>
  </bean>

  <bean id="moteur" class="exemples.Moteur"/>
  <bean id="chassis" class="exemples.Chassis"/>
</beans>
```

Injection de dépendances

Question : dans l'exemple précédent, comment sont créés et configurés les beans Chassis et Moteur ?



Injection de dépendances

Réponse : comme n'importe quel autre bean, ils sont instanciés par l'IoC container :

- Soit au moyen de leur constructeur par défaut
- Soit par appel d'une méthode de fabrication, statique ou non

et leurs dépendances leur sont injectées :

- Soit par passage d'arguments au constructeur ou à la méthode de fabrication
- Soit par appel de setters

Injection de dépendances

Si un constructeur de bean prend comme argument d'autres beans, **la classe de ces derniers étant spécifiée**, il n'y a pas d'ambiguïté.

Si par contre un constructeur prend comme argument des **valeurs de type simple** (i.e. pas des objets Java), il peut y avoir ambiguïté :

- Une valeur numérique indiquée dans les métadonnées, comme par exemple 123, représente t'elle une chaine de caractères ou une valeur numérique entière ou une valeur numérique flottante ou ... ?

Injection de dépendances

Pour lever cette ambiguïté, Spring propose 3 possibilités :

- Utilisation d'un attribut `type` de la balise `constructor-arg`, pour indiquer le **type** de l'argument
- Utilisation d'un attribut `index`, pour spécifier l'**index** de l'argument parmi les arguments du constructeur
- Utilisation d'un attribut `name`, pour faire le lien avec le **nom** de l'argument correspondant dans le constructeur.

Grâce à ces informations, l'IoC container peut savoir **quelle valeur utiliser** pour chacun des paramètres de type simple.

Injection de dépendances

Exemple de constructeur prenant en entrée des valeurs de type simple :

```
class Moteur {  
    private String constructeur;  
    private int numeroDeSerie;  
    public Moteur (String constructeur, int numeroDeSerie) {  
        this.constructeur = constructeur;  
        this.numeroDeSerie = numeroDeSerie;  
    }  
    ...  
}
```

Injection de dépendances

On veut construire un objet `Moteur` dont `Toyota` est le constructeur et `1234567890` est le numéro de série.

Première possibilité : indiquer le **type** des arguments.

```
<bean id="moteur" class="exemples.Moteur">  
    <constructor-arg type="int" value="1234567890" />  
    <constructor-arg type="java.lang.String" value="Toyota" />  
</bean>
```


Injection de dépendances

NB. Si le constructeur comporte plusieurs arguments de même type, les valeurs indiquées dans le fichier de métadonnées seront appliquées aux arguments du constructeur dans l'ordre.

Injection de dépendances

```
public class UnBean {  
    private int valeur1;  
    private int valeur2;  
  
    public UnBean(int valeur1, int valeur2) {  
        this.valeur1 = valeur1;  
        this.valeur2 = valeur2;  
    }  
    ...  
}
```

Injection de dépendances

Quelle sera la valeur des propriétés `valeur1` et `valeur2` d'une instance de `UnBean` configurée au moyen des métadonnées suivantes ?

```
<bean id="unBean" class="unpackage.UnBean">  
    <constructor-arg type="int" value="2" />  
    <constructor-arg type="int" value="1" />  
</bean>
```



Injection de dépendances

La valeur de la propriété `valeur1` sera 2 et la valeur de la propriété `valeur2` sera 1.

Injection de dépendances

Deuxième possibilité : indiquer l'**indice** des arguments.

```
<bean id="moteur" class="exemples.Moteur">  
  <constructor-arg index="1" value="1234567890" />  
  <constructor-arg index="0" value="Toyota" />  
</bean>
```

NB. Les indices commencent à 0.

Injection de dépendances

Troisième possibilité : indiquer le **nom** des arguments.

```
<bean id="moteur" class="exemples.Moteur">  
    <constructor-arg name="numeroDeSerie" value="1234567890"/>  
    <constructor-arg name="constructeur" value="Toyota" />  
</bean>
```

NB. Pour que cela fonctionne, il faut que les classes Java soient **compilées en mode debug**, que Spring puisse disposer des informations d'inspection nécessaires.

Injection de dépendances

Si les classes Java ne sont pas compilées en mode debug, il reste la possibilité d'utiliser l'annotation standard **@ConstructorProperties** pour expliciter le nom des arguments du constructeur :

```
@ConstructorProperties({"constructeur", "numeroDeSerie"})  
public Moteur (String constructeur, int numeroDeSerie) {  
    this.constructeur = constructeur;  
    this.numeroDeSerie = numeroDeSerie;  
}
```

Injection de dépendances

Autre possibilité pour injecter les dépendances : passage d'arguments à une méthode de fabrication.

```
class FabriqueVoiture {  
    public static Moteur creeMoteur(int numeroDeSerie,  
        String constructeur) {  
        return new Moteur(constructeur, numeroDeSerie);  
    }  
  
    ...  
}
```


Injection de dépendances

Dans le fichier de métadonnées, il suffit d'ajouter une balise `<constructor-arg>` pour chaque argument à la définition du bean construit par méthode de fabrication :

```
<bean id="moteur" class="exemples.FabriqueVoiture" factory-  
method="creeMoteur">
```

```
    <constructor-arg type="int" value="13579" />
```

```
    <constructor-arg type="java.lang.String" value="Lamborghini"/>
```

```
</bean>
```

Injection de dépendances

On a vu comment injecter des dépendances par arguments de constructeur ou méthode de fabrication.

Il reste à voir comment injecter des dépendances par setter...

Injection de dépendances

Rappel sur les setters.

Un setter (modificateur) est une méthode publique d'instance de classe permettant d'**affecter une valeur** à **une propriété** bien précise d'une instance de cette classe.

La propriété est normalement privée (`private`), le setter est le **seul moyen** pour les utilisateurs de cette classe de modifier la propriété.

Injection de dépendances

```
class PiloteVoiture {  
    private Moteur moteur;  
    private Chassis chassis;  
  
    public void setMoteur(Moteur moteur) {  
        this.moteur = moteur;  
    }  
  
    public void setChassis(Chassis chassis) {  
        this.chassis = chassis;  
    }  
  
    ...  
}
```

Injection de dépendances

L'utilisation d'un setter est indiquée par une balise `<property>`.

```
<bean id="moteur" class="exemples.Moteur" />
```

```
<bean id="chassis" class="exemples.Chassis" />
```

```
<bean id="piloteVoitureBean" class="..." />
```

```
    <property name="moteur" ref="moteur" />
```

```
    <property name="chassis" ref="chassis" />
```

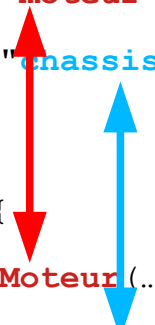
```
</bean>
```

Injection de dépendances

Attention ! L'attribut `name` de la balise `property` doit avoir **exactement** la même valeur que le nom du setter correspondant...

```
<bean id="piloteVoitureBean" class="..." />
    <property name="moteur" ref="moteur" />
    <property name="chassis" ref="chassis" />
</bean>

class PiloteVoiture {
    public void setMoteur(...) { ... }
    public void setChassis(...) { ... }
    ...
}
```



Injection de dépendances

Bien sûr, il est tout à fait possible de mixer la façon d'injecter les dépendances :

- Certaines dépendances peuvent être construites par un constructeur
- D'autres dépendances peuvent être construites par une méthode de fabrication
- Et enfin d'autres peuvent aussi être construites par utilisation de setters.

Injection de dépendances

Mise en pratique :

- Installation des librairies Spring
- Installation de Spring Tool Suite
- Exercice 01 : injection de dépendances “classique” (constructeurs par défaut et setters)



Injection de dépendances

Mise en pratique :

- Exercice 02 : invocation de constructeurs avec arguments



Injection de dépendances

Mise en pratique :

- Exercice 03 : instanciation par fabrique



Injection de dépendances

Reprenons l'exemple du `PiloteVoiture`.

Il dépend des classes `Moteur` **et** `Chassis`.

On veut pouvoir valider le fonctionnement du `PiloteVoiture`.

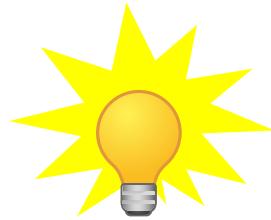
Problème : l'équipe en charge de la classe `Moteur` n'a pas encore terminé son développement...

Comment faire pour tester le `PiloteVoiture` quand même ?



Injection de dépendances

Réponse : en utilisant un **simulateur** de Moteur !



Injection de dépendances

Un simulateur de `Moteur` est une classe beaucoup plus simple à développer qu'une vraie classe `Moteur`, mais fournissant néanmoins les mêmes services que la classe `Moteur`.

```
public class Moteur {  
    public void demarrer();  
    public void arreter();  
    ...  
}
```

Injection de dépendances

En remplaçant la vraie classe `Moteur` par un simulateur (appelé `SimuMoteur`), on peut avancer dans les tests et le développement du `PiloteVoiture` sans devoir attendre que le `Moteur` soit développé.

C'est une technique très répandue pour **découpler les activités** dans un projet informatique.

Injection de dépendances

OK, donc on choisit cette solution. Que faut-il faire dans notre projet pour que l'on puisse ainsi remplacer la classe `Moteur` ?



Injection de dépendances

1. La classe `PiloteVoiture` ayant une dépendance vers `Moteur`, il faut la modifier pour qu'elle utilise `SimuMoteur` à la place.
2. Le fichier de dépendances XML utilise aussi directement la classe `Moteur` – sauf si on utilise une fabrique de `Moteur`. Il faut donc a priori modifier aussi ce fichier.

Injection de dépendances

Modification de la classe `PiloteVoiture` :

```
class PiloteVoiture {  
    private SimuMoteur moteur;  
    public void setMoteur(SimuMoteur moteur) {  
        this.moteur = moteur;  
    }  
    ...  
}
```

Modification du fichier de dépendances XML :

```
<bean id="moteur" class="exemples.SimuMoteur" />
```

Injection de dépendances

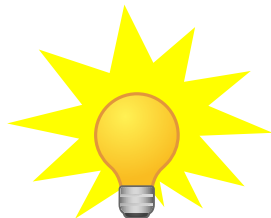
N'y aurait-il pas été possible de limiter l'impact de cette modification (changement de `Moteur`) ?



Injection de dépendances

Réponse : en utilisant une **interface** plutôt qu'une classe concrète `Moteur` dans la classe utilisatrice (`PiloteVoiture`).

Après tout, le `PiloteVoiture` n'a pas besoin de savoir **comment** les services du `Moteur` sont rendus...



Injection de dépendances

Définition de IMoteur, l'interface du Moteur

```
interface IMoteur {  
    void demarrer();  
    void arreter();  
    ...  
}
```

Injection de dépendances

Nouvelle version du PiloteVoiture

```
class PiloteVoiture {  
    private IMoteur moteur;  
    private Chassis chassis;  
    public PiloteVoiture (IMoteur moteur, Chassis chassis) {  
        this.moteur = moteur;  
        this.chassis = chassis;  
    }  
    ...  
}
```

Injection de dépendances

Nouvelle version du Moteur

```
class Moteur implements IMoteur {  
    private int numeroDeSerie;  
    private String constructeur;  
    public Moteur (String constructeur, int numeroDeSerie) {  
        this.numeroDeSerie = numeroDeSerie;  
        this.constructeur = constructeur;  
    }  
    ...  
}
```

Injection de dépendances

Nouvelle version du SimuMoteur

```
class SimuMoteur implements IMoteur {  
    private int numeroDeSerie;  
    private String constructeur;  
    public SimuMoteur (String constructeur, int numeroDeSerie) {  
        this.numeroDeSerie = numeroDeSerie;  
        this.constructeur = constructeur;  
    }  
    ...  
}
```

Injection de dépendances

Pas de modification nécessaire du fichier de dépendances XML, car ce dernier **crée des instances**, il utilise donc des **classes concrètes** (`Moteur`, `SimuMoteur`, ...).

Au travers de l'utilisation de l'interface `IMoteur`, on peut basculer du `Moteur` final vers le simulateur de `Moteur` ou vice-versa sans changer le code, simplement en changeant de **classe concrète** à instancier dans le fichier XML de métadonnées

==> gain très net en souplesse

Injection de dépendances

Mise en pratique :

- Exercice 04 : utilisation d'interfaces



Injection de dépendances

Il est possible d'injecter un ensemble de beans comme dépendance.

Un bean peut avoir comme propriété une collection d'objets (`List`, `Set`, `Map` ou simple tableau).

Pour cela, on utilise une sous-balise `<list>`, `<set>` ou `<map>` de la balise `<property>`.

Injection de dépendances

Exemple : on veut insérer dans un bean une liste de noms comme dépendances. Ces noms sont des String et sont définis dans un fichier XML de métadonnées.

On choisit comme noms `Element 1`, `Element 2` **et** `Element 3`.

Injection de dépendances

Extrait du fichier XML de métadonnées `liste_elements.xml`.

```
<bean class="exemples.spring.formation.CollectionBean">
```

```
  <property name="elements">
```

```
    <list>
```

```
      <value>Element 1</value>
```

```
      <value>Element 2</value>
```

```
      <value>Element 3</value>
```

```
    </list>
```

```
  </property>
```

```
</bean>
```

Injection de dépendances

La classe CollectionBean.

```
public class CollectionBean {  
    private List<String> elements;  
  
    public void setElements(List<String> elements) {  
        this.elements = elements;  
    }  
  
    public void affiche() {  
        System.out.println(this.elements);  
    }  
}
```

Injection de dépendances

Le programme de test.

```
public class TestCollection {  
  
    public static void main(String arguments) {  
  
        ClassPathXmlApplicationContext context = new  
ClassPathXmlApplicationContext("liste_elements.xml") ;  
  
        CollectionBean listeDElements = (CollectionBean)  
context.getBean(CollectionBean.class) ;  
  
        listeDElements.affiche() ;  
  
        context.close() ;  
  
    }  
  
}
```

Injection de dépendances

Output.

```
[Element 1, Element 2, Element 3]
```

Injection de dépendances

La méthode « standard » d'instanciation de bean et d'injection de dépendances, c'est :

- constructeur par défaut
- et utilisation de setters.

Mais bien sûr, le développeur a toute latitude pour faire autrement.

Injection de dépendances

Portée d'un bean

Injection de dépendances

La définition d'un bean correspond à une recette de cuisine : elle permet de **créer** quelque chose et peut être utilisée de **nombreuses** fois.

Si la définition n'est utilisée qu'**une seule fois** dans l'application, alors il n'y aura qu'un seul bean de ce type : on parle de bean « **singleton** ».

Si la définition d'un bean est utilisée de nombreuses fois, alors de nombreux beans de ce type seront créés : on parle alors de beans « **prototypes** ».

Injection de dépendances

Un bean peut aussi avoir une durée de vie bien précise dans le cas d'une application web :

- Limitée au traitement d'une **requête** HTTP
- Limitée à la **session** HTTP en cours
- Limitée à l'**application web** elle-même

Ou bien, dans le cas des **websockets**, un bean peut avoir une durée de vie limitée à celle d'une websocket.

Injection de dépendances

Petite pause culturelle : qu'est-ce qu'une websocket ?



Injection de dépendances

Cette notion de durée de vie ou de nombre d'occurrences est appelée la **portée** du bean, alias « **scope** » en anglais.

Le scope d'un bean peut être précisé dans les métadonnées de l'application, par exemple :

```
<bean id="Moteur" class="exemples.Moteur"  
scope="singleton"/>
```

```
<bean id="Chassis" class="exemples.Chassis"  
scope="prototype"/>
```

Injection de dépendances

Spring supporte 6 scopes différents :

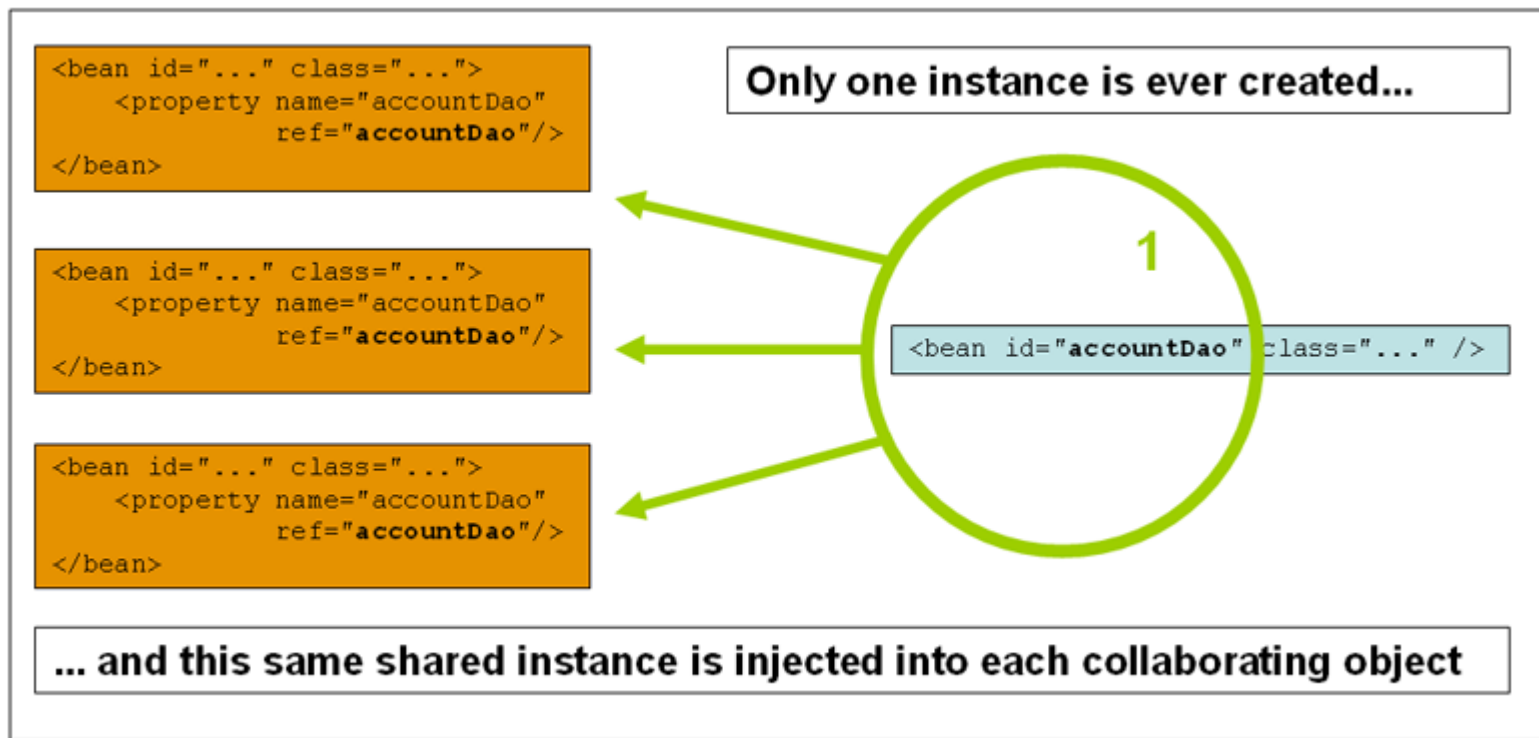
- `singleton` (la valeur par défaut)
- `prototype`
- `request`
- `session`
- `application`
- `websocket`

Injection de dépendances

Fort logiquement, les 4 dernières valeurs de scope de la liste précédente ne sont disponibles que pour des `ApplicationContext` de type web.

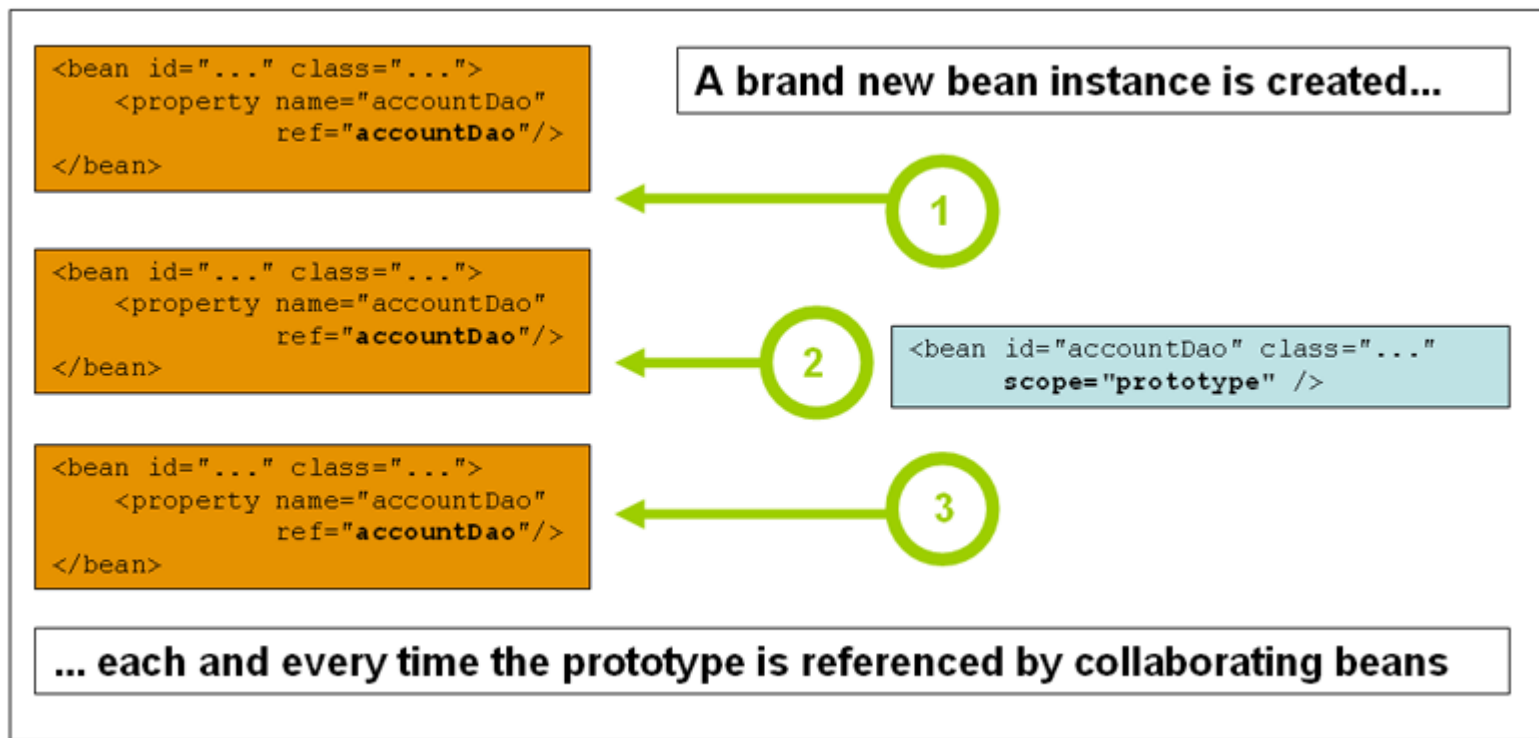
Elles seront discutées dans la partie Spring MVC.

Injection de dépendances



@Pivotal

Injection de dépendances



@Pivotal

Injection de dépendances

Customisation du cycle de vie des beans

Injection de dépendances

Il est possible de réaliser des traitements :

- juste après qu'un bean ait été créé par l'IoC container
- ou bien juste avant qu'il soit supprimé.

On peut ainsi reprendre temporairement la main sur l'IoC container pour réaliser des traitements applicatifs lors de ces moments particuliers.

Ceci est réalisé en accrochant des **callbacks** à une définition de bean.

Injection de dépendances

Association d'une callback sur **initialisation** d'un bean :

```
<bean id="exempleBean" class="exemples.ExempleBean"  
init-method="init"/>
```

`init()` est une méthode définie dans la classe `ExempleBean`.

Injection de dépendances

Code de la classe ExempleBean :

```
public class ExempleBean {  
    public void init() {  
        // traitements d'initialisation à réaliser  
    }  
    ...  
}
```

Injection de dépendances

Association d'une callback sur **destruction** d'un bean :

```
<bean id="exempleBean" class="exemples.ExempleBean"  
destroy-method="cleanup"/>
```

`cleanup()` est une méthode définie dans la classe `ExempleBean`.

Injection de dépendances

Code de la classe ExempleBean :

```
public class ExempleBean {  
    public void cleanup() {  
        // traitements de nettoyage à réaliser  
    }  
    ...  
}
```

Injection de dépendances

Le même résultat peut être obtenu au moyen des annotations standard `@PostConstruct` **et** `@PreDestroy`.

Injection de dépendances

```
public class ExempleBean {  
    @PostConstruct  
    public void init() {  
        // traitements d'initialisation à réaliser  
    }  
    @PreDestroy  
    public void cleanup() {  
        // traitements de nettoyage à réaliser  
    }  
    ...  
}
```

Injection de dépendances

Il existe une 3ème manière de faire : en implémentant les interfaces **spécifiques de Spring** `InitializingBean` et `DisposableBean`.

```
import org.springframework.beans.factory;  
  
public class ExempleBean implements InitializingBean {  
    public void afterPropertiesSet() {  
        // traitements d'initialisation à réaliser  
    }  
    ...  
}
```

Injection de dépendances

```
import org.springframework.beans.factory;

public class ExempleBean implements DisposableBean {

    public void destroy() {

        // traitements de nettoyage à réaliser

    }

    ...

}
```

Injection de dépendances

Bien entendu, dans ce dernier cas, il n'est pas nécessaire de préciser les attributs `init-method` et `destroy-method` dans la définition du bean.

```
<bean id="exempleBean" class="exemples.ExempleBean" />
```

Injection de dépendances

Pourquoi cette dernière manière de faire n'est-elle pas recommandée ?



Injection de dépendances

Parce qu'en dérivant d'une classe spécifique de Spring, on crée une dépendance vers le framework Spring.

Par contre, les annotations `@PreDestroy` et `@PostConstruct` sont standard des annotations Java standard, elles sont donc supportées par d'autres frameworks que Spring.

Elles permettent de conserver aux classes de bean leur caractère de POJO.

Injection de dépendances

Il est possible de préciser à l'IoC container un nom de méthode d'initialisation ou de nettoyage **par défaut** pour tous les beans d'un `ApplicationContext`, au moyen des attributs `default-init-method` et `default-destroy-method` de la balise `<beans>` :

```
<beans default-init-method="init" default-destroy-  
method="destroy">
```

```
    <bean id="exempleBean1" class="exemples.ExempleBean1"/>
```

```
    <bean id="exempleBean2" class="exemples.ExempleBean2"/>
```

```
    <bean id="exempleBean3" class="exemples.ExempleBean3"/>
```

```
</beans>
```

Injection de dépendances

Dans l'exemple précédent, les classes `ExempleBean1`, `ExempleBean2` et `ExempleBean3` portant une méthode `init()` (resp. `destroy()`) verront cette méthode automatiquement appelée lors de l'instanciation d'un bean de cette classe (resp. lors de la suppression d'un bean de cette classe).

Si une classe ne porte pas cette ou ces méthode(s), alors il ne se passera rien à l'instanciation ou à la suppression d'un bean de ce type.

Injection de dépendances

Il peut être nécessaire que des beans soient informés du démarrage et de l'arrêt du système (et donc de l'IoC container). C'est le cas par exemple si des beans dépendent d'un processus s'exécutant en background (antivirus, spooler d'impression, etc.).

Il est possible d'attacher une callback de beans à ces événements en leur faisant **implémenter l'interface Lifecycle**.

Injection de dépendances

Définition de l'interface

```
public interface Lifecycle {  
    void start();  
    void stop();  
    boolean isRunning();  
}
```

Injection de dépendances

Mise en oeuvre

```
import org.springframework.context;  
  
public class ExempleBean implements Lifecycle {  
    public void start() {  
        // traitements à réaliser au démarrage du système  
    }  
}
```

Injection de dépendances

```
public void stop() {  
    // traitements à réaliser à l'arrêt du système  
}  
  
public boolean isRunning() {  
    // permet d'indiquer si le bean est en cours  
d'exécution  
}  
}
```

Injection de dépendances

NB. Cette interface n'est active que pour les beans **singleton** de plus haut niveau.

Injection de dépendances

Utilisation des annotations

Injection de dépendances

La définition de métadonnées par fichier XML est un moyen de regrouper la configuration de tout le système dans un ou plusieurs fichiers XML.

Cette méthode permet également de changer la configuration du système sans toucher au code.

Il existe une autre manière de faire : utiliser les **annotations**.

Il s'agit d'annotations placées directement dans le code Java, **au plus près** des beans ou de leurs propriétés ainsi contrôlés.

Injection de dépendances

L'utilisation d'annotations et de fichiers XML de métadonnées peuvent être **combinées**.

NB important : les annotations sont traitées **avant** l'analyse des fichiers XML de métadonnées, par conséquent les instructions données par ces derniers **supplantent** les éventuelles annotations correspondantes.

Injection de dépendances

Les annotations supportées sont :

- Celles de la **JSR-250**
- Celles de la **JSR-330**
- Celles spécifiques à Spring
 - Plus nombreuses et généralement plus puissantes que celles des JSR

Injection de dépendances

Qu'est-ce qu'une JSR ?



Injection de dépendances

1998 → définition par la société Sun Microsystems (créateur du langage) d'un mécanisme appelé **Java Community Process**.

Ce mécanisme permet à tout le monde de proposer des **évolutions** du langage Java, au travers de spécifications techniques (documents) appelées **Java Specification Request** (JSR).

C'est l'équivalent dans le monde Java des **Request For Comments** (RFC) de l'Internet Society.

Injection de dépendances

Exemples de JSR :

- JSR-53 → Servlets 2.3 et JSP
- JSR-315 → Servlets 3.0
- JSR-250 → annotations générales (définition de ressources, gestion du cycle de vie des beans, gestion des droits d'accès et priorité)
- JSR-330 → annotations pour **injection de dépendances**

Injection de dépendances

NB. Il n'y a pas d'équivalence parfaite entre ce qui peut être configuré au moyen d'un fichier XML de métadonnées et ce qui peut être configuré au moyen d'annotations.

Injection de dépendances

Les annotations sont activées en incluant une balise `<context:annotation-config />` dans le fichier de configuration XML.

Le namespace `context` correspond au schéma `http://www.springframework.org/schema/context` :

```
xmlns:context="http://www.springframework.org/schema/context"
```

Injection de dépendances

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-
                           beans.xsd
                           http://www.springframework.org/schema/context
                           https://www.springframework.org/schema/context/spring-
                           context.xsd">
  <context:annotation-config />
</beans>
```

Injection de dépendances

Il est aussi possible d'utiliser une implémentation d'`ApplicationContext` qui reconnaît automatiquement les annotations : `AnnotationConfigApplicationContext`.

```
import  
org.springframework.context.annotation.AnnotationConfigApplicati  
onContext;
```

```
...
```

```
AnnotationConfigApplicationContext context = new  
AnnotationConfigApplicationContext (AClass.class,  
Bclass.class, ...);
```


Injection de dépendances

Une autre manière de faire consiste à faire scanner des classes par ce type d'ApplicationContext :

```
import
org.springframework.context.annotation.AnnotationConfigApplicati
onContext;

...

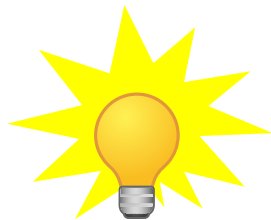
AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext();

context.scan("a.given.package");

context.refresh();
```

Injection de dépendances

Pour cet `ApplicationContext`, scanner une classe consiste à rechercher dans cette classe les informations d'injection de dépendance qui sont indiquées au travers des... **annotations !**



Injection de dépendances

La ou les classes à scanner sont indiquées à cet `ApplicationContext` via les paramètres de son constructeur ou au travers de sa méthode `scan()`.

Injection de dépendances

NB.

La directive `<context:annotation-config />` active les annotations sur les beans qui se trouvent **déjà** dans le contexte applicatif, elle ne crée **pas** de bean.

“Activer” veut dire qu’un outil va les repérer et prendre des actions en conséquence. Un exemple classique d’action est d’injecter des dépendances pour les champs et méthodes annotées par `@Autowired`.

Injection de dépendances

Par contre, la directive `<context:component-scan />` fait deux choses :

- Elle crée les beans qui sont indiqués dans le code source au moyen d'annotations
- Elle active les annotations sur ces beans, permettant ainsi dans le cas de `@Autowired` d'injecter les dépendances, puisque tous les beans sont alors définis.

Injection de dépendances

Annotation **@Component**

Annotation spécifique de Spring, permettant d'indiquer à l'IoC container qu'une classe est un bean.

```
import org.springframework.stereotype.Component;
```

@Component

```
public class Voiture {  
    ...  
}
```

Injection de dépendances

Exemple

```
package exemples.spring.formation;  
  
import org.springframework.stereotype.Component;
```

@Component

```
public class Voiture {  
    public String getManufacturer() {  
        return "Renault";  
    }  
}
```

Injection de dépendances

```
package exemples.spring.formation;

import
org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class TestComponent {

    public static void main(String[] arguments) {

        AnnotationConfigApplicationContext context = new
        AnnotationConfigApplicationContext();

        context.scan("exemples.spring.formation");

        context.refresh();
    }
}
```


Injection de dépendances

```
Voiture voiture = (Voiture) context.getBean(Voiture.class);  
  
System.out.println("Fabriquant de la voiture : " +  
voiture.getManufacturer());  
  
context.close();  
  
}
```

```
}
```

Output :

```
Fabriquant de la voiture : Renault
```

Injection de dépendances

Et si on veut récupérer le bean par son nom ?

```
package exemples.spring.formation;

import org.springframework.stereotype.Component;

@Component("voiture")

public class Voiture {

    public String getManufacturer() {

        return "Renault";

    }

}
```

Injection de dépendances

```
Voiture voiture = (Voiture) context.getBean("voiture");  
System.out.println("Fabriquant de la voiture : " +  
voiture.getManufacturer());  
context.close();  
}
```

```
}
```

Output :

Fabriquant de la voiture : Renault

Injection de dépendances

L'annotation `@Component` est une annotation générique.

Elle est souvent remplacée par des annotations indiquant un type de composant précis, comme :

- les contrôleurs dans le modèle MVC → `@Controller`
- les DAO → `@Repository`
- ou les classes principales de servlet → `@Service`

Ces types de composant sont appelés des **stéréotypes** Spring.

Injection de dépendances

L'annotation Java standard (presque) équivalente est `@Named`.

Elle est définie dans le package `javax.inject`.

Injection de dépendances

Mise en pratique :

- Exercice 05 : définition de bean par annotation



Injection de dépendances

Annotation **@Autowired**

Annotation spécifique de Spring, permettant d'indiquer à l'IoC container un constructeur, un setter, voire toute méthode arbitraire, à utiliser pour injecter **automatiquement** des dépendances.

Automatiquement, cela veut dire qu'on n'a pas besoin de préciser les choses dans un fichier XML de métadonnées.

Injection de dépendances

Exemple.

Une Voiture contient un Moteur.

On veut que lorsqu'une instance de Voiture est créée, son Moteur soit automatiquement instancié.

Et on ne souhaite pas utiliser de fichier XML de métadonnées pour préciser cela...

Injection de dépendances

```
package exemples.spring.formation;

import org.springframework.stereotype.Component;

@Component

public class Moteur {

    public String getManufacturer() {
        return "Renault";
    }

}
```

Injection de dépendances

```
@Component("voiture")  
  
public class Voiture {  
    @Autowired  
    private Moteur moteur;  
  
    public String getInformations() {  
        return "Voiture avec moteur " +  
            this.moteur.getManufacturer();  
    }  
}
```

Injection de dépendances

```
package exemples.spring.formation;

import
org.springframework.context.annotation.AnnotationConfigApplication
nContext;

public class TestAutowired {

    public static void main(String args[]) {

        AnnotationConfigApplicationContext context = new
        AnnotationConfigApplicationContext();

        context.scan("exemples.spring.formation");

        context.refresh();
    }
}
```

Injection de dépendances

```
Voiture voiture = context.getBean("voiture");  
System.out.println("Infos sur la voiture : " +  
voiture.getInformations());  
context.close();  
}  
}
```

Output :

Infos sur la voiture : Voiture avec moteur Renault

Injection de dépendances

Il est important de noter que cette mécanique fonctionne car :

- La propriété `moteur` de cette classe est annotée `@Autowired`
 - ==> l'IoC container recherchera donc parmi les classes scannées une classe dont le type sera `Moteur`
- La classe `Moteur` est annotée comme `@Component`
 - ==> l'IoC container considérera donc cette classe comme éligible

Injection de dépendances

Si ce dernier point n'est pas respecté, alors la résolution automatique des dépendances ne fonctionnera pas.

L'loC container lèvera une exception à l'exécution pour signaler le problème.

Injection de dépendances

```
package exemples.spring.formation;
```

```
public class Moteur {  
    public String getManufacturer() {  
        return "Renault";  
    }  
}
```

Injection de dépendances

AVERTISSEMENT: **Exception encountered during context initialization** - cancelling refresh attempt: org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean with name 'voiture': **Unsatisfied dependency expressed through field 'moteur'**; nested exception is

org.springframework.beans.factory.NoSuchBeanDefinitionException: No qualifying bean of type 'exemples.spring.formation.Moteur' available: expected at least 1 bean which qualifies as autowire candidate. Dependency annotations:

{@org.springframework.beans.factory.annotation.Autowired(**required=true**)}

Exception in thread "main"

org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean with name 'voiture': Unsatisfied dependency expressed through field 'moteur'; nested exception is org.springframework.beans.factory.NoSuchBeanDefinitionException: No qualifying bean of type 'exemples.spring.formation.Moteur' available: expected at least 1 bean which qualifies as autowire candidate. Dependency annotations:

{@org.springframework.beans.factory.annotation.Autowired(required=true)}

Injection de dépendances

L'exception précise :

```
Dependency annotations: {  
org.springframework.beans.factory.annotation  
.Autowired(required=true) }
```

En effet, l'annotation `@Autowired` supporte un attribut booléen de nom `required`, qui permet d'indiquer si la dépendance doit obligatoirement être résolue ou pas.

Injection de dépendances

Essayons de rendre la résolution de la dépendance non obligatoire...

```
@Component("voiture")  
  
public class Voiture {  
    @Autowired(required = false)  
    private Moteur moteur;  
    ...  
}
```

Injection de dépendances

Résultat

Exception in thread "main" java.lang.**NullPointerException**

at

exemples.spring.formation.Voiture.**getInformations** (Voiture.java:13)

at

exemples.spring.formation.TestAutowired.main (TestAutowired.java:14)

Injection de dépendances

Effectivement, la résolution de la dépendance n'est plus obligatoire...

...mais cela ne nous arrange pas du tout, car l'erreur survient à **l'exécution**, pas à la compilation.

==> ceci est un problème à ne pas sous-estimer

Injection de dépendances

L'annotation Java standard équivalente est `@Inject`.

Cette annotation ne fournit pas l'attribut `required`.

Le même résultat peut cependant être obtenu au moyen d'un container `Optional`, utilisé pour encapsuler la propriété optionnelle (`moteur` dans cet exemple).

Injection de dépendances

```
@Component("voiture")

public class Voiture {

    @Inject

    private Optional<Moteur> moteur;

    public String getInformations() {
        return "Voiture avec moteur " +
            this.moteur.get().getManufacturer();
    }
}
```

Injection de dépendances

Résultat si la classe Moteur ne porte pas l'annotation
`@Component` :

```
Exception in thread "main" java.util.NoSuchElementException: No  
value present
```

```
at java.util.Optional.get(Optional.java:135)
```

```
at  
exemples.spring.formation.Voiture.getInformations (Voiture.java:17)
```

```
at  
exemples.spring.formation.TestAutowired.main (TestAutowired.java:14)
```

Injection de dépendances

L'annotation `@Autowired` peut aussi être utilisée sur un constructeur pour lever une ambiguïté quant au moyen d'injecter une dépendance.

Si un seul constructeur est défini pour une classe-propriété (comme `Moteur` dans l'exemple précédent), alors il n'y a pas d'ambiguïté, et l'annotation `@Autowired` est inutile.

Injection de dépendances

```
@Component("voiture")  
  
public class Voiture {  
    private Moteur moteur;  
  
    public Voiture(Moteur unMoteur) {  
        this.moteur = unMoteur;  
    }  
}
```

Injection de dépendances

```
@Component("voiture")  
  
public class Voiture {  
    private Moteur moteur;  
  
    public Voiture() {  
    }  
  
    @Autowired  
    public Voiture(Moteur unMoteur) {  
        this.moteur = unMoteur;  
    }  
}
```

Injection de dépendances

L'annotation `@Autowired` peut aussi être utilisée sur un setter ou toute autre méthode permettant d'injecter une dépendance.

Injection de dépendances

```
@Component("voiture")  
  
public class Voiture {  
    private Moteur moteur;  
  
    @Autowired  
  
    public void setMoteur(Moteur unMoteur) {  
        this.moteur = unMoteur;  
    }  
}
```

Injection de dépendances

Processus d'injection de dépendances en mode “autowiring”

L'IoC container va scanner les packages ou classes qui lui seront indiqués.

Il va ainsi enregistrer des informations sur les beans et leurs dépendances qu'il va découvrir.

Muni de ces informations, il va fournir les beans configurés qui lui seront demandés via une requête `getBean()`.

Il va d'abord chercher si un bean de type ou d'identifiant demandé est connu.

Injection de dépendances

Il va ensuite déterminer comment instancier ce bean (constructeur par défaut, constructeur unique ou constructeur annoté par `@Autowired`) et comment lui injecter ses dépendances.

Ce processus est bien sûr itératif car s'appliquant aussi aux dépendances du bean recherché.

Injection de dépendances

Dans l'exemple précédent, l'IoC container a noté que :

- La classe Voiture est un bean avec un constructeur par défaut

```
@Component
```

```
public class Voiture
```

- La classe Moteur est un bean avec un constructeur par défaut

```
@Component
```

```
public class Moteur
```

Injection de dépendances

Dans l'exemple précédent, l'IoC container aussi a noté que :

- Le bean `Voiture` a besoin d'un `Moteur`

```
public class Voiture {  
    private Moteur moteur;
```

- La classe `Voiture` possède un setter pour injecter la dépendance "moteur"

```
@Autowired  
public void setMoteur(Moteur unMoteur) {
```


Injection de dépendances

Compte tenu de ces informations, si on réclame à l'IoC container un bean de type `Voiture`, il va :

- Instancier ce bean au moyen du constructeur par défaut de `Voiture`
- Instancier un bean `Moteur` au moyen du constructeur par défaut de la classe
- Injecter ce `Moteur` au moyen du setter de la classe `Voiture`.

Injection de dépendances

Mise en pratique :

- Exercice 06 : injection de dépendances par annotation



Injection de dépendances

Annotation `@Qualifier`

Annotation spécifique de Spring (avec un équivalent très limité dans la JSR-330), permettant de **préciser** à l'IoC container quel bean particulier doit être choisi pour injecter une dépendance.

Cette annotation peut être utilisé sur les arguments d'un constructeur ou de toute méthode.

Injection de dépendances

Typiquement, on définit plusieurs beans au moyen d'un fichier XML de métadonnées, sans spécifier les dépendances de ces beans, et on utilise cette annotation `@Qualifier` pour préciser quel bean doit être utilisé comme dépendance d'un autre bean.

L'IoC container fera le travail de choisir le bean ainsi qualifié parmi ceux qu'il connaît (i.e. ceux qui ont été définis dans le fichier XML).

Injection de dépendances

Exemple : sélection d'un oiseau parmi un ensemble d'oiseau à placer dans une cage.

On définit plusieurs oiseaux au travers d'un fichier XML de métadonnées, on définit la cage dans un autre fichier XML, on définit un setter pour injecter un oiseau dans une instance de cage (le mettre en cage), et on qualifie dans ce setter l'oiseau à placer dans la cage.

Et on récupère ensuite la cage dans le programme de test.

Injection de dépendances

@Component

```
public class Oiseau {  
    private String nom;  
    public Oiseau(String nom) {  
        this.nom = nom;  
    }  
    public String getNom() {  
        return this.nom;  
    }  
}
```

Injection de dépendances

```
@Component("cage")
public class Cage {
    private Oiseau oiseau;

    @Autowired
    public setOiseau(@Qualifier("titi") Oiseau oiseau) {
        this.oiseau = oiseau;
    }

    public String affiche() {
        return "L'oiseau est " + this.oiseau.getNom();
    }
}
```

Injection de dépendances

```
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class TestQualifier {

    public static void main(String[] args) {

        ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("cage.xml", "oiseaux.xml");

        Cage cage = (Cage) context.getBean("cage");

        System.out.println(cage.affiche());

        context.close();

    }

}
```


Injection de dépendances

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<beans ...>
```

```
    <bean id="cage" class="exemples.spring.formation.Cage">
```

```
    </bean>
```

```
</beans>
```

Injection de dépendances

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<beans ...>
```

```
  <bean id="oiseau1" class="exemples.spring.formation.Oiseau">
```

```
    <constructor-arg type=java.lang.String" value="Titi" />
```

```
    <qualifier value="titi" />
```

```
  </bean>
```

```
  <bean id="oiseau2" class="exemples.spring.formation.Oiseau">
```

```
    <constructor-arg type=java.lang.String" value="Toto" />
```

```
    <qualifier value="toto" />
```

```
  </bean>
```

Injection de dépendances

```
<bean id="oiseau3" class="exemples.spring.formation.Oiseau">  
    <constructor-arg type="java.lang.String" value="Tutu" />  
    <qualifier value="tutu" />  
</bean>  
</beans>
```

Injection de dépendances

Output :

```
L'oiseau est Titi
```

Injection de dépendances

A noter : l'**identifiant** d'un bean sert de **qualificateur** du bean, en l'absence d'un autre qualificateur.

Injection de dépendances

On modifie le fichier `oiseaux.xml` :

```
<beans ...>

    ...

    <bean id="oiseau3" class="exemples.spring.formation.Oiseau">
        <constructor-arg type=java.lang.String" value="Tutu" />
        <!-- <qualifier value="tutu" /> -->
    </bean>
</beans>
```

Injection de dépendances

On modifie le fichier `Cage.java` :

...

```
@Autowired
```

```
    public void setOiseau(@Qualifier("oiseau3")  
Oiseau oiseau) {  
        this.oiseau = oiseau;  
    }
```

...

Injection de dépendances

Output :

L'oiseau est Tutu

Injection de dépendances

Retour sur l'injection d'un ensemble d'objets.

Exemple précédent : injection d'une liste de String.

Ce serait sympa de pouvoir sélectionner les objets à insérer comme dépendances dans le bean englobant...

N'avons-nous pas vu une annotation qui pourrait servir à cela ?



Injection de dépendances

L'annotation `@Qualifier` a pour but d'associer une sémantique à certains beans (plutôt que de définir un identifiant de bean comme on l'a fait précédemment).

Cela permet de sélectionner des beans au travers de leur sémantique plutôt que de leur identifiant.

Exemple : la sélection d'élèves dans un collège en fonction de leur genre.

Injection de dépendances

Dans cet exemple, on souhaite définir une liste d'élèves d'un collège, et remplir deux propriétés du collège contenant les références des élèves selon leur genre.

Si les élèves sont Adam, Beatrice, Charlotte, Damien, Enzo et Fanny, on souhaite que le collège contienne Adam, Damien et Enzo dans la propriété "élèves garçons" et Beatrice, Charlotte et Fanny dans la liste "élèves filles".

On peut associer un `@Qualifier` décrivant le genre aux beans représentant les élèves.

Injection de dépendances

NB. On va utiliser la balise

```
<context:component-scan base-package="xxx" />
```

pour indiquer à l'IoC container :

- De scanner le package `xxx` pour ainsi définir les beans qu'il trouvera au moyen des annotations
- D'activer les annotations sur les beans trouvés.

Ceci nous évitera de définir le bean `College` dans le fichier XML de métadonnées.

Injection de dépendances

De plus, nous utiliserons l'annotation `@Autowired` sur des setters pour injecter les beans **qualifiés** comme dépendances du `College`, ce qui nous évitera de décrire le contenu du `College` dans le fichier XML de métadonnées.

Injection de dépendances

```
<beans>
```

```
  <context:component-scan base-package="exemples.spring.formation" />
```

```
  <bean class="exemples.spring.formation.Eleve">
```

```
    <constructor-arg type="java.lang.String" value="Adam" />
```

```
    <qualifier value="garçon" />
```

```
  </bean>
```

```
...
```

Injection de dépendances

```
<bean class="exemples.spring.formation.Eleve">
```

```
    <constructor-arg type="java.lang.String" value="Beatrice" />
```

```
    <qualifier value="fille" />
```

```
</bean>
```

```
...
```

```
</beans>
```

Injection de dépendances

```
package exemples.spring.formation;

...

@Component

public class College {

    private List<Eleve> garcons;

    @Autowired

    public void setGarcons(@Qualifier("garçon") List<Eleve>
eleves) {

        this.garcons = eleves;

    }
}
```


Injection de dépendances

```
private List<Eleve> filles;  
  
@Autowired  
  
public void setFilles(@Qualifier("fille") List<Eleve>  
eleves) {  
  
    this.filles = eleves;  
  
}
```

Injection de dépendances

```
public void affiche() {  
    System.out.println("Eleves garçons dans le collège :");  
    for (Eleve eleve: this.garcons) {  
        System.out.println("Eleve : " + eleve.getNom());  
    }  
    System.out.println("\nEleves filles dans le collège :");  
    for (Eleve eleve: this.filles) {  
        System.out.println("Eleve : " + eleve.getNom());  
    }  
}  
}
```

Injection de dépendances

```
public class TestCollege {  
    public static void main(String[] args) {  
        ClassPathXmlApplicationContext context = new  
ClassPathXmlApplicationContext("college.xml");  
  
        College elevesDuCollege = (College)  
context.getBean(College.class);  
  
        elevesDuCollege.affiche();  
  
        context.close();  
    }  
}
```

Injection de dépendances

Output :

Eleves garçons dans le collège :

Eleve : Adam

Eleve : Damien

Eleve : Enzo

Eleves filles dans le collège :

Eleve : Beatrice

Eleve : Charlotte

Eleve : Fanny

Injection de dépendances

Mise en pratique :

- Exercice 07 : injection de collections



Injection de dépendances

Annotation `@Resource`

Cette annotation standard Java (JSR-250) permet d'injecter directement un bean identifié par :

- Son nom
- Ou son type si un nom n'est pas spécifié
- Ou un qualificateur (`@Qualifier`)

Injection de dépendances

Exemple : la classe contenante (match par nom).

```
public class PiloteVoiture {  
  
    @Resource(name="chassis")  
    private Chassis chassis;  
  
    @Resource(name="moteur")  
    private Moteur moteur;  
  
    public String id() {  
        return "Chassis : " + chassis.id() + ", moteur : " + moteur.id();  
    }  
}
```

Injection de dépendances

Exemple : le fichier XML de métadonnées (match par nom).

```
<beans ...>
```

```
    <context:annotation-config />
```

```
    <bean id="moteur"  
class="exemples.spring.formation.Moteur" />
```

```
    <bean id="chassis"  
class="exemples.spring.formation.Chassis" />
```

```
    <bean id="piloteVoitureBean"  
class="exemples.spring.formation.PiloteVoiture" />
```


Injection de dépendances

Exemple : la classe contenante (match par type).

```
public class PiloteVoiture {  
  
    @Resource  
    private Chassis chassis;  
  
    @Resource  
    private Moteur moteur;  
  
    public String id() {  
        return "Chassis : " + chassis.id() + ", moteur : " + moteur.id();  
    }  
}
```

Injection de dépendances

Exemple : le fichier XML de métadonnées (match par type).

```
<beans ...>
```

```
    <context:annotation-config />
```

```
    <bean class="exemples.spring.formation.Moteur" />
```

```
    <bean class="exemples.spring.formation.Chassis" />
```

```
    <bean id="piloteVoitureBean"  
class="exemples.spring.formation.PiloteVoiture" />
```

Injection de dépendances

NB. Comme d'habitude, cette annotation `@Resource` peut être placée sur une propriété (comme dans l'exemple précédent) ou sur un setter.

Et on peut bien sûr utiliser un `AnnotationConfigApplicationContext` comme contexte applicatif pour récupérer les beans, à condition d'annoter les beans avec `@Component` pour les indiquer comme tels au contexte applicatif.

Injection de dépendances

Annotation @Value

Spring fournit cette annotation pour indiquer une valeur à injecter.

Cette valeur peut être statique :

```
@Value("Lotus")
```

```
public String manufacturer;
```

```
@Value("Exige 240 R")
```

```
public String model;
```

Injection de dépendances

Une valeur statique ne présente que peu d'intérêt.

Mais cette valeur peut aussi être dynamique et recherchée dans un **fichier de propriétés**, une **ressource Spring**, une **propriété système**, une **variable d'environnement**, voire le résultat de l'évaluation d'une **expression SpEL** (Spring Expression Language).

On reviendra plus loin sur le concept de ressource Spring.

Par manque de temps, on ne détaillera pas par contre le langage SpEL.

Injection de dépendances

Rappel (normalement...).

Java fournit accès à deux ensembles de propriétés :

- Les propriétés système

```
System.getProperties()
```

- Les variables d'environnement

```
System.getenv()
```

Spring fournit automatiquement accès à ces deux ensembles de propriétés.

Injection de dépendances

De telles propriétés peuvent être mentionnées dans une annotation `@Value.`, en plaçant la clé entre `${` et `}`.

```
@Value("${java.class.path}")    // propriété Java
```

```
String classPath;
```

```
@Value("${java_home}")          // variable d'environnement
```

```
String javaHome;
```

```
@Value("${username}")           // variable d'environnement
```

```
String userName;
```

Injection de dépendances

Dans le cas où la propriété ne peut être trouvée, on peut spécifier une valeur par défaut; il suffit de faire suivre la valeur indiquée par le caractère : suivi de la valeur par défaut.

```
@Value("${ma.variable.inconnue:la valeur par défaut}")
```

```
String valeurInconnue;
```


Injection de dépendances

Il est aussi possible d'ajouter le contenu d'un ou de plusieurs fichiers de propriétés aux valeurs connues, au moyen de l'annotation `@PropertySource` placée sur un bean.

```
@PropertySource("conf/mes_propriétés.txt")
```

```
@Component
```

```
public class Systeme {
```

```
    ...
```

```
}
```

Injection de dépendances

Quelques mots sur SpEL.

SpEL est un langage puissant pouvant être mis en oeuvre dans des annotations `@Value` mais aussi dans l'attribut `value` de balises `property` dans un fichier XML de métadonnées, dont un **interpréteur** peut être **invoqué par programme**, et qui peut même être **compilé à l'exécution**.

SpEL permet entre autres l'appel de méthodes statiques.

Injection de dépendances

Dans une annotation `@Value`, les valeurs obtenues par évaluation d'une expression SpEL sont indiquées en remplaçant le caractère `$` dans le paramètre de l'annotation par un caractère `#`.

```
@Value("#{<expression SpEL>}")
```

Injection de dépendances

Exemple d'une expression SpEL qui génère une valeur aléatoire flottante dans l'intervalle [0 ; 100.0[par appel à la méthode `random()` de la classe `java.lang.Math`.

```
<bean id="generateur" class="...">
    <property name="aleatoire"
        value="#{ T(java.lang.Math).random() * 100.0}" />
</bean>
```

Injection de dépendances

Mise en pratique :

- Exercice 08 : injection de valeurs de propriétés



Injection de dépendances

Configuration « java-based » de l'IoC container

Injection de dépendances

On a vu qu'au moyen des annotations il est possible de :

- **Définir** des objets comme étant des beans
 - `@Component`, `@Controller`, `@Named`, `@Resource`, **etc.**
- Décrire **comment** injecter automatiquement les dépendances
 - `@Autowired`, `@Inject`
- **Sélectionner** les dépendances à injecter
 - `@Qualifier`
- Indiquer quelles **valeurs** injecter
 - `@Value`

Injection de dépendances

On a aussi vu qu'au moyen d'un fichier XML de métadonnées, on peut spécifier les dépendances, choisir les beans à injecter et surtout **faire créer** des instances de bean.

Autrement dit :

- Avec les annotations Java, on peut **décrire le système** en terme de composants (beans) et en termes de dépendances
- Avec le fichier XML de métadonnées, on peut en plus **faire créer le système** au moyen de l'instanciation de beans.

Injection de dépendances

Il y a une autre manière de **faire créer** des instances de bean : en décrivant des classes **porteuses de méthodes** pour créer des beans (des **méthodes-factories**, en quelque sorte).

On parle alors de « **Java-based container configuration** ».

Cette description de ces classes de configuration est faite en Java au moyen de l'annotation `@Configuration`.

La description de ces méthodes-factories est faite au moyen de l'annotation `@Bean`.

Injection de dépendances

@Configuration

```
public class ConfigProjet {  
  
    @Bean  
  
    public Service1 creeService1() {  
        return new Service1();  
    }  
  
    @Bean  
  
    public Service2 creeService2() {  
        return new Service2();  
    }  
  
}
```

Injection de dépendances

Les beans eux-mêmes n'ont pas besoin de porter d'annotation les spécifiant comme des beans à partir du moment où la méthode portant l'annotation `@Bean` indique leur type comme valeur de retour.

Par contre, si l'on veut que les dépendances soient injectées automatiquement, l'annotation `@Autowired` doit être indiquée sur les propriétés à injecter ou sur les setters de ces propriétés.

Injection de dépendances

NB. Le nom de la méthode annotée `@Bean` est l'**identifiant** de ce bean.

```
@Configuration  
  
public class ConfigProjet {  
    @Bean  
    public UnBean toto() {  
        return new UnBean();  
    }  
}
```



Injection de dépendances

```
class Testeur {  
    public static void main(String arguments) {  
        ApplicationContext contexte = ...;  
        UnBean bean = (UnBean) contexte.getBean("toto");  
        ...  
        contexte.close();  
    }  
}
```

Injection de dépendances

Les dépendances d'un bean sont déclarées au travers des paramètres de la méthode-factory.

Le mécanisme de résolution des dépendances mis en oeuvre est équivalent à celui utilisé pour injecter les dépendances au moyen d'un constructeur.

```
@Bean
```

```
public Service1 monService(Dependance1 dep1,  
    Dependance2 dep2) {
```

```
    ...
```

```
}
```

Injection de dépendances

Les dépendances d'un bean peuvent aussi être déclarées tout simplement par création du bean-dépendance dans une méthode-factory.

```
@Bean
```

```
public Service1 monService() { return new Service1(dep1()); }
```

```
@Bean
```

```
public Dependence1 dep1() {  
    return new Dependence1();  
}
```

Injection de dépendances

Le contexte applicatif à utiliser pour supporter la configuration IoC basée sur Java est l'`AnnotationConfigApplicationContext`.

Pour préciser au contexte applicatif la ou les classes de configuration à utiliser, il suffit de passer cette ou ces classes(s) en argument au contexte applicatif.

```
ApplicationContext contexte = new  
AnnotationConfigApplicationContext(Service1Impl.class  
, Service2Impl.class);
```


Injection de dépendances

On peut aussi instancier

`l'AnnotationConfigApplicationContext` sans argument et enregistrer ensuite les classes de configuration au moyen de sa méthode `register()`.

```
ApplicationContext contexte = new  
AnnotationConfigApplicationContext();  
  
contexte.register(Service1Impl.class, Service2Impl.class);
```

Injection de dépendances

Enfin, on peut indiquer par l'annotation

`@ComponentScan` (`basePackages = "..."`) quels sont les packages à scanner pour déterminer automatiquement les beans que contient l'application.

```
@Configuration
```

```
@ComponentScan(basePackages = "racine.des.packages")
```

```
public class ConfigApplication {
```

```
    ...
```

```
}
```

Injection de dépendances

C'est l'équivalent par annotation de la directive de configuration suivante, placée dans un fichier XML de métadonnées.

```
<beans>
```

```
    <context:component-scan base-package="..." />
```

```
</beans>
```

Ou encore :

```
contexte.scan();
```

```
contexte.refresh();
```

Injection de dépendances

Il est possible d'annoter une classe de configuration avec une ou plusieurs indications `@PropertySource` pour indiquer qu'un ou plusieurs fichier(s) de propriétés est(sont) à utiliser.

```
@Configuration
```

```
@PropertySource("resources/fichier.txt")
```

```
public class ... {
```

```
    ...
```

```
}
```

Injection de dépendances

Mise en pratique :

- Exercice 09 : configuration de container basée sur Java



Injection de dépendances

Quels sont les arguments pour ou contre l'utilisation d'annotations Spring plutôt que l'utilisation d'annotations standard Java ?



Injection de dépendances

Et caetera

Injection de dépendances

L'IoC container Spring gère beaucoup d'autres choses :

- Conversions de valeurs de propriétés complexes
 - Par exemple pour définir des datasources
 - Avec possibilité de définir ses propres classes de conversion
- Collections diverses
 - Set, List, Map
- Beans internes
- Directives pour forcer l'ordre d'initialisation de beans

Injection de dépendances

L'IoC container Spring gère beaucoup d'autres choses :

- Customisation du cycle de vie des beans
 - Les annotations JSR-250 `@PostConstruct` et `@PreDestroy` sont cependant préférables
- Scope des beans définissable
 - Singleton, Prototype ou autres si application web
 - Request, Session, Application, WebSocket
- Gestion de l'héritage de la configuration d'un bean parent
- Initialisation tardive de beans (mode `lazy`)

Injection de dépendances

L'IoC container Spring gère beaucoup d'autres choses :

- Spécification fine de la configuration d'une application au travers du langage Java
 - Choix du scope des beans
 - Gestion customisée du cycle de vie des beans
 - Enregistrement de la description des beans
 - Définition d'alias de beans
 - Composition de configurations basées sur Java
 - Internationalisation
 - ...

Injection de dépendances

Pause culturelle : que sont les principes SOLID ?



Injection de dépendances

Les principes **SOLID** :

- **S**ingle responsibility
- **O**pen/closed principle
 - Open for derivation, closed for modification
- **L**iskov substitution principle
- **I**nterface definition
- **D**ependency injection