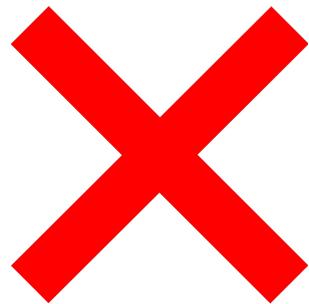




QU'EST CE QUE SPRING N'EST PAS?



Spring n'est pas un framework Web



SPRING ET LE WEB

Spring permet de créer des applications diverse et variée dans l'environnement **java**.

... Dont des **serveurs web**.

SPRING ET LE WEB

SpringBoot est le framework web.



MAIS

Avant de courir il faut savoir marcher.

Jarvis, il faut parfois savoir courir
avant de savoir marcher.

Iron Man (Tony Stark à Jarvis)

www.citation-celebre.com

QU'EST CE QUE SPRING ?

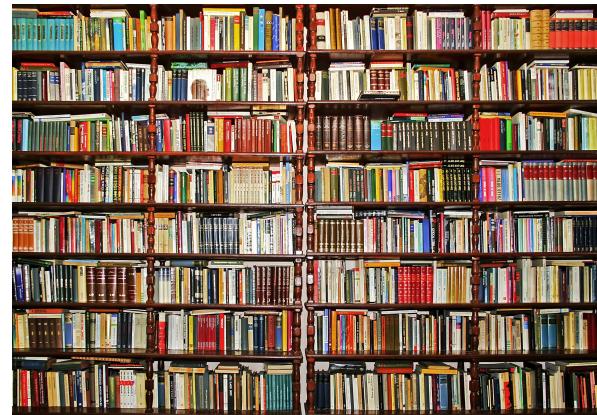
QU'EST CE QUE SPRING

Un framework Java open source



QU'EST CE QUE SPRING

- Une librairie pour l'**injection de dépendance**.



DES NOTIONS IMPORTANTES

SPRING ET IOC

Pour comprendre spring il faut connaître 2 notions:

- l'injection de dépendance ou IOD
- l'inversion de contrôle ou IOC
- la notion d'annotation en java

INJECTION DE DÉPENDANCES

L'**injection de dépendances** est une notion de programmation orientée objet. Elle stipule que les classes ne doivent pas dépendre des autres classes.

EXEMPLE

Vous voulez fabriquer une voiture. Vous avez besoin d'un moteur pour la faire fonctionner.

Quand vous créer une voiture, vous pouvez créer le moteur directement dans la classe voiture:

```
1 public class Voiture {  
2     private Moteur moteur = new Moteur();  
3  
4     public void rouler() {  
5         moteur.demarrer();  
6     }  
7 }
```

EXEMPLE

Dans ce cas, la classe voiture dépend de la classe moteur. Si vous voulez changer de moteur, vous devez changer la classe voiture.



```
1 public class Voiture {  
2     private Moteur moteur = new MoteurElectrique();  
3  
4     public void rouler() {  
5         moteur.demarrer();  
6     }  
7 }
```

DÉPENDANCE FORTE

On parle ici de **dépendance forte**. La classe voiture dépend de la classe moteur. Si vous voulez changer de moteur, vous devez changer la classe voiture.

Les **dépendances fortes** sont mauvaises car elles rendent le code difficilement maintenable.

SOLUTION

La solution est d'injecter le moteur dans la classe voiture en passant par le constructeur:

```
1 public class Voiture {  
2     private Moteur moteur;  
3  
4     public Voiture(Moteur moteur) {  
5         this.moteur = moteur;  
6     }  
7  
8     public void rouler() {  
9         moteur.demarrer();  
10    }  
11 }
```

SOLUTION

Dans ce cas, la classe voiture n'a plus besoin de connaître la classe moteur. Elle peut donc être utilisée avec n'importe quel moteur.



```
Voiture voitureElectrique = new Voiture(new MoteurElectrique());  
  
Voiture voitureEssence = new Voiture(new MoteurEssence());  
  
Voiture voitureHybride = new Voiture(new MoteurHybride());
```

INJECTION DE DÉPENDANCES

C'est le principe de l'**injection de dépendances**. On injecte les dépendances dans les objets.

Spring est un framework qui permet de faire de l'injection de dépendances!

IL Y A 2 TYPES D'INJECTION DE DÉPENDANCES

- L'injection par constructeur

```
1 public class Voiture {  
2     private Moteur moteur;  
3  
4     public Voiture(Moteur moteur) {  
5         this.moteur = moteur;  
6     }  
7  
8     public void setMoteur(Moteur moteur) {  
9         this.moteur = moteur;  
10    }  
11  
12    ...  
13 }
```

IL Y A 2 TYPES D'INJECTION DE DÉPENDANCES

- L'injection par constructeur
- L'injection par setter

```
1 public class Voiture {  
2     private Moteur moteur;  
3  
4     public Voiture(Moteur moteur) {  
5         this.moteur = moteur;  
6     }  
7  
8     public void setMoteur(Moteur moteur) {  
9         this.moteur = moteur;  
10    }  
11  
12    ...  
13 }
```

19.1

IL Y A 2 TYPES D'INJECTION DE DÉPENDANCES

- L'injection par constructeur
- L'injection par setter

```
1 public class Voiture {  
2     private Moteur moteur;  
3  
4     public Voiture(Moteur moteur) {  
5         this.moteur = moteur;  
6     }  
7  
8     public void setMoteur(Moteur moteur) {  
9         this.moteur = moteur;  
10    }  
11  
12    ...  
13 }
```

19.2

INJECTION DE DÉPENDANCES

La solution par **constructeur** est la plus utilisée car:

INJECTION DE DÉPENDANCES

La solution par **constructeur** est la plus utilisée car:

- Plus facile à tester

INJECTION DE DÉPENDANCES

La solution par **constructeur** est la plus utilisée car:

- Plus facile à tester
- Plus facile à maintenir

INJECTION DE DÉPENDANCES

La solution par **constructeur** est la plus utilisée car:

- Plus facile à tester
- Plus facile à maintenir
- Plus facile à réutiliser

INVERSION DE CONTRÔLE

DÉFINITION

L'inversion de contrôle (inversion of control, IoC) est un patron d'architecture commun à tous les frameworks (ou cadre de développement et d'exécution).

Il fonctionne selon le principe que le flot d'exécution d'un logiciel n'est plus sous le contrôle direct de l'application elle-même mais du framework ou de la couche logicielle sous-jacente.

Exemple(Sans inversion de contrôle): Un virement

```
public void virement(int c1, int c2, double montants) {  
    // Création d'une transaction  
    EntityTransaction transaction = entityManager.getTransaction();  
  
    // Démarrer la transaction  
    transaction.begin();  
    try{  
        retirer(c1, montant);  
        ajouter(c2, montant);  
  
        // Valider la transaction  
        transaction.commit();  
    } catch(Exception e){  
        // Annuler la transaction en cas d'exception  
        transaction.rollback();  
    }  
}
```

Mauvaise pratique:

- Modification du code source
- changement du code lourd pour de futures implémentations

Avec l'inversion de contrôle:

```
@Transactional // Annotation pour délégué la transaction à spring
public void virement(int c1, int c2, double montants) {
    retirer(c1, montant);
    ajouter(c2, montant);
}
```

Permet:

- De ne plus modifier le code source
- D'avoir un code plus léger et souple

LES ANNOTATIONS JAVA



QU'EST CE QU'UNE @annotation

Une annotation est:

- Un marqueur sur un élément de code.
- N'exécute pas de code.
- Permet de retrouver des éléments
- Permet une abstraction

EQUIVALENCE

Les annotations en java sont comparables aux **décorateurs** en python ou typescript.

Mais elles n'ont pas réellement le même fonctionnement.

CRÉER SON ANNOTATION

Créer une annotation est très simple. Il suffit de créer une interface avec le mot clé `@interface`:

```
public @interface MonAnnotation {  
}
```

UTILISATION DE L'ANNOTATION

En java il est possible d'utiliser **les @annotations** sur:

- Une classe ou une interface
- Une méthode ou un controller
- Un paramètre
- Un attribut.

SUR UNE CLASSE

```
@MonAnnotation  
class MaClass {  
}
```

SUR UNE MÉTHODE

```
class MaClass {  
    @MonAnnotation  
    public void maMethode() {  
    }  
}
```

SUR UN PARAMÈTRE

```
class MaClass {  
    public void maMethode(@MonAnnotation String param) {  
    }  
}
```

SUR UN ATTRIBUT

```
class MaClass {  
    @MonAnnotation  
    private String attribut;  
}
```

[AVANCÉ] RÉCUPÉRATION

Il est possible de savoir si une annotation est présente sur un élément de code.

```
public class MaClass {  
    @MonAnnotation  
    private String attribut;  
  
    public static void main(String[] args) {  
        if (MaClass.class.isAnnotationPresent(MonAnnotation.class)) {  
            System.out.println("L'annotation est présente");  
        }  
    }  
}
```

[AVANCÉ] INTROSPECTION

Avec la librairie `Reflections` il est possible de retrouver l'ensemble des éléments annotés.

```
public class MaClass {  
    @MonAnnotation  
    private String attribut;  
  
    public static void main(String[] args) {  
        Reflections reflections = new Reflections("com.example");  
        Set<Class<?>> annotated = reflections.getTypesAnnotatedWith(MonAnnotation.class);  
    }  
}
```

ANNOTATION ET SPRING

Depuis la version 5 de Spring, nous utilisons les annotations pour le développement de nos applications.

PREMIER PROJET AVEC SPRING

Pour créer un projet avec `spring-core`, il faut utiliser les dépendances de spring core et spring contexte.

- `Spring Core` est la base de spring, il contient les annotations et les interfaces de base.
- `Spring Context` est le conteneur d'objets de spring, il permet de créer des beans et de les injecter dans d'autres beans. C'est lui qui permet de faire de l'injection de dépendances.

CRÉATION D'UN PROJET SPRING AVEC MAVEN

1. Créer un projet maven sans archétype particulier
2. Ajouter les dépendances de spring-core et spring-context dans le fichier pom.xml:

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>6.0.3</version>
</dependency>
```

CRÉATION D'UN PROJET SPRING AVEC GRADLE

1. Créer un projet gradle sans archétype particulier
2. Ajouter les dépendances de spring-core et spring-context dans le fichier build.gradle:

```
dependencies {  
    implementation 'org.springframework:spring-core:6.0.3'  
    implementation 'org.springframework:spring-context:6.0.3'  
}
```

LE CONTEXTE SPRING



En réponse à [@NotaBeneMovies](#)

contexte ?

CRÉATION D'UN CONTEXTE SPRING

Pour créer un contexte Spring, il faut utiliser la classe
AnnotationConfigApplicationContext:

```
class Main {  
    public static void main(String[] args) {  
        AnnotationConfigApplicationContext context = new AnnotationConfigApp  
        context.refresh();  
    }  
}
```

QU'EST CE QU'UN CONTEXTE SPRING?

Un contexte Spring est un conteneur d'objets. Il permet de créer des objets et de les injecter dans d'autres objets.

AJOUTER UN OBJET AU CONTEXTE SPRING

Pour ajouter un objet au contexte Spring, il faut utiliser la méthode
register:

```
class Main {  
    public static void main(String[] args) {  
        AnnotationConfigApplicationContext context = new AnnotationConfigApp  
        context.refresh();  
        context.register(Voiture.class);  
    }  
}
```

RÉCUPÉRER UN OBJET DU CONTEXTE SPRING

Pour récupérer un objet du contexte Spring, il faut utiliser la méthode
getBean:

```
class Main {  
    public static void main(String[] args) {  
        AnnotationConfigApplicationContext context = new AnnotationConfigApp  
        context.refresh();  
        context.register(Voiture.class);  
        Voiture voiture = context.getBean(Voiture.class);  
    }  
}
```

Spring va créer une instance de la classe Voiture automatiquement.

INJECTION DE DÉPENDANCES

Imaginons que la classe **Voiture** dépend de la classe **Moteur**:

```
public class Voiture {  
    private Moteur moteur;  
  
    public Voiture(Moteur moteur) {  
        this.moteur = moteur;  
    }  
  
    public void rouler() {  
        moteur.demarrer();  
    }  
}
```

INJECTION DE DÉPENDANCES

```
class Moteur {  
    public void demarrer() {  
        System.out.println("Le moteur démarre");  
    }  
}
```

INJECTION DE DÉPENDANCES

Si vous ajoutez la classe **Moteur** au contexte Spring, **Spring** va automatiquement injecter la classe **Moteur** dans la classe **Voiture**:

```
class Main {  
    public static void main(String[] args) {  
        AnnotationConfigApplicationContext context = new AnnotationConfigApp  
        context.refresh();  
        context.register(Voiture.class);  
        context.register(Moteur.class);  
        Voiture voiture = context.getBean(Voiture.class);  
        voiture.rouler();  
    }  
}
```

TERMINOLOGIE: BEAN

Un **bean** en anglais veut dire **chose**.

En **Spring**, un **bean** est un objet qui est géré par le contexte Spring.

Ici les classes **Voiture** et **Moteur** sont des beans.



LES BEANS

Les beans sont caractérisés par:

- Un Type (ici Voiture et Moteur)
- Un identifiant (ici voiture et moteur) qui doit être unique
- Une portée (ici singleton)

AJOUTER UN BEAN AU CONTEXTE SPRING

Il y a 3 façons d'ajouter un bean au contexte Spring:

- En utilisant la méthode `register`
- En utilisant l'annotation `@Bean`
- En utilisant l'annotation `@Component`

AJOUTER UN BEAN AU CONTEXTE SPRING AVEC REGISTER

La méthode `register` permet d'ajouter un bean au contexte Spring:

```
context.register(Voiture.class);
```

REGISTER

Il est possible d'ajouter plusieurs beans en une seule fois:

```
context.register(Voiture.class, Moteur.class);
```

REGISTER

Vous pouvez aussi ajouter un bean en lui définissant un identifiant:

```
context.registerBean(Voiture.class, "maVoiture");
Voiture voiture = context.getBean("maVoiture", Voiture.class);
```

AJOUTER UN BEAN AU CONTEXTE SPRING AVEC @COMPONENT

Une autre façon d'ajouter un bean au contexte Spring est d'utiliser l'annotation **@Component** sur la classe:

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

    public Voiture(Moteur moteur) {
        this.moteur = moteur;
    }

    public void rouler() {
        moteur.demarrer();
    }
}
```

@COMPONENT

L'annotation `@Component` permet de définir un bean dans le contexte Spring.

Pour que Spring puisse détecter l'annotation `@Component`, il faut ajouter l'annotation `@ComponentScan` sur la classe `Main`:

```
@ComponentScan
class Main {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApp
        Voiture voiture = context.getBean(Voiture.class);
        voiture.rouler();
    }
}
```

@COMPONENTSCAN

L'annotation `@ComponentScan` permet de scanner le package de la classe `Main` et de détecter les annotations `@Component`.

AJOUTER UN BEAN AU CONTEXTE SPRING AVEC @CONFIGURATION

La dernière façon d'ajouter un bean au contexte Spring est d'utiliser l'annotation `@Configuration` sur la classe:

```
@Configuration
public class Config {
    @Bean
    public Voiture voiture() {
        return new Voiture(moteur());
    }

    @Bean
    public Moteur moteur() {
        return new Moteur();
    }
}
```

@CONFIGURATION

Cette méthode demande l'utilisation de `@Configuration` sur la classe
`Main`:

L'annotation `@Configuration` permet de définir que cette classe est
une classe de configuration.

@BEAN

L'annotation `@Bean` permet de définir un bean dans le contexte Spring.

```
@Bean  
public Voiture voiture() {  
    return new Voiture(moteur());  
}
```

Ici `Spring` va créer un bean de type `Voiture` et d'identifiant `voiture` et l'ajouter au contexte Spring.

C'est la méthode `voiture()` qui est appelée pour créer l'instance de la classe `Voiture`.

QUAND UTILISER @CONFIGURATION

Il est préférable d'utiliser l'annotation `@Configuration` pour définir les beans.

Cela permet de séparer la configuration du code métier.

CAS OÙ IL Y A PLUSIEURS BEANS DU MÊME TYPE

Imaginons que vous avez plusieurs classes Voiture:

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

    public Voiture(Moteur moteur) {
        this.moteur = moteur;
    }

    public void rouler() {
        moteur.demarrer();
    }
}

@Component
public class VoitureDeCourse {
    private Moteur moteur;
```

CAS OÙ IL Y A PLUSIEURS BEANS DU MÊME TYPE

Si je veux récupérer une instance de la classe `VoitureDeCourse`, je peux utiliser la méthode `getBean`:

```
VoitureDeCourse voitureDeCourse = context.getBean(VoitureDeCourse.class);
```

Mais si je veux récupérer une instance de la classe `Voiture`, je ne peux pas utiliser la méthode `getBean`:

```
Voiture voiture = context.getBean(Voiture.class); // Retourne une erreur
```

CAS OÙ IL Y A PLUSIEURS BEANS DU MÊME TYPE

Quand **Spring** ne sait pas comment choisir le bean à retourner, il retourne une erreur.

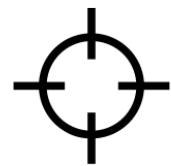
Pour résoudre ce problème, il faut ajouter l'annotation **@Primary** sur la classe **Voiture**:

```
@Component  
@Primary  
public class Voiture {  
    private Moteur moteur;  
  
    public Voiture(Moteur moteur) {  
        this.moteur = moteur;  
    }  
  
    public void rouler() {  
        moteur.demarrer();  
    }  
}
```

@PRIMARY

L'annotation `@Primary` permet de dire à `Spring` que cette classe est la classe par défaut.

LES SCOPES



CRÉER DEUX FOIS LE MÊME BEAN

```
@Component
public class MonBean {
    private static int compteur = 0;

    public MonBean() {
        compteur++;
        System.out.println("Création du bean n°" + compteur);
    }

    public void direBonjour() {
        System.out.println("Bonjour " + compteur);
    }
}
```

CRÉER DEUX FOIS LE MÊME BEAN

```
MonBean monBean1 = context.getBean(MonBean.class);
MonBean monBean2 = context.getBean(MonBean.class);
monBean1.direBonjour();
monBean2.direBonjour();
System.out.println(monBean1 == monBean2); // True
```

POURQUOI?

Par défaut, **Spring** crée une seule instance pour chaque type de bean.

On dit que le scope est **singleton**.

@SCOPE

L'annotation `@Scope` permet de changer le scope d'un bean.

Il y a plusieurs scopes:

- `singleton` (par défaut)
- `prototype` : Créer une nouvelle instance à chaque fois

@SCOPE

```
@Component
@Scope("prototype")
public class MonBean {
    private static int compteur = 0;

    public MonBean() {
        compteur++;
        System.out.println("Création du bean n°" + compteur);
    }

    public void direBonjour() {
        System.out.println("Bonjour " + compteur);
    }
}
```

@AUTOWIRED



@AUTOWIRED

L'annotation `@Autowired` permet de dire à `Spring` où injecter les dépendances. Et de lui demander de faire une injection de dépendance.

Elle peut être utilisée sur:

- Un constructeur quand il y a plusieurs constructeurs
- Un champ (même privé)
- Une méthode

@AUTOWIRED SUR UN CONSTRUCTEUR

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

    // Constructeur appelé par Spring
    @Autowired
    public Voiture(Moteur moteur) {
        this.moteur = moteur;
    }

    // Constructeur appelé par l'utilisateur mais pas par Spring
    public Voiture() {
    }
}
```

@AUTOWIRED SUR UN CHAMP

```
@Component
public class Voiture {
    @Autowired
    private Moteur moteur;

    public void rouler() {
        moteur.demarrer();
    }
}
```

Spring va injecter le bean `moteur` dans le champ `moteur`.

@AUTOWIRED SUR UNE MÉTHODE

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

    @Autowired
    public void setMoteur(Moteur moteur) {
        this.moteur = moteur;
    }

    public void rouler() {
        moteur.demarrer();
    }
}
```

@AUTOWIRED BONNES PRATIQUES

- Ne pas utiliser `@Autowired` sur les champs privés
- Utiliser `@Autowired` sur les constructeurs dans certains cas (quand il y a plusieurs constructeurs et qu'il faut en choisir un)

@QUALIFIER

L'annotation `@Qualifier` permet de préciser le nom du bean à injecter.

```
@Component
public class Voiture {

    private Moteur moteur;

    public Voiture(
        @Qualifier("moteurEssence") Moteur moteur
    ) {
        this.moteur = moteur;
    }

    public void rouler() {
        moteur.demarrer();
    }
}
```

Le code précédent est équivalent à:

```
@Component
public class Voiture {

    private Moteur moteur;

    public Voiture(
        Moteur moteurEssence
    ) {
        this.moteurEssence = moteurEssence;
    }

    public void rouler() {
        moteur.demarrer();
    }
}
```

EXERCICE 1:

Vous voulez créer un **burger** avec un **pain**, un **steak**, un **salade** et un **fromage**.

1. Créer les classes **Pain**, **Steak**, **Salade** et **Fromage**
2. Créer une classe **Burger** qui contient les 4 ingrédients
3. Ajouter toutes les classes dans le contexte de **Spring** avec
`context.register`
4. Récupérer le **burger** et afficher les ingrédients

EXERCICE 1.2:

1. Garder les mêmes classes que l'exercice 1.
2. Ajouter les classes avec l'annotation `@Component`.
3. Récupérer le `burger` et afficher les ingrédients

EXERCICE 1.3:

1. Garder les même classes que l'exercice 2
2. Supprimer l'annotation `@Component` des classes et basculer sur l'annotation `@Configuration` et `@Bean` pour l'ajout des beans.
3. Comment faire pour avoir plusieurs `burger` qui ont des ingrédients différents?

SUITE

Spring boot