



SPRING

Jérémy PERROUAULT



SPRING CORE

Introduction à SPRING

PRÉSENTATION DE SPRING

Framework, standard industriel

Facilite le développement et les tests

Spring gère des *JavaBean*, appelés *beans*

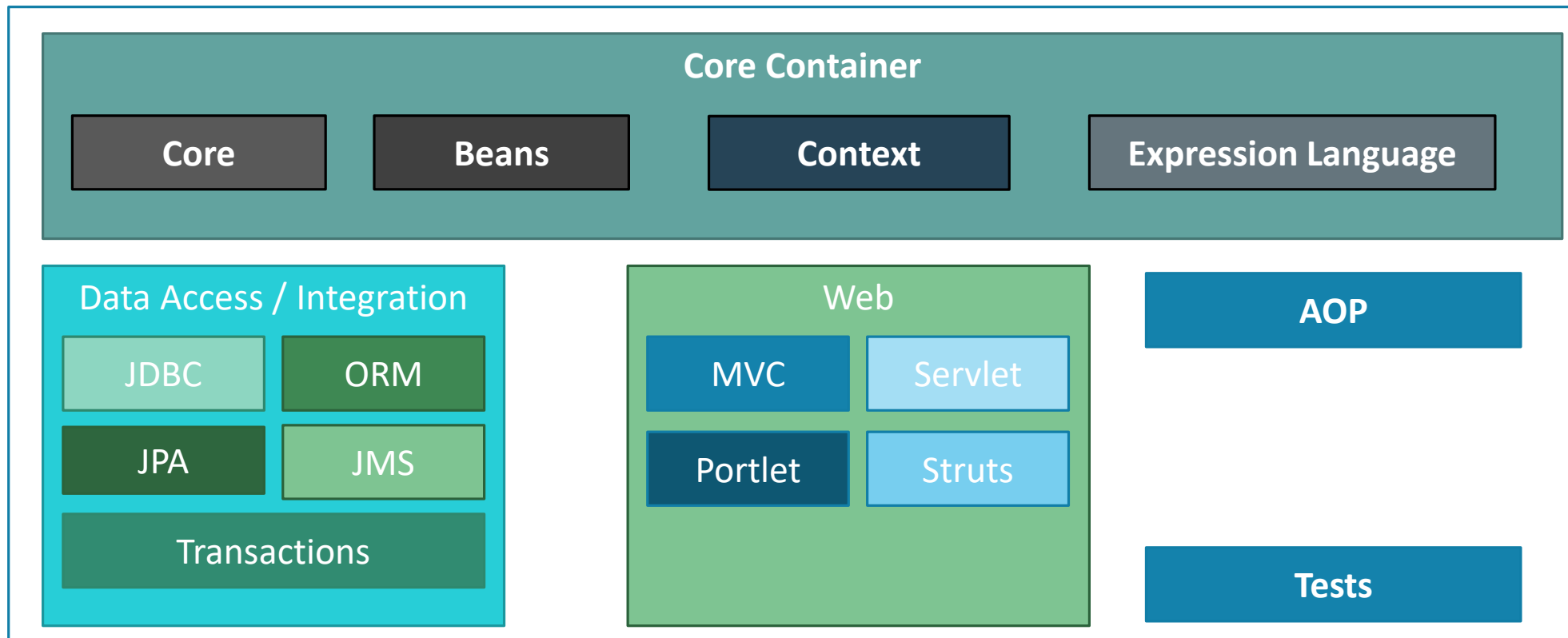
- Classes POJO

Fournit les mécanismes

- De fabrique d'objets (**BeanFactory**)
- D'inversion de contrôle (IoC – Inversion of Control) : l'injection de dépendances

PRÉSENTATION DE SPRING

Constitués de modules



PRÉSENTATION DE SPRING

Dépendance Maven

- **spring-context**

Maven chargera les dépendances de Spring-Core

INJECTION DE DÉPENDANCES

Dependency Injection (DI)

RAPPEL : On parle de dépendance entre objets lorsque

- **ClasseA** a un attribut de type **ClasseB**
- **ClasseA** est de type **ClasseB**, ou implémente *InterfaceB*
- **ClasseA** dépend d'un type **ClasseC** qui lui-même dépend d'un type **ClasseB**
- Une méthode de **ClasseA** appelle une méthode de **ClasseB**

Les interfaces permettent de nous abstraire de l'implémentation finale

- Mais nécessite toujours l'instanciation d'une classe concrète, l'instanciation de l'implémentation

INJECTION DE DÉPENDANCES

Soit les classes suivantes

```
public interface IMusicien {  
    public void jouer();  
}
```

```
public class Guitariste implements IMusicien {  
    private IInstrument instrument = new Guitare();  
  
    public void jouer() {  
        System.out.println("Le guitariste joue : " + this.instrument);  
    }  
}
```

```
public interface IInstrument { }
```

```
public class Guitare implements IInstrument {  
    public String toString() {  
        return "GLINK GLINK GLINK";  
    }  
}
```

- C'est fonctionnel, mais ce n'est pas bien parce que ce n'est pas son rôle d'instancier l'instrument !

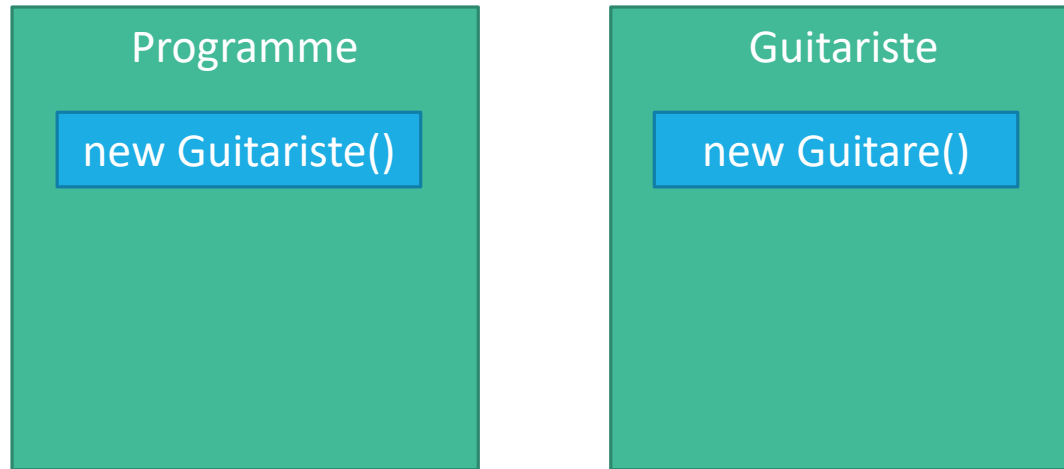
INJECTION DE DÉPENDANCES

Dans un programme principal

```
public class Principal {  
    public static void main(String[] args) {  
        IMusicien myMusicien = new Guitariste();  
        myMusicien.jouer();  
    }  
}
```

- C'est fonctionnel
 - Mais ce n'est pas bien parce que ce n'est pas son rôle d'instancier le musicien !

INJECTION DE DÉPENDANCES



- Le problème : chaque classe dépend d'une autre classe parce qu'elle doit l'instancier
 - **Principal** dépend de **Guitariste**
 - **Guitariste** dépend de **Guitare**
- Si on ne veut plus d'un **Guistariste** mais d'un **Pianiste**, il faudra changer les implémentations

INJECTION DE DÉPENDANCES

Pour résoudre ce problème, créons une Factory qui se chargera

- D'instancier l'instrument

```
public class InstrumentFactory {  
    public static IInstrument getInstrument() {  
        return new Guitare();  
    }  
}
```

INJECTION DE DÉPENDANCES

Créons une Factory qui se chargera

- D'instancier le musicien

```
public class MusicienFactory {  
    public static IMusicien getMusicien() {  
        return new Guitariste(InstrumentFactory.getInstrument());  
    }  
}
```

- *Remarque : on va donner au musicien sa dépendance à l'instrument, en le transmettant par son constructeur (on aurait eu une classe abstraite, on aurait pu le gérer au niveau supérieur)*

```
public class Guitariste implements IMusicien {  
    private IInstrument instrument;  
  
    public Guitariste(IInstrument instrument) {  
        this.instrument = instrument;  
    }  
}
```

INJECTION DE DÉPENDANCES

Dans notre programme principal, nous avons ceci

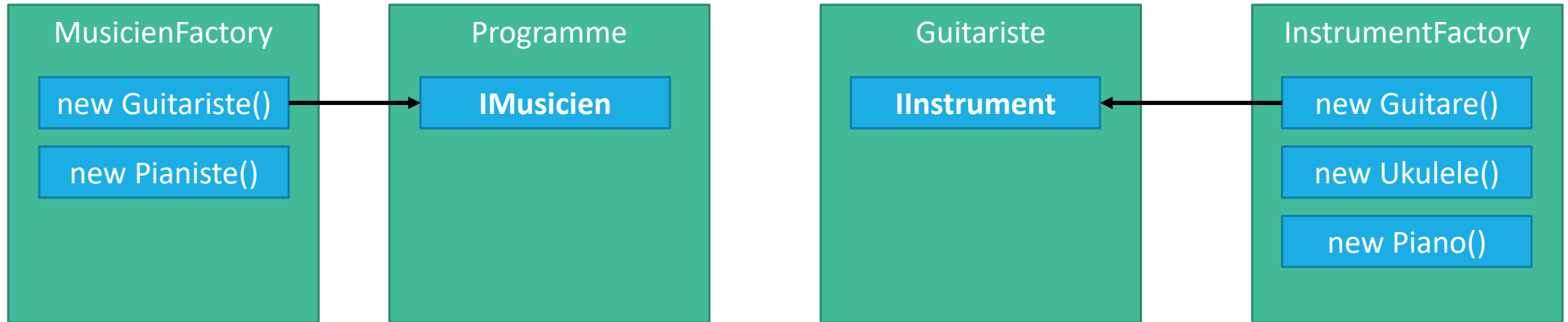
```
IMusicien myMusicien = new Guitariste();
```

Remplacé par ça

```
IMusicien myMusicien = MusicienFactory.getMusicien();
```

- La classe n'instancie plus, mais récupère sa dépendance via la Factory !

INJECTION DE DÉPENDANCES



- Nos classes ont toujours des dépendances, mais elles sont récupérées depuis les Factories
 - Si l'implémentation est à changer, on ne modifie que les Factories !

INJECTION DE DÉPENDANCES

Autre exemple, les DAO

```
public class Principal {  
    public static void main(String[] args) {  
        IDAOProduit daoProduit = new DAOProduitSQL();  
        daoProduit.findAll();  
    }  
}
```

Bien plus intéressant d'avoir une factory qui fabrique nos DAO

- Les implémentations se changent toutes dans la factory

```
public class Principal {  
    public static void main(String[] args) {  
        IDAOProduit daoProduit = DAOFactory.getDAOProduit();  
        daoProduit.findAll();  
    }  
}
```

INJECTION DE DÉPENDANCES

Si nous voulons changer les implémentations, seules les Factories sont à modifier

Les classes Musicien et Programme ne gèrent plus l'instanciation

- C'est ce qu'on appelle l'injection de dépendances, le pattern « Inversion of Control » (IoC)

C'est un travail fastidieux, en partie possible grâce à l'abstraction (interfaces)

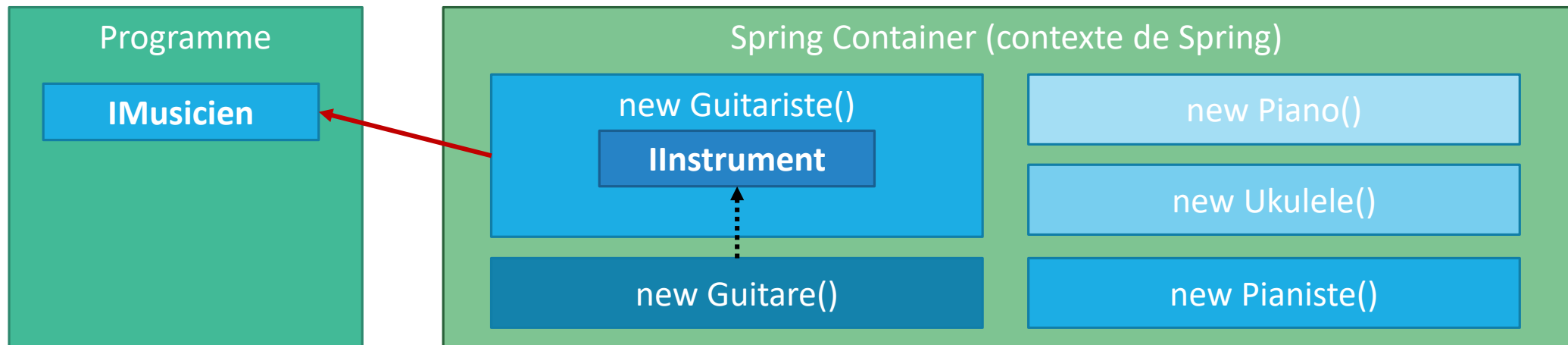
- C'est là que SPRING entre en jeu !

INJECTION DE DÉPENDANCES

Spring s'appuie sur le pattern *Inversion of Control* (IoC)

Permet de rendre indépendantes les couches techniques

- IoC se charge d'instancier et de donner la référence créée !



INJECTION DE DÉPENDANCES

L'injection de dépendances n'est possible que dans le contexte de Spring Container

- Les objets y ont accès par un quelconque moyen
- Les objets sont déjà dans le contexte de Spring

Spring ne peut nous injecter que des objets qu'il manage

- Depuis un programme principal (qui n'est pas géré par Spring)
 - Il faudra charger ce contexte pour récupérer les instances gérées par Spring

Spring injectera les dépendances après l'instanciation des objets

- Donc les références injectées ne sont pas disponibles dans le constructeur de l'objet

INJECTION DE DÉPENDANCES

Par chance, Spring ne va pas gérer toutes les instances de toutes les classes ...

Il faut lui préciser

- Quelle(s) instance(s) il doit manager
- Quelle instance injecter dans quel attribut

Ces précisions se font par déclaration (XML ou annotations)

Pour nous aider dans les étapes de configuration, Spring propose un plugin Eclipse

- Plugin *Spring IDE* (ou *Spring Tools*)



CONFIGURATION XML

Déclaration des *beans* en XML

CONFIGURATION XML

Pour configurer Spring avec XML

- Généralement, un fichier de configuration par utilité
 - Configuration générale
 - Configuration Web
 - Configuration API
 - Configuration de la sécurité
 - ...
- Fichier de configuration générale « application-context.xml » à placer dans *main/resources*

CONFIGURATION XML

Pour déclarer un *bean*, on précise son nom (son identifiant) et son type complet

Pour déclarer un *bean* nommé **guitare**, de type *fr.formation.instrument.Guitare*

```
<bean id="guitare" class="fr.formation.instrument.Guitare" />
```

CONFIGURATION XML

Pour injecter une dépendance dans un *bean*

- On utilise le setter des attributs, qu'on appelle ici « propriétés »

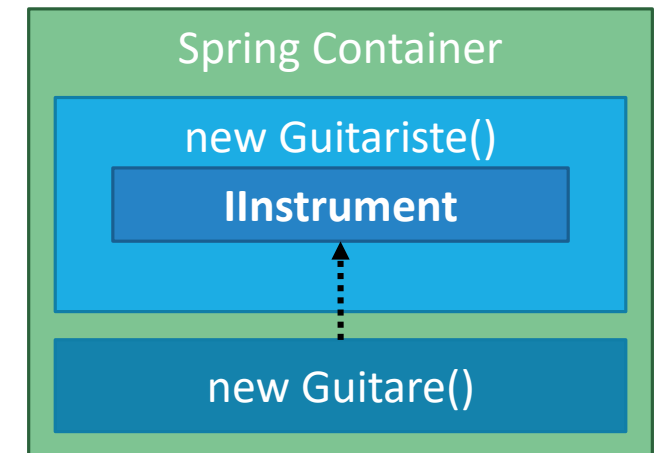
Pour injecter une référence « **guitare** » dans la propriété « instrument »

- On déclare le *bean* « **guitare** »
- On déclare le *bean* « **guitariste** » en précisant la référence du *bean* « **guitare** » pour sa propriété

```
<bean id="guitare" class="fr.formation.instrument.Guitare" />  
  
<bean id="guitariste" class="fr.formation.musicien.Guitariste">  
  <property name="instrument" ref="guitare" />  
</bean>
```

NOTE

Il faut au moins un setter sur l'attribut
L'ordre des déclarations n'a pas d'importance



CONFIGURATION XML

Pour charger le contexte Spring (XML) depuis une instance non managée par Spring

- Comme la classe du programme principal par exemple

```
ClassPathXmlApplicationContext myContext = new ClassPathXmlApplicationContext("classpath:application-context.xml");
```

Pour récupérer un *bean* dans le contexte Spring

- Par son nom

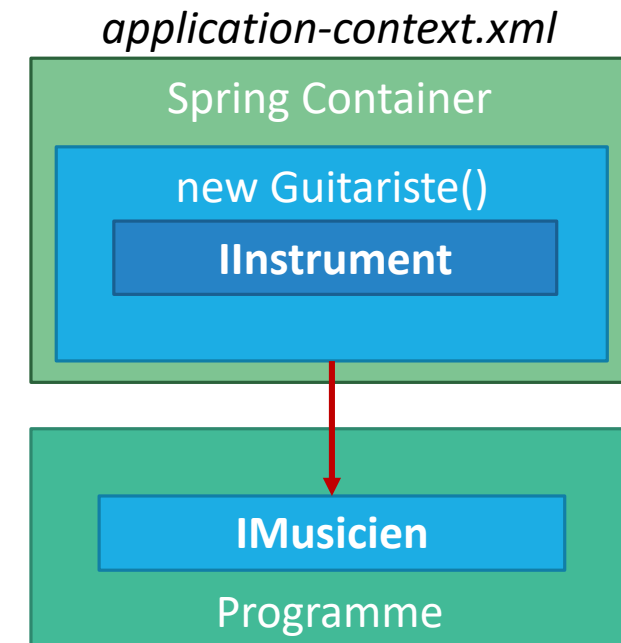
```
IMusicien myMusicien = (IMusicien)myContext.getBean("guitariste");
```

- Par son type

```
IMusicien myMusicien = myContext.getBean(IMusicien.class);
```

- Par son nom et son type

```
IMusicien myMusicien = myContext.getBean("guitariste", IMusicien.class);
```



EXERCICE

Installer Spring IDE (ou Spring Tools)

Créer un nouveau projet (Maven), « formation-spring »

Créer une classe principale **Principal** avec sa méthode *main*

Créer le modèle vu précédemment (Guitariste, Guitare et les interfaces)

Configurer Spring dans l'application

- application-context.xml
- (s'aider du complément HTML « Configuration SPRING »)

Faire jouer le guitariste !

Ne faire aucune instanciation!



DÉCLARATION PAR ANNOTATION

Configuration XML

DÉCLARATION PAR ANNOTATION

Déclaration d'un *bean*

- Annotations
 - **@Component**
 - **@Controller** / **@RestController**
 - **@Service**
 - **@Repository**

Injection d'un *bean* (instance managée par Spring uniquement)

- **@Autowired** (avec **@Qualifier("nom")** possible)
- **@Inject**
- **@Resource("nom")**

DÉCLARATION PAR ANNOTATION

Annoter les classes

Annotation	Définition	Cas d'utilisation
@Component	Composant Spring	Tous les cas, sauf ... (voir ci-dessous)
@Controller	Composant de type Controller	Point d'accès (comme les Servlets)
@RestController	Composant de type Controller	Point d'accès Service Web REST
@Repository	Composant de type Repository	Classe entrepôt (DAO par exemple)
@Service	Composant de type Service	Fournisseur de service

Par défaut, le nom du *bean* sera le nom de la classe (avec une minuscule)

- On peut modifier ce comportement en donnant une valeur aux annotations

DÉCLARATION PAR ANNOTATION

Pour injecter une référence gérée par Spring

- Utilisation de l'annotation **@Autowired** sur une propriété
- Pour que ce soit fonctionnel, il faut que toutes les références soient gérées par Spring

```
@Component
public class Guitariste implements IMusicien {
    @Autowired
    private IInstrument instrument;
}
```

- Ne fonctionnera que s'il y a un *IInstrument*
 - Annoté de **@Component**
 - Ou déclaré en tant que **bean** dans le fichier de configuration XML

DÉCLARATION PAR ANNOTATION

@Autowired retrouve une dépendance

- 1- Via le type de l'attribut ou de l'argument
- 2- Via le nom de l'attribut ou de l'argument
- Fonctionne très bien quand une seule référence est disponible, sinon :
 - Il faut utiliser **@Qualifier("nom")** pour retrouver la dépendance
 - Et/ou préciser **@Primary** (sur le bean) si on veut utiliser un bean spécifique en priorité

DÉCLARATION PAR ANNOTATION

Il faut que les classes annotées soient scannées par Spring

- Préciser les packages à scanner pour cette configuration

```
<context:component-scan base-package="fr.formation.musicien, fr.formation.instrument" />
```

- Spring va scanner le package et les sous-packages

EXERCICE

Remplacer la déclaration des *beans* XML en déclaration par annotation

EXERCICE

Reprendre l'exercice sur les musiciens

Cette fois-ci :

- Choisir un instrument pour le musicien
 - Pour le guitariste : Guitare ou Ukulele
 - Pour le pianiste : Piano ou Synthé
- Choisir qui doit jouer (le guitariste ou le pianiste)
 - Demander à l'utiliser de choisir le musicien qui doit jouer !

Utiliser l'annotation **@Qualifier("nom")** avec **@Autowired**

- Pour retrouver un *bean* par son nom



CONFIGURATION JAVA

Configuration et déclaration
JAVA

CONFIGURATION JAVA

Il est possible de configurer Spring au travers d'une classe de configuration

- Annotée de **@Configuration**
- Les packages scannés sont précisés dans l'annotation **@ComponentScan**
- Les *beans* créés sont annotés de **@Bean**

```
@Configuration
public class AppConfig {
    @Bean
    public IInstrument guitare() {
        return new Guitare();
    }

    @Bean
    public IMusicien guitariste() {
        return new Guitariste();
    }
}
```

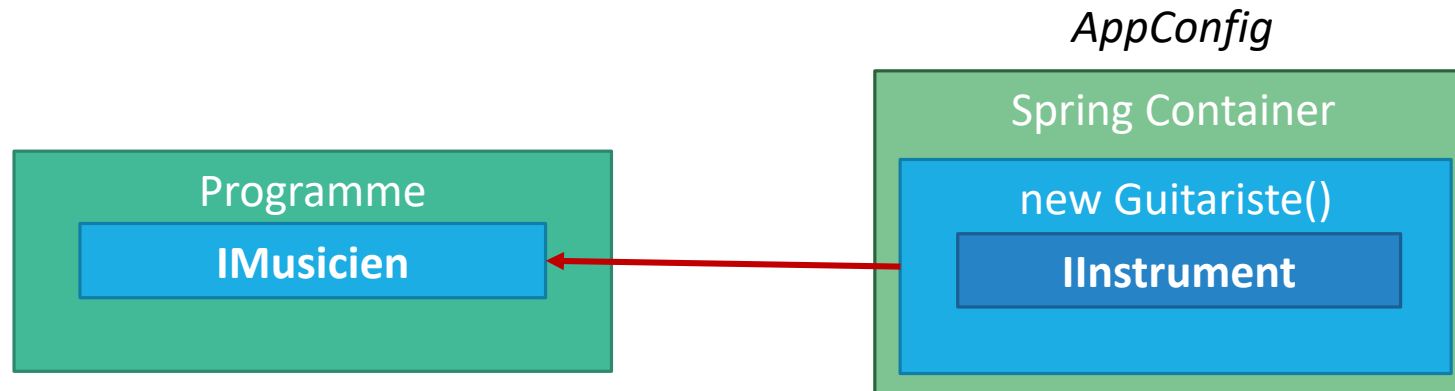
```
@Configuration
@ComponentScan({ "fr.formation.musicien", "fr.formation.instrument" })
public class AppConfig {
}
```

Dans ce cas, le nom du *bean* sera le nom de la méthode
On peut modifier ce comportement en donnant une valeur à l'annotation **@Bean**

CONFIGURATION JAVA

Pour charger cette configuration dans le programme principal

```
AnnotationConfigApplicationContext myContext = new AnnotationConfigApplicationContext(AppConfig.class);  
IMusicien myMusicien = myContext.getBean(IMusicien.class);
```



CONFIGURATION JAVA

On peut inclure un fichier de configuration XML pour une configuration par classe

- Annotation **@ImportResource**

```
@Configuration
@ImportResource("classpath:application-context.xml")
public class AppConfig {
}
```

On peut inclure une configuration par classe dans une configuration XML

- Déclaration de Annotation-Config
- Déclaration du *bean* de configuration (ou scan du package dans lequel se trouve ce *bean*)

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

```
<bean id="config" class="fr.formation.config.AppConfig" />
```

EXERCICE

Remplacer toute la configuration XML par de la configuration Java (par classe)

- Création d'une classe **AppConfig**
- Déclaration par annotation (scan des packages)



INJECTION DES DÉPENDANCES

Aller plus loin sur l'injection

INJECTION DE DÉPENDANCES

En résumé

- On déclare en tant que *bean* les dépendances par une approche déclarative (XML ou annotations)
- Toutes nos déclarations de dépendances sont au même endroit
- Spring injectera les dépendances après l'instanciation des objets
 - Donc les références injectées ne sont pas disponibles dans le constructeur de l'objet

INJECTION DE DÉPENDANCES — CONSTRUCTEUR

Au besoin, utiliser l'annotation **@PostConstruct** sur une méthode

- Elle s'exécutera **juste après** l'instanciation !

```
@Component
public class Guitariste implements IMusicien {
    @Autowired private IInstrument instrument;

    public Guitariste() {
        this.instrument; //Pas dispo
    }

    @PostConstruct
    public void init() {
        this.instrument; //Disponible
    }
}
```


INJECTION DE DÉPENDANCES — CONSTRUCTEUR

Ou utiliser **@Autowired** sur le constructeur

- (avec un argument du type de la dépendance)
- Dans ce cas, Spring cherchera à injecter la dépendance dès la construction de l'objet

```
@Component
public class Guitariste implements IMusicien {
    private IInstrument instrument;

    @Autowired
    public Guitariste(IInstrument instrument) {
        this.instrument = instrument;
    }
}
```



UNE APPLICATION SPRING ?

Créer une application dans le
contexte de Spring

APPLICATION DANS LE CONTEXTE DE SPRING

La classe principale ne peut pas être gérée par Spring

- L'utilisation de **@Autowired** est impossible
- Obligation de charger le contexte de Spring, et de récupérer les instances avec ce contexte ...

Opérons quelques modifications

1. Notre programme principal voit sa *main* devenir non statique, et renommée *run*
2. On ajoute une nouvelle classe **SpringApplication**, avec une méthode statique *main*
 1. C'est cette nouvelle classe qui devra être exécutée au démarrage

APPLICATION DANS LE CONTEXTE DE SPRING

Application **Principal**

- La méthode *main* est renommée *run*, et n'est plus statique
- Sera exécutée dans le contexte de Spring grâce à **SpringApplication**
 - **@Autowired** devient accessible !

```
public class Principal {  
    @Autowired  
    private IMusicien musicien;  
  
    public void run(String[] args) {  
        musicien.jouer();  
    }  
}
```

APPLICATION DANS LE CONTEXTE DE SPRING

SpringApplication

- Charge le contexte de Spring
- Crée une instance du programme principal dans ce contexte
- Appelle la méthode *run* du programme principal !
- Ferme le contexte de Spring

```
public class SpringApplication {  
    public static void main(String[] args) {  
        AnnotationConfigApplicationContext myContext = new AnnotationConfigApplicationContext(AppConfig.class);  
        myContext.getBeanFactory().createBean(Principal.class).run(args);  
        myContext.close();  
    }  
}
```

EXERCICE

Mettre en place la classe **SpringApplication**

Modifier la classe principale, en y ajoutant les **@Autowired** nécessaires !



CYCLES DE VIE

Les scopes et cycles de vie

CYCLES DE VIE

Par défaut, toutes les instances sont des instances *Singleton*

- Créées au démarrage du contexte de SPRING
 - (et par extension, souvent au démarrage de l'application)

Possible de modifier ce comportement avec l'annotation **@Scope**

- prototype *Une instance est créée à chaque demande*
- request *Une instance est créée par requête (contexte Web)*
- session *Une instance est créée par session (contexte Web)*

Si on souhaite rester en *Singleton*, mais empêcher l'instanciation au démarrage

- Utilisation de l'annotation **@Lazy** sur la classe ou le bean déclaré



PROFILES

Les profiles de configuration

PROFILES

Il est possible d'activer une configuration spécifique à un instant donné

- Environnement de développement
- Tests unitaires
- Environnement de test
- Environnement de production
- Activer une journalisation
- ...

Ceci se fait via l'annotation **@Profile**, sur la classe de configuration, ou sur les beans

PROFILES

Pour activer un profile, il suffit de démarrer l'application en spécifiant l'option

- -Dspring.profiles.active, exemple :
 - -Dspring.profiles.active=dev
- (Dans Eclipse, ajouter à *VMArguments* dans le configurateur des démarrages)



FICHIERS PROPERTIES

Variables dans fichier

CONFIGURATION AVEC PROPERTIES

Mise en place d'une configuration avec un fichier .properties (clé = valeur)

- Fichier main/resources/data-source.properties

```
musique.instrument = guitare
```

CONFIGURATION AVEC PROPERTIES

On précise à Spring d'aller récupérer ce fichier properties

```
<context:property-placeholder location="classpath:data-source.properties" />
```

On manipule les propriétés avec une SpEL (Spring Expression Language) \${ }

- Dans le fichier de configuration XML

```
<bean id="guitariste" class="fr.formation.musicien.Guitariste">  
  <property name="instrument" value="${musique.instrument}" />  
</bean>
```

- Sur l'attribut d'une classe

```
@Resource(name="${musique.instrument}")  
private IInstrument instrument;
```

```
@Value("${musique.instrument}")  
private String instrumentNom;
```

CONFIGURATION AVEC PROPERTIES

On précise à Spring le ou les fichiers properties avec l'annotation `@PropertySource`

Puis on injecte les propriétés dans un objet de type **Environment**

```
@Configuration
@PropertySource("classpath:data-source.properties")
public class AppConfig {
    @Autowired
    private Environment env;

    @Bean
    public Guitariste guitariste() {
        //env.getProperty("musique.instrument") ...
    }
}
```

EXERCICE

Mettre en place un fichier .properties

- Préciser quel musicien doit jouer
- Adapter le programme en conséquence



ASYNCHRONISATION

Des méthodes asynchrones

CONFIGURATION

Annoter la classe de configuration de **@EnableAsync**

UTILISATION

Annoter la méthode de **@Async**

- **ATTENTION**
 - Ne fonctionne que sur les classes managées par SPRING (beans)
 - Ne fonctionne que lors d'un appel d'une classe externe (Classes Proxy)

EXERCICE

Mettre en place une méthode asynchrone

- Dans une classe de *Service*
- Avec un temps de pause du **Thread** de quelques secondes
- Et une impression de messages dans la console

L'appeler depuis le programme principal



TÂCHES PLANIFIÉES

Des méthodes planifiées

CONFIGURATION

Annoter la classe de configuration de **@EnableScheduling**

UTILISATION

Annoter la méthode de **@Scheduled**

- Ne fonctionne que sur les classes managées par SPRING
- Nul besoin d'appeler cette méthode, SPRING le fait
- La méthode ne retourne rien

Options possibles

- `initialDelay` *Durée en millisecondes avant la première exécution*
- `fixedRate` *Durée en millisecondes entre une exécution et la suivante*
- `delay` *Durée en millisecondes entre la fin d'une exécution et le début la suivante*
- `cron` *Expression Cron*

EXERCICE

Mettre en place une tâche planifiée

- Dans une classe de *Service*
- Imprime du texte dans la console toutes les 2 secondes