TP Spring n°2
IoC et DI

Nous allons aborder dans ce TP avec Spring Framework le principe d'Inversion Of Control (**IoC**) et également le principe d'injection de dépendance (**DI**). Il s'agit d'un processus par lequel les objets définissent leurs dépendances (c'est-à-dire les autres objets avec lesquels ils travaillent) uniquement via les arguments du constructeur, ou par les attributs de la classe en passant par les méthodes get et set. Le conteneur injecte ensuite ces dépendances lorsqu'il crée le Bean.

Un **Bean** est une technologie de composants logiciels écrits en Java.
La spécification JavaBeans d'Oracle définit les composants de type JavaBeans comme « des composants logiciels réutilisables manipulables visuellement dans un outil de conception ».

Ils sont utilisés pour encapsuler plusieurs objets dans un seul objet : le « bean » (abréviation de coffee bean, soit « grain de café » en français). Le « bean » regroupe alors tous les attributs des objets encapsulés, et peut définir d'autres attributs si besoin. Ainsi, il représente une entité plus globale que les objets encapsulés de manière à répondre à un besoin métier.

Un Bean est défini par la convention suivante :

- la classe doit être « Serializable »
- la classe doit posséder un constructeur sans paramètre;
- les attributs privés de la classe doivent être accessibles publiquement via des méthodes get ou set suivi du nom de l'attribut avec la première lettre capitalisée.

1° - Création de service

Un service est un composant de Spring qui offre des fonctionnalités selon un type d'entité. On va donc créer un composant de type service afin de gérer un produit (ici une liste de produits).

1° - Création du package service dans lequel on va créer l'interface représentant les fonctionnalités du service ainsi que l'unique instance d'implémentation.

2° - Création de l'interface ProductService.
L'interface définit les méthodes, ici notre service permet de faire les choses suivantes :

```
public List<Product> getAllProducts();
public Product getProductById(Long id);
public Product save(Product product);
public boolean isProductAvailable(Product product, int quantity);
public void removeProduct(Product product, int quantity) throws StockException;
```

3° - Créez une nouvelle classe, *ProductServiceImpl*, dans ce package qui implémente l'interface. C'est une classe qui a pour but d'être instanciée qu'une seule fois et qui va contenir l'unique instance de l'arraylist de produits.
Elle a donc comme attribut :

```
private final List<Product> allProducts = new ArrayList<>();
```

4° - Implémentez les méthodes de l'interface :

- La méthode *getAllProducts* retourne la liste des produits.
- La méthode *save* ajoute le produit à la liste.
- Pour la méthode *getProductById*, si elle ne trouve pas le produit, elle lève une exception de type *ResourceNotFoundException*. Pour cela, créez dans un package *exception* la classe *ResourceNotFoundException* qui hérite de RuntimeException (pas obligatoire à vérifier).
- La méthode *removeProduct* lève une exception de type *StockException* (créez StockException dans le package exception, elle extends de Exception).

Nous avons maintenant un Service défini par une interface qui nous permet de manipuler notre modèle de Product.

2° - Metadata : fichier de configuration

Le fichier de configuration metadata est au format xml. Ce fichier définit la manière dont les composants vont être instanciés. C'est ici le point d'inversion de contrôle (IoC), les classes vont être instanciées "comme par magie". Nous n'allons jamais manipuler l'instance de la classe que l'on a créée. Cette instance sera créée et manipulée par les autres classes du framework, ça sera transparent pour nous.

On dit seulement ce que l'on veut faire et comment mais on ce n'est pas nous qui appelons cette fonctionnalité, nous n'avons pas le contrôle (IoC).

Nous avons pour l'instant un seul composant c'est le fichier *ProductServiceImpl* qui doit être instancié (une fois) afin de contenir l'unique instance contenant la liste des produits. Cette instance gère cette liste avec les fonctionnalités de lister, trouver un item, ajouter un nouvel item. Le fichier de configuration déclare ce Bean via la déclaration XML d'un élément <bean/>.

1° - Créez ce fichier de la façon suivante, nommez le services.
New > Other > Spring > Spring Bean Configuration File > Next
File name : services
Placez ce fichier dans le dossier java de votre projet.

Attention si vous n'avez pas le dossier Spring lorsque vous faites New > Other, ajoutez à votre eclipse l'add on suivant :
Help > Eclipse MarketPlace > Find : Spring ROO > Go
Install : spring tools 3 add-on for spring tools 4

Vous devez avoir maintenant le fichier de configuration suivant :

```
<?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="...">
    </bean>
</beans>
```

2° - Déclaration d'un bean :

- **id** : attribut string qui représente le nom du bean
- **class** : attribut qui définit le type du bean avec son nom complet.

On obtient :

```
<bean id="products" class="fr.peter.ecommerce.service.ProductServiceImpl">
</bean>
```

3° - On va tester la création de ce bean au lancement du programme.
Dans la méthode main, mettez en commentaire (ou effacez ce qui était écrit), on va écrire le code suivant :

```
ApplicationContext context = new ClassPathXmlApplicationContext("services.xml");
```

Cela permet de charger le fichier de configuration afin de créer le bean.
Vous devez voir dans la console que le bean a été instancié.

On va récupérer l'instance de ce bean construit avec le code suivant :

```
ProductService productService = context.getBean("products", ProductService.class);
```

On peut maintenant créer 3-4 produits et les ajouter au bean :

```
Product product1 = new Product(1l, "Produit 1", "desc du prod 1", 12d, "url1", 1l);
Product product2 = new Product(2l, "Produit 2", "desc du prod 2", 32d, "url2", 4l);
Product product3 = new Product(3l, "Produit 3", "desc du prod 3", 22d, "url3",
54l);
productService.save(product1);
productService.save(product2);
productService.save(product3);
```

On affiche pour voir que ça fonctionne :

```
productService.getAllProducts().stream().forEach(System.out::println);
```

3° - Les autres services

1° - Client

On va maintenant créer un service définissant les fonctionnalités d'un client. Création de l'interface ClientService, avec les méthodes suivantes :

```
public List<Client> getAllClient();
public Client getClientById(Long clientId);
public Client save(Client client);
```

Créer la classe *ClientServiceImpl*. Cette classe contient la liste des clients :

```
private final List<Client> allClient = new ArrayList<>();
```

Ajoutez et implémentez les méthodes de l'interface.

Attention :

- La méthode *getClientById* lève une exception de type *ResourceNotFoundException* si le client n'existe pas.

Testez en créant le bean dans le fichier de configuration et ajoutez votre client au bean dans le main.

2° - Order

On fait la même chose pour Order.

Voici les méthodes de l'interface :

```
public List<Order> getAllOrders();
public Order create(Order order);
public void update(Order order) throws StockException;
```

Créer la classe qui implémente cette interface *OrderServiceImpl*. Cette classe contient la liste des commandes mais doit également connaître les quantités de stock des produits :

```
private List<Order> allOrders = new ArrayList<Order>();
private ProductService productService;
```

On voit ici que l'on a besoin de l'instance de ProductService. Car afin de faire des opérations sur une commande, on a besoin de connaître les produits et les quantités des produits en stock (donc rien de tel que le service product pour cela).

Ajoutez à cet attribut une méthode get et set.

Attention :

- La méthode *create* modifie le statut de la commande à "En cours".
- La méthode update met à jour la commande et "l'enregistre" avec le statut marqué "Payée". Dans ce cas, il faut mettre à jour les stocks pour les produits de cette commande selon les quantités. Attention cette méthode peut lever une exception dans le cas où il n'y a pas assez de stock pour un produit (le statut n'est pas modifié dans ce cas).
- Si la méthode update est appelée à nouveau alors que le statut est déjà marqué "Payée", rien ne se passe.

Créez le bean dans le fichier de configuration :

```
<bean id="orders" class="fr.peter.ecommerce.service.OrderServiceImpl">

</bean>
```

Mais nous avons maintenant un problème, car nous avons besoin d'initialiser productService par l'instance du Bean créée grâce à ce fichier de configuration.

On va donc utiliser **l'injection de dépendance** afin d'initialiser "comme par magie" productService par le bean du même type qui a l'id *products.*

On ajoute entre les balises du bean *orders* la propriété que productService (l'attribut) face référence au bean qui a l'id *products* :

```
<property name="productService" ref="products"></property>
```

On peut maintenant tester ce que l'on vient de faire.

```
// Création d'une instance d'une nouvelle commande -> order
Order order = new Order(1l, LocalDate.now(), null, client, new
                        ArrayList<OrderProduct>());
order.addProduct(product1, 2);
order.addProduct(product3, 4);
System.out.println(order);   // Le statut doit être à "null"
```

On récupère le bean dans le main puis on ajoute une commande de produits :

```
// On utilise le bean de type service pour ajouter cette commande
orderService.create(order);
System.out.println(order);   // Le statut doit être à "En cours"
```

Modifiez le statut pour simuler le paiement :

```
// On passe la commande à "Payée"
orderService.update(order);
System.out.println(order);   // Le statut doit être à "Payée"
```

Important : Vérifiez en affichant que les produits de la commande ne sont plus dans les mêmes quantités.
Vérifiez qu'une commande ne passe pas à "Payée" si un produit n'est pas en quantité suffisante.

3° - OrderProduct

De la même manière créer l'interface et la classe d'implémentation. On les laissera vide pour le moment.