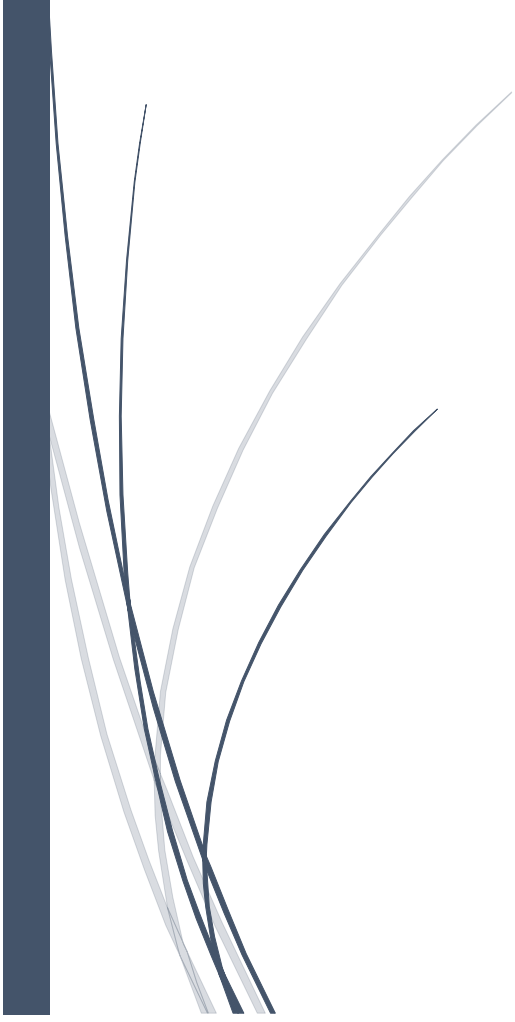
TP : Implémenter le Design Pattern IOC avec Spring IOC



Architecture des composants d’entreprise

Table des matières

I. Objectif du TP

L'objectif de ce TP est de vous montrer comment Spring IOC implémente le Design Pattern IOC. Vous allez comprendre comment Spring implémente :

- L'injection par modificateur.
- L'injection par constructeur.
- L'injection par Factory.

NB : Le code source du TP est disponible sur GITHUB :
https://github.com/abbouformations/tpioc.git

II. Prérequis

- IntelliJ IDEA ou autre IDE ;
- JDK version 17 ;
- Une connexion Internet pour permettre à Maven de télécharger les librairies.

III. C'est quoi l'IOC

- IOC (Inversion Of Control), est une Design Pattern qui corrige le problème de couplage fort.
- IOC permet de ne pas utiliser les classes concrètes.
- Dans l'IOC, la relation entre les classes se fait par interfaces.
- L'IOC permet d'injecter les objets moyennant trois méthodes :
    - Injection par le modificateur ;
    - Injection par constructeur ;
    - Injection par Factory.

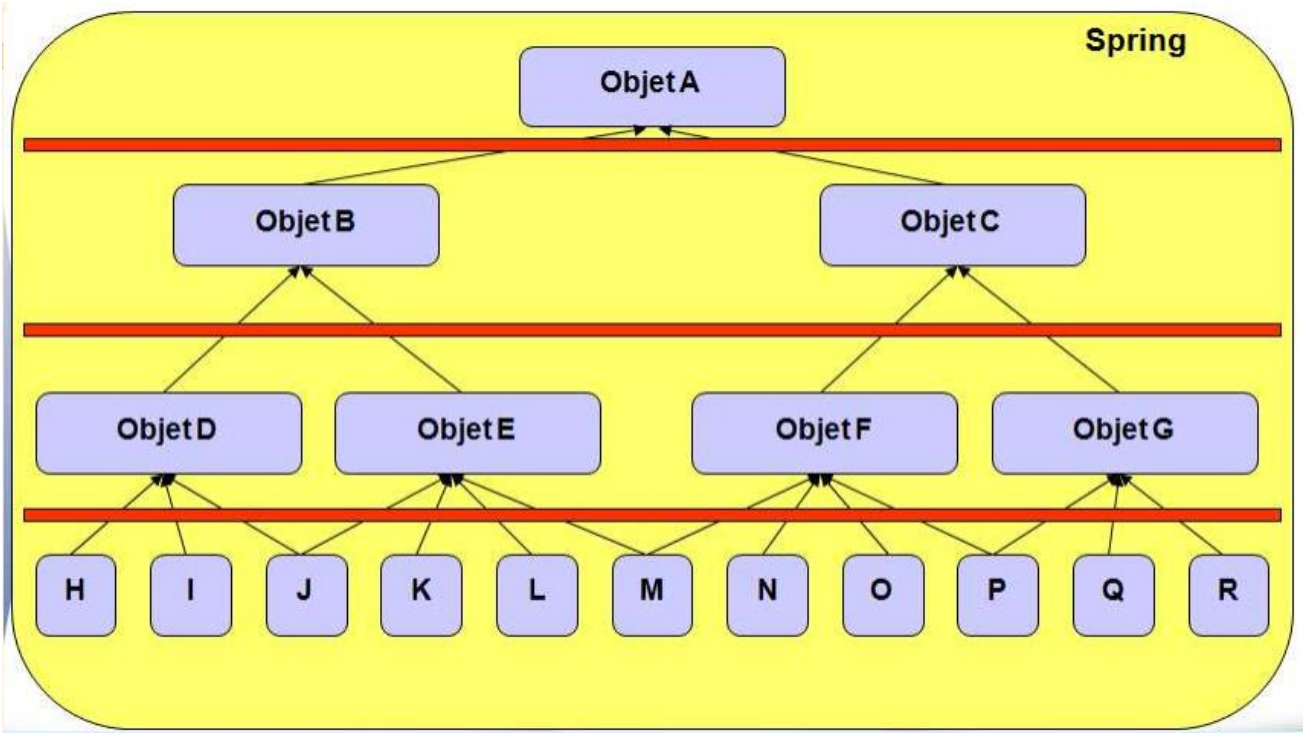
IV. C'est quoi Spring IOC

- Spring IOC ou Spring Core est le noyau du Framework Spring :
    - Est basé sur le Design Pattern IOC
    - Se charge de l'instanciation de tous les objets de l'application et de la résolution des dépendances entre eux.

- Deux API très importantes dans ce module :
    - org.springframework.beans
    - org.springframework.context

- Offrent les bases pour le Design Pattern IOC.
- BeanFatory (*org.springframework.beans.factory.BeanFactory*) permet de configurer les Beans (dans un fichier XML ou via les annotations) et les gérer (instanciation, gestion de la dépendance).

- ApplicationContext (org.springframework.context.ApplicationContext) ajoute des fonctionnalités avancées au BeanFactory (facilite l'intégration avec Spring AOP, …).

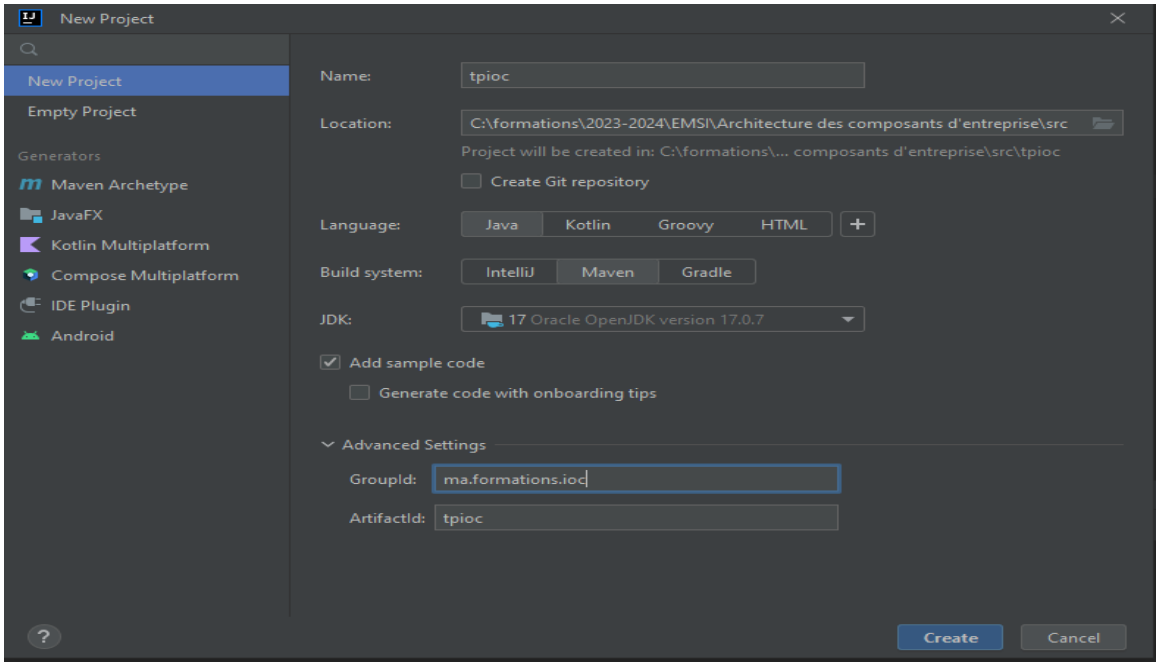


- Spring va permettre à chaque couche de s'abstraire de sa ou ses couches inférieures (injection de dépendance) :
    - Le code de l'application est beaucoup plus lisible.
    - Le maintien de l'application est facilité.
    - Les tests unitaires sont simplifiés.
    - Spring gère les dépendances entre les beans dans un fichier XML ou via les annotations.

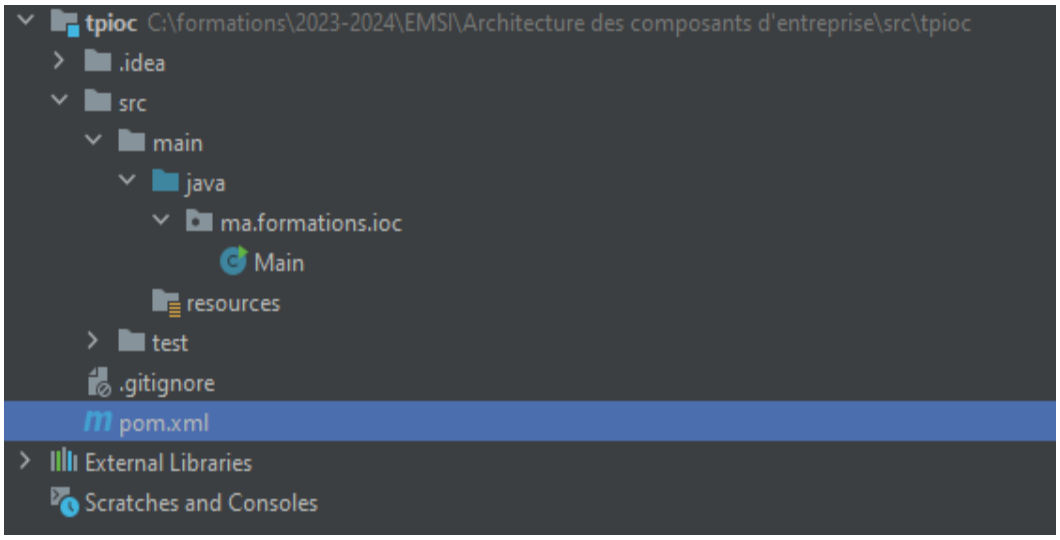
V. Réalisation du TP

a. Création du projet Maven

1. Créer un projet Maven comme le montre la fenêtre suivante :



2- Entrer le nom du projet (dans *Name*), cliquer sur Java (dans *Language*), cliquer sur Maven (dans Build system), choisir JDK 17 (dans JDK), entrer le group Id (dans GroupId), entrer l'artifact Id (dans ArtifactId) et enfin cliquer sur *Create*. L'arborescence du projet ci-dessous sera créée :



2. Modifier le fichier **pom.xml** comme suit :

Ouvrez le fichier pom.xml et ajouter les dépendances suivantes :

```xml
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>6.0.12</version>
  </dependency>
  <dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.30</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.7.2</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Explications :

- La dépendance suivante permet de récupérer toutes librairies nécessaires pour *Spring IOC* et *Spring Context* :

```xml
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>6.0.12</version>
</dependency>
```

  - ✓ Le conteneur léger de Spring est implémenté par *Spring IOC*.
  - ✓ *Spring Context* offre plusieurs services, notamment les annotations *@Service*, *@Repository*, *@Autowired* et *@Bean*.

- La dépendance suivante permet de récupérer le Framework Lombok :

```xml
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <version>1.18.30</version>
  <scope>provided</scope>
</dependency>
```

- ✓ Lombok permet d'ajouter au niveau du fichier byte-code (le fichier .class), les getters, les setters, le constructeur par défaut, le constructeur avec des paramètres, la méthode *toString*, la méthode *hashCode*, la méthode equals, etc.

- La dépendance suivante permet de récupérer les librairies nécessaires pour Junit version 5 :

```
<dependency>
   <groupId>org.junit.jupiter</groupId>
   <artifactId>junit-jupiter-api</artifactId>
   <version>5.7.2</version>
   <scope>test</scope>
</dependency>
```

- ✓ Junit est un Framework *open source* de la communauté Jakarta Apache.
- ✓ Junit permet de réaliser les tests unitaires très facilement.
- ✓ Junit offre plusieurs annotations : @Test, @TestBeforeAll, @TestAfterAll, etc.

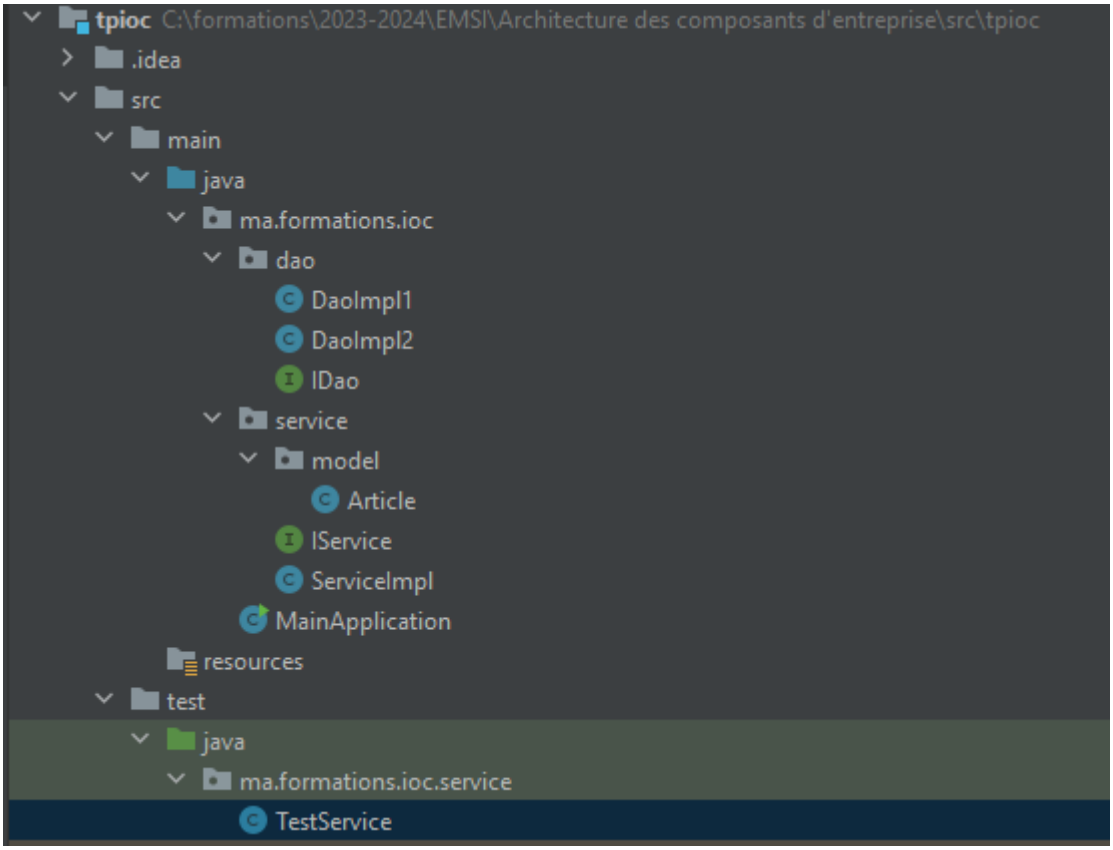
NB : Pour précision, le contenu global du fichier pom.xml est le suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns="http://maven.apache.org/POM/4.0.0"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>

   <groupId>ma.formations.ioc</groupId>
   <artifactId>tpioc</artifactId>
   <version>1.0-SNAPSHOT</version>

   <properties>
      <maven.compiler.source>17</maven.compiler.source>
      <maven.compiler.target>17</maven.compiler.target>
      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
   </properties>

   <dependencies>
      <dependency>
         <groupId>org.springframework</groupId>
         <artifactId>spring-context</artifactId>
         <version>6.0.12</version>
      </dependency>
      <dependency>
         <groupId>org.projectlombok</groupId>
         <artifactId>lombok</artifactId>
```

```
        <version>1.18.30</version>
        <scope>provided</scope>
      </dependency>
      <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.7.2</version>
        <scope>test</scope>
      </dependency>
    </dependencies>
</project>
```

b. Création de l'arborescence du projet

L'arborescence de votre projet est la suivante :



- La classe **MainApplication** :

```
package ma.formations.ioc;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
```

```java
@Configuration
@ComponentScan(basePackages = "ma.formations.ioc")
public class MainApplication {

}
```

Explications :

- L'annotation *@Configuration* est une classe Java Config. Elle est équivalente à un fichier XML. Cette classe devrait contenir les Bean à utiliser dans l'application.

- L'annotation *@ComponentScan* permet à Spring de gérer toutes les classes se trouvant au niveau du package *ma.formations.ioc* et également au niveau des sous packages de ce dernier.

- Spring ignorera toute classe ne se trouvant pas dans ces packages.

- La classe ***Article*** :

```java
package ma.formations.ioc.service.model;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class Article {
   private Long id;
   private String description;
   private Double price;
   private Double quantity;

}
```

Explications :

- *@Data* est une annotation de Lombok. Elle permet d'ajouter au niveau du fichier .class
    - Les getters, les setters, la méthode *equals,* la méthode *hashCode* et la méthode *toString*.

- *@NoArgsConstructor* est une annotation de Lombok. Elle permet d'ajouter au niveau du fichier .class le constructeur par défaut.

- *@AllArgsConstructor* est une annotation de Lombok. Elle permet d'ajouter au niveau du fichier .class le constructeur avec l'ensemble des paramètres.

- L'interface ***IDao*** :

```
package ma.formations.ioc.dao;

import ma.formations.ioc.service.model.Article;
import java.util.List;

public interface IDao {
   List<Article> getAll();
   void save(Article article);
   void deleteById(Long id);
   Article findById(Long id);

}
```

- La classe ***DaoImpl1*** :

```
package ma.formations.ioc.dao;

import ma.formations.ioc.service.model.Article;
import org.springframework.stereotype.Repository;

import java.util.List;

@Repository
public class DaoImpl1 implements IDao {
   private static final List<Article> repository = List.of(
         new Article(1L, "PC HP I7", 25000d, 5d),
         new Article(2L, "PC HP I5", 15000d, 10d),
         new Article(3L, "TV LG 32p", 3500d, 8d),
         new Article(4L, "TV SAMSUNG 60p", 9000d, 15d));

   @Override
   public List<Article> getAll() {
      return repository;
   }

   @Override
   public void save(Article article) {
      repository.add(article);
   }

   @Override
```

```
    public void deleteById(Long id) {
        repository.remove(repository.stream().filter(a ->
a.getId().equals(id)).findAny().orElse(null).getId().intValue());
    }

    @Override
    public Article findById(Long id) {
        return repository.stream().filter(a -> a.getId().equals(id)).findAny().orElse(null);
    }
}
```

Explications :

- L'annotation *@Repository* de *Spring Context* permet à Spring de gérer le Bean annoté par cette dernière.

- L'annotation *@Repository* hérite de l'annotation *@Component*.

- L'annotation *@Component* permet d'ajouter le bean annoté par cette dernière au conteneur de Spring. Les beans se trouvant dans le conteneur de Spring sont des « Managed Bean ».

  NB : Spring crée par défaut les Beans sous forme de Singleton (une seule instance). Si vous voulez que Spring crée à chaque fois une nouvelle instance du même Bean, il suffit d'annoter votre classe par : ***@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)***.

- La classe ***DaoImpl2*** :

```
package ma.formations.ioc.dao;

import ma.formations.ioc.service.model.Article;

import java.util.List;

//@Repository
public class DaoImpl2 implements IDao {
    private static final List<Article> repository = List.of(
            new Article(1L, "DESCRIPTION_1", 100d, 3d),
            new Article(2L, "DESCRIPTION_2", 300d, 11d),
            new Article(3L, "DESCRIPTION_3", 15000d, 33d),
            new Article(4L, "DESCRIPTION_4", 11000d, 4d));

    @Override
    public List<Article> getAll() {
        return repository;
    }
```

```
    @Override
    public void save(Article article) {
        repository.add(article);
    }

    @Override
    public void deleteById(Long id) {
        repository.remove(repository.stream().filter(a ->
a.getId().equals(id)).findAny().orElse(null).getId().intValue());
    }

    @Override
    public Article findById(Long id) {
        return repository.stream().filter(a -> a.getId().equals(id)).findAny().orElse(null);
    }
}
```

Explications :

- Remarquez que nous avons commenté la ligne où se trouve l'annotation @Repository. En effet, Spring lève une exception s'il trouve deux Bean de même type gérés par Spring (Managed bean). Dans cet exemple, les deux classes *DaoImpl1* et *DaoImpl2* sont de même type (càd : elles implémentent la même interface IDao).

- Spring gère les beans de deux façons différences :
    - Soit via un fichier XML (les beans sont définis au niveau de ce fichier). Cette façon a été abandonnée.

    - Soit via les annotations :
        - @Component
        - @service
        - @Repository
        - @Bean

- L'interface **IService** :

```
package ma.formations.ioc.service;

import ma.formations.ioc.service.model.Article;

import java.util.List;
```

```java
public interface IService {
   List<Article> getAll();

   void save(Article article);

   void deleteById(Long id);

   Article findById(Long id);

}
```

- La classe **ServiceImpl** :

```java
package ma.formations.ioc.service;

import ma.formations.ioc.dao.IDao;
import ma.formations.ioc.service.model.Article;
import org.springframework.stereotype.Service;

import java.util.List;

//@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
@Service
public class ServiceImpl implements IService {
   private IDao dao;

   @Override
   public List<Article> getAll() {
      return dao.getAll();
   }

   @Override
   public void save(Article article) {
      dao.save(article);
   }

   @Override
   public void deleteById(Long id) {
      dao.deleteById(id);
   }

   @Override
   public Article findById(Long id) {
      return dao.findById(id);
   }
}
```

Explications :

- L'annotation *@Service* de *Spring Context* permet à Spring de gérer le Bean annoté par cette dernière.

- L'annotation @Service est utilisée sur les classes de la couche Service (ou Métier).

c. L'injection par Modificateur

Pour utiliser l'injection par Modificateur (par le setter), ajouter la méthode *setDao(IDao dao)* suivante au niveau de la classe *ServiceImpl*:

```
// Injection par modificateur
@Autowired
public void setDao(IDao dao) {
   this.dao = dao;
}
```

Explications :

- L'annotation @Autowired appliquée sur le *setter* permet d'injecter le Bean implémentant l'interface *IDao*. Dans ce cas, Spring IOC passe une instance de la classe implémentant l'interface *IDao* en paramètre de la méthode *setDao(IDao dao)* : **c'est l'injection par Modificateur**. A partir de la version 4.3 de Spring, l'annotation @Autowired est facultative **à condition que la classe ait un seul constructeur**. Le cas échéant, il faut annoter le setter par cette dernière.

- Vous constatez que la classe *ServiceImpl* n'a aucune connaissance de la classe implémentant l'interface *IDao*. En effet, nous pouvons changer l'implémentation de l'interface *IDao* sans modifier le code source de la classe *ServiceImpl* : **c'est le principe de couplage faible**.

- Pour que Spring puisse injecter une instance de la classe *DaoImpl2* au lieu de la classe *DaoImpl1*, il suffit d'annoter la classe *DaoImpl2* par *@Repository*. Dans ce cas, il faut supprimer l'annotation *@Repository* de la classe *DaoImpl1*.

- Si vous souhaitez garder l'annotation *@Repository* sur les deux classes *DaoImpl1* et *DaoImpl2*, vous pouvez utiliser l'annotation *@Qualifier* comme expliqué ci-après :

```
@Repository
@Qualifier("dao1")
public class DaoImpl1 implements IDao {…}
```

```
@Repository
@Qualifier("dao2")
public class DaoImpl2 implements IDao {...}
```

```
@Service
public class ServiceImpl implements IService {
   private IDao dao;

   // Injection par modificateur
   @Autowired
   @Qualifier("dao2")
   public void setDao(IDao dao) {
      this.dao = dao;
   }
...
}
```

De cette façon, Spring injectera une instance de la classe DaoImpl2.

d. Création de la classe de test

Créer la classe *TestService* suivante :

```
package ma.formations.ioc;

import ma.formations.ioc.service.IService;
import ma.formations.ioc.service.ServiceImpl;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class TestService {

   static AnnotationConfigApplicationContext context = null;

   @BeforeAll
   static void init() {
      context = new AnnotationConfigApplicationContext(MainApplication.class);
   }

   @AfterAll
   static void close() {
```

```
        context.close();
    }

    @Test
    void test1() {
        IService service = context.getBean(ServiceImpl.class);
        Assertions.assertAll("données erronées",
                () -> Assertions.assertEquals(1L, service.findById(1L).getId(), "First Id article is
not correct"),
                () -> Assertions.assertEquals("PC HP I7", service.findById(1L).getDescription(),
"First article description is not correct"),
                () -> Assertions.assertEquals(25000d, service.findById(1L).getPrice(), "First article
price is not correct"),
                () -> Assertions.assertEquals(5d, service.findById(1L).getQuantity(), "First article
quantity is not correct"));
    }

}
```

Explications :

- L'annotation *@BeforeAll* de *JUNIT* permet d'exécuter la méthode une seule fois au début (avant toutes les méthodes annotées par *@Test*).

- La méthode annotée par *@BeforeAll* doit être *static*.

- L'annotation *@AfterAll* de *JUNIT* permet d'exécuter la méthode une seule fois à la fin (après toutes les méthodes annotées par *@Test*).

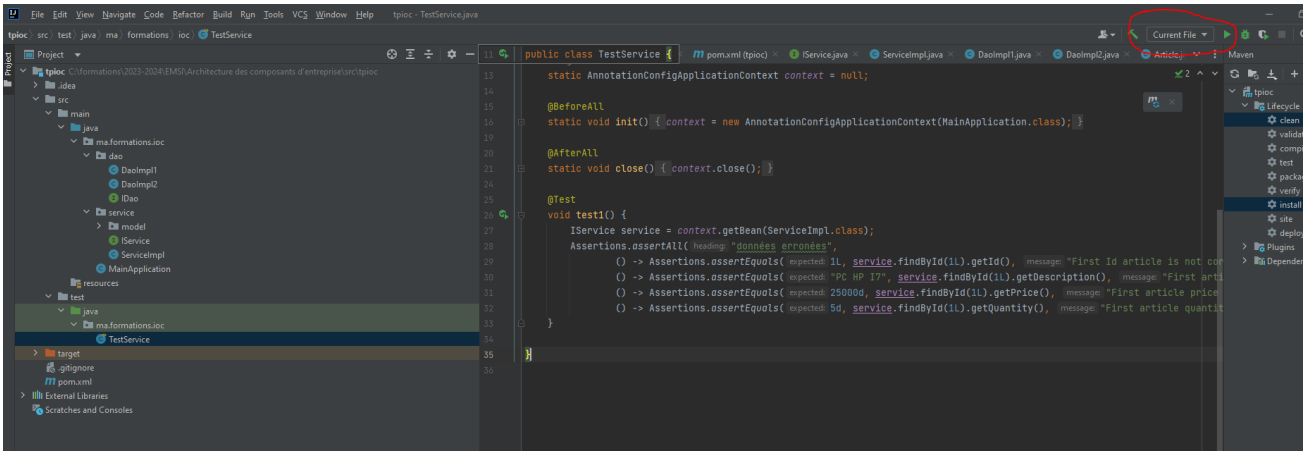
- La méthode annotée par *@AfterAll* doit être *static*.

- Remarquez que nous avons commencé par instancier la classe *AnnotationConfigApplicationContext* en lui passant en paramètre la classe *MainApplication*. En effet, ceci permettra de créer le conteneur de Spring et d'ajouter à ce dernier toutes les classes se trouvant au niveau du package *ma.formations.ioc*.

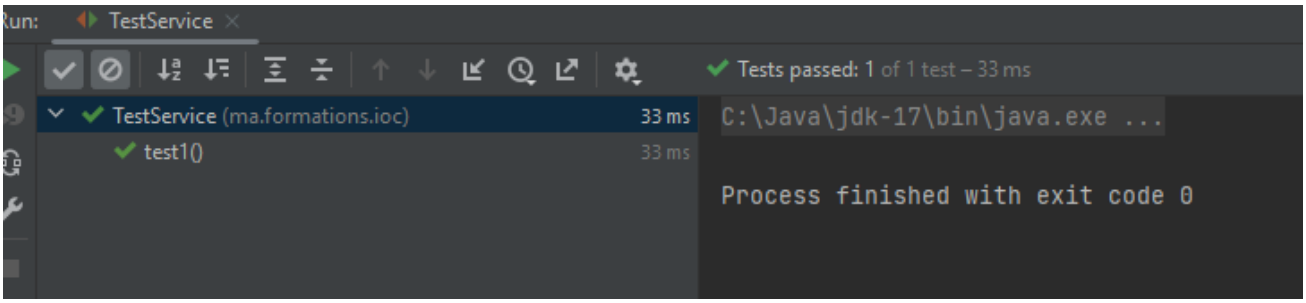
- La méthode *AnnotationConfigApplicationContext.getBean(ServiceImpl.class)* permet de récupérer une instance de la classe ServiceImpl.

- Remarquez que nous avons utilisé la méthode *Assertions.assertAll* qui permet d'exécuter plusieurs vérifications.

- Exécuter le test comme suit :



- Vérifier que le test est concluant :



e. L'injection par Constructeur

- Modifier la classe *ServiceImpl* comme suit :

```
package ma.formations.ioc.service;

import ma.formations.ioc.dao.IDao;
import ma.formations.ioc.service.model.Article;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Service;

import java.util.List;

//@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
@Service
public class ServiceImpl implements IService {
   private final IDao dao;

   public ServiceImpl(@Qualifier("dao1") IDao dao) {
      this.dao = dao;
   }

   @Override
```

```
    public List<Article> getAll() {
        return dao.getAll();
    }

    @Override
    public void save(Article article) {
        dao.save(article);
    }

    @Override
    public void deleteById(Long id) {
        dao.deleteById(id);
    }

    @Override
    public Article findById(Long id) {
        return dao.findById(id);
    }
}
```

Explications :

- Remarquez que nous n'avons pas annoté le constructeur par *@Autowired.* Car, comme expliqué ci-dessus, puisque la classe *ServiceIml* contient un seul constructeur, l'annotation *@Autowired* devient facultative.

- Remarquez que Spring injectera le Bean moyennant cette surcharge du constructeur. Il s'agit ici de **l'injection par Constructeur**.

- Nous avons utilisé *@Qualifier* puisque dans cet exemple, nous avons deux Bean qui implémentent la même interface IDao.

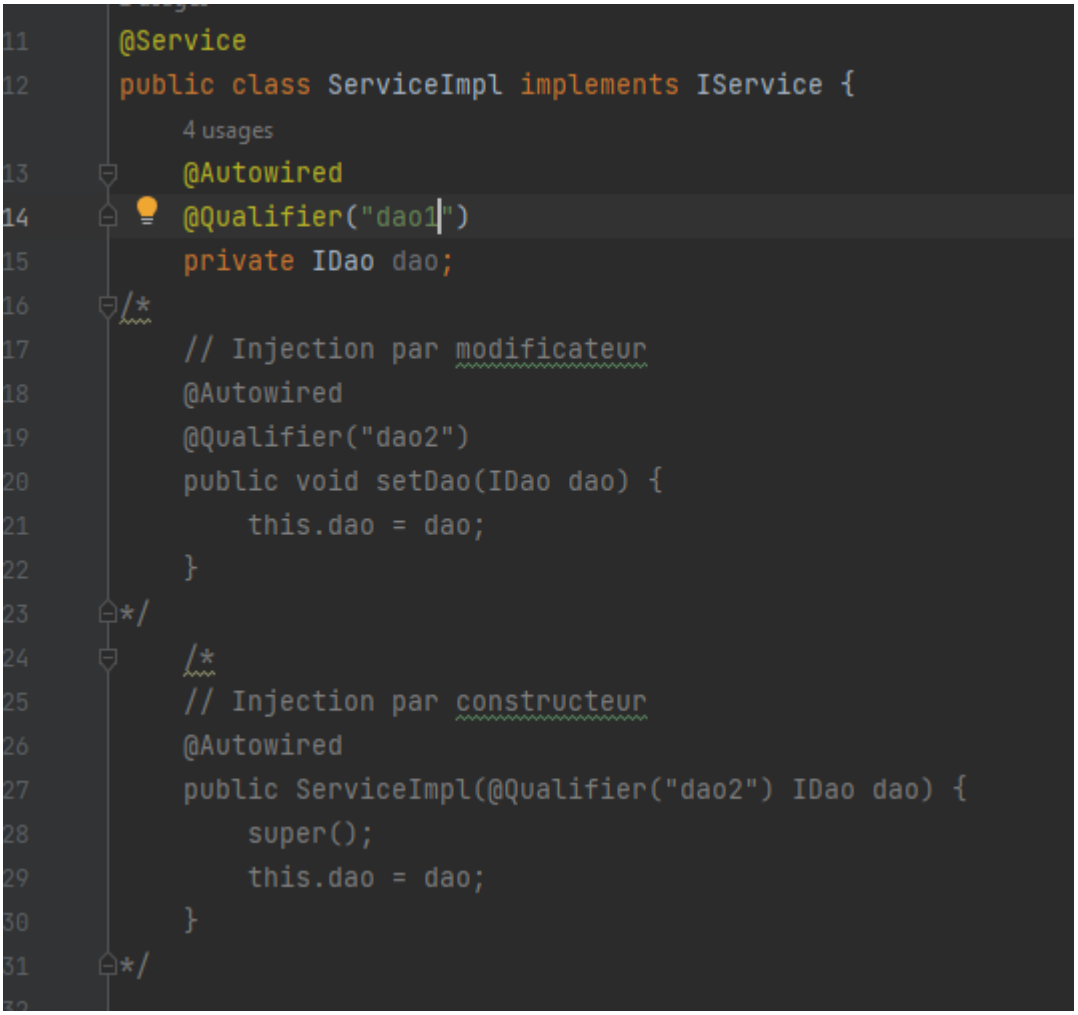
- Refaire le test précédent et vérifier que l'injection du Bean se fait sans aucun problème.

f. L'injection par Factory

- Commenter les deux lignes @Repository et @Qualifier au niveau de la classe DaoImpl1 :

```
//@Repository
//@Qualifier("dao1")
2 usages
public class DaoImpl1 implements IDao {
    5 usages
```

- Modifier la classe *ServiceImpl* comme suit :



```
@Service
public class ServiceImpl implements IService {
    4 usages
    @Autowired
    @Qualifier("dao1")
    private IDao dao;
/*
    // Injection par modificateur
    @Autowired
    @Qualifier("dao2")
    public void setDao(IDao dao) {
        this.dao = dao;
    }
*/
    /*
    // Injection par constructeur
    @Autowired
    public ServiceImpl(@Qualifier("dao2") IDao dao) {
        super();
        this.dao = dao;
    }
*/
```

Remarquez que la variable d'instance dao est annotée par @Autowired (ligne n°14).

- Modifier la classe *MainApplication* comme suit :

```
package ma.formations.ioc;

import ma.formations.ioc.dao.DaoImpl1;
import ma.formations.ioc.dao.IDao;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan(basePackages = "ma.formations.ioc")
public class MainApplication {
   // Injection par factory
   @Bean
   @Qualifier("dao1")
```

```
    public IDao getDao() {
        return new DaoImpl1();
    }
}
```

Explications :

- L'annotation *@Bean* s'applique sur les méthodes.
- L'instance retournée par la méthode annotée par *@Bean* est ajoutée au niveau du conteneur de Spring.
- Remarquez que la méthode *getDao()* ci-dessus joue le rôle d'une fabrique. C'est le principe de ***l'injection par Factory***.

- Refaire le test précédent et vérifier que l'injection du Bean se fait sans aucun problème.