

## Rappel Séance 5

(week-end 11-12/12/2021)

1. Trouver le contenu de la collection ArrayList suivante:

```
package cigma.pfe;

import java.util.ArrayList;
import java.util.List;

class Laptop{
    double price;
    public Laptop(double price) {
        this.price = price;
    }
    @Override
    public String toString() {
        return "Laptop{" +
            "price=" + price +
            '}';
    }
}

public class Rappel {
    public static void main(String[] args) {
        List<Laptop> l = new ArrayList<>();
        l.add(new Laptop(10));
        l.add(new Laptop(20));
        l.add(1,new Laptop(5));
        l.forEach(i-> System.out.println(i));
    }
}
```

## 2. Trouver le contenu de la collection HashSet suivante:

```
package cigma.pfe;

import java.util.*;

class Laptop{
    double price;
    public Laptop(double price) {
        this.price = price;
    }
    @Override
    public String toString() {
        return "Laptop{" +
            "price=" + price +
            '}';
    }

    @Override
    public int hashCode() {
        return (""+this.price).length();
    }
    @Override
    public boolean equals(Object o) {
        return this.price==(Laptop)o.price;
    }
}

public class Rappel {
    public static void main(String[] args) {
        Set<Laptop> l = new HashSet<>();
        l.add(new Laptop(12.000));
        l.add(new Laptop(13.6));
        l.add(new Laptop(12));
        l.add(new Laptop(15.4));
        l.add(new Laptop(13.600));
        l.add(new Laptop(13.60));

        l.forEach(i-> System.out.println(i));
    }
}
```

3. Trouver le contenu de la collection TreeSet suivante:

```
package cigma.pfe;

import java.util.HashSet;
import java.util.Set;
import java.util.TreeSet;

class Point{
    double x;
    double y;
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public String toString() {
        return "Point{" +
            "x=" + x +
            ", y=" + y +
            '}';
    }
}

public class Rappel2 {
    public static void main(String[] args) {
        Set<Point> l = new TreeSet<>((p1,p2)->(int) (p1.x-p2.x));
        l.add(new Point(12.5,13));
        l.add(new Point(14.4,15));
        l.add(new Point(12.50,15));
        l.add(new Point(11.50,15));
        l.forEach(i-> System.out.println(i));
    }
}
```

## Objectifs de la séance 6

(week-end 18-19/12/2021)

Les objectifs de la séance d'aujourd'hui:

**Objectif 6.1** : Création des **annotations** en Java

**Objectif 6.2** : Création des projets Maven **Three Layered Architecture**

Complément Youtube de cette séance : [https://youtu.be/KsO3Uqf\\_oxw](https://youtu.be/KsO3Uqf_oxw)

### I. Création et utilisation des annotations en java

1. Dans votre projets "Mes TPs Java", créer un package "ma.education.tp6.annotations"
2. Créer une annotation Programmer, les annotations sont créées en utilisant le mot réservé **@interface** . Cette annotation contient les deux signatures: `int id();` et `String name();`

```
package ma.education.tp6.annotations;  
public @interface Programmer {  
    abstract int id();  
    String name();  
}
```

3. Cette annotation sera appliquée seulement aux classes et aux interfaces. Pour le dire, l'annotation Programmer doit être annoté par l'annotation **@Target**

```
@Target(ElementType.TYPE)  
public @interface Programmer {  
    abstract int id();  
    String name();  
}
```

*ElementType.TYPE = Class, interface (including annotation type), or enum declaration*

4. Dans ma.education.tp6.annotations, créer la classe Calculatrice, en utilisant l'annotation précédente **@Programmer** mentionner le programmeur qui a développé la classe Calculatrice.

```
@Programmer(id=10,name="Said ALAMI")  
public class Calculatrice {  
}
```

5. Créer une classe TestReflectionAnnotation pour afficher les valeurs de l'annotation **@Programmer** utilisées dans la classe Calculatrice

```
public class TestReflectionAnnotation {  
    public static void main(String[] args) {  
        Class c = Calculatrice.class;  
        Programmer programmer = (Programmer)
```

```
c.getDeclaredAnnotation(Programmer.class);

System.out.println(programmer.id()+":"+programmer.name());
}
}
```

Exécuter cette classe et remarquer que l'annotation n'existe pas au moment de l'exécution  
**Exception in thread "main" java.lang.NullPointerException**



Une annotation est définie par sa rétention, c'est-à-dire la façon dont elle sera conservée. La rétention est définie grâce à la méta-annotation `@Retention`. Les différentes rétentions d'annotation possibles sont :

**SOURCE** : L'annotation est accessible durant la compilation mais n'est pas intégrée dans le fichier `.class` généré.

**CLASS** : L'annotation est accessible durant la compilation, elle est intégrée dans le fichier `.class` généré mais elle n'est pas chargée dans la JVM à l'exécution.

**RUNTIME** : L'annotation est accessible durant la compilation, elle est intégrée dans le fichier `.class` généré et elle est chargée dans la JVM à l'exécution. Elle est accessible par introspection (la réflexion).

6. Ajouter alors `@Retention(RetentionPolicy.RUNTIME)` à l'annotation `@Programmer` et exécuter encore une fois la classe `TestReflectionAnnotation`

```
package ma.education.tp6.annotations;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Programmer {
    abstract int id();
    String name();
}
```

7. Créer une classe fille de `Calculatrice` et appeler la `CalculatriceMath`. Est ce que la classe fille va hériter l'annotation `@Programmer` de sa classe mère `Calculatrice`.

```
public class CalculatriceMath extends Calculatrice{
}
```

8. Changer la classe `TestReflectionAnnotation` pour vérifier si `CalculatriceMath` est aussi annotée par `@Programmer`

```
public class TestReflectionAnnotation {
    public static void main(String[] args) {
        Class c = CalculatriceMath.class;
        Programmer programmer = (Programmer)
c.getAnnotation(Programmer.class);
    }
}
```

```
System.out.println(programmer.id()+" "+programmer.name());
    }
}
```

N'oublier pas de changer `c.getDeclaredAnnotation(Programmer.class)` par `c.getAnnotation(Programmer.class)`

Exécuter cette classe et remarquer l'exception : `java.lang.NullPointerException`

- Annoter l'annotation `@Programmer` par l'annotation `@Inherited` et refaire l'exécution de la classe `TestReflectionAnnotation`. C'est quoi votre remarque?

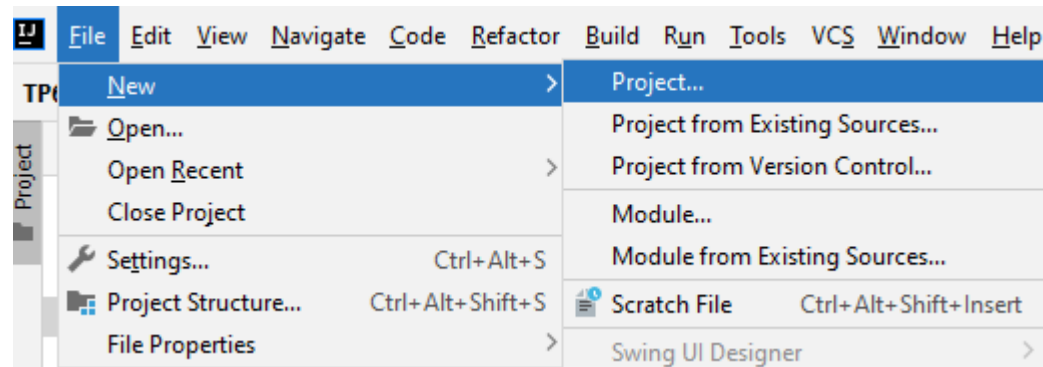


Annoter le package "ma.education.tp6.annotations" par l'annotation `@Programmer` et afficher les valeurs de l'annotation en utilisant la réflexion.

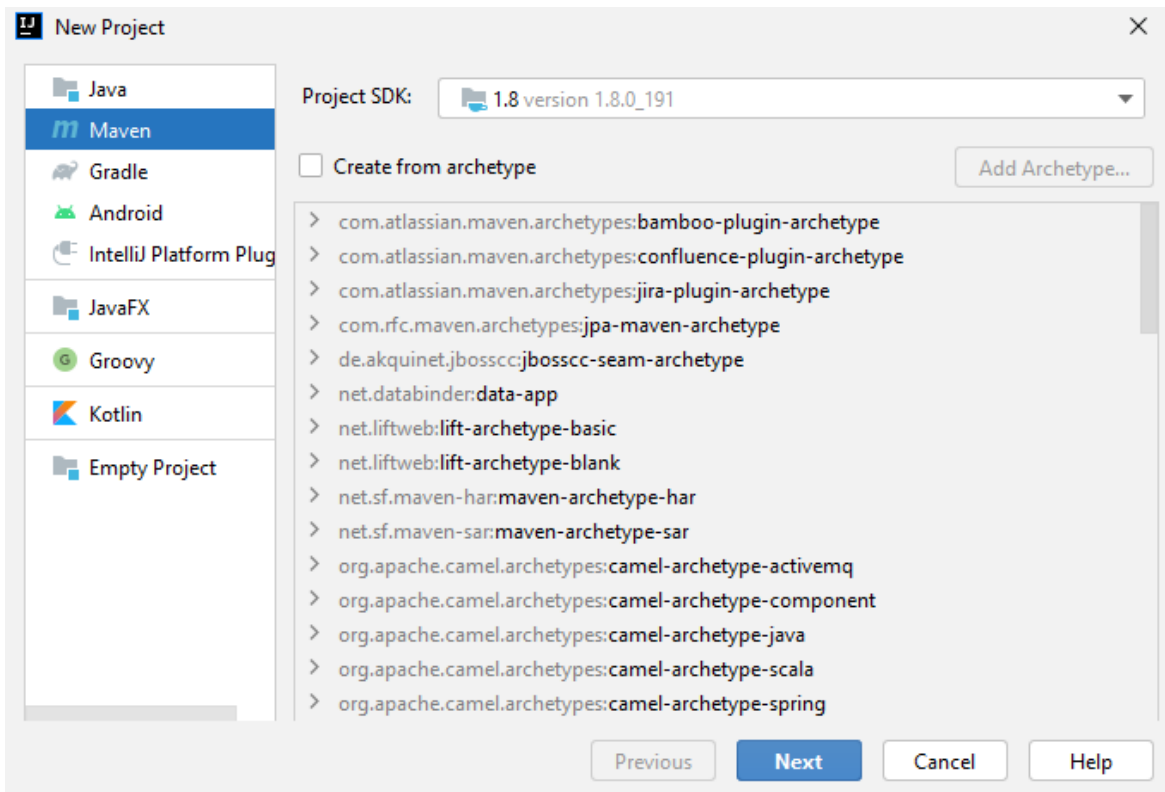
## II. Création du projet Maven sous IntelliJ et Eclipse

### A. Maven sous IntelliJ

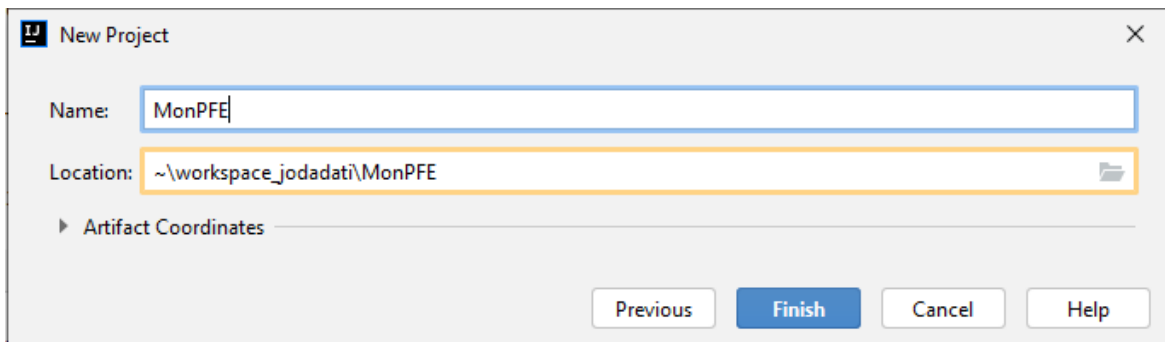
Pour créer un projet maven sous intellij, cliquer sur **File>New>Project**



Choisir Maven et la version SDK pour cliquer sur **Next**.



Donner un nom à votre projet : Mon PFE à titre d'exemple puis cliquer sur **Finish**



Dans le projet crée, remarquer le fichier **pom.xml** et les sources Folders suivants:

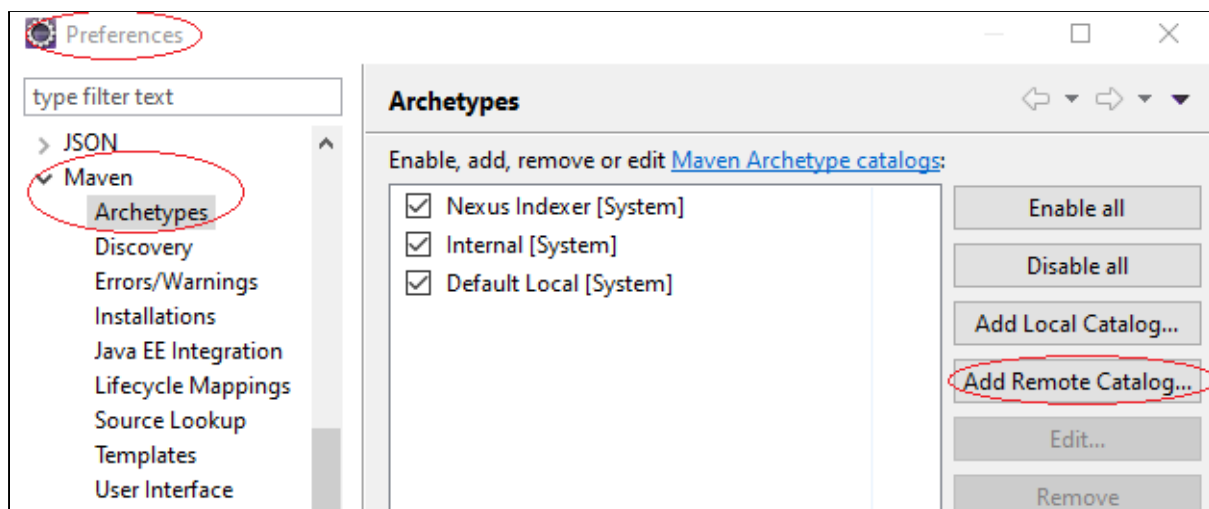
**src/main/java**  
**src/main/resources**  
**src/test/java**



## B. Maven sous Eclipse

Ajoutez le catalogue Maven distant. [Il faut être connecté à Internet]

a. Windows -> Preferences -> Maven -> Archetypes->

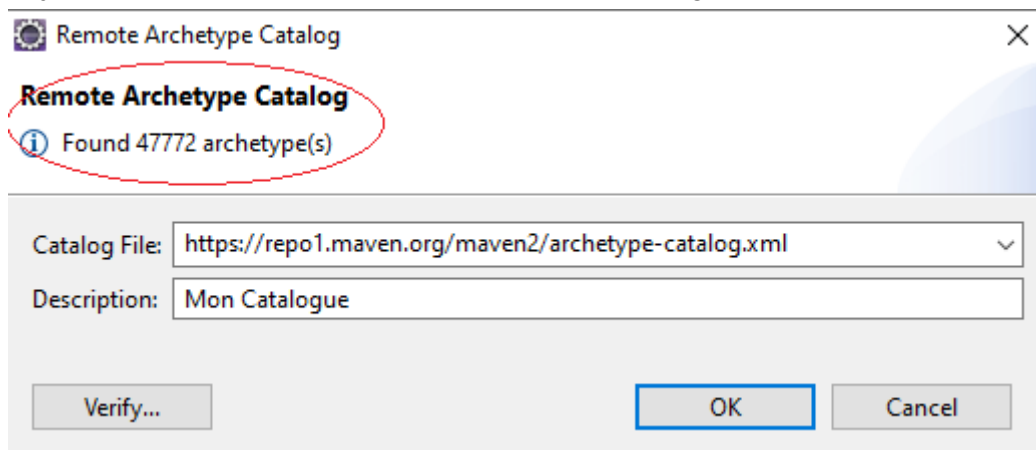


b. Cliquez sur le bouton Add Remote Catalog.

Saisir le lien du fichier catalogue :

**<https://repo1.maven.org/maven2/archetype-catalog.xml>**

Ajouter une description de votre choix, "Mon Catalogue" à titre d'exemple

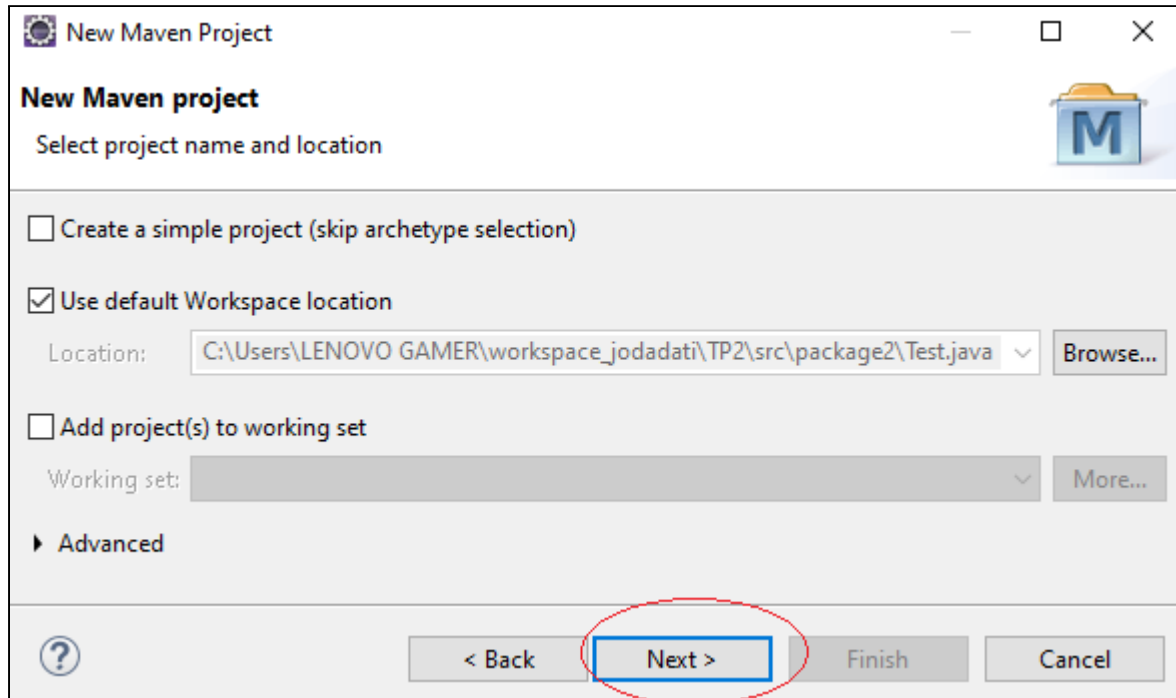




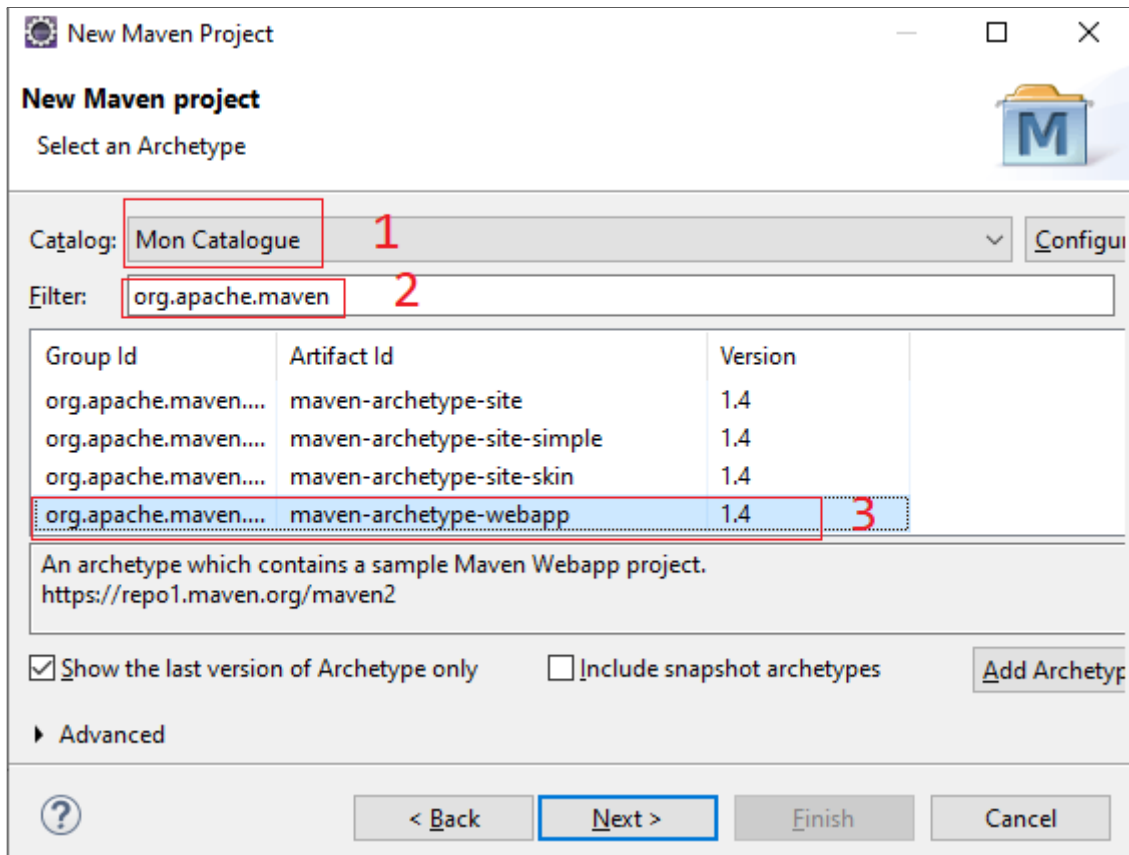
c. Cliquez sur le bouton **Verify** pour voir le nombre des types d'architecture trouvées.

## 2. Maintenant créer un projet Maven

**File->New->Project->Maven->MavenProject**



Cliquer sur "Next"



- Choisir le catalogue que vous avez créé dans l'étape (1) . "Mon catalogue" dans mon cas.
- Filtrer les architectures du catalogue par "org.apache.maven"
- Choisir l'artifactId = **maven-archetype-webapp** version **1.4**
- Cliquer sur "Next"

**New Maven Project**

**New Maven project**  
Specify Archetype parameters

Group Id:

Artifact Id:

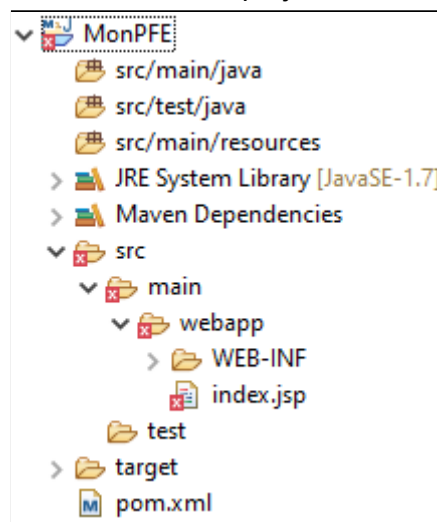
Version:

Package:

Properties available from archetype:

Name	Value

- Dans le **"Group Id"** saisir le nom du Package "ma.education.tp3"
- Dans "Artifactid" saisir le nom de votre projet PFE. "MonPFE"
- Cliquer sur Finish et vérifier la structure du projet Maven suivante:



- Supprimer "index.jsp" qui présente une erreur de compilation.
- Remarque que les composants de base d'un projet maven sont : JRE System Library, **Maven Dependencies**, src,target et **pom.xml**.

### Le package "src" :

Dans "src", on trouve un dossier "main" qui contient un autre répertoire "java" : c'est l'emplacement de nos classes java.

On peut aussi voir qu'il y a un répertoire **"ressources"** qui va contenir les images et la configuration nécessaire pour notre projet.

### Le fichier **"pom.xml"** :

Ce fichier xml permettra d'ajouter des bibliothèques à notre projet ( dependencies)

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

A titre d'exemple les balises ci-dessus vont ajouter les jars suivants:

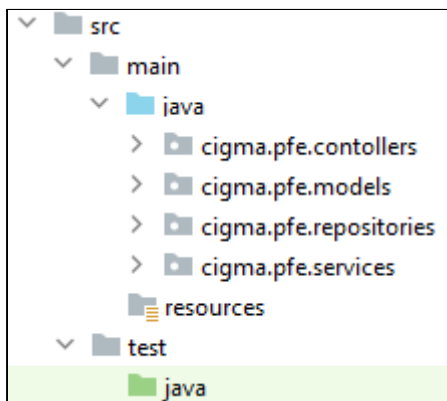
```

v Maven Dependencies
  > junit-4.11.jar - C:\Users\LENOVC
  > hamcrest-core-1.3.jar - C:\Users\
```

Le pom.xml permet aussi de faire le build du projet pour le déployer en production.

## III. Création d'une **"Three Layered architecture"**

Dans Sources Folder **"src/main/java"**, créer quatres packages: **"cigma.pfe.models"**, **"cigma.pfe.repositories"**, **"cigma.pfe.services"** et **"cigma.pfe.controllers"**



Dans le package **"cigma.pfe.models"**, créer la classe Client suivante:

```
public class Client {
    public long id;
    public String name;
    public Client(long id, String name) {
        this.id = id;
        this.name = name;
    }
    public Client() {
    }
}
```

Dans le package “cigma.pfe.repositories”, créer l'interface **ClientRepository** et son implémentation **ClientRepositoryImpl** comme suivant:

```
// l'interface ClientRepository
package cigma.pfe.repositories;
import cigma.pfe.models.Client;
public interface ClientRepository {
    Client save(Client c);
}

// La classe d'implémentation de l'interface ClientRepository
package cigma.pfe.repositories;
import cigma.pfe.models.Client;

public class ClientRepositoryImpl implements ClientRepository{
    @Override
    public Client save(Client c) {
        System.out.println("DAO Layer : ClientRepositoryImpl Level");
        return null;
    }
}
```

Dans le package “cigma.pfe.services”, créer l'interface **ClientService** et son implémentation **ClientServiceImpl** comme suivant:

```
// l'interface ClientService
package cigma.pfe.services;

import cigma.pfe.models.Client;

public interface ClientService {
    Client save(Client c);
}

// La classe d'implémentation de l'interface Client Service
package cigma.pfe.services;
import cigma.pfe.models.Client;
import cigma.pfe.repositories.ClientRepository;
import cigma.pfe.repositories.ClientRepositoryImpl;

public class ClientServiceImpl implements ClientService{
    ClientRepository clientRepository = new ClientRepositoryImpl();
    @Override
    public Client save(Client c) {
        System.out.println("Service Layer : ClientServiceImpl Level... ");
        return clientRepository.save(c);
    }
}
```

Dans le package “cigma.pfe.controllers”, créer la classe ClientController

```
package cigma.pfe.controllers;

import cigma.pfe.services.ClientService;
import cigma.pfe.models.Client;
```

```
import cigma.pfe.services.ClientServiceImpl;

public class ClientController {

    ClientService clientService = new ClientServiceImpl();

    public Client save(Client c ){
        System.out.println("ClientController level...");
        return clientService.save(c);
    }
}
```

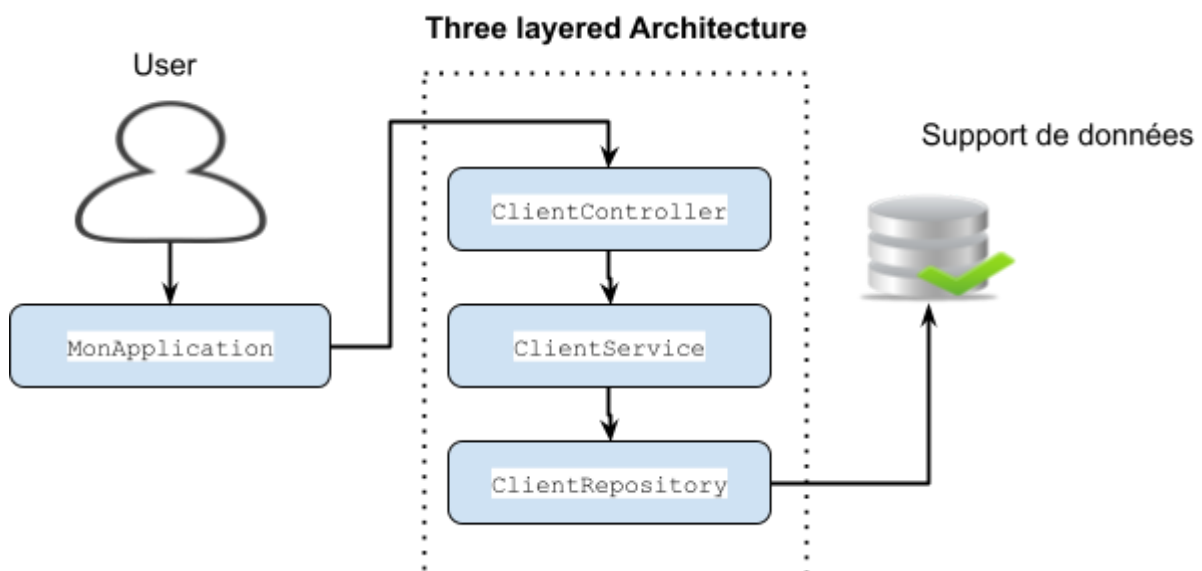
Dans le package "cigma.pfe", créer la classe de lancement MonApplication

```
package cigma.pfe;

import cigma.pfe.controllers.ClientController;
import cigma.pfe.models.Client;

public class MonApplication {
    public static void main(String[] args) {
        ClientController ctrl1 = new ClientController();
        Client client = new Client(1L, "testing");
        ctrl1.save(client);
    }
}
```

Debugger l'application et suivre l'enchaînement des appels selon le schéma suivant::



#### IV. Inversion of control (IOC) : Spring beans, Spring Context et Spring Core

Le but de cette partie est de supprimer les trois instantiations créés dans le projet précédent:

Dossier des travaux pratiques. Module 1 : Java de base .Années scolaire 2021/2022. Niveau : Licence FST Settati

Professeur : M. Boulchahoub Hassan [hboulchahoub@gmail.com](mailto:hboulchahoub@gmail.com)

Mise à jour 18 Nov 2021

Composant	Elimination des instantiations du développeur
Mon Application	<pre>public class MonApplication {     public static void main(String[] args) {         ClientController ctrl1 = new ClientController();     } }</pre>
Controller	<pre>public class ClientController {      ClientService clientService = new ClientServiceImpl(); }</pre>
Service	<pre>public class ClientServiceImpl implements ClientService{     ClientRepository clientRepository = new ClientRepositoryImpl(); }</pre>

Commençons par éliminer l'instanciation du controlleur Client Controller  
Créer le fichier spring.xml suivant dans le package "src/main/resources"

```
<beans>
    <bean id="controller"
class="cigma.pfe.contollers.ClientController"></bean>
</beans>
```

Modifier la classe MonApplication pour instancier le bean (*new ClientController()*) en utilisant le conteneur ApplicationContext de Spring IOC.

ApplicationCongtext exits Spring-context dependency. We must import it from the org.springframework.context package.

```
package cigma.pfe;

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

public class MonApplication{
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("spring.xml");
    }
}
```

Ajouter des traces dans les constructeurs des trois beans pour vérifier que ApplicationContext procède aux instantiations au démarrage.

Add this constructor to the ClientController class.	<pre>public ClientController() {     System.out.println("Call ClientController ...."); }</pre>
Add this constructor to the	<pre>public ClientServiceImpl() {</pre>

ClientServiceImpl class.	<pre>System.out.println("Call ClientServiceImpl ...."); }</pre>
Add this constructor to the ClientRepositoryImpl class.	<pre>public ClientRepositoryImpl() {     System.out.println("Call ClientRepositoryImpl ...."); }</pre>

Après avoir ajouté les constructeurs, Exécuter la classe Mon Application.

```
Run: MonApplication x
"C:\Program Files\Java\jdk1.8.0_191\bin\java.exe" ...
Call ClientRepositoryImpl .... Called by new at service level
Call ClientServiceImpl .... Called by new at controller level
Call ClientController .... Called by ApplicationContext at MonApplication level
```

Changer le scope du bean ClientController à prototype dans le fichier spring.xml

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
    "https://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
    <bean id="controller" class="cigma.pfe.controllers.ClientController"
        scope="prototype"></bean>
</beans>
```

Exécuter la classe Mon Application. Remarquer qu' aucun bean n'est pas créé car Client Controller **est déclaré prototype**. Dans ce cas pour créer le bean, il faut utiliser la méthode *getBean* de l'application contexte comme suivant:

```
package cigma.pfe;

import cigma.pfe.controllers.ClientController;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MonApplication {
    public static void main(String[] args) {
        ApplicationContext context = new
        ClassPathXmlApplicationContext("spring.xml");
        ClientController ctrl = (ClientController)
        context.getBean("controller"); // controller est l'id dans le fichier
        Spring.xml
    }
}
```

Exécuter la classe Mon Application. Puis vérifier la création des beans à travers les traces des constructeurs.

## V. Injection des dependances (ID): Spring beans, Spring Context et Spring Core

Dossier des travaux pratiques. Module 1 : Java de base .Années scolaire 2021/2022. Niveau : Licence FST Settat

Professeur : M. Boulchahoub Hassan [hboulchahoub@gmail.com](mailto:hboulchahoub@gmail.com)

Mise à jour 18 Nov 2021



Nous remarquons la classe ClientController contient toujours l'attribut:

```
ClientService clientService = new clientServiceImpl();
```

On dit que la classe ClientController dépend de la classe ClientServiceImpl. Dans les bonnes pratiques récentes, cette dépendance ne doit pas être créée en utilisant le new keyword.

ApplicationContext container peut injecter cette dépendance (ID) en suivant les techniques suivantes:

### Injection des dépendances par Setter

Modifier le fichier Spring.xml comme suivant.

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
    "https://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
    <bean id="controller" class="cigma.pfe.controllers.ClientController"
        scope="prototype">
        <property name="clientService" ref="service"/>
    </bean>
    <bean id="service"
        class="cigma.pfe.services.ClientServiceImpl"
        scope="prototype"></bean>
</beans>
```

Dans la classe ClientController supprimer l'instanciation par le new. Créer le Setter de l'attribut ClientService. La classe ClientController devient:

```
package cigma.pfe.controllers;

import cigma.pfe.services.ClientService;
import cigma.pfe.models.Client;
import cigma.pfe.services.ClientServiceImpl;

public class ClientController {

    ClientService clientService;

    public void setClientService(ClientService clientService) {
        this.clientService = clientService;
    }

    public Client save(Client c){
        System.out.println("ClientController level...");
        return clientService.save(c);
    }

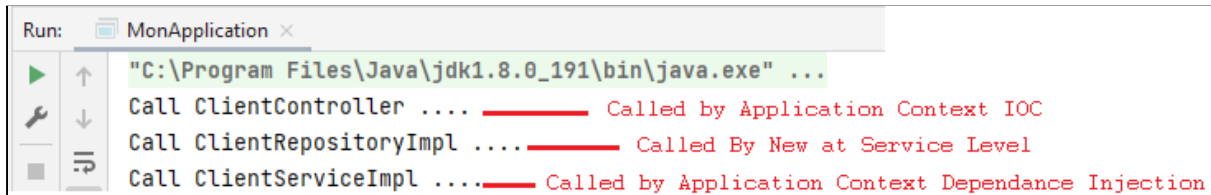
    public ClientController() {
```

```

        System.out.println("Call ClientController ....");
    }
}

```

Exécuter la classe Mon Application. Puis vérifier la création des beans à travers les traces des constructeurs.



### Injection des dépendances par Constructor

Il est aussi possible d'injecter les dépendances par constructeur.

Modifier le fichier Spring.xml comme suivant.

```

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
    "https://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
    <bean id="controller" class="cigma.pfe.contollers.ClientController"
        scope="prototype">
        <constructor-arg ref="service"/>
    </bean>
    <bean id="service" class="cigma.pfe.services.ClientServiceImpl"
        scope="prototype"></bean>
</beans>

```

Dans la classe ClientController supprimer l'instanciation par le new. Créer le constructeur qui prend un paramètre de type ClientService. La classe ClientController deviendra:

```

package cigma.pfe.contollers;

import cigma.pfe.services.ClientService;
import cigma.pfe.models.Client;
import cigma.pfe.services.ClientServiceImpl;

public class ClientController {

    ClientService clientService;
    public ClientController(ClientService clientService) {
        System.out.println("Call ClientController with clientService
param....");
        this.clientService = clientService;
    }

    public Client save(Client c ){
        System.out.println("ClientController level...");
        return clientService.save(c);
    }
    public ClientController() {
        System.out.println("Call ClientController ....");
    }
}

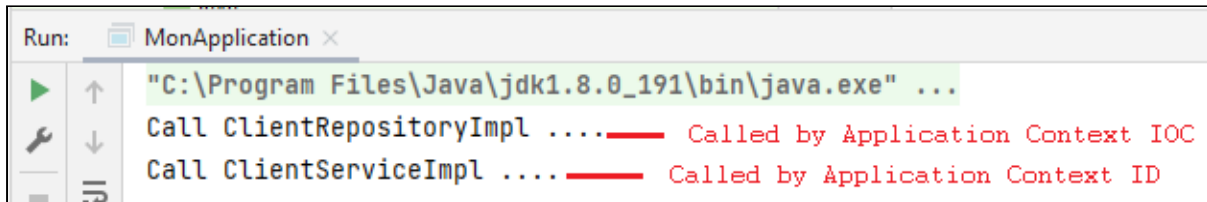
```

```

    }
}

```

Exécuter la classe Mon Application. Puis vérifier la création des beans à travers les traces des constructeurs.



```

Run: MonApplication x
"C:\Program Files\Java\jdk1.8.0_191\bin\java.exe" ...
Call ClientRepositoryImpl .... Called by Application Context IOC
Call ClientServiceImpl .... Called by Application Context ID

```

Nous remarquons que le constructeur sans paramètres n'est plus appelé par l'Application Context container. Ce dernier appellera le constructeur équivalent aux constructor-args cités dans le fichier de configuration spring.xml

```
<constructor-arg ref="service"/>
```

**Exercice** : Injecter la dépendance entre la classe ClientServiceImpl et ClientRepositoryImpl par Setter et par Constructor.

C-a-d : Éliminer l'utilisation de `new ClientRepositoryImpl();` dans la classe ClientServiceImpl

### Solution injection par constructeur :

1) Modifier la classe ClientServiceImpl en créant le constructeur qui prend Client Repository comme paramètre

```

ClientRepository clientRepository ;
public ClientServiceImpl(ClientRepository clientRepository) {
    System.out.println("Call ClientServiceImpl with ClientRepository param...");
    this.clientRepository = clientRepository;
}

```

Modifier le fichier de configuration spring.xml en ajoutant

- un bean pour Client Repository
- une référence par constructeur dans le bean Client Service

```

<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
    "https://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>

    <bean id="controller" class="cigma.pfe.controllers.ClientController"
scope="prototype">
        <constructor-arg ref="service"/>
    </bean>

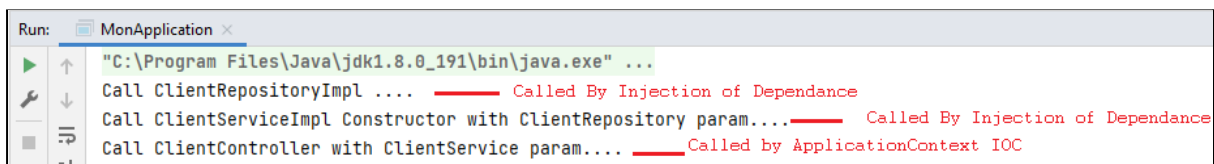
    <bean id="service" class="cigma.pfe.services.ClientServiceImpl"
scope="prototype">
        <constructor-arg ref="repository"/>
    </bean>

```

```
<bean id="repository"
class="cigma.pfe.repositories.ClientRepositoryImpl"
scope="prototype"></bean>

</beans>
```

Exécuter la classe Mon Application. Puis vérifier la création des beans à travers les traces des constructeurs.



```
Run: MonApplication x
"C:\Program Files\Java\jdk1.8.0_191\bin\java.exe" ...
Call ClientRepositoryImpl .... Called By Injection of Dependence
Call ClientServiceImpl Constructor with ClientRepository param.... Called By Injection of Dependence
Call ClientController with ClientService param.... Called by ApplicationContext IOC
```

Pour compléter l'application, ajouter à la méthode main de la classe MonApplication:

```
public class MonApplication {
    public static void main(String[] args) {
        ApplicationContext context = new
        ClassPathXmlApplicationContext("spring.xml");
        ClientController ctrl = (ClientController)
        context.getBean("controller"); // controller est l'id dans le fichier
        Spring.xml

        Client client = new Client(1, "ALAMI");
        ctrl.save(client);
    }
}
```

Prière de noter que pour le moment la seule utilisation du **new** keyword dans notre petite application existe dans la méthode main de la classe MonApplication.

```
Client client = new Client(1, "ALAMI");
```

Cette instanciation sera supprimée en utilisant l'API Jackson dans la suite de notre formation.