L’injection de dépendances

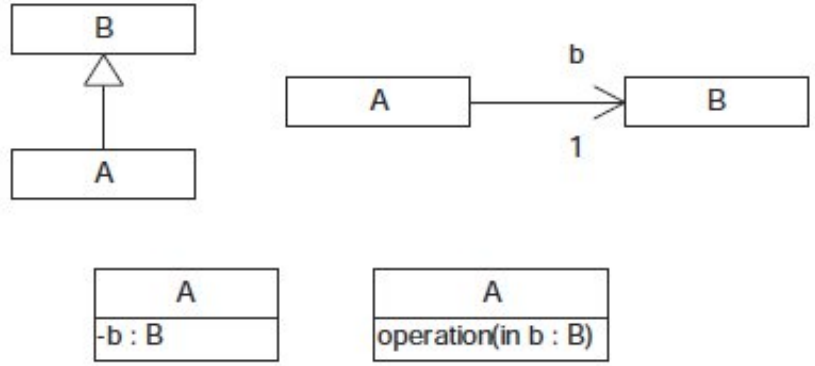
Définition

- Une classe A dépend d’une classe B si elle est liée par l’un de ses attributs, l’une de ses associations ou par l’une de ses méthodes à un ensemble de classes ou à une classe d’un ensemble de classes.
- Si A dépend de B, il n’est pas possible d’utiliser A sans disposer de B.
- La relation de dépendances est transitive : si A dépend de B et B dépend de C, alors A dépend de C

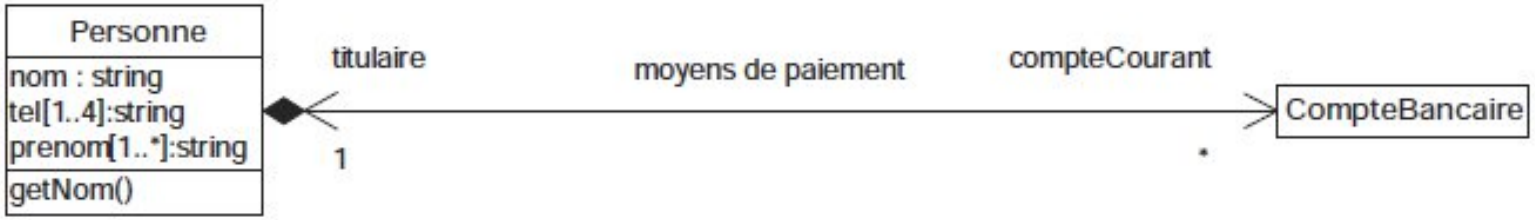
- Notation UML d’une dépendance entre 2 classes : une flèche pointillée

Cadre d’application

- On considère qu’une classe A dépend de B si et seulement si :
  - A hérite de B
  - A est associée à B et l’association est au moins navigable de A vers B
  - A possède un attribut dont le type est B
  - A possède une méthode dont le type de l’un des paramètres est B



- Exemple de dépendances mutuelles entre 2 classes (navigabilité dans les 2 sens)



Principe

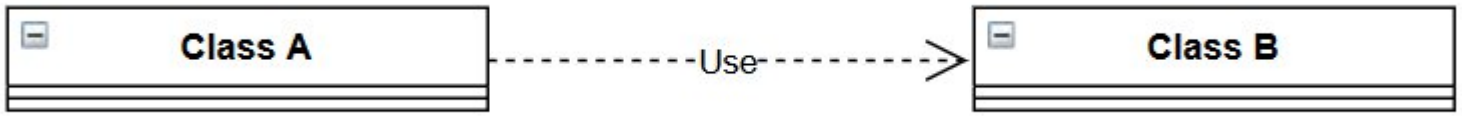
- Afin d'éviter de tout avoir dans un seul fichier, le développeur Java écrit plusieurs classes (une classe par fichier). Cela permet non seulement d'assurer une meilleure lisibilité mais aussi une meilleure maintenance.
- Lorsque l'on passe en plusieurs classes, on souhaite pouvoir enlever une classe et la remplacer par une autre classe : Cela n'est malheureusement pas possible en java de base, il faut passer par de l'injection de dépendances. Cela permet entre autre "d'échanger" une classe par une autre et d'avoir plusieurs configurations d'applications.

  Il existe 4 types d'injections de dépendances :

  - Injection par constructeur
  - Injection par interface
  - Injection par mutateur
  - Injection par champs

Injection à la main

- Lors de la séparation d'un code en plusieurs classes, nous avons une dépendance directe d'une classe a une autre.
- Nous séparons le code général en une classe **A** et une classe **B**. La classe **A** doit utiliser une méthode de la classe **B**.



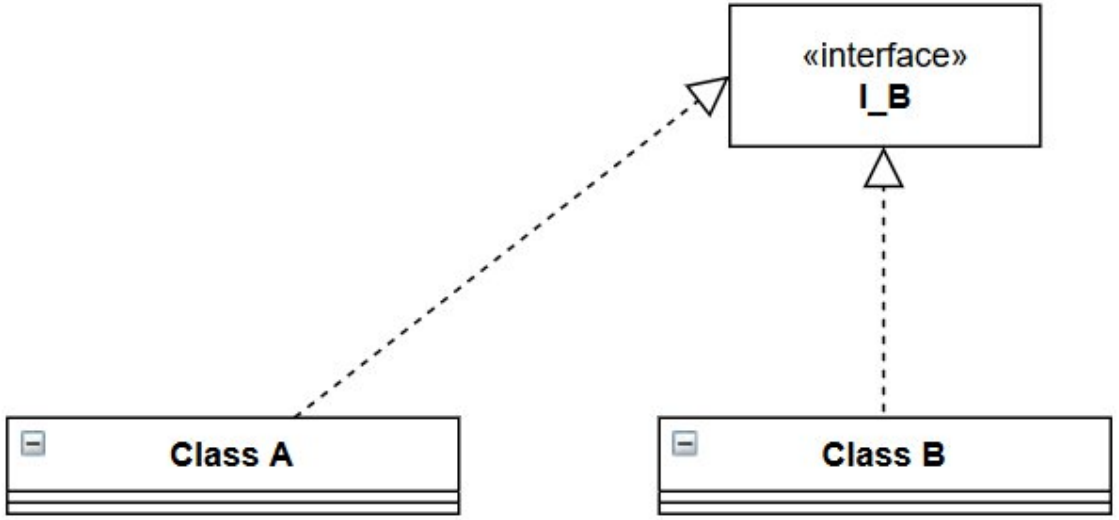
Exemple :

```
public class A {
    B b = new B();
    b.someMethod();
  }
```

Injection à la main : les interfaces

- Moyen simple de gérer les dépendances.
- Les interfaces en java permettent de définir des méthodes et leurs paramètres sans en définir le code.
- Ces interfaces sont ensuite implémentées par les classes dont on dépend.
  Ici, on crée une interface **I_B** définissant la méthode *someMethod()* et la classe **A** ne dépend donc plus de **B** directement mais de **I_B** qui peut être implémentée par n'importe quelle autre classe :

```
public class A implements I_B {
    public void someMethod();
}
```



Les interfaces

Avantages :

Toujours rapide à développer

Possibilité de changer d'implémentation

Inconvénients :

Dépendance toujours là

Disperse les dépendances dans le code

Remarque : Injection à la main par la super classe

- On peut procéder de la même manière en « déplaçant » la méthode *someMethod()* dans une super classe de B que l’on ajoute, au lieu d’utiliser une interface.
- Mêmes avantages et inconvénients que pour les interfaces.

Les interfaces Java

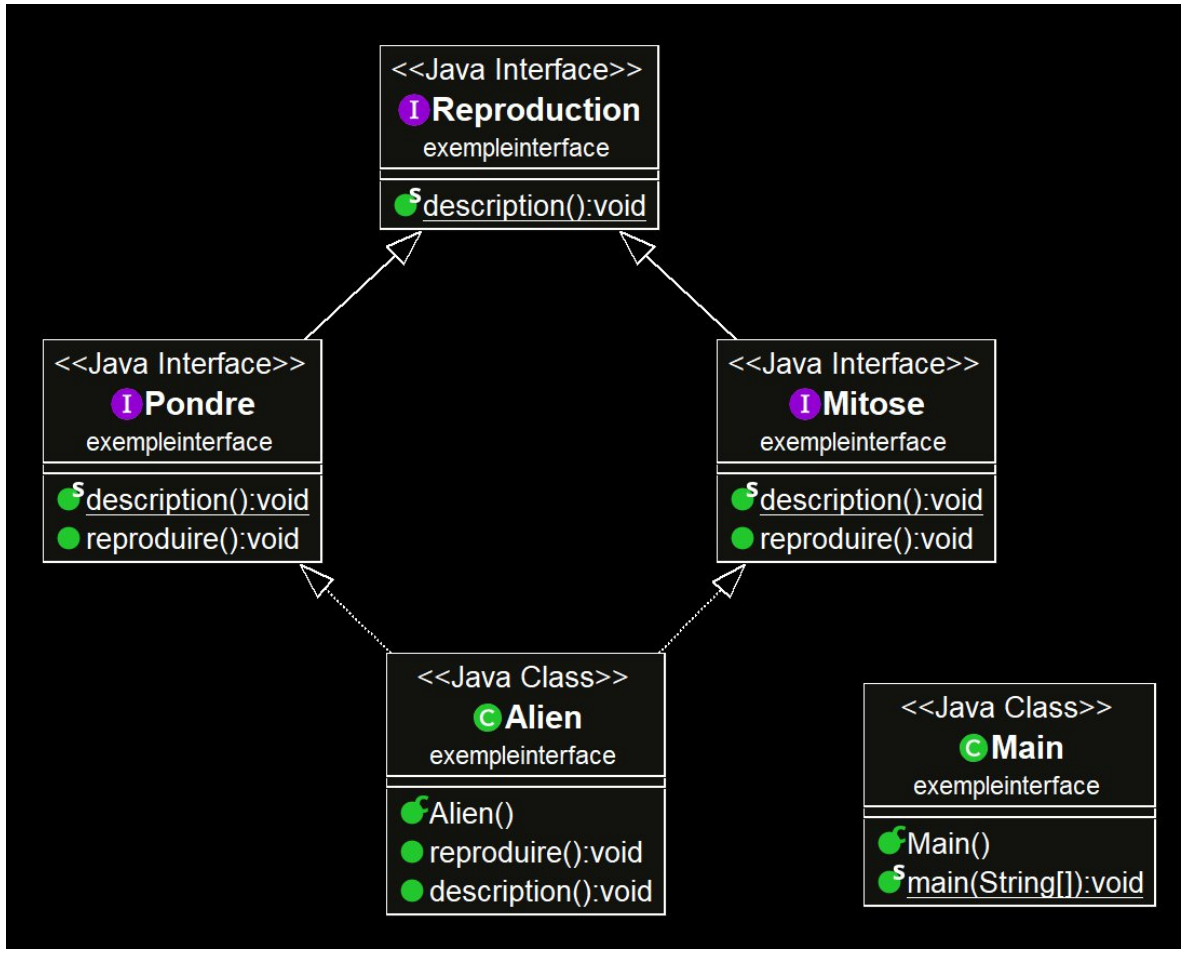
Avant Java 8 :

Ajouter une méthode dans une interface signifie repasser sur toutes les classes qui implémentent l’interface pour la redéfinir

A partir de Java 8 :

- Une interface n’est plus une classe 100% abstraite
- Elle peut contenir du code sous deux formes :
  - Avec des méthodes statiques
  - Avec une définition par défaut d’une méthode

Les interfaces Java



Pondre.java | Mitose.java | Reproduction.java

```java
package exempleinterface;

public interface Pondre extends Reproduction {

    public static void description() {
        Reproduction.description();
        System.out.println("Redéfinie dans Pondre.java");
    }

    default void reproduire() {
        System.out.println("Je ponds des oeufs !");
    }

}
```

compatibilité avec les interfaces
lors de l’ajout de méthodes dans celles-ci

Outils d’injection de dépendances

Des design patterns ont une utilité dans le découplage et la maintenance du code tels que **Factory** et **Facade**, mais ils ne présentent pas le même intérêt par rapport aux Mocks.

Outils pour les Mocks : Spring, Dagger 2, EasyMock, …

Framework EasyMock

Créer un projet Maven et ajouter une dépendance EasyMock (attention à la version JRE)

```
<dependency>
  <groupId>org.easymock</groupId>
  <artifactId>easymock</artifactId>
  <version>4.2</version>
  <scope>test</scope>
</dependency>
```

Exemple EasyMock (http://easymock.org/getting-started.html)

ClassTested.java

```java
package exempleeasymock;

public class ClassTested {

    private Collaborator listener;

    public void setListener(Collaborator listener) {
        this.listener = listener;
    }

//  @Override
//  public void documentAdded(String title) {
//      // TODO Auto-generated method stub
//  }

    public void addDocument(String title, String document) {
        listener.documentAdded(title);
    }
}
```

Exemple EasyMock

1. Créez le mock
2. Mettez-le sur la classe testée
3. Enregistrez ce que vous voulez que le mock fasse
4. Dire à tous les mocks que nous allons faire le test
5. Faire le test
6. Assurez-vous que ce qui devait être appelé l’a été

Nous testons que documentAdded() est appelée seulement une fois et reçoit le paramètre exact.
Tout autre appel vers notre mock est un test d’échec.

ExampleTest.java | ClassTested.java | Collaborator.java

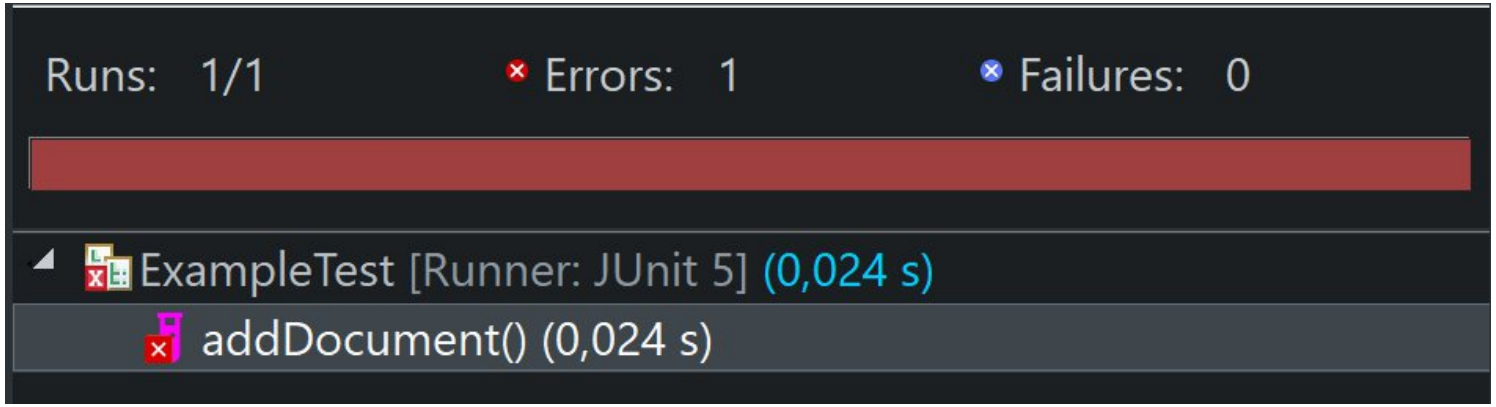
```java
package exempleeasymock;

import org.easymock.EasyMockRule;
import org.easymock.EasyMockSupport;
import org.easymock.Mock;
import org.easymock.TestSubject;
import org.junit.Rule;
import org.junit.jupiter.api.Test;

class ExampleTest extends EasyMockSupport {

    @Rule
    public EasyMockRule rule = new EasyMockRule(this);

    @Mock
    private Collaborator collaborator; // 1

    @TestSubject
    private ClassTested classUnderTest = new ClassTested(); // 2

    @Test
    public void addDocument() {
        collaborator.documentAdded("New Document"); // 3
        replayAll(); // 4
        classUnderTest.addDocument("New Document", "content"); // 5
        verifyAll(); // 6
    }
}
```

Exemple EasyMock

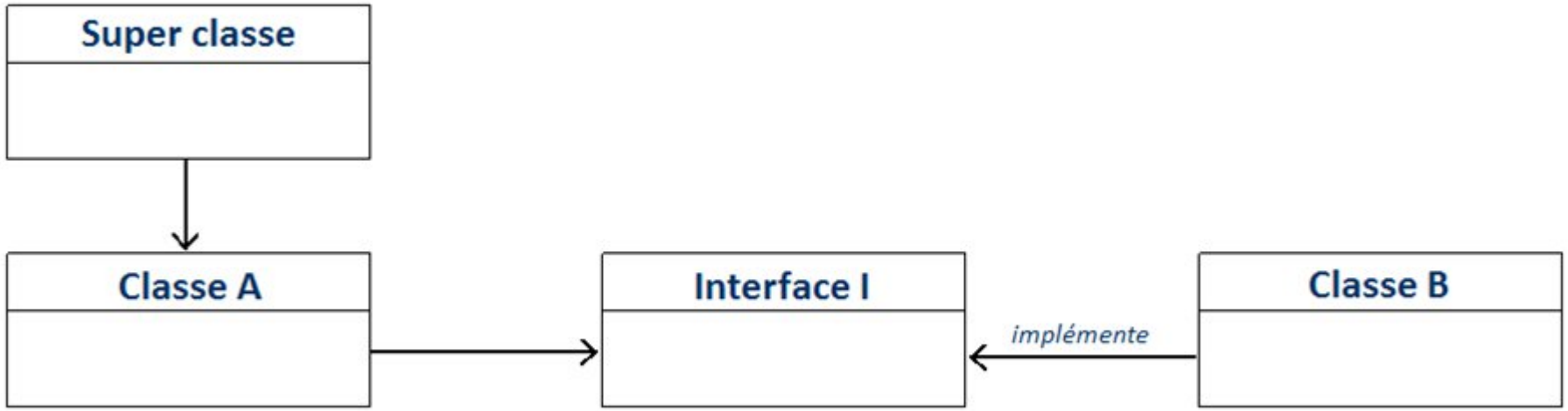
Fonctionne avec JUnit 4
(pas @Rule avec JUnit 5)
JUnit 5 fonctionne avec le JRE 8

ExampleTest.java | ClassTested.java | Collaborator.java

```
1 package exempleeasymock;
2
3 public interface Collaborator {
4     void documentAdded(String title);
5 }
```



Injection par constructeur

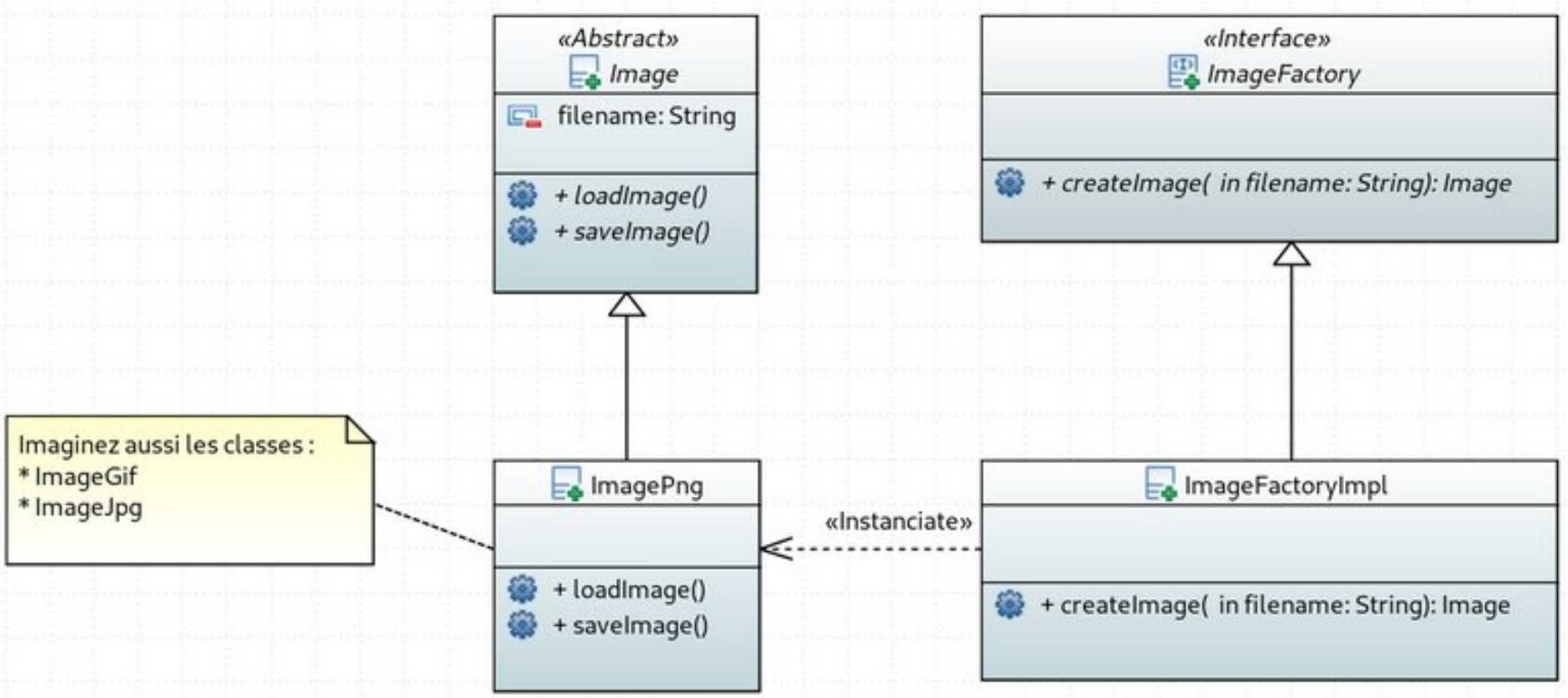



```
public class A {
  I inst;

  public A(I inst) {
    this.inst=inst;
  }

  public void doWork() {
    inst.someMethod();
  }
}
```

```
public static void main(String[] args) {
  A a = new A(factory().getDependency());
  a.doWork();
}
```

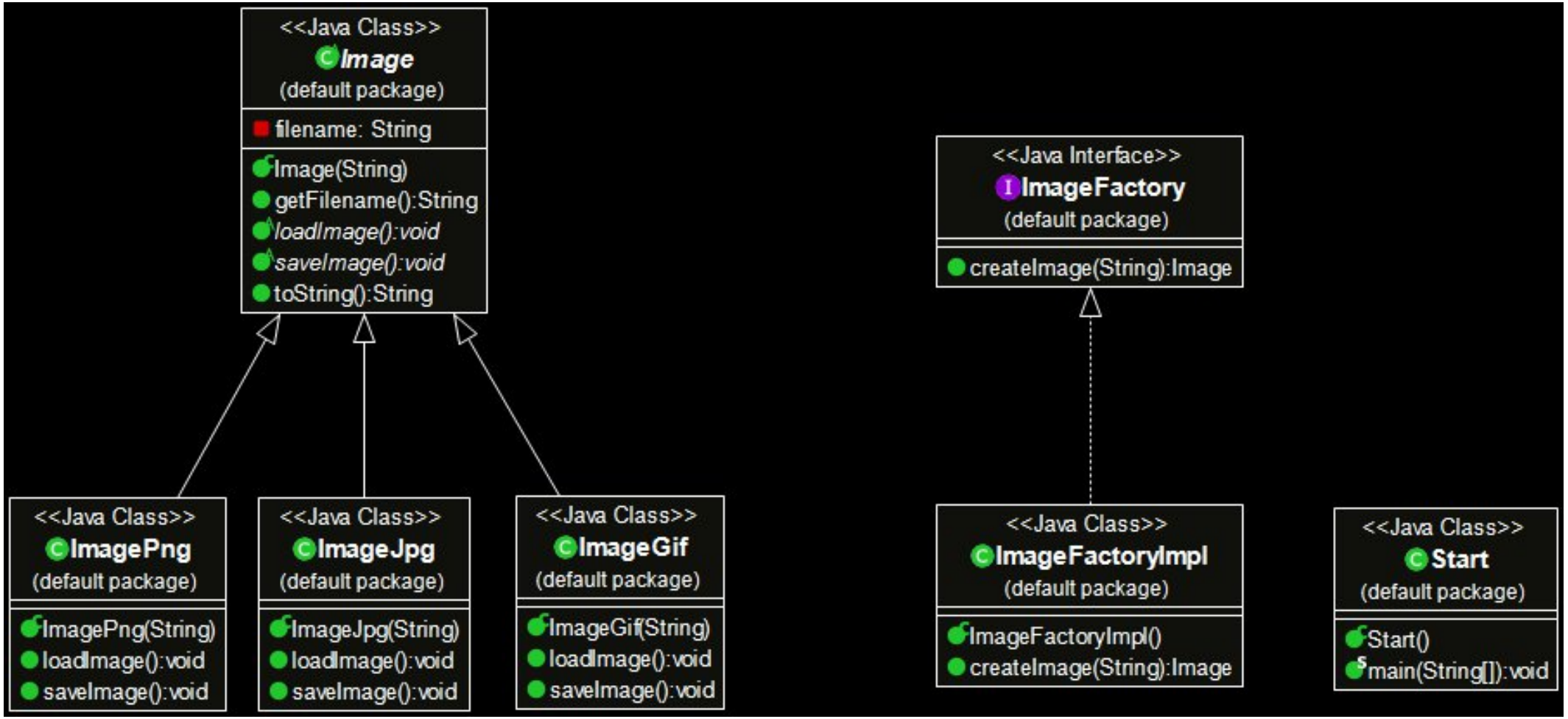
Conseil : utiliser des frameworks pour l’injection !

Pattern Factory Method (fabrique d’images)

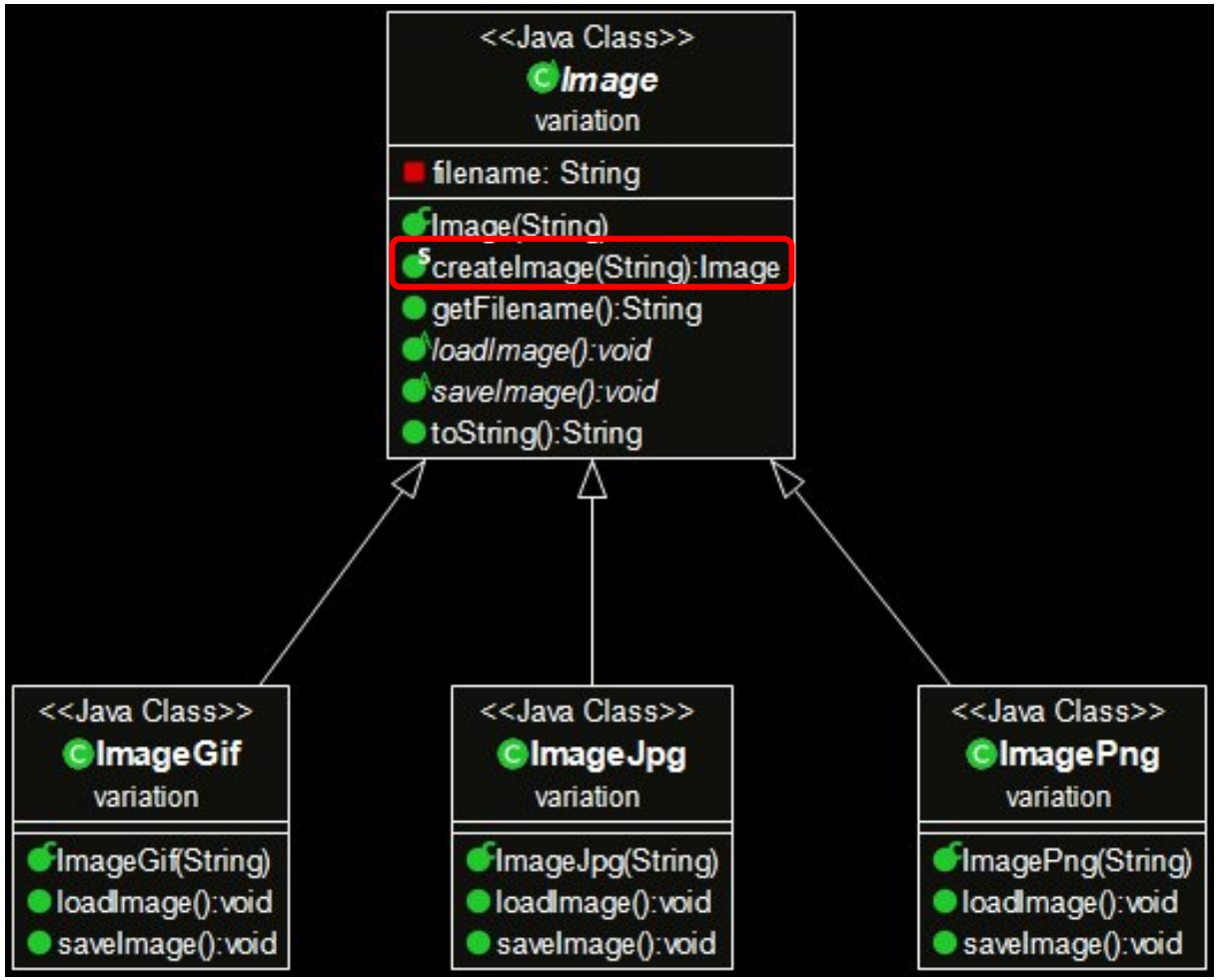


Ce design pattern permet de créer une instance à partir d'une classe dérivée d'un type abstrait.

Pattern Factory Method



Variation Pattern Factory Method



createImage() : méthode statique