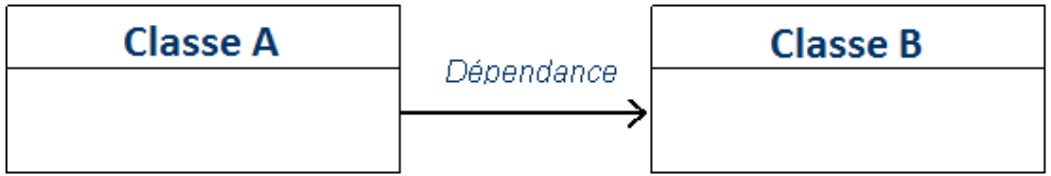
Injection de dépendances

Un objet A dépend d'un objet B si :

- **Il existe au moins 1 champ de type B dans A**
- **A hérite de B**
- **Une méthode de A appelle une méthode de B**



Comme A dépend de B, il faut importer B dans A.
Si l'implémentation des éléments public de B changent, alors celle de A devra être modifiée en conséquences.
Dans un gros projet, le code devient moins maintenable :

- Des effets de bord difficiles à identifier
- Des modifications en cascade entre les dépendances.

L'injection de dépendances est un moyen de résoudre ces problèmes.
Cela **consiste à être capable de fournir une instance d'une classe donnée en fonction d'une « recette » pour créer l'instance en question**.
On peut demander une instance de n'importe quel objet depuis n'importe quelle classe.

Exemples d'utilisation :

```
var registry = new InjectorRegistry();
registry.registerInstance(String.class, "hello");
```

```
var bob = new Person("Bob");
registry.registerInstance(Person.class, bob);
```

Recette

Injection de dépendances : Protocoles d'injection

<u>Il existe 3 protocoles connus d'injection de dépendances :</u>

Par constructeur ⇒ meilleure façon :

- Le constructeur annoté est appelé avec les bons arguments
- on peut faire des classes immuables

```
record Point(int x, int y) {}

class Circle {
  private final Point center;
  private String name;

  @Inject
  public Circle(Point center) {
    this.center = center;
  }
```

Par setter ⇒ Inconvénient :

Les setters peuvent être appelés dans n'importe quel ordre (après appel du constructeur par défaut)

```
@Inject
public void setString(String s) {
  this.s = s;
}
@Inject
public void setInteger(Integer i) {
  this.i = i;
}
```

Par champ/attribut ⇒ Mauvaise manière de faire de l'injection :

Utilise la deep reflection pour remplir les champs.

Problème : Accès et modification des champs privés d'une classe à l'extérieur de celle-ci.

Plusieurs utilisation possibles :

- Configurer l'injecteur pour y ajouter automatiquement les dépendances via :
  - Un fichier XML
  - Un scan des classes annotées au démarrage de l'application (démarrage lent).
- Créer une **API pour enregistrer manuellement les recettes** associées aux dépendances et les récupérer.

Injection de dépendances : Enregistrer et récupérer une instance pour un type donné

Enregistrement

On associe l'instance à enregistrer avec son type dans une **HashMap<Class<?>, Object>**.
Chaque type est une classe quelconque (**Class<?>**) et est **associé à une instance**.
Il faut une méthode qui assure que l'instance à enregistrer soit une instance du type passé en paramètre ➔ utilisation d'un type paramétré pour les 2 paramètres.

Prototype : public <T> void registerInstance(Class<T> type, T instance)

Récupération

Récupérer si elle existe dans la HashMap **l'instance associée au type donné**.
L'instance en question est de type Object dans la HashMap, mais il faut qu'elle soit **cast dans le même type que le type passé en paramètre** de la méthode :
➔ Utilisation d'un type paramétré pour appeler la méthode **cast()** afin de cast l'instance récupérée dans le bon type.

Prototype : public <T> T lookupInstance(Class<T> type)

Utilisation de cast : type.cast(instance);

Exemple d'utilisation :

```
var registry = new InjectorRegistry();
registry.registerInstance(Point.class, new Point(0, 0));
var circle = registry.lookupInstance(Circle.class);
System.out.println(circle.center);  // Point(0, 0)
```

Injection de dépendances : Enregistrer un Supplier pour un type donné

Modification de la Map qui stockait des instances pour stocker des Suppliers :
HashMap<Class<?>, Supplier<?>>

Remarque : Supplier<?> équivalent à : Supplier<? extends Object> ⇒ Stocke un supplier de n'importe quel type.

On veut également garantir que le Supplier retourne une instance qui est un sous-type du type donné en premier paramètre de la méthode (**le même type ou un sous-type**)

Prototype : `public <T> void registerProvider(Class<T> type, Supplier<? extends T> supplier)`

Refactoring

RegisterInstance : Appelle juste **registerProvider** avec un supplier : **registerProvider(type, () -> instance);**

LookupInstance : Appelle le Supplier récupéré avec **get()** avant de le cast : **type.cast(instance.get());**

Exemple d'utilisation :

```
registry.registerProvider(String.class, () -> "hello");
var circle = registry.lookupInstance(Circle.class);
System.out.println(circle.center);  // Point(0, 0)
System.out.println(circle.name);  // hello
```

Injection de dépendances : Récupérer les propriétés Bean d'une classe qui a des setters annotés

Il faut créer une annotation pour que l'utilisateur puisse annoter les setters et le constructeur injectables :

```
@Retention(RUNTIME)
@Target({METHOD, CONSTRUCTOR})
public @interface Inject { }
```

Retention ⇒ 3 types : RUNTIME, SOURCE et CLASS
L'annotation est juste pour le compilateur ou elle existe aussi en runtime ?
Exemple :
@Override ⇒ retention SOURCE : elle n'existe pas en runtime
@Inject ⇒ retention RUNTIME : on récupère les setters et constructeur annotés lors de l'exécution.

Target ⇒ Sur quoi on a le droit de mettre l'annotation.
Ici, on peut mettre @Inject sur une méthode ou un constructeur.

Créer un stream pour récupérer les propriétés souhaitées dans une **List<PropertyDescriptor>** :

1) Récupérer les **propriétés Bean** de la classe souhaitée ⇒ **BeanInfo.getPropertyDescriptors()**
2) Filtrer ces propriétés pour ne garder que celles dont le **setter est annoté par @Inject** :

- **property.getWriteMethod()** (classe **PropertyDescriptor**) ➔ **récupérer le setter du Bean**
- **AnnotatedElement.isAnnotationPresent(annotation)**
  - **Retourne true si l'appelant (un objet pouvant être annoté) est annoté avec l'annotation donnée.**
- On a donc un **filter()** qui prend le **Predicate** suivant :
  - **setter != null && setter.isAnnotationPresent(Inject.class)**

Injection de dépendances : Implémenter l'injection par setter

On veut toujours garantir que la classe à enregistrer soit un sous-type du type auquel elle est associée :
Prototype : **public <T> void registerProviderClass(Class<T> type, Class<? extends T> providerClass)**

On va écrire un Supplier pour enregistrer l'instance créée à partir du constructeur par défaut :

1) A l'extérieur de la lambda :
- On commence par **récupérer le constructeur par défaut de la classe à enregistrer** (**providerClass**) :
- Appel de **Class.getConstructor** sans paramètre pour récupérer le constructeur qui ne prend pas d'argument :
  - **providerClass.getConstructor()** ⇒ retourne le default **Constructor**
- **Récupérer les propriétés de la classe** (voir slide précédente).

2) Dans la lambda :
- On **créer l'instance** via la méthode : **Constructor.newInstance()** ➔ **Sans arguments**
- Pour chaque setter annoté :
- On récupère **son unique paramètre** en appelant **lookupInstance** avec le type de ce paramètre que l'on récupère via :
  - **setter.getParameterTypes()[0]** ➔ **renvoie le type du paramètre prit par la Method appelante (le setter)**

- On appelle le setter de la nouvelle instance : **invokeMethod(instance, setter, arg) <=>** instance.setter(arg)
- On **cast la nouvelle instance** pour qu'elle soit du même type que le premier paramètre de **registerProviderClass**

Injection de dépendances : Implémenter l'injection par Constructeur

NB : On suppose que l'utilisateur a enregistré une recette associée à chaque argument du constructeur de la classe.

1) Faire un stream à partir des constructeurs de la classe récupérables via : class.getConstructors()

- Récupérer le **constructeur annoté en filtrant avec : constructor.isAnnotationPresent(Inject.class)**
- **Si il existe plusieurs constructeurs annotés dans le stream :**
  - **reduce** ➔ permet de récupérer 2 constructeurs annotés dans un **BinaryOperator<Constructor>**
  - On peut donc lever une **IllegalStateException** dans le reduce
- Si aucun constructeur n'est annoté par **@Inject** on **récupére le default Constructor** via :
  - **beanType.getConstructor()**

2) Récupérer les arguments du constructeur :

- **Récupérer les types de ses arguments ➔ constructor.getParameterTypes()**
- Faire un stream **à partir des types pour récupérer les arguments** en appelant **lookupInstance** pour chaque type :
  - **map(this::lookupInstance)**
    - **➔ récupère les arguments en fonction de leurs types via les recettes enregistrées**

3) Créer l'instance et appeler les setters

- **Créer l'instance en appelant le constructeur avec ses arguments** : **constructor.newInstance(arguments)**
- On appelle **chaque setter annoté avec son paramètre respectif** (voir slide précédente)
- On **cast la nouvelle instance** pour qu'elle soit du même type que le premier paramètre de **registerProviderClass**