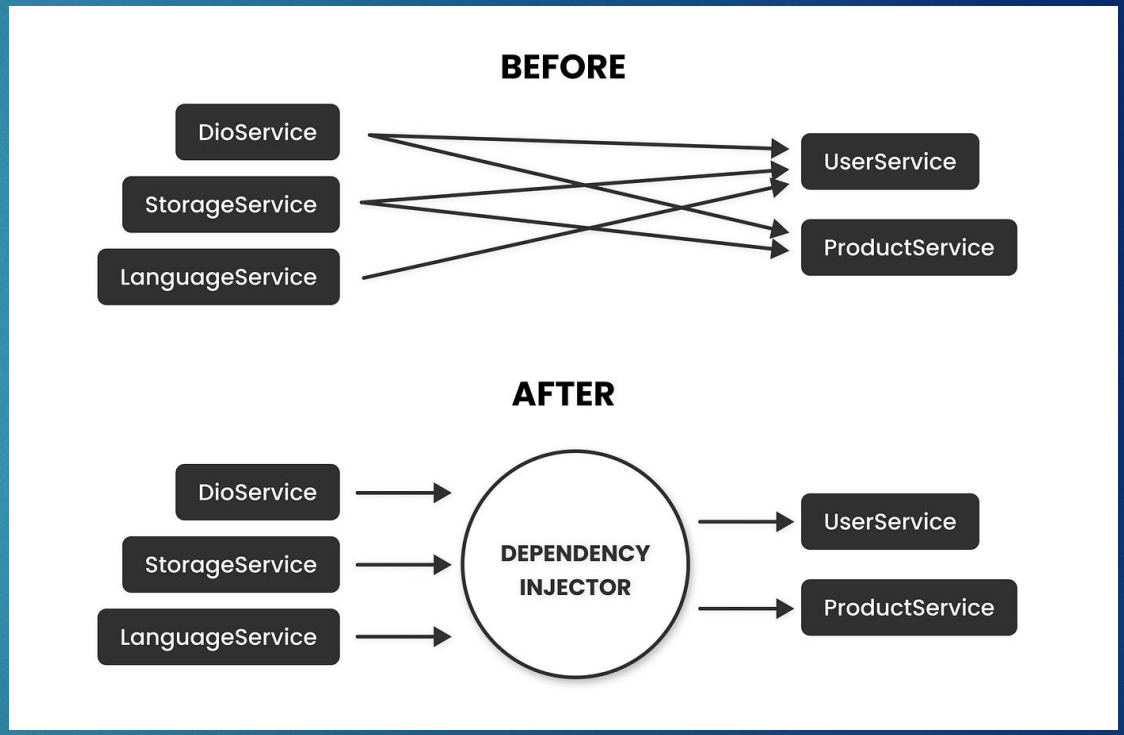
Injection de dépendances

- Établir les liens entre les classes / interfaces pour faciliter le
- Récupération des champs, méthodes, constructeurs et l'instance de la classe
- Fournir les objets nécessaires à un autre (ex : Services)
- Ce dernier n'a donc pas besoin de les construire et peut directement les utiliser
- Annotation @Inject



Création d'un registre d'injection


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

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

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

     @Inject
     public void setName(String name) {
          this.name = name;
     }
}
```

- 👉 Historiquement injections via les setters
- 👉 Injections via les constructeurs privilégiées (i.e. records, objets non mutables)
- 👉 Permettre l'enregistrement d'une instance via elle-même, une lambda, une java bean
- 👉 Seulement un seul constructeur peut être injecté sinon exception levée
- 👉 Constructeur par défaut sans arguments utilisé s'il n'y aucune annotation @Inject

Injection d'un constructeur


```
public <T> void registerProviderClass(Class<T> type, Class<? extends T> providerClass) {

    // Récupération du constructeur tagué avec @Inject
    var constructor = findInjectableConstructor(providerClass);

    // Récupération des setters injectables via une méthode locale
    var injectableProperties = findInjectableProperties(providerClass);
    var parametersTypes = constructor.getParameterTypes();

    // Ajout de l'instance au registre via la méthode prenant une lambda (supplier) en argument
    registerProvider(type, () -> {

        // Récupération des éventuels arguments du constructeurs
        var args = Arrays.stream(parametersTypes).map(this::lookupInstance).toArray();

        // Création de l'objet (équivalent à l'instruction new Truc())
        var instance = Utils.newInstance(constructor, args);

        // Ici c'est équivalent aux this.truc = truc des constructeurs (initialisation des champs)
        for (var property: injectableProperties) {
            var propertyType = property.getPropertyType();
            var value = lookupInstance(propertyType);
            Utils.invokeMethod(instance, property.getWriteMethod(), value); // appel du setter
        }
        return providerClass.cast(instance); // (cf. slide 6)
    });
}
```

Types paramétrés (Generics)

```java
public <T> void registerInstance(Class<T> type, T instance) {
     Objects.requireNonNull(type);
     Objects.requireNonNull(instance);
     var oldValue = registry.putIfAbsent(type, instance);
     if (oldValue != null) throw new IllegalStateException("injector for type " + type + " already exists");
}
```

- 👉 Code réutilisable par plusieurs types différents
- 👉 Permet la vérification des types à la compilation
- 👉 Type exact non définit, au contraire d'Object
- 👉 Utilisés par les collections, les map et de nombreuses API
- 👉 Ici pour tout type T, on doit avoir en 1er paramètre sa classe et une instance de T en 2ème paramètre

Wildcards


```java
// Un des constructeur d'ArrayList
public ArrayList(Collection<? extends E> c) {
    . . .
}

// Question 7 TP
public void registerProviderClass(Class<?> providerClass) {
    registerProviderClassImpl(providerClass);
}
```

- MyGenerics<? extends E> : le type E et tous ses sous-types
- MyGenerics<? super E> : le type E et tous ses types parents
- MyGenerics<?> : n'importe quel type (équivalent à <? extends Object>)
- Les types appliqués dans les generics sont en général des interfaces

  Donc choix possible entre plusieurs implémentations

Runtime type VS static type VS cast


```java
class Animal {}

class Monkey extends Animal {}

public class Test {

     public static void main(String[] args) {
          Animal animal = new Monkey();
          Class runtimeType = animal.getClass();
          Class staticType = Animal.class;
          System.out.println(runtimeType); // Monkey
          System.out.println(staticType); // Animal
          System.out.println(runtimeType == staticType); // false
     }
}
```

- A.class != a.getClass()
- Java typé statiquement
- Types -> Compilation
- Instances -> Exécution
- L'objet Class permet la « réflexion », c-à-d modifier les propriétés d'un objet à l'exécution (instance)

```java
providerClass.cast(instance);
```

- Cast sécurisé via une méthode prévu dans la classe « Class »
- Type vérifié à l'exécution permettant d'éviter les exceptions en runtime
- Cast classique pas sécurisé dans le cas des generics

Règles à respecter en Java

```
private final HashMap<Class<?>, Supplier<?>> registry = new HashMap<>();

static List<PropertyDescriptor> findInjectableProperties(Class<?> type){
    . . .
}

private Constructor<?> findInjectableConstructor(Class<?> type) {
    var constructors = Arrays.stream(type.getConstructors())
        .filter(constructor -> constructor.isAnnotationPresent(Inject.class))
        .toList();
    return switch (constructors.size()) {
        case 0 -> Utils.defaultConstructor(type);
        case 1 -> constructors.getFirst();
        default -> throw new IllegalStateException("multiple constructors annotated with @Inject");
    };
}
```

- 👉 Pas de typage avec l'interface pour les champs privés
- 👉 Visibilité package pour les méthodes utilitaires appelées autre part que leur classe
- 👉 RequireNonNull() uniquement dans les méthodes publiques
- 👉 ISE -> Mauvais état de l'objet, IAE -> Mauvais argument