SPRING AND DEPENDENCY INJECTION

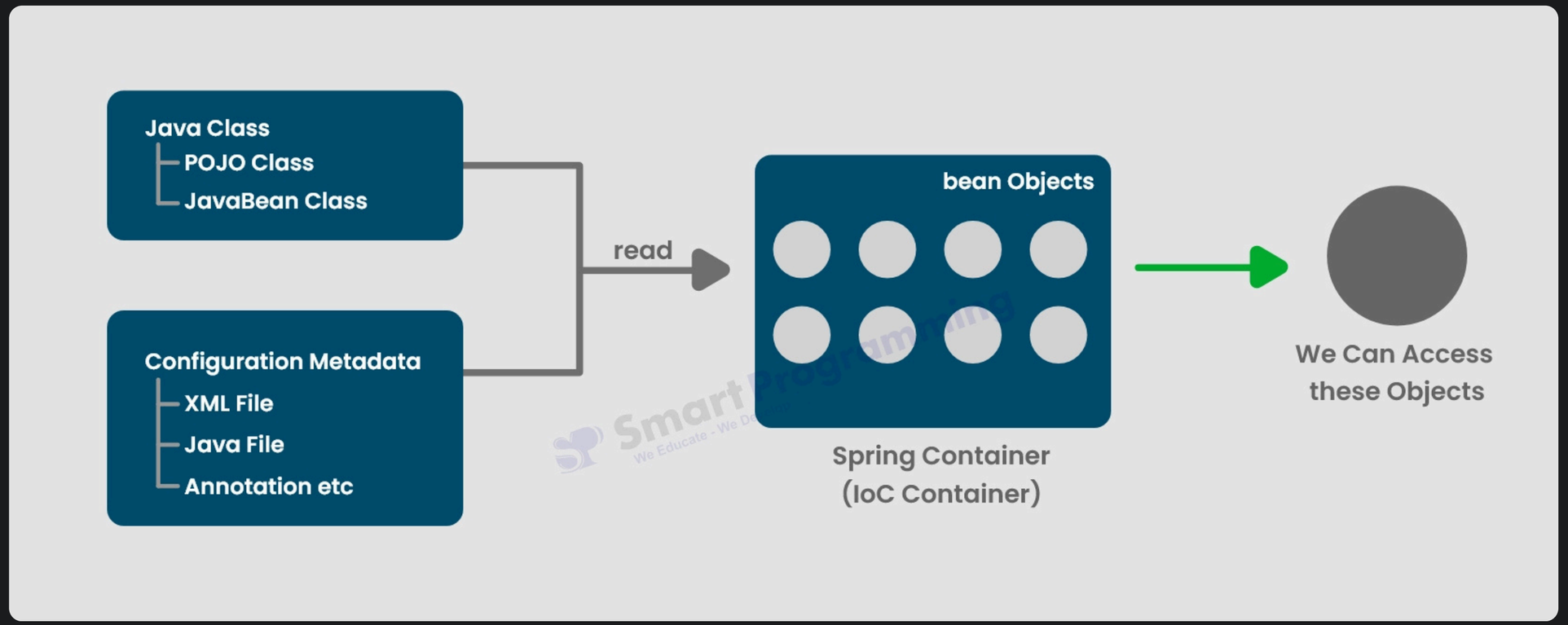
@med Mohammed Ezzaim

INTRODUCTION

Dependency Injection

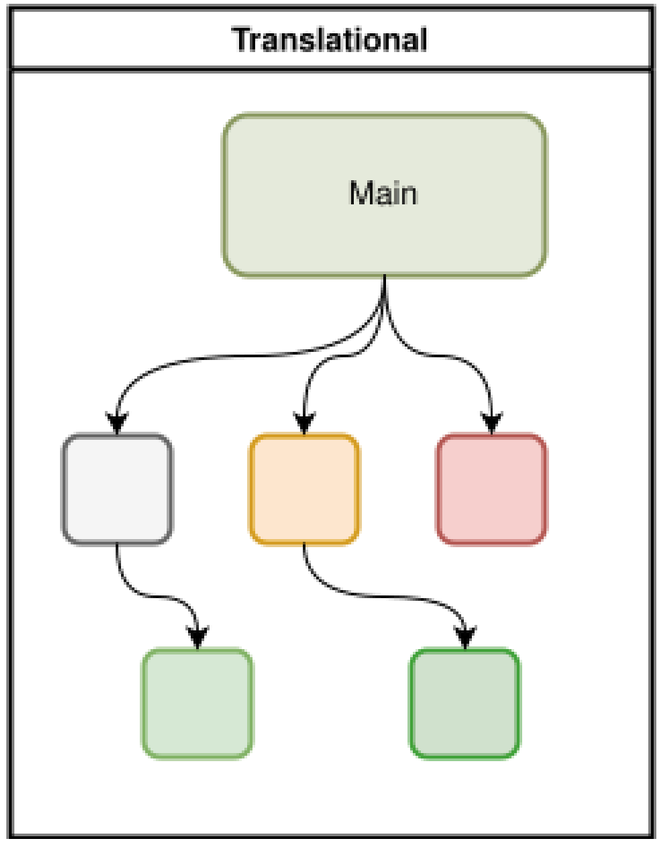
Qu'est-ce que l'IoC ?

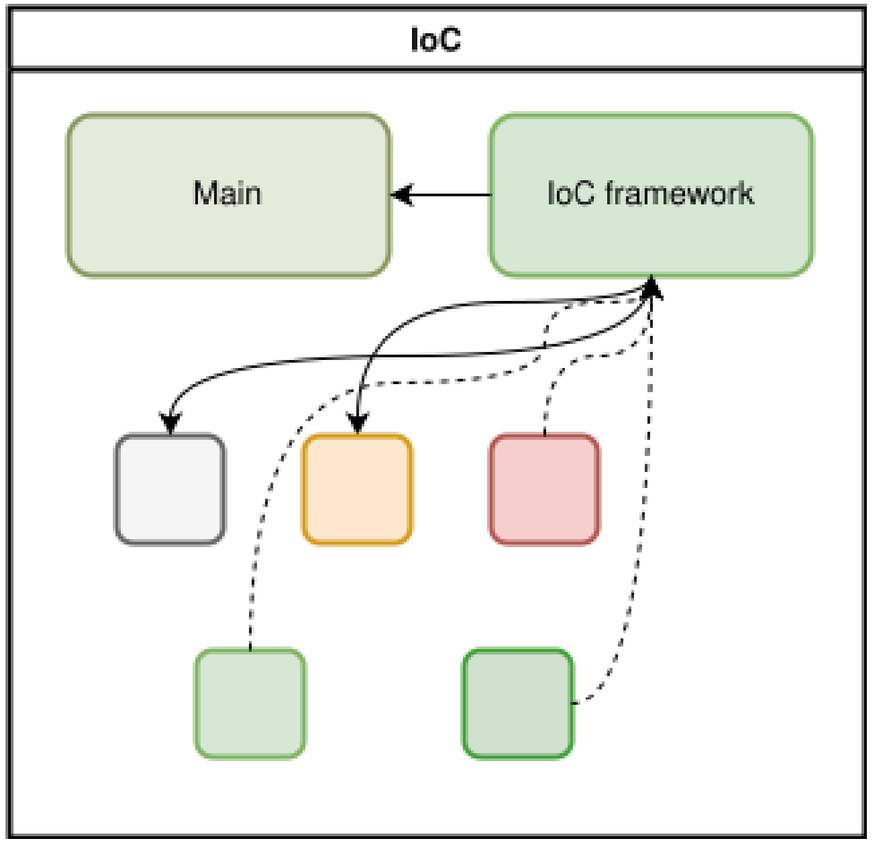
Inversion of Control signifie que ce n'est plus l'application qui crée et gère les objets, mais le conteneur Spring. Cela permet de découpler les classes et d'améliorer la testabilité et la maintenabilité.



Dependency Injection (DI):

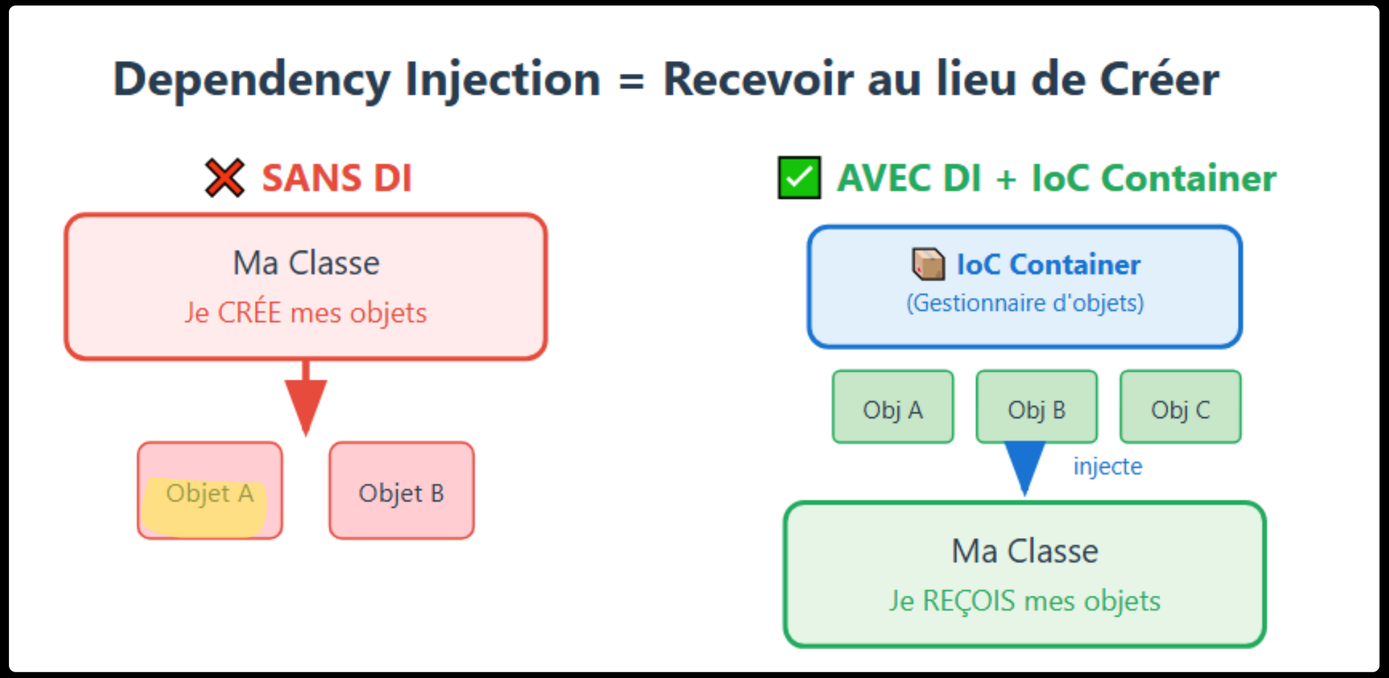
Dependency Injection (DI) est un design pattern qui permet d'injecter les dépendances d'un objet plutôt que de les créer directement dans la classe.

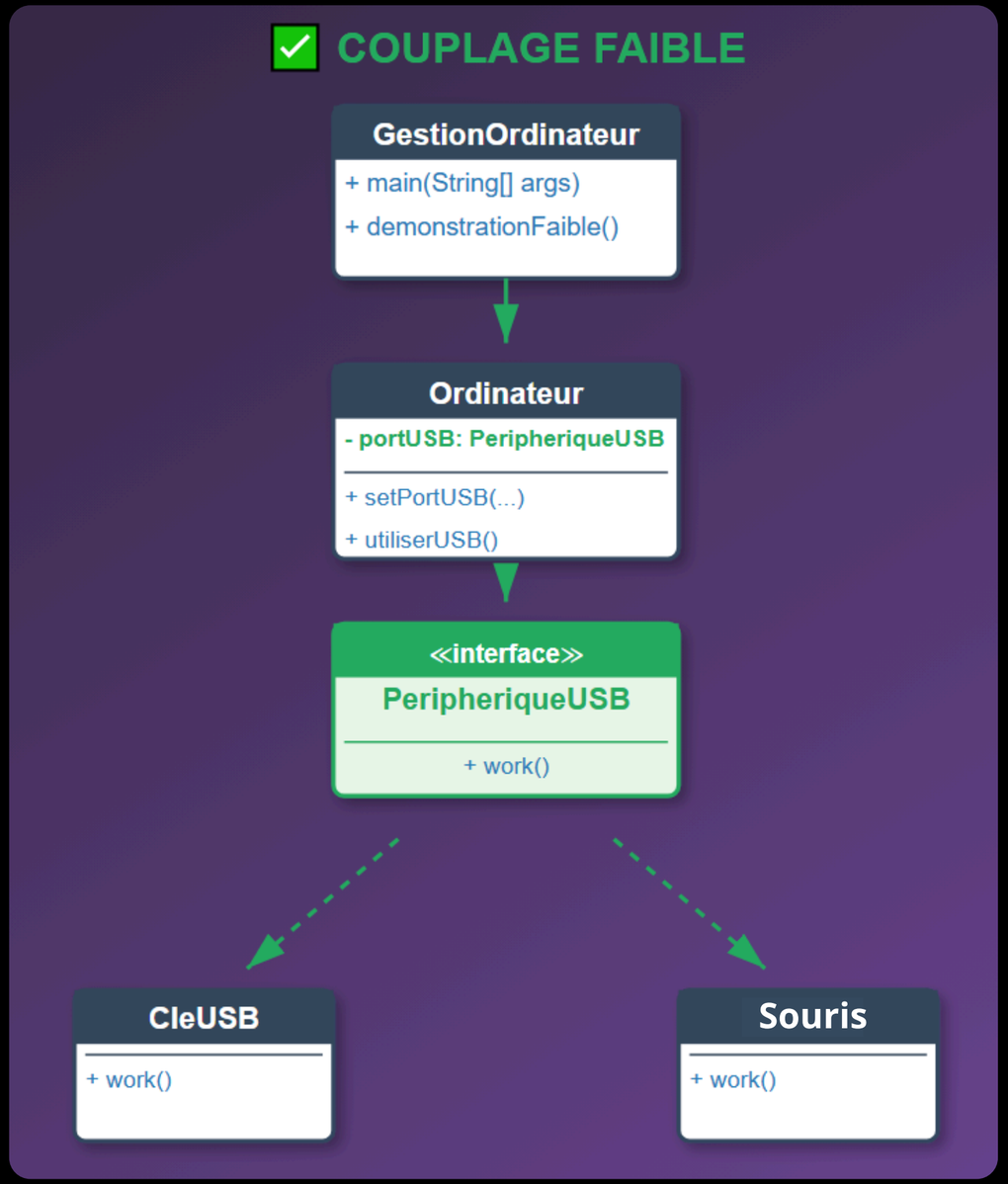


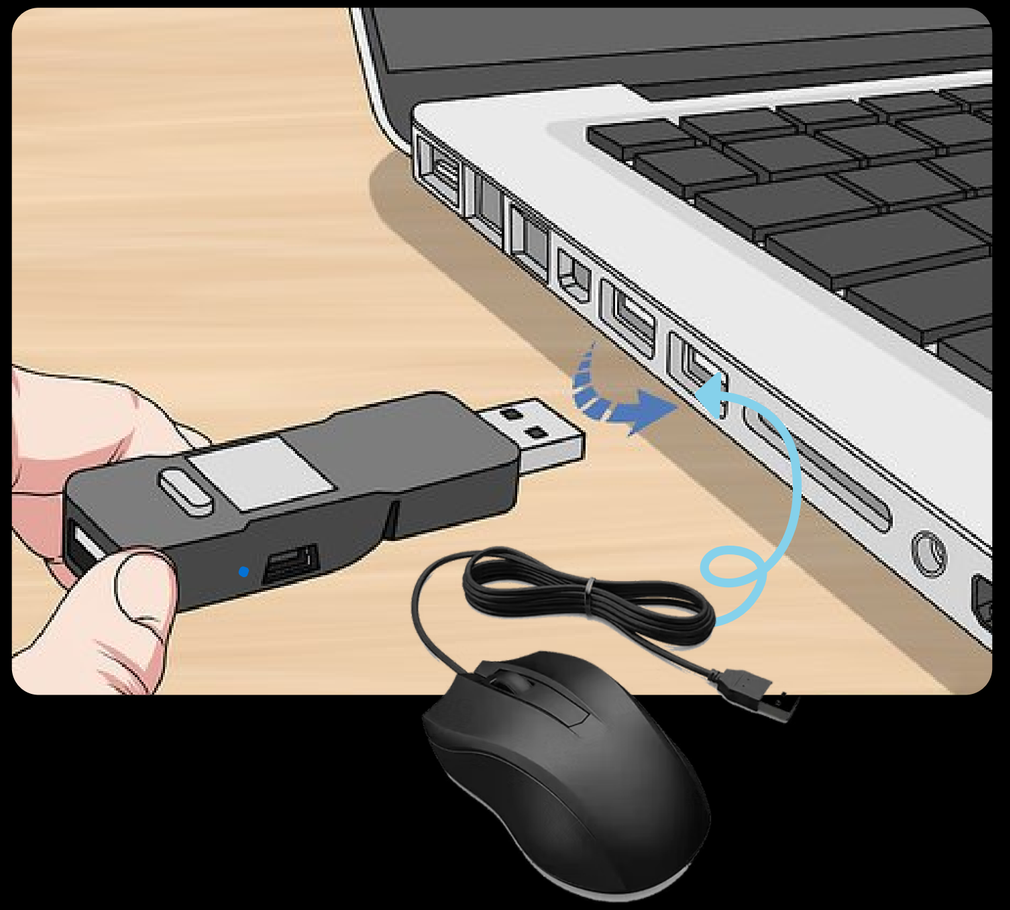


Dependency Injection (DI):

Dependency Injection (DI), c’est quand une classe reçoit les objets dont elle a besoin au lieu de les fabriquer elle-même.




COUPLAGE FAIBLE
GestionOrdinateur
+ main(String[] args)
+ demonstrationFaible()
Ordinateur
- portUSB: PeripheriqueUSB
+ setPortUSB(...)
+ utiliserUSB()
«interface»
PeripheriqueUSB
+ work()
CleUSB
+ work()
Souris
+ work()

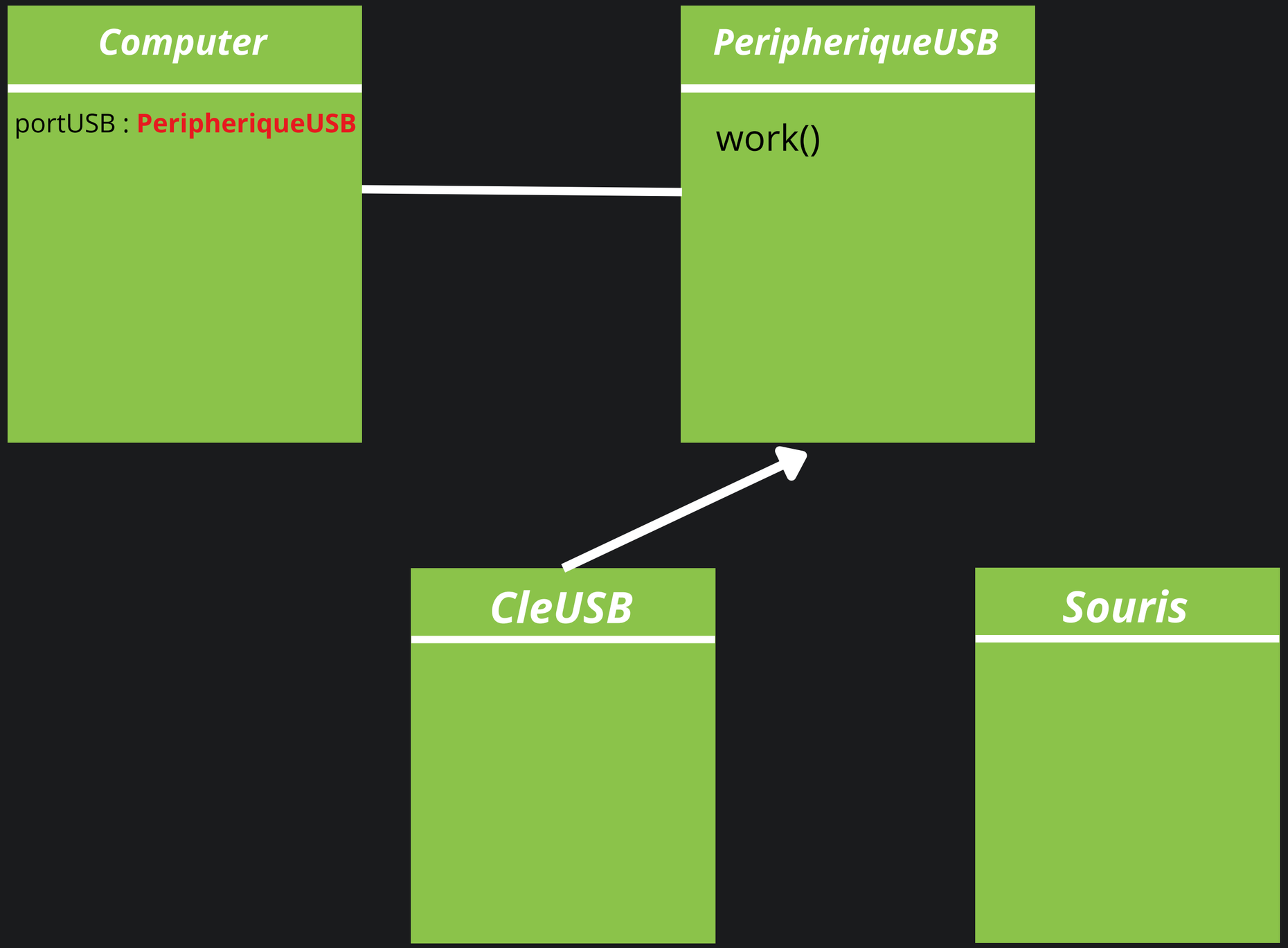


Dependency Injection (DI):

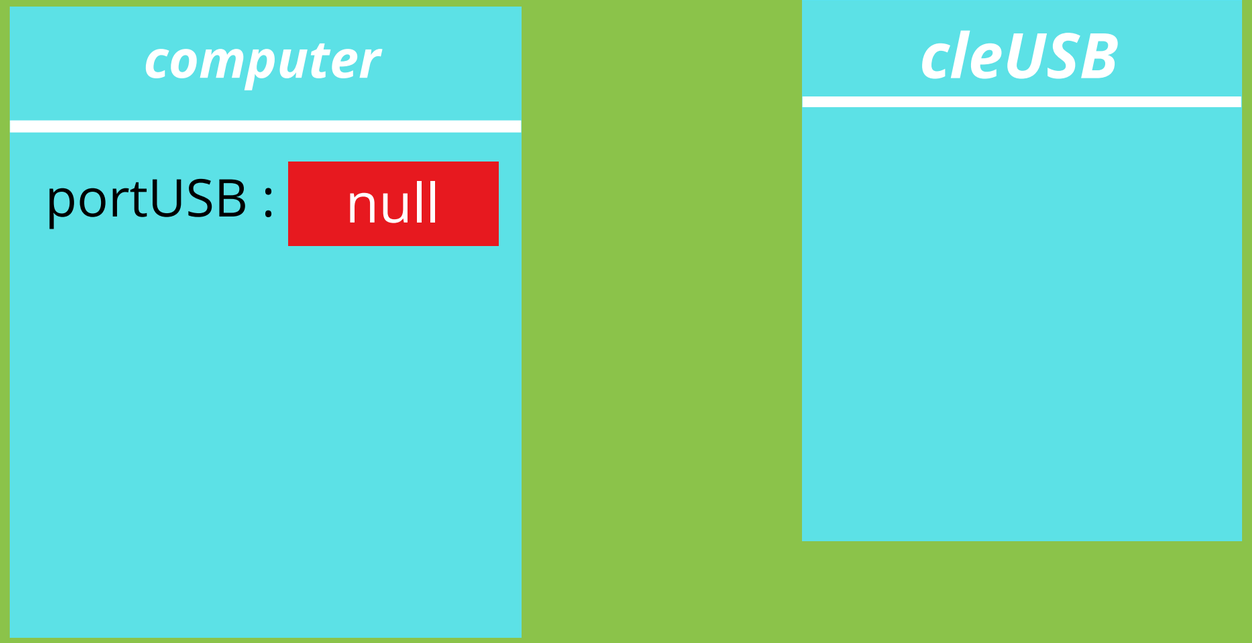
```java
// ❌ MAUVAIS : Création directe (Couplage Fort)
public class Computer {
    private PeripheriqueUSB portUSB = new Souris(); // Dépendance créée ici

    public void demarrer() {
        System.out.println("💻 Ordinateur démarré");
        portUSB.work(); // Couplé à une implémentation spécifique
    }
}
```

Dependency Injection (DI):

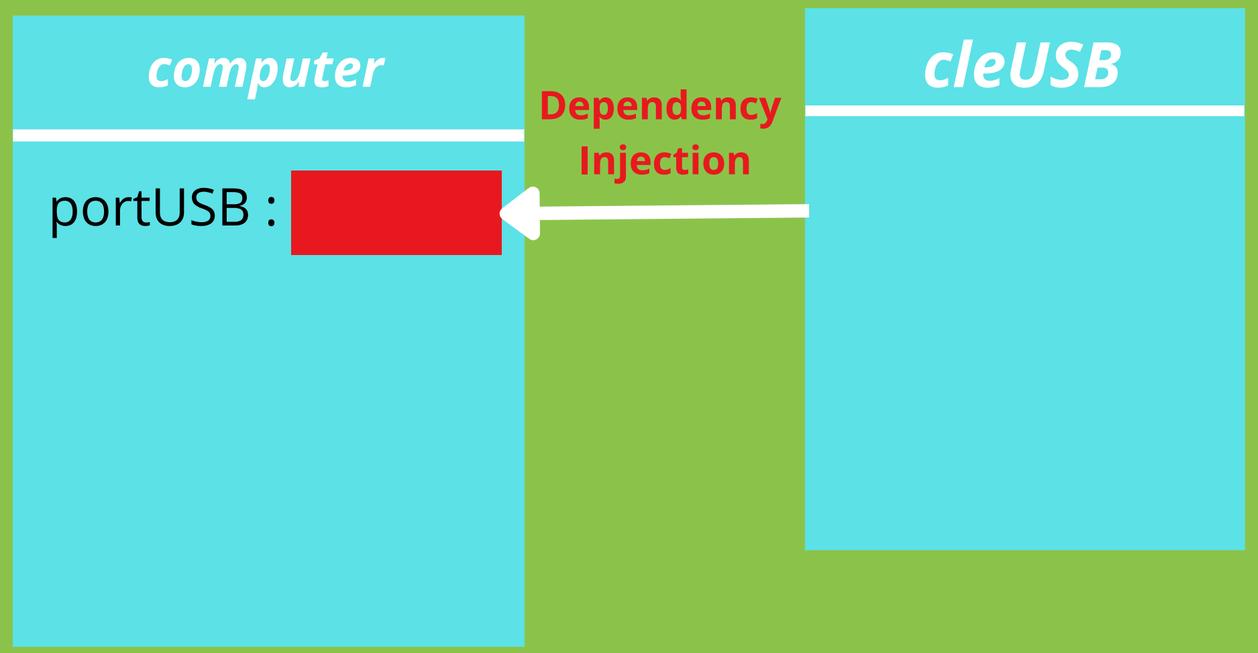
```java
// ✅ BON : Injection de dépendance (Couplage Faible)
@Component
public class Computer {
    private final PeripheriqueUSB portUSB; // Dépendance injectée

    public Computer(PeripheriqueUSB portUSB) {
        this.portUSB = portUSB; // Spring injecte la dépendance
    }

    public void demarrer() {
        System.out.println("💻 Ordinateur démarré");
        portUSB.work(); // Flexible, peut être n'importe quelle implémentation
    }
}
```


Computer
portUSB : PeripheriqueUSB
PeripheriqueUSB
work()
CleUSB
Souris

IOC Container

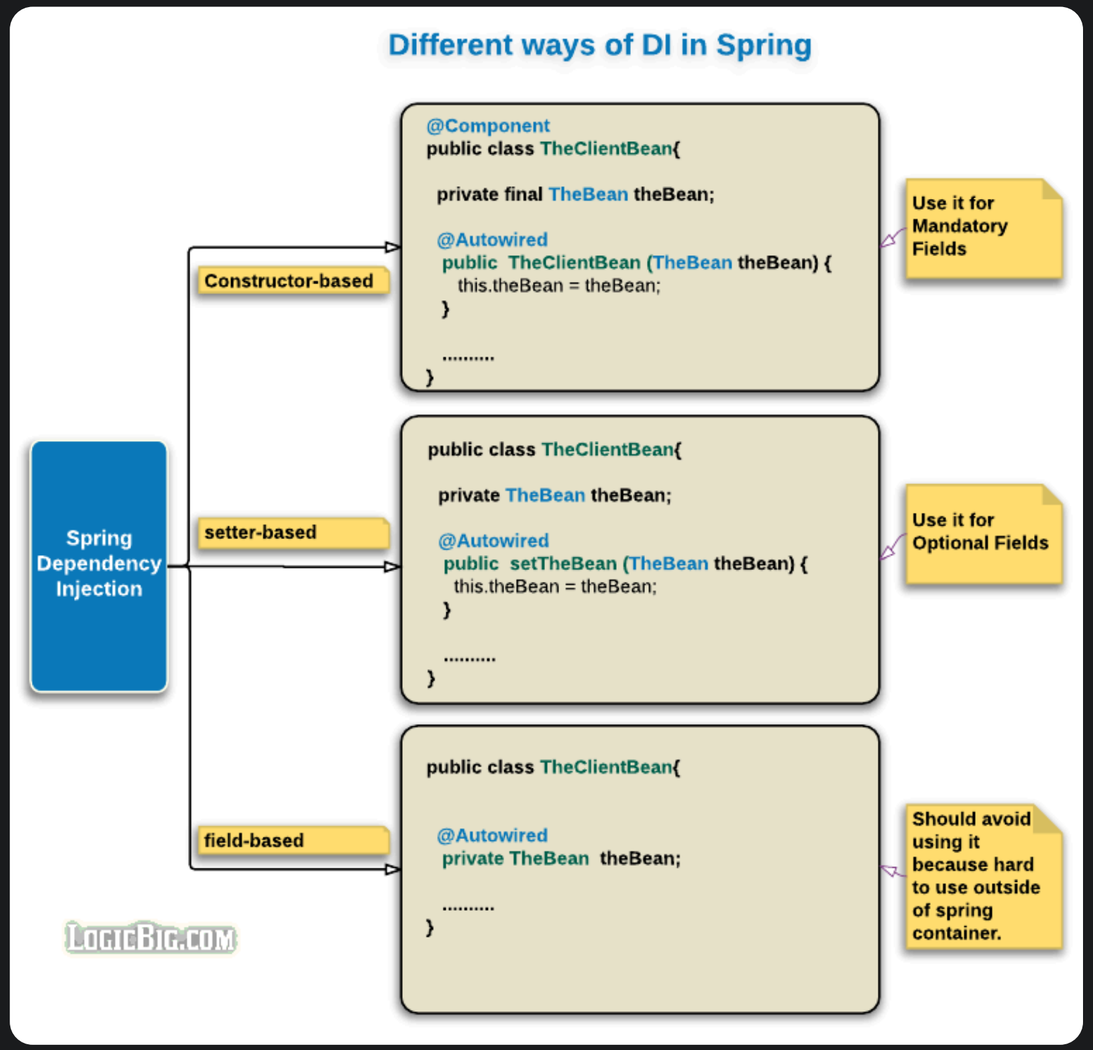


IOC Container



Les 3 Types d'Injection :

- Constructor Injection
- Setter Injection
- Field Injection

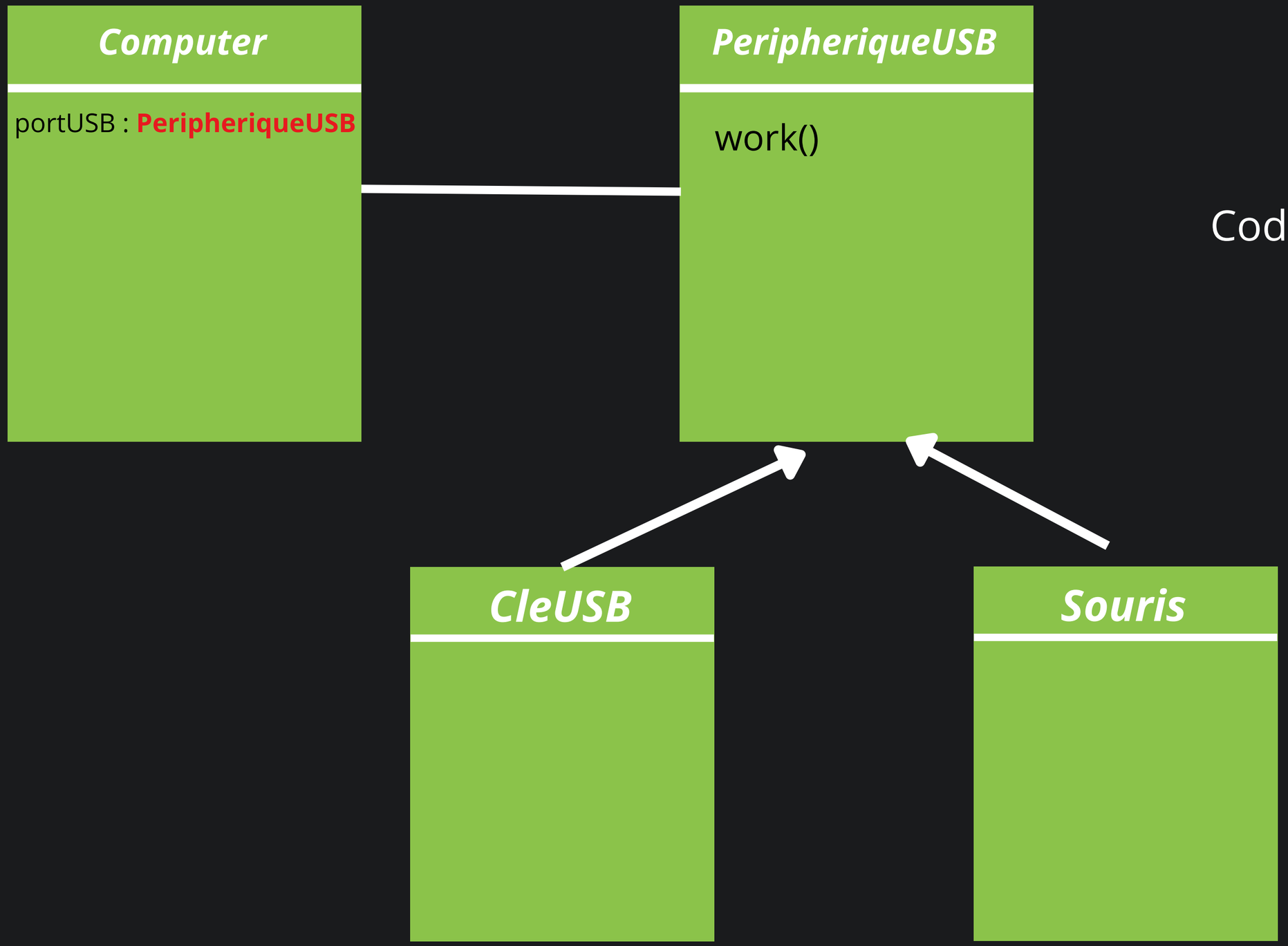


CONSTRUCTOR INJECTION

1- Constructor Injection:

Pourquoi Constructor Injection ?

Immutabilité : Les dépendances ne peuvent pas changer après création
Testabilité : Facilite les tests unitaires
Fail-fast : Erreurs détectées au démarrage
Sécurité : Garantit que l'objet est complètement initialisé


Computer
portUSB : PeripheriqueUSB
PeripheriqueUSB
work()
CleUSB
Souris

Code

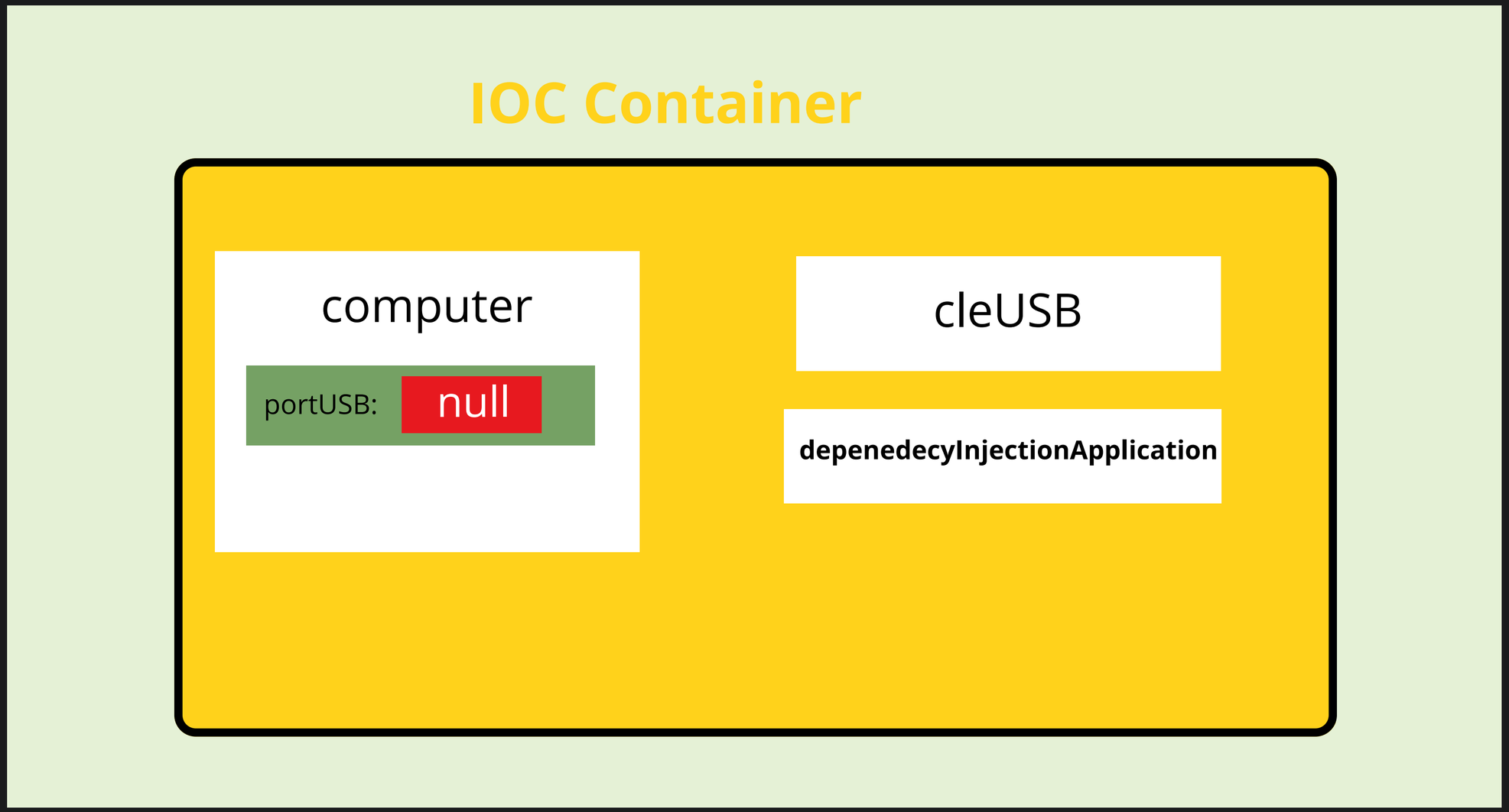
Constructor Injection:

```java
// Interface pour les périphériques USB
public interface PeripheriqueUSB {
    void work();
}
```

```java
// Implémentation Clé USB
@Component
public class CleUSB implements PeripheriqueUSB {
    @Override
    public void work() {
        System.out.println("💾 Clé USB : Transfert de données en cours...");
    }
}
```

```java
// Implémentation Souris
@Component
public class Souris implements PeripheriqueUSB {
    @Override
    public void work() {
        System.out.println("🖱️ Souris : Détection des mouvements et clics...");
    }
}
```

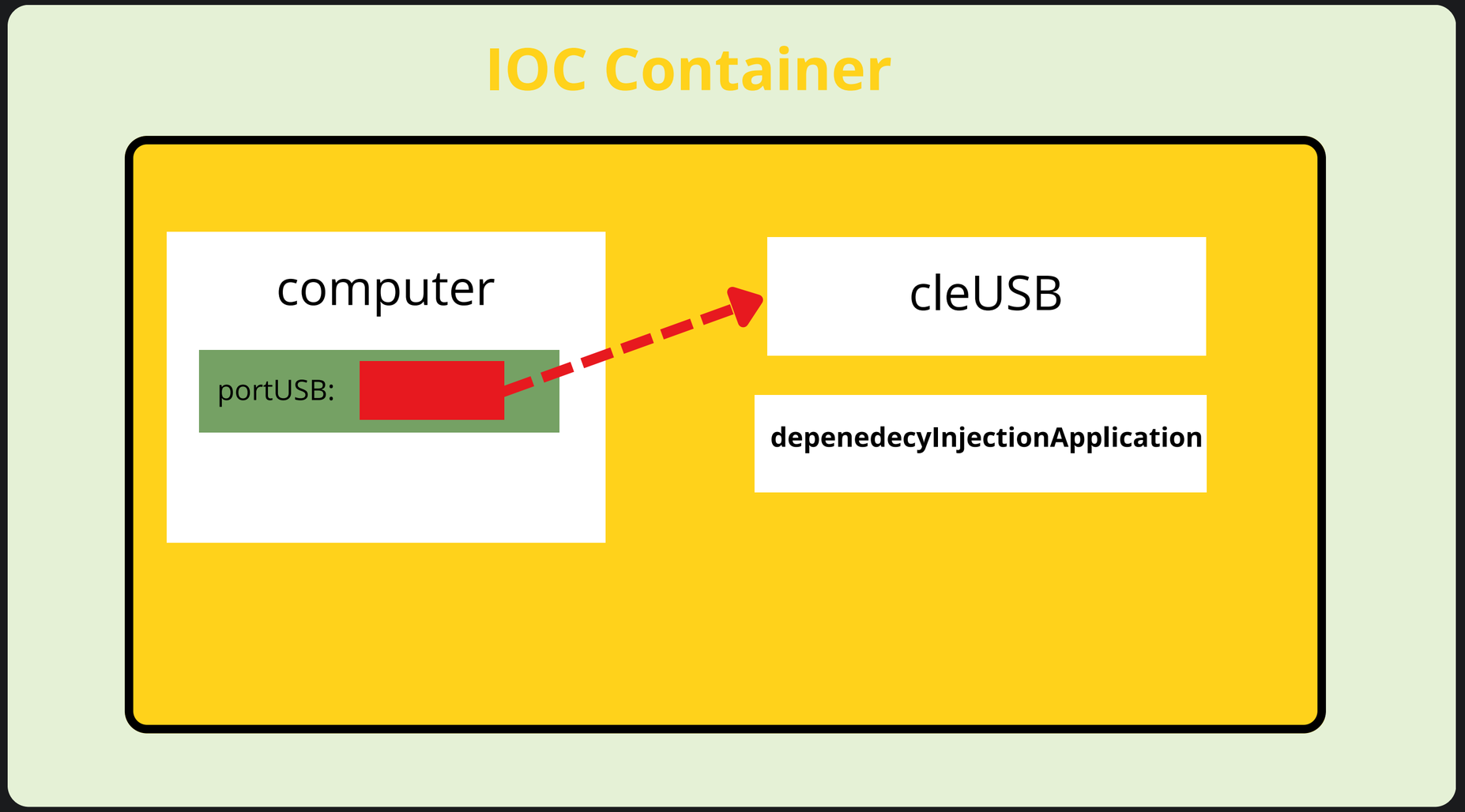
Constructor Injection:



Constructor Injection:

```java
// Classe Computer avec Constructor Injection
@Component
public class Computer {

    private final PeripheriqueUSB portUSB;

    // Constructor Injection
    public Computer(PeripheriqueUSB portUSB) {
        this.portUSB = portUSB;
    }

    public void demarrer() {
        System.out.println("💻 Ordinateur démarré");
        System.out.println("🔌 Périphérique connecté au port USB :");
        portUSB.work();
    }

    public PeripheriqueUSB getPortUSB() {
        return portUSB;
    }
}
```

Constructor Injection:



Constructor Injection avec @Autowired :

❗ Quand @Autowired devient obligatoire :

Si tu as plusieurs constructeurs, Spring ne sait pas lequel utiliser. Dans ce cas, tu dois le guider avec @Autowired :

```java
// Classe Computer avec Constructor Injection
@Component
public class Computer {

    private final PeripheriqueUSB portUSB;

    // Constructor Injection
    public Computer(PeripheriqueUSB portUSB) {
        this.portUSB = portUSB;
    }

    public void demarrer() {
        System.out.println("💻 Ordinateur démarré");
        System.out.println("🔌 Périphérique connecté au port USB :");
        portUSB.work();
    }

    public PeripheriqueUSB getPortUSB() {
        return portUSB;
    }
}
```

Constructor Injection avec @Autowired :

```java
@Component
public class Computer {

    private final PeripheriqueUSB portUSB;

    // ✅ Constructeur avec @Autowired (Spring utilise celui-ci)
    @Autowired
    public Computer(PeripheriqueUSB portUSB) {
        this.portUSB = portUSB;
    }

    // 🔶 Constructeur sans paramètres (Spring ignore celui-ci)
    public Computer() {
        this.portUSB = null;
    }

    public void demarrer() {
        System.out.println("💻 Ordinateur work");
    }
}
```

SETTER INJECTION

2- Setter Injection:

Qu'est-ce que Setter Injection ?

L'injection par setter permet d'injecter les dépendances après la création de l'objet en utilisant des méthodes setter. Spring appelle automatiquement ces méthodes pour injecter les dépendances.

Caractéristiques de Setter Injection :

Optionnalité : Les dépendances peuvent être optionnelles
Mutabilité : Les dépendances peuvent être changées après création
Cycle de vie : Injection après la construction de l'objet

2- Setter Injection:

```java
@Component
public class Computer {
    private PeripheriqueUSB portUSB;

    // Setter avec @Autowired
    @Autowired
    public void setPortUSB(PeripheriqueUSB portUSB) {
        this.portUSB = portUSB;
    }

    public void work() {
        if (portUSB != null) {
            portUSB.work();
        }
    }
}
```

```java
@Component
public class CleUSB implements PeripheriqueUSB {
    @Override
    public void work() {
        System.out.println("CleUSB est en cours d'utilisation");
    }
}
```

2- Setter Injection:

Diagramme - Setter Injection :

1. Création de l'objet

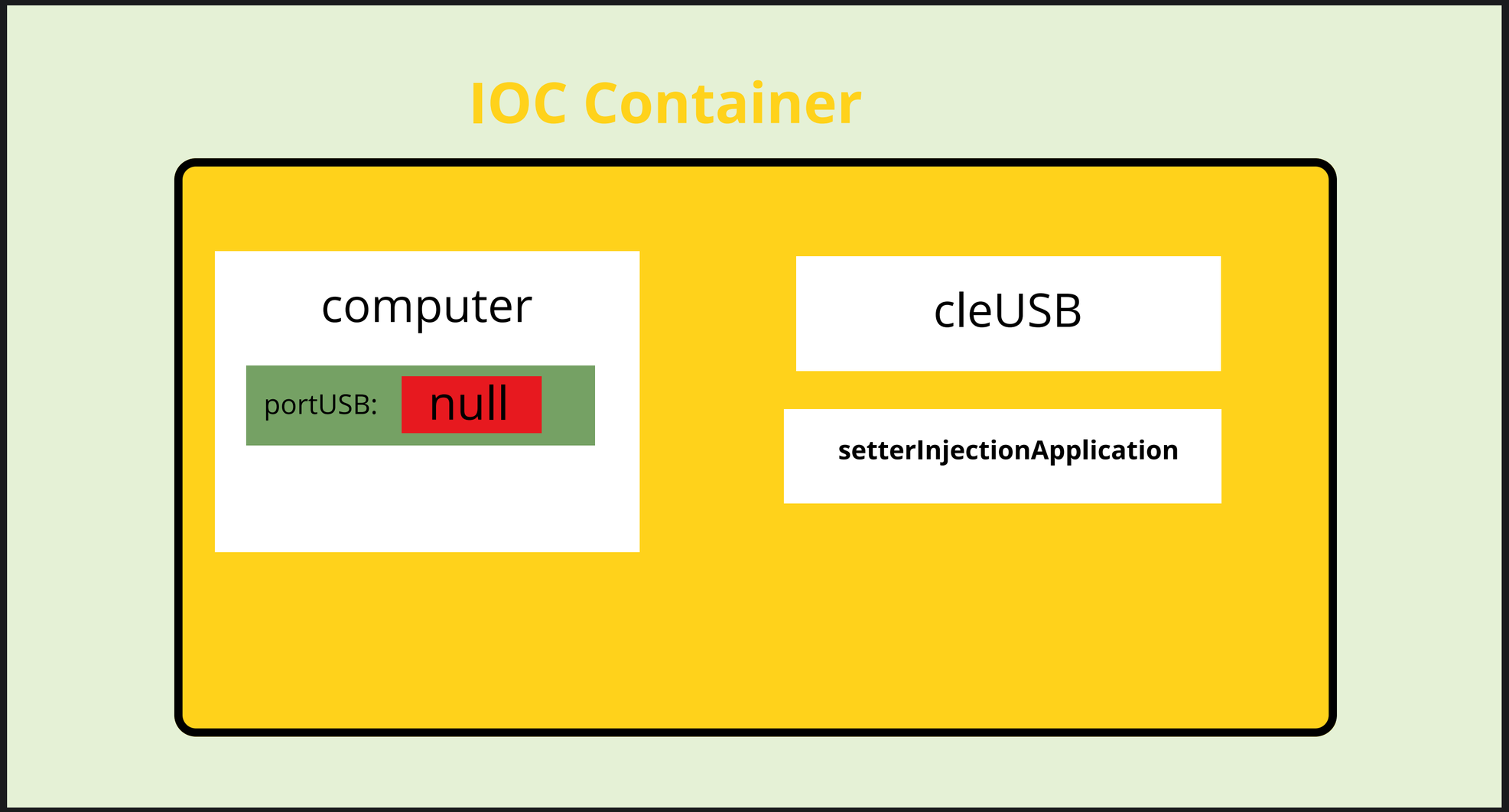
- IOC Container
- computer (créé)
- portUSB : null

2. Injection via setter

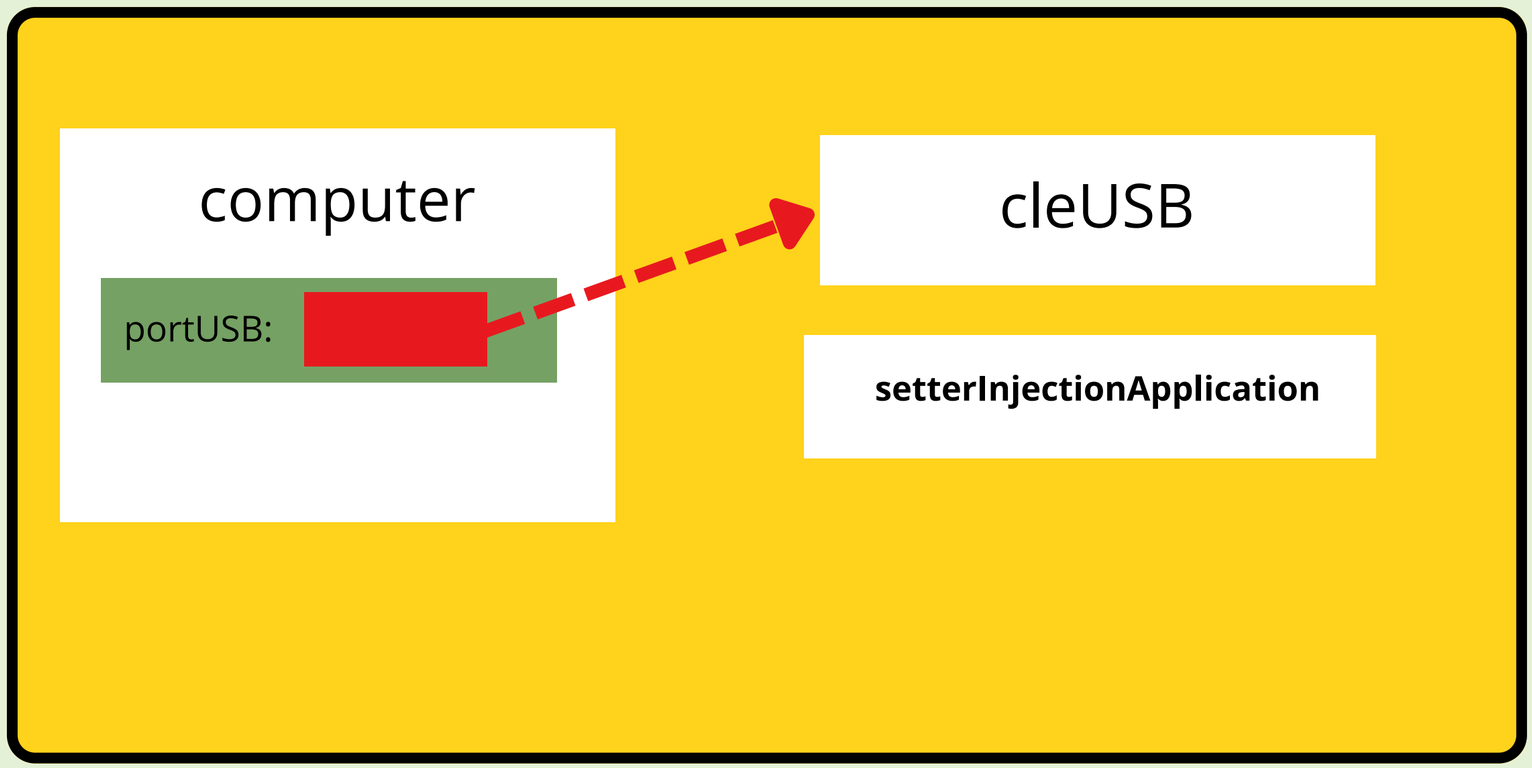
- IOC Container
- computer
- portUSB : cleUSB (injecté via setPortUSB())

```java
@Component
public class Computer {
    private PeripheriqueUSB portUSB;

    // Setter avec @Autowired
    @Autowired
    public void setPortUSB(PeripheriqueUSB portUSB) {
        this.portUSB = portUSB;
    }

    public void work() {
        if (portUSB != null) {
            portUSB.work();
        }
    }
}
```

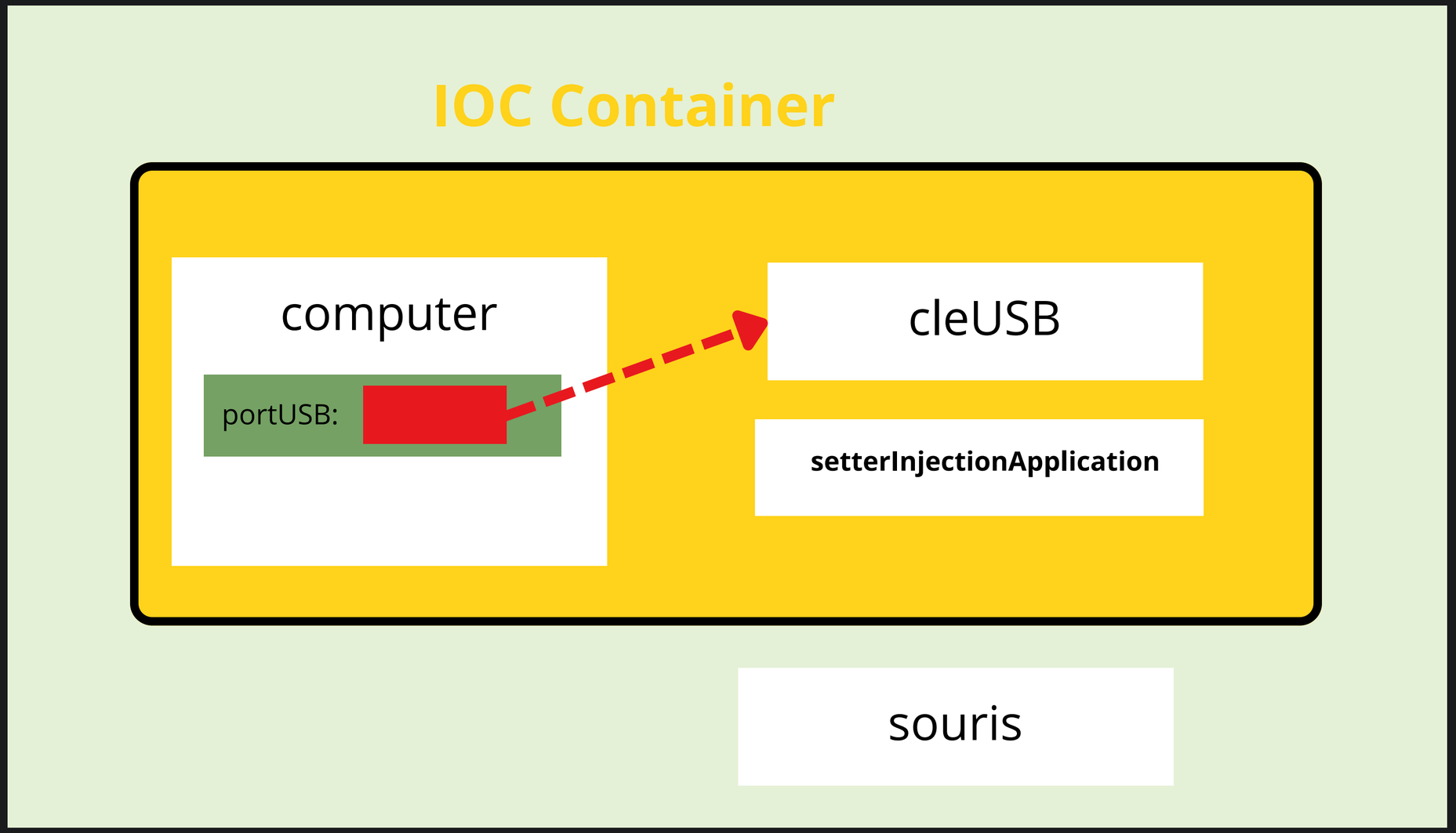
2- Setter Injection:



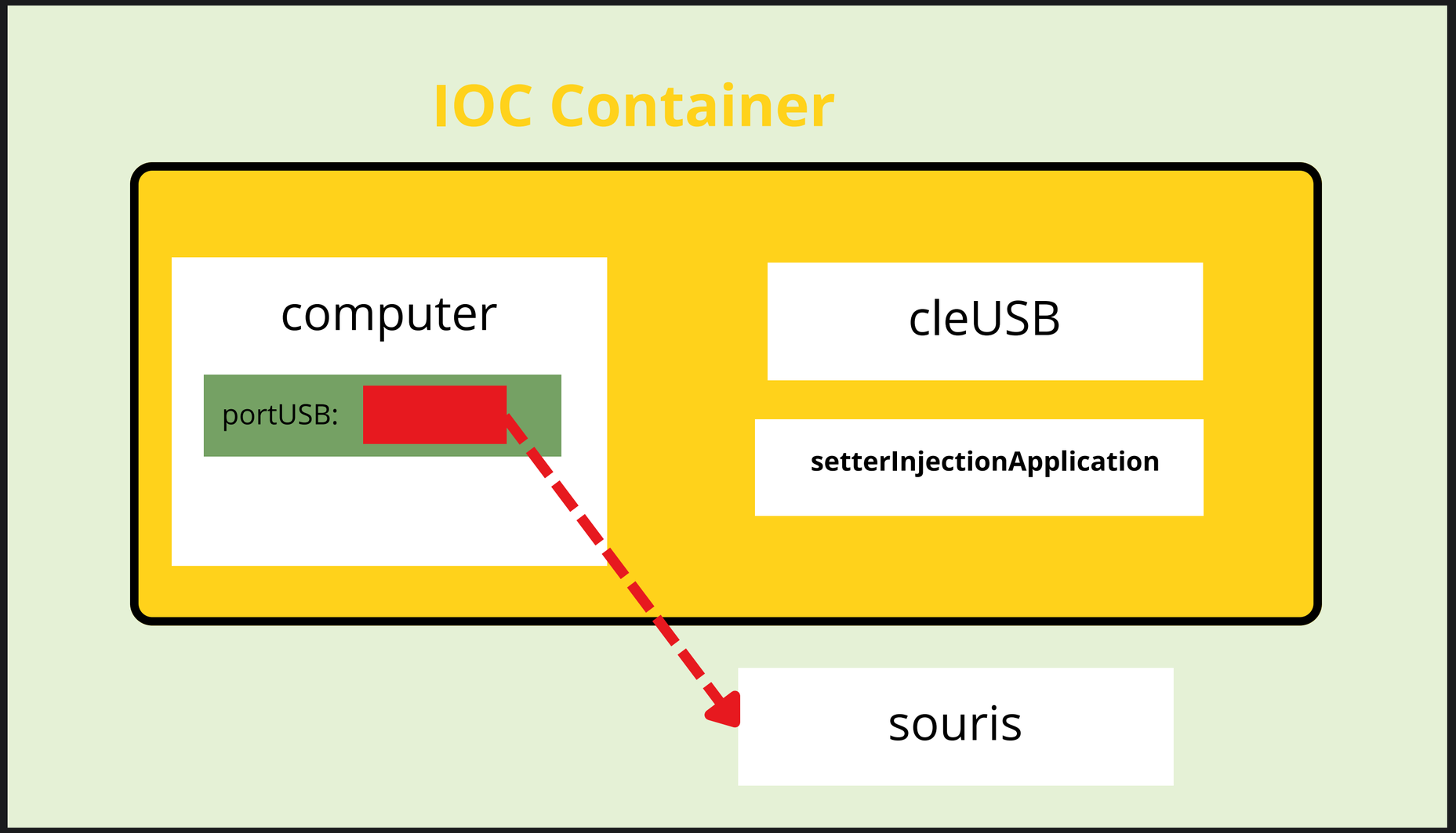
2- Setter Injection:



2- Setter Injection:



2- Setter Injection:



2- Setter Injection:

Inconvénients de Setter Injection :

❌ **Mutabilité** : Les dépendances peuvent être modifiées
❌ **Tests** : Plus difficile à tester (état partiel possible)

Quand utiliser Setter Injection ?

- Quand les dépendances sont optionnelles
- Quand vous avez besoin de reconfigurer l'objet

2- Setter Injection:

Comparaison Constructor vs Setter :

Aspect	Constructor Injection	Setter Injection
Immutabilité	✅ Oui	❌ Non
Dépendances optionnelles	❌ Difficile	✅ Facile
Fail-fast	✅ Oui	❌ Non
Testabilité	✅ Excellente	⚠️ Moyenne
Lisibilité	✅ Claire	⚠️ Moyenne

2- Setter Injection:

Recommandation Spring :

Spring recommande **Constructor Injection** comme pratique par défaut, et Setter Injection uniquement pour les cas spécifiques où la flexibilité est nécessaire.

FIELD INJECTION

3- Field Injection:

Qu'est-ce que Field Injection ?

Field Injection est une méthode d'injection de dépendances où Spring injecte directement les dépendances dans les champs (attributs) de la classe en utilisant l'annotation @Autowired.

Caractéristiques de Field Injection :

- **Simplicité : Code plus concis, moins de boilerplate**
- **Injection directe : Pas besoin de constructeurs ou sette**

3- Field Injection:

```java
@Component
public class Computer {

    @Autowired
    private PeripheriqueUSB portUSB;

    public void work() {
        System.out.println("Computer working with: ");
        portUSB.connect();
    }
}
```

```java
@Component
public class CleUSB implements PeripheriqueUSB {
    @Override
    public void connect() {
        System.out.println("CleUSB connected!");
    }
}
```

```java
@SpringBootApplication
public class FieldInjectionApplication {
    public static void main(String[] args) {
        ApplicationContext context = SpringApplication.run(FieldInjectionApplication.class
        Computer computer = context.getBean(Computer.class);
        computer.work();
    }
}
```

3- Field Injection:

Comment Spring injecte-t-il ces champs privés ? — **Via réflexion**

- **Java Reflection (Réflexion)** est une API puissante qui permet à un programme Java d'examiner et manipuler dynamiquement des classes, objets, méthodes et champs à l'exécution, même s'ils sont privés.
- En Java, normalement, **tu ne peux pas accéder directement aux champs privés** d'une classe hors de cette classe.
- **Spring utilise la réflexion pour "forcer" l'accès à ce champ privé** et y injecter l'instance de la dépendance.

java.lang.reflect

3- Field Injection:

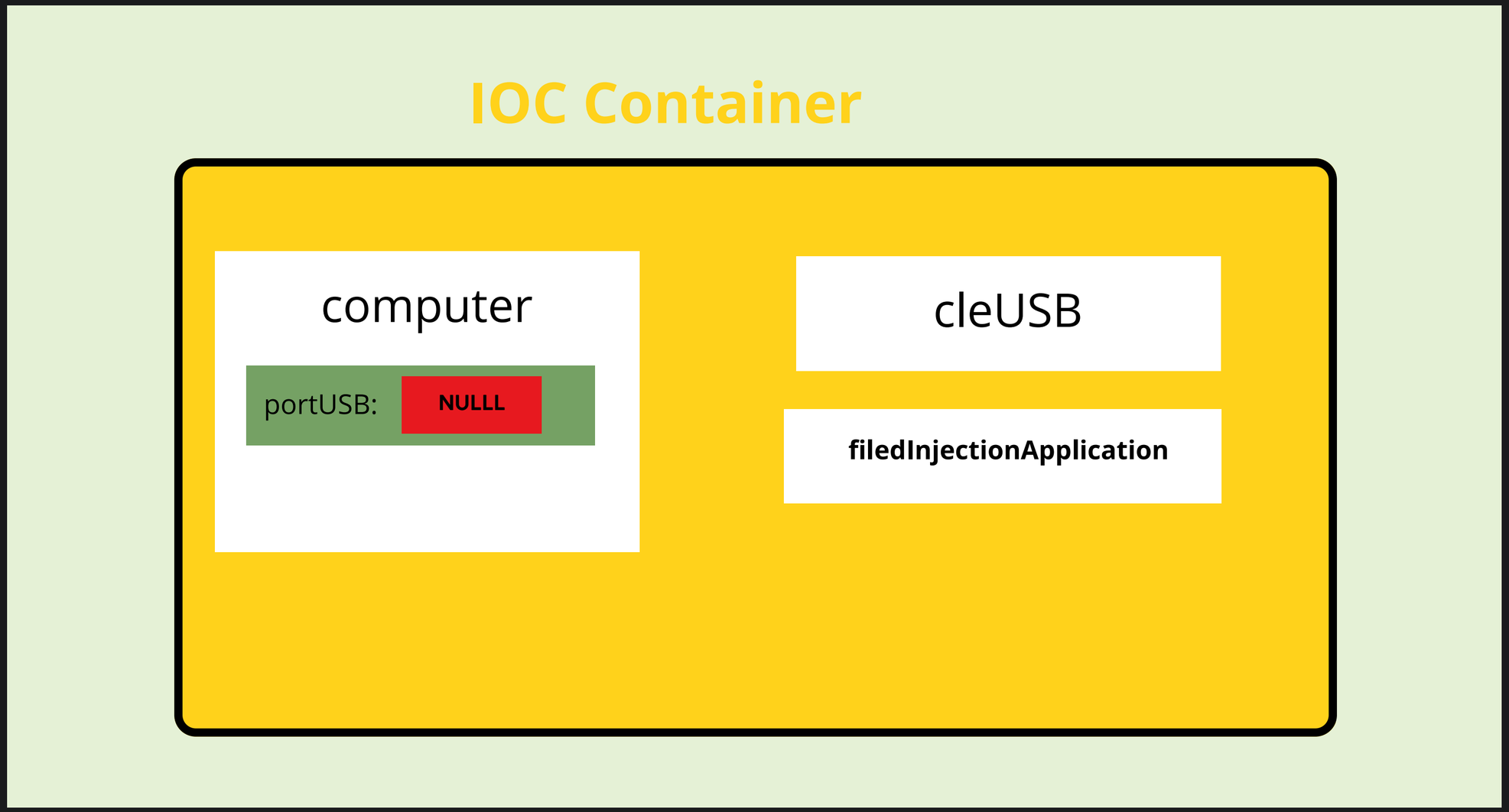
```java
@Component
public class Computer {

    @Autowired
    private PeripheriqueUSB portUSB;

    public Computer() {
        System.out.println("PC Worked!");
    }

    public void utliserPeripheriqueUSB() {
        portUSB.work();
    }
}
```

```java
@Component
public class CleUSB implements PeripheriqueUSB{

    @Override
    public void work() {
        System.out.println("Cle USB est utiliser!");
    }
}
```

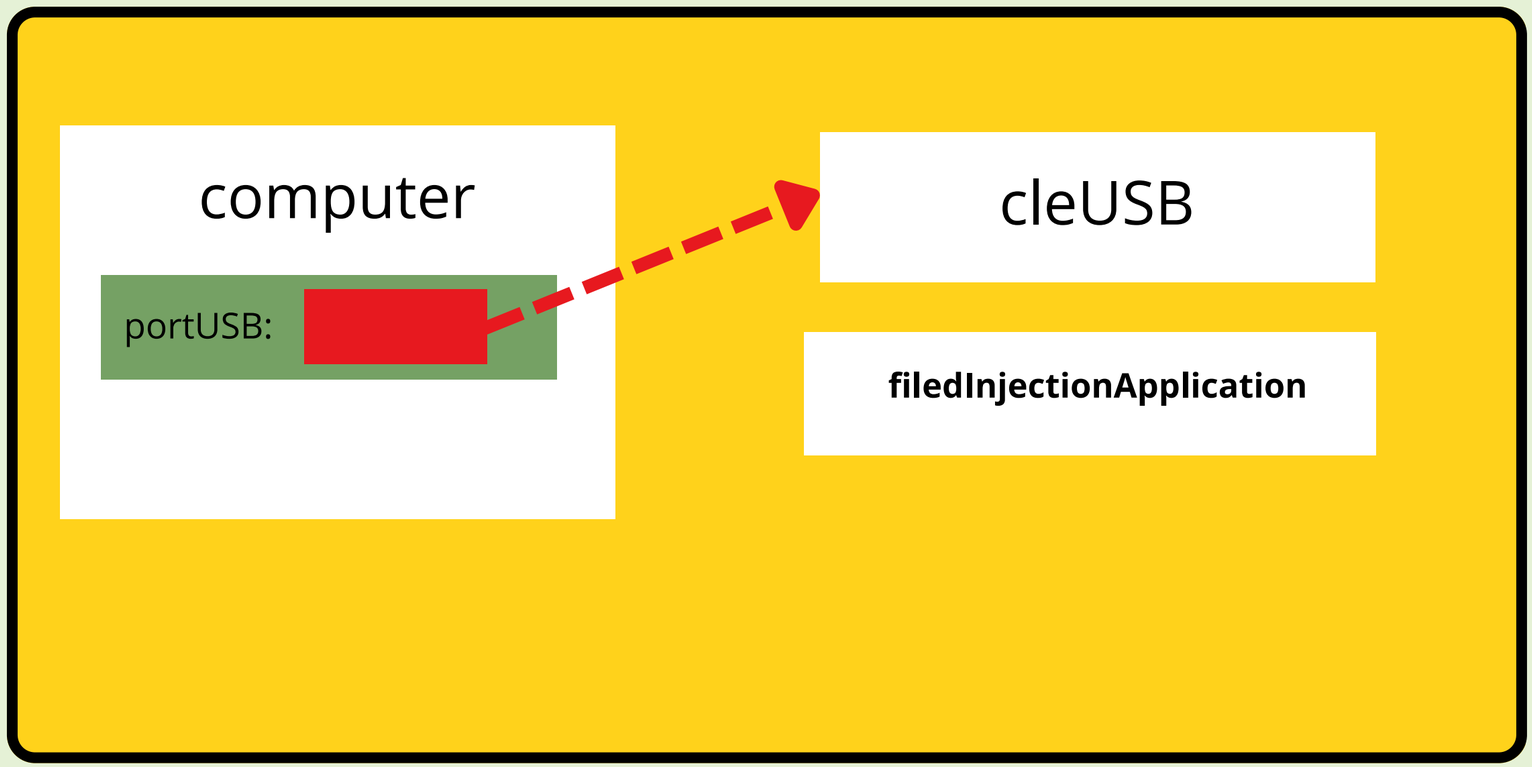
3- Field Injection:

```java
import java.lang.reflect.Field;

@SpringBootApplication
public class FiledInjectionApplication {

    public static void main(String[] args) throws NoSuchFieldException, IllegalAccessException {
        ApplicationContext ioc_container = SpringApplication.run(FiledInjectionApplication.class, args);

        Computer computer = ioc_container.getBean(Computer.class);
        PeripheriqueUSB cleUSB = ioc_container.getBean(CleUSB.class);

        Field field = Computer.class.getDeclaredField("portUSB");;
        field.setAccessible(true);
        field.set(computer, cleUSB);  // Injection effective

        computer.utliserPeripheriqueUSB();

    }

}
```

3- Field Injection:



3- Field Injection:



3- Field Injection:

Comparaison des 3 Types d'Injection

Critère	Constructor	Setter	Field
Immutabilité	✅ Oui	❌ Non	❌ Non
Testabilité	✅ Excellente	⚠️ Moyenne	❌ Difficile
Optionalité	❌ Non	✅ Oui	❌ Non
Sécurité	✅ Fail-fast	⚠️ Runtime	⚠️ Runtime
Simplicité code	⚠️ Verbeux	⚠️ Verbeux	✅ Concis
Découplage Spring	✅ Indépendant	✅ Indépendant	❌ Couplé

@PRIMARY ET @QUALIFIER

@Primary et @Qualifier:

```java
public interface PeripheriqueUSB {
    void work();
}
```

```java
@Component
public class CleUSB implements PeripheriqueUSB {
    @Override
    public void work() {
        System.out.println("💾 CleUSB fonctionne...");
    }
}
```

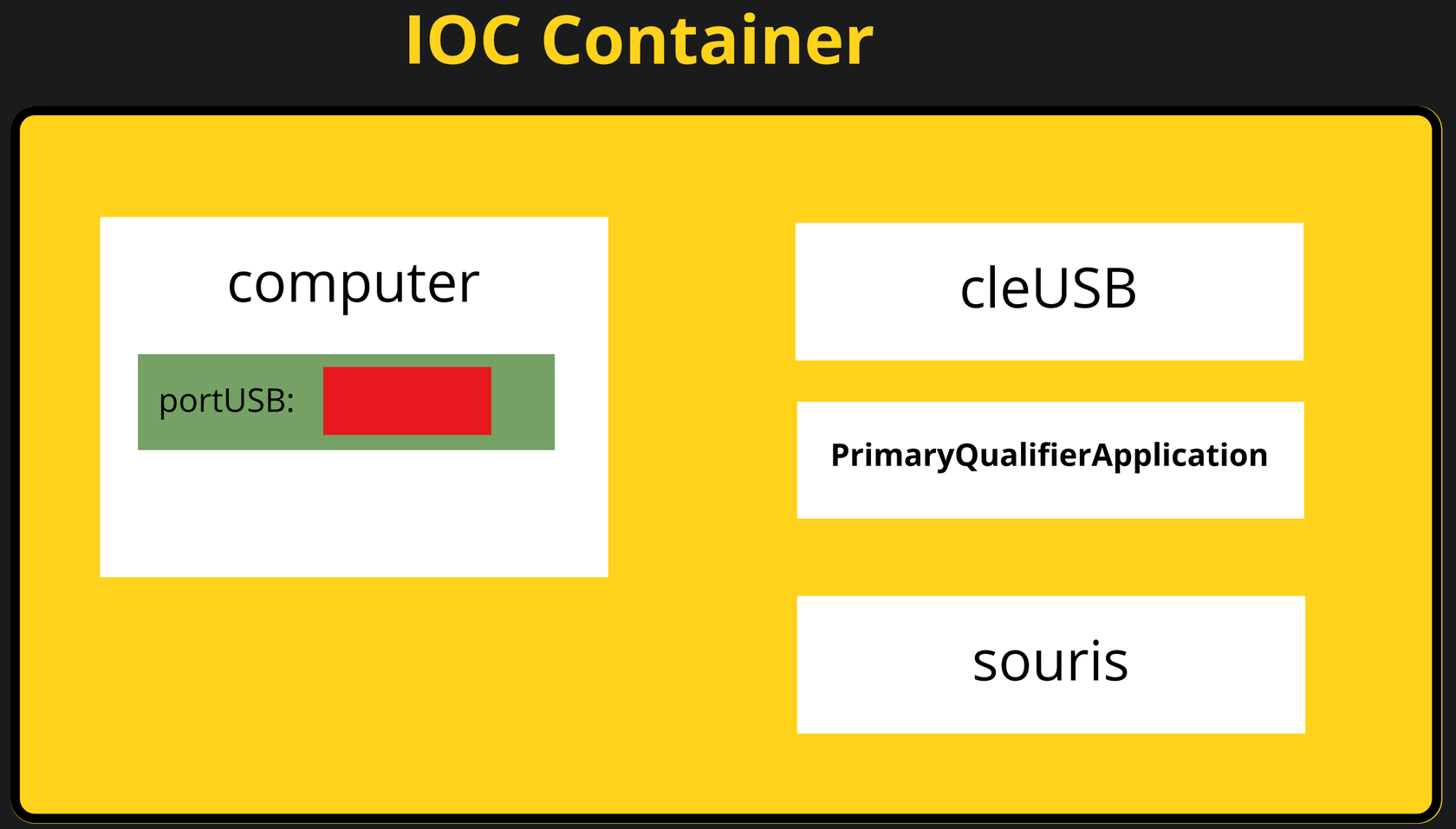
```java
@Component
public class Souris implements PeripheriqueUSB {
    @Override
    public void work() {
        System.out.println("🖱️ Souris fonctionne...");
    }
}
```

@Qualifier et @Primary :

```java
@Component
public class Computer {

    @Autowired
    private PeripheriqueUSB portUSB; // ❌ ERREUR !

    public void demarrer() {
        System.out.println("💻 Computer démarre...");
        portUSB.work(); // Quel périphérique ???
    }
}
```

Erreur Spring :
NoUniqueBeanDefinitionException:
No qualifying bean of type 'PeripheriqueUSB' available: expected single matching bean but found 3: cleUSB, souris

@Qualifier et @Primary :



SOLUTION 1 : @PRIMARY

@Qualifier et @Primary :

Solution 1 : @Primary

@Primary dit à Spring : "Si tu ne sais pas lequel choisir, prends celui-ci par défaut"

```java
@Component
public class CleUSB implements PeripheriqueUSB {
    @Override
    public void work() {
        System.out.println("💾 CleUSB fonctionne...");
    }
}
```

```java
@Component
@Primary  // ✅ Périphérique par défaut
public class Souris implements PeripheriqueUSB {
    @Override
    public void work() {
        System.out.println("🖱️ Souris fonctionne (par défaut)...");
    }
}
```

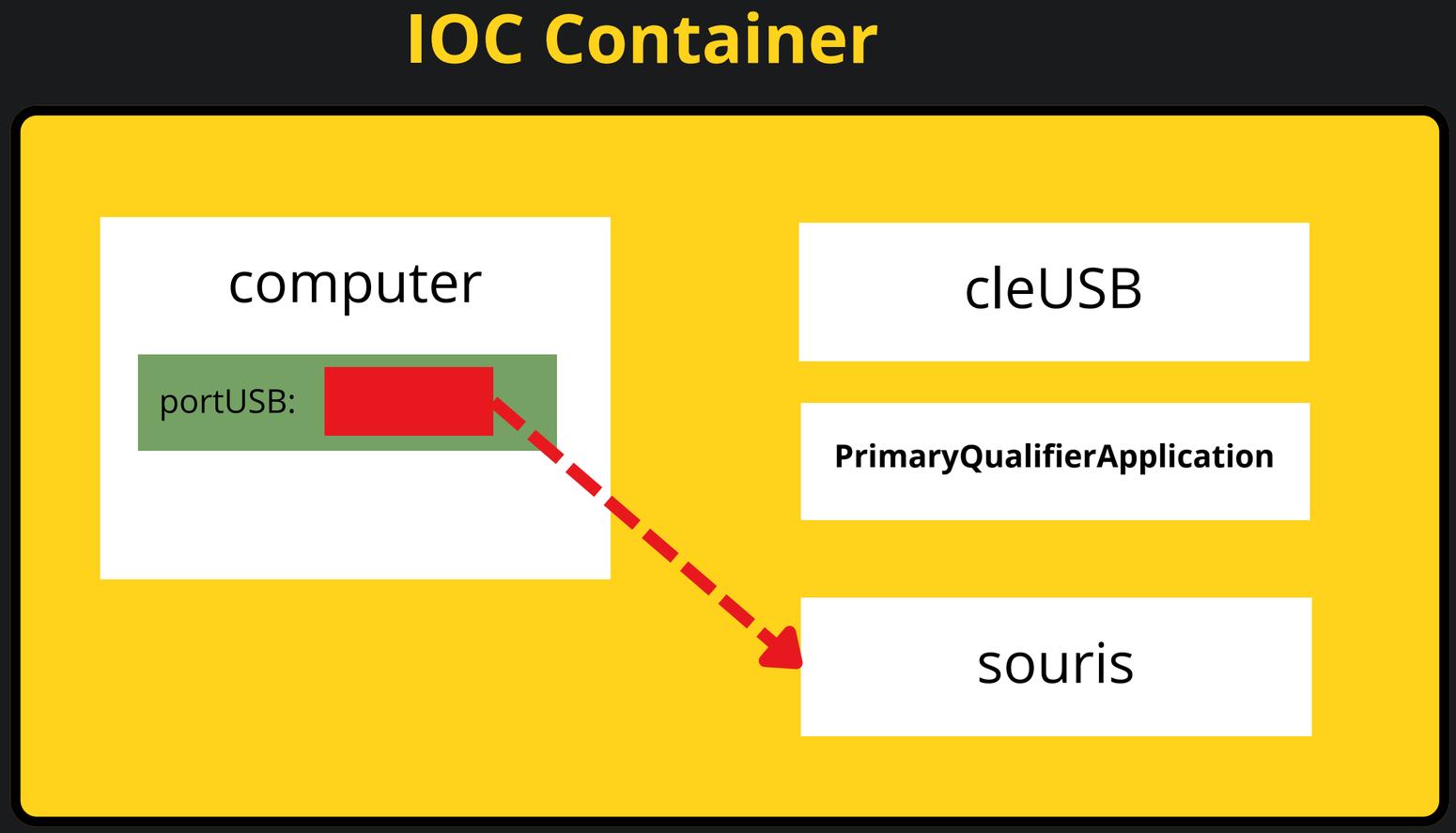
@Qualifier et @Primary :

Solution 1 : @Primary

```
@Component
public class Computer {

    @Autowired
    private PeripheriqueUSB portUSB; // ✅ Injectera Souris (@Primary)

    public void demarrer() {
        System.out.println("💻 Computer démarre...");
        portUSB.work(); // 🖱️ Souris fonctionne (par défaut)...
    }
}
```

@Qualifier et @Primary :

Solution 1 : @Primary



SOLUTION 2 : @QUALIFIER

@Qualifier et @Primary :

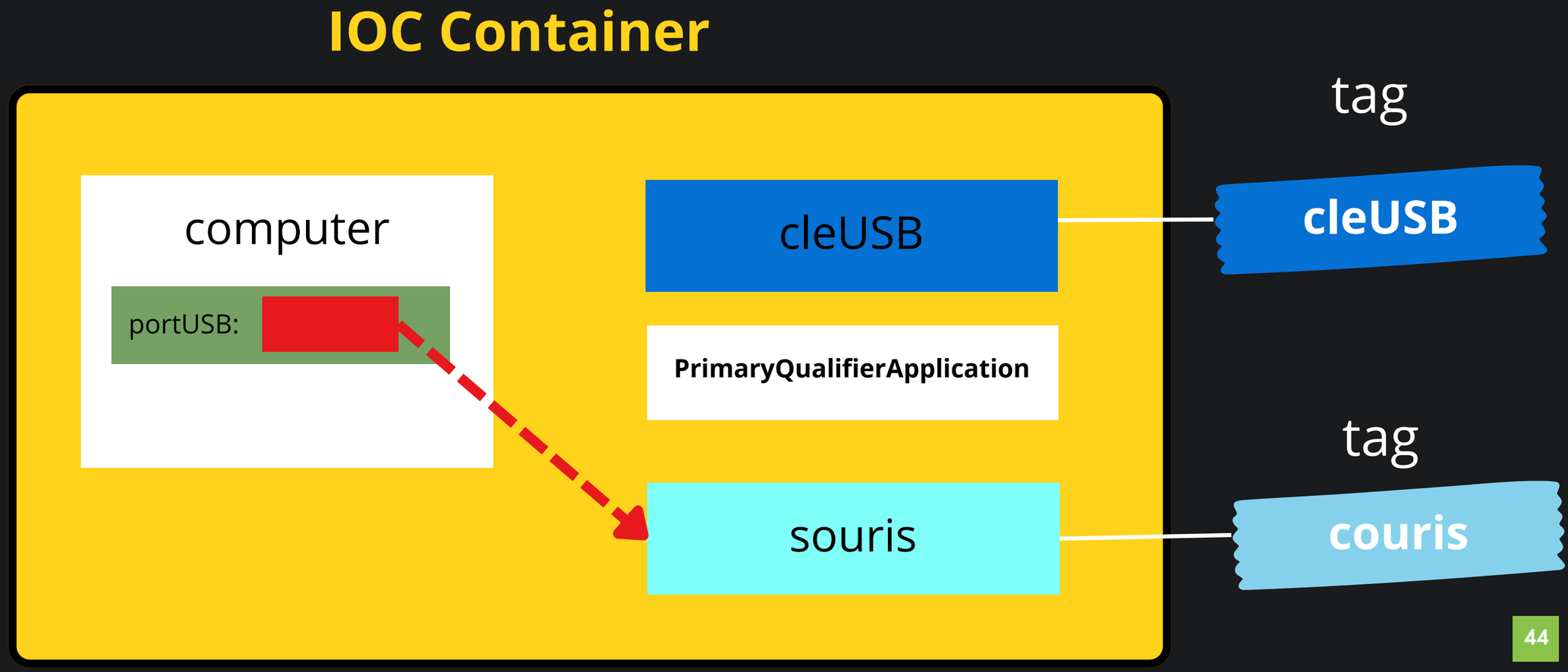
Solution 2 : @Qualifier

@Qualifier permet de choisir exactement quel périphérique vous voulez.

```
@Component
public class Computer {

    @Autowired
    @Qualifier("cleUSB")  // ✅ Choisit spécifiquement CleUSB
    private PeripheriqueUSB portUSB;

    public void demarrer() {
        System.out.println("💻 Computer démarre...");
        portUSB.work(); // 💾 CleUSB fonctionne...
    }
}
```

@Qualifier et @Primary :

Solution 2 : @Qualifier



@Qualifier et @Primary :

Solution 2 : @Qualifier

```java
@Component
@Qualifier("souris")
public class Souris implements PeripheriqueUSB {
    @Override
    public void work() {
        System.out.println("🖱️ Souris fonctionne...");
    }
}
```

```java
@Component
@Qualifier("stockage")
public class CleUSB implements PeripheriqueUSB {
    @Override
    public void work() {
        System.out.println("💾 CleUSB fonctionne...");
    }
}
```

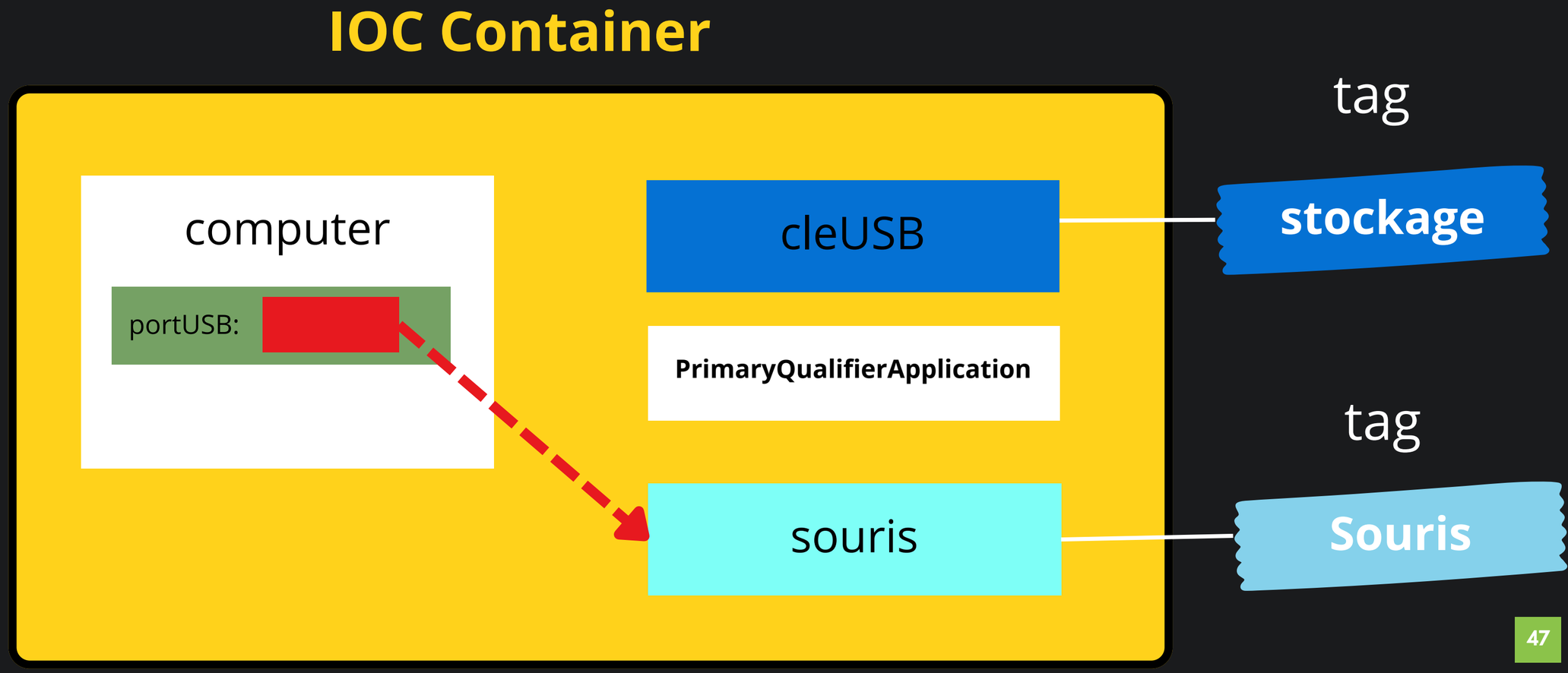
@Qualifier et @Primary :

Solution 2 : @Qualifier

```java
@Component
public class Computer {

    @Autowired
    @Qualifier("stockage")
    private PeripheriqueUSB portUSB;

    public void demarrer() {
        System.out.println("💻 Computer démarre ....");
        portUSB.work(); // 💾 CleUSB fonctionne...
    }
}
```

@Qualifier et @Primary :

Solution 2 : @Qualifier



MERCI !

@med Mohammed Ezzaim