IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Contrats et Interfaces

M.T. Segarra & F. Dagnat &
A. Beugnard & J.C. Royer
DLR – MAPD – C4
2023

Objectif de ce cours

1. Notion de contrat
2. Interface le retour
3. Classe abstraite le retour
4. Classe interne
5. Assemblons tout cela

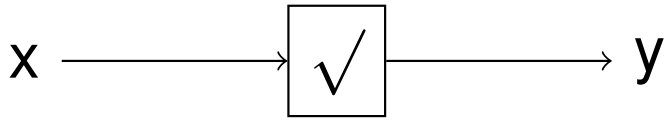
Contribution aux compétences

1. Spécification
2. Conception
3. Développement

Je vous invite à prendre des notes...

- Expliciter les interactions entre objets sous la forme d'un contrat
  - les obligations de l'appellant
  - les garanties sur le résultat
- Définition d'un contrat à l'aide de
  - préconditions et postconditions sur les méthodes
  - invariants de la classe
- Vérification des contrats à l'exécution (si possible avant l'exécution)

Comment spécifier cette fonction ?



La signature `float sqrt(float)` suffit-elle ?

- C'est le contrat syntaxique/statique ; il garantit la compilation. Il ne garantit pas l'effet du calcul.

Pour le sens (la sémantique) on utilise un couple précondition/postcondition :

Précondition : $x \geq 0$

Postcondition : $y \times y = x \wedge y \geq 0$

C'est Tony Hoare qui a introduit les notions de pré- et postcondition.

Programmation contractuelle : chaque fonction définit une précondition et une postcondition.

	obligation	avantage
client	PRE	POST
fonction	POST	PRE

Les conditions peuvent être du code qui s'exécute en phase de test et qu'on n'évalue pas en phase de production (utilisation).

Annotations JML (*The Java Modeling Language*) :

```
1  public class CompteBancaireContrats {
2      private int soldeMin = 0;
3      private int solde;
4      //@invariant solde >= soldeMin;
5      //@ensures solde == soldeMin;
6      public CompteBancaireContrats() { ... }
7      //@requires montant > 0;
8      //@ensures solde == \old(solde) + montant;
9      public void crediter(int montant) { ... }
10     //@requires montant > 0;
11     //@ensures solde == \old(solde) - montant;
12     public void debiter(int montant) { ... }
13     //@ensures \return == solde;
14     public int getSolde() { ... }
15 }
```

- Code de meilleure qualité
  - Écrire une (partie de la) spécification avant d'écrire le code = moins d'erreurs a priori
  - Si spécification exécutable alors détection de non conformité à l'exécution
- Une autre manière de documenter un logiciel
  - Écriture de la spécification en langage proche d'un langage de programmation
  - Documentation « générale »
  - Outils de calcul pour vérifier (JML, spec#, . . .)

- Code de meilleure qualité
  - Écrire une (partie de la) spécification avant d'écrire le code = moins d'erreurs a priori
  - Si spécification exécutable alors détection de non conformité à l'exécution
- Une autre manière de documenter un logiciel
  - Écriture de la spécification en langage proche d'un langage de programmation
  - Documentation « générale »
  - Outils de calcul pour vérifier (JML, spec#, . . .)
- Le test a les mêmes objectifs mais
  - le contrat est plus général, il couvre tous les cas alors que le test couvre un (ou des) cas
  - le contrat est donc souvent plus difficile à définir
  - Remarque : la postcondition est un oracle de test

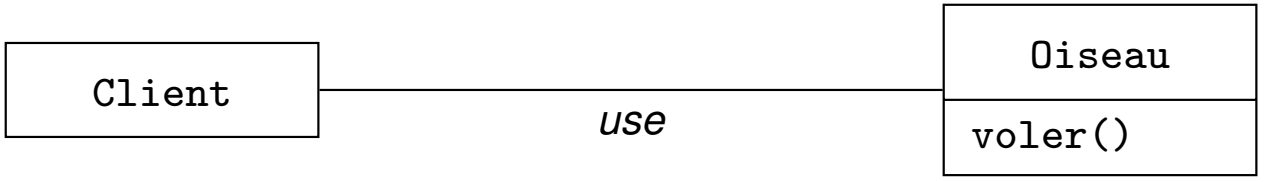
- Pas de construction native dans le langage
- Utilisation des assertions
  - Assertions mélangées avec le code
  - Pas de génération de documentation
  - Option *-ea* de la VM pour activer les assertions
- Lire la documentation `http://download.oracle.com/javase/8/docs/technotes/guides/language/assert.html`

```
1  public class CompteBancaireAssertions {
2      public int minSolde = 0;
3      private int solde;
4      public CompteBancaireAssertions() {
5        ...
6        assert(solde == minSolde);
7      }
8      public void crediter(int montant) throws ExceptionMontantIllegal {
9        if (montant <= 0)
10         throw new ExceptionMontantIllegal();
11       int oldSolde = solde;
12       solde = solde + montant;
13       assert(solde == (oldSolde + montant));
14       assert((solde >= minSolde));
15     }
16     public void debiter(int montant) throws ExceptionMontantIllegal { ... }
17     public int getSolde() { ... }
18 }
```

Programmation défensive :

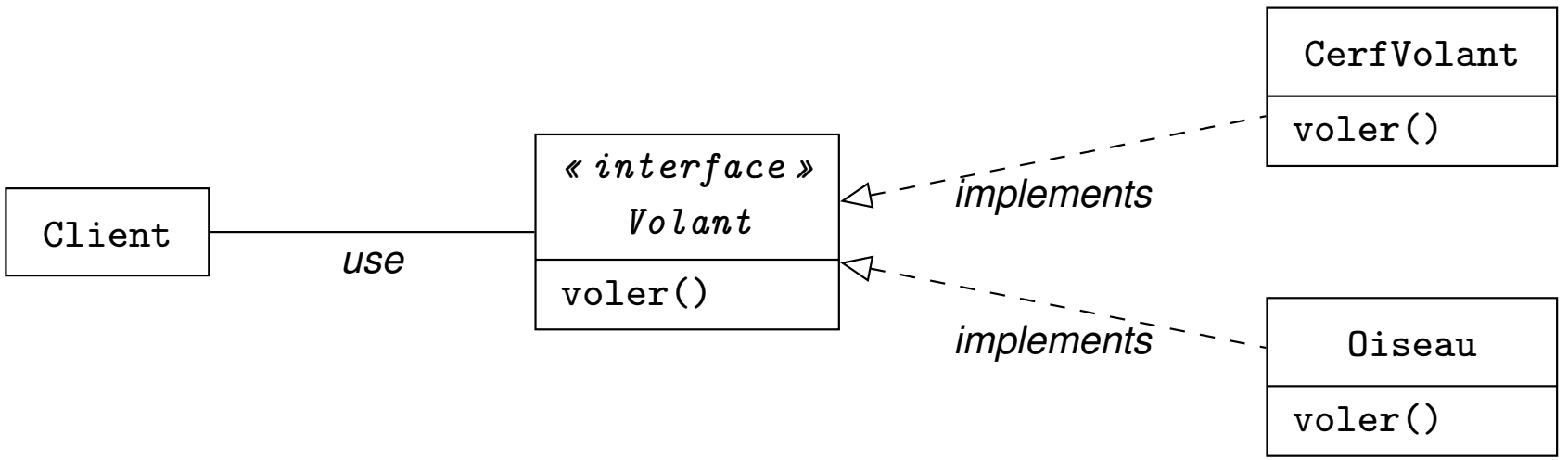
- lever une exception si les préconditions ne sont pas réalisées
- vérifier les postconditions à l'aide d'assertions (des fonctions « pures » qui ne changent pas l'état.

- Des animaux : `Animal`, `Poisson`, `Oiseau`
- Tous les animaux mangent et dorment mais les poissons nagent et les oiseaux peuvent voler
- Un client pourra faire voler tous les oiseaux



- L'association impose aux instances de `Client` de n'être en lien qu'avec des sous-types de `Oiseau`
- Et si le client voulait faire voler des avions ou des cerf-volants ?

- On définit un contrat (l'*interface*)
  - forme très simple de contrat
  - signature des méthodes pouvant être invoquées
- Une instance de `Client` peut être en lien avec tout objet respectant le contrat (*réalisant* l'interface)



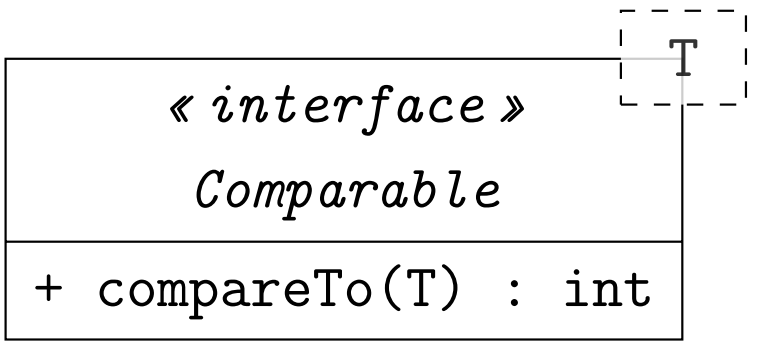
```
1  public class ComparableProduct extends Product
2                                  implements Comparable<Product> {
3    ComparableProduct(String name) {
4      super(name);
5    }
6    public int compareTo(Product other) {
7      if (other == null)
8        throw new NullPointerException();
9      return getName.compareTo(other.getName());
10   }
11 }
```

Une interface définit donc des obligations de programmation qu'il faut satisfaire.

L'interface `Comparable`

```java
package java.lang;

public interface Comparable<T> {
  public int compareTo(T o);
}
```



Permet de définir un tri de tableau

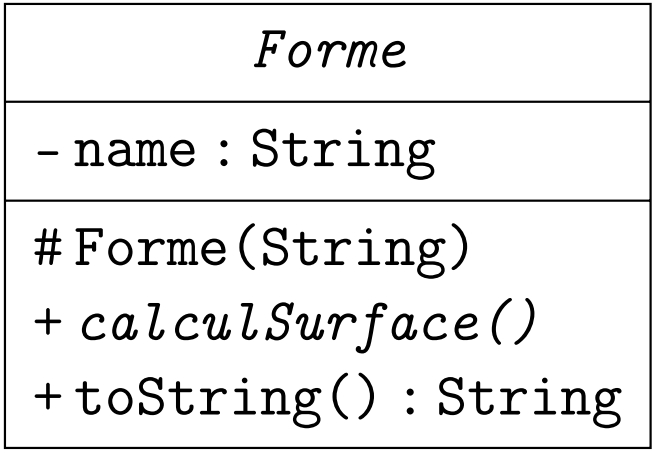
```java
void trier(Comparable[] tab) { ... }
```

- C'est une liste de services $\Rightarrow$ un Type
- Une (sorte de) classe :
  - sans code (les méthodes n'ont pas de corps, sauf "default method" à partir de Java 8)
  - sans état
- Une interface peut hériter (mot clé `extends`) de plusieurs interfaces
- Une classe peut réaliser (mot clé `implements`) plusieurs interfaces
- Une classe qui réalise une interface doit fournir toutes les méthodes de l'interface ou être abstraite
- Permet de séparer *une spécification* de *ses implantations*

- On veut pouvoir appliquer des opérations sur des objets
- Les opérations doivent s'appliquer sur des classes différentes (`CerfVolant`, `Oiseau`)
- Les classes n'ont pas la même classe mère, donc impossible d'utiliser l'héritage
- On veut être indépendant (de la forme) des fournisseurs...

- Classe abstraite = classe avec des méthodes abstraites
- Une méthode abstraite est un contrat (une obligation de programmation)
- C'est donc un mélange de classe et d'interface
  - Une classe abstraite peut avoir des constructeurs pour initialiser ses attributs
- Les héritières concrètes doivent réaliser toutes les méthodes abstraites
- Mais historiquement en PPO la classe abstraite est un concept antérieur à l'interface
- Une classe abstraite peut avoir une structure, du code concret et même des constructeurs

```java
package dessin;
public abstract class Forme {
  private String name;
  protected Forme(String name) { this.name = name; }
  public String toString() { return name; }
  public abstract float calculSurface() ;
}
```



- Les classes abstraites ne sont pas instanciables
- Mais elles peuvent avoir des constructeurs
- Une héritière qui réalise toutes les méthodes abstraites est dite concrète sinon elle est abstraite
- Hériter vs Réaliser :
  - L'interface :
    - définit un contrat (syntaxique)
    - permet d'hériter de plusieurs contrats (interface)
    - demande d'écrire plus de code
  - La classe abstraite :
    - permet d'hériter d'un contrat, d'une structure et de code
    - on ne peut hériter que d'une seule classe... (en Java)

- Une classe définie à l'intérieur d'une autre
  - le nom de la classe `I` interne à `A` est `A.I`
  - `A.I` peut être visible ou non de l'extérieur
- La classe interne a accès à toute la classe englobante et vice versa (même les informations privés)
- Pour grouper des classes qui sont logiquement liées
  - une instance de `A.I` est liée à une instance de `A`
  - pour instancier `A.I`, il faut demander à une instance de `A`

Voir les *inner classes* ici : `https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html`.

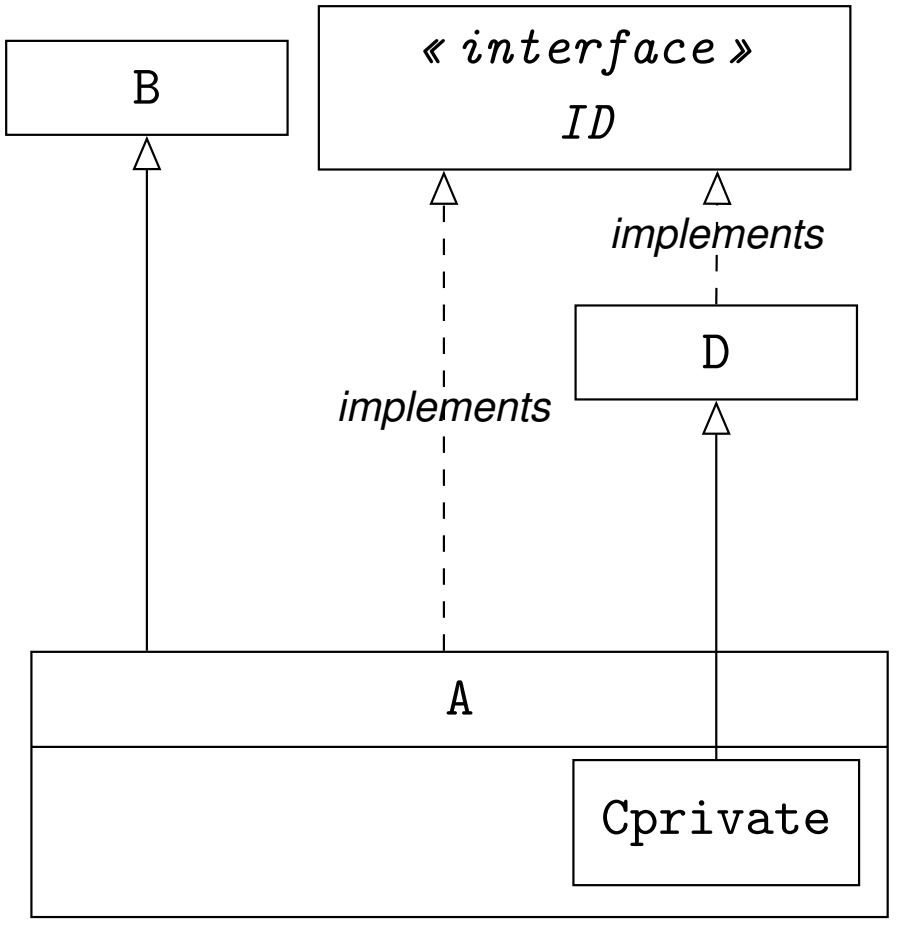
```java
public class A {
  private String b = "A.b";
  private I i;
  // Classe visible
  public class I {
    private String a = "A.I.a";
    public void m() {
      System.out.println("I->A : "
       + a);
      System.out.println("I->A : "
       + b);
    }
  }
  public I newInterne() {
    i = new I();
    return i;
  }
  public void m() {
    System.out.println("A : " + i.a);
  }
}
```

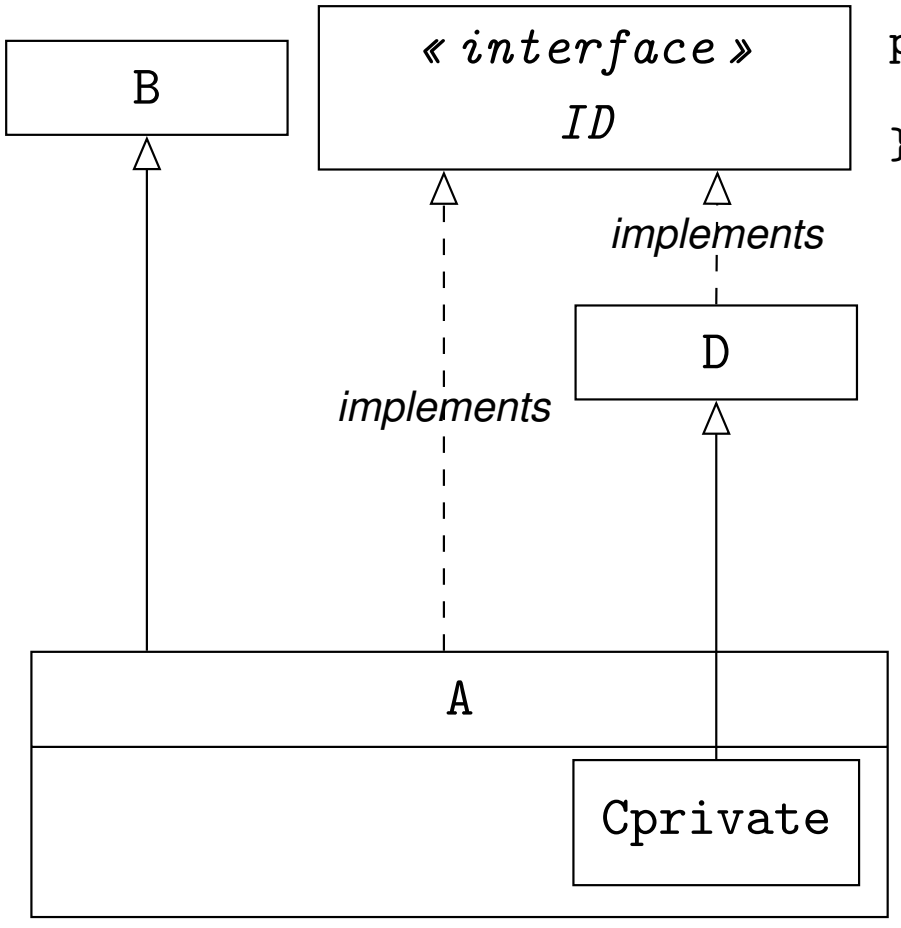
```java
public class Main {
  public static void main(String[] args)
    A objA = new A();

    //Instanciation par methode
    A.I objI1 = objA.newInterne();
    //Instanciation par constructeur
    A.I objI2 = objA.new I();

    // Affiche I->A : A.I.a
    // puis I->A : A.b
    objI1.m();

    // Affiche A : A.I.a
    objA.m();
  }
}
```


B
« interface »
ID
implements
D
implements
A
Cprivate



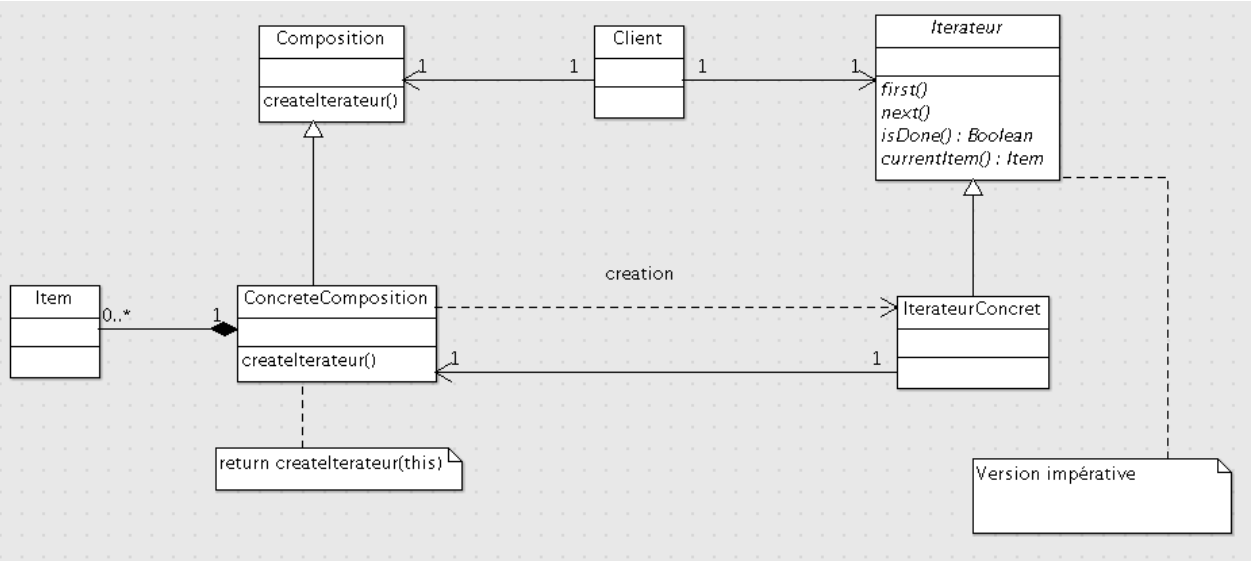

```java
public interface ID {
  void sayHello();
}
```

```java
public class D implements ID {
  public void sayHello() {
    System.out.println("Hello");
  }
}
```

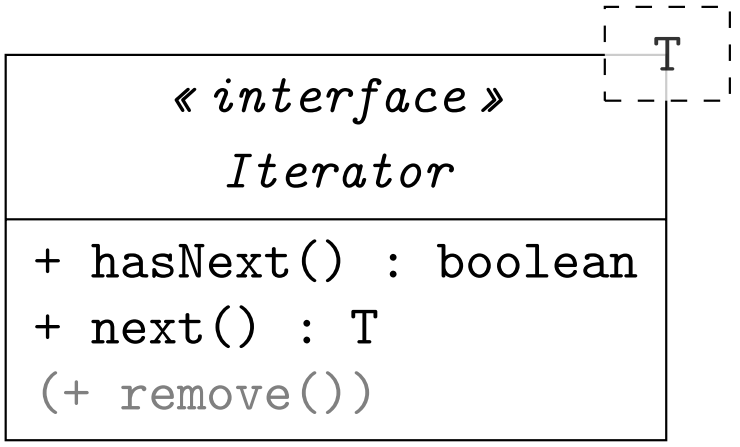
```java
public class A extends B implements ID {
  private CPrivate cPriv;
  private class CPrivate extends D { }
  public A() { cPriv = new CPrivate(); }
  public void sayHello() {
    cPriv.sayHello();
  }
}
```

Exemple : Itérateur

- Parcourir des structures de données (listes, arbres...) de manière itérative et exhaustive
- Sans révéler la structure interne
- Avec une interface uniforme

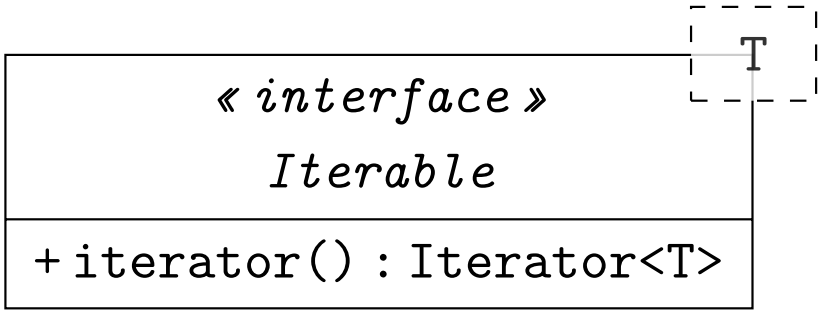


- Le client accède à une composition, en extrait un itérateur et utilise cet itérateur pour accéder à tous les éléments de la composition.



- Pour parcourir

```
1 // Variable l est de type ArrayList par exemple
2 Iterator<Elem> it = l.iterator();
3 while(it.hasNext()) {
4   Elem element = it.next();
5   // Travailler avec element
6 }
```



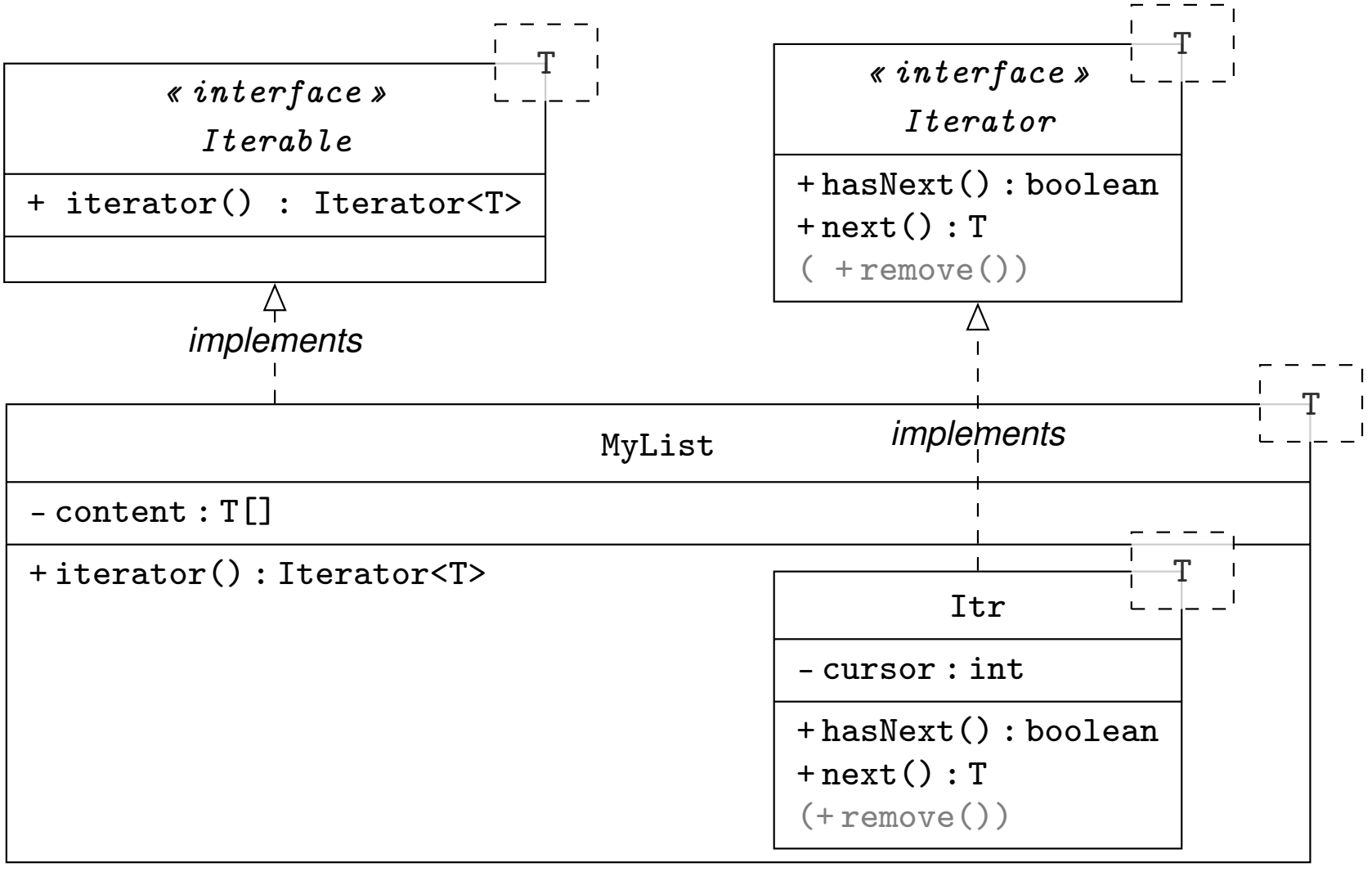
- Faciliter le parcours par un *foreach* (Java > 5)

```
Iterable<T> l;
for (T elem : l)
  ...
```

- Les tableaux sont *itérables* (plus besoin d'indice)

```
int result = 0;
for (int i : tab)
  result += i;
```

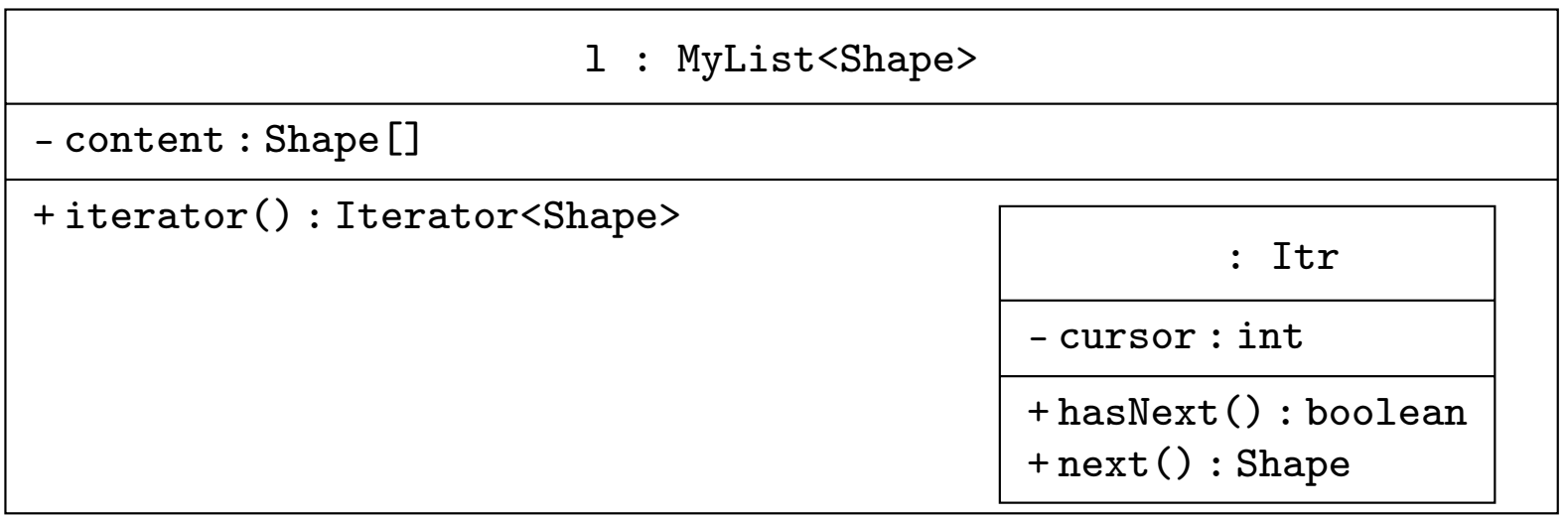
NB : attention, itère TOUT le tableau (pb s'il n'est pas "plein")


T
« interface »
Iterable
+ iterator() : Iterator<T>
T
« interface »
Iterator
+hasNext() : boolean
+next() : T
(+remove())
implements
implements
T
MyList
- content : T[]
+iterator() : Iterator<T>
T
Itr
- cursor : int
+hasNext() : boolean
+next() : T
(+remove())

Un exemple en exécution

```
1 void drawAll (Iterable<Shape> l) {
2   for (Shape shape : l)
3     shape.draw();
4 }
```

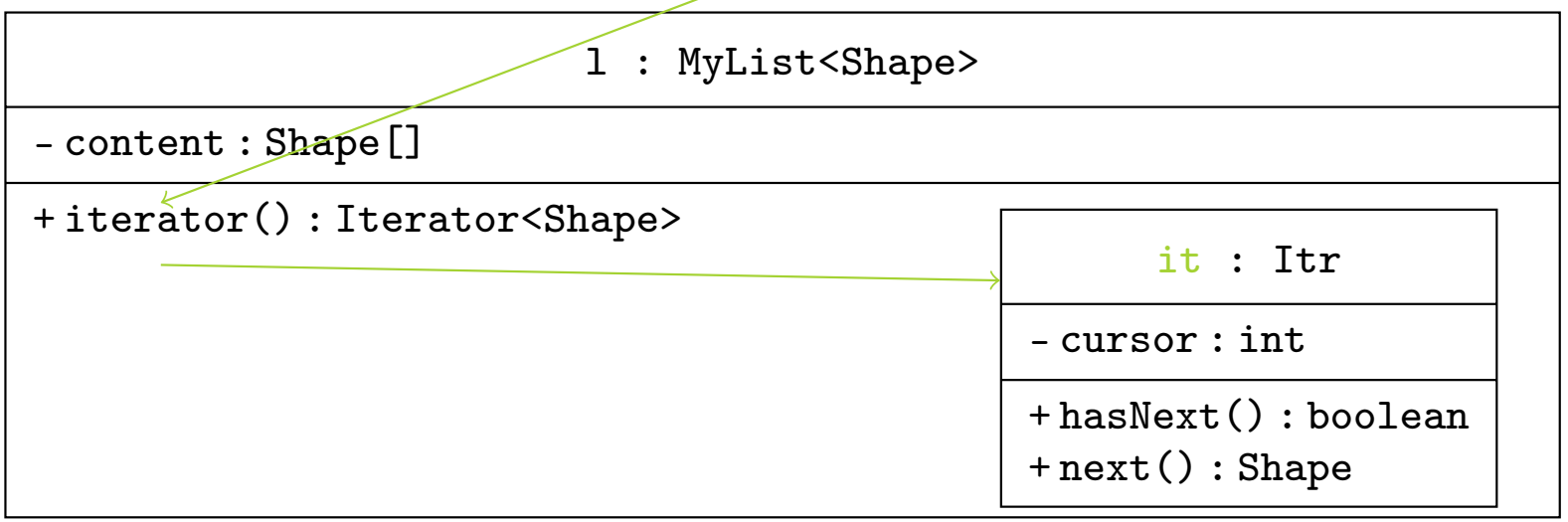
```
1 void drawAll (MyList<Shape> l) {
2   Iterator<Shape> it = l.iterator()
3   for (;it.hasNext();)
4     Shape shape = it.next();
5     shape.draw();
6 }
```



Un exemple en exécution


```
1 void drawAll (Iterable<Shape> l) {
2   for (Shape shape : l)
3     shape.draw();
4 }
```

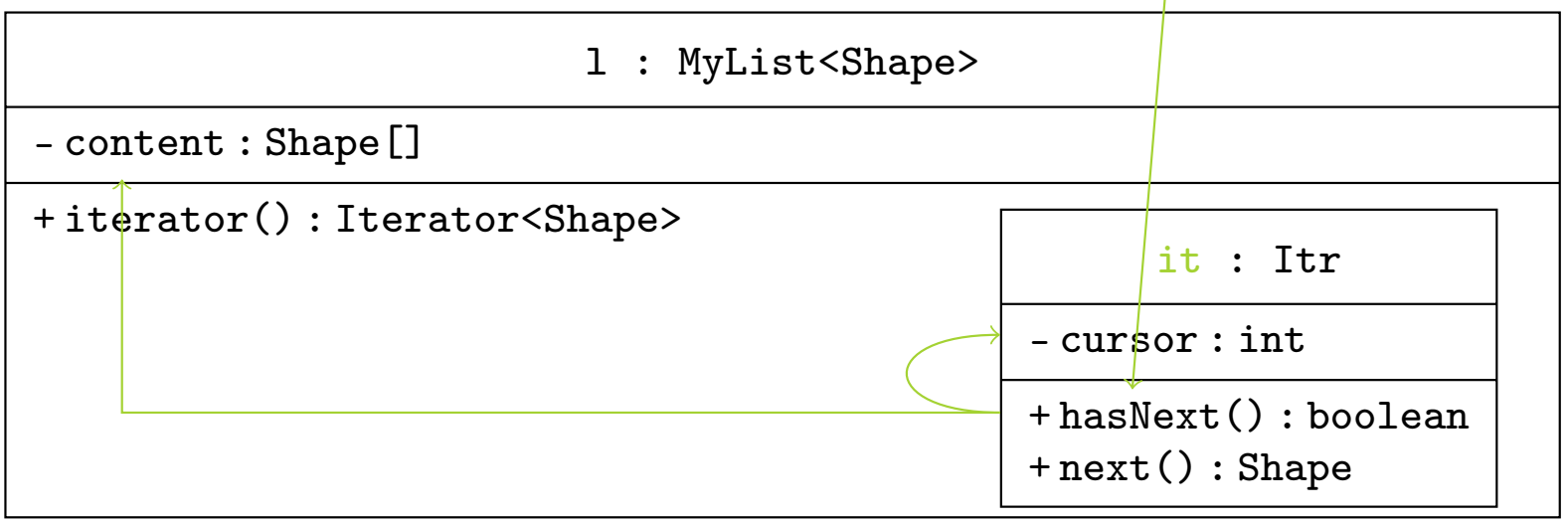
```
1 void drawAll (MyList<Shape> l) {
2   Iterator<Shape> it = l.iterator()
3   for (;it.hasNext();)
4     Shape shape = it.next();
5     shape.draw();
6 }
```



Un exemple en exécution

```
1 void drawAll (Iterable<Shape> l) {
2   for (Shape shape : l)
3     shape.draw();
4 }
```

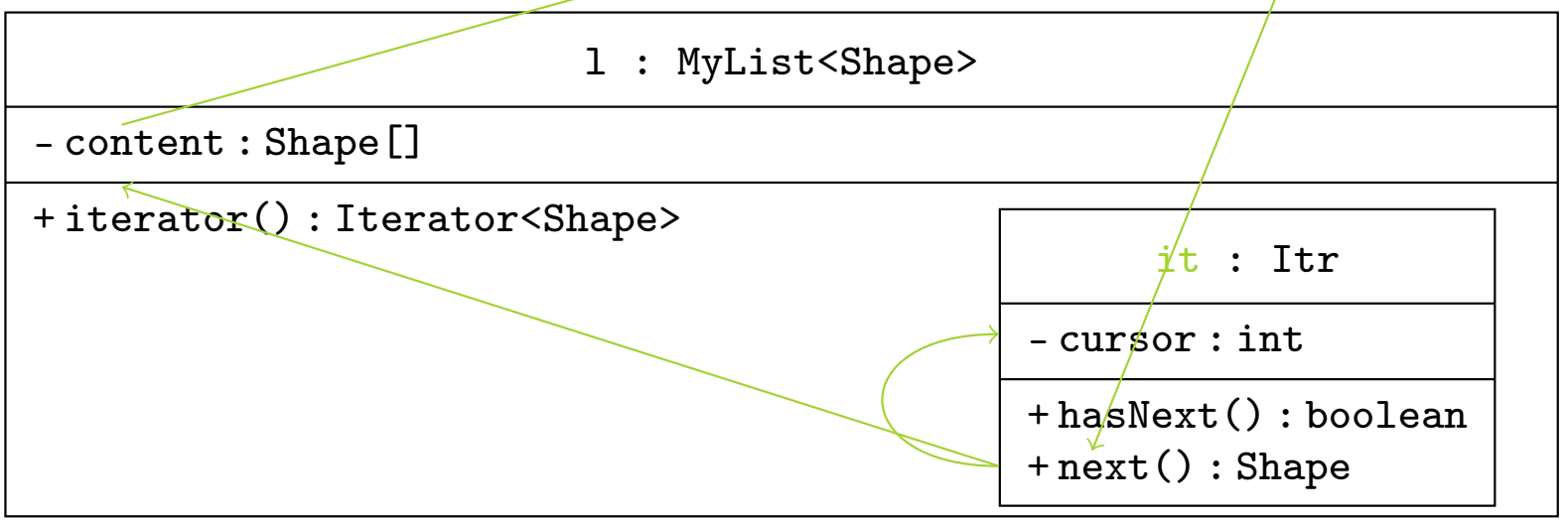
```
1 void drawAll (MyList<Shape> l) {
2   Iterator<Shape> it = l.iterator()
3   for (;it.hasNext();)
4     Shape shape = it.next();
5     shape.draw();
6 }
```



Un exemple en exécution

```
1 void drawAll (Iterable<Shape> l) {
2   for (Shape shape : l)
3     shape.draw();
4 }
```

```
1 void drawAll (MyList<Shape> l) {
2   Iterator<Shape> it = l.iterator()
3   for (;it.hasNext();)
4     Shape shape = it.next();
5     shape.draw();
6 }
```



Le paradigme objet offre une façon de penser les programmes en mettant l'accent sur la notion de responsabilité.

Un objet a la responsabilité de gérer correctement les évolutions de son état (donc d'informations.) C'est pour cela qu'on regroupe attributs et méthodes.

Une classe a la responsabilité de définir et créer des instances correctes. C'est pour cela que les constructeurs fabriquent des objets corrects et, puisque c'est possible automatiquement, le programmeur ne gère pas la destruction des objets.

Cette notion de responsabilité peut ainsi être utilisée très largement pour :

- les méthodes (services)
- les objets
- les classes, classes abstraites, interfaces
- les packages

Organisation

- Documenter
- Vérifier, valider
- Faire relire

Conception

- Chercher la modularité
- Réutiliser
- Contractualiser
- Couplage et cohésion

Programmation

- Séparer la spécification de sa (ou ses) réalisation(s)
- Déclarer abstrait, instancier, concrétiser

Vocabulaire (Bilan)

- classe, objet, instance
- classe abstraite, interface
- type*
- attribut, méthode, constructeur
- signature*, corps*
- méthode abstraite
- visibilité*, portée*
- héritage
- redéfinition, surcharge
- liaison dynamique
- généricité*

* pas spécifique au paradigme objet