Haute École Léonard de Vinci
SCIENCES ET TECHNIQUES

Analyse et Programmation Orientée Objet

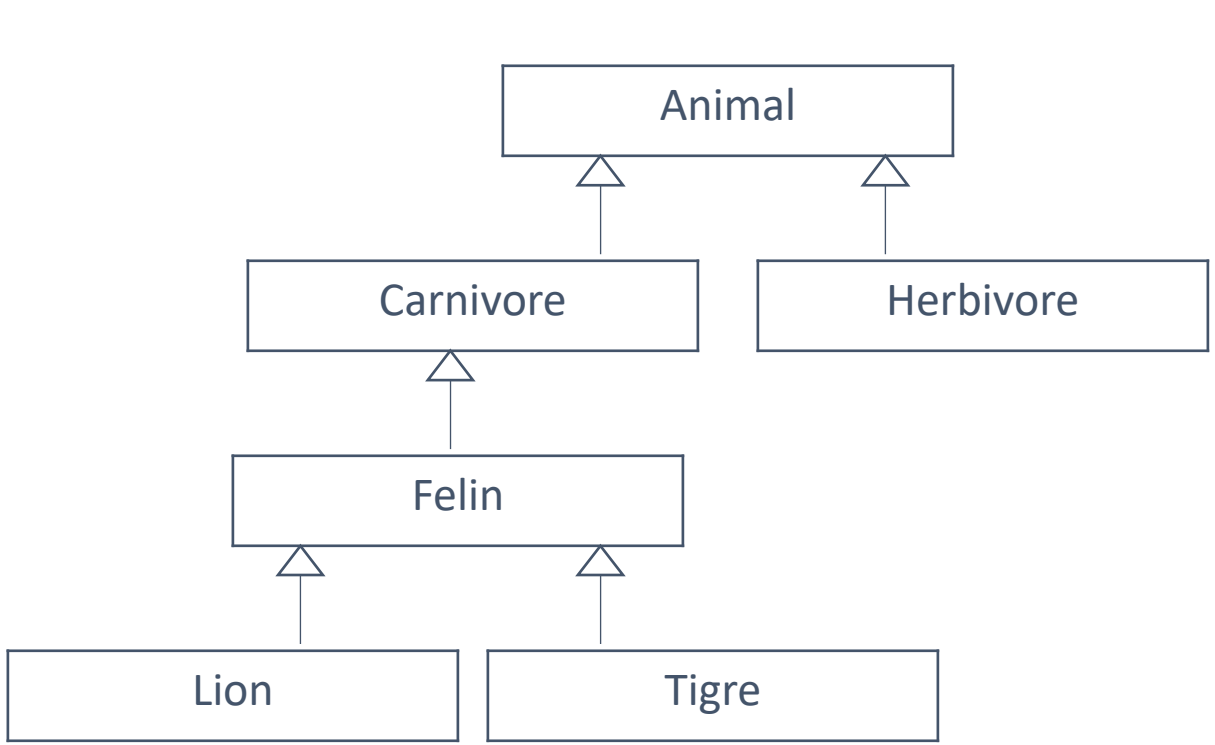
Présentation 7

Classes abstraites, interfaces, `Iterable`

Classes abstraites

Niveaux d'héritage

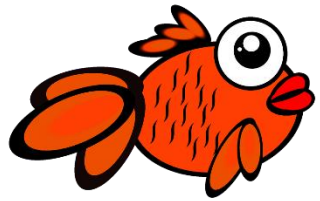
De plus en plus abstrait



Classe abstraite

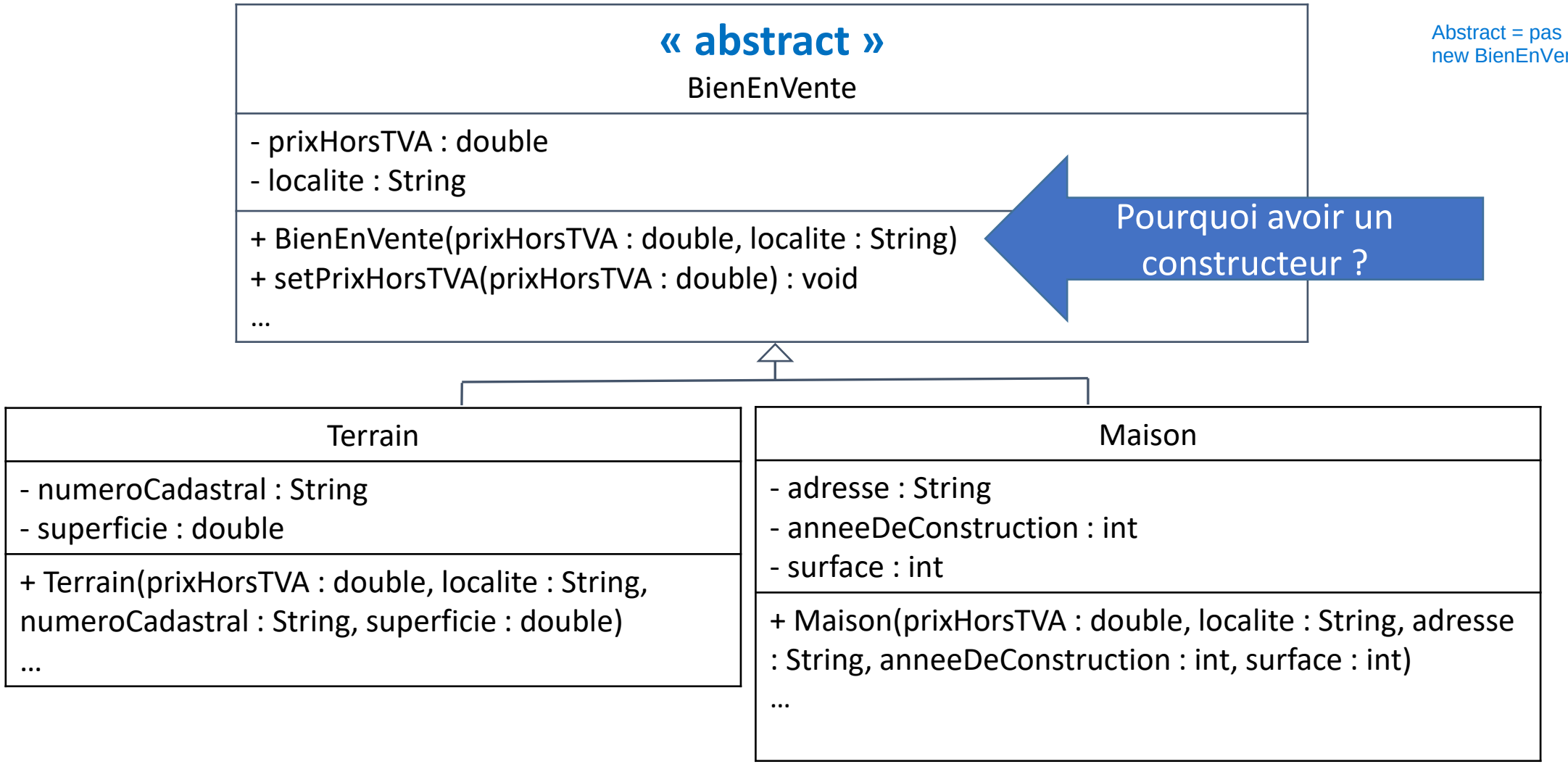
- Instancier Animal ? Carnivore ? Lion ?

- classe **abstraite** :
  - **pas** destinée à être **instanciée**
  - permet de définir des propriétés (attributs et méthodes) communes à différentes classes une seule fois
  - évite donc des répétitions



???

Classe abstraite en UML



Abstract = pas de new BienEnVente

Classe abstraite en Java

```
public abstract class BienEnVente{
  private double prixHorsTVA;
  private String localite;

  public BienEnVente(double prixHorsTVA,String localite){
     …
  }
  …
}
```

```
public class Terrain extends BienEnVente{
  private String numeroCadastral;
  private double superficie

  public Terrain(double prixHorsTVA, String localite,
  String numeroCadastral, double superficie){
     super(prixHorsTVA, localite);
     this.numeroCadastral = numeroCadastral;
     this.superficie = superficie;
  }
  …
}
```

Remarques

- Impossible d'invoquer un constructeur d'une classe abstraite autrement qu'en utilisant l'instruction **`super(…)`**
  - Impossible donc de de l'utiliser en faisant `new Constructeur()`

```
BienEnVente bien;
bien = new BienEnVente(50000,"Namur");
```

Erreur de compilation

- Pour le reste, tout fonctionne comme dans l'héritage classique
- Une classe peut être **abstraite** même si elle **hérite** d'une classe **concrète**

Méthode abstraite

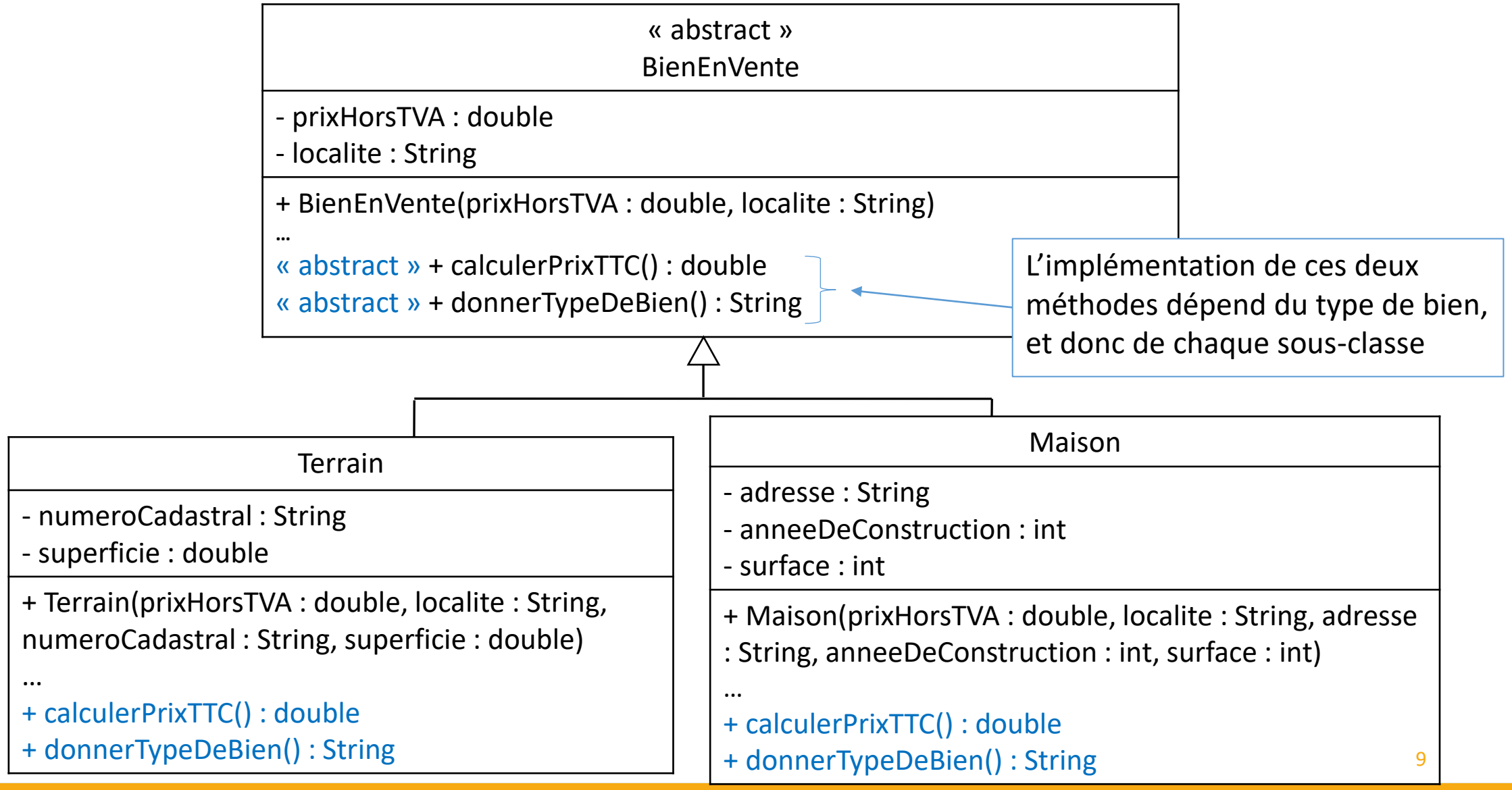
- Une méthode abstraite = méthode **sans code** (uniquement un en-tête)

Une méthode abstraite :

- ne peut pas être implémentée dans la superclasse car son **implémentation dépend de la sous-classe**
- **doit** donc être définie dans chacune des sous-classes

- Si une classe contient **une méthode abstraite**, cette classe doit être **abstraite**

Méthode abstraite en UML



Méthode abstraite en Java

```
public abstract class BienEnVente{
  private double prixHorsTVA;
  …
  public double getPrixHorsTVA(){return prixHorsTVA;}
  public abstract double calculerPrixTTC();
  public abstract String donnerTypeDeBien();
}
```


```
public class Terrain extends BienEnVente{
  …
  public double calculerPrixTTC(){
    return getPrixHorsTVA()* 1.125;
  }
  public String donnerTypeDeBien(){
    return "Terrain";
  }
}
```

```
public class Maison extends BienEnVente{
  …
  public double calculerPrixTTC(){
    if (…)
       return getPrixHorsTVA() * 1.21;
    return getPrixHorsTVA() * 1.06;
  }
  public String donnerTypeDeBien(){
    return "Maison";
  }
}
```

Remarques

Une classe qui ne fournit pas une **implémentation** (dans la classe elle-même ou par héritage) de **toutes les méthodes abstraites**, héritées ou non, doit être elle-même **abstraite**

Dans une méthode d'une classe, on peut **faire appel** à une autre **méthode abstraite** de la classe

Les méthodes **static**, **final** ou **private** ne peuvent **pas** être **abstraites**

Interfaces

Interface

!= une classe abstraite

= un autre moyen de faire de l'abstraction en Java

- Définit un ensemble de méthodes disponibles (**non implémentées !**) pour un objet en cachant leur implémentation dans une (ou plusieurs) autre(s) classe(s)

= contrat de services proposés

- Ce sont les classes qui implémentent l'interface qui implémenteront les méthodes proposées par l'interface
- Une interface peut être implémentée par plusieurs classes

Interface

Une interface contient

- Des constantes (toujours *public static final*)
- Des méthodes non implémentées (toujours *public* !)

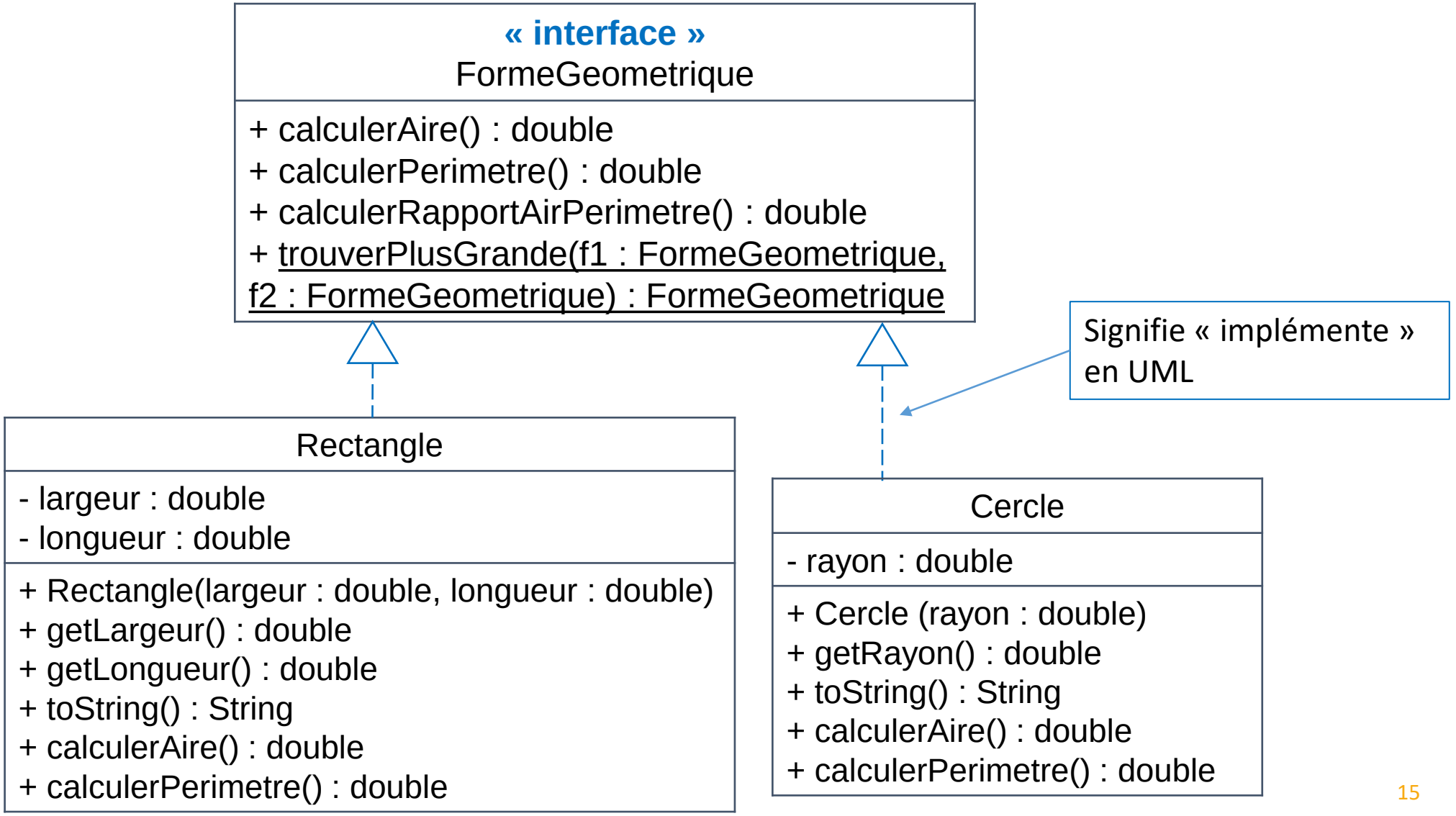
Parfois :

- Des méthodes de classe implémentées (*static*)
- Des méthodes implémentées par défaut (très rare)

Une interface ne contient pas

- D'attributs d'instance (un attribut est toujours *une constante* dans une interface)
- De constructeur (on ne peut pas instancier une interface)

Interface en UML

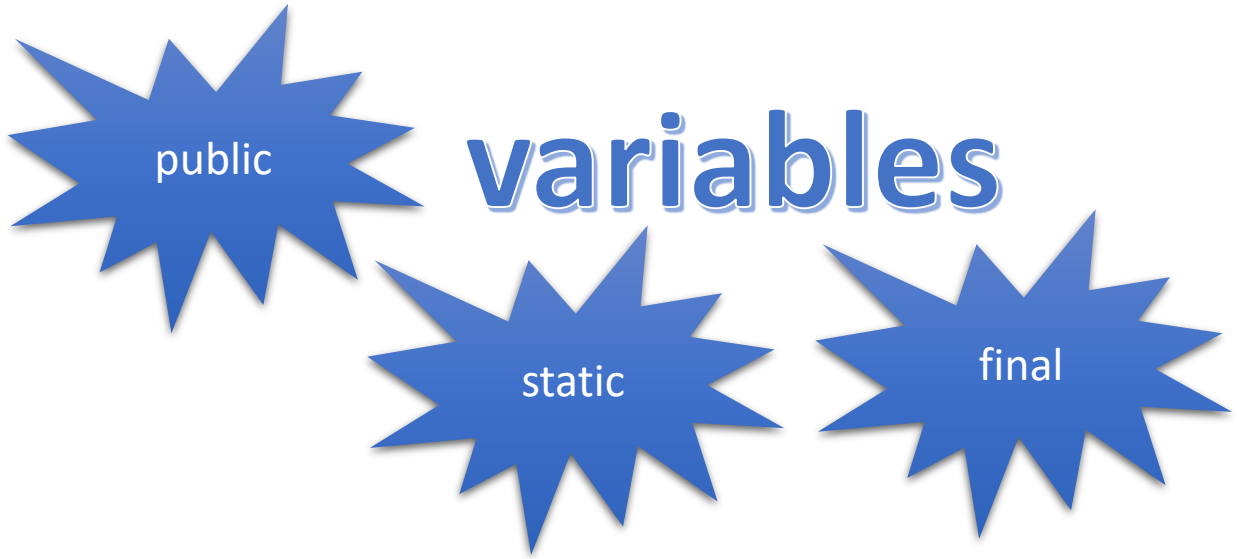


Les méthodes et variables d'une interface

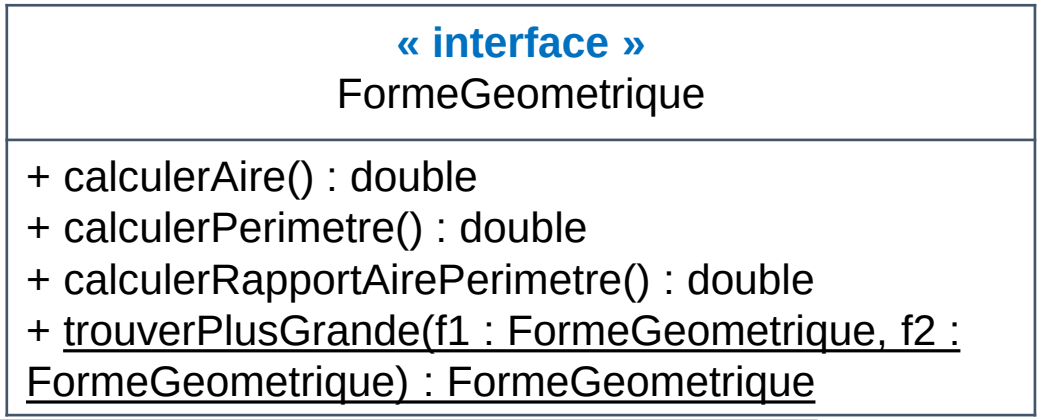
En java, pour une interface, par défaut

- toutes les méthodes sont `public`
- toutes les variables sont `public`, `static` et `final` (= constante)

Il est inutile d'indiquer ces modificateurs, ils sont là par défaut



Interface en Java




```
public interface FormeGeometrique{
    /* renvoie l'aire de la forme géométrique en cm2 */
    double calculerAire();
    /* renvoie le périmètre de la forme géométrique en cm */
    double calculerPerimetre();
    default double calculerRapportAirePerimetre(){
        return calculerAire()/calculerPerimetre();
    }
    static FormeGeometrique trouverPlusGrande(FormeGeometrique f1,
                                              FormeGeometrique f2){
        if (f1.calculerAire() > f2.calculerAire())
            return f1;
        return f2;
    }
}
```

Implémenter une interface en Java

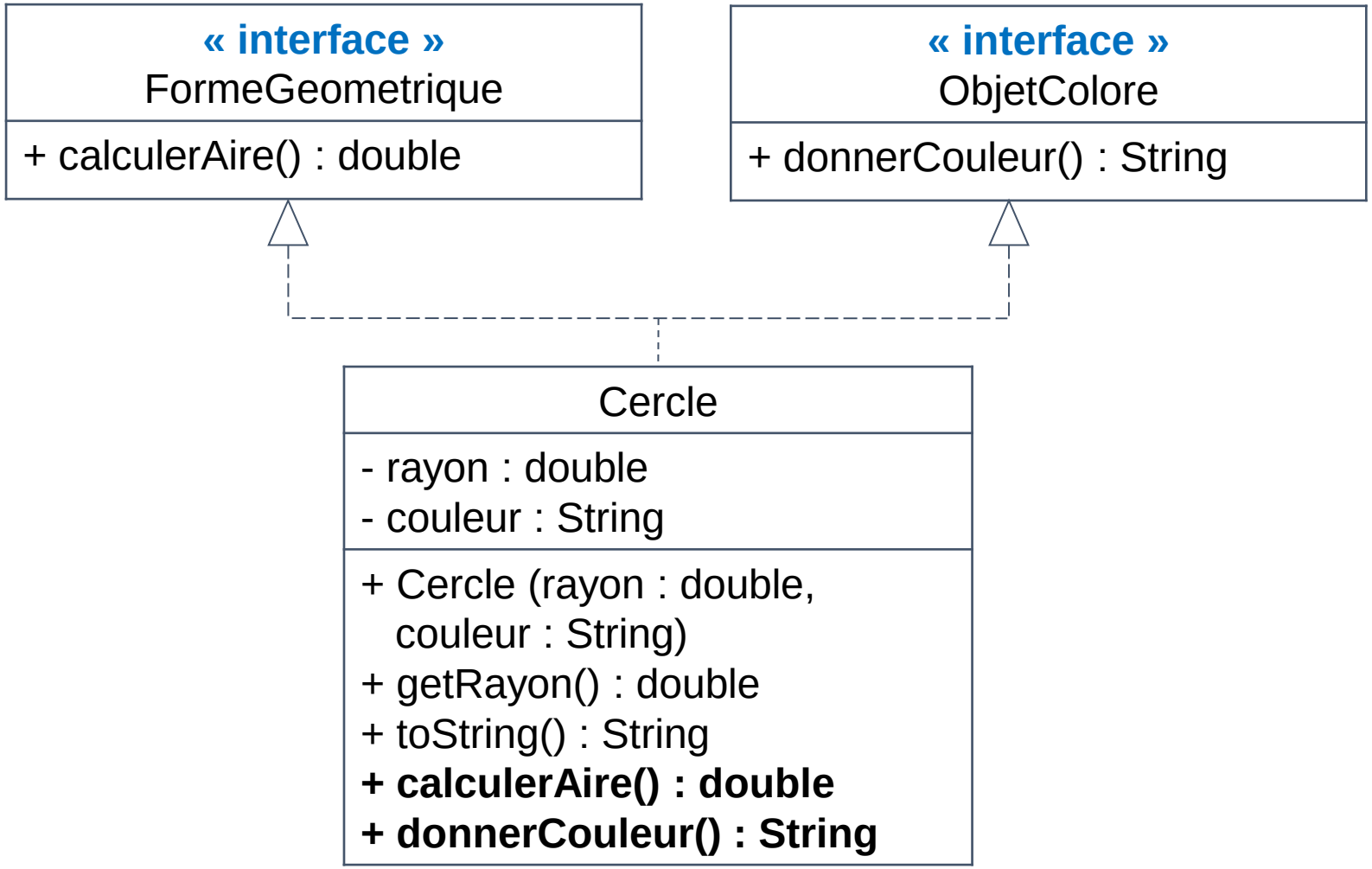
```
public interface FormeGeometrique{
     double calculerAire();
     double calculerPerimetre();
     default double calculerRapportAirePerimetre(){
         return calculerAire()/calculerPerimetre();
     }
     …
}
```

En Java, on utilise le mot **implements** pour signifier qu'une classe implémente une interface!

Cercle
- rayon : double
+ Cercle (rayon : double) + getRayon() : double + toString() : String + calculerAire() : double + calculerPerimetre() : double

```
public class Cercle implements FormeGeometrique{
   private double rayon;
   …
   public double calculerAire(){
       return rayon * rayon * Math.PI;
   }
   public double calculerPerimetre(){
       return 2 * rayon * Math.PI;
   }
   …
}
```

Une classe peut implémenter plusieurs interfaces



Implémentation de plusieurs interfaces

```
public interface FormeGeometrique{
    /* renvoie l'aire de la forme géométrique en cm2 */
    double calculerAire();
}
```

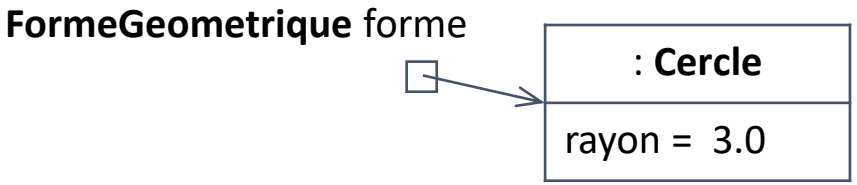
```
public interface ObjetColore{
    /* renvoie la couleur de l'objet */
    String donnerCouleur();
}
```

```
public class Cercle implements FormeGeometrique, ObjetColore{
  private double rayon;
  private String couleur;
  …
  public double calculerAire(){
    return rayon * rayon * Math.PI;
  }
  …
  public String donnerCouleur(){
    return couleur;
  }
  …
}
```

La classe Cercle implémente 2 interfaces

Utilisation d'une interface

- On peut déclarer des variables de type interface :
  - `FormeGeometrique forme;`
- Pour instancier une variable de type interface, il faut utiliser une classe qui implémente cette interface :
  - `forme = new Cercle(3.0);`
  - ~~`forme = new FormeGeometrique();`~~ Il n'y a pas de constructeur dans une interface
- On ne peut utiliser **que** les méthodes qui sont déclarées dans l'interface :
  - `double aire = forme.calculerAire();`
  - ~~`double rayon = forme.getRayon();`~~ `getRayon()` n'est pas déclarée dans l'interface



Intérêt de l'interface

Écrire du code indépendant de l'implémentation

```
public class DesFormesGeometriques{
    private ArrayList<FormeGeometrique> formes;
        …
    public FormeGeometrique trouverPlusGrande(){
        if (formes.size()== 0) return null;
        FormeGeometrique formeMax = null;
        double max = 0;
        for (FormeGeometrique forme : formes){
            if(forme.calculerAire()>=max){
                formeMax = forme;
                max = forme.calculerAire();
            }
        }
        return formeMax;
    }
    …
(suite)
```

Il peut y avoir des cercles, des rectangles, … dans la liste

L'implémentation de `calculerAire()` sera différente selon la forme

Suite de la classe `DesFormesGeometriques`

```
    …
    public boolean contient(FormeGeometrique forme){
            return formes.contains(forme);
    }
}
```

Où faut-il écrire les méthodes `equals()` et `hashCode()` ?

Il faut le faire dans chaque classe implémentant l'interface ! Java n'autorise pas l'écriture de la méthode `equals()` dans une interface.

L'interface `Iterable`

Interface `Iterable<T>`

T correspond à un **type générique** que l'on peut remplacer par le type souhaité : FormeGeometrique, Cercle, Personne, …

= Interface implémentée lorsqu'on a une classe possédant une « liste » d'objets qu'on veut pouvoir parcourir à l'aide d'un `for each` lorsqu'on est en dehors de la classe (et qu'on n'a donc pas accès à la liste à cause de l'encapsulation)

- 1 seule méthode imposée par l'interface `Iterable<T>` :
  - `iterator() : Iterator<T>`

Implémenter `Iterable<T>`

```
import java.util.ArrayList;
import java.util.Iterator;

public class DesFormesGeometriques implements Iterable<FormeGeometrique>{

    private ArrayList<FormeGeometrique> formes;

    @Override
    public Iterator<FormeGeometrique> iterator() {
        return formes.iterator();
    }
    …
}
```

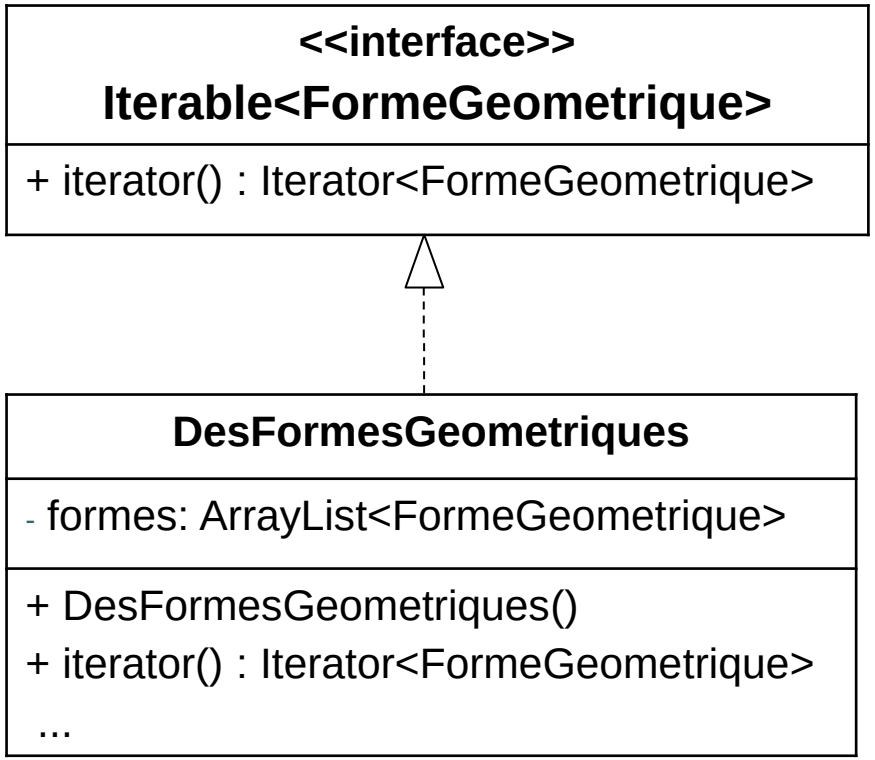
For each – interface `Iterable<T>`

Classe qui implémente `Iterable<T>` pour pouvoir utiliser un `for each` en dehors de la classe

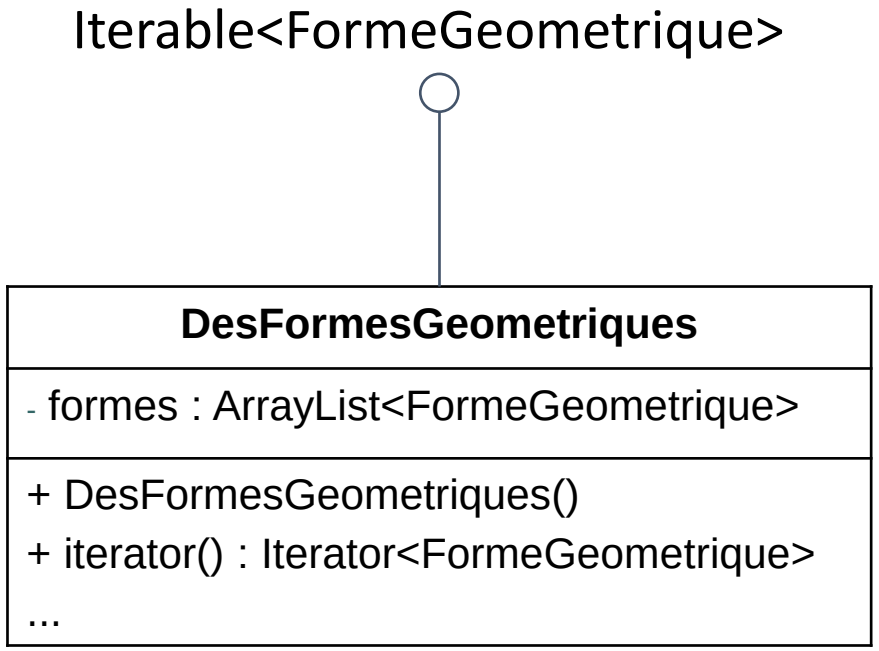
Type des objets renvoyés par l'itérateur

```
public class TestFormesGeometriques {
    public static void main(String[] args) {
        DesFormesGeometriques liste = new DesFormesGeometriques();
        liste.ajouter(new Cercle(5.0));
        liste.ajouter(new Cercle(8.0));
        //parcourir les formes géométriques pour sommer les
        // aires de toutes les formes
        double somme = 0;
        for (FormeGeometrique forme : liste){
            somme += forme.calculerAire();
        }
        System.out.println(somme);
    }
}
```

Interfaces en UML



ou



Remarque : dans le cours, on n'utilisera la deuxième notation que lorsque l'interface est fournie par java

Classe abstraite vs Interface

	Attributs	Constructeurs	Méthodes	But
Classe abstraite	Pas de restrictions	Invoqués que par les constructeurs des sous-classes uniquement	• Pas de restrictions • Peuvent être abstract (pas implémentées) ou implémentées	Factoriser du code (éviter de dupliquer)
Interface	Uniquement des constantes : public, static et final	Pas de constructeur	• Toutes public et pas implémentées • Dans la classe qui implémente l'interface, toutes les méthodes doivent être réimplémentées	Définir des contrats de service (lister les méthodes disponibles /à implémenter)

Une classe peut implémenter plusieurs interfaces mais ne peut hériter que d'une seule classe !

Des questions

Les slides suivants sont pour information

Interface `Iterator<T>`

- Un itérateur permet de parcourir les objets qui se trouvent dans une « liste »
- `<T>` : type des objets contenus dans la liste
- Cette interface impose les méthodes suivantes :
  - `hasNext():boolean`
  - `next():T`
  - `remove():void`

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Iterator.html

`hasNext():boolean`

- Sa responsabilité :
  - Signaler s'il reste un élément de la liste qui n'a pas encore été consulté
- Renvoie :
  - true : si encore un élément à consulter
  - false : si plus d'élément

`next():T`

- Sa responsabilité :
  - Renvoyer l'élément suivant, présent dans la liste parcourue par l'itérateur
- Renvoie :
  - L'élément suivant de type T
- Exception :
  - Une `NoSuchElementException` est levée s'il n'y a plus d'élément à consulter
  - Une `ConcurrentModificationException` est levée si la liste a été modifiée, autrement que via l'itérateur, depuis le moment où l'itérateur a été créé

`remove():void`

- Sa responsabilité :
  - Supprimer l'élément de la collection qui a été renvoyé par le dernier `next()`
- Exceptions :
  - `UnsupportedOperationException` si les suppressions sont interdites
  - `IllegalStateException` si un appel à `remove()` vient d'être fait ou si la méthode `next()` n'a pas encore été appelée
  - `ConcurrentModificationException` si la liste a été modifiée, autrement que via l'itérateur, depuis le moment où l'itérateur a été créé

For each : ce que fait Java

```
double somme = 0;
for (FormeGeometrique forme: liste){
    somme += forme.calculerAire();
}
```

```
double somme = 0;
Iterator<FormeGeometrique> formeIt = liste.iterator();
while(formeIt.hasNext()){
        FormeGeometrique forme = formeIt.next();
        somme += forme.calculerAire();
}
```