



# Programmation orientée objet avancée

## *Introduction*

# Programmation orientée objet avancée

## 1. Prérequis

# Prérequis

- Encapsulation – Information Hiding
  - Variables d'instance privées
  - Getters/setters (éventuels) publiques
- Liens entre classes
  - Relations 1 à N
    - Une variable d'instance de type référence dans une classe
    - Un tableau d'objets dans l'autre classe
  - Relations N à N
- Héritage
- Polymorphisme
- Variable et méthode de classe (**static**)
- Classe abstraite et interface

# Type de passage des arguments

```
public class Person {  
    private String name ;  
    public Person (String name) {  
        this.name = name ; }  
    public void setName (String name)  
...  
    public String getName( ) ...  
}
```

```
public class Modifier {  
    void intModifier (int a) {  
        a ++ ; }  
    void personModifier (Person p) {  
        p.setName ("Jules") ; }  
}
```

Passage par copie

Passage par référence

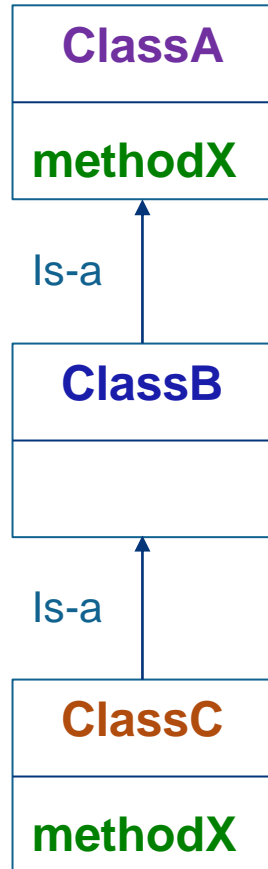
```
public class Main {  
    public static void main(String[] args) {  
        Modifier modifier = new Modifier( ) ;  
  
        int x = 20 ;  
        modifier.intModifier(x) ;  
        System.out.println(x) ;  
        (1)  
  
        Person pierre = new Person("Pierre") ;  
        modifier.personModifier(pierre) ;  
        System.out.println(pierre.getName( )) ;  
        (2)  
    }  
}
```

## Affichages

(1) ?  
(2) ?

(1) 20  
(2) Jules

# Polymorphisme



**ClassA** a = ... ; // initialisation de a  
**a.methodX()** ; ⇒ OK à la compilation!

*Quelle méthode sera appelée?*

⇒ *Dépend de l'initialisation de l'objet a*

## Exemples

a = new **ClassA()** ;

⇒ **methodX** de la **ClassA** qui sera exécutée

a = new **ClassB()** ;

⇒ **methodX** de la **ClassA** qui sera exécutée

a = new **ClassC()** ;

⇒ **methodX** de la **ClassC** qui sera exécutée

# static

Une **variable de classe**

= une **caractéristique** de la classe

= une **propriété** de la classe

⇒ **un seul espace alloué en mémoire** pour une variable de classe

**quel que soit le nombre d'objets de la classe créés : 0, 1 ou plusieurs**



Mot réservé **static**

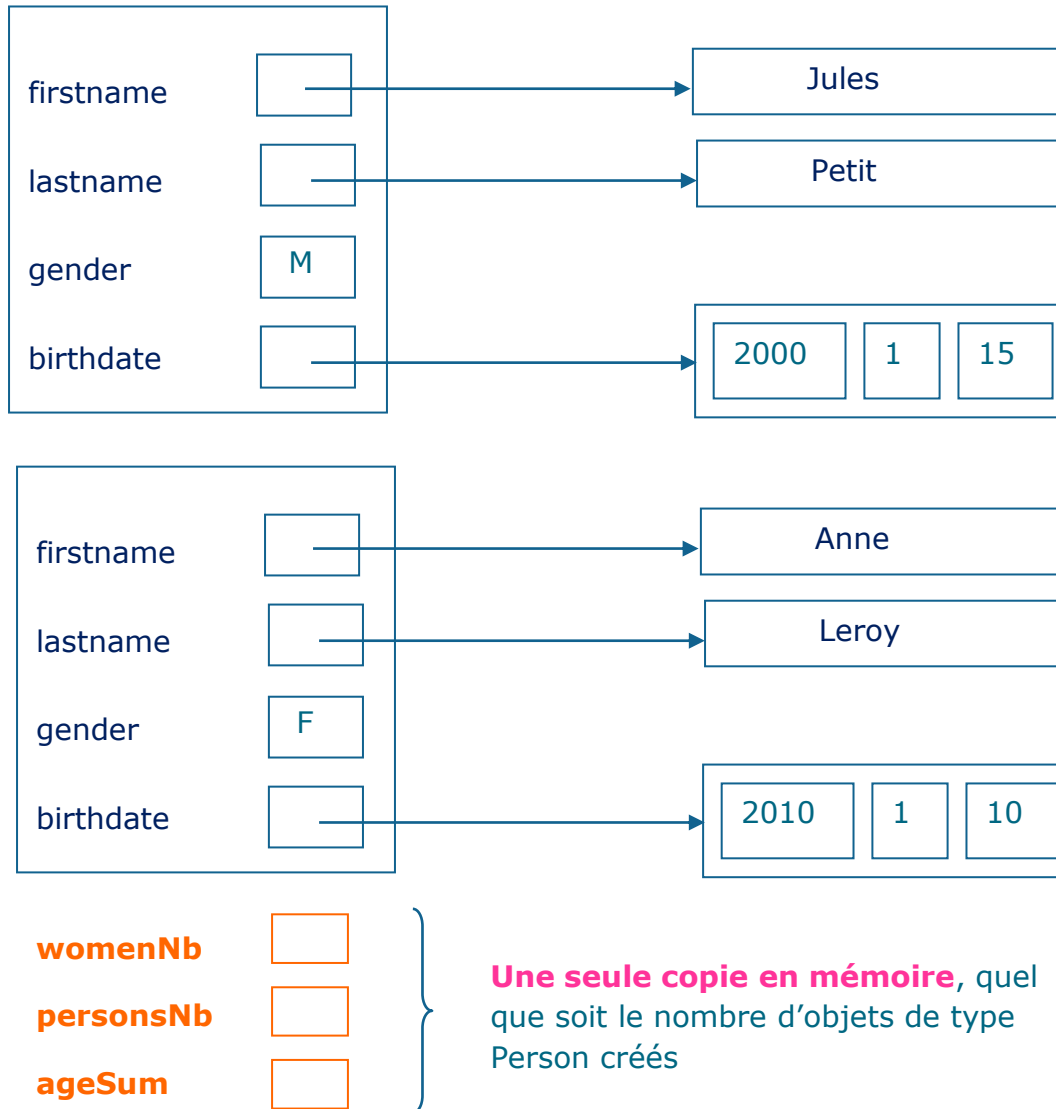
# static

Person
<ul style="list-style-type: none"><li>- firstname</li><li>- lastname</li><li>- gender</li><li>- birthdate</li><li>- <b>static</b> womenNb</li><li>- <b>static</b> personsNb</li><li>- <b>static</b> ageSum</li></ul>
<ul style="list-style-type: none"><li>+ age()</li><li>+ toString()</li></ul>

} Variables de  
classe

```
public class Person {  
    private String firstname;  
    private String lastname;  
    private char gender;  
    private LocalDate birthdate;  
  
    private static int womenNb = 0 ;  
    private static int personsNb = 0;  
    private static int ageSum = 0;  
  
    ...  
}
```

# static





```
public class Person {
    private String firstname;
    private String lastname;
    private char gender;
    private LocalDate birthdate;
    private static int womenNb = 0;
    private static int personsNb = 0;
    private static int ageSum = 0;

    public Person(String firstname, String lastname, char gender, int day, int month, int year) {
        ...
        Person.personsNb ++;
        if (gender == 'F')
            Person.womenNb ++;
        Person.ageSum += age();
    }
    public int age() { ... }
}
```

# static

Une caractéristique de classe = une **variable** ou une **méthode**

⇒ Méthode déclarée **static** et appelée via le lastname de la classe

## *Exemple*

*Méthode qui calcule moyenne des ages*

```
public static double ageAverage() {  
    if (Person.personsNb != 0)  
        return Person.ageSum / (double) Person.personsNb;  
    else  
        return 0;  
}
```

# static

Comment appeler une méthode de classe ?

On appelle une méthode de classe via le nom de la classe :

**NomClasse**.methodeDeClasse(...)

*Exemple*

```
public static void main(String[ ] args) {  
    Person jules = new Person("Jules", "Petit", 'M', 15, 1, 2000);  
    Person anne = new Person("Anne", "Leroy", 'F', 10, 1, 2010);  
    System.out.println(Person.ageAverage());  
}
```

# Méthode abstraite

## Pas d'implémentation

⇒ Déclaration de méthode sans code

⇒ Méthode déclarée **abstract**

Déclaration correcte d'une méthode abstraite :

**abstract** typeRetour methodeX( ...);

Déclaration incorrecte d'une méthode abstraite :

typeRetour methodeX( ...) {}

# Classe abstraite

Classe qui contient au moins une méthode abstraite

⇒ Classe déclarée **abstract**

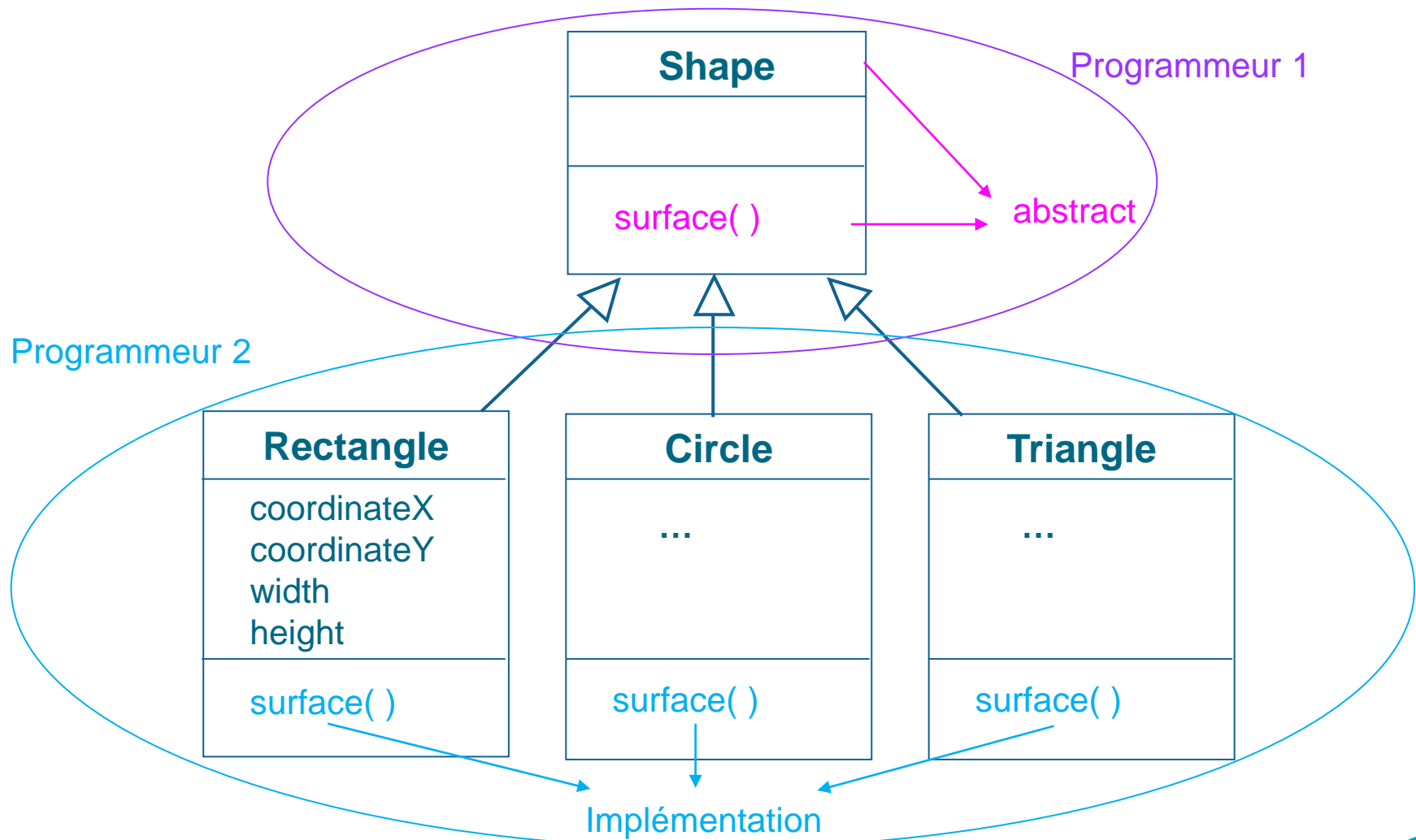
**Ne peut avoir d'occurrences**

⇒ On ne peut pas créer d'objets de cette classe

Intérêt ?

Seulement si on crée des sous-classes qui implémenteront les méthodes abstraites héritées

# Classe abstraite



# Classe abstraite

```
public abstract class Shape {  
    public abstract int surface();  
}
```

```
public class Rectangle extends Shape {  
    private int xCoordinate;  
    private int yCoordinate;  
    private int width;  
    private int height;  
  
    public Rectangle(int xCoordinate, int yCoordinate, int width, int height) {  
        this.xCoordinate = xCoordinate;  
        this.yCoordinate = yCoordinate;  
        this.width = width;  
        this.height = height;  
    }  
  
    public int surface() {  
        return width * height;  
    }  
}
```

# Classe abstraite

## En résumé

- ① Une **méthode sans implémentation**  $\Rightarrow$  abstraite  
`abstract typeRetour methodeX ( ... ) ;`
- ② Une classe qui contient **au moins une méthode abstraite**  $\Rightarrow$  abstraite  
`abstract Class ...`
- ③ On ne peut **pas créer d'occurrence** d'une classe abstraite même si celle-ci contient un constructeur
- ④ On peut créer des **sous-classes d'une classe abstraite**  
Une sous-classe d'une classe abstraite doit, si elle n'est pas elle-même déclarée abstraite, **implémenter toutes les méthodes abstraites** dont elle hérite



# Interface

// Classe **abstraite** : on ne peut pas créer d'objets à partir d'une classe abstraite

- ⇒ Ne peut contenir **ni variables d'instance ni constructeur**
- ⇒ Peut contenir des **constantes** (final static)
- ⇒ **Toutes ses méthodes sont abstraites** : ne contiennent pas de code

**Toutes les méthodes** déclarées implicitement **abstract** et **public**

⇒ abstract public facultatifs

Conclusion

Contient des constantes et des déclarations de méthodes

Déclaration d'une interface

```
interface InterfaceName {           // class ClassName
    ...
}
```

# Interface

Utilité ?

Si des classes s'engagent à redéfinir les méthodes de l'interface

⇒ à leur donner une implémentation

= sorte de **contrat** soumis par l'interface à toute classe qui souhaite l'implémenter

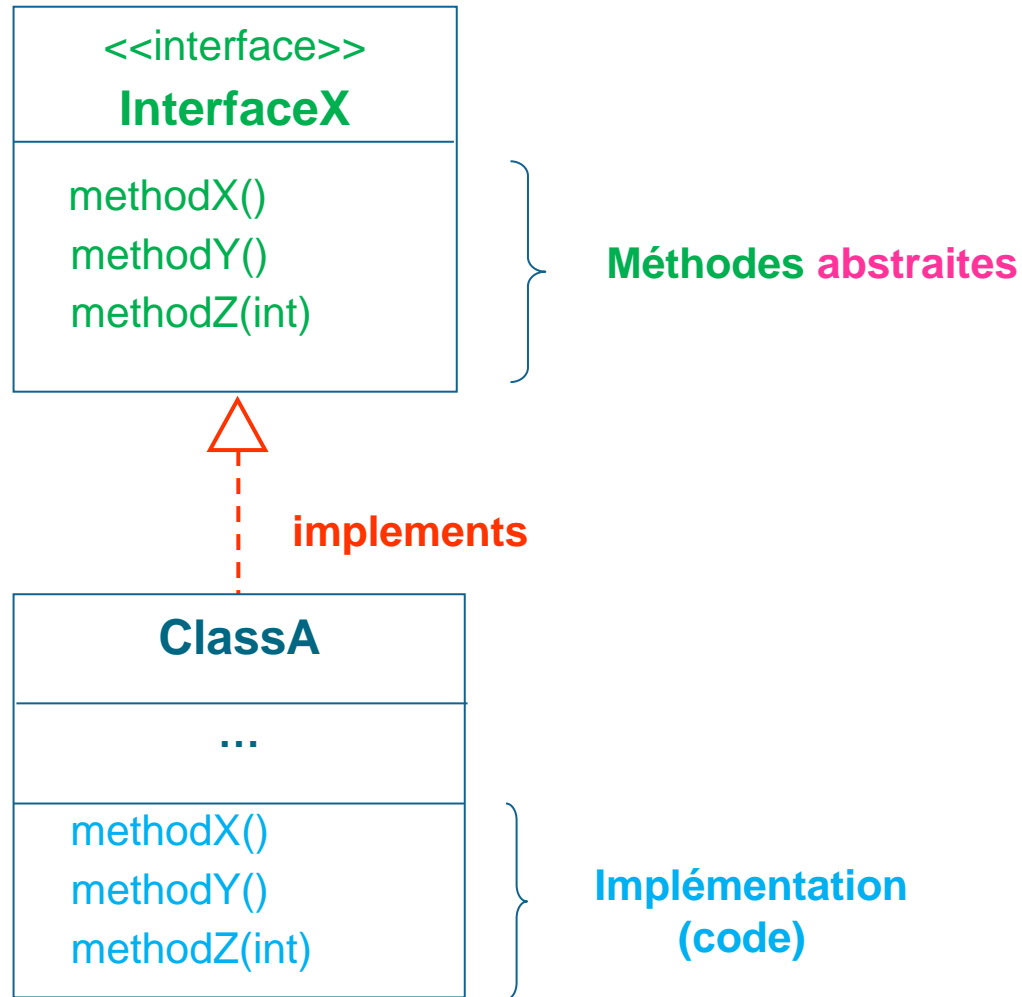
Classe qui s'engage à implémenter toutes les méthodes déclarées dans une interface :

```
class ClassName implements InterfaceName {  
    ...  
}
```

N.B. Si une classe implements une interface sans donner une implémentation pour toutes les méthodes de l'interface

⇒ **classe abstraite**

# Interface



# Interface

```
public interface InterfaceX {  
    void methodX();  
    int methodY();  
    void methodZ(int a);  
}
```

Toutes les méthodes sont implicitement **public** et **abstract**

```
public class ClassA implements InterfaceX {
```

```
    public void methodX() {  
        ...  
    }  
    public int methodY() {  
        ...  
    }  
    public void methodZ(int a) {  
        ...  
    }  
}
```

Code correspondant à l'implémentation  
des méthodes de l'interface

# Interface

```
public interface InterfaceX {  
    void methodX() ;  
    ... // déclarations de méthodes (sans implémentation)  
}  
  
public class ClassA implements InterfaceX {  
    void methodX() { ... // implémentation }  
    ... // redéfinir TOUTES les méthodes de InterfaceX  
}
```

## Utilisation

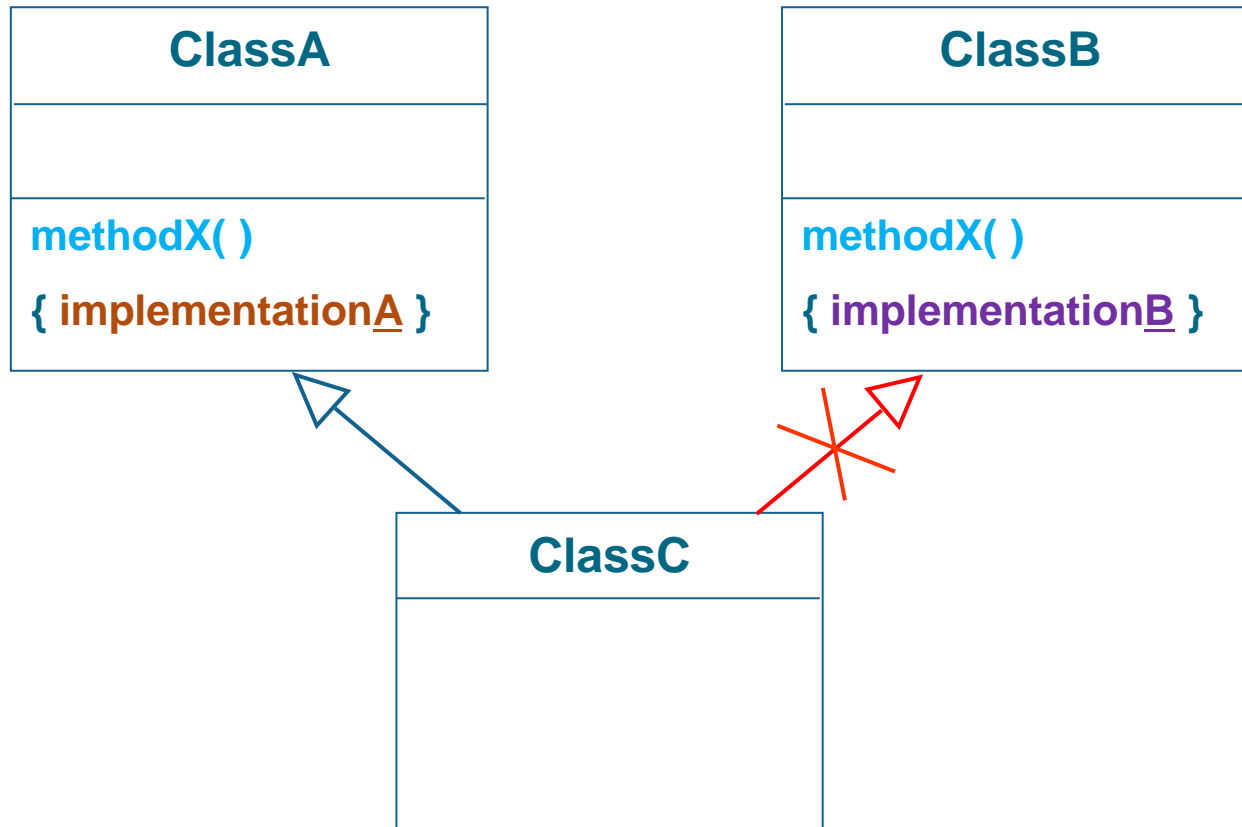
**On peut déclarer une variable de type interface !!!**

Exemple : **InterfaceX** variable = ... ;  
          variable.methodX() ;

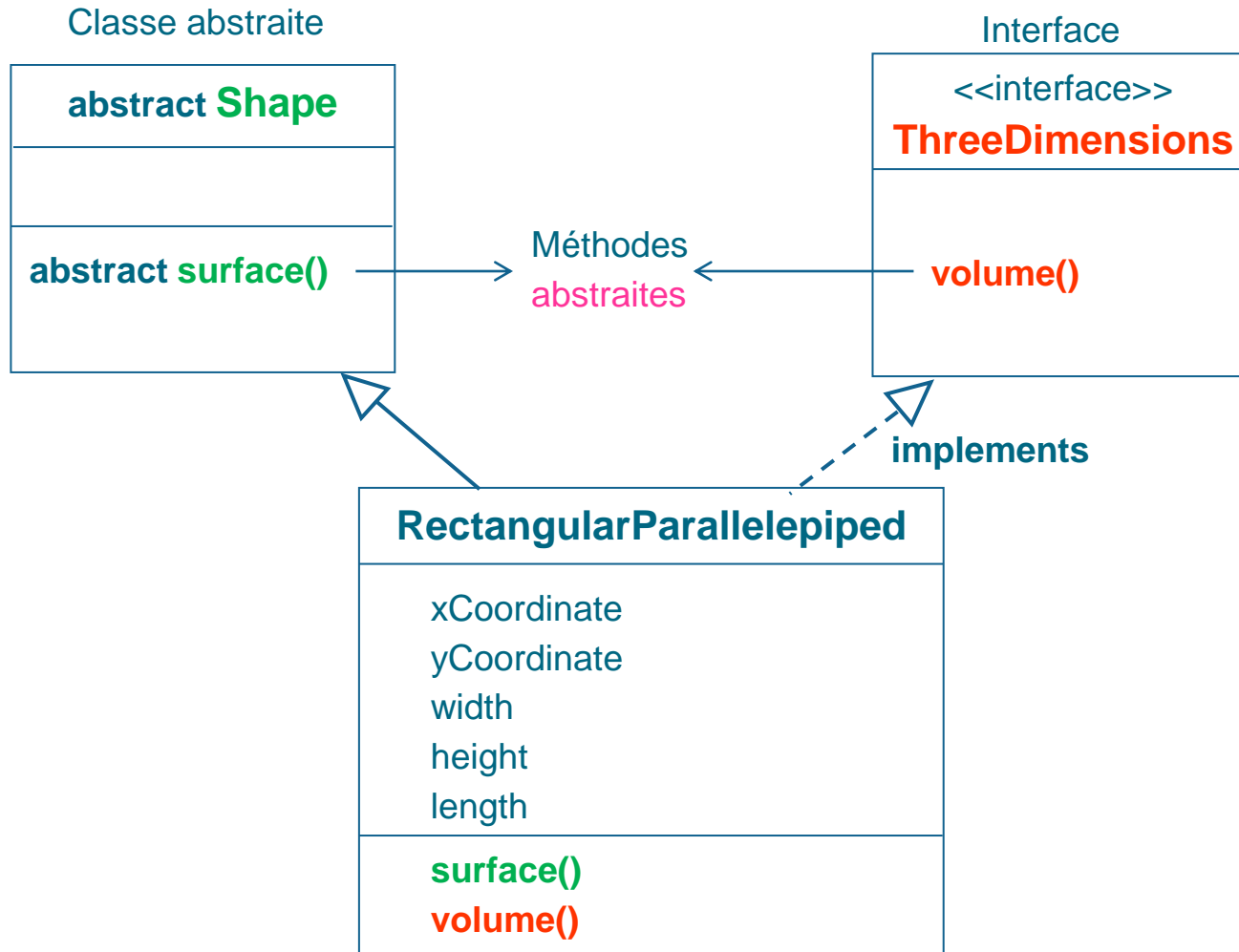
**A condition que *variable* soit instanciée par un objet d'une classe qui implémente InterfaceX, par exemple : *variable* = new ClassA() ;**

# Pas d'héritage multiple en java

Une classe ne peut hériter de plusieurs super-classes à la fois car **risque de conflit** si héritage multiple



# Pas d'héritage multiple en java



```
public abstract class Shape {  
    public abstract int surface();  
}  
  
public interface ThreeDimensions {  
    int volume();  
}  
  
public class ParallelipedeRectangle extends Shape implements ThreeDimensions {  
    private int xCoordinate;  
    private int yCoordinate;  
    private int width;  
    private int height;  
    private int length;  
    ...  
    public int surface() {  
        ...  
    }  
    public int volume() {  
        ...  
    }  
}
```

The diagram illustrates the relationships between the code elements:

- A green arrow points from the `surface()` method in the `Shape` class to the `surface()` method in the `ParallelipedeRectangle` class.
- A red arrow points from the `volume()` method in the `ThreeDimensions` interface to the `volume()` method in the `ParallelipedeRectangle` class.
- A green oval highlights the `extends Shape` keyword.
- A red oval highlights the `implements ThreeDimensions` keyword.



Pas d'héritage multiple en java mais une classe peut implémenter plus d'une interface

```
public interface Shape {  
    int surface();  
}
```

```
public interface ThreeDimensions {  
    int volume();  
}
```

```
public class ParallelipedeRectangle implements Shape, ThreeDimensions {  
    private int xCoordinate;  
    private int yCoordinate;  
    private int width;  
    private int height;  
    private int length;  
  
    ...  
    public int surface() {  
        ...  
    }  
    public int volume() {  
        ...  
    }  
}
```

# Interface

## En résumé

- ① La déclaration d'une interface commence par le mot réservé interface  
Une interface = **constantes** et/ou de **méthodes abstraites**
- ② Méthodes d'une interface : **implicitement public et abstract**
- ③ Une classe qui implémente une interface s'engage à fournir une implémentation pour **toutes** les méthodes de l'interface  
class ClassName **implements** InterfaceName
- ④ Toute classe qui implémente une interface doit implémenter **chacune** des méthodes de l'interface en les déclarant avec la protection **public**
- ⑤ Une classe qui contient une clause implements dans sa déclaration mais n'implémente pas toutes les méthodes reprises dans l'interface ⇒ **abstract**
- ⑥ **Pas d'héritage multiple**  
une classe peut être **sous-classe d'une super-classe** tout en implémentant une ou **plusieurs interface(s)**

# Programmation orientée objet avancée

**1. Prérequis**

**2. Contenu du cours**

# Contenu du cours

- Gestion des exceptions
- Collections génériques d'objets
- Processus parallèles (threads)
- Gestion des événements
- **Architecture des applications**
  - Découpe en couches
- **Accès (en lecture/écriture) à une base de données**
- **Design Patterns**
- Validations des formulaires
- Tests unitaires
- Streams

# Contenu du cours

- 20h de théorie
- 40h de labo
  - Sur IntelliJ

# Programmation orientée objet avancée

- 1. Prérequis**
- 2. Contenu du cours**
- 3. Evaluation**

- Évaluation intégrée de l'UE
  - Projet intégré
    - En Java
    - Architecture en couches
    - Accès à la BD
      - Script SQL de création de la BD du projet
      - Accès en lecture et écriture à la BD en SQL
    - Sécurité
  - Instructions avancées en SQL
    - Jointures
    - Transactions
    - Top N Analyse...
  - Design Patterns

- Examen
  - Programme Java
    - Par 2 étudiants
    - Lien FACULTATIF avec le sujet du cours d'analyse
  - Partie écrite
    - SQL
  - Partie orale
    - Défense du projet
    - Sur n'importe quelle ligne de code



# Evaluation

- Design Patterns
  - Interro dispensatoire pour l'examen

# Programmation orientée objet avancée

- 1. Prérequis**
- 2. Contenu du cours**
- 3. Evaluation**
- 4. Nombre d'arguments variable**

# Nombre d'arguments variable

## Nombre variable d'arguments dans une méthode

### Via utilisation d'ellipsis ...

#### Conditions

- Un seul argument de type ellipsis
- Obligatoirement le dernier de tous les arguments
- Arguments en nombre variable : de type primitif ou référence
- Syntaxe : ***argumentType***... ***argumentName***

Dans le code de la méthode :

Accès aux différents arguments via un tableau dont le nom est ***argumentName***

# Nombre d'arguments variable

## Exemple de méthode avec un nombre variable d'arguments

Arguments: 0, 1 ou plusieurs objets de type Book

```
public class Library {  
    public int countPages (Book... books)  
    {  
        int pagesTotal = 0 ;  
        for (int i = 0 ; i < books.length ; i++)  
            { pagesTotal += books[i].getPagesCount( ) ; }  
        return pagesTotal;  
    }  
}
```

# Nombre d'arguments variable

## Exemple d'appel de méthode avec un nombre variable d'arguments

```
Book book1, book2, book3, book4;
```

```
Library library = new Library ();
```

```
int pagesTotal;
```

```
...
```

```
// Exemples d'appel de la méthode
```

```
pagesTotal = library.countPages() ;
```

```
pagesTotal = library.countPages(book1) ;
```

```
pagesTotal = library.countPages(book1, book2, book3, book4) ;
```

# Nombre d'arguments variable

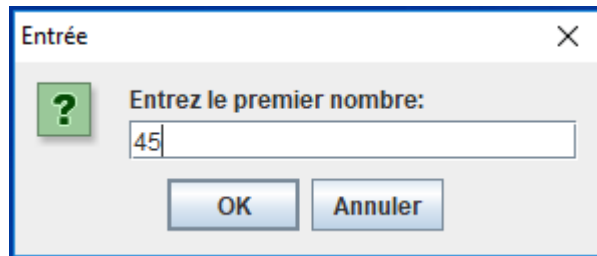
## Autre exemple

```
public static void main(String... args)
```

# Programmation orientée objet avancée

- 1. Prérequis**
- 2. Contenu du cours**
- 3. Evaluation**
- 4. Nombre d'arguments variable**
- 5. JOptionPane**

# JOptionPane



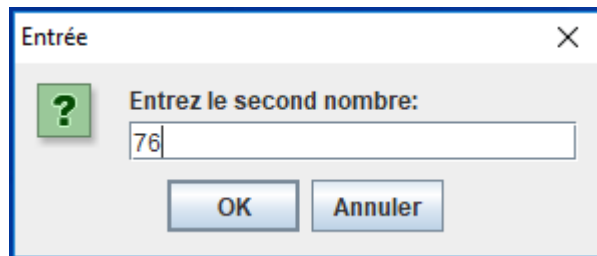
Entrée

?

Entrez le premier nombre:

45

OK Annuler



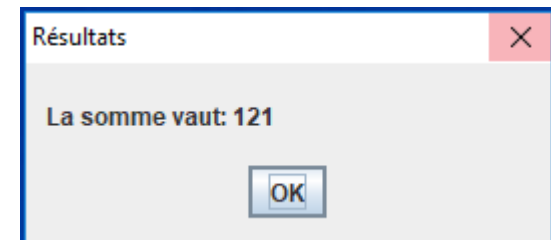
Entrée

?

Entrez le second nombre:

76

OK Annuler



Résultats

La somme vaut: 121

OK



```
import javax.swing.*;
```

```
public class Main {
```

```
    public static void main(String[] args) {
```

```
        int number1 ; int number2 ; int sum ;
```

```
        String first = JOptionPane.showInputDialog ("Entrez le premier nombre :) ;
```

```
        String second = JOptionPane.showInputDialog ("Entrez le second nombre : ") ;
```

```
        number1 = Integer.parseInt(first) ;
```

→ Transforme un String en un entier

```
        number2 = Integer.parseInt(second) ;
```

```
        sum = number1 + number2 ;
```

Contenu de la boîte de dialogue

```
JOptionPane.showMessageDialog (null, "La somme vaut: " + sum,
```

icône

```
"Résultats", JOptionPane.PLAIN_MESSAGE) ;
```

Titre de la boîte de dialogue

```
        System.exit(0) ;
```

```
    }
```

```
}
```

# JOptionPane

JOptionPane.QUESTION\_MESSAGE



JOptionPane.ERROR\_MESSAGE



JOptionPane.INFORMATION\_MESSAGE



JOptionPane.WARNING\_MESSAGE

