

Programmation et Algorithmique II

Ch.5 – Interfaces, Classes abstraites et internes

Bruno Quoitin
(bruno.quoitin@umons.ac.be)

Table des Matières

1. Interfaces

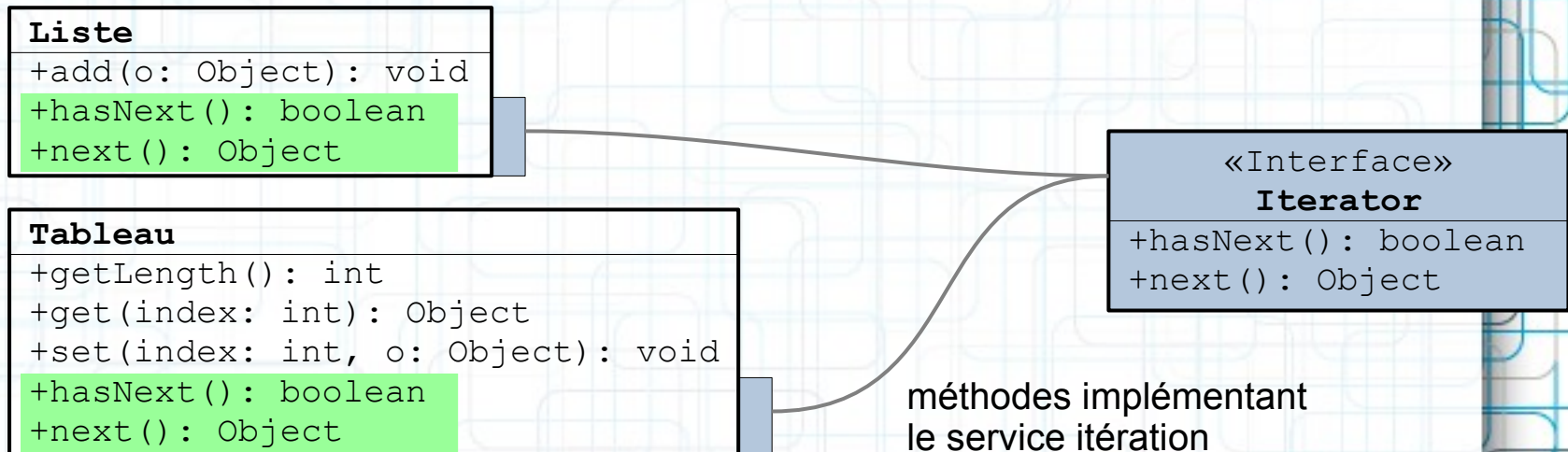
1. Déclaration, Implémentation
2. Références, transtypage
3. Héritage
4. Interfaces fonctionnelles, lambdas
5. Gestion d'événements
6. Interface `Comparable`

2. Classes abstraites

3. Classes internes

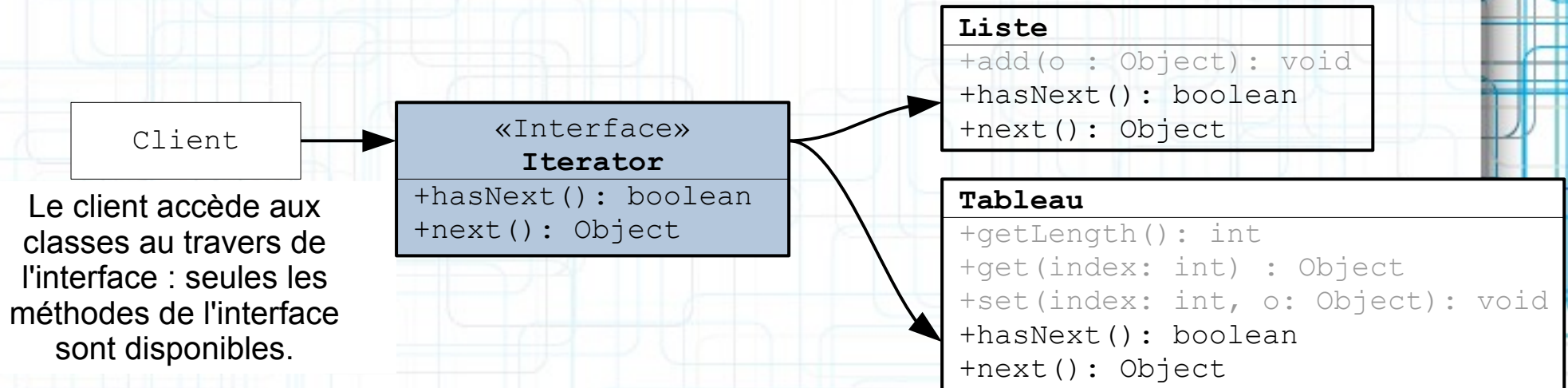
Interfaces

- **Interface = contrat de service**
 - Une **interface** est un moyen de spécifier un service commun à plusieurs classes. Ce service est souvent implémenté sous la forme d'une liste de méthodes.
 - Exemple : supposons que nous disposions de plusieurs façons de stocker une collection d'objets en mémoire (tableau, liste, etc.). Un **service commun** à ces collections consisterait à itérer sur l'ensemble de leurs valeurs.



Interfaces

- **Interface = dénominateur commun**
 - Toutes les classes qui supportent une même interface peuvent être manipulées de la même façon, comme si elles ne supportaient que les méthodes de leur interface commune.



- La relation d'héritage **is-a** peut être étendue aux interfaces. Si la classe A supporte l'interface B, alors A **is-a** B.

Interfaces

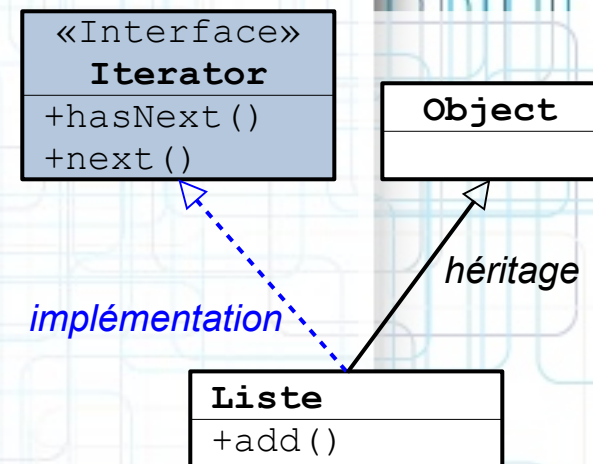
- **Introduction**

- Le mécanisme d'interface en Java comporte deux parties
- **Déclaration** : liste des méthodes et leurs signatures

```
public interface Iterator {  
    boolean hasNext();  
    Object next();  
}
```

- **Implémentation** par une ou plusieurs classes

```
public class Liste implements Iterator {  
    public boolean add(Object o) { /* ... */ }  
    public boolean hasNext() { /* ... */ }  
    public Object next() { /* ... */ }  
}
```



- Une classe a un parent unique, mais potentiellement de multiples interfaces. Les interfaces permettent de s'approcher de l'héritage multiple.

Interfaces

- **Déclaration**

- La déclaration d'une interface comporte les éléments suivants

- un **nom** (identifiant)
- des déclarations de **constantes**
- des **signatures de méthodes**

- Syntaxe

```
public interface nomInterface
{
    corps de l'interface
}
```

- De la même façon qu'une classe, une interface est déclarée dans un fichier `.java` séparé. Ce fichier porte le même nom que celui de l'interface.

Classe

- **Nommage des Interfaces**

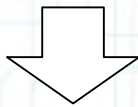
- Un nom d'interface est un identifiant. Il suit la **notation chameau**, en commençant par une majuscule.
- Les noms d'interfaces sont généralement
 - des **adjectifs exprimant une "capacité de"** (suffixe "able").
Par exemple : Cloneable, Serializable, Comparable, Resizable, Readable
 - des **noms de types**. Par exemple : List, Collection, Map

Interfaces

- **Déclaration**

- Modificateurs et spécificateurs d'accès implicites
 - Les signatures de méthodes d'une interface sont automatiquement déclarées avec le spécificateur d'accès **public**.
 - Les « variables » d'une interface sont automatiquement déclarés avec **public static final** (constantes).
- Exemple

```
public interface MonInterface {  
    int NOMBRE_TACHES= 2;  
    void faitCeci();  
}
```



```
public interface MonInterface {  
    public static final int NOMBRE_TACHES= 2;  
    public void faitCeci();  
}
```

Il n'est pas nécessaire d'indiquer les spécificateurs d'accès **public** et les modificateurs **static** et **final**. Ils sont implicites.

Interfaces

- **Implémentation**

- Une classe peut **implémenter une (ou plusieurs) interface(s)**.
- Implémenter une interface consiste à
 1. mentionner le nom de l'interface
 2. fournir l'implémentation des méthodes de l'interface
- Syntaxe

```
public class nomClasse
implements nomInterface1, ..., nomInterfaceN
{
    corps de la classe
}
```

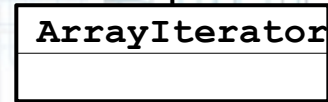
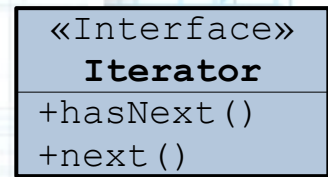
Interfaces

- Implémentation d'une interface

```
public interface Iterator {  
    boolean hasNext();  
    Object next();  
}
```

```
public class ArrayIterator implements Iterator {  
    private int index;  
    private Object[] a;  
    public ArrayIterator(Object[] a) {  
        this.a = a;  
    }  
    public boolean hasNext() {  
        return (index < a.length);  
    }  
    public Object next() {  
        return a[index++];  
    }  
}
```

```
String[] auteurs = { "Horowitz", "Proakis", "Lyons" };  
Iterator iter = new ArrayIterator(auteurs);  
while (iter.hasNext())  
    System.out.println(iter.next());
```



Les implémentations des méthodes doivent être **publiques**.

Table des Matières

1. Interfaces

1. Déclaration, Implémentation
2. **Références, transtypage**
3. Héritage
4. Interfaces fonctionnelles, lambdas
5. Gestion d'événements
6. Interface `Comparable`

2. Classes abstraites

3. Classes internes

Interfaces

- **Références de type interface**

- Créer une instance d'interface n'a pas de sens. Une telle création n'est pas permise par le compilateur.

```
x= new Iterator(); /* Pas permis */
```

- Des variables et paramètres de type interface peuvent être déclarés afin de référencer des instances de classes qui implémentent cette interface.

```
Iterator x; /* référence null */
```

```
Iterator x= new ArrayIterator();
```

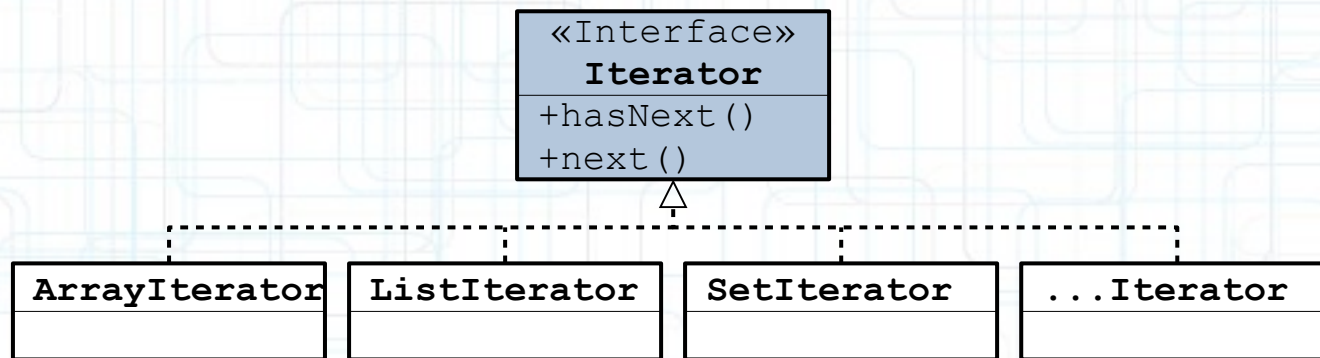
- Le compilateur et la machine virtuelle vérifient la compatibilité des types.

Une référence de type interface **A** ne peut référencer que des instances de classes **B** qui implémentent **A** (on dit aussi que **B is-a A**).

Interfaces

- **Polymorphisme avec les interfaces**

- Manipuler des instances au travers d'une référence "interface" permet de
 - se limiter strictement aux méthodes de cette interface.
 - **remplacer facilement l'implémentation par une autre** (on ne fait pas d'hypothèses sur l'implémentation).



```
Iterator iter= /* whatever iterator */;
while (iter.hasNext())
    System.out.println(iter.next());
```

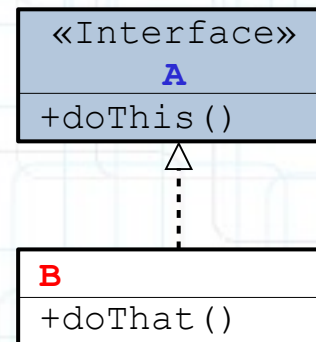
Bonne pratique : utiliser le plus petit dénominateur commun (interface) pour référencer des objets.

Interfaces

- **Opérateur de transtypage**

- L'opérateur de transtypage peut aussi être utilisé avec les types "interfaces". Les contraintes sont similaires à celles vues au Chapitre 4 : le compilateur et la machine virtuelle vérifient la compatibilité de l'instance et du nouveau type.
- Exemple

```
A ref= new B (...);  
ref.doThis();  
ref.doThat();  
((B) ref).doThat();
```



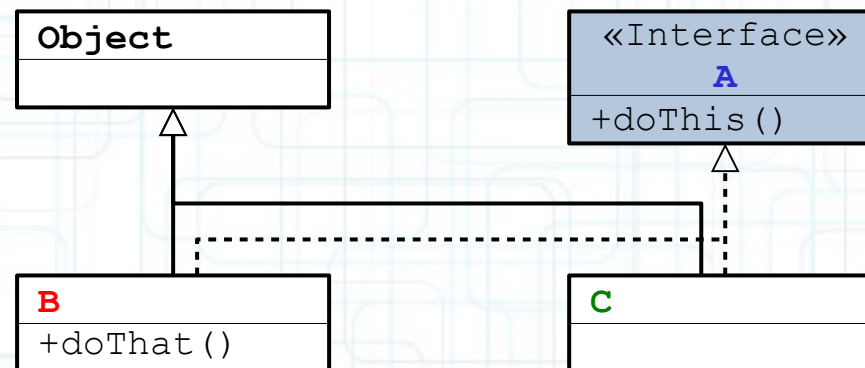
Interfaces

- **Opérateur de transtypage**

- Exemple : transtypage vers le bas invalide (détecté par la machine virtuelle)

```
A ref= new C();  
C c1= (C) ref;  
B c2= (B) ref; // Erreur !!
```

```
bash-3.2$ java MaClasse  
Exception in thread "main" java.lang.ClassCastException:  
MonAutreClasse cannot be cast to MaClasse  
at MaClasse.main(MaClasse.java:15)
```



Interfaces

- **Opérateur instanceof**

- L'opérateur **instanceof** est également utilisable pour tester si une instance implémente une interface donnée.
- Exemple

```
Object obj= new MaClasse();  
  
if (obj instanceof MonInterface) {  
    MonInterface ref= (MonInterface) obj;  
    ref.faitCeci();  
} else  
    System.err.println("L'instance ne supporte pas "+  
                        "l'interface MonInterface !");
```

- Attention, l'utilisation d' **instanceof** pour prendre des décisions n'est généralement pas compatible avec une approche orientée-objet.

Interfaces

- Implémentation de multiples interfaces

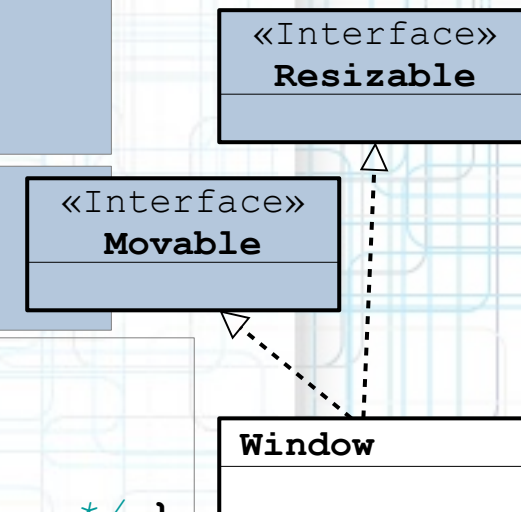
- Exemple

```
public interface Resizable {  
    void resize(double dWidth, double dHeight);  
}
```

```
public interface Movable {  
    void move(double dX, double dY);  
}
```

```
public class Window  
implements Resizable, Movable {  
    void move(double dX, double dY) { /* ... */ }  
    void resize(double dWidth, double dHeight) { /* ... */ }  
}
```

```
Window cls= new Window();  
Movable mv= cls; // Compatible  
Resizable rs= cls; // Compatible  
mv.move(10.0, 7.2);  
rs.resize(0.5, 2.0);  
mv.resize(0.5, 2.0); // méthode non définie  
rs.move(10.0, 7.2); // méthode non définie
```



Interfaces

- **Implémentation de multiples interfaces**
 - L'implémentation de multiples interfaces **peut être impossible** lorsque des interfaces sont incompatibles.
 - Exemple : interfaces qui spécifient la même méthode avec des valeurs de retour différentes : impossible de les implémenter simultanément (violation des règles de la surcharge de méthode).

```
public interface Interface1 {  
    void m();  
}
```

```
public interface Interface2 {  
    int m();  
}
```

```
public class MaClasse  
implements Interface1, Interface2 {  
    public void m() { ... };  
    public int m() { ... };  
}
```

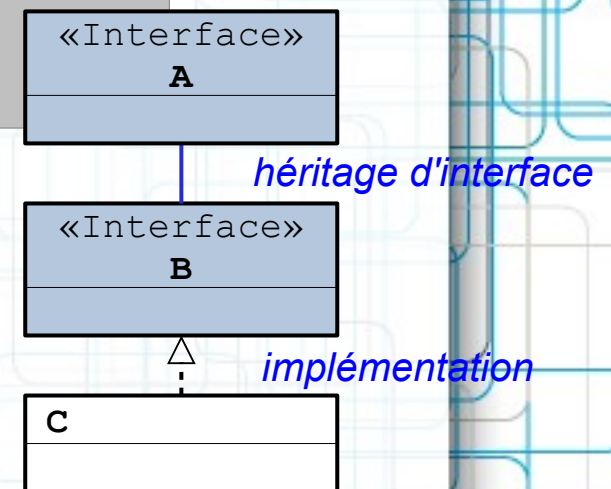
→ Surcharge impossible !

Interfaces

- **Héritage entre interfaces**

- Il est possible de dériver une interface à partir d'une (ou plusieurs) autre(s) interface(s) par héritage. A cet effet, le mot-clé **extends** est utilisé de la même façon que pour l'héritage de classe.
- Syntaxe

```
public interface nomInterface
extends nomInterfaceParent1, ..., nomInterfaceParentN
{
    corps de l'interface
}
```



Interfaces

- **Héritage entre interfaces**

- Exemple

```
public interface MonInterface1 {  
    void m1 ();  
}
```

```
public interface MonInterface2  
extends MonInterface1 {  
    void m2 ();  
}
```

```
public class MaClasse  
implements MonInterface2 {  
    public void m1 () {  
        ...  
    }  
    public void m2 () {  
        ...  
    }  
}
```

L'interface MonInterface2 hérite de la signature de maMethode1

→ une classe qui implémente MonInterface2 doit fournir une implémentation pour maMethode1 et maMethode2

Table des Matières

1. Interfaces

1. Déclaration, Implémentation
2. Références, transtypage
3. Héritage
- 4. Interfaces fonctionnelles, lambda**
5. Gestion d'événements
6. Interface `Comparable`

2. Classes abstraites

3. Classes internes

Interfaces

- **Méthode passée en argument**

- Il n'existe **pas de type méthode ou type fonction** en Java. Il n'est donc pas possible de manipuler des méthodes et de les donner en argument d'autres méthodes.
- Les **interfaces fonctionnelles** permettent d'émuler ce comportement. Une interface fonctionnelle est une interface qui ne définit qu'une seule méthode.
- Exemple

```
public interface MathFunction {  
    double eval(double x);  
}
```

Interfaces

- Interface fonctionnelle

- Exemple

```
public interface MathFunction {  
    double eval(double x);  
}
```

```
public class Sinus implements MathFunction {  
    public double eval(double x) {  
        return Math.sin(x);  
    }  
}
```

```
public class Sqr implements MathFunction {  
    public double eval(double x) {  
        return x * x;  
    }  
}
```

```
MathFunction fct= /* ... */;  
for (int i= borneInf; i < borneSup; i++) {  
    graphics.dessinerPoint(i, fct.eval(i));  
}
```


Interfaces

- **Expressions lambda**

- Java supporte⁽¹⁾ un mécanisme appelé « **lambda expression** » permettant d'exprimer facilement des fonctions anonymes.

```
public double integrate(double a, double b,  
                        MathFunction f) {  
    /* ... */  
}
```

```
double result= integrate(-2, 2, x -> x * x);
```

- Les lambdas sont en fait des "sucres syntaxiques" pour implémenter des interfaces.

```
public interface MathFunction {  
    double eval(double x);  
}
```

⁽¹⁾ Depuis la version 8.

Interfaces

- **Expressions lambda**

- Souvent, une expression lambda est juste un moyen d'appeler une autre méthode.

```
double result= integrate(-2, 2, x -> Math.sin(x));
```

- Dans ce cas, il est plus simple de passer directement la référence⁽¹⁾ vers la méthode en question.

```
double result= integrate(-2, 2, Math::sin);
```

⁽¹⁾ Depuis la version 8.

Interfaces

- Expressions lambda

- A plusieurs arguments

```
(x, y) -> Math.sin(x) * Math.sin(y)
```

- Annotation **@FunctionalInterface**

```
@FunctionalInterface
public interface BiFunction {
    double eval(double x, double y);
}
```

Le compilateur vérifie qu'il s'agit bien d'une interface fonctionnelle (notamment une seule méthode définie).

- Interfaces fonctionnelles fournies par la bibliothèque Java (package `java.util.function`)

```
@FunctionalInterface
public interface Function<R,T> {
    R apply(T t);
}
```


Interfaces

- **Exercice**

- En utilisant les interfaces, implémentez un **algorithme d'intégration numérique** qui approxime l'intégrale d'une fonction $f(x)$ sur un intervalle $[a, b]$ par une somme de surfaces (formule dite « de quadrature »).
 - Différentes méthodes d'approximations peuvent être utilisées telles que la méthode des rectangles, des trapèzes et de Simpson (polynôme de degré 2).
 - Testez votre implémentation sur des fonctions dont la dérivée inverse est connue. Par exemple : x , $\sin(x)$, x^2 , ...
 - Comparez les résultats obtenus avec les différentes méthodes (rectangle, trapèze et Simpson). Faites varier le nombre d'intervalles.
 - Comparez avec le résultat attendu. Par exemple, l'intégrale de $\sin(x)$ sur l'intervalle $[0, \pi/2]$ peut être calculée comme $-\cos(\pi/2) + \cos(0) = 1$.

Table des Matières

1. Interfaces

1. Déclaration, Implémentation
2. Références, transtypage
3. Héritage
4. Interfaces fonctionnelles, lambda
5. **Gestion d'événements**
6. Interface `Comparable`

2. Classes abstraites

3. Classes internes

Interfaces

- **Gestion d'événements**

- Une autre application des interfaces est dans la gestion d'événements, au travers de **fonctions "callback"**.
- Une *callback* est une fonction transmise à un processus et qui est appelée de façon asynchrone lors de la survenance d'un événement.
- Exemples d'événements
 - pression d'un bouton ou d'un clic de souris dans une GUI
 - réception d'un message
 - notification de fin d'un traitement, expiration d'un timer

Interfaces

- Gestion d'événements

```
public interface MouseListener {  
    void mouseClicked(MouseEvent e);  
    void mouseEntered(MouseEvent e);  
    void mouseExited(MouseEvent e);  
    void mousePressed(MouseEvent e);  
    void mouseReleased(MouseEvent e);  
}
```

```
public class MyMouseListener  
implements MouseListener  
{  
    public void mouseClicked(MouseEvent e) {  
        System.out.println("Bouton pressé");  
    }  
    public void mouseEntered(MouseEvent e) { /* ... */ }  
    public void mouseExited(MouseEvent e) { /* ... */ }  
    public void mousePressed(MouseEvent e) { /* ... */ }  
    public void mouseReleased(MouseEvent e) { /* ... */ }  
}
```

```
/* ... */  
panel.addMouseListener(new MyMouseListener());  
/* ... */
```

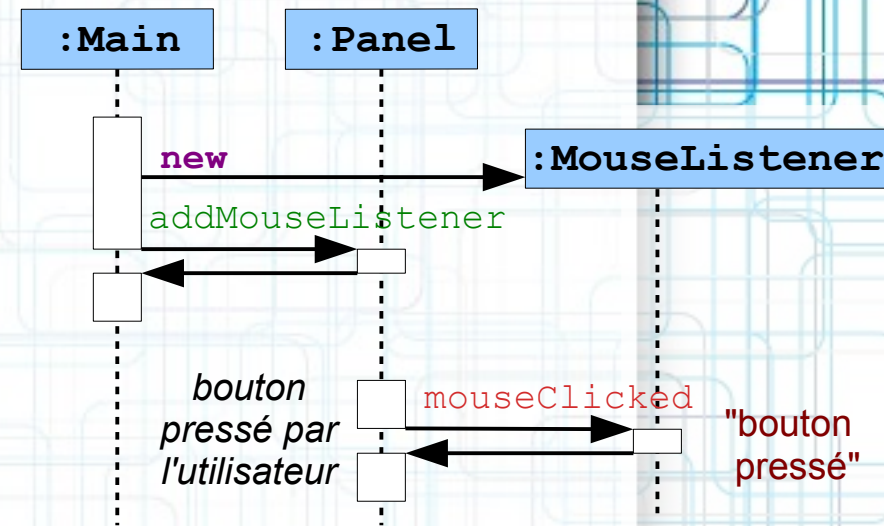


Table des Matières

1. Interfaces

1. Déclaration, Implémentation
2. Références, transtypage
3. Héritage
4. Interfaces fonctionnelles, lambdas
5. Gestion d'événements
6. **Interface Comparable**

2. Classes abstraites

3. Classes internes

Interface Comparable

- **Contexte**

- Pour permettre de **trier le contenu d'un tableau**, la bibliothèque Java fournit les méthodes de classe `sort` dans la classe `java.util.Arrays`.

```
public static void Arrays.sort(Object[] a);  
public static void Arrays.sort(int[] a);  
public static void Arrays.sort(double[] a);  
/* ... */
```

- Exemple 1

```
String[] prenom = { "Nestor", "Archibald", "Hugolin", "Ernest" };  
Arrays.sort(prenom);
```

- Exemple 2

```
Carre[] carres = new Carre[5];  
for (int i = 0; i < carres.length; i++)  
    carres[i] = new Carre(Math.random(), Math.random());  
Arrays.sort(carres);
```


Interface Comparable

- **Contexte**

- Pour trier les éléments d'un tableau, il faut qu'une **relation d'ordre total** soit définie sur ces éléments.
- Etant donné le caractère polymorphique de la référence `Object []`, les éléments du tableau peuvent être des carrés, des `String`, des nombres, ...
- **Comment `sort()` compare-t-il deux éléments entre eux ?**
- En utilisant une interface pardi !
 - `Comparable` : ordre naturel d'un objet, utilisé avec `sort(Object [])`
 - `Comparator` : ordre additionnel ou personnalisé, utilisé avec `sort(Object [], Comparator)`

Interface Comparable

- Introduction

- La bibliothèque Java définit l'interface **Comparable** afin de définir la relation d'ordre naturelle d'une classe.
- Comparable définit une méthode unique permettant de comparer un objet (*this*) à un autre (*other*).

```
public interface Comparable {  
    public int compareTo(Object other);  
}
```

- `x.compareTo(y)` retourne les valeurs suivantes

0	si	<code>x = y</code>
< 0	si	<code>x < y</code>
> 0	si	<code>x > y</code>

Interface Comparable

- **Limitation des interfaces**

- Attention : la définition de l'interface Comparable n'impose que l'existence de la méthode `compareTo` ainsi que sa signature.
- Le comportement de la méthode (valeurs de retour) ne peut pas être spécifié au travers des interfaces ! Il doit donc être spécifié par un autre moyen. Ici il s'agit de la **documentation** de la méthode `compareTo`.

compareTo

```
public int compareTo(Object o)
```

Compares this object with the specified object for order. Returns a **negative integer**, **zero**, or a **positive integer** as this object is **less than**, **equal to**, or **greater than** the specified object.
(...)

Interface Comparable

- **Relation d'ordre total**

- Soit R une relation binaire
- Une relation d'ordre satisfait les propriétés suivantes

- Anti-symétrie

$$\forall x, \forall y, (x R y) \text{ et } (y R x) \Leftrightarrow x = y$$

- Transitivité

$$\forall x, \forall y, \forall z, (x R y) \text{ et } (y R z) \Rightarrow x R z$$

- Réflexivité

$$\forall x, x R x$$

- Relation d'ordre total (tous les éléments sont comparables)

$$\forall x, \forall y, (x R y) \text{ ou } (y R x)$$

Interface Comparable

- **Cohérence avec equals**
 - Il est recommandé, mais pas obligatoire, que l'implémentation de l'interface `Comparable` soit cohérente avec l'implémentation de la méthode `equals` (héritée de la classe `Object` et éventuellement surchargée).
 - si `x.compareTo(y) == 0`,
alors `x.equals(y) == true`
 - et vice versa.

Interface Comparable

- **Exemple**

- Notre objectif est de définir une relation d'ordre total sur les carrés : un **ordre lexicographique** sur leurs coordonnées.
- Les coordonnées sont des couples de **double**, maintenus dans une référence vers une instance de `Position`. L'ensemble des **double** est déjà complètement ordonné avec l'opérateur de comparaison `<`
- Définissons l'ordre lexicographique suivant sur les positions (x, y) :
 - $(x1, y1) < (x2, y2) \Leftrightarrow (x1 < x2) \text{ ou } ((x1 = x2) \text{ et } (y1 < y2))$
 - $(x1, y1) = (x2, y2) \Leftrightarrow (x1 = x2) \text{ et } (y1 = y2)$
- Cette relation définit un ordre total car la relation `<` sur les réels est un ordre total (l'est-il sur les **doubles** ?).

Interface Comparable

- Exemple

```
public class Carre
implements Comparable {
    /* ... */
    public int compareTo(Object other) {
        Carre otherC= (Carre) other;
        if (otherC.pos.x < pos.x)
            return 1;
        if (otherC.pos.x > pos.x)
            return -1;
        if (otherC.pos.y < pos.y)
            return 1;
        if (otherC.pos.y > pos.y)
            return -1;
        return 0;
    }
    public boolean equals(Object other) {
        Carre otherC= (Carre) other;
        return (pos.x == otherC.pos.x) &&
            (pos.y == otherC.pos.y);
    }
}
```

Le transtypage échouera si l'instance de Carre est comparé à un objet qui n'est pas un Carre.

$(x1 > x2) \rightarrow (pos1 > pos2)$ [1]

$(x1 < x2) \rightarrow (pos1 < pos2)$ [-1]

$(x1 = x2)$ et $(y1 > y2) \rightarrow (pos1 > pos2)$ [1]

$(x1 = x2)$ et $(y1 < y2) \rightarrow (pos1 < pos2)$ [-1]

$(x1 = x2)$ et $(y1 = y2) \rightarrow (pos1 = pos2)$ [0]

Interface Comparable

- Exemple

```
Random r= new Random();
Carre[] carres= new Carre[10];
for (int i= 0; i < carres.length; i++)
    carres[i]= new Carre(r.nextInt(10), r.nextInt(10));

Arrays.sort(carres);

for (int i= 0; i < carres.length; i++)
    System.out.println(carres[i]);
```

Des coordonnées entières sont générées aléatoirement de façon à augmenter la probabilité d'obtenir des abscisses égales.

```
bash-3.2$ java ArraySort
Carre@x=1.0;y=9.0
Carre@x=2.0;y=6.0
Carre@x=3.0;y=3.0
Carre@x=3.0;y=5.0
Carre@x=3.0;y=7.0
Carre@x=6.0;y=3.0
Carre@x=7.0;y=4.0
Carre@x=8.0;y=8.0
Carre@x=9.0;y=2.0
Carre@x=9.0;y=4.0
```

En cas d'égalité des abscisses, l'ordre lexicographique considère les ordonnées.

Interface Comparable

- **Interface générique**

- L'interface Comparable est une interface **générique**. Le type **T** des objets qu'il est possible de comparer peut être spécifié à l'interface.

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

- Exemple

```
public class Carre  
implements Comparable<Carre>  
...  
    public int compareTo(Carre other) {  
        /* ... */  
    }  
}
```

- **Note** : nous traiterons les détails des classes, méthodes et interfaces génériques dans un chapitre séparé.

Interfaces

- **Comparaison avec l'héritage**

- La définition d'interfaces semble déjà être faisable grâce à l'héritage. Quelles sont les différences ?
 1. Deux classes qui supportent une même interface ne doivent pas nécessairement avoir un ancêtre commun (autre qu'`Object`).
 2. Une classe définissant une liste de méthodes (à la manière d'une interface) nécessite de fournir l'implémentation de ces méthodes. Cela n'a pas toujours du sens. (voir les classes abstraites)
 3. Le support de plusieurs interfaces est possible alors que l'héritage de plusieurs classes ne l'est pas.
 4. Une classe peut définir des variables, des méthodes non publiques, etc alors qu'une interface ne le peut pas.

Table des Matières

1. Interfaces

2. Classes abstraites

1. Application : `Timer` et `TimerTask`

3. Classes internes

1. Classes internes

2. Classes locales

3. Classes anonymes

Classes Abstraites

- **Introduction**

- Une **classe abstraite** est une classe qui ne définit pas l'implémentation de toutes les méthodes qu'elle déclare. Par opposition, une **classe concrète** implémente toutes les méthodes qu'elle déclare.
- Une classe abstraite ne peut pas être instanciée.
- Une classe abstraite est utilisée comme classe parent d'autres classes.
- Cas particulier : une classe abstraite peut implémenter partiellement une interface.

Classes Abstraites

- **Déclaration**

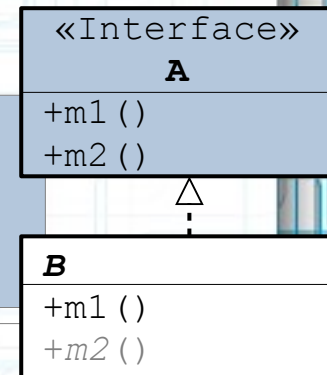
- Une classe est déclarée abstraite en utilisant le mot-clé **abstract** dans sa déclaration

```
public abstract class nomClasse
{
    corps de la classe
}
```

- Exemple

```
public interface A {
    public void m1 ();
    public void m2 ();
}
```

```
public abstract class B implements A {
    public void m1 () {
        System.out.println("Youhou, je suis implémentée");
    }
}
```



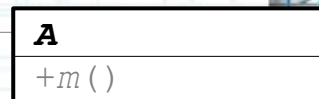
A définit deux méthodes.
La classe abstraite B
n'en définit qu'une.

Classes Abstraites

- **Méthodes Abstraites**

- Une classe abstraite peut définir des méthodes qu'elle n'implémente pas. Ces méthodes doivent elles aussi être déclarées *abstraites*.
- Le modificateur **abstract** est utilisé dans la déclaration d'une méthode pour indiquer qu'elle est abstraite.
- Exemple

```
public abstract class A
{
    public abstract void m();
}
```



- Toute classe qui définit au moins une méthode abstraite doit être rendue abstraite.

Classes Abstraites

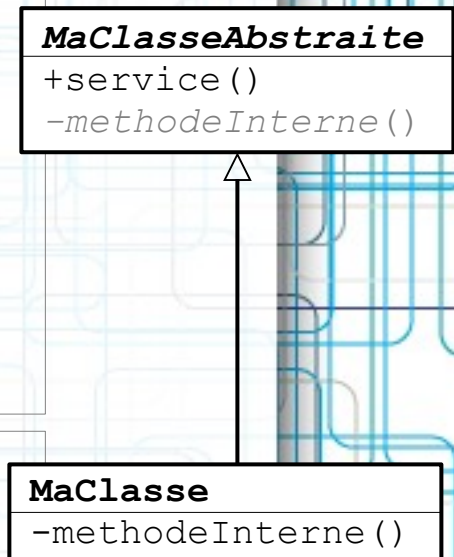
- **Classes Abstraites vs Interfaces**
 - Quelle sont les différences entre interfaces et classes abstraites ?
 - Il n'est pas possible d'hériter de plusieurs classes abstraites car l'héritage multiple de classes n'est pas permis en Java. C'est possible pour les interfaces.
 - Les classes abstraites peuvent définir des variables. Les interfaces ne peuvent définir que des constantes (les modificateurs **public static final** y sont implicites).
 - Les classes abstraites peuvent aussi définir des méthodes « concrètes ». Les interfaces ne définissent que des signatures de méthodes.
 - Les classes abstraites peuvent déclarer des méthodes abstraites non publiques. Les interfaces ne définissent que des signatures de méthodes publiques (**public** est implicite).

Classes Abstraites

- **Classes Abstraites vs Interfaces**
 - Exemple : une interface ne permet pas de déclarer une méthode protégée alors qu'une classe abstraite le peut.

```
public abstract class MaClasseAbstraite
{
    protected abstract void methodeInterne();
    public void service() {
        ...
        methodeInterne();
        ...
    }
}
```

```
public class MaClasse
extends MaClasseAbstraite
{
    protected void methodeInterne() {
        ...
    }
}
```



Classes Abstraites

- **Classes Abstraites vs Interfaces**
 - Les classes abstraites et les interfaces sont parfois utilisées en tandem, selon l'organisation suivante:
 - L'**interface** spécifie les méthodes à supporter
 - La classe abstraite implémente partiellement les comportements communs aux sous-classes
 - Les sous-classes implémentent complètement l'interface

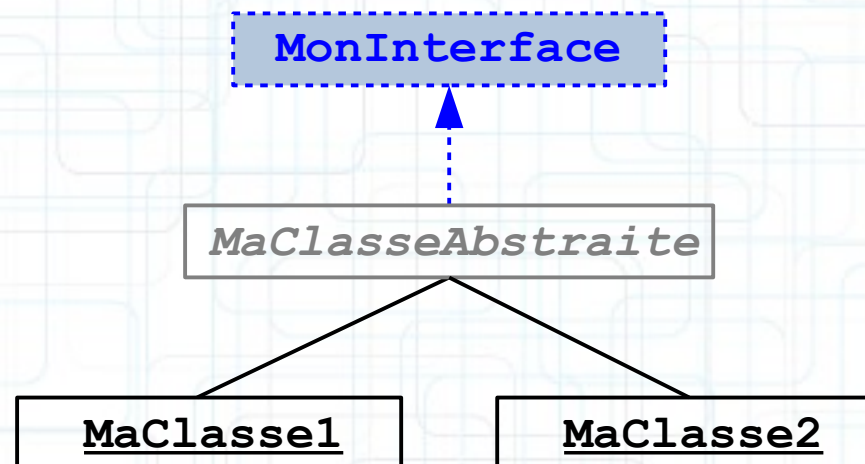


Table des Matières

1. Interfaces
2. Classes abstraites
 1. **Application : Timer et TimerTask**
3. Classes internes
 1. Classes internes
 2. Classes locales
 3. Classes anonymes

Classes `Timer` et `TimerTask`

- **Introduction**

- Un ***timer*** permet de déclencher l'exécution d'une action après l'écoulement d'un certain temps ou à intervalle régulier.
- Il existe plusieurs types de *timers* dans la bibliothèque Java. Un exemple est la classe `java.util.Timer` . qui repose sur l'utilisation de classes abstraites.
- **Note** : La bibliothèque Java fournit également une gestion de timer dans la classe `javax.swing.Timer`. Cet autre timer repose sur l'utilisation d'interfaces et est plus approprié dans un contexte d'interface graphique.

Classes `Timer` et `TimerTask`

- Classe `java.util.Timer`

- La classe `Timer` fournit notamment les deux méthodes montrées ci-dessous.

- permet d'exécuter une action après un temps donné.

```
public void schedule(TimerTask t, long delay);
```

- permet en plus de répéter l'action à intervalle régulier.

```
public void schedule(TimerTask t, long delay, long period);
```

- Les tâches à exécuter par la classe `Timer` sont définies sous la forme d'instances d'une sous-classe de la classe abstraite `TimerTask`.

Note: pour utiliser la classe `TimerTask`, il est nécessaire de l'importer à partir du *package* `java.util` en utilisant la clause `import` en début de fichier (en dehors du corps de la classe).

Classes `Timer` et `TimerTask`

- **Classe `TimerTask`**

`TimerTask`

```
+run()  
+cancel()  
+scheduledExecutionTime()
```

- La classe abstraite `TimerTask` définit plusieurs méthodes.

- action programmée, définie sous forme d'une méthode abstraite qu'il faut surcharger:

```
public abstract void run();
```

- annuler les exécutions suivantes de la tâche:

```
public void cancel();
```

- récupérer le temps de la prochaine exécution. Ce temps est mesuré en nombre de millisecondes écoulées depuis le 1er janvier 1970 minuit UTC.

```
public long scheduledExecutionTime();
```

Note: pour utiliser la classe `TimerTask`, il est nécessaire de l'importer à partir du *package* `java.util` en utilisant la clause `import` en début de fichier (en dehors du corps de la classe).

Classes Timer et TimerTask

- Exemple

Importe les classes TimerTask et Timer à partir du package java.util

```
import java.util.TimerTask;

public class MyTimerTask extends TimerTask {
    public void run() {
        System.out.println("action exécutée");
    }
}
```

```
import java.util.Timer;
/* ... */
Timer t= new Timer();
TimerTask task= new MyTimerTask();
System.out.println(System.currentTimeMillis());
t.schedule(task, 1000);
System.out.println(task.scheduledExecutionTime());
```

TimerTask
+run()
+cancel()
(...)

MyTimerTask
+run()

Récupère le temps actuel
(ms à partir du 1/1/1970
minuit UTC)

Programme la tâche
dans 1000 ms

```
t=0 1268056528480
     1268056529480
t=1000ms action exécutée
```

Classes Timer et TimerTask

- Autre exemple

```
import java.util.TimerTask;

public class MyTimerTask2 extends TimerTask {
    public void run() {
        System.out.println("action exécutée "+
            System.currentTimeMillis());
    }
}
```

```
import java.util.Timer;
/* ... */
Timer t= new Timer();
t.schedule(new MyTimerTask2(), 0, 5000);
```

Programme la tâche
immédiatement pour une
exécution toutes les
5000 ms

```
action exécutée 1268058555198
action exécutée 1268058560198
action exécutée 1268058565198
action exécutée 1268058570198
action exécutée 1268058575198
action exécutée 1268058580198
...
```

Table des Matières

1. Interfaces

2. Classes abstraites

1. Application : `Timer` et `TimerTask`

3. Classes internes

1. Classes internes

2. Classes locales

3. Classes anonymes

Classes internes

- **Introduction**

- Une **classe interne** (*inner class*) est une classe définie à l'intérieur d'une autre classe.
- On distingue
 - **classe interne** (*inner class*): définie dans le corps d'une autre classe
 - **classe imbriquée** (*nested class*) : définie avec **static** dans le corps d'une autre classe
 - **classe locale** : définie dans une méthode
 - **classe anonyme** : classe locale sans nom

Classes internes

- **Introduction**

- Pourquoi s'intéresser aux classes internes ?
 - Syntaxe de déclaration et propriétés légèrement différentes de celles des classes que nous avons définies jusqu'ici.
- Les classes internes offrent les **avantages suivants**
 - L'utilisation de classes internes permet de renforcer l'encapsulation en « cachant » l'implémentation d'une classe de l'extérieur (notamment aux classes du même package).
 - Elles peuvent être définies de manière concise (voir classes anonymes)
 - Elles peuvent avoir accès à certaines données (arguments de méthodes et données privées).

Table des Matières

1. Interfaces

2. Classes abstraites

1. Application : `Timer` et `TimerTask`

3. Classes internes

1. **Classes internes**

2. Classes locales

3. Classes anonymes

Classes internes

- Dans une classe
 - La déclaration d'une classe interne **dans le corps d'une classe** est effectuée selon la syntaxe suivante

```
public class nomClasseExterne
{
    méthodes
    champs

    specificateurAcces modificateurs class nomClasseInterne
    {
        méthodes
        champs
    }
}
```

- Note : l'ordre de déclaration des méthodes/champs/classe interne utilisé ci-dessus n'a pas d'importance.

Classes internes

- **Classe interne d'instance**

- Exemple

```
public class OuterClass
{
    protected int x= 3;
    public class InnerClass
    {
        public void plip() {
            System.out.println(x);
        }
    }
}
```

Accès à un membre d'instance de la classe englobante (réf. **this**)

```
OuterClass oc= new OuterClass();
OuterClass.InnerClass ic= oc.new InnerClass();
ic.plip();
```

Le sens des spécificateurs d'accès et modificateurs vu précédemment est inchangé. La classe interne définie ici est publique et d'instance (pas de mot clé **static**). Il faut donc disposer d'une instance de la classe externe pour accéder à la classe interne.

Classes internes

- **Classe interne de classe (imbriquée)**

- Le modificateur **static** permet de donner accès à une classe interne sans disposer d'une instance de la classe externe dans laquelle elle est définie. On parle alors de **classe imbriquée** (*nested class*).
- Exemple

```
public class OuterClass
{
    public static class InnerClass
    {
    }
}
```

```
OuterClass.InnerClass c= new OuterClass.InnerClass ();
```

Cette fois, la classe interne est définie ici publique et de classe (mot clé **static**).
Il est possible d'y accéder directement avec la classe externe.

Classes internes

- **Poupées russes**

- Il est possible de définir une classe interne à l'intérieur d'une classe interne, i.e. des classes internes imbriquées.
- Exemple

```
public class OuterClass
{
    public class InnerClass
    {
        public class InnerInnerClass
        {
            public class InnerInnerInnerClass
            {
            }
        }
    }
}
```

-----> Classe la plus externe
(top-level class)



- Note : il s'agit d'une possibilité, pas nécessairement d'une recommandation.

Classes internes

- **Spécification d'accès**

- Le spécificateur d'accès **private** a une signification particulière dans une classe interne.
- La spécification Java (*Java Language Specification*, JLS) indique en sa section 6.6.1 *Determining Accessibility*
 - « *Otherwise, if the member or constructor is declared **private**, then access is permitted if and only if it occurs within the body of the top level class (§7.6) that encloses the declaration of the member or constructor.* »
- Ce qui signifie que **les membres privés d'une classe interne peuvent être accédés de l'intérieur de la classe la plus externe (top-level class)**. S'il y a plusieurs niveaux de classes externes, toutes pourront accéder au membre privé.

Table des Matières

1. Interfaces

2. Classes abstraites

1. Application : `Timer` et `TimerTask`

3. Classes internes

1. Classes internes

2. **Classes locales**

3. Classes anonymes

Classes internes

- **Dans une méthode = class locale**
 - Il est possible de déclarer une classe interne **à l'intérieur d'une méthode** (**classe locale**). La déclaration d'une telle classe ne contient pas de spécificateur d'accès. Sa portée est limitée à la méthode qui la contient.
 - Syntaxe

```
public class nomClasseExterne
{
    signatureMethode( ...) {

        modificateurs class nomClasseInterne
        {
            méthodes
            champs
        }
    }
}
```

Classes internes

- **Classe locale**

- Exemple : déclaration de 2 classes internes à une méthode.

```
public class OuterClass
{
    public void method()
    {
        class InnerClass1
        {
        }
        class InnerClass2
        {
        }
        InnerClass1 ic1= new InnerClass1 ();
        InnerClass2 ic2= new InnerClass2 ();
    }
}
```

Classes internes

- **Classe locale**

- Les classes définies dans une méthode ont une portée limitée à cette méthode. Elles ne sont donc **pas visibles en dehors de cette méthode**.

```
public class OuterClass
{
    public void method1 ()
    {
        class InnerClass
        {
        }
        InnerClass ic= new InnerClass ();
    }
    public void methode2 ()
    {
        InnerClass ic= new InnerClass (); /* Pas possible */
    }
}
```

→ Symboles non définis
à ce niveau !

Classes internes

- **Corollaire**

- Par conséquent, il n'est pas possible d'utiliser comme type de retour d'une méthode **M** une classe **A** définie à l'intérieur de cette même méthode !
- En effet, le type **A** est inconnu à l'extérieur de la méthode **M**.
- Exemple

```
public class OuterClass
{
    public InnerClass method()
    {
        class InnerClass
        {
        }
        return new InnerClass();
    }
}
```

→ Symbole InnerClass
non défini à ce niveau !

Classes internes

- **Solution: interface ou héritage**
 - Comment **retourner une instance d'une classe interne définie dans une méthode** ?
 - Il faut utiliser un type de référence visible de l'extérieur de la méthode. Une solution souvent utilisée est d'imposer à la classe interne
 - soit l'implémentation d'une interface
 - soit l'héritage d'une super classe.
 - La méthode peut alors retourner la référence sous le type de l'interface supportée ou de la super classe.

Classes internes

- **Solution : interface ou héritage**

- Exemple : implémentation d'une interface.

```
public interface UneInterface { /* ... */ }

public class OuterClass
{
    public UneInterface method()
    {
        class InnerClass implements UneInterface
        {
        }
        return new InnerClass();
    }
}
```

Le type de retour de la méthode est une **interface**.
La classe interne supporte cette **interface**.
→ il est possible de retourner une référence vers un instance de la classe interne à la méthode.

Classes internes

- Exemple d'utilisation
 - Revisitons notre exemple de Timer...

```
import java.util.*;

public class OuterClass
{
    private static class MyTimerTask extends TimerTask {
        public void run() {
            System.out.println("action exécutée");
        }
    }

    public static void main(String[] args) {
        Timer t= new Timer();
        TimerTask task= new MyTimerTask();
        System.out.println(System.currentTimeMillis());
        t.schedule(task, 1000);
        System.out.println(task.scheduledExecutionTime());
    }
}
```

MyTimerTask est définie
comme classe interne.

Table des Matières

1. Interfaces

2. Classes abstraites

1. Application : `Timer` et `TimerTask`

3. Classes internes

1. Classes internes

2. Classes locales

3. Classes anonymes

Classes internes

- **Classes anonymes**

- Il est également possible de définir des **classes internes anonymes**. Les classes internes anonymes combinent une déclaration de classe à une instantiation. Les classes anonymes ne portent pas de nom.

- Syntaxe

```
new nomClasseParent ( parametres ) {  
    méthodes  
    champs  
}
```

- Les restrictions principales des classes anonymes sont
 - Le constructeur ne peut être surchargé. Le constructeur utilisé est celui de la classe parent.
 - Une classe anonyme a une instance unique.

Classes internes

- **Classes anonymes**

- Il est également possible de créer une classe anonyme sur base d'une interface. Dans ce cas, le constructeur utilisé est le constructeur par défaut. Il ne prend pas de paramètre.
- La syntaxe est alors la suivante

```
new nomInterface ( ) {  
    méthodes  
    champs  
};
```

Classes internes

- **Classes anonymes**

- Exemple : utilisation du `Timer` revisitée (une fois de plus).

```
import java.util.*;

public class OuterClass
{
    public static void main(String[] args) {
        Timer t= new Timer();
        TimerTask task=
            new TimerTask() {
                public void run() {
                    System.out.println("action exécutée");
                }
            };
        System.out.println(System.currentTimeMillis());
        t.schedule(task, 1000);
        System.out.println(task.scheduledExecutionTime());
    }
}
```

L'instance de `TimerTask` est définie comme classe anonyme.

Classes internes

- **Classes anonymes**

- Quand faut-il utiliser des classes anonymes ?

- Quand le code de la classe ainsi définie est relativement **court** (typiquement une 10^{aine} de lignes).
 - Quand ce code doit être utilisé à **un seul endroit**. Sinon, il vaut mieux définir une classe interne avec un nom afin d'éviter la duplication de code.

- De manière générale, un code source contenant de nombreuses classes anonymes peut vite devenir illisible
→ à utiliser parcimonieusement !!

Table des Matières

1. Interfaces

2. Classes abstraites

1. Application : `Timer` et `TimerTask`

3. Classes internes

1. Classes internes

2. Classes locales

3. Classes anonymes

4. Annexe

Classes internes

- **Restrictions d'accès**

- Les classes internes peuvent avoir accès aux membres (attributs et méthodes) définis dans les classes englobantes, voir même aux variables et arguments de méthodes dans le cas des classes internes à une méthode (*local classes*).
- **Mais** certaines restrictions s'appliquent à ces accès.

Classes internes

- Restrictions d'accès

- Une classe interne a accès à tout les membres de la ou des classe(s) englobantes.
 - Les restrictions habituelles s'appliquent : **une classe interne de classe ne peut avoir accès qu'aux membres de classe**

```
public class OuterClass
{
    public int membre1;
    public static int membre2;
    public void methode() { }

    public class InnerClass {
        public InnerClass() {
            membre1= 17;
            membre2= 23;
            methode();
        }
    }
}
```

```
public class OuterClass
{
    public int membre1;
    public static int membre2;
    public void methode() { }

    public static class InnerClass {
        public InnerClass() {
membre1= 17;
            membre2= 23;
methode();
        }
    }
}
```

La classe interne de classe (*nested class*) ne peut accéder aux membres d'instance.

Classes internes

- Restrictions d'accès

- Une classe interne d'instance (sans le modificateur **static**) **ne peut pas définir de membres de classe** (avec le modificateur **static**).
 - Exception : membres constants (**static final**).


```
public class OuterClass {  
  
    public class InnerClass {  
  
        public int membre1;  
public static int membre2;  
        public static final String MSG= "Hello";  
  
        public void methode1() { }  
  
public static void methode2() { }  
  
    }  
  
}
```

Classes internes

- Restrictions d'accès

- Une classe locale à une méthode a accès aux variables locales et aux arguments de cette méthode à condition que ceux-ci soient déclarés avec le modificateur **final**.

```
public class OuterClass {  
  
    public void methode(final int arg) {  
        final int var= 5;  
        int var2= 10;  
  
        class LocalClass {  
            public LocalClass()  
            {  
                System.out.println(var);  
                System.out.println(arg);  
                System.out.println(var2);  
            }  
        }  
    }  
}
```

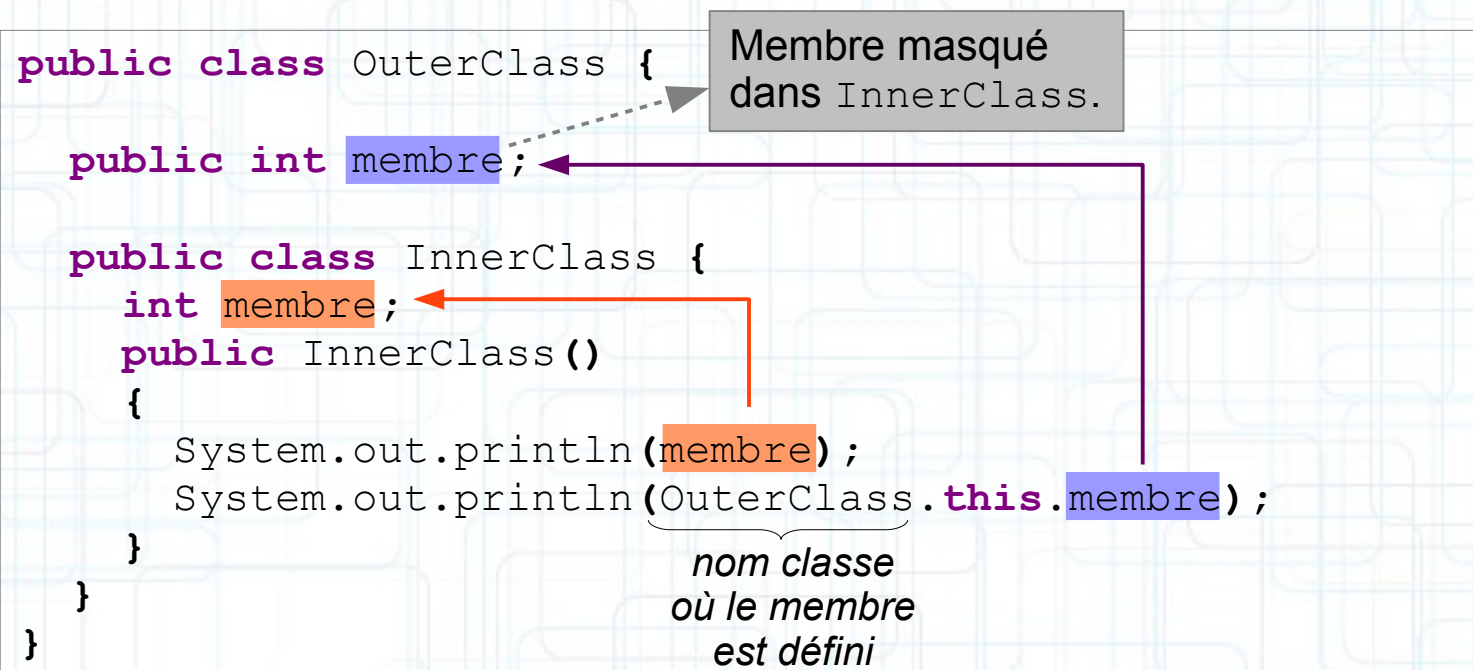
A dashed line originates from the `final` keyword in the method signature `final int arg` and points to the `arg` parameter in the `System.out.println(arg);` statement inside the `LocalClass`. Another dashed line originates from the `final` keyword in the variable declaration `final int var= 5;` and points to the `var` variable in the `System.out.println(var);` statement inside the `LocalClass`. The line for `System.out.println(var2);` is crossed out with a red line, indicating that the local variable `var2` is not accessible from the inner class.

Note : depuis Java 8, les variables ne doivent plus être "explicitly final" (usage du mot-clé **final**), mais "effectively final" (pas de modification).

Classes internes

- Restrictions d'accès

- Dans le cas où la déclaration d'un membre local à une classe interne masque un membre d'une classe englobante (*shadowing*), ce dernier n'est plus accessible directement.
- il est possible d'accéder à un membre masqué en utilisant la référence **this** préfixée du nom de la classe dans laquelle le membre est défini.



Classes internes

- **Restrictions d'accès - Résumé**

- Une classe interne a accès à tout les membres de la ou des classe(s) englobantes.
 - Les restrictions habituelles s'appliquent : une classe interne de classe ne peut avoir accès qu'aux membres de classe
 - *shadowing* : il est possible d'accéder à un membre masqué (redéfini localement avec le même nom) en utilisant la référence **this** préfixée du nom de la classe dans laquelle le membre est défini. Exemple : `InnerClassAccess.this.member3`
- Une classe locale à une méthode (ou un bloc) a accès aux variables locales à cette méthode (ou ce bloc)
 - Les variables doivent être déclarées **final**.
- Classes internes non statiques ne peuvent pas définir de membres de classe.