

# Introduction à la programmation en Java

## Cours 10

Stéphane Airiau

Université Paris-Dauphine

## Classes Internes

Il est possible de définir une classe à l'intérieur d'une classe !

➡ de telles classes peuvent simplifier le développement

- cacher des détails d'implémentation (visibilité restreinte)
- avoir des packages avec moins de classes

Il y a deux grands types de classes internes, qui sont assez différents

- classe interne d'instance

- cette classe va donc accéder à tous les champs de sa classe (même les privés)
- on ne peut créer une instance d'une classe interne que depuis une méthode d'instance de sa classe « englobante »

- classe interne statique

- pas de référence à des membres d'instance de sa classe « englobante »
  - on peut instancier cette classe (elle est **static** vis à vis de sa classe englobante).
- ➡ une classe interne static est vraiment comme une classe classique (sauf qu'elle est codée à l'intérieur d'une autre classe)

## Exemple classe interne `static` et `private`

---

```
1 public class Facture{
2     private static class Item {
3         String description;
4         int quantite;
5         double prixUnitaire;
6
7         double montant () {return quantite*prixUnitaire;}
8     }
9
10    private List<Item> liste = new ArrayList<> ();
11    ...
12 }
```

Ici, la classe interne est privée (pas besoin d'utiliser `private` pour les attributs).

⇒ seulement des méthodes de la classe englobante (ici `Facture`) ont accès à la classe interne.

## Exemple classe interne `static` et `private`

```
1 public class Facture{
2     private static class Item {
3         String description;
4         int quantite;
5         double prixUnitaire;
6
7         double montant () {return quantite*prixUnitaire;}
8     }
9
10    private List<Item> liste = new ArrayList<> ();
11    ...
12    public void addItem (String des, int q, double pU) {
13        Item newItem = new Item ();
14        newItem.description = des;
15        newItem.quantite = q;
16        newItem.prixUnitaire = pU;
17        liste.add (newItem);
18    }
19 }
```

Exemple de construction d'une instance de la classe interne.  
Tout se passe comme si la classe interne est une classe normale.

## Exemple classe interne `static` et `public`

```
1 public class Façture{
2     public static class Item {
3         private String description;
4         private int quantite;
5         private double prixUnitaire;
6         public Item(String desc, int q, double pU) {
7             this.description = desc;
8             this.quantite = q;
9             this.prixUnitaire = pU;
10        }
11        double montant () {return quantite*prixUnitaire;}
12    }
13
14    private List<Item> liste = new ArrayList<> ();
15
16    public void addItem(Item it) {
17        liste.add(it);
18    }
19
20 }
```

Utilisation de l'encapsulation (comme d'habitude).

N'importe qui peut alors construire un objet de la classe interne en utilisant le nom "complet", ici `Facture.Item`.

```
Facture.Item newItem = new Facture.Item("Moka Yrgacheffe", 2, 5.5);
```

Le compilateur génère deux classes différentes :

- `Facture.class`
- `Facture$Item.class`

Ce qui montre bien qu'il n'y a pas trop de différences entre une classe **static** et une classe "classique"

➡ juste un moyen pour cacher une classe !

Deux grosses différences :

- une classe interne **a accès** aux éléments de sa classe englobante !
- il faut donc un objet de la classe englobante pour qu'un objet de la classe interne puisse exister !

```
1 public class ReseauSocial {
2
3     public class Membre {
4         private String nom;
5         private List<Membre> amis;
6
7         public Membre(String nom) {
8             this.nom=nom;
9             amis = new LinkedList<>;
10        }
11        ...
12    }
13
14    private List<Member> membres;
15    ...
16 }
```



## Classes internes

---

```
1 public class ReseauSocial {
2
3     public class Membre {
4         private String nom;
5         private List<Membre> amis;
6
7         public Membre (String nom) {
8             this.nom=nom;
9             amis = new LinkedList<>;
10        }
11        ...
12    }
13
14    private List<Member> membres;
15    ...
16    public void accepte (String nom) {
17        membres.add (new Membre (nom) );
18    }
19 }
```

## Classes internes : utilisation

---

```
ReseauSocial faceBouc = new ReseauSocial();  
ReseauSocial.Membre chevre = faceBook.accepte("chevre");
```

Si la chevre veut quitter le réseau, elle peut appeler la méthode pour partir. On peut implémenter la méthode comme suit :

```
1 public class ReseauSocial {  
2  
3     public class Membre {  
4         ...  
5         public void quitter() {  
6             membres.remove(this);  
7         }  
8     }  
9  
10    List<Membre> membres;  
11 }
```

La classe interne peut accéder aux attributs de la classe englobante !

## Classes internes : utilisation

---

```
ReseauSocial faceBouc = new ReseauSocial ();  
ReseauSocial.Membre chevre = faceBook.accepte ("chevre");
```

```
1 public class ReseauSocial {  
2  
3     public class Membre {  
4         ...  
5         public void quitter() {  
6             efface(this);  
7         }  
8     }  
9  
10    List<Membre> membres;  
11    public void accepte (String nom) { ... }  
12    public void efface (Membre m) { ... }  
13 }
```

La classe interne peut accéder aux méthodes de la classe englobante !

## Classes internes : accès à la classe englobante

---

Imaginons que l'on veuille vérifier si un membre appartient au bon réseau.

```
1 public class ReseauSocial {
2
3     public class Membre {
4         ...
5         public void appartient(ReseauSocial rs) {
6             return ? == rs;
7         }
8     }
9 }
```

## Classes internes : accès à la classe englobante

---

Imaginons que l'on veuille vérifier si un membre appartient au bon réseau.

```
1 public class ReseauSocial {
2
3     public class Membre {
4         ...
5         public void appartient (ReseauSocial rs) {
6             return ReseauSocial.this == rs;
7         }
8     }
9 }
```

## Classes internes : création d'une instance

---

```
1 public class ReseauSocial {
2
...
13
14     private List<Member> membres;
15     ...
15     public void accepte (String nom) {
16         membres.add (new Membre (nom) );
17     }
18 }
```

Ici `new Membre (nom)` est un raccourci pour `this.new Membre (nom)`.

En dehors de la classe englobante, on peut créer un objet de la classe interne à partir d'une instance de la classe englobante :

```
ReseauSocial.Membre mark = faceBouc.new Membre ("Mark");
```

## Petit détail

---

- on ne peut déclarer dans une classe interne que des variables **static** qui soient des constantes (sinon, il y aurait une ambiguïté)

## Exemples

### Classe interne **static**

```
1 public class Externe {  
2     int nbEtudiants;  
3     public static class Interne {  
4         public int nbEtudiantsMax=25;  
5     }  
6 }
```

```
1 Externe a = new Externe ();  
2 Externe.Interne b=  
3     new Externe.Interne ();
```

### Classe interne **non static**

```
1 public class Externe {  
2     int nbEtudiants;  
3     public class Interne {  
4         public int nbEtudiantsMax=25;  
5     }  
6 }
```

```
1 Externe a = new Externe ();  
2 Externe.Interne b=  
3     a.new Interne ();
```



## Classes Locales

---

ou comment définir de manière concise des classes qui implémentent une interface

On peut définir une classe à l'intérieur d'une **méthode**!

Considérons la méthode suivante :

```
public static IntSequence randomInts(int low, int high)
```

- IntSequence est une interface.

ou comment définir de manière concise des classes qui implémentent une interface

On peut définir une classe à l'intérieur d'une **méthode**!

Considérons la méthode suivante :

```
public static IntSequence randomInts (int low, int high)
```

- IntSequence est une interface.
- le but de randomInts est de retourner une séquence d'entiers aléatoires entre deux bornes.

ou comment définir de manière concise des classes qui implémentent une interface

On peut définir une classe à l'intérieur d'une **méthode**!

Considérons la méthode suivante :

```
public static IntSequence randomInts(int low, int high)
```

- `IntSequence` est une interface.
- le but de `randomInts` est de retourner une séquence d'entiers aléatoires entre deux bornes.
- la méthode doit retourner un objet qui implémente l'interface `IntSequence`

ou comment définir de manière concise des classes qui implémentent une interface

On peut définir une classe à l'intérieur d'une **méthode** !

Considérons la méthode suivante :

```
public static IntSequence randomInts(int low, int high)
```

- `IntSequence` est une interface.
- le but de `randomInts` est de retourner une séquence d'entiers aléatoires entre deux bornes.
- la méthode doit retourner un objet qui implémente l'interface `IntSequence`
- l'appelant n'est pas intéressé par le type de l'objet  
Tout ce qu'il l'intéresse, c'est que l'objet implémente l'interface

ou comment définir de manière concise des classes qui implémentent une interface

On peut définir une classe à l'intérieur d'une **méthode** !

Considérons la méthode suivante :

```
public static IntSequence randomInts(int low, int high)
```

- `IntSequence` est une interface.
  - le but de `randomInts` est de retourner une séquence d'entiers aléatoires entre deux bornes.
  - la méthode doit retourner un objet qui implémente l'interface `IntSequence`
  - l'appelant n'est pas intéressé par le type de l'objet  
Tout ce qu'il l'intéresse, c'est que l'objet implémente l'interface
- ➡ on va faire une classe *ad hoc* à l'intérieur de la méthode !

## Exemple Classes Locales

---

```
1 private static Random gen = new Random();
2
3 public static IntSequence randomInts(int low, int high) {
4     class RandomSequence implements IntSequence {
5         public int next () {
6             return low + gen.nextInt (high-low+1);
7         }
8         public boolean hasNext () {return true;}
9     }
10    return new RandomSequence ();
11 }
```

- pas besoin de spécifier si la classe locale est **public** ou **private** puisqu'elle ne sera jamais accessible à l'extérieur de la méthode!
- la classe a accès aux variables accessible à l'intérieur de son scope. Dans l'exemple, elle a **directement** accès à
  - low et high qui sont des paramètres de la méthode
  - gen qui est une variable **static** de la classe

## Classe Anonyme

---

```
1 private static Random gen = new Random();
2
3 public static IntSequence randomInts(int low, int high) {
4     class RandomSequence implements IntSequence {
5         public int next () {
6             return low + gen.nextInt (high-low+1);
7         }
8         public boolean hasNext () {return true;}
9     }
10    return new RandomSequence ();
11 }
```

Dans l'exemple, on a utilisé le nom `RandomSequence` une fois pour créer l'objet.

Dans ce cas, on pourrait se passer d'utiliser un nom ➡ classe **anonyme**.

## Classe Anonyme

---

```
1 private static Random gen = new Random();
2
3 public static IntSequence randomInts(int low, int high) {
4     return new IntSequence() {
5         public int next() {
6             return low + gen.nextInt(high-low+1);
7         }
8         public boolean hasNext() {return true;}
9     }
10
11 }
```

C'est assez pratique et concis.

Très utilisé pour les interfaces graphiques et les threads.

Maintenant, Java propose un autre moyen pour faire cela : l'utilisation de "lambda expressions"

Mais on garde cela pour le cours de Java avancé.



## λ expression

---

```
(String first, String second) -> first.length()-second.lenght()
```

```
Runnable task () -> {for (int i=0;i<1000;i++) doWork();}
```

Une interface avec une seule méthode à implémenter est appelée une interface fonctionnelle

ex: Runnable et Comparator.

Au lieu de créer un objet qui implémente l'interface Comparator, on peut fournir une λ expression

```
Arrays.sort(words,  
    (String first, String second) -> first.length()-second.lenght())
```

On peut même omettre les paramètres String car Java pourra les inférer car Java sait qu'il va attendre un Comparator<String>!

```
Arrays.sort(words,  
    (first, second) -> first.length()-second.lenght())
```