Introduction programmation Java

Cours 7

Stéphane Airiau

Université Paris-Dauphine

Classes Internes

Des classes à l'intérieur de classe

- avec les packages, on peut gérer ses classes sans risque de collision entre noms.
- on voudrait éviter d'avoir trop de classes, en particulier, d'avoir des classes "outils" ou "support" qui servent seulement à des classes principales dans notre hierarchie de classe

⇨ on peut "cacher" des classes à l'intérieur d'une classe.

Exemple classe interne `static` et `public`

```
1  public class Facture{
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).

classes internes `static` et `public`

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 !

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.

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)

Classes internes

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   ...
15   public void accepte(String nom){
16     membres.add(new Membre(nom));
17   }
18 }
```

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 ambiguité)

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.

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

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.
- 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`

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.
- 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

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.
- 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 had 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"

Expressions λ

Il n'y a pas de type "fonction" en `Java`[a].

En `Java`, tout est *objet* et une fonction est donc exprimée par un objet qui va implémenter une certaine interface.

Les expressions λ sont un moyen syntaxique pour créer facilement de telles instances.

---

a. Certes, il y a une classe `Function`, mais on voudrait un type à part entière

Syntaxe d'une expression λ

La syntaxe est très similaire à la syntaxe utilisée usuellement en mathématique.

```
(String left, String right) -> left.length()-right.length();
```

Si le résultat peut difficilement être exprimé en une expression, on peut écrire un *bloc* de code qui contient une instruction **`return`**.

```
(String left, String right) -> {
    if (left.length() < right.length())
        return -1;
    else if (left.length() == right.length())
        return 0;
    else
        return +1;
}
```

On peut omettre les types s'ils peuvent être inférés par `Java`.

```
Comparator<String> comp = (left, right) -> left.length()-right.length();
```

S'il n'y a pas de paramètres, on utilise quand même `()`.

Utilisation d'une expression λ

On peut utiliser une expresssion λ dès qu'on attend un objet qui implémente une interface avec une seule méthode.

On appelle une **interface** fonctionnelle une interface qui ne contient qu'une méthode.

C'est bien le cas avec `Comparator`

Il y a d'autres cas existants dans `Java` comme `Runnable` pour implémenter des application multi thread.

Références méthode

Supposons qu'on a un tableau de `String` appelé `strings` que l'on veut trier sans tenir compte de la casse.

```
Arrays.sort(strings, (x,y)-> x.compareToIgnoreCase(y));
```

On a le droit d'écrire

```
Arrays.sort(strings, String::compareToIgnoreCase);
```

`String::compareToIgnoreCase` joue le rôle de l'expression λ.

Autres exemples :

```
list.removeIf(Object::isNull);
```

ici, cet appel va enlever toute valeur `null` dans la liste `list`.

```
list.forEach(System.out.println);
```

Cet appel imprime chaque élément de la liste `list`.

Références méthode

- `classe::méthode_d_instance`
  Le premier paramètre est le "receveur" de la méthode, toute autre paramètre est passé à la méthode.

```
String::compareToIgnoreCase
```

  a le même sens que

```
(x,y)-> x.compareToIgnoreCase(y)
```

- `classe::méthode_de_classe`
  tous les paramètres sont passés à la méthode de classe

```
list.removeIf(Object::isNull);
```

- `objet::méthode_d_instance`
  la méthode est invoquée sur l'objet et tous les paramètres sont passés à la méthode
  `System.out::println` est équivalent à
  `x-> System.out.println(x).`

Utilisation avec les `Stream`

Supposons qu'on a une collection de chaînes de caractères et que l'on veut compter les mots de plus de 12 caractères.

```
1 int count =0;
2 for (String m: words)
3     if (m.length()>12)
4         count++;
```

L'idée avec les `streams` sera d'écrire le code suivant :

```
long count = words.stream().filter(w-> w.length > 12).count();
```

En lisant, on comprends exactement ce qui se passe.
Désormais, `Java` peut optimiser l'exécution de ce code

1. création d'un stream
2. opérations intermédiaires transformant le stream initial (en d'autres, pourrait utiliser plusieurs étapes)
3. opération terminale pour produire le résultat.

Nous allons voir plus en détail ces trois phases.

Création de `streams`

- A partir d'une collection : appel de la méthode `stream()`
- A partir d'un tableau :
  - utiliser la méthode de classe `of` de la classe `Stream` :

```
Stream<String> words = Stream.of(line.split(","));
// split découpe une chaîne de caractères et retourne un tableau de String
```

  - appel de la méthode de classe `stream(array, from, to)` de la classe `Arrays`

```
Stream<String> words = Arrays.stream(line.split(","), 3, 7);
```

- on peut créer un stream vide `Stream.empty();`
- on peut créer des stream infini
  - avec la méthode `generate` :

```
Stream<String> echos = Stream.generate( () -> "Echo”);
```

```
Stream<Double> randDoubles =
        Stream.generate(Math::random);
```

  - avec la méthode `iterate`

```
Stream<Integer> intSeq = Stream.iterate(0, n -> n.add(1);)
```

Création de `streams` (suite)

On peut aussi avoir une séquence finie avec `iterate` en ajoutant un test d'arrêt.

autres exemples de de création

- la méthode d'instance `tokens()` de la classe `Scanner` retourne un `Stream<String>`.
- la méthode de classe `lines(Path p)` de la classe `Files` retourne également un `Stream<String>`.

Transformation de `streams` : filtre

un filtre permet de récupérer un nouveau stream dont les éléments ont passé un test.
Le test doit donc être une fonction qui retourne un booléen (et doit donc suivre l'interface fonctionnelle `Predicate<T>`).

```
long count = words.stream().filter(w-> w.length > 12).count();
```

Transformation de `streams` : `map`

L'opérateur `map` permet de *transformer* le stream.

```
Stream<String> lowerCase =
          words.stream().map(String::toLowerCase);
```

La plupart du temps, il n'existera pas une référence méthode adéquate, on pourra donc utiliser une expression λ.

```
Stream<String> firstLetters =
          word.stream().map( s -> s.substring(0,1));
```

transformation de stream

La méthode `limit(int n)` retourne un nouveau stream qui se termine après au plus `n` éléments

```
Stream<Double> randDoubles =
        Stream.generate(Math::random).limit(100);
```

La méthode `skip(int n)` fait l'opposé : elle retourne un stream sans les premiers `n` éléments

La méthode `takeWhile(predicate)` prend tout élément tant que le prédicat est vrai, et s'arrête ensuite.

```
Stream.of(1,3,5,6,8,6,2,18)
.takeWhile(no -> no<=5).forEach(System.out::println);
```

La méthode `dropWhile(predicate)` fait le contraire : elle ne prend pas les éléments tant que le prédicat est vrai, et retourne donc le stream partant du moment ou le prédicat devient faux.

```
Stream.of(1,3,5,6,8,6,2,18).
dropWhile(no -> no<=5).forEach(System.out::println);
```

Concaténation de streams

La méthode de classe `concat` de la classe Stream concatène deux streams. Evidemment, il vaudrait mieux que le premier stream ne soit pas infini !

La méthode `distinct` retourne un stream qui n'a pas de doublons.

La méthode `sorted` permet de trier un stream (mieux vaut qu'il contienne des objets d'une classe qui implémente `Comparable`!)

La méthode `peek` retourne le même stream, mais applique une fonction à chaque élément

```
Integer[] powersOfTwo =
          Stream.iterate(1.0, n ->2*n)
              .peek(e -> System.out.println("treating "+e))
              .limit(20).toArray();
```

Réduction de streams

Réduction simple : `count`, `min`, `max`
Point délicat : que se passe-t-il quand le stream est vide ?
⇨ certaines opération de réduction retourne une valeur de type `Optional<T>`.

```
Optional<String> largest = words.max(String::compareToIgnoreCase);
```

findFirst retourne la première valeur dans une collection non vide

```
Optional<String> startsWithW =
        words.filter(s -> s.startsWith("W")).findFirst();
```

Il existe de la même manière `findAny` si on ne se focalise pas vraiment sur le premier.
Si on veut juste savoir s'il y a au moins un élément, `AnyMatch` sera alors judicieux.

```
boolean b = words.anyMatch(s -> s.startsWith("W"));
```

Il existe également des méthodes `allMatch` et `NoneMatch` qui vérifient si tous ou aucun élément satisfait un prédicat.

Le type optionel

L'idée est de ne pas utiliser qu'une méthode retourne `null` quand il n'y a pas de résultat (provoque un `NullPointerException` pas toujours simple à corriger)

```
String result = optionalString.orElse("");
```

```
String result =
      optionalString.orElseThrow(IllegalStateException::new);
```

il est aussi possible de lancer un calcul alternatif avec `orElseGet()`. `ifPresent` prend comme paramètre une fonction : si la valeur optionelle existe, elle est passée à cette fonction.

```
optionalValue.ifPresent(v -> results.add(v));
```

Obtenir le résultat

ex affichage :

```
stream.forEach(System.out::println);
```

ex : obtenir un tableau :

```
String[] result = stream.toArray(String[]::new);
```

Notez que le type de `stream.toArray()` est `Object[]` car on ne peut pas faire un tableau avec des génériques.

L'interface `Collector` est utilisée pour collecter les éléments dans une autre structure. La classe `Collectors` propose des méthodes pour des structures classiques.

```
List<String> result = stream.collect(Collectors.toList());
```

```
Set<String> result = stream.collect(Collectors.toSet());
```

```
TreeSet<String> result =
      stream.collect(Collectors.toCollection(TreeSet::new));
```

Obtenir le résultat (suite)

concatenation de chaînes de caractères :

```
String result = stream.collect(Collectors.joining());
```

concatenation de chaînes de caractères avec ajout d'un délimiteur :

```
String result = stream.collect(Collectors.joining(", "));
```

On peut utiliser une méthode d'aggrégation (somme, compter, moyenne, min, max) :

```
IntSummaryStatistics summary = stream.collect(
                Collectors.summarizingInt(String::length)));
double avg = summary.getAverage();
double max = summary.getMax();
```

Obtenir le résultat : reduce

```
Optional<Integer> sum =
      values.stream().reduce((agg, next) -> agg +next);
```

est équivalent à la somme.
On retrouve ici des choses similaires à la programmation fonctionnelle.