Programmation orienté objet JAVA

deuxième partie

JAVA

Programmation orientée objet

Pr Hafidi Imad **imad.hafidi@gmail.com**

Classe intern

Types de classes internes

- Depuis la version 1.1, Java permet de définir des classes à l'intérieur d'une classe
- Il y a 2 types de classes internes :
  - classes définies à l'extérieur de toute méthode (au même niveau que les méthodes et les variables d'instance ou de classe)
  - classes définies à l'intérieur d'une méthode

Classes internes
non incluses dans une méthode

- Le code de ces classes est défini à l'intérieur d'une autre classe, appelée classe englobante, au même niveau que les autres membres :

```
public classe ClasseE {
        private int x;
        class ClasseI {
                Code de la classe interne
        }
         ...
        public String m() { . . . }
        ...
}
```

Modificateurs

- Une telle classe peut avoir les mêmes degrés d'accessibilite que les membres d'une classe : **private**, *package*, **protected**, **public**
- Elle peut aussi être **abstract** ou **final**

Nommer une classe interne

- La classe englobante (**ClasseE**) fournit un espace de noms pour une classe interne (**ClasseI**) : son nom est de la forme « **ClasseE.ClasseI** »

Importer des classes internes

- On peut importer une classe interne : **import ClasseE.ClasseI;**
- On peut aussi importer toutes les classes internes d'une classe :
- **import ClasseE.*;**
- Une classe interne ne peut avoir le même nom qu'une classe englobante (quel que soit le niveau d'imbrication)

2 types de classes internes définies à l'extérieur d'une méthode

- ***Classes static*** : leurs instances ne sont pas liées à une instance de la classe englobante
- ***Classes non static*** : une instance d'une telle classe est liée à une instance de la classe englobante

Classes internes **static**

- Une classe interne **static** joue à peu près le même rôle que les classes non internes
- En définissant une telle classe, le programmeur indique que la classe interne n'a de sens qu'en relation avec la classe externe

Visibilite pour les classes internes **static**

- Une classe interne **static** a accès à toutes les variables **static** de la classe englobante, même les variables **static private**
- Elle n'a pas accès aux variables non **static**
- La classe englobante a accès à tous les membres de la classe interne, qu'ils soient **static** ou non, et même s'ils sont **private**

Création d'une instance d'une classe interne **static**

- A l'extérieur de la classe englobante, on peut créer une instance de la classe interne par :
- **ClasseE.ClasseI x = new ClasseE.ClasseI(...);**

Exemple de classe interne **static**

- On veut récupérer les valeurs minimale et maximale d'un tableau, variable d'instance d'une classe
- Pour cela on écrit une méthode qui renvoie une paire de nombres (pour éviter un double parcours du tableau)
- On crée une classe interne **static Extrema** pour contenir cette paire de nombres

Classe TableauInt

- Classe qui enveloppe un tableau : Elle définit une classe interne **Extrema** dont les instance contiendront la plus petite et la plus grande valeur du tableau

```
public class TableauInt {
        private int[] valeurs;
        public static class Extrema {
                private int min, max;
                private Extrema(int min, int max) {
                        this.min = min;
                        this.max = max;
                }
                public int getMin() { return min; }
                public int getMax() { return max; }
```

```
public Extrema getMinMax() {
        if (nbElements == 0) return null;
        int min = valeurs[0];
        int max = min;
        for (int i = 1; i < nbElements; i++) {
                if (valeurs[i] < min)
                        min = valeurs[i];
                if (valeurs[i] > max)
                        max = valeurs[i];
        return new Extrema(min, max);
}
```

Utilisation de getMinMax()

- **Dans une autre classe que TableauInt TableauInt t;**

```
...
TableauInt.Extrema extrema = t.getMinMax();
System.out.println("Min =" + extrema.getMin());
System.out.println("Max = "+ extrema.getMax());
```

Classes internes non **static**

- Une instance d'une classe interne non **static** ne peut exister que liée à une instance de la classe englobante (appelée **ClasseE** pour la suite)
- Le code de la classe interne peut désigner cette instance de la classe englobante par : **ClasseE.this**
- Les classes internes non **static** ne peuvent avoir de variables **static**

Visibilité pour les classes internes non **static**

- Une classe interne non **static** partage tous les membres (même privés) avec la classe dans laquelle elle est définie :
  - la classe interne a accès à tous les membres de la classe englobante
  - la classe englobante a accès à tous les membres de la classe interne

Nommages particuliers liés aux classes internes

- Une classe interne non **static** peut accéder à tout membre (variable ou méthode) ou constructeur de la classe dans laquelle elle est définie
- Si le membre n'est pas caché, elle peut le nommer simplement par son nom
- Si le membre est une méthode, l'appel s'applique évidemment au *this* englobant

Nommages particuliers liés aux classes internes (suite)

- Si le membre est caché, elle le nomme en le préfixant par « **ClasseE.this** »
  - ce qu'elle peut aussi faire, pour des raisons de lisibilité, même si le membre n'est pas caché
  - Pas d'ambiguïte car une classe interne ne peut avoir le même nom qu'une classe englobante

Création d'une instance d'une classe interne non **static**

- Soit **ClasseI** une classe interne non **static** de **ClasseE**
- Soit **instanceClasseE** est une instance de **ClasseE**
- Dans le code d'une autre classe, une instance de **ClasseI** liée à l'instance de la classe englobante **instanceClasseE** est créée par **instanceClasseE. new ClasseI(...)**

Exemple de classe interne non **static**

```
public class Tableau<T> un tableau  implements Iterable<T> {
 private Object[] valeurs;
private int nbElements;
public Tableau(int n) {
valeurs = new Object[n];
}
public Iterator<T> iterator() {
        return new IterTableau();
 }
private class IterTableau implements Iterator<T> {
```

Définition de la classe IterTableau

```
private class IterTableau implements Iterator<T> {
private int indiceCourant = 0;
public boolean hasNext() {
}
publicT next() {
return (T)valeurs[indiceCourant++];
}
public void remove() { throw new
UnsupportedOperationException(); }
```

Utilisation de Tableau

```
Tableau<Integer> t1 = new Tableau<Integer>(10), t2
= new Tableau<Integer>(5);
// Ajout de quelques éléments dans t1 et t2 ...
// Affichage des éléments de t1 et t2 for(int i : t1) {
System.out.println(i); }
for(int i : t2) { System.out.println(i);
}
```

Avantage des classes internes

- Dans l'exemple précèdent, le fait que la classe interne **IterTableau** puisse accéder aux variables privées de la classe **Tableau** permet d'encapsuler complètement la structure de données de **Tableau**
- Pour faire la même chose avec une classe externe, il aurait fallu ajouter une méthode **get(int i)** à **Tableau**, pour que **IterTableau** puisse obtenir le ième élément d'une instance de **Tableau**

Classe interne **static** ou non ?

- Le critère de choix est le suivant :
  - ne choisir non **static** que si la classe interne doit accéder à une variable d'instance de la classe englobante,
  - sinon, choisir **static**

Héritage des types internes

- Les types internes (classes ou interfaces) s'héritent comme les autres membres
- Une sous-classe peut ainsi utiliser une classe interne d'une classe mère si elle est **protected**, par exemple, pour définir une sous-classe de cette classe

Classe interne abstraite

- Une classe peut avoir une interface interne ou une classe interne abstraite
- On ne doit pas pour autant la déclarer abstraite (bien qu'il soit difficile de voir un exemple où on pourrait créer une instance d'une telle classe)
- Ses classes filles devront fournir la classe non abstraite (classe fille de la classe abstraite ou classe qui implémente l'interface) qui permettra le bon fonctionnement de la classe

Classes internes locales, définies à l'intérieur d'une méthode

- 2 types de telles classes (font partie des *inner classes* en anglais) :
  - classes avec un nom
  - classes anonymes
- Les classes anonymes sont utilisées plus souvent que les classes avec nom

Utilisation des classes anonymes

- Une classe interne anonyme sert à redéfinir (resp. implémenter) « à la volée » une ou plusieurs méthodes d'une classe (resp. d'une interface)
- Remarque : ca ne sert à rien d'ajouter de nouvelles méthodes publiques car on ne pourra les appeler de l'extérieur de la classe

Classes internes anonymes

- On peut utiliser des instances de classes internes à une méthode, sans nom, dites anonymes, sous- classe d'un classe mère ***ClasseMère***
- L'instance est crée par
  new *ClasseMère*(*listeParamètres*) {// Définition de la classe (variables, // méthodes)
  où ***listeParamètres*** doit correspondre à la signature d'un des constructeurs de ***ClasseMère***

Création d'une instance d'une classe interne anonyme

- Ces classes anonymes n'ont pas de constructeur puisqu'elles n'ont pas de nom
- En fait, la création de l'instance de la classe anonyme fait un appel au constructeur de la classe mère dont la signature correspond à ***listeParamètres*** :

```
Cercle c = new Cercle(p, r) {
        public void dessineToi(Graphics g) { ... }
        };
```

Initialiseurs

- Il existe des initialiseurs non **static** qui peuvent être utilisés pour initialiser les instances des classes anonymes
- C'est parfois utile puisque que les classes anonymes n'ont pas de constructeur
- La syntaxe et l'emplacement sont semblables à un initialiseur **static**, mais sans le mot clé **static** ; c'est en fait un simple bloc placé en dehors de toute méthode

Classes anonymes qui implémentent une interface

- ***ClasseMère*** peut être remplacé par un nom d'interface qu'implémentera la classe anonyme ; dans ce cas la liste des paramètres doit être vide (appel du constructeur de **Object**) :
- New Interface(){Définition de la classe}
- **Par exemple,**
  new Iterator<T>() { . . . }

Exemple de classe anonyme

- **public class Tableau<T> implements Iterable<T> { ...**
- **public Iterator<T> iterator() { return new Iterator<T>() {**
  // Définition classe qui implémente Iterator private int indiceCourant = 0;
  public boolean hasNext() {
  return indiceCourant < valeurs.length; }
  ...
  }; // Fin de la classe anonyme
  } // Fin de la méthode iterator()

Avantages des classes anonymes

- Si le code de la classe anonyme est court, l'utilisation d'une classe anonyme améliore la lisibilité car le code de la classe est proche de l'endroit où il est utilisé
- Ça évite aussi d'avoir à chercher un nouveau nom de classe ;-)

Inconvénients des classes anonymes

- Si le code est long, la classe anonyme va au contraire nuire à la lisibilité
- Pas possible de créer plusieurs instances d'une classe anonyme

Contexte d'exécution des classes internes locales

- Les classes internes locales ont accès
  – aux variables d'instance et de classe de la
- classe englobante (même privées)
  – aux paramètres **final** et aux variables
- locales **final** de la méthode

Utilisation des variables final

- Comment faire si la classe interne utilise une variable de la méthode englobante et que cette variable doit être modifiée par la méthode ?
- Une solution est d'utiliser un tableau pour contenir la valeur de la variable
- Ainsi le tableau est déclaré **final** mais il est tout de même possible de modifier les éléments du tableau

Interfaces internes

- Une classe ou une interface peut contenir des interfaces internes, implicitement **static**, qui peuvent avoir une accessibilité **public** ou *package*
- Le paquetage **java.util** du JDK fournit par exemple l'interface **Entry** interne à l'interface **Map**

Classes internes à une interface

- Une interface peut (rarement) contenir des classes internes d'accessibilité quelconque
- (**public**, *package*, **protected**, **private**)
- Cette classe peut, par exemple,
  - être une implantation par défaut de l'interface
  - fournir un type utilisé par les méthodes de l'interface
  - être le type d'une instance partagée (**public static final**) par toutes les classes qui implémentent l'interface

Types énumérés

Types énumérés

- ls ont éte□ ajoutés par le JDK 5.0
- Ils permettent de définir un nouveau type en énumérant toutes ses valeurs possibles (par convention, les valeurs sont en majuscules)
- Plus sûrs que d'utiliser des entiers pour coder les différentes valeurs du type (vérifications à la compilation)
- Utilisés comme tous les autres types

Exemple d'erreur sans énumération

- Une méthode **setDate(int, int, int)**
- Que signifie **setDate(2010, 5, 8)** ?
- 8 mai 2010 ? 5 août 2010 ?
- Il est facile d'inverser les significations des paramètres si on utilise cette méthode
- Problème sérieux si on veut indiquer la date de largage du satellite !
- Ce type d'erreur sera repéré à la compilation si on crée un type énumére pour les mois

Enumération « interne » à une classe

- On peut définir une énumération à l'intérieur
- d'une classe :
- **public class Carte { public enum Couleur**
- **{TREFLE, CARREAU, COEUR, PIQUE};**
- **private Couleur couleur; ...**
- **this.couleur = Couleur.PIQUE;** Depuis une autre classe :
- **carte.setCouleur(Carte.Couleur.TREFLE);**

Énumération « externe » à une classe

- En fait les types énumérés sont des classes qui héritent de la classe **java.lang.Enum**
- Comme les classes ils peuvent être définis indépendamment d'une classe
- Le fichier qui contient une énumération publique doit avoir le nom de l'énumération avec le suffixe « .java »

Exemple d'énumération externe

- **public enum CouleurCarte {TREFLE, CARREAU, COEUR, PIQUE;**
- **}**
- **public class Carte { private CouleurCarte couleur; …**
- **}**

Les valeurs

- **toString()** retourne le nom de la valeur sous forme de **String** ; par exemple, "**CouleurCarte.TREFLE.toString()**" retourne "TREFLE"
- **static valueOf(String)** renvoie la valeur de l'énumération correspondant à la **String**
- La méthode **static values()** retourne un tableau contenant les valeurs de l'énumération ; le type des éléments du tableau est l'énumération

Utilisable avec ==

- Dans le code de la classe **Carte** :
- **CouleurCarte couleurCarte;**

  …

  if(couleurCarte == CouleurCarte.PIQUE))

Utilisable dans un **switch**

- Dans le code de la classe **Carte** :
- **CouleurCarte couleurCarte; ...**
  switch(couleurCarte) { case PIQUE:
- **. . . break;**
- **case TREFLE: ...**
- **break; default:**
- **... }**

Méthodes et constructeurs

- Comme les classes les énumérations peuvent comporter des méthodes et des constructeurs

Exemple

- Associer des **String** aux valeurs :
- **public enum Couleur {TREFLE("Trèfle"), CARREAU("Carreau"), COEUR("Coeur"), PIQUE("Pique"); private String couleur; Couleur(String couleur) {**
- **this.couleur = couleur; }**
- **public String toString() { return couleur;**

```
public enum ValeurCarte { DEUX(2),TROIS(3),
QUATRE(4), CINQ(5), SIX(6), SEPT(7), HUIT(8),
NEUF(9), DIX(10),VALET(10), DAME(10), ROI(10),
AS(11); private int valeur;ValeurCarte(int valeur) {
this.valeur = valeur;}
public int getValeur() { return valeur;
}
```

- Une énumération peut implémenter une interface
- Toutes les énumérations implémentent automatiquement l'interface **Serializable**
- (voir cours sur les entrées-sorties)
- Il n'est pas possible d'hériter de la classe **Enum** ni d'une énumération

Collections et énumérations

- 2 types de collections de **java.util** (voir cours sur les collections) sont particulièrement adaptés pour les énumérations :
- – **EnumMap** est une **Map** dont les clés ont leur valeur dans une énumération
- – **EnumSet** est un **Set** dont les éléments appartiennent à une énumération

Collections

Exceptions