Compléments sur le langage Java

Université de Nice - Sophia Antipolis

Version 3.4.4 – 16/2/10

Richard Grin

Plan de cette partie

- Classes internes, initialiseurs non static
- Clonage
- Finalize
- ShutdownHook

Classes et interfaces internes

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

Emplacement du code

- 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 {
    . . .
  }
  public String m() { . . . }
  . . .
}
```

Code de la classe interne

Modificateurs

- Une telle classe peut avoir les mêmes degrés d’accessibilité 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.*;`

Restriction sur les noms des classes internes

- 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`** (*nested class*) : leurs instances ne sont pas liées à une instance de la classe englobante
- Classes non **`static`** (*inner class* en anglais) : 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

Visibilité 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`

```
public class TableauInt {
  private int[] valeurs;
  . . . // suite sur transparents suivants
```

- 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

Classe interne `Extrema`

```
public class TableauInt {
  . . .
  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; }
  }
```


Suite classe `TableauInt`

```
  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'ambiguïté 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>
       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> {
  }
}
```

Classe qui enveloppe un tableau

Elle fournit un itérateur pour parcourir ce tableau ; cet itérateur est associé à une instance particulière de `Tableau`

Définition de la classe interne dans le transparent suivant

Définition de la classe **IterTableau**

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

Accès direct aux attributs privés de la classe englobante

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 précédent, 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 i^{ème} 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 (incluses dans une méthode)

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 : ça ne sert à rien d'ajouter de nouvelles méthodes publiques car on ne pourra les appeler de l'extérieur de la classe (vous voyez pourquoi ? essayez de répondre après avoir lu la suite)

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 créée 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) { … }
};
```

Définition de la classe anonyme

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

Cloner un objet

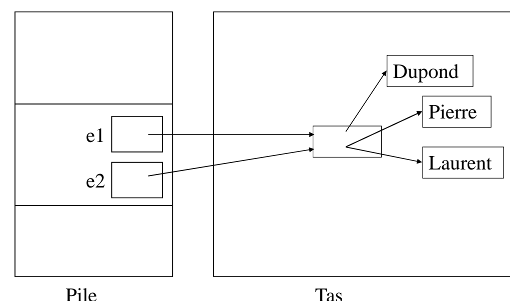
Copier un objet

- Si on copie une variable qui référence un objet, on obtient une autre référence au même objet :

```
Employe e1 = new Employe("Dupond", "Pierre");
e2 = e1;
e2.setPrenom("Laurent");
e1.getPrenom();
```

fournira Laurent

Copier un objet



Cloner un objet

- Dans certains cas on veut effectuer une copie conforme d'un objet pour obtenir un autre objet, qui n'a pas la même adresse en mémoire que l'original
- On dit qu'on clone un objet

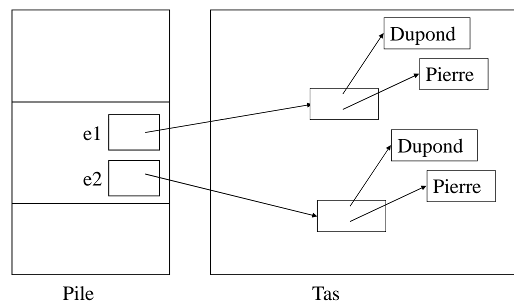
Pourquoi cloner un objet ?

- Pour éviter de fournir à des objets extérieurs une référence à un objet sensible
  - On évite ainsi des modifications de l'objet sensible qui pourrait nuire au bon fonctionnement de l'application
- On peut aussi vouloir conserver un objet sous son état initial avant de le modifier (pour réutiliser ensuite cet état initial)

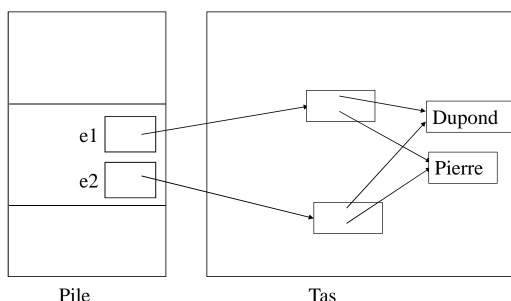
Clonage de surface ou clonage en profondeur

- Si l'état d'un objet (les variables) référence des objets, un clonage de l'objet peut se contenter de copier l'adresse de ces objets (copie de surface) ou faire un clonage de ces objets (copie en profondeur)
- Il est souvent nécessaire de faire un clonage en profondeur, ou au moins à une certaine profondeur (voir exemple de clonage de facture)

Clonage en profondeur



Clonage de surface



Méthode `clone()` de `Object`

- Cette méthode de la classe `Object` permet le clonage de l'objet courant ; son profil est

```
protected Object clone()
  throws CloneNotSupportedException
```

- Elle est `protected`. On ne peut donc l'utiliser sans créer une sous-classe qui redéfinit `clone()` (car les nouvelles classes que l'on écrit ne sont pas dans le même paquetage que `Object`)
- Elle effectue un clonage de surface
- Rappel : les tableaux héritent de cette méthode

Interface `Cloneable`

- Un objet ne peut être cloné que s'il est une instance d'une classe qui implémente l'interface `Cloneable`
- Si la classe de l'objet courant n'implémente pas `Cloneable`, la méthode `clone()` de `Object` renvoie une exception `CloneNotSupportedException`
- Cette interface est vide ; elle sert seulement de « marqueur » (*flag*) pour indiquer si un objet peut être cloné ou non

Comment permettre le clonage

- La classe dont on va cloner les instances doit
  - implémenter l'interface `Cloneable`
  - redéfinir la méthode `clone()` en effectuant une copie profonde si on le souhaite
  - rendre `public` la méthode `clone()` (si on veut l'utiliser depuis n'importe quelle classe)
  - lancer une exception `CloneNotSupportedException` au cas où la copie profonde rencontrerait un objet non clonable

Écriture de la méthode `clone()`

- Quand on redéfinit `clone()` pour une classe `C1`, on doit commencer par appeler la méthode `clone()` de la classe mère par `super.clone()` pour créer le clone de base (par remontée jusqu'à la méthode `clone()` de la classe `Object`)
- On ne doit pas créer l'objet clone par « `new C1()` » ; sinon, une sous-classe `C2` de `C1` qui voudrait utiliser cette nouvelle méthode `clone()` produirait une instance de la classe `C1` et pas de la classe `C2`

Type retour de `clone`

- Depuis le JDK 5, le type retour d'une méthode peut être un sous-type du type retour de la méthode qui est redéfinie
- On peut en profiter pour que le type retour de la méthode `clone` soit la classe dans laquelle elle est redéfinie

Exemple : cloner une facture

```
public class Facture implements Cloneable {
  private ArrayList<Ligne> lignesFact =
    new ArrayList<Ligne>();
  . . .
  public Facture clone()
      throws CloneNotSupportedException {
    Facture fact2 = (Facture)super.clone();
    fact2.lignesFac = (ArrayList)lignesFact.clone();
    // Clonage des lignes de facture
    // (mais pas des articles)
    . . .
    return fact2;
  }
. . .
}
```


Méthode finalize()

Fonctions d’une méthode `finalize()`

- Avant la suppression d’un objet par le ramasse-miettes, le système appelle la méthode `finalize()` de la classe de l’objet (si elle existe)
- On peut ainsi libérer des ressources système utilisées par un objet qui va être supprimé
- Le ramasse-miette ne fonctionne qu’aléatoirement (ou même jamais), on évitera donc d’utiliser `finalize()` pour libérer *rapidement* des ressources

Méthode `finalize()` de `Object`

- La classe `Object` définit une méthode `finalize()` qui ne fait rien
- Les classes qui ont des instances qui doivent libérer des ressources avant d’être supprimées doivent donc redéfinir la méthode `finalize()`

Écriture d’une méthode `finalize()`

- La dernière instruction d’une méthode `finalize()` devrait être toujours « `super.finalize()` » pour libérer des ressources éventuellement acquises par des méthodes héritées de la classe mère

ShutdownHook

Présentation

- Il est possible d’indiquer à la JVM un ou plusieurs traitements (« *traitements de fin* ») à exécuter juste avant de stopper son exécution (cf commande trap des shells Unix)
- Ces traitements seront exécutés en parallèle que la JVM termine son exécution normalement ou qu’elle se termine brutalement parce que l’utilisateur a appuyé sur Ctrl-C

Traitements de fin d’une JVM

- Juste avant de stopper son exécution une JVM
  1. exécute en parallèle tous les traitements de fin (*shutdown hooks*)
  2. exécute tous les *finalizers* non encore exécutés

Traitement de fin

- Un tel traitement est une classe fille de **`Thread`** (voir cours sur les *threads*)
- Le programmeur met dans la méthode **`run`** le traitement qui sera exécuté juste avant l'arrêt de la JVM
- On peut, par exemple, y mettre la suppression d'un fichier temporaire ou la fermeture d'un fichier ou d'un *socket*

Ajout d'un traitement de fin

```
TraitementFin tf = new TraitementFin();
Runtime.getRuntime()
       .addShutdownHook(tf);
```

```
class TraitementFin extends Thread {
  public void run() {
    // Ce qui sera exécuté
    . . .
  }
}
```