POO-IG

# Programmation Orientée Objet et Interfaces Graphiques

Cristina Sirangelo
IRIF, Université Paris Cité
cristina@irif.fr

Exemples et matériel empruntés :
* Transparents de cours de H.Fauconnier
* Core Java - C.Horstmann - Prentice Hall Ed.

# Classes internes et expressions lambda

# Classes internes (“inner classes”)

# Classes internes (inner classes)

- Classes définies dans d'autres classes

- Different types
  - Classes membres
    - membres (possiblement statiques) d'une autre classe
  - Classes locales
    - classes définies dans un bloc de code
    - Classes anonymes
      - classes locales sans nom

# Classe membre non-statique

- Membre non statique d'une classe englobante

```
public class TextDocument {
    private String[] pages;
    private int size = 0;
    ...
    private class Cursor implements Iterator<String> {
        public final int step;
        private int nextIndex;
        …
    }
}
```

- Mêmes modificateurs d'accès que tous les membres de classe
  - `private`, `protected`, (package), `public`
- Remarque : cela n'introduit pas automatiquement un champ de type `Cursor` dans la classe englobante !

# Classe membre non-statique : exemple

- Un curseur sur un document de texte est un objet qui permet de parcourir le document, page après page
  - maintient un index "marque-page" (`nextIndex`)
  - le marque-page peut être avancé d'un certain nombre de pages à la fois (`step` : pas d'avancement)
  - Un curseur doit toujours être attaché à un document

```
public class TextDocument {
    private String[] pages;
    private int size = 0;
    ...

    private class Cursor implements Iterator<String> {
        public final int step;
        private int nextIndex;
        …
    }
```

# Classe membre non-statique

Utilité :

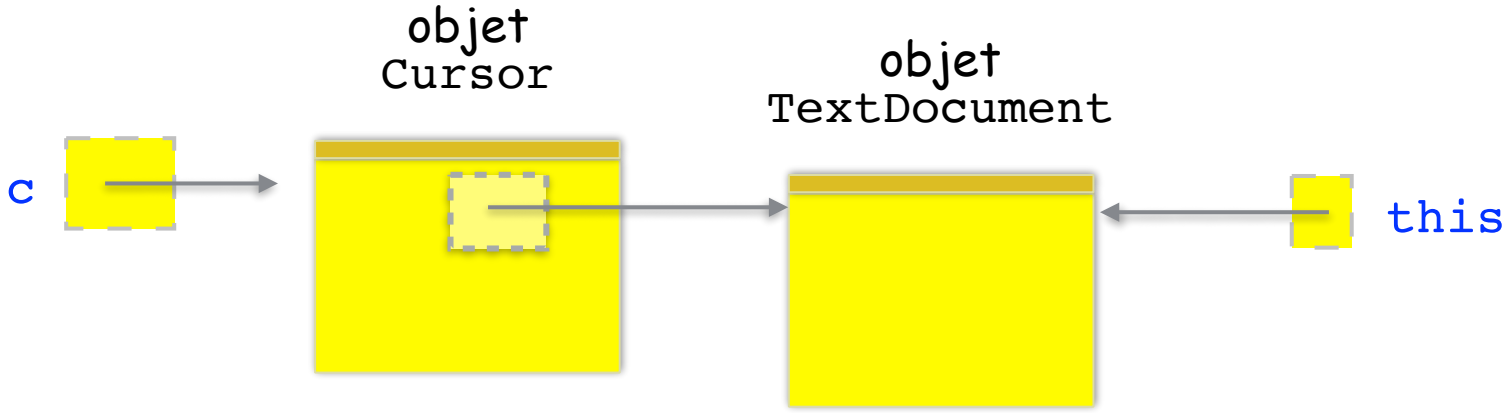
- Définir des **classes qui ont du sens uniquement en relation à la classe englobante**
  - Ex. un Curseur qui parcourt les pages a du sens seulement si attaché à un TextDocument

- En plus, quand la classe membre est `private` : **limiter la visibilité d'une classe à une seule autre classe**
  - Ex. Un Curseur est utilisable uniquement par un TextDocument

# Classe membre non-statique : objet englobant

- **Un objet de la classe membre peut être créé uniquement à partir d'un objet de la classe englobante**
- et il en possède automatiquement une référence implicite

```
public class TextDocument {
   …
   public void print() {
     Cursor c = this.new Cursor(); //"this." optionnel
     printPages (c);
   }
   …
}
```



# Classe membre non-statique : objet englobant

- L'objet de la class englobante est donc accessible depuis l'objet de la classe interne
- Dans la classe interne la référence implicite à l'objet de classe englobante est

*NomClasseEnglobante*.this

  - mais elle peut être omise en général (sauf si occultée)

```
public class TextDocument {
   private String[] pages;  private int size = 0;
    …
   private class Cursor implements Iterator<String> {
       public String next() {
          String retPage = pages[nextIndex];
          //équivalent à TextDocument.this.pages[nextIndex]
          …
       }
    }
    …
}
```

# Classe membre non-statique : exemples

- Dans les exemples on va implémenter l'interface Iterator de `java.util`

```
public interface Iterator<E> {
    boolean hasNext();
    E next() throws NoSuchElementException;
    void remove()throws UnsupportedOperationException,
                        IllegalStateException;
}
```

code/poo/inner/TextDocument.java

code/poo/inner/CompteBancaire.java

# Classe membre non-statique : membres

- pas de membres statiques

```
public class TextDocument {
   private String[] pages;  private int size = 0;
    …
   private class Cursor implements Iterator<String> {
     public static void f() {…};
     public static int g = 0;
     …
    }
    …
}
```

# Classe membre non-statique : contrôle d'accès

- La classe membre a accès aux autres membres de la classe englobante indépendamment de leur modificateur d'accès :

```
public class TextDocument {
    private String[] pages;
    …
    private class Cursor implements Iterator<String> {
        public String next() {
            String retPage = pages[nextIndex];
            // accès possible à pages même si private
        }
    }
    …
}
```

# Classe membre non-statique : contrôle d'accès

- La classe englobante a accès à ses classes membre, ainsi qu'à leur champs et méthodes, indépendamment de leur modificateurs d'accès

```
public class TextDocument {
    …
    public void print() {
        Cursor c = this.new Cursor();
        //acces possible à Cursor même si private
        printPages (c);
    }
   private class Cursor implements Iterator<String> {
    …
    }
    …
}
```

# Classe membre non-statique : contrôle d'accès

- Exemple

```
package poo.inner;
public class A {
    private int a = 0; private B x = new B(); // B et D et leurs
    membres sont visibles dans A…
    public int a1 = x.b1;
    //…ainsi que dans tous les membres de A  (méthodes, classes
    internes...)
    private class B {
        private int b1 = a; //a (private) est visible dans B
        protected int b2 = 2; public int b3 = 1;
    }
    public class D {
        private int z;
        protected int w = 2;
        public D() {
            z = x.b1; z = x.b2; z = x.b3; // x et tous ses champs
                                  //(même private) sont visibles dans D
        }
    }
}
```

# Classe membre non-statique : contrôle d'accès

- Une classe fille d'une classe membre (bien qu'interne à la même classe englobante) maintient les règles habituelles de visibilité **des champs hérités**

```
package poo.inner;
public class F {
    private int a = 0; private G x = new G();
    public int a1 = x.b1;
    private class G {
      private int b1 = a; protected int b2 = 2; public int b3 = 1;
    }
   public class H extends G {
       public int c1 = x.b1; //x.b1 (private) est visible dans H
       private int c2 = 0;
       public H() {
         c1 = b2 + b3; // b2 et b3 sont visibles dans H…
         c1 = b1; //…mais pas b1 (erreur)
        c1 = super.b1; // OK!
       }
    }
}
```

# Classe membre non-statique : contrôle d'accès

- le modificateur d'accès d'une classe membre et des ses champs et méthodes règle uniquement leur visibilité en dehors de la classe englobante

```
package poo.inner;
class E {
 //voit la classe interne H de F, mais pas G
}
```

# Classe membre non-statique : contrôle d'accès

- Si une classe membre est visible en dehors de sa classe englobante
  - elle est dénotée :

*NomClasseEnglobante.NomClasseMembre;*

  - on peut créer ses objets à partir de n'importe quel objet de la classe englobante comme suit :

```
public class F {
      public class H extends G {
        …
      }
}

  …
  F a = new F();
  F.H x = a.new H();
```

# Classe membre non-statique : contrôle d'accès

- Exemple

```
package poo.inner;

public class F {
   private class G {
      private int b1 = a; protected int b2 = 2; public int b3 = 1;
   }
   public class H extends G {private int c2 = 0;…}
}
class E {
   //voit la classe interne H de F, mais pas G
    F a = new F();
    F.H d = a.new H(); // F.H est visible dans E
    F.G x = a.new G(); // F.G n'est pas visible dans E (erreur)
    public E() {
        int z = d.b2; // d.b2 (protected) est visible dans E
        z = d.c2; //mais pas d.c2 (private) - erreur
    }
}
```

# Classe membre non-statique : classe ou interface

- Une interface aussi peut être membre
- Mais une interface membre est considérée comme un membre static de la classe englobante (static implicite)

```
package poo.inner;
public class A {
    public interface I {
        int i = 0;
        default void f(){}
    };
    public class D implements I {
        …
    }
}
```

- Donc cela n'a pas de sens de parler d'objet englobant pour une interface membre

# Classe membre statique

- Classe membre statique d'une autre classe
  - classe ou interface
  - mot clé static
  - similaire aux champs ou méthodes statiques: n'est pas associée à une instance et donc a accès uniquement aux champs statiques de la classe englobante
  - En particulier un objet d'un classe membre static ne reçoit pas de référence à un objet englobant
  - même règles de contrôle d'accès qu'avec les classes membre non-statiques
    - mais pas d'accès à *`NomClasseEnglobante`*`.this`

# Classe membre statique

```
package poo.inner;

public class A {
    private static int c = 1;
    private int a = 0; …
    public static class B {
        int x = a; //erreur, pas d'acces à A.this
        int x = c; // OK, c est static
    }
}

…
//dans une méthode d'une autre classe
A.B f = new A.B()
```

# Classe membre statique : exemple

```
package poo.inner;
class PileChainee{
        public static abstract class Noeud{
           private Noeud next;
           public Noeud getSuivant(){ return next; }
           public void setSuivant(Noeud n) { next=n; }
        }
        private Noeud tete;
        public boolean estVide(){ return tete == null; }
        public void empiler(Noeud n){
            n.setSuivant(tete); tete=n;
        }
        public Noeud depiler(){
            Noeud tmp;
            if (!estVide()){
                tmp=tete;
                tete=tete.getSuivant();
                return tmp;
            }
            else return null;
        }
}
```

# Classe membre statique : exemple (suite)

```
package poo.inner;
class NoeudEntier extends PileChainee.Noeud {
    private int i;
    public NoeudEntier(int i){ this.i = i; }
    public int val(){return i;}
}
```

```
public static void main(String[] args) {
    PileChainee p = new PileChainee();
    NoeudEntier n;
    for(int i=0; i < 12;i++){
        n = new NoeudEntier(i);
        p.empiler(n);
    }
    while (!p.estVide()){
       System.out.println(
              ((NoeudEntier)(p.depiler())).val());
    }
}
```

# Classe membre statique : remarques

- Noter l'usage du nom hiérarchique avec '.'

```
PileChainee.Noeud
```

- On peut l'éviter avec un `import` :

```
import poo.inner.PileChainee.Noeud;

class NoeudEntier extends Noeud {
    ...
}
```

# Classe membre statique : utilité

- Définir des classe /interfaces dont les objets sont à utiliser en relation avec les objets d'une classe englobante, sans nécessiter une reference à un objet englobant
  - E.g : un objet de type `PileChainee.Noeud` est à utiliser en tant que noeud d'une `PileChainee`
  - mais il ne nécessite pas une référence à la pile

- Regrouper les classes dans une hiérarchie logique, avec répétition possible des noms
  - E.g. `PileChainee.Noeud, Liste.Noeud, ListeDouble.Noeud`

- Avec modificateur `private` : définir une classe qui est utilisée dans un seule autre classe - et ne nécessite pas de référence à un objet englobant

# Classe membre statique : accès

```
package poo.inner;
public class A {
    private static int h = 1; private int a = 0; …
    private class B {
        private int b1 = a; protected int b2 = 2;
        public int b3 = 1;
    }
    public class D {…}
    private static class F {
        A e = new A();
        B b = e.new B(); // B (private) est visible dans F
        int y = b.b1 ; // ainsi que ses champs
        int w = a; //erreur, pas d'acces à A.this
         int w = h; // OK, h est static
    }
    public static class G {…}
}
```

# Classe membre statique : accès

```
package poo.inner;

public class E {
    A a = new A();
    A.F f = new A.F(); //erreur : A.F privée
    A.G g = new A.G(); // OK : A.G publique
    A.D d = a.new D() ; //remarquer la difference avec A.G
}
```

# Classe membre et héritage

- Façons d'étendre une classe membre (statique ou pas)

1)

```
class Externe{
    class Interne {}
    class InterneEtendue extends Interne{}
}
```

# Classe membre et héritage

- Façons d'étendre une classe membre (statique ou pas)

2)

```
class Externe{
    class Interne {}
}
class ExterneEtendue extends Externe{
    // Interne est aussi une classe interne de
    // ExterneEtendue
    // elle est héritée comme tous les autres membres
    // elle peut donc être étendue ici
    class InterneEtendue extends Interne{…}
    Interne r = new InterneEtendue();
}
```

# Classe membre et héritage

3) Une classe interne peut également être étendue en dehors de sa classe englobante

- cas de classe interne statique

```
class Externe{
    static class InterneStatique {
          int i;
          InterneStatique (int i) { this.i = i;}
    }
}

class Autre extends Externe.InterneStatique {
    Autre(int i){ super(i);}
}
```

pas besoin d'un objet `Externe` pour créer un `InterneStatique`

# Classe membre et héritage

3) Une classe interne peut également être étendue en dehors de sa classe englobante

- cas `Interne` non-statique : pour cela il faut lui associer un objet `Externe`

```
class Externe{
    class Interne {
        int i;
        Interne (int i) { this.i = i;}
    }
}

class Autre extends Externe.Interne {
    Autre(Externe r, int i){ r.super(i);}
}
```

  - `r.super(i)` invoque le constructeur de `Externe.Interne` à partir de l'objet `r` (qui sera donc l'objet englobant de l'objet `Autre`)
  - Raison : un objet `Interne` (ou d'une de ses extensions) n'a de sens qu'attaché à un objet `Externe`

# Classe membre et occultation

- Le champs des classes internes et les paramètres peuvent occulter les champs du même nom de la classe externe. Mais il est possible d'y accéder

```
public class ShadowTest {
    public int x = 0;
    class FirstLevel {
          public int x = 1;
          void methodInFirstLevel(int x) {
              System.out.println("x = " + x);
              System.out.println("this.x = " + this.x);
              System.out.println("ShadowTest.this.x = " +
                                          ShadowTest.this.x);
          }
     }
    public static void main(String... args) {
          ShadowTest st = new ShadowTest();
          ShadowTest.FirstLevel fl = st.new FirstLevel();
          fl.methodInFirstLevel(23); // -> 23 1 0
     }
}
```

# Classe membre et occultation

- Idem pour l'occultation des méthodes

```
class H{
    void print(){System.out.println ("print() de H");}
    void print(int i){
          System.out.println ("print("+i+") de H”);
     }
    class I {
        void print(){System.out.println ("print() de I");}
        void show(){
            print(); //print() de I
            H.this.print(); // print() de H
            H.this.print(1); //print (1) de H
            //H.this nécessaire aussi dans le dernier cas :
            // tous les print sont occultés
        }
    }
}
```