POO-IG

Programmation Orientée Objet et Interfaces Graphiques

Cristina Sirangelo
IRIF, Université Paris Diderot
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 :

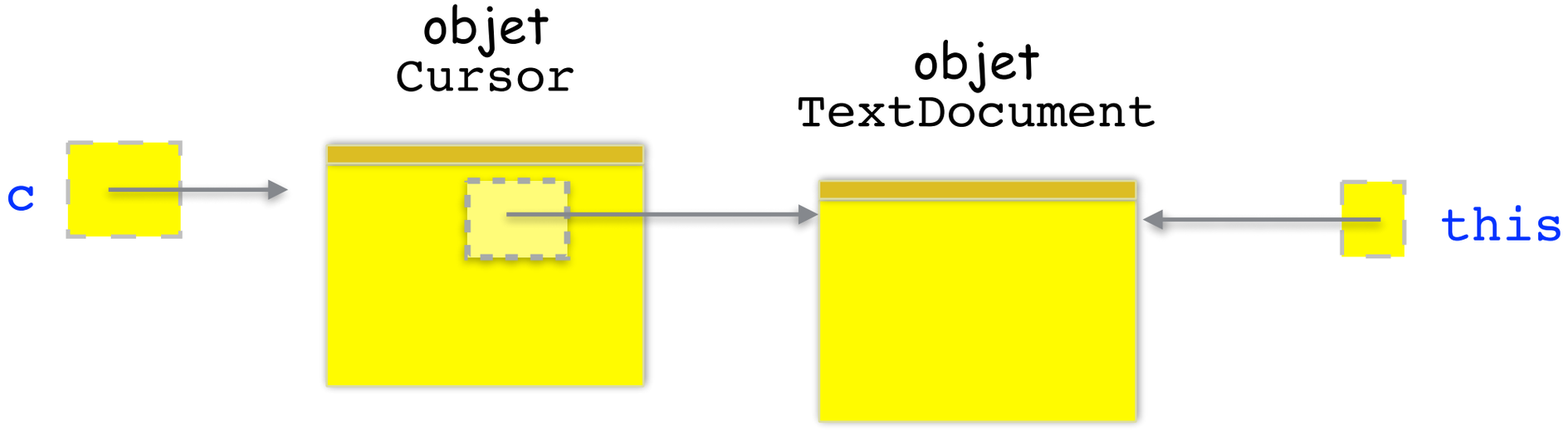
- Définir des **classes qui ont du sens uniquement en relation à la classe englobante**
  - Ex. un Curseur qui parcourt les pages a du sens seulement si attaché à un TextDocument

- 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

- Une classe interne ne peut pas avoir 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 meme 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 que 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 :

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

Classe membre non-statique : contrôle d'accès

- Exemple

```
package poo.inner;
class E {
 //voit la classe interne H de F, mais pas G
    F a = new F();
    F.G x = a.new G(); // F.G n'est pas visible dans E (erreur)
    F.H d = a.new H(); // mais C est visible
    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 non statique
- mais cela n'a pas de sens de parler d'objet englobant pour une interface membre

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

- Une interface membre est considérée comme un membre static de la classe englobante (static implicite)

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 associé à une instance et donc 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 : exemple

```
package poo.inner;
class PileChainee{
        public static interface Chainable{
                public Chainable getSuivant();
                public void setSuivant(Chainable noeud);
        }
        Chainable tete;
        public boolean estVide(){ return tete == null; }
        public void empiler(Chainable n){
            n.setSuivant(tete); tete=n;
        }
        public Object depiler(){
            Chainable tmp;
            if (!estVide()){
                tmp=tete;
                tete=tete.getSuivant();
                return tmp;
            }
            else return null;
        }
}
```

Classe membre statique : exemple (suite)

```
package poo.inner;
class EntierChainable implements PileChainee.Chainable {
    int i; PileChainee.Chainable next;
    public EntierChainable(int i){ this.i = i; }
    public PileChainee.Chainable getSuivant(){ return next; }
    public void setSuivant(PileChainee.Chainable n) { next=n; }
    public int val(){return i;}
}
```

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

Classe membre statique : remarques

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

`PileChainee.Chainable`

- On peut l'éviter avec un `import` :

```
import poo.inner.PileChainee.Chainable;

class EntierChainable implements Chainable {
    int i; Chainable next;
    public EntierChainable(int i){ this.i = i; }
    public Chainable getSuivant(){ return next; }
    public void setSuivant(Chainable n) { next=n; }
    public int val(){return i;}
}
```

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.Chainable` 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.Chainable, Liste.Chainable, ListeDouble.Chainable`

- 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

1)

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

Classe membre et héritage

- Façons d'étendre une classe membre

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

- Façons d'étendre une classe membre

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

3) `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 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 héritage

- Façons d'étendre une classe membre

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

3) `Interne` peut également être étendue en dehors de sa classe englobante

- cas de classe interne statique

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

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

Classe membre et occultation

- Le champs des classes internes et les paramètres peuvent occulter les champs de la classe externe du même nom. 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
        }
    }
}
```

Classes locales

- Classes définies à l'intérieur d'un bloc de code (par exemple méthode)
- analogue à des variables locales:
  - une classe interne locale n'est pas membre de la classe externe
  - elle est visible uniquement dans le bloc de code dans laquelle elle est définie
- usage:
  - créer des objets d'une classe qui n'a de sens que localement dans un bloc de code (e.g. dans les interfaces graphiques)
  - en particulier définir des extensions de classes qui n'ont du sens que localement

Classe locale : exemple

- Classes Collections (ou Containers) : classes correspondant à des structures de données
  - exemples: List, Set, Queue, Map.

- L'interface Iterator est souvent utilisée pour parcourir tous les éléments composant une structure de données

Classe locale : exemple (suite)

```
class MaCollection {
    Object[] data;
    MaCollection(int i){
        data=new Object[i];
    }
    MaCollection(Object ... l){
        data=new Object[l.length];
        for(int i = 0; i < l.length; i++)
            data[i] = l[i];
    }
    public Iterator<Object> parcourir(boolean up){
        class Iter implements Iterator<Object>{…}
        return new Iter();
    }
}
```

Classe locale : exemple (suite)

```
public class MaCollectionTest {
    public static void afficher(Iterator it){
        while(it.hasNext()){
            System.out.println(it.next());
        }
    }
    public static void main(String[] args) {
        MaCollection m = new MaCollection(1,2,3,5,6,7);
        afficher(m.parcourir(true));
        afficher(m.parcourir(false));
    }
}
```

Classe locale : accès aux variables locales

- Comment est définie la classe `Iter` dans `parcourir()`?

- Une classe locale a accès :
  - aux membres de la classe englobante
  - aux variables locales visibles dans le bloc de code englobant
    - à condition que ces variables locales soit "effectively final"
      - effectively final : variable jamais modifiée après sa création (même pas par la classe locale)
      - (`final` explicite nécessaire avant Java8)

Classe locale : exemple (suite)

```
class MaCollection {
   Object[] data;
   …
   public Iterator<Object> parcourir(boolean up){
        class Iter implements Iterator<Object> {
            private int pos = up? 0 : data.length-1;
            public boolean hasNext(){
                return (up? pos < data.length : pos >= 0);
            }
            public Object next() throws NoSuchElementException{
               if (up && pos >= data.length)
                        throw new NoSuchElementException();
               if (!up && pos < 0) throw new NoSuchElementException();
               Object ret = data[pos];
               pos = up? pos+1 : pos-1;
               return ret;
            }
            public void remove(){
                throw new  UnsupportedOperationException();
            }
        }
        return new Iter();
    }
}
```

Classe locale : exemple - remarques

- `Iter` peut accéder à `up` et `data`
  - `data[]` : champ de la classe englobante
  - `up` : variable locale de la methode `parcourir`

- `up` est effectively final : il y aurait une erreur de compilation si up était modifiée après sa création
- `data[]` n'est pas effectively final, cela n'est pas nécessaire pour le membres de la classe englobante

- Raison de la restriction "effectively final" :
  - une copie de la variable locale `up` est passée implicitement par le compilateur au constructeur de la classe `Iter`
  - La classe `Iter` aura un champ implicite `up` initialisé à cette valeur, pour pouvoir y accéder même après que `parcourir()` a terminé (et la variable locale `up` est détruite)

Classe anonyme

- Mais était-il utile de donner un nom à cette classe qui ne sert qu'à créer **un seul objet** `Iter`?

Classe anonyme

```
public Iterator<Object> parcourir(boolean up){
  return new Iterator<Object> (){
     private int pos = up? 0 : data.length-1;
     public boolean hasNext(){
             return (up? pos < data.length : pos >= 0);
     }
     public Object next() throws NoSuchElementException{
        if (up && pos >= data.length)
                    throw new NoSuchElementException();
        if (!up && pos < 0) throw new NoSuchElementException();
        Object ret = data[pos];
        pos = up? pos+1 : pos-1;
        return ret;
     }
     public void remove(){
        throw new  UnsupportedOperationException();
     }
  }
}
```

Classe anonyme

Créer un objet d'une classe anonyme qui hérite d'une classe/ interface *TypeDeBase*

```
new TypeDeBase (paramètres du constructeur de base) {
   champs et méthodes additionnelles de la classe anonyme
};
```

Classe anonyme : exemple

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

}

class AnonymousTest {
 public static void main (String args[]) {
       TypeDeBase obj = new TypeDeBase (3) {
          int j = 1;
          public String toString () {
               return i + " " + j;
          }
       };

    System.out.println(obj); // 3 1
  }
}
```

Classe anonyme : initialisation

- Une classe anonyme ne peut pas avoir de constructeur puisqu'elle n'a pas de nom
  - initialisation de la classe mère : par new
  - initialisations des champs additionnels : par les initialisateurs ou dans les blocs d'initialisation

```
TypeDeBase obj = new TypeDeBase (3) {
   int j = 1;
   ...
}
```

Expressions lambda

- Introduites en Java 8
- Définissent un bloc de code qui peut être affecté / passé en paramètre, pour être exécuté plus tard
  - exemple typique : dans les interfaces graphiques
    - e.g. on construit un objet "bouton" et on lui passe en paramètre la fonction qui doit être exécutée quand le bouton sera cliqué par l'utilisateur
- Les expressions lambda se rapprochent de la notion de passage des fonctions en paramètre qui existe dans les langages fonctionnels

- Il a toujours été possible en Java de passer du code en paramètre en créant expressément un objet qui contient ce code dans une méthode
- Mais cela est lourd et artificiel (même en utilisant des classes anonymes…)
- Les expression Lambda ont été introduites pour simplifier et rendre plus directe cette tâche

Expressions lambda : définition

- Une expression lambda est un bloc de code, précédé par la liste de tous les paramètres utilisés dans ce code

```
(int k) -> { for (int i = k; i >= 0; i--)
                    System.out.println(i); }
```

e.g. l'expression ci dessus définit une fonction qui prend un paramètre `k` de type `int` et affiche tous les entiers de `k` à `0`

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

définit une fonction qui prends deux paramètres (`first` et `second`) de type `String` et renvoie leur difference de taille

- Si le type des paramètres est déterminé par le contexte, il peut être omis (voir plus loin)

Interfaces fonctionnelles

- Le interfaces possédant **une seule méthode abstraite** sont appelées fonctionnelles
- La bibliothèque Java en possède plusieurs
  - `Comparator<T>` avec méthode `int compare(T o1, T o2);`
    censée renvoyer `>0,0,<0` si `o1<o2, o1==o2, o1>o2,` resp.
  - `ActionListener` avec méthode
    `void actionPerformed(ActionEvent e)`
    invoquée quand un événement a lieu (`e` est l'événement à traiter)
  - `Function<T,R>` avec méthode `R apply (T t)`
    censé implementer une fonction arbitraire à un seul paramètre
  - `Predicate<T>` avec méthode `boolean test(T t)`
    censée implementer une fonction booléenne arbitraire à un seul paramètre
  - `IntConsumer` avec méthode `void accept(int i)`
  - `IntSupplier` avec méthode `int getAsInt()`
- etc. plusieurs interfaces fonctionnelles génériques dans `java.util.funtion`

Expressions lambda et interfaces fonctionnelles

- Une expression Lambda peut être affectée à une variable de type interface fonctionnelle;
- l'expression lambda elle fournit une definition pour la méthode abstraite de l'interface. Exemples :

1)
```
Comparator<String> c = (String first, String second) ->
                          first.length() - second.length();
```

ou

```
Comparator<String> c = (first, second) ->
                          first.length() - second.length();
```

Ensuite `c.compare ("abc", "a");` applique la fonction passée et renvoie la difference de taille (2)

2)
```
Predicate<Point> positif = (Point p) ->
                            p.getX() > 0 && p.getY() > 0;
```

Ensuite `positif.test (new Point(2,3))` renvoie true et `positif.test (new Point(-1,2))` renvoie faux

Expressions lambda et interfaces fonctionnelles

- Plus en général : une expression lambda peut être utilisée à chaque fois qu'on s'attend un objet de type `I`, où `I` est une interface fonctionnelle dont la méthode abstraite est compatible avec l'expression lambda

Expressions lambda et interfaces fonctionnelles

Exemple 1

- Methode `Arrays.sort()`
  - paramètres : un tableau et un objet de type `Comparator`
  - trie le tableau selon le critère de comparaison fourni

```
String[] t = {“Rome”, “Barcelone”, “Paris”};
Arrays.sort ( t, (first, second) ->
                    first.length() - second.length());
// nouvelle valeur de t : {“Rome”, “Paris”, “Barcelone”};
```

Expressions lambda et interfaces fonctionnelles

Exemple 2

- La classe `ArrayList` possède une méthode `removeIf`
  - paramètre : un `Predicate` pour tester les valeurs à supprimer

```
ArrayList<Point> l = new ArrayList<Point>();
l.add (new Point(2,3)); l.add (new Point(-1,2));
// l est la liste [<2,3>, <-1,2>]

l.removeIf (p -> p.getX() > 0 && p.getY() > 0);

// nouvelle valeur de l : [<-1,2>];
```

Expressions lambda et interfaces fonctionnelles

Exemple 3

```
//réitère une action n fois, l'action depend de l'iteration i
public static void repeat(int n, IntConsumer action) {
   for (int i = 0; i < n; i++) action.accept(i);
}

//on lui passe un nombre d'iterations et une action à répéter
repeat(8, i -> {
            for (int j = 0; j <= i; j++) System.out.print("* ");
            System.out.println();
           }
      );
//affiche un triangle de * de hauteur 8
```

Expressions lambda et interfaces fonctionnelles

Exemple 4

- L'interface `ActionListener` avec méthode
  `void actionPerformed(ActionEvent e)`
  décrit des objets qui "écoutent" et réagissent à des événement

```
ActionListener x = event -> System.out.println(event);
```

La méthode `actionPerformed` de `x` sera la fonction

```
event ->  System.out.println(event)
```

Ensuite `x` peut être associé par ex. à un bouton de l'interface graphique,

```
jButton1.addActionListener(x);
```

- à chaque fois que ce bouton est pressé, un événement sera généré
- en reaction à l'événement `actionPerformed` de `x` sera invoqué
- effet : l'événement sera affiché sur la console

Expressions lambda et interfaces fonctionnelles

Exemple 4 (suite)

- Ou de façon plus compacte :

```
jButton1.addActionListener(event ->
                    System.out.println(event));
```

- Remarquer la différence avec l'idiome utilisé avant Java8 (avec classes anonymes) :

```
jButton1.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent event){
            System.out.println(event);
    }
});
```

Beaucoup moins compacte et moins lisible !

References de méthodes

- Parfois la fonction qu'on veut passer en paramètre existe déjà.
- Dans ce cas au lieu de définir une nouvelle expression lambda, on passe une reference à la méthode existante

`nomClasse :: nomMethode`

est une référence à la méthode `nomMethode` (d'instance ou statique) de la classe `nomClasse`

`o :: nomMethode`

est une référence à la méthode de l'objet o

- Exemples
  - `System.out::println <=> s -> System.out.println (s)`
  - `Math::pow <=> x -> Math.pow(x)`
  - `String::concat <=> (x,y) -> x.concat (y)`

References de méthodes

- Référence de constructeur
  - *NomClasse*`::new` est une reference à un constructeur de `NomClasse`
    - celui compatible avec la signature de l'interface fonctionnelle

```
class Point {
    Point(){…};
    Point (double x, double y) {…}
    …
};
@FunctionalInterface
interface I {
        Point f (double a, double b);
}
…
I i = Point::new;
```

Affecte à `i` une instance de type `I` ayant `f` définie comme le constructeur `Point(double, double)`

Visibilité des variables

- Comme une classe locale, une expression lambda a accès :
  - à tous les champs et méthodes de la classes englobante, indépendamment de leur modificateur d'accès
  - à toute variable locale visible dans le bloc de code englobant, qui soit "**effectively final**"
    - phénomène appelé **capture** des variables locales
    - les variables locales utilisés sont aussi appelées **variables libres** d'une expression lambda
  - Exemple

```
public class LambdaTest {
 private String bj = "Bonjour ";
 public void repeatMessage(String text, int delay) {
   ActionListener l = event -> System.out.println(bj + text);
   new Timer(delay, l).start();
 }
 …
 repeatMessage(":-)", 500);
//affiche Bonjour :-) sur une ligne tous les 500ms
}
```

Expressions lambda vs classes locales

- Même règles d'accès aux variables que les classes locales / anonymes !
- De faitJava implémente les lambda expr. en créant une instance `l` d'une classe anonyme qui implémente l'interface fonctionnelle (e.g. `ActionListener`)
- Cette instance (comme l'on sait) aura comme champs implicites des copies des variables locales qu'elle utilise (e.g. `text`)
- Différence avec les classes locales / anonymes :
  - `this` dans une expression lambda fait reference à l'objet de la classe englobante (e.g. `LambdaTest`)
  - et non pas à l'objet de l'interface fonctionnelle (e.g. `ActionListener`) implicitement crée