Programmation modulaire, bibliothèque de conteneurs

30/09/09

Bibliothèques

- Quand on a un projet à développer,
  - on ne réécrit pas tout, on n'écrit pas non plus tout d'un coup !

- Structurer le programme en « module » + Utiliser des modules existants
  - soit qu'on a déjà développé avant, (Complexe et Pile de Complexe)
  - soit qui existe ailleurs (String, Scanner)

- Bibliothèques *libraries* (classiques distribuées avec les langages OO)
  - librairies graphiques, librairies de conteneurs, librairies d'accès à une base de données...

  - c'est déjà fait (se concentrer sur les vraies difficultés du programme, gain de temps !)
  - c’est bien fait (performances)
  - c'est utilisé (par plein de gens dans plein de contextes différents et donc beaucoup testé)
  - c’est maintenu

Les conteneurs

Pourquoi ?

- Utilisation d’une bibliothèque de conteneurs
  - = un ensemble de classes
  - Qui fournissent des *structures de données* dont on a toujours besoin
  - Et les *algorithmes* pour les manipuler
- Deux exemples :
  - listes d’objets
  - Dictionnaires (ex: annuaire téléphonique associant des numéros et des noms)
- Avantages:
  - Programmer rapidement et proprement
  - *Performance* des implémentations réalisées
  - Réutilisation du logiciel
  - Maintenance facilitée car *API standard*
  - Permet de réfléchir à la conception du logiciel plutôt qu’à son implémentation

Les conteneurs : principe

Structures de données classiques + principaux algorithmes

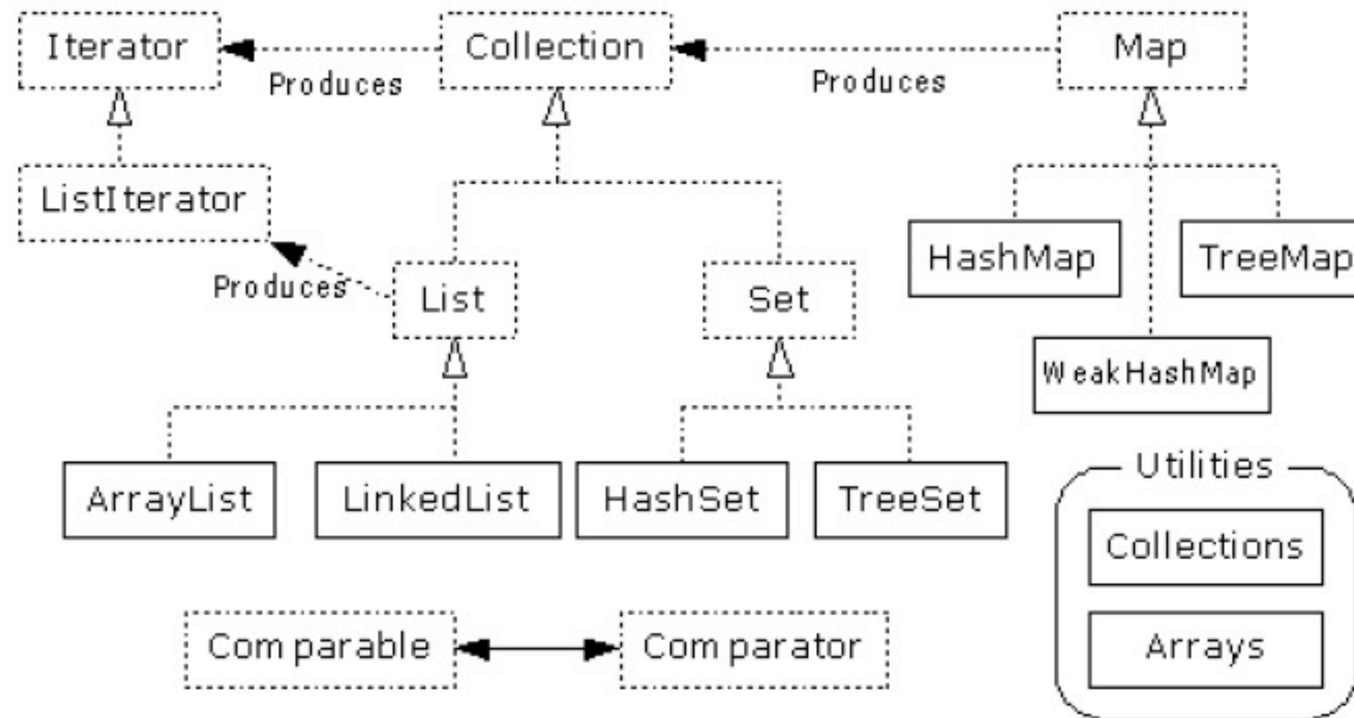
- Collections d 'éléments :
  - Listes : l'ordre des éléments est précis et connu
  - Ensembles : éléments ne peuvent être dupliqués

- Structures associatives ou dictionnaire *Map*
  - Ensemble d'entrées (clé, valeur)

- Dans un conteneur, on stocke : **des objets** (type référence) **de même type**
  - `List<A>` : classe qui représente une liste qui va contenir des objets de type A
  - `<A>` : **généricité** (mais pas la même qu'en Ada ou C++)
  - *Généricité dans ce cours? Uniquement comme utilisation simple via les bibliothèques de conteneurs*

- Méthodes et algorithmes définis
  - Ajout, suppression, appartenance, nombre d'éléments
  - Tri, Recherche …

Classification simplifiée



Classe concrete

Interface

2 catégories de conteneurs

- `Collection<E>`
  - `List<E>` structure séquentielle, l'utilisateur contrôle l'ordre des éléments.
    - `ArrayList<E>` implémentation dans tableau de taille variable, accès par indice
    - `Vector<E>` toujours implémentation dans tableau … (plus vieux)
    - `LinkedList<E>` implémentation par double chaînage et pointeurs
  - `Queue<E>` FIFO, file à priorité
  - `Set<E>` (pas deux éléments identiques)
    - `HashSet<E>` ensemble non ordonné, impléménté par table de hachage
    - `TreeSet<E>` ensemble ordonné implémenté par arbre binaire de recherche équilibré (red black tree)
- `Map<K,V>` ensemble associatif
  - `HashMap<K,V>` les clés sont un ensemble non ordonné
  - `TreeMap<K,V>` : item les clés sont un ensemble ordonné

Quelques méthodes communes aux collections

```
Collection<E>

    boolean isEmpty();
    int size();
    boolean add(E elt);  // vrai s'il est effectivement ajouté
    boolean remove(Object o); // vrai s'il est effectivement retiré
    void clear();
    boolean contains(Object o);
```

- Pour l'instant, utiliser contains et remove avec le profil :
  - `boolean remove(E elt);`
  - `boolean remove(E elt);`
- Méthodes *static* de la classe outil Collections : des algo génériques
  - `void Collections.sort(List<E>);`
  - `E Collections.max(Collection<E>); E Collections.min(Collection<E>);`
  - `void Collections.reverse(List<E>);`
- Aller voir l'API pour plus !!!

Listes : quelques méthodes et implémentations

List<E>

```
boolean add(int index, E elt);
E get(int index);
E remove(int index);
```

- Attention au coût des méthodes selon le type de liste :

LinkedList<E>

- Implantation par une liste doublement chainée
- Accès direct au début et à la fin
- Accès séquentiel aux éléments (attention au coût des fonctions utilisant un indice)
- Méthodes supplémentaires
  - `addFirst, addLast,, getFirst, getLast, removeFisrt, removeLast…`

ArrayList<E>

- Implantation par un tableau
- Accès direct à tous les éléments
- Insertion et suppression obligent à décaler les éléments

Ensembles : implémentations

`Set<E>`

- Rien en plus par rapport aux collections

- Attention au coût des méthodes selon l'implémentation :

`HashSet<E>`

- Implantation par table de hachage
- Opérations en temps constant si peu de collisions

`TreeSet<E>`

- Implantation arbre binaire de recherche équilibré rouge et noir
- Opérations en log
- Méthodes supplémentaires
  - `E first(); E last();`

Associations : méthodes et implémentations

Map<K,V>

```
V put (K key, V value); // si la clé existait déjà, renvoie l'ancien
                        // objet associé,  null sinon
boolean containsKey (Object key);
boolean containsValue (Object value);

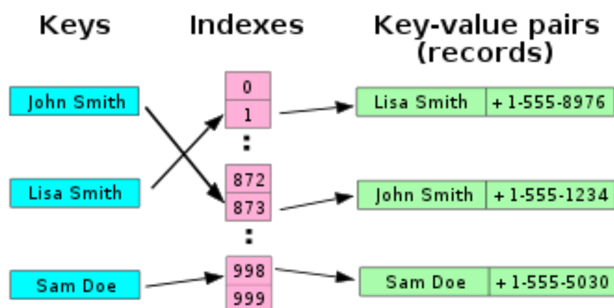
V get (Object key);
V remove (Object key);

int size ();
boolean isEmpty ();
clear ();
```

- Attention au coût des méthodes selon l'implémentation :

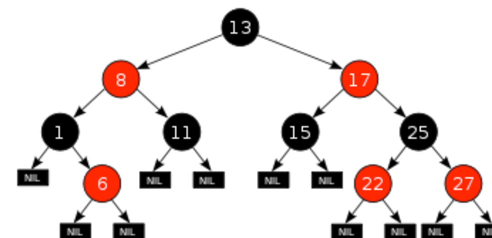
HashMap<K,V>

- Implantation par table de hachage



TreeMap<K,V>

- Implantation arbre binaire de recherche équilibré rouge et noir

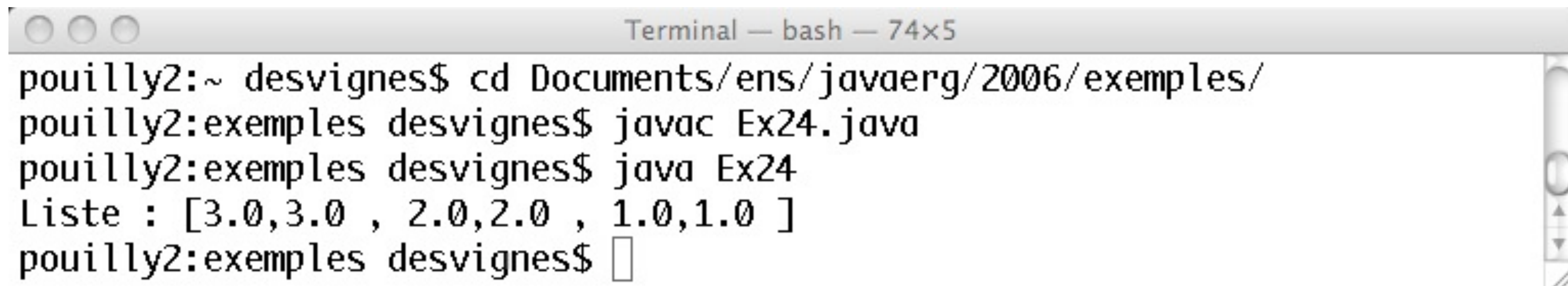


Exemple : liste de points

```
public class Ex24 {
  public static void main (String []arg) {
    LinkedList<Point> liste;           // Définir une liste de point
    liste = new LinkedList<Point>(); // Créer la liste de point vide
    Point a = new Point(3,3);

    liste.add(a);                      // Ajout des points
    liste.add(new Point(2,2));         // dans la liste
    liste.add(new Point(1,1));

    System.out.println("Voici la liste= «  +
                      liste.toString ()); // Affichage de la liste
  }
}
```



- Le programme suivant devrait écrire « Blanc » mais lève une exception. Pourquoi ?

```
import java.util.*;
public class SortMe
{
  public static void main(String args[])
  {
    TreeSet<StringBuffer> s = new TreeSet<StringBuffer>();
    s.add(new StringBuffer("Bleu"));
    s.add(new StringBuffer("Blanc"));
    s.add(new StringBuffer("Rouge"));
    System.out.println(s.first());
  }
}
```

Parcours de collections : itérateurs

- **Itérateur** = outil qui permet de parcourir tous les éléments d'une collection
  - passe d'un élément à un autre pour y accéder et lui appliquer un traitement
  - **en utilisant un code générique de haut niveau**

- En java, pour utiliser un itérateur

  - Déclarer un itérateur : (celui-ci va parcourir des objets de classe Point)

    ```
    Iterator<Point> it;
    ```

  - Créer un itérateur SUR la collection à parcourir : (ici, it va parcourir la liste l)

    ```
    Liste<Point> l = …
    it = col.iterator();
    ```

  - Parcourir la collection :
    - `it.hasNext()` : y a t il encore des éléments à parcourir ?
    - `it.next()` : retourne l'élément suivant dans la collection

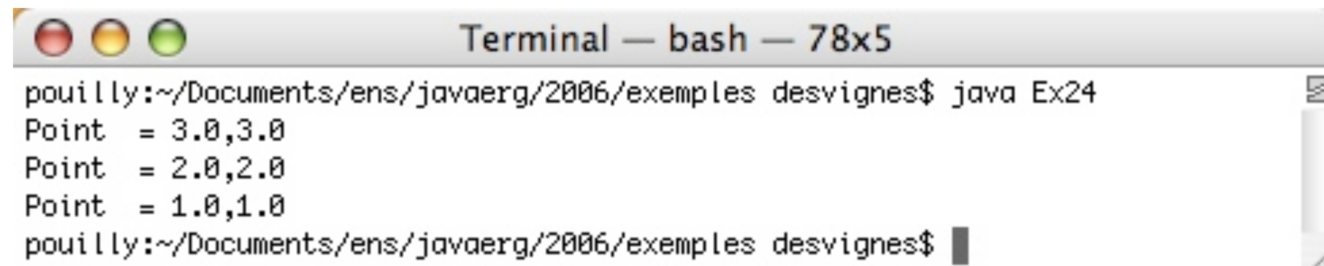
    - ```
      while (it.hasNext ()) {
         Point pointCourant = it.next ();
         ... // traiter pointCourant
      }
      ```
      ```

Exemple : liste de points

```
public class Ex24 {
  public static void main (String []arg) {
    LinkedList<Point> liste;              // Définir la liste de Point
    liste = new LinkedList<Point>(); // Créer la liste de point vide
    Point a = new Point(3,3);

    liste.add(a);                          // Ajout des points
    liste.add(new Point(2,2));             // dans la liste
    liste.add(new Point(1,1));

    Iterator<Point> it = liste.iterator(); // parcours avec itérateur
    while (it.hasNext()) {
       Point p = it.next();
       System.out.println("Point = "  + p);
    }
  }
}
```



Parcourir plus simplement : foreach

- La boucle for permet une écriture plus rapide depuis la version 1.5 de java
- On peut aussi l'utiliser sur les tableaux

```
public class Ex24 {
  public static void main (String []arg) {

    LinkedList<Point> liste;            // Définir la liste de Point
    liste = new LinkedList<Point>(); // Créer la liste de point vide
    Point a = new Point(3,3);

    liste.add(a);                        // Ajout des points
    liste.add(new Point(2,2));           // dans la liste
    liste.add(new Point(1,1));

    // Pour tous les points de la liste
    for (Point b: liste) {
       System.out.println("Point = " + b);
    }
  }
}
```

Parcourir une association (map)

- Les entrées sont des couples clés-valeur (K,V) → différents parcours possibles

- Des méthodes permettent d'obtenir :
  - Ensemble des clés **public Set<K> keySet ();**
  - Ensemble des valeurs **public Collection<classe> values ();**
  - Ensemble des entrées **public Set<Map.Entry<K,V>> entrySet ();**

- **Map.Entry<K,V>**
  - Type des objets contenus dans l'ensemble retourné par entrySet
  - Enregistrement à 2 champs (paire), l'un de type K, l'autre de type V
  - Accès aux champs par **: public K getKey (); public V getValue**

Parcourir une association (map) : exemple

```
// déclaration et création d'une association vide
TreeMap <String, String > map = new TreeMap <String , String >();

// remplissage…
map.put(" rouge ", " ... définition  du mot  rouge ...");
map.put(" vélo ", "... définition  du  mot  vélo ...");
map.put(" artichaut ", " ... définition  du mot  artichaut ... ");

// parcours de l'ensemble des mots (= les clés) :
System.out.print("ensemble  des  mots  :");
for (String mot: map.keySet()) System.out.print(mot + ", ");

// parcours de l'ensemble des définitions (= les valeurs) :
System.out.print(« \nensemble  des  définitions  :");
for (String def: map.values()) System.out.println(def. toString () + "; ");

// parcours de l'ensemble des paires (cle , valeur)
System.out.println("\nensemble des paires (mot,définition) : ");

Iterator<Map.Entry<String,String>> itAssoc = map.entrySet().iterator();

while (itAssoc.hasNext()) {
    Map.Entry<String,String> e = itAssoc.next();
    String mot = e.getKey();
    String def = e.getValue();
    System.out.println(mot + " est défini par : " + def);
}
```

Quelques éléments indispensables

Classes wrappers ou enveloppes

- Les conteneurs *ne* peuvent *pas* contenir de types primitifs (int, double…)

- Pour faire des conteneurs de valeurs primitives, il existe des classes *wrappers*
  - `Integer, Double, Boolean, Character …`
  - Attention ! Ils sont non mutables ! (références partagées)
  - D'un primitif à son wrapper : constructeur `Integer boxI = new Integer(5);`
  - D'un wrapper à sa valeur : `valueOf()` `int valI = boxI.valueOf();`

```
HashSet<Integer> s = new HashSet<Integer> ();
s.add (new Integer (1)); s.add (new Integer (2));
s.add (new Integer (3)); s.add (new Integer (1));

int som=0;
Iterator<Integer> it = s.iterator ();
while (it.hasNext ()) {
        Integer i = it.next ();
        som+= i.intValue ();
} // !! som vaut 6 et non 7 ...
```

AutoBoxing, AutoUnboxing

- On peut donc faire des conteneurs de Integer, mais pas d'int

- Pour simplifier les manipulations, l'autoboxing et l'autounboxing permettent de manipuler les int comme des Integer et réciproquement
  - = rendre le passage par l'utilisation de wrappers transparentes au développeur

```
public class Ex24 {
  public static void main (String []arg) {
    HashSet <Integer> s = new HashSet <Integer> ();

    s.add(1); // au lieu de (new Integer(1))
    s.add(2); s.add(3); s.add(1);

    int som =0;
    Iterator <Integer> it = s.iterator();
    while (it.hasNext ()) {
       int i = it.next (); // au lieu de Integer i = it.next();
       som +=i;;
    } // som vaut 6 et non 7 a cause du equals de Integer
 }
}
```

Egalité d'objets : equals

- Tout objet comporte une méthode permettant de tester son égalité à un autre objet
  - `public boolean equals (Object o);`
- Par défaut, teste si les deux références pointent le même objet
- Est utilisée dans les conteneurs : recherche, appartenance, etc…
- Peut être rédéfinie
  - La méthode equals de la classe String compare bien le contenu de 2 chaînes
  - L'opérateur == sur 2 Strings compare uniquement les pointeurs
- Doit *souvent (toujours?)* être redéfinie dans vos classes

  quand elles sont destinées à remplir des conteneurs

Objets et fonction de hachage : hashCode

- Pour utiliser un HashSet/HashMap, il faut définir une fonction de hachage
- elle est calculée par l'objet lui-même
- Tout classe comporte une méthode (existante)

  ```
  public int hashCode ();
  ```

- **Les méthodes equals et hascode doivent etre cohérentes :**

  → deux objets equals doivent avoir le même hashCode

  → la fonction de hachage dépend uniquement des « valeurs » testant l'égalité

```
classA {
   int a;
   public boolean equals ( Object o) {... // l'égalite est basée sur
                                          // la valeur de a
        return a ==(( A) o).a;
   }
   public int hashCode () {          // le hashCode dépend de a uniquement
        return f(a);
   }
}
```

Objets et Comparaison

- La comparaison est aussi utilisée dans les conteneurs utilisant un ordre (TreeSet, TreeMap…)
- L'ordre est donné par une méthode de la classe

```
public int compareTo(Objet a);
```

- La valeur de retour de cette fonction est
  - Négative si `this` est plus petit que a
  - Nulle si les objets sont égaux
  - Positive si `this` est plus grand que a
- **Cette fonction doit être cohérente avec les 2 autres**

Interface

Qu'est ce ?

- Une interface est un ensemble d'opérations utilisée pour spécifier un service offert par une classe.
  - Elle peut être vue comme une classe sans attributs et dont toutes les opérations sont spécifiées mais pas définies à priori (ie vides)
  - Elle peut être vue comme un contrat ou un modèle que doivent offrir toutes les classes qui se réclame (implémente) de cette interface

```
public interface Comparable<T> {
     public int compareTo(T elem);
}
```

- Une classe qui implémente une interface doit définir toutes les méthodes de l'interface

```
class String implements Comparable<String>, ... { ... }
```

--> dans la classe String est implémentée la méthode compareTo :

```
class String implements Comparable<String>, ... {
     ...
     public int compareTo(String elem) { ...// ordre lexico }
}
```

Détails de syntaxe

- Les méthodes d’une interface sont toutes publiques
- Une interface n’a pas d’attributs, sauf si ce sont des constantes publiques

- Une classe peut implémenter plusieurs interface

```
class Exemple implements ItfA, ItfB { … }
```

- On ne peut pas créer un objet de type Interface

Utilisation d’interface

- 1ère utilisation : « puisque la classe String implémente l’interface Comparable, elle possède la méthode compareTo »

```
String str = «abc»;
if (str.compareTo(«cde»)<0) System.print(«+petit»);
else System.print(«+grand»);
```

- Une interface peut être utilisée *comme un « type »*

```
Comparable<String> str = new String(« abc »);
if (str.compareTo(«cde»)<0) System.print(«+petit»);
else System.print(«+grand»);
```

Utilisation d'interface

```
Comparable<String> str = new String(« abc »);
if (str.compareTo(«cde»)<0) System.print(«+petit»);
else System.print(«+grand»);
```

- 2 types !
  - Type statique : type avec lequel la variable a été déclarée
    la variable `str` est de type (statique) `Comparable<String>`
  - Type dynamique : classe avec laquelle l'objet a été construit
    l'objet référencé par `str` est de type (dynamique) `String`

- Liaison dynamique :
  la fonction exécutée est celle de la classe avec laquelle l'objet a été construit
  = le type dynamique
  - -> str.compareTo est donc la méthode compareTo de la classe String

  - NB: ceci est déterminé *à l'exécution*, pas à la compilation

Interface et conteneurs

- Collection<E> est une interface héritée par les interfaces List<E> et Set<E>
- List<E> est une interface implémentée par les classes ArrayList<E> et LinkedList<E>
- Set<E> est une interface implémentée par les classes HashSet<E> et TreeSet<E>
- Map<K,V> est une interface implémentée par les classes HashMap<K,V> et TreeMap<K,V>

- En pratique
  - Souvent, on déclare le conteneur le plus général pour son utilisation :
  - Donc les variables sont plutôt déclarée de type Collection ou List ou Set
  - Et les objets créés sont des objets des classes concrètes LinkedList, ArrayList, HashMap,…

- De même, Iterator<E> est une interface
- Les objets retournés par les fonctions iterator() des collections sont des objets de différentes classes privées selon le conteneur.

Exemple : changer le type de la collection

```
public class Ex24 {
  public static void main (String []arg)
{
    // Definir une collection
    Collection<Point> macoll;

    // Créer la liste de points
    macoll = new LinkedList<Point>();

    // Ajout des points dans la liste
    macoll.add(new Point(3,3));
    macoll.add(new Point(2,2));
    macoll.add(new Point(1,1));
    // la fct add est celle de la classe
    // LinkedList

   // Affichage de la liste
   System.out.println("Voici la liste=  »
                              + macoll);
    }
  }
}
```

```
public class Ex24 {
  public static void main (String[]arg)
  {
    // Definir une collection
    Collection<Point> macoll;

    // Créer le tableau de points
    macoll = new ArrayList<Point>();

    // Ajout des points dans le tableau
    macoll.add(new Point(3,3));
    macoll.add(new Point(2,2));
    macoll.add(new Point(1,1));
    // la fct add est celle de la
    // classe ArrayList

   // Affichage du tableau
   System.out.println("Voici la liste=  »
                              + macoll);
    }
  }
}
```

Nommage et paquetages

Portée des noms en Java

- 4 espaces de nom en Java
  - noms de variables dans les *blocs* `{ ... }`
    - Des variables de même nom dans des blocs différents ne sont pas en conflit
  - noms de variables dans les *fonctions*
    - Des variables de même nom dans des fonctions différentes ne sont pas en conflit
  - noms d'attributs et de méthodes dans les *classes*
    - Des méthodes / attributs de même nom dans des classes différentes ne sont pas en conflit

- Et pour les classes ? Deux classes de même nom n'entrent pas en conflit quand …?
  - Exemple : deux classes de même nom, il existe 2 classes Date en java :
    la standard et celle à utiliser quand on fait du sql (base de données)

- Il existe un 4ième niveau : le paquetage *package*
  - java.util.Date et java.sql.Date
  - Tous les conteneurs sont dans java.util

Package

- Espace de nom : définit les limites de validité des symboles (classes) utilisés
- Cohérence : essayer de mettre dans un même paquetage des classes associées à une fonctionnalité et/ou qui coopèrent

- Hiérarchie et lien avec les fichiers
  - hiérarchie des paquetages identiques à celles des fichiers les contenant avec 1 répertoire par paquetage.
  - java.awt contient le paquetage java.awt.image

- Définir le package d'appartenance d'une classe :
  - `package « nom du paquetage »;`
  - Optionnelle, mais première instruction du fichier si présente

- Utiliser les classes d'un autre package
  - Soit utiliser le nom complet de la classe : `java.lang.Math.sin()`
  - Soit importer le package contenant la classe avec l'instruction import
  - `import java.lang.Math; // On peut utiliser sin ()`
  - `import paquetage.*;` signifie qu'on importe toutes les classes du paquetage, NON compris celles des sous paquetages

Exemple

- Dans le fichier p1/Un.java

```
package p1;
public class Un {};
```

- Dans un autre fichier p1/Deux.java

```
package p1;
public class Deux {};
```

- Dans un autre fichier p2/Un.java

```
package p2;
public class Un{};
```

- Dans un autre fichier p2/sp1/Trois.java

```
package p2.sp1;
//public class Un{}
// Erreur car p2.Un

public class Trois {};
```

- Utilisation : nom complet

```
public class Essai{
    p1.Un  v1 = new p1.Un();
    p2.Un  v2 = new p2.Un();
};
```

- Utilisation : nom incomplet

```
import p1;
public class Essai{
    Un  v1 = new Un();
    p2.Un  v2 = new p2.Un();
};
```

- Utilisation : conflit !

```
import p1;
import p2;
// Impossible, conflit à la
   compilation
public class Essai{
    Un  v1 = new Un();
    Un  v2 = new Un();
};
```

Package et visibilité

- Paquetage et *classe* publique (ou pas) ?
  - **public** : la classe est accessible partout
    (dans et en dehors du package auquel elle appartient)
  - **<rien> :** (package visible) la classe n'est accessible qu'aux autres classes du package

- Pour les *membres* (attributs / méthodes)
  - **private** : accessible uniquement dans la classe où il est défini,
  - **public** : accessible dans n'importe quelle classe de n'importe quel package,
  - **protected** : … (cf héritage)
  - **<rien>** : accessible depuis le code de n'importe quelle classe du même package.