Les packages

Cours 7 / GL / LP DA2I

Cédric Lhoussaine

2019-2020

Outline

Packages

`Collection`, `Iterator`, `Map`, etc.

Énumérations

Packages

Pourquoi ?

Usages très variés :

- pour découper l'application en modules fonctionnels et regrouper les objets en relation
- pour masquer les détails d'implémentation (en cachant les classes non publiques)
- pour pouvoir charger un module indépendamment des autres
- pour fournir des bibliothèques d'outils

Nomenclature des packages

hiérarchie des packages Java = arborescence de fichiers

- `java.lang` → `java/lang`
- `java.util` → `java/util`
- `javax.swing` → `javax/swing`

Attention: un package et un "sous-package" (p.ex. `java.util` et `java.util.zip`) se comportent comme des packages distincts.

Créer un package

- faire débuter le fichier source de chaque classe par la clause package `<nom.complet>`;
- respecter autant que possible l'arborescence des sources = placer tous les fichiers sources d'un même package dans le répertoire approprié

  `package toto.divers` → répertoire `toto/divers`
- exemple: fichier `toto/Classe1.java`

```
package toto ;
public class Classe1 {
    int x ;
    public Classe1 (int x) { this.x = x ; }
    public String toString() {
        return "" + x ;
    }
    public static void main (String [] a) {
        System.out.println(new Classe1(5)) ;
    }
}
```

Utilisation

- la classe déclarée dans un package a pour nom complet `nom.du.package.LaClasse`
- dans le code des autres classes du même package, on peut la désigner par `LaClasse`.
- dans le code des classes extérieures au package:
  - par son nom complet
  - par `LaClasse` à condition d'avoir placé une clause `import` en début de fichier:

```
import nom.du.package.LaClasse ;
```

La clause import ne fait que définir un raccourci d'écriture

Compiler un package

Syntaxiquement, la compilation d'un package est réalisée comme n'importe quelle compilation:

```
javac -sourcepath <source> -d <destination> chemin/des/fichiers.java
```

Attention: `sourcepath` $\neq$ répertoire du package! Par exemple,

```
javac -sourcepath sources -d classes sources/toto/Classe1.java
```

$\rightarrow$ crée automatiquement une arborescence dans `classes` reflétant la hiérarchie des packages.

Exécuter une classe d'un package

```
java -classpath <chemin des classses> <nom complet des la classe>
```

Exemple:

```
java -classpath classes toto.Classe1
```

Documenter un package

```
javadoc -sourcepath <chemin des sources> -d <destination> <nom des packages>
```

Exemple:

```
javadoc -sourcepath sources -d doc toto truc
```

- les classes non publiques n'apparaissent pas (notion d'interface)
- le fichier `sources/toto/package.html` donne la description de l'ensemble du package dans la javadoc.

Packages et encapsulation

- Exemple: fichier `toto/Classe2.java`

```
package toto;
class Classe2 {
    private int x;
    Classe2(int x) { this.x = x; }
    public String toString() {
        return "" + x;
    }
    public static void main (String [] a) {
        System.out.println(new Classe2(5));
    }
}
```

```
package toto ;
/** Une classe publique du package toto */
public class Classe3 {
    public static void main (String [] a) {
        System.out.println(new Classe2(5));
    }
}
```

Packages et encapsulation

- la classe `Classe2` n'est utilisable que par d'autres classes du package `toto`;
- une clause `import toto.*;` ne la rend pas plus accessible;
- elle peut être compilée séparément;
- elle peut être exécutée séparément;
- on peut changer son accessibilité (`public class...`) sans changer celle de son constructeur ou de ses méthodes.

Packages de l'API Java

- toute classe Java appartient à un package
- les classes "usuelles" appartiennent à `java.lang` (`Object`, `Integer`, `String`, `System`...)
- toute classe a pour nom complet `nom.du.package.LaClasse`
  $\rightarrow$ `Object` = `java.lang.Object`
- les classes définies par l'utilisateur sont par défaut dans un package anonyme (`unnamed package`)

Quelques packages Java usuels

- `java.lang` toujours "chargé"
- `java.awt`, `java.awt.event`
- `javax.swing`, `javax.swing.event`
- `java.sql`
- `java.util`

La package `java.lang`

Ensemble de classes et d'interfaces de base du langage:

- interfaces: `Comparable`, `Iterable`, `Runnable`, `Cloneable`...
- classes "wrapper": `Boolean`, `Integer`, `Character`, `Double`, `Float`...
- classes d'usage courant: `String`, `Math`, `Object`...
- classes système: `System`, `Class`, `Compile`, `Thread`...
- classes d'exceptions et d'erreurs: `ArithmeticException`, `ArrayOutOfBoundsException`, `NoClassDefFoundError`...

La classe `Object`

- tout objet java bénéficie du type `Object` (cf. prochain cours)
- `public String toString()`
  - appelée automatiquement par `println`
  - concaténation de chaînes avec +
- `public boolean equals(Object o)`
  - teste l'égalité logique (en général égalité des attributs) $\rightarrow$ définition à la charge du programmeur
  - à ne pas confondre avec == qui évalue l'égalité physique (égalité des références)

Le package `java.util`

Ensemble de classes utilitaires avec leurs interfaces

- interfaces : `Collection`, `Iterator`, `Map`, `Set`, `List`...
- classes d'implémentation: `ArrayList`, `Stack`, `HashMap`...
- classes outils: `BitSet`, `Random`, `Scanner`, `StringTokenizer`, `Arrays`, `Collections`...
- classes exceptions: `EmptyStackException`, `NoSuchElementException`...

Collection, Iterator, Map, **etc.**

L'interface `Collection<E>`

Définit le comportement abstrait de tout regroupement d'objets d'un même type générique:

- ajouter des éléments
- supprimer des éléments
- test d'appartenance
- nombre d'éléments
- collection vide ?
- opérations en masse
- parcours itératif des éléments

L'interface Collection<E>

java.util.Collection<E>:

```
public interface Collection<E> {
    boolean isEmpty();
    void clear();
    int size();
    boolean equals(Object o);
    boolean add(E o);
    boolean remove(Object o);
    boolean contains(E o);
    boolean addAll(Collection<? extends E> c);
    boolean containsAll(Collection<?> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    Iterator<E> iterator();
    int hashCode();
    Object [] toArray();
    <T> T[] toArray(T [] a);
}
```

L'interface `Iterator<E>`

Un itérateur permet de parcourir une séquence d'objets

`java.util.Iterator<E>`:

```
public interface Iterator<E>
{
    boolean hasNext() ;
    E next() ;
    void remove() ;
}
```

Utilisation d'un itérateur

Principe: accès séquentiel aux éléments, à la demande

```
Collection c = ...;
Iterator it = c.iterator();
while (it.hasNext())
    System.out.println(it.next());
```

`next()` :

- renvoie un objet
- et positionne l'itérateur "devant" l'objet suivant !

Utilisation d'un itérateur

```
Collection c = ...;
Iterator it = c.iterator();
while (it.hasNext()) {
    Object o = it.next();
    System.out.println(o);
    if (o.toString().length() > 10)
        it.remove();
}
```

```
// collection ne contenant que des Integer
Collection<Integer> c = ...;
// alors l'itérateur opère sur des Integer
Iterator<Integer> it = c.iterator();
int somme = 0;
while (it.hasNext()) {
    // next retourne un Integer
    somme = somme + it.next();
}
```

Petite parenthèse : l'interface `Iterable`

`Collection<E>` est une sorte d'objet "itérable" (un sous-type de l'interface `Iterable<E>`)

L'interface java.lang.Iterable:

```
public interface Iterable<E> {
    Iterator<E> iterator();
}
```

La nouvelle syntaxe du `for`

Simplification du `for` pour le parcours d'un objet implémentant `Iterable` depuis Java 1.5:

```
// collection ne contenant que des Integer
Collection<Integer> c = ...;
int i = 0;
for (Integer x : c) {
    i = i + x;
}
```

La nouvelle syntaxe du `for`

Simplification du `for` pour le parcours d'un objet implémentant `Iterable` depuis Java 1.5:

```
// collection ne contenant que des Integer
Collection<Integer> c = ...;
int i = 0;
for (Integer x : c) {
    i = i + x;
}
```

- *cette syntaxe n'est valable que pour les itérables;*
- *en fait un appel implicite à l'itérateur.*

Une collection usuelle : `ArrayList`

Collection qui est plus précisément une liste

- sert à stocker en file une séquence d'objets

```
ArrayList<Integer> l = new ArrayList<Integer>();
for (int i=1; i<=10; i++)
    l.add(3*i) ; // = l.add(new Integer(3*i));
// pour l'utiliser...
Iterator<Integer> it = l.iterator();
```

Une collection usuelle : `ArrayList`

Collection qui est plus précisément une liste

- sert à stocker en file une séquence d'objets

```
ArrayList<Integer> l = new ArrayList<Integer>();
for (int i=1; i<=10; i++)
    l.add(3*i) ; // = l.add(new Integer(3*i));
// pour l'utiliser...
Iterator<Integer> it = l.iterator();
```

  - *accès direct possible : `get(i)` (d'où "array")*
  - *se comporte en fait comme un tableau extensible*

Un ensemble : `HashSet`

Implémentation de `Collection` sous la forme d'un ensemble (pas de doublons ni d'ordre a priori)

```
String s = "Il était un petit navire...";
HashSet<Character> e = new HashSet<Character>();
for (int i=0; i<s.length(); i++)
    e.add(s.charAt(i));
Iterator<Character> it = e.iterator();
while(it.hasNext())
    System.out.print(it.next() + " ");
```

Résultat:

```
v u e t r é p a n l . I i
```

Une collection particulière : `Stack`

implémentation de `Collection` sous la forme d'une pile

- structure LIFO (*Last In, First Out*)
- définit en plus ses propres méthodes (primitives de pile): `empty`, `push`, `pop`, `peek`

```
Stack<Character> pile = new Stack<Character>();
String mot = "INVERSER";
for (int i=0; i<mot.length(); i++)
    pile.push(mot.charAt(i));
while (!pile.empty())
    System.out.print(pile.pop()+"\t");
```

Résultat:

R E S R E V N I

Les dictionnaires

Un dictionnaire (`Map`) permet de stocker des associations entre des clefs et des valeurs

- les clefs sont toutes distinctes (logiquement donc physiquement)
- des valeurs peuvent être identiques (logiquement ou physiquement)

L'interface `java.util.Map`:

```
public interface Map<K,V> {
    int size();
    void clear();
    boolean equals(Object o);
    int hashCode();
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    boolean isEmpty();
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    Set<K> keySet();
    Collection<V> values();
    void putAll(Map<? extends K,? extends V> m);
    ...
}
```

La classe `HashMap`

- map = dictionnaire = tableau associatif
- association entre des clefs et des valeurs
- l'accès aux valeurs se fait par les clefs
- doublons possibles dans les valeurs
- unicité des clefs (selon l'égalité logique)

Exemple : dates d'anniversaire

clef	valeur
"Auguste"	5/5/1990
"Brutus"	3/2/1979
"César"	26/8/1985

Quelques méthodes de `HashMap<K,V>`

- `boolean containsKey(Object key)`: teste la présence d'une clef
- `V put(K key, V value)`: associe une valeur à une clef
- `V get(Object key)`: retourne la valeur associée à une clef
- `int size()`: nombre d'associations
- `boolean isEmpty()`: teste si dictionnaire vide
- `V remove(Object key)`: enlève l'entrée associée à la clef (et retourne la valeur correspondante)

Utilisation de HashMap

```
  public static void main(String [] args) {
      Map<String,Date> m = new HashMap<String,Date>();
      m.put("Auguste", new Date(5, 5, 1990)); // remplissage
      m.put("Brutus", new Date(3,2,1979));
      m.put("César", new Date(26,8,1985));
      // valeur associée à la clef "César"
      System.out.println(m.get("César"));
      // l'association existe toujours
      System.out.println(m.containsKey("César"));
      // remplacement de la valeur associée à la clef "Brutus"
      System.out.println(m.put("Brutus", new Date(2,3,1999)));
      // voir la nouvelle valeur
      System.out.println(m.get("Brutus"));
      // enlever cette association
      System.out.println(m.remove("Brutus"));
      System.out.println(m.containsKey("Brutus"));
  }
```

```
26 août 1985
true
3 février 1979
2 mars 1999
2 mars 1999
false
```

Un outil: `StringTokenizer`

- fonctionnement très proche de `Iterator`
- sert à découper une chaîne en "mots" (tokens)
- définit ses propres méthodes retournant `String`
- permet de définir arbitrairement le séparateur de mots

Utilisation de `StringTokenizer`

```
String s = "Les oiseaux chantent";
StringTokenizer t = new StringTokenizer(s);
while (t.hasMoreTokens())
    System.out.println(t.nextToken() + " / ");
```

Résultat: `Les / oiseaux / chantent /`

Utilisation de `StringTokenizer`

```
String s = "Les oiseaux chantent";
StringTokenizer t = new StringTokenizer(s);
while (t.hasMoreTokens())
    System.out.println(t.nextToken() + " / ");
```

Résultat: `Les / oiseaux / chantent /`

```
String s = "Les oiseaux chantent";
StringTokenizer t = new StringTokenizer(s, "ex");
while (t.hasMoreTokens())
    System.out.println(t.nextToken() + " / ");
```

Résultat: `L / s ois / au / chant / nt /`

Usages courants:

- opérations lexicales en général
- lecture de fichiers CSV

La classe `BitSet`

- représente un tableau de booléens
- extensible automatiquement
- associe à des entiers des valeurs `true` / `false`

Quelques méthodes:

- `void set(int index)`
- `void clear(int index)`
- `boolean get(int index)`
- `int length()`
- méthodes logiques (`and`, `intersects`...)

Utilisation de `BitSet`

On veut mémoriser les longueurs des mots d'une phrase

```
String s = "Les oiseaux chantent ici";
BitSet bs = new BitSet();
StringTokenizer t = new StringTokenizer(s);
while (t.hasMoreTokens())
    bs.set(t.nextToken().length());
for (int i=0; i<bs.length(); i++)
    if (bs.get(i)) System.out.print(i + "\t");
```

Utilisation de `BitSet`

On veut mémoriser les longueurs des mots d'une phrase

```
String s = "Les oiseaux chantent ici";
BitSet bs = new BitSet();
StringTokenizer t = new StringTokenizer(s);
while (t.hasMoreTokens())
    bs.set(t.nextToken().length());
for (int i=0; i<bs.length(); i++)
    if (bs.get(i)) System.out.print(i + "\t");
```

Résultat:

```
3 7 8
```

Énumérations

Type énuméré

Un type énuméré est un type qui ne peut prendre qu'un nombre fini de valeurs.

Exemples:

- les numéros ou les noms des mois;
- les valeurs ou les couleurs d'un jeu de cartes;
- les espèces d'un système monétaire;
- les mots-clefs d'un langage de programmation

Les énumérations

Réalisation "à la main" possible :

```
public class Carte {
    public static final int PIQUE=0;
    public static final int TREFLE=1;
    public static final int COEUR=2;
    public static final int CARREAU=3;
    //...
    private int valeur, couleur;

    public Carte(int valeur, int couleur){
        //...
    }
}

//...
new Carte(Carte.ROI, Carte.PIQUE);
```

Inconvénients d'un tel "bricolage":

- dépend explicitement d'un type (arbitraire) pourquoi `int` plutôt que `char` ou `String` ?
- possibilité d'utiliser des valeurs incorrectes

```
new Carte(1024, -1);
```

- opérations aberrantes:

```
Carte.PIQUE + Carte.CARREAU = ? ? ?
```

- les valeurs affichées (par `println`) ne donnent pas d'information intelligible (il faut une méthode pour les "retransformer")

Les énumérations en Java

Deux nouveaux types

```
public enum Couleur { PIQUE, TREFLE, COEUR, CARREAU }
public enum Figure { VALET, DAME, ROI, AS }
public class Carte {
    private Couleur couleur;
    private Figure valeur;
    public String toString() {
        return valeur + " de " + couleur;
    }
}
```

- instanciation : `new Carte(Figure.ROI, Couleur.PIQUE);`
- affichage : `"ROI de PIQUE"`

Les énumérations en Java

La méthode de classe `values()` renvoie les valeurs de l'énumération sous la forme d'un objet `Iterable`:

```
for (Figure f: Figure.values())
   System.out.println(f);
```

Résultat:

```
VALET
DAME
ROI
AS
```

Les énumérations en Java

Les énumérations sont en fait de véritables classes dont les instances sont en nombre limité, identifiées et invariables → possibilité de leur ajouter:

- des attributs (obligatoirement final)
- des constructeurs (par défaut : sans paramètres)
- des méthodes d'instance ou de classe
- l'implémentation d'interfaces

Les énumérations en Java

Les énumérations sont en fait de véritables classes dont les instances sont en nombre limité, identifiées et invariables → possibilité de leur ajouter:

- des attributs (obligatoirement final)
- des constructeurs (par défaut : sans paramètres)
- des méthodes d'instance ou de classe
- l'implémentation d'interfaces

  toute énumération `MonEnum` implémente automatiquement `Comparable<MonEnum>` (les éléments de l'énumération sont censés être donnés dans l'ordre croissant)

En savoir plus : cf. `java.lang.Enum`

Une énumération sophistiquée

```
enum Valeur {
    deux(2), trois(3), quatre(4), cinq(5), six(6), sept(7),
    huit(8), neuf(9), dix(10), valet, dame, roi, as;
    private final int val_num;      // attributs !
    private final boolean has_val_num;
    Valeur() {       // et constructeurs !
        has_val_num = false;
        val_num = 0; }
    Valeur(int v) {
        has_val_num = true;
        val_num = v; }
    public String toString() {
        if (has_val_num)
            return "" + val_num;
        else return super.toString();
    }
}
for (Valeur v: Valeur.values())
    System.out.print(v + " ");
```

Une énumération sophistiquée

```
enum Valeur {
    deux(2), trois(3), quatre(4), cinq(5), six(6), sept(7),
    huit(8), neuf(9), dix(10), valet, dame, roi, as;
    private final int val_num;      // attributs !
    private final boolean has_val_num;
    Valeur() {       // et constructeurs !
        has_val_num = false;
        val_num = 0; }
    Valeur(int v) {
        has_val_num = true;
        val_num = v; }
    public String toString() {
        if (has_val_num)
            return "" + val_num;
        else return super.toString();
    }
}
for (Valeur v: Valeur.values())
    System.out.print(v + " ");
```

Résultat: 2 3 4 5 6 7 8 9 10 valet dame roi as

La classe Date revisitée

```
public enum Mois {
    janvier(31), fevrier(28), mars(31), avril(30),
    mai(31), juin(30), juillet(31), aout(31),
    septembre(30), octobre(31), novembre(30), decembre(31);
    private final int nbJours;

    Mois(int nbJours) {
        this.nbJours = nbJours;
    }

    public int nbJours(int annee) {
        if ((this == fevrier) &&
            (SuperDate.estBissextile(annee)))
            return nbJours + 1;
        return nbJours;
    }

    public static Mois numero(int n) {
        if ((n > 0) && (n <= 12))
            return values()[n-1];
        return null;
    }
}
```

La classe Date revisitée

```
public enum Langue {
    francais, english ;
    final String [][] NOMS={
        {"janvier", "février", "mars", "avril",
         "mai", "juin", "juillet", "août",
         "septembre", "octobre", "novembre", "décembre"},
        {"January", "February", "March", "April",
         "May", "June", "July", "August",
         "September", "October", "November", "December"}} ;
    public String nommer(int mois) {
        return NOMS[this.ordinal()][mois] ;
    }
}
```

La classe Date revisitée

```
public class SuperDate {
    private int jour, annee;
    private Mois mois;
    public static Langue langue = Langue.francais;

    public SuperDate(int j, int m, int a) {
        this.jour = j; this.annee = a;
        this.mois = Mois.numero(m);
    }
    public static boolean estBissextile(int a) {
        return ((a % 4 == 0) && (a % 100 != 0)) ||
            (a % 400 == 0);
    }
    public String toString() {
        return this.jour + " " +
            langue.nommer(this.mois.ordinal())
            + " " + this.annee;
    }
    public boolean estValide() {
        return (this.jour > 0) &&
            (this.jour <= this.mois.nbJours(this.annee));
    }
    public static void main(String [] args) {
        System.out.println(new SuperDate(8, 4, 2005));
    }
}
```

Emacs 26.2 (Org mode 9.1.9)