Introduction programmation Java

Cours 4: Map et Notion d'ordre

Stéphane Airiau

Université Paris-Dauphine

Gestion des classes à l'aide de Packages

(espaces de noms)

Motivations

- garantir l'unicité des noms de classes.

ex : Par exemple, deux développeurs peuvent avoir l'idée de développer une classe `Dauphine`. Tant que les classes se trouvent dans des espaces de noms différents, cela ne posera aucun problème et les deux classes pourront être utilisables.

- nombre (parfois important) de classes dans un projet

⇨ utiliser une structure hiérarchique de répertoires pour ordonner les classes

⇨ on rassemble dans un package (~ répertoire) les classes reliées.

Déclaration d'une classe

- nom qualifié : nom de la classe à partir de la racine
- ⇨ c'est le nom qualifié de la classe qui est unique
- pour les classes de la librairie standard, le nom qualifié commence par `java`, par exemple `java.util.Vector` est le nom qualifié de la classe `Vector`
- Attention, en `Java`, les espaces de noms ne s'emboitent pas les uns dans les autres. `java.util` et `java.util.function` sont deux espaces de noms distincts avec chacun leurs classes et interfaces.

Dans le fichier source `Java`, on commence toujours à indiquer à quel espace de noms la classe appartient en commençant le fichier par la ligne `package <nom_du_package>`.
information redondante ⇨ on connait l'emplacement du fichier dans le disque.
Mais très utile lorsqu'on lit le code dans un éditeur !
`Java` vérifie la position du fichier dans le répertoire correspondant

Utiliser une classe d'un espace de noms

- Pour utiliser une classe qui se trouve dans le même espace de noms

⇨ utiliser son nom simple.

- Pour utiliser une classe à l'extérieur de son espace de noms

⇨ on peut toujours utiliser son nom qualifié.
mais cela peut être lourd

- on fait un `import`
tout se passe comme si la/les classe(s) importée(s) font partie(s) de l'espace de nom courant.

NB L'import se fait toujours au début de la classe.

Attention

1- `import qqch.*`, vous importez seulement les classes, **pas les sous espaces** de noms.
   ⇨ `import java.*` ne permet pas d'obtenir tous les espaces de noms qui commencent par `java.`.

2- import de plusieurs espaces de noms
   ⇨ car il est possible d'avoir des conflits de noms

L'utilisation de packages permet alors de désigner sans ambigüité une classe.

⇨

On peut donc avoir deux méthodes avec exactement la même signature tant que ces deux méthodes appartiennent à des espaces de noms différents.

Librairie standard

- `java.lang` contient les classes fondamentales du langage.
- `java.util` contient les classes pour manipuler des collections d'objets, des modèles d'évènements, des outils pour manipuler les dates et le temps, et beaucoup de classes utiles.
- `java.io` contient les classes relatives aux entrées et sorties
- ...

Compilation et Exécution

Pour compiler une classe, il faut indiquer son chemin depuis la racine.

```
javac important/premier/MaSecondeClasse.java
```

Le compilateur générera le fichier `important/premier/MaSecondeClasse`
Si `MaSecondeClasse` possède une méthode `main`, on lance l'exécution en spécifiant le nom complet de la classe. Pour l'exemple, on lancerait donc

```
java important.premier.MaSecondeClasse
```

un `Map` représente une relation binaire surjective : chaque élément d'un `Map` est une paire qui met en relation une une clé à une valeur : chaque clé est unique, mais on peut avoir des doublons pour les valeurs.
⇨un tableau à deux colonnes !

Interface Map<K,V>

`V`	`get(Object key)` Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
`V`	`put(K key, V value)` Associates the specified value with the specified key in this map. If the map previously contained a mapping for the key, the old value is replaced by the specified value.
`V`	`remove(Object key)` Removes the mapping for a key from this map if it is present.
`default V`	`replace(K key, V value)` Replaces the entry for the specified key only if it is currently mapped to some value. Returns the value to which this map previously associated the key, or null if the map contained no mapping for the key.

Parcours d'un `Map`

Attention, `Map` n'est pas une sous interface de `Iterable`, donc on ne peut pas parcourir un `Map` avec une boucle `for each`!

On peut obtenir l'ensemble des clés, l'ensemble des valeurs, et l'ensemble des paires (clé,valeur) grace aux méthodes suivantes :

- `Set<K> keySet()` donne l'ensemble des clés (bien un `Set`)
- `Collection<V> values()` donne la collection des valeurs
- `Set<Map.Entry<K,V» entrySet()` donne l'ensemble des paires

Classe `Map.Entry`

`Map.Entry` désigne une interface `Entry` qui est interne à l'interface `Map`. On peut créer des classes à l'intérieur de classes, mais je n'en parlerai pas plus aujourd'hui.

`Interface Map.Entry<K,V>`

`K`	`getKey()` Returns the key corresponding to this entry.
`V`	`getValue()` Returns the value corresponding to this entry.
`V`	`setValue(V value)` Replaces the value corresponding to this entry with the specified value (optional operation).

Exemple parcours d'une `Map`

```
1  Map<Personnage,Region> origines = new HashMap<>();
2  ...
3  for (Map.Entry<Personnage,Region> paire: origines.entrySet()){
4    Personnage p = paire.getKey();
5    Region r = paire.getValue();
6    if (r.getName().equals("Ibère"))
7      System.out.println(p);
8   }
```

Dans cet exemple, on part une `Map` qui associe à chaque personnage sa région d'origine et on affiche seulement les personnages qui sont des ibères.

Performances sur des opérations

Pour de larges volume de données, les méthodes usuelles (add, remove, contains, size) devraient être rapides.

Implémentation de l'interface `Set`

	`add`	`remove`	`contains`
`HashSet`	temps constant	temps constant	temps constant
`TreeSet`	$log(n)$	$log(n)$	$log(n)$

Implémentation de l'interface `List`

	`get`	`set`	autre
`LinkedList`	n	n	
`ArrayList`	temps constant*	temps constant	n

Performances empirique

Je veux comparer les performances de la méthode`contains` pour les classes `ArrayList`, `LinkedList`, `HashSet` et `TreeSet`.

J'ai un dictionnaire d'environs 25,000 mots à ma disposition
Pour 100 exécutions
1- je tire au sort environ 10,000 mots du dictionnaire et je les places dans la structure
2- je tire au sort 1000 mots du dictionnaire et je les place dans un tableau de `String`
3- je mesure le temps qu'il faut pour savoir si chaque mot du tableau de `String` se trouve dans ma structure
Je calcule le temps total pour chaque structure

structure	`ArrayList`	`LinkedList`	`HashSet`	`TreeSet`
`contains`	3.245 s	5.286 s	0.004 s	0.024 s
`remove`	3.155 s	5.378 s	0.003 s	0.041 s

Temps total

Ordre

Ordre

L'interface `Comparable` contient une seule méthode :

```
public int compareTo(T o)
```

Cette méthode retourne

- un entier négatif si l'objet est plus petit que l'objet passé en paramètre
- zéro s'ils sont égaux
- un entier positif si l'objet est plus grand que l'objet passé en paramètre.

les classes `String`, `Integer`, `Double`, `Date`, `GregorianCalendar` et beaucoup d'autres implémentent toutes l'interface `Comparable`.

Exemple

```
1  public class Gaulois extends Personnage
2                  implements Comparable<Gaulois>{
3    String nom;
4    int quantiteSanglier;
5      ...
6
7    public int compareTo(Gaulis ixis) {
8      return this.quantiteSanglier - ixis. quantiteSanglier;
9    }
10 }
```

interface `Comparator`

Une classe qui implémente l'interface comparator représente une notion d'ordre / un critère d'ordre.
A priori, on n'a besoin d'implémenter une seule méthode, la méthode pour comparer deux éléments.

```
1 public interface Comparator<T> {
2    int compare(T o1, T o2);
3    boolean equals(Object obj);
4 }
```

Pour comparer des `Gaulois`, et même tous les `Personnage` selon leur taille, on peut écrire la classe suivante :

```
1 public class OrdreHauteur implements Comparator<Personnage> {
3     public int compare(Personnage gauche, Personnage droit){
4       return gauche.hauteur < droite.hauteur ? -1:
5             (gauche.hauteur == droite.hauteur ? 0 : 1);
5     }
6   }
```

Trier les éléments d'une collection

La méthode `sort` de l'interface `List` effectue le tri.

```
void sort(Comparator<? super E> c)
```

Sorts this list according to the order induced by the specified Comparator. All elements in this list must be mutually comparable using the specified comparator (that is, c.compare(e1, e2) must not throw a ClassCastException for any elements e1 and e2 in the list).

If the specified comparator is null then all elements in this list must implement the Comparable interface and the elements' natural ordering should be used.

- appel avec le paramètre `null` : il vaudrait mieux que la liste contienne des instances d'une classe implémentant l'interface `Comparable`, sinon, `Java`ne va pas savoir comment comparer deux éléments !
  on utilise le critère d'ordre implémenté dans la classe des éléments de la liste
- appel avec une instance d'une classe implémentant l'interface `Comparator` : on donne directement le critère pour comparer !

Trier les éléments d'une collection : autre solution

Deux méthodes de tri sont implémentées dans `Java`et se trouvent dans la classe `Collections` (attention avec un **s**).

- La première méthodea un seul argument : une collection d'instance d'une classe qui implémente l'interface `Comparable`. Le tri se fait donc en utilisant la méthode `compareTo` codée dans la classe `T`.
- La seconde méthode nécessite deux arguments : la collection d'instance d'une classe `T` et une notion d'ordre sur la classe `T`. Le tri se fera donc en utilisant la méthode `compare` codée dans la classe implémentant `Comparator`.

bon exercice pour les méthodes **`static`** avec paramètre de type !

```
   // classe Collections
1  public static <T extends Comparable<? super T»
2                  void sort(List<T> list)
3  public static <T> void sort
4                  (List<T> list, Comparator<? super T> c)
```

Exemple

```
1  public static void main(String[] args){
2    List<Personnage> personnages = new ArrayList<>();
3    personnages.add(new IrreductibleGaulois("Obelix", 1.81));
4    personnages.add(new IrreductibleGaulois("Astérix", 1.60));
5    personnages.add(new Personnage("César", 1.75));
6
7    for (Personnage p: personnages)
8      System.out.println(p.presentation());
9
10   personnages.sort(null);
11     // ou bien Collections.sort(personnages);
12   for (Personnage p: personnages)
13     System.out.println(p.presentation());
14
15   Comparator<Personnage> ordre = new OrdreHauteur();
16   personnages.sort(ordre);
17     // ou bien Collections.sort(personnages, ordre);
18   System.out.println(personnages);
```

L'utilisation d'une instance d'une classe implémentant l'interface `Comparator` est quelque peu lourde. On n'a pas le temps pour introduire des notions pour permettre une écriture plus élégante.

Gestion des Exceptions

Gérer l'inattendu !

Exemple

```
1  public static Random generator = new Random();
2
3  public static int randInt(int low, int high){
4     return low + (int) (generator.nextDouble() * (high-low +1));
5  }
```

Que se passe-t-il si l'utilisateur écrit

```
   randInt(10,5);
```

On pourrait modifier le code de `randInt` pour prendre en compte ce type d'erreur (mais peut être que l'utilsateur a fait une erreur d'interprétation plus profonde...)

Exemple (suite)

Java nous permet de lever une **exception** adaptée, par exemple :

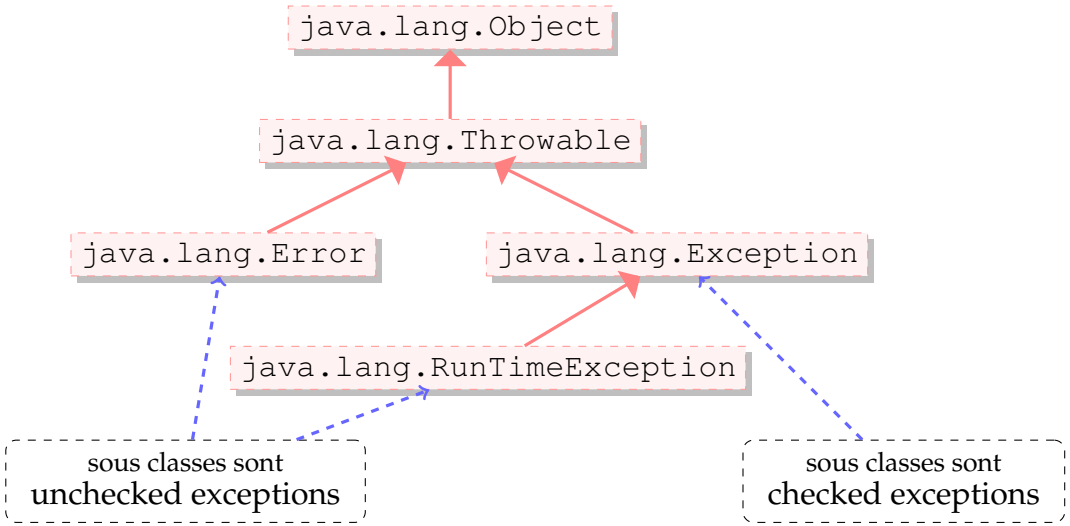
```
if (low > high)
  throw new IllegalArgumentException(
    "low should be <= high but low is " + low + " and high is " + high);
```

On lève une exception d'une classe qui porte un nom parlant `IllegalArgumentException` avec un message qui aidera au debogage.

Quand une exception est levée, l'exécution "normale" est interrompue. Le contrôle de l'exécution passe à un gestionnaire d'erreur.

Gestion des erreurs

`Java` possède un mécanisme de gestion des erreurs, ce qui permet de renforcer la sécurité du code. On peut avoir différents niveaux de problèmes :

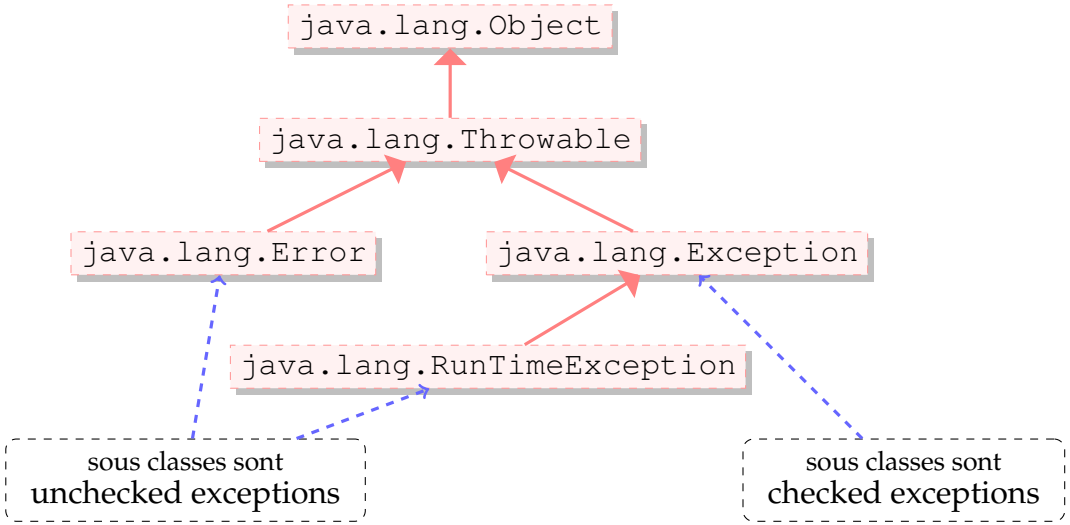


- `Error` représente une erreur grave intervenue dans la machine virtuelle (par exemple `OutOfMemory`)
- La classe `Exception` représente des qui sont reportées au développeur.

⇨ le développeur a la possibilité de **gérer** de telles erreurs et **éviter** que l'application ne se termine

⇨ on veut gérer les erreurs que l'on peut anticiper

Gestion des erreurs



- `checked Exception` représente des cas où on peut anticiper le problème (par exemple entrée-sortie).
- `RunTimeException` ne peuvent pas être vérifiée lors de la compilation
  plutôt des erreurs causées par le programmeur (`NullPointerException`) que des erreurs causées par une source incontrolable (ex entrée-sortie).

Gestion des erreurs

- Utilisez des exceptions pour des conditions exceptionnelles
- checked exception : conditions pour lesquelles on peut raisonnablement espérer pouvoir récupérer et poursuivre l'exécution
  ⇨ force le développeur à gérer une condition exceptionnelle (éviter qu'il ne la néglige trop)
  ⇨ *"exception d'utilisation"* : ces erreurs doivent être déclarées dans la signature de la fonction si elles ne sont pas capturées par la fonction.
- run time exception : indique une erreur de programmation `NullPointerException` est "unchecked" (sinon, il faudra toujours traiter cette erreur car elle peut intervenir partout !)

Code reuse : utilisez au plus les exceptions de la bibliothèque standard

- `IllegalArgumentException` : argument dont la valeur n'est pas appropriée
- `IllegalStateException` l'objet n'est pas "prêt" pour faire cet appel
- … on pourrait utiliser seulement ces deux exceptions car la plupart des erreurs correspondent à un argument ou un état illégal
  - `NullPointerException`
  - `IndexOutOfBoundsException`
  - `ArithmeticException`
  - `NumberFormatException`
- pas une science exacte :
  ex : on possède une méthode pour tirer au sort k dominos dans la pioche, k étant un argument de ma méthode. Supposons qu'un utilisateur demande de tirer k dominos alors qu'il y a $k-1$ dominos dans la pioche. On peut utiliser
  - `IllegalArgumentException` : k est trop grand
  - `IllegalStateException` pas assez de dominos dans la pioche

Déclarer des checked exceptions

On peut utiliser une clause **`throws`** pour déclarer une méthode dont on peut anticiper l'échec.

```
public void write(Object obj, String filename)
    throws IOException, ReflectiveOperationException
```

Dans la déclaration, on peut grouper les erreurs dans une super classe, ou bien les lister. Dans l'exemple, on lève des exceptions de type `IO`, mais on pourrait nommer plus précisemment toutes les types possibles (sous classes de `IOException`).

Levée d'exception

Lors de la détection d'une erreur

- un objet qui hérite de la classe `Exception` est créé
- ⇨ ce qui s'appelle **lever une exception**
- l'exception est propagé à travers la pile d'exécution jusqu'à ce qu'elle soit traitée.

```
1 int[] tab = new int[5];
2 tab[5]=0;
```

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 5
at Personnage.main(Personnage.java:2)
```

```
1 int d=10,t1=5,t2=5;
2 System.out.println("vitesse:" + d / (t2-t1));
```

```
Exception in thread "main"
java.lang.ArithmeticException: / by zero
at Personnage.main(Personnage.java:21)
```

Le bloc `try ... catch`

- bloc `try` : le code qui est susceptible de produire des erreurs
- on récupère l'exception créée avec le `catch`.
- on peut avoir plusieurs blocs `catch` pour capturer des erreurs de types différentes.
- en option, on peut ajouter un bloc `finally` qui sera toujours exécuté (qu'une exception ait été levée ou non)

Lorsqu'une erreur survient dans le bloc `try`,

- la suite des instructions du bloc est abandonnée
- les clauses `catch` sont testés **séquentiellement**
- le premier bloc `catch` correspondant à l'erreur est exécuté.
  ⇨ l'ordre des blocs `catch` est donc de l'erreur la plus spécifique à la plus générale !
  ⇨ on a un mécanisme de **filtre**

Choisir où traiter une exception

Lorsqu'on utilise un morceau de code qui peut lever une exception, on peut

- traiter l'exception immédiatement dans un bloc `try ... catch`
- propager l'erreur :

```
public Domino[] tirer(int nb) throws IllegalStateException {
   ...
}
...
public void distribuer() {
  ...
  Domino[] jeu = tirer(7);
  ...
}
```

Ici, on a une erreur, car l'exception qui peut être levée dans `tirer` n'est pas traitée dans la méthode `distribue`.

Choisir où traiter une exception

Lorsqu'on utilise un morceau de code qui peut lever une exception, on peut

- traiter l'exception immédiatement dans un bloc `try ... catch`
- propager l'erreur :

```
public Domino[] tirer(int nb) throws IllegalStateException {
   ...
}
...
public void distribuer() throws IllegalStateException {
  ...
  Domino[] jeu = tirer(7);
  ...
}
```

Dans l'exemple, on ne traite pas l'exception dans la méthode `distribuer`, on la propage à la méthode qui appelle la méthode `distribue`.

⇨ choisir judicieusement l'endroit où l'erreur est traitée.

Déclarer et bien Documenter les exceptions

- une méthode peut lever plusieurs exceptions
  ⇨ on peut être tenté de les regrouper dans une classe exception mère (mais on perd un peu d'information)
  ⇨ on peut déclarer chaque exception individuellement
  (N.B. le langage ne requiert pas la déclaration que chaque exception que le code peut lever)

⇨ permet au développeur d'être plus attentif sur certains aspects

- `@throws` balise `javadoc` permet de documenter chaque exception
- Depuis `Java` 7, on peut gérer plusieurs exceptions dans une même clause catch

```
catch(ExceptionType1 | ExceptionType2 | ExceptionType 3 e){
   ...
}
```

N'ignorez pas les exceptions

- si une méthode peut lever une exception
  ⇨ le développeur de la méthode n'a pas fait ce choix sans raison
  ⇨ ne l'ignorez pas !
- il est facile d'ignorer une exception sans rien faire.
- au minimum, écrivez un message indiquant pourquoi vous vous êtes permis d'ignorer l'exception.

Pour les unchecked exceptions

- on peut utiliser la sortie erreur (différente de la sortie standard) pour afficher des message
- La méthode `printStackTrace()` de la classe `Throwable` donne le parcours complet de l'exception du moment où elle a été levée jusqu'à celui où elle a été capturée.

Exemple

```
1  int d=10,t1=5,t2=5;
2  try{
3    System.out.println("vitesse:" + d / (t2-t1));
4  }
5  catch(ArithmeticException e){
6     System.out.println(" vitesse non valide ");
7  }
8  catch(Exception e){
9     e.printStackTrace();
10 }
```

Créer sa propre exception

- La classe `MyException` hérite de la classe `Exception`.
- Une méthode qui risque lever une exception de type `MyException` l'indique à l'aide de **throws**

```
1  public class PotionMagiqueException extends Exception {
2     public PotionMagiqueException(){
3       super();
4     }
5     public PotionMagiqueException(String s){
6       super(s);
7     }
8  }
9
10 public class GourdePotionMagique {
11    private int quantite, gorgee=2,contenance=20;
12    public GourdePotionMagique(){ quantite=0;}
13
14    public boolean bois() throws PotionMagiqueException {
15      if (quantite-gorgee <0)
16        throw new PotionMagiqueException
17                    (" pas assez de potion magique!");
18    }
19 }
```

Exceptions et entrée/sortie

```
1  try {
2    FileInputStream fis = new FileInputStream(new File("test.txt"));
3    byte[] buf = new byte[8];
4    int nbRead = fis.read(buf);
5    System.out.println("nb bytes read: " + nbRead);
6    for (int i=0;i<8;i++)
7      System.out.println(Byte.toString(buf[i]));
8    fis.close();
9
10   BufferedReader reader =
11       new BufferedReader(new FileReader(new File("test.txt")));
12   String line = reader.readLine();
13   while (line!= null){
14     System.out.println(line);
15     line = reader.readLine();
16   }
17   reader.close();
18 } catch (FileNotFoundException e) {
19   e.printStackTrace();
20 }
21 catch(IOException e){
22   e.printStackTrace();
23 }
```