Collections : Listes

Théorie et pratique de la programmation
Michel Schinz – 2013-02-18

Collections

Collection

On appelle **collection** un objet dont le but est de stocker un certain nombre d'autres objets.

Exemples :

1. tableau,
2. liste,
3. ensemble,
4. table associative,
5. queue de priorité,
6. etc.

Nous étudierons les quatre premiers.

Liste

Une **liste** (*list* en anglais) est une collection ordonnée d'éléments.

Les opérations principales sur une liste sont :

- l'ajout d'un élément, souvent à une extrémité,
- la suppression d'un élément,
- l'obtention de l'élément d'indice donnée,
- le parcours ordonné des éléments.

Exemple : liste des étudiants participant à un cours.

Ensemble

Un **ensemble** (*set* en anglais) est une collection non-ordonnée d'éléments qui n'admet pas les éléments dupliqués.
Les opérations principales sur un ensemble sont :

- l'ajout d'un élément,
- la suppression d'un élément,
- le test d'appartenance d'un élément,
- le parcours des éléments, dans un ordre (souvent) quelconque.

Exemple : ensemble des cours suivis par un étudiant durant ses études.

Table associative

Une **table associative** ou **dictionnaire** (*map* ou *dictionary* en anglais) est une collection associant des valeurs à des clefs.

Les opérations principales d'une table associative sont :

- l'ajout d'une association clef→valeur,
- le remplacement d'une valeur associée à une clef,
- la suppression d'une clef, et donc de la valeur associée,
- la recherche de la valeur associée à une clef.

Exemple : table associant la liste des participants à un cours.

Exercice

Quel(s) type(s) de collection – liste, ensemble ou table associative – serait-il raisonnable d'utiliser pour stocker les éléments suivants :

1. la date de naissance de scientifiques célèbres,
2. les messages échangés par les participants à une discussion en ligne (*chat*),
3. le nom de la capitale de chaque pays du monde,
4. le nom de toutes les substances dopantes interdites aux sportifs,
5. la totalité des nombres premiers entre 0 et 10000,
6. les « diapositives » d'un programme comme PowerPoint.

Mises en œuvre

Pour chaque genre de collection, on peut imaginer plusieurs **mises en œuvre** (*implementations* en anglais) concrètes.
Par exemple, une liste peut être mise en œuvre au moyen d'un tableau qu'on redimensionne au besoin, ou au moyen d'éléments chaînés entre-eux.

Pour traduire cela en Java, nous utiliserons :

- une interface par genre de collection,
- une classe par mise en œuvre de chaque collection (cette classe implémente, au sens de Java, l'interface de la collection).

Listes (de chaînes de caractères)

Interface

Dans un premier temps, nous ne considérons que les listes de chaînes de caractères, décrites par l'interface `StringList`.

```
interface StringList {
  boolean isEmpty();
  int size();
  void add(String newElem);
  void remove(int index);
  String get(int index);
  void set(int index, String elem);
}
```

Exemple d'utilisation

```
StringList workingDays = new …;
workingDays.add("lundi");
workingDays.add("mardi");
workingDays.add("mercredi");
workingDays.add("jeudi");
workingDays.add("vendredi");
System.out.println("nb. de jours ouvrés : " +
                   workingDays.size());
System.out.println("2e jour ouvré : " +
                   workingDays.get(1));
```

Mises en œuvre

Nous examinerons deux mises en œuvre du concept de liste :

1. les **tableaux-listes**, dont les éléments sont stockés dans un tableau qui est « redimensionné » (copié, en réalité) au besoin,
2. les **listes chaînées**, dont les éléments sont stockés dans des nœuds liés entre-eux.

La complexité des opérations de base diffèrent entre ces deux mises en œuvre, de même que l'utilisation mémoire.

Complexité comparée

Opération	Tableau-liste	Liste chaînée
Ajout (`add`)	O(1) amorti	O(1)
Suppression (`remove`)	O(n)	O(n)*
Obtention (`get`)	O(1)	O(n)*
Taille (`size`)	O(1)	O(1)**

* O(1) pour les éléments situés aux extrémités de la liste.

** pour peu que la taille soit mémorisée, sinon O(n)

Mise en œuvre 1 : tableau-liste

Tableau-liste

L'idée des tableaux-listes est la suivante :

- on utilise un tableau, appelé tableau sous-jacent, pour stocker les éléments du tableau-liste,
- si le tableau sous-jacent est plein et qu'un élément doit être inséré, on en crée un nouveau plus grand que le premier dans lequel on copie les anciens éléments ; finalement, on remplace l'ancien tableau sous-jacent par le nouveau.

La taille du tableau sous-jacent détermine le nombre d'éléments qu'on peut stocker dans le tableau-liste avant de devoir redimensionner ce premier. Cette taille est appelée la **capacité** du tableau-liste.

Tableau-liste

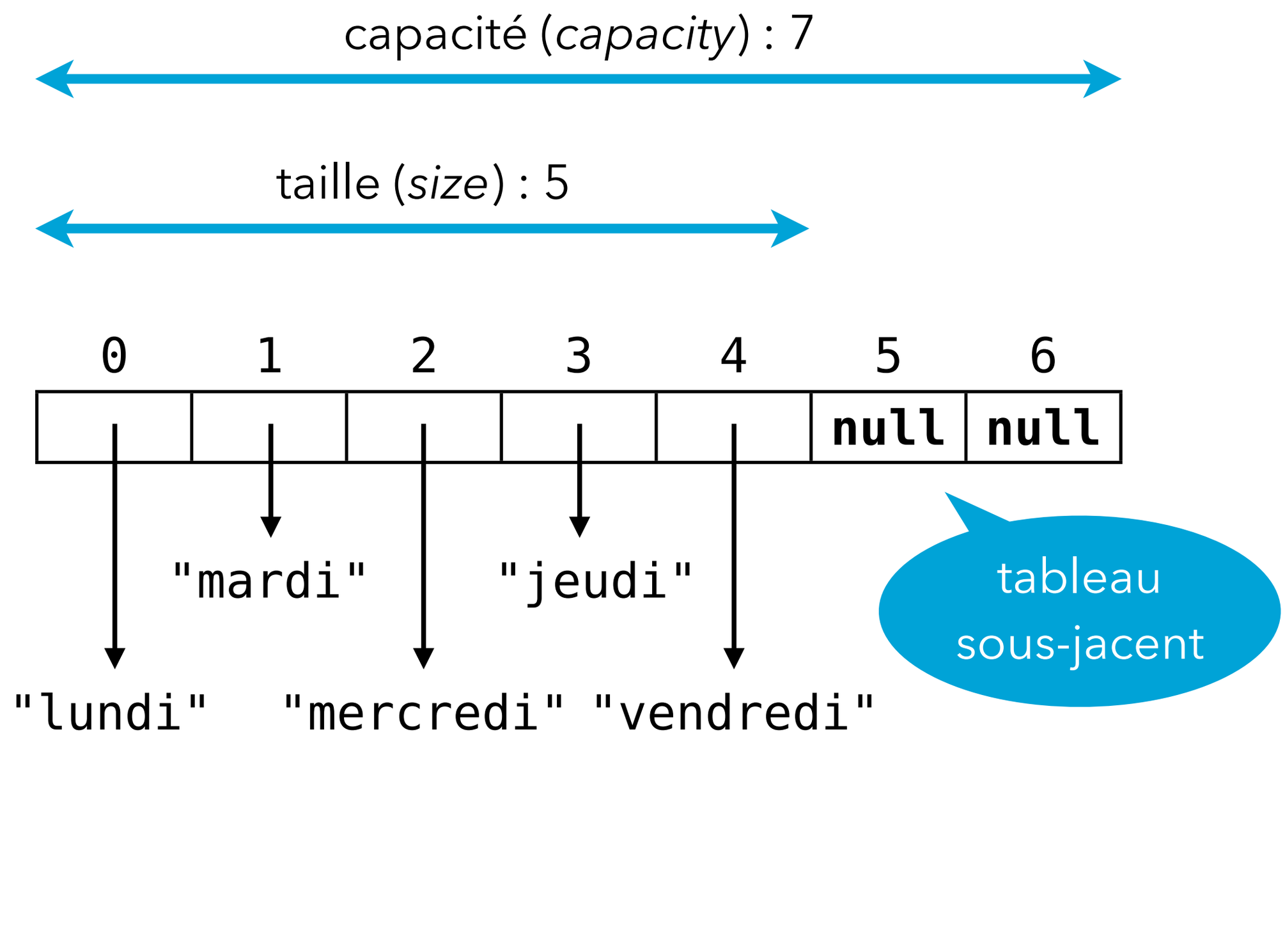
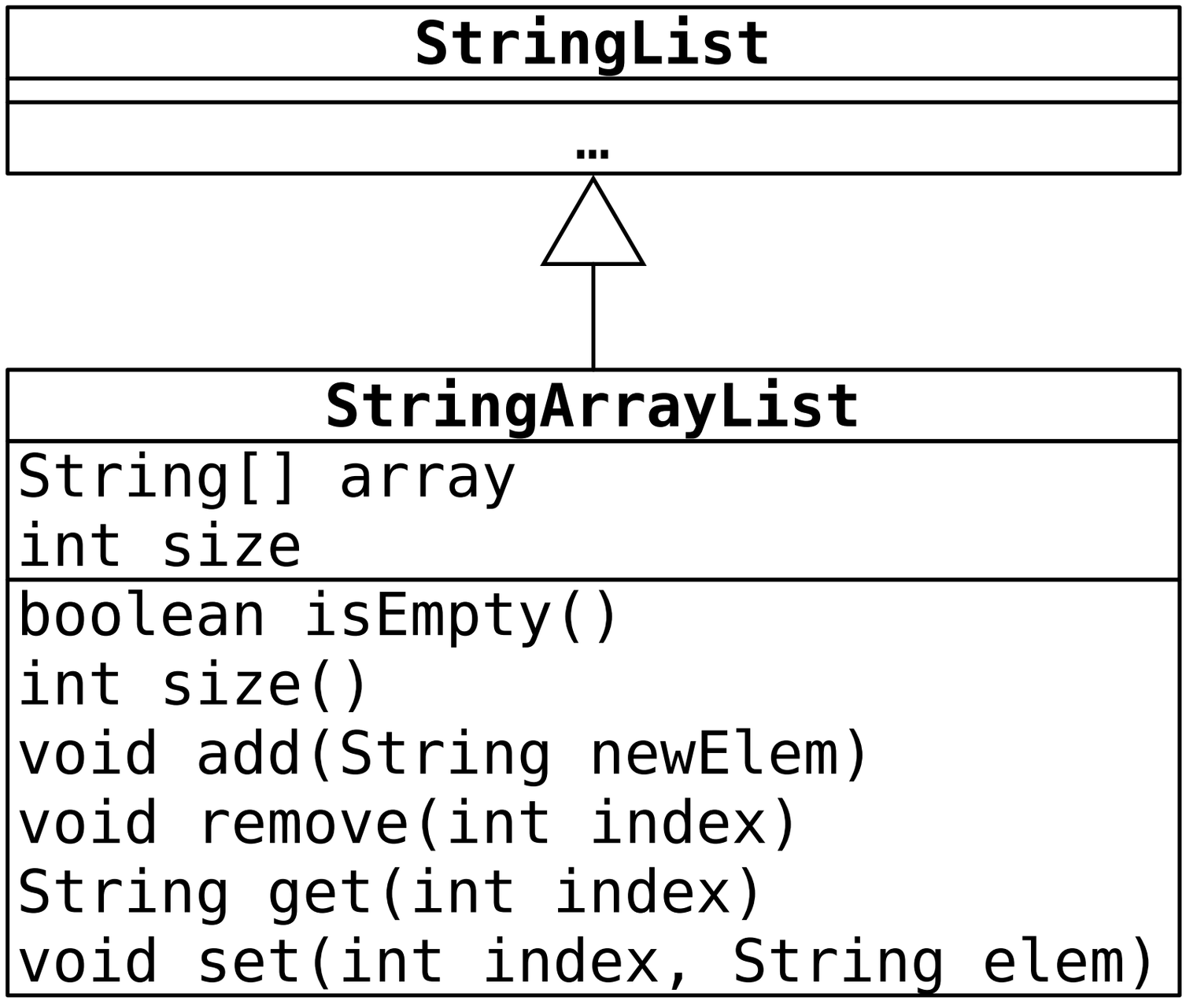


Diagramme de classes



Code (1) : taille

```
class StringArrayList implements StringList {
  private String[] array = new String[1];
  private int size = 0;

  public boolean isEmpty() {
    return size == 0;
  }

  public int size() {
    return size;
  }
  …
}
```

Code (2) : accès

```
class StringArrayList implements StringList {
  …
  public String get(int index) {
    checkIndex(index);
    return array[index];
  }
  public void set(int index, String elem) {
    checkIndex(index);
    array[index] = elem;
  }
  private void checkIndex(int index) {
    if (! (0 <= index && index < size))
      throw new IndexOutOfBoundsException(…);
  }
  …
}
```

Code (3) : ajout

```
class StringArrayList implements StringList {
  …
  public void add(String newElem) {
    if (size == array.length)
      array = Arrays.copyOf(array,
                            array.length * 2);
    array[size++] = newElem;
  }
  …
}
```

`Arrays.copyOf` retourne une copie du tableau donné en 1er argument, de longueur donnée en 2e argument. Les éventuels emplacements vides sont mis à `null`.

Code (4) : suppression

```
class StringArrayList implements StringList {
  …
  public void remove(int index) {
    checkIndex(index);
    System.arraycopy(array, index + 1,
                     array, index,
                     size - 1 - index);
    array[--size] = null;
  }
}
```

`System.arraycopy` copie un certain nombre d'éléments d'un premier tableau, à partir d'un index donné, vers un second tableau (qui peut être le même), à partir d'un autre index.

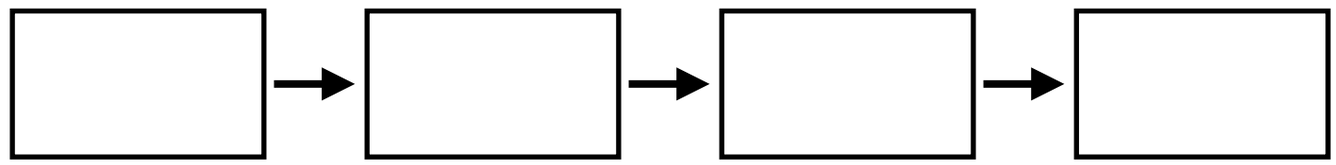
Mise en œuvre 2 : liste chaînée

Liste chaînée

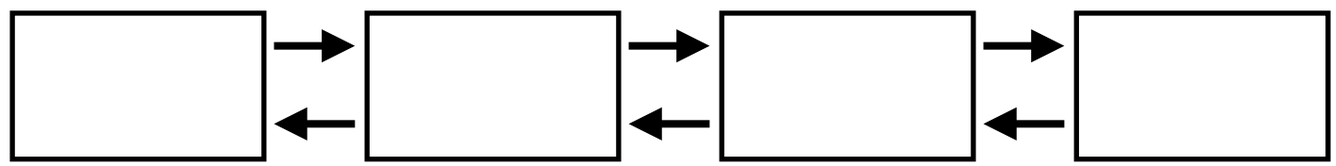
Une **liste chaînée** (*linked list*) est composée d'une séquence de **nœuds** liés entre-eux et référençant les éléments de la liste.
Dans une liste **simplement chaînée** (*singly linked list*), chaque nœud contient uniquement un lien vers son successeur.
Dans une liste **doublement chaînée** (*doubly linked list*), chaque nœud contient un lien vers son successeur et son prédécesseur.
Finalement, une liste est **circulaire** si le premier et le dernier nœuds sont liés entre-eux (simplement ou doublement).

Variantes de chaînage

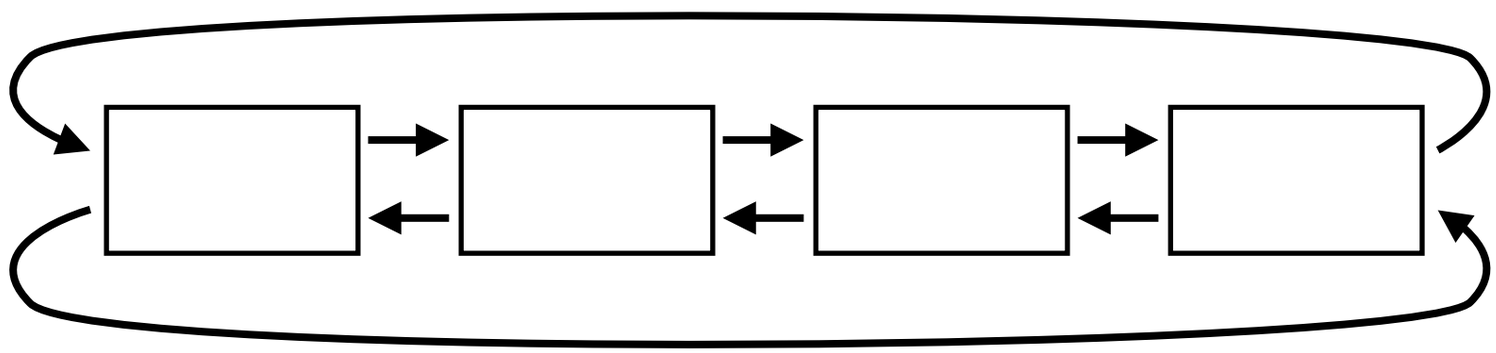
simple
(*singly-linked*)



double
(*doubly-linked*)



double, circulaire
(*doubly-linked, circular*)



Liste chaînée

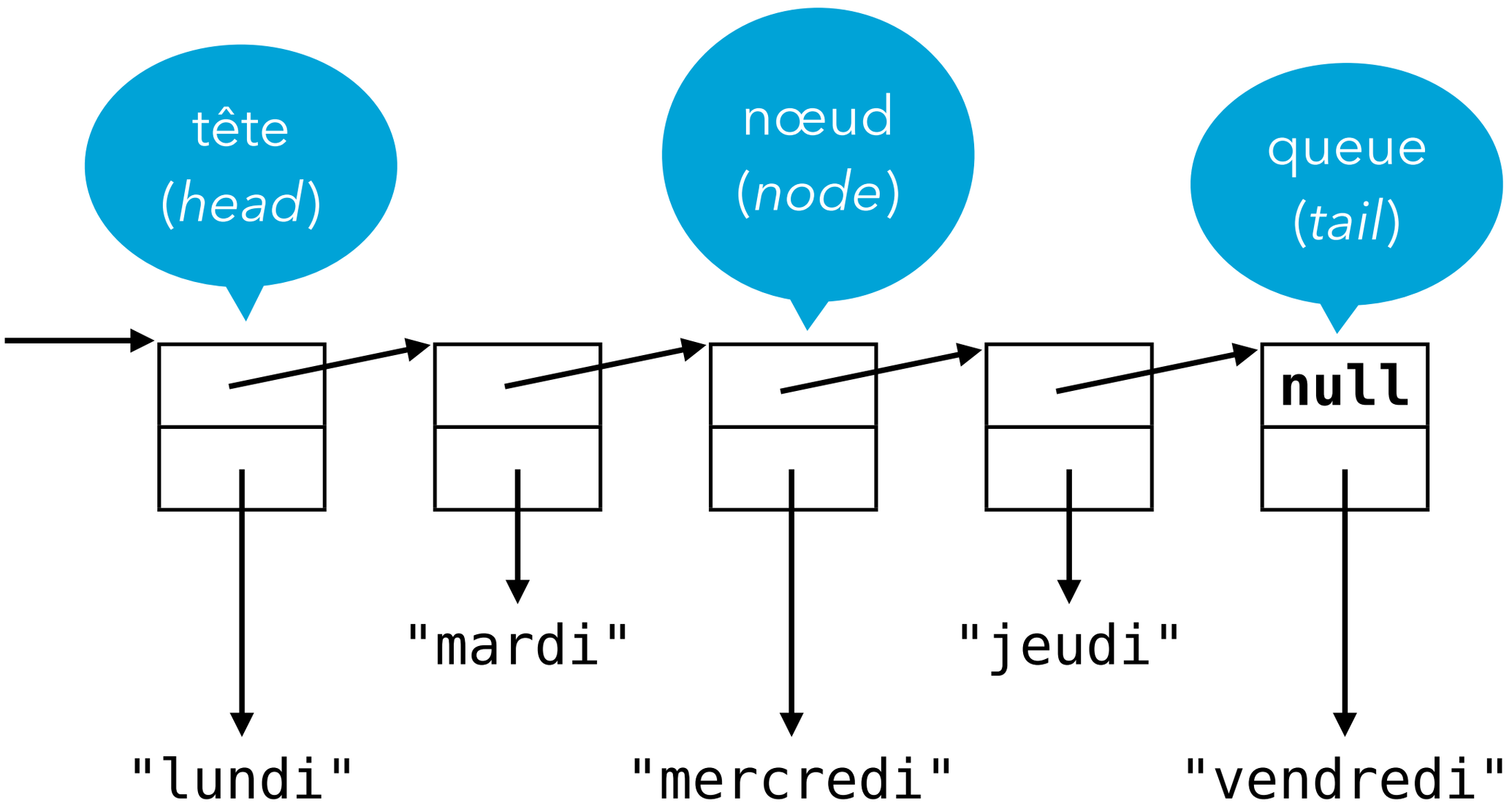
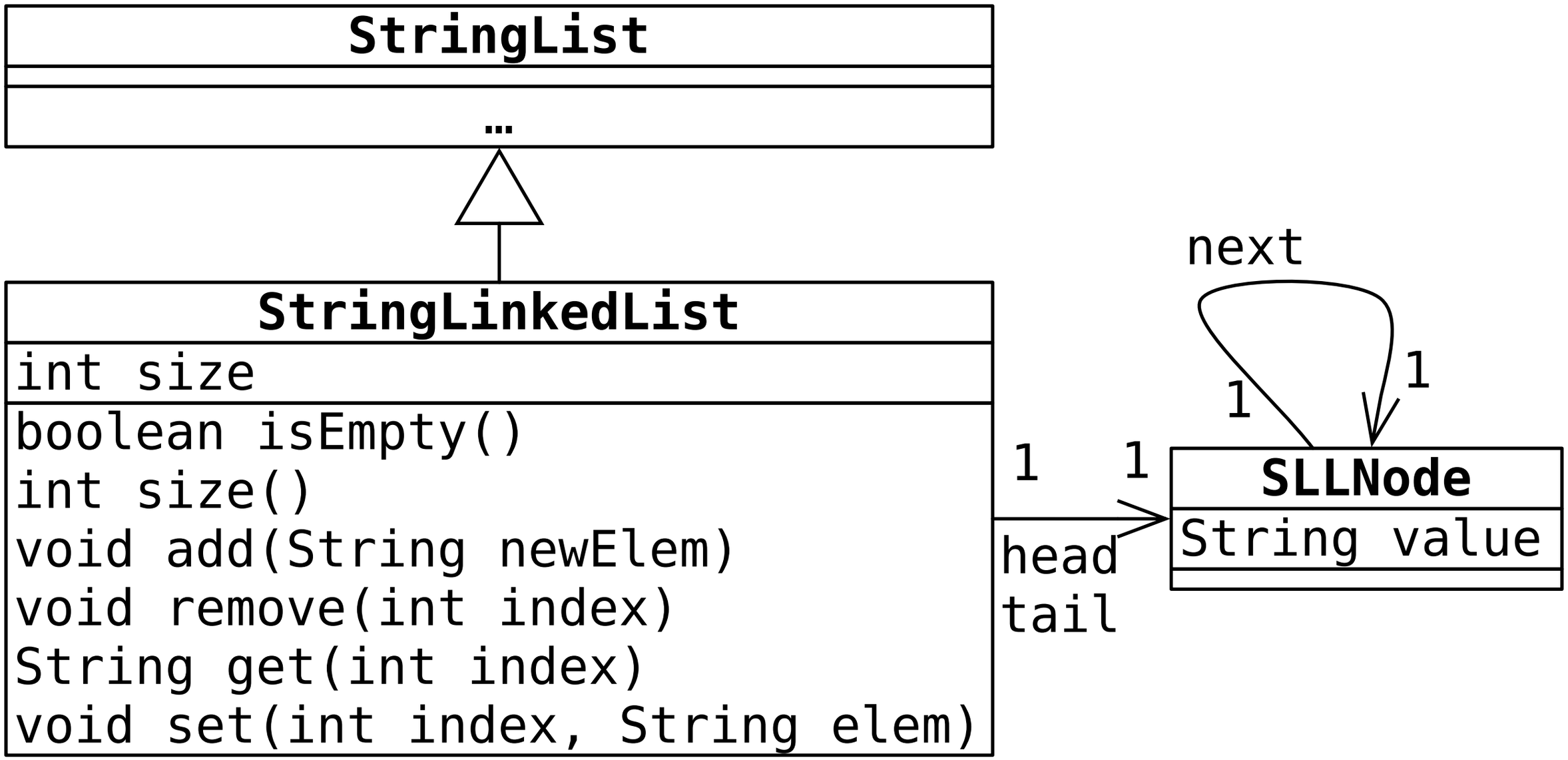


Diagramme de classes



Code (1) : nœuds

La classe **`SLLNode`** modélise un nœud de liste (simplement chaînée).

```
class SLLNode {
  String value;
  SLLNode next;

  public SLLNode(String value) {
    this.value = value;
    this.next = null;
  }
}
```

Code (2) : taille

```
class StringLinkedList implements StringList {
  private SLLNode head = null, tail = null;
  private int size = 0;

  public boolean isEmpty() {
    return size == 0;
  }

  public int size() {
    return size;
  }
  …
}
```

Code (3) : accès

```
class StringLinkedList implements StringList {
  private SLLNode head = null, tail = null;
  private int size = 0;
  …
  public String get(int index) {
    return nodeAtIndex(index).value;
  }
  public void set(int index, String elem) {
    nodeAtIndex(index).value = elem;
  }
  …
  private SLLNode nodeAtIndex(int index) {
    // exercice
  }
}
```

Code (4) : ajout

```
class StringLinkedList implements StringList {
  private SLLNode head = null, tail = null;
  private int size = 0;
  …
  public void add(String s) {
    SLLNode newNode = new SLLNode(s);
    if (size == 0) {
      head = tail = newNode;
    } else {
      tail.next = newNode;
      tail = newNode;
    }
    ++size;
  }
  …
}
```

Parcours de listes

Parcours naïf

La boucle suivante affiche les éléments de la liste `l` dans l'ordre :

```
StringList l = …;
for (int i = 0; i < l.size(); ++i)
  System.out.println(l.get(i));
```

Quel est le problème de cette boucle ?

Itérateur

Pour éviter les problèmes de performance de ce type de parcours, on introduit la notion d'**itérateur** ou **curseur** (*iterator* ou *cursor* en anglais).

Un itérateur est un objet qui désigne un élément d'une collection et qui peut passer (efficacement) d'un élément à son successeur.

Itérateur

Un itérateur offre (au moins) deux opérations :

- le test d'existence d'un successeur à l'élément courant,
- la combinaison de l'obtention de l'élément courant et de l'avancement de l'itérateur sur son successeur.

Un tel itérateur peut être spécifié par l'interface suivante :

```
interface StringIterator {
  boolean hasNext();
  String next();
}
```

Parcours par itérateur

La boucle ci-dessous affiche les éléments de la liste `l` dans l'ordre, mais de manière efficace !

```
StringIterator i = l.iterator();
while (i.hasNext())
  System.out.println(i.next());
```

On fait l'hypothèse que la liste possède une méthode `iterator`, définie p.ex. ainsi pour `StringLinkedList` :

```
public StringIterator iterator() {
  return new SLLIterator(head);
}
```

Itérateur (liste chaînée)

```
class SLLIterator implements StringIterator {
  private SLLNode nextNode;
  SLLIterator(SLLNode head){
    nextNode = head;
  }
  public boolean hasNext() {
    return nextNode != null;
  }
  public String next() {
    String elem = nextNode.value;
    nextNode = nextNode.next;
    return elem;
  }
}
```

Listes arbitraires

L'interface `StringList` et les classes qui l'implémentent ne peuvent représenter que des listes de chaînes de caractères.
Comment faire pour représenter des listes d'un type arbitraire ?

- écrire autant d'interfaces et de classes qu'il existe de types (spécialisation), ou
- faire des listes de `Object`, ou
- utiliser la généricité.

Ces notions seront l'objet du prochain cours.

Paquetages

(digression)

Problème du nommage

Les collections sont utiles dans presque tous les programmes. Dès lors, comment peut-on nommer les classes et interfaces pour éviter les conflits ?
Par exemple, une table associative s'appelle *map* en anglais, il paraît donc logique de nommer `Map` l'interface correspondante.
Mais une application cartographique a de grandes chances d'avoir également un type nommé `Map` pour les cartes !
Comment permettre l'utilisation des deux dans le même programme ?

Mauvaise solution : préfixes

Idée : préfixer tous les noms par une chaîne qu'on espère globalement unique.
Par exemple, l'interface `Map` pour les tables associatives pourrait être nommée `CollMap`, tandis que la classe `Map` des cartes pourrait être nommée `CartoMap`.

De tels préfixes ont plusieurs problèmes :

- ils devraient être longs pour être uniques, mais
- ils devraient être courts pour ne pas gêner, et
- ils doivent être utilisés même en l'absence de conflit puisqu'ils font partie du nom.

Paquetage

Pour tenter de résoudre le problème du nommage, Java offre la notion de **paquetage**.
Un paquetage est une entité nommée qui contient un certain nombre de types (classes et/ou interfaces).
Au début de chaque fichier source, il est possible de spécifier le paquetage dans lequel placer les classes et interfaces qu'il contient au moyen de l'énoncé `package`.

Exemple :

```
package collections;
public interface Map { … }
```

Noms qualifiés

Le nom complet (ou **nom complètement qualifié**, *fully-qualified name* en anglais) d'un type déclaré à l'intérieur d'un paquetage inclut le nom du paquetage.

Ainsi, avec la déclaration suivante :

```
package collections;
public interface Map { … }
```

le nom complètement qualifié de l'interface `Map` est `collections.Map`.

Les paquetages jouent donc un rôle similaire aux préfixes, avec l'avantage de pouvoir être omis dans la majorité des cas, comme nous allons le voir.

Utilisation des noms

Tous les noms d'un paquetage sont utilisables sans préfixe à l'intérieur de ce même paquetage. Cela reste vrai même si un paquetage est réparti sur plusieurs fichiers ! Exemple :

```
package collections;
public class HashMap implements Map { … }
```

A l'extérieur d'un paquetage donné, les noms publics de ce dernier sont utilisables en version totalement qualifiée. Exemple :

```
package wordprocessor;
class Dictionary
  implements collections.Map { … }
```

Importation

Pour éviter de devoir utiliser la version complètement qualifiée des noms définis dans un autre paquetage, il est possible d'**importer** les noms utilisés, au moyen de l'énoncé `import`.

L'exemple précédent peut se récrire ainsi :

```
package wordprocessor;
import collections.Map;
class Dictionary implements Map { … }
```

≡ collections.Map

Il est interdit d'importer deux noms identiques, ou de définir un nom identique à un nom importé.

Tous les énoncés `import` doivent apparaître juste après l'énoncé `package` (et nulle part ailleurs).

Visibilité des noms

Les paquetages influent sur la visibilité des noms :

- Les types des classes ou interfaces qui ne sont pas déclarées `public` sont visibles uniquement dans le paquetage dans lequel ils sont déclarés.
- Les membres qui ne sont déclarés ni `public` ni `private` sont visibles dans le paquetage dans lequel leur propriétaire est déclaré.
- Les membres qui sont déclarés `protected` sont visibles dans le paquetage dans lequel leur propriétaire est déclaré, et dans toutes les sous-classes, indépendamment de leur paquetage.

Hiérarchie

Les paquetages peuvent être organisés en hiérarchie, c-à-d qu'un paquetage peut contenir d'autres paquetages, et ainsi de suite.

Par exemple, il existe un paquetage standard nommé `java`, dans lequel se trouvent plusieurs sous-paquetages comme `java.lang` et `java.util`.

A l'intérieur de ce dernier se trouvent aussi bien des classes et interfaces (p.ex. `List`, `Set`, `Map`) que d'autres sous-paquetages comme `java.util.concurrent`.

Nommage des paquetages

Utiliser des paquetages ne fait que repousser un peu plus loin le problème de l'unicité des noms... Comment éviter que deux programmeurs définissent des paquetages de même nom ?

Idée : utiliser le nom de domaine Internet de l'organisation (unique), inversé, comme préfixe du nom de paquetage.

Exemple : le nom de domaine de l'EPFL est `epfl.ch`. Tous les paquetages développés à l'EPFL peuvent commencer par `ch.epfl`, p.ex. `ch.epfl.collections`.

Problème : les organisations sont renommées, rachetées (Sun par Oracle, p.ex.), disparaissent, etc.

Répertoires et fichiers

Le contenu d'un paquetage peut être réparti sur plusieurs fichiers. Chacun d'entre-eux doit commencer avec un énoncé `package` approprié.
Les fichiers doivent être placés dans des répertoires portant le nom du paquetage.
Par exemple, si l'interface `Map` est déclarée à l'intérieur d'un fichier débutant ainsi :

```
package java.util;
public interface Map { … }
```

alors ce fichier doit être stocké dans un répertoire nommé `util`, lui-même placé dans un répertoire nommé `java`.
Et bien entendu, le fichier doit être nommé `Map.java`.

Résumé

Une collection est un objet qui stocke un certain nombre d'autres objets.
Nous étudierons les tableaux, listes, ensembles et tables associatives. Pour chacune, plusieurs mises en œuvre sont possibles.
Pour les listes, nous en avons examiné deux : les tableaux-listes, mis en œuvre via un tableau « redimensionné » au besoin, et les listes chaînées, mises en œuvre via des nœuds liés entre-eux.

* * *

Les paquetages permettent de grouper des classes et interfaces constituant un tout, et d'éviter les conflits de nommage en partitionnant l'espace de noms.