Les Conteneurs Java

Licence d'informatique 3ème année

Préambule

It’s a fairly simple program that has only a fixed quantity of objetcs with known lifetimes

Introduction

- Lors de leur exécution, les programmes que vous écrirez créeront souvent de nouveaux objets basés sur des informations qui ne seront pas connues avant le lancement du programme.
- Le nombre voire même le type des objets nécessaire ne seront pas connus avant la phase d'exécution du programme.
- Il faut donc être capable de créer un certain nombre d'objets, de les stocker et de les manipuler.
- Le simple usage de références nommées sur ces objets ne peut suffire.

Introduction

- Vous avez étudiez des structures de données (file, pile, liste, arbre, etc), et même sous forme de TDA.
- Ces structures de données ainsi que les tableaux permettent justement de stocker et de manipuler des objets créés lors de l'exécution.
- Java propose une bibliothèque de classes qui fournit des implémentations de nombreuses structures de données permettant de stocker et de manipuler des références d'objets.
- L'utilisation des ces classes devrait augmenter de façon significative votre productivité de développement (et ce ne sera pas du luxe !!!)

Introduction

- La bibliothèque de conteneurs de Java 2 se divise en deux concepts distincts :
- Collection : un groupe d'éléments individuels, souvent associés à une règle définissant leur comportement, par exemple :
  - dans une List les éléments sont ordonnés ;
  - dans un Set il ne peut y avoir de doublon.
- Map : un ensemble de paires clef – valeur.
  - Il est possible d'extraire un sous-ensemble d'une Map dans une Collection.
  - Une Map peut donc renvoyer un Set de ses clefs, une Collection de ses valeurs, etc.

Le type des objets

- Les conteneurs sont conçus pour stocker n'importe quel type d'objet.
- Les conteneurs stockent donc des références sur des Object puisque quelque soit sont type, un objet descend toujours de la classe Object.
- Ceci a pour but la généricité des conteneurs.
- Et permet de stocker des objets hétérogènes dans un même conteneur.

Le type des objets

- Cette solution a toutefois 2 inconvénients :
  - Puisque l'information de type est ignorée lorsqu'on stocke une référence dans un conteneur, on ne peut placer aucune restriction sur le type de l'objet stocké dans le conteneur, même si on l'a créé pour ne contenir, par exemple, que des chats. Quelqu'un pourrait très bien ajouter un chien dans le conteneur.
  - Puisque l'information de type est perdue, la seule chose que le conteneur sache est qu'il contient une référence sur un objet. Il faut réaliser un transtypage sur le type adéquat avant de l'utiliser.

Le type des objets

```
public class Cat {
  private int catNumber;
  Cat(int i) { catNumber = i; }
  void print() {
    System.out.println("Cat #" + catNumber);
  }
}


public class Dog {
  private int dogNumber;
  Dog(int i) { dogNumber = i; }
  void print() {
    System.out.println("Dog #" + dogNumber);
  }
}
```

Le type des objets

```
import java.util.*;

public class CatsAndDogs {
  public static void main(String[] args) {
    ArrayList cats = new ArrayList();
    for(int i = 0; i < 7; i++)
      cats.add(new Cat(i));
    // Ce n'est pas un problème d'ajouter un chien parmi les chats :
    cats.add(new Dog(7));
    for(int i = 0; i < cats.size(); i++)
      ((Cat)cats.get(i)).print();
    // Le chien est détecté seulement lors de l'exécution.
  }
}
```

Créer une ArrayList consciente du type

```
public class MouseList {
  private ArrayList list = new ArrayList();
  public void add(Mouse m) {
    list.add(m);
  }
  public Mouse get(int index) {
    return (Mouse)list.get(index);
  }
  public int size() { return list.size(); }
}
public class MouseListTest {
  public static void main(String[] args) {
    MouseList mice = new MouseList();
    for(int i = 0; i < 3; i++)
      mice.add(new Mouse(i));
    for(int i = 0; i < mice.size(); i++)
      MouseTrap.caughtYa(mice.get(i));
  }
}
```

Afficher les conteneurs

- Pour afficher un conteneur on peut simplement utiliser la méthode :
  - System.out.println
- Affichage d'une ArrayList :
  - [dog, dog, cat]
- Affichage d'un HashSet :
  - [cat, dog]
- Affichage d'une HashMap :
  - [cat=Rags, dog=Spot]

Les Itérateurs

- Un itérateur est un objet dont le travail est de se déplacer dans une séquence d'objets et de sélectionner chaque objet de cette séquence sans que le programmeur client n'ait à se soucier de la structure sous-jacente de cette séquence.
- Un itérateur est généralement ce qu'il est convenu d'appeler un objet « léger » : un objet bon marché à construire. Pour cette raison, vous trouverez souvent des contraintes étranges sur les itérateurs ; par exemple, certains itérateurs ne peuvent se déplacer que dans un sens.

Les Itérateurs

- L'Iterator Java est l'exemple type d'un itérateur avec ce genre de contraintes. On ne peut pas faire grand-chose avec mis à part :
  - Demander à un conteneur de renvoyer un Iterator en utilisant une méthode appelée iterator(). Cet Iterator sera prêt à renvoyer le premier élément dans la séquence au premier appel à sa méthode next().
  - Récupérer l'objet suivant dans la séquence grâce à sa méthode next().
  - Vérifier s'il reste encore d'autres objets dans la séquence via la méthode hasNext().
  - Enlever le dernier élément renvoyé par l'itérateur avec la méthode remove().

Les Itérateurs

```
import java.util.*;

public class CatsAndDogs2 {
  public static void main(String[] args) {
    ArrayList cats = new ArrayList();
    for(int i = 0; i < 7; i++)
      cats.add(new Cat(i));
    Iterator e = cats.iterator();
    while(e.hasNext())
      ((Cat)e.next()).print();
  }
}
```

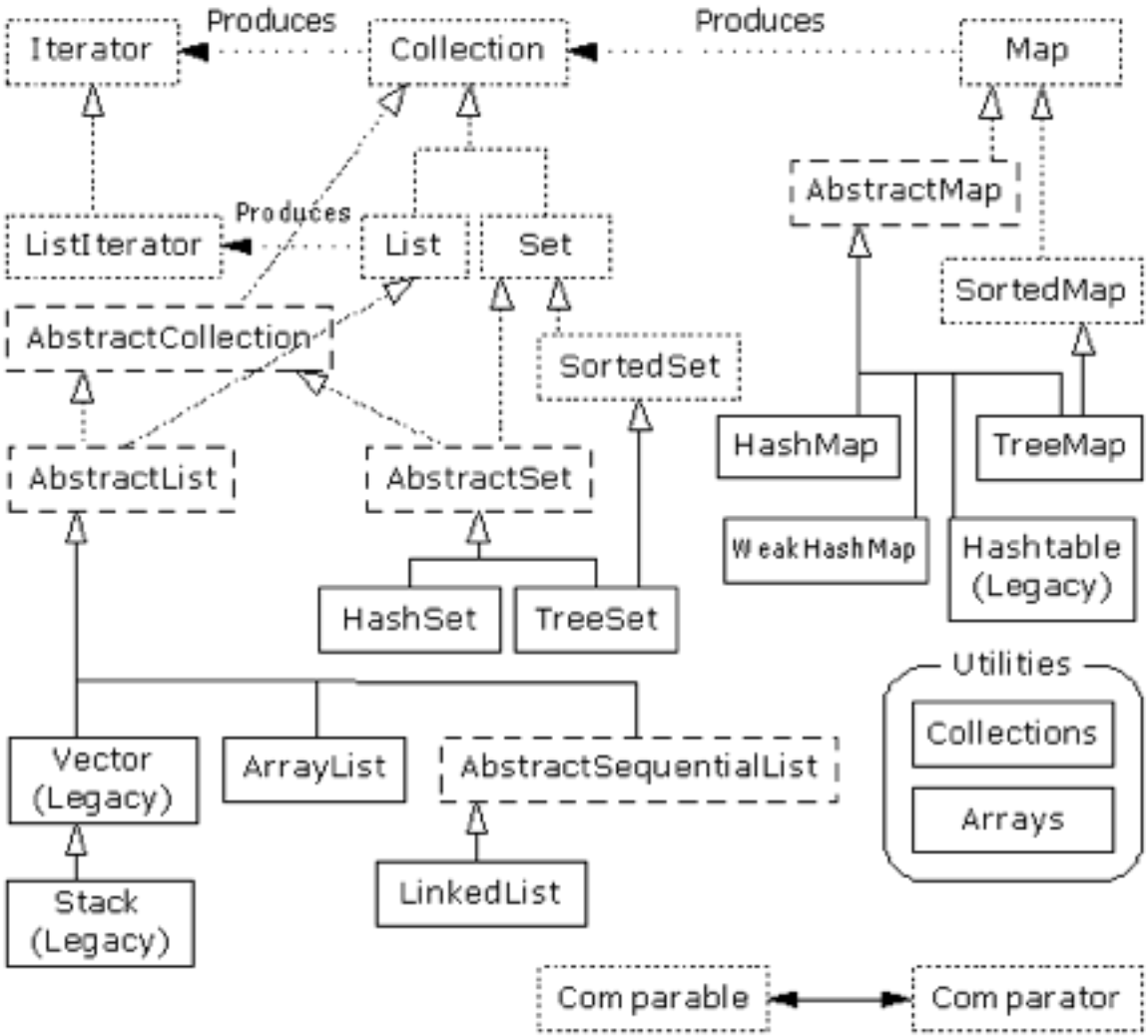
Les Itérateurs

- L'utilisation d'itérateurs permet d'écrire des portions de code générique, i.e qui ne dépendent pas du conteneur ayant construit l'itérateur.
- Ainsi on peut réutiliser une même portion de code pour une ArrayList que pour une LinkedList.
- Voyons un exemple…

Les Itérateurs

```
class Hamster {
  private int hamsterNumber;
  Hamster(int i) { hamsterNumber = i; }
  public String toString() {
    return "This is Hamster #" + hamsterNumber;
  }
}
class Printer {
  static void printAll(Iterator e) {
    while(e.hasNext())
      System.out.println(e.next());
  }
}
public class HamsterMaze {
  public static void main(String[] args) {
    ArrayList v = new ArrayList();
    for(int i = 0; i < 3; i++)
      v.add(new Hamster(i));
    Printer.printAll(v.iterator());
  }
}
```

Classification des conteneurs



Classification des conteneurs

- Les interfaces concernées par le stockage des objets sont Collection, List, Set et Map.
- Idéalement, la majorité du code qu'on écrit sera destinée à ces interfaces, et le seul endroit où on spécifiera le type précis utilisé est lors de la création.
- On pourra donc créer une List de cette manière :
  - `List x = new LinkedList();`

Classification des conteneurs

- Bien sûr, on peut aussi décider de faire de **x** (de l'exemple précédent) une **LinkedList** (au lieu d'une **List** générique) et véhiculer le type précis d'informations avec **x**.
- La beauté (et l'intérêt) de l'utilisation d'une interface est que si on décide de changer l'implémentation, la seule chose qu'on ait besoin de changer est l'endroit où la liste est créée, comme ceci :
  - `List x = new ArrayList();`
- Et on ne touche pas au reste du code

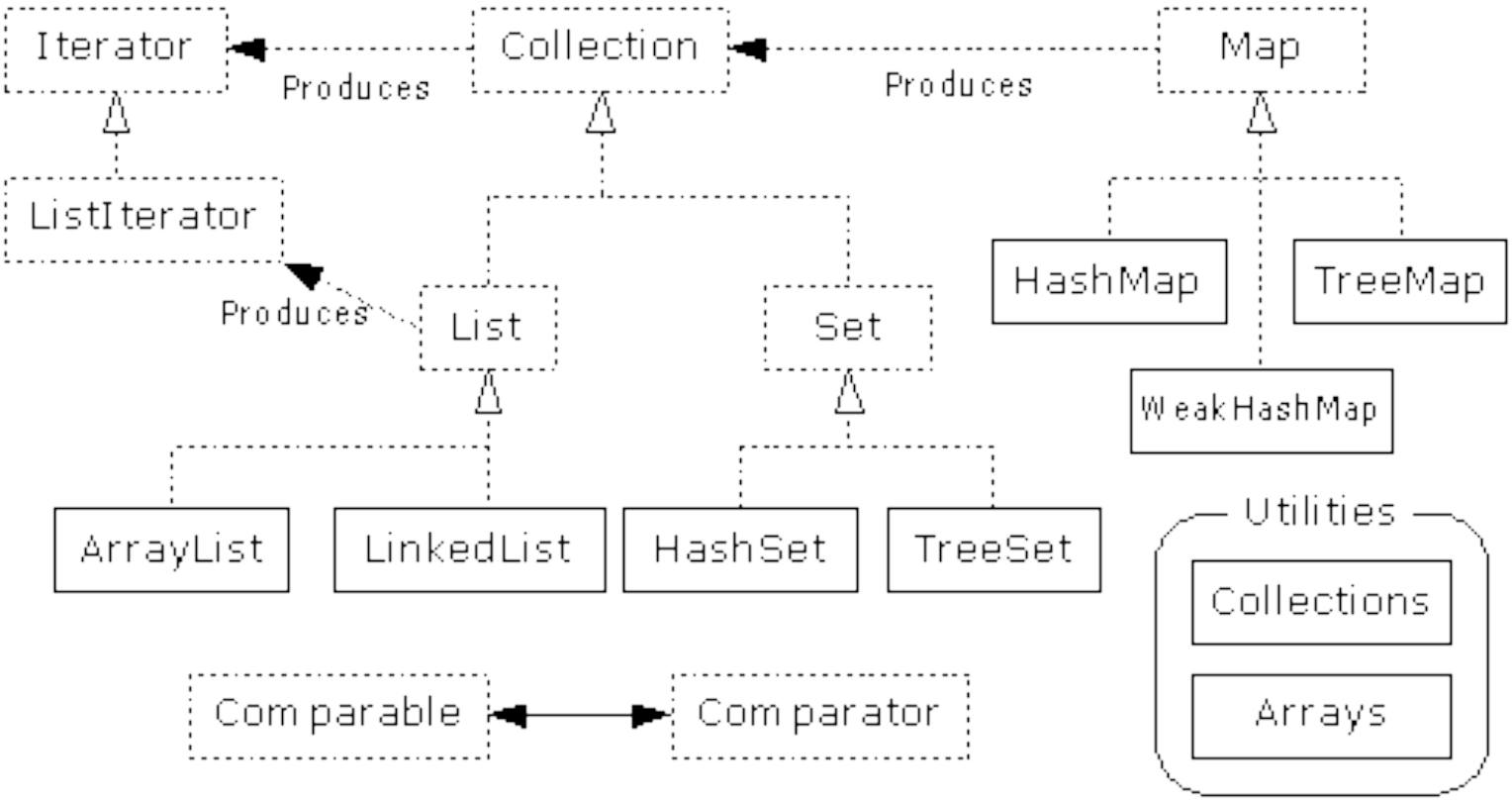
Classification des conteneurs

- Dans la hiérarchie de classes, on peut voir un certain nombre de classes dont le nom débute par « **Abstract** », ce qui peut paraître un peu déroutant au premier abord.
- Ce sont simplement des outils qui implémentent partiellement une interface particulière.
- Si on voulait réaliser notre propre Set, par exemple, il serait plus simple de dériver AbstractSet et de réaliser le travail minimum pour créer la nouvelle classe, plutôt que d'implémenter l'interface Set et toutes les méthodes qui vont avec.
- Cependant, la bibliothèque de conteneurs possède assez de fonctionnalités pour satisfaire quasiment tous nos besoins. De notre point de vue, nous pouvons donc ignorer les classes débutant par « **Abstract** ».

Classification des conteneurs

- Quand on regarde le diagramme, on n'est réellement concerné que par les **interface**s du haut du diagramme et les classes concrètes (celles qui sont entourées par des boîtes solides).
- Le plus souvent, on se contentera de créer un objet d'une classe concrète, de la transtyper dans son **interface** correspondante, et ensuite utiliser cette **interface** tout au long du code.

Classification des conteneurs



Un exemple simple

```
import java.util.*;

public class SimpleCollection {
  public static void main(String[] args) {
    // Transtypage ascendant parce qu'on veut juste
    // travailler avec les fonctionnalités d'une Collection
    Collection c = new ArrayList();
    for(int i = 0; i < 10; i++)
      c.add(Integer.toString(i));
    Iterator it = c.iterator();
    while(it.hasNext())
      System.out.println(it.next());
  }
}
```

Un exemple simple

- La première ligne de **main()** crée un objet **ArrayList** et le transtype ensuite en une **Collection**. Puisque cet exemple n'utilise que les méthodes de **Collection**, tout objet d'une classe dérivée de **Collection** fonctionnerait, mais l'**ArrayList** est la **Collection** à tout faire typique.
- La méthode **add()**, comme son nom le suggère, ajoute un nouvel élément dans la **Collection**. En fait, la documentation précise bien que **add()** « assure que le conteneur contiendra l'élément spécifié ». Cette précision concerne les **Set**s, qui n'ajoutent un élément que s'il n'est pas déjà présent. Avec une **ArrayList**, ou n'importe quel type de **List**, **add()** veut toujours dire « stocker dans », parce qu'une **List** se moque de contenir des doublons.
- Toutes les **Collection**s peuvent produire un **Iterator** grâce à leur méthode **iterator()**. Ici, un **Iterator** est créé et utilisé pour traverser la **Collection**, en affichant chaque élément.

Fonctionnalités des Collections

- **boolean add(Object)**
  - Assure que le conteneur stocke l'argument. Renvoie **false** si elle n'ajoute pas l'argument.
- **boolean addAll(Collection)**
  - Ajoute tous les éléments de l'argument. Renvoie **true** si un élément a été ajouté.
- **void clear()**
  - Supprime tous les éléments du conteneur.
- **boolean contains(Object)**
  - **true** si le conteneur contient l'argument.

Fonctionnalités des Collections

- **boolean containsAll(Collection)**
  - **true** si le conteneur contient tous les éléments de l'argument.
- **boolean isEmpty()**
  - **true** si le conteneur ne contient pas d'éléments.
- **Iterator iterator()**
  - Renvoie un **Iterator** qu'on peut utiliser pour parcourir les éléments du conteneur.
- **boolean remove(Object)**
  - Si l'argument est dans le conteneur, une instance de cet élément est enlevée. Renvoie **true** si c'est le cas.

Fonctionnalités des Collections

- **boolean removeAll(Collection)**
  - Supprime tous les éléments contenus dans l'argument. Renvoie **true** si au moins une suppression a été effectuée.
- **boolean retainAll(Collection)**
  - Ne garde que les éléments contenus dans l'argument (une « intersection » selon la théorie des ensembles). Renvoie **true** s'il y a eu un changement.
- **int size()**
  - Renvoie le nombre d'éléments dans le conteneur.

Fonctionnalités des Collections

- **Object[] toArray()**
  - Renvoie un tableau contenant tous les éléments du conteneur.
- **Object[] toArray(Object[] a)**
  - Renvoie un tableau contenant tous les éléments du conteneur, dont le type est celui du tableau **a** au lieu d'**Object**s génériques (il faudra toutefois transtyper le tableau dans son type correct).

Fonctionnalités des Lists

- La **List** de base est relativement simple à utiliser, comme vous avez pu le constater jusqu'à présent avec les **ArrayList**s.
- Mis à part les méthodes courantes **add()** pour insérer des objets, **get()** pour les retrouver un par un, et **iterator()** pour obtenir un **Iterator** sur la séquence, les listes possèdent par ailleurs tout un ensemble de méthodes qui peuvent se révéler très pratiques.

Fonctionnalités des Lists

- Les **List**s sont déclinées en deux versions :
  - l'**ArrayList** de base, qui excelle dans les accès aléatoires aux éléments,
  - la **LinkedList**, bien plus puissante (qui n'a pas été conçue pour un accès aléatoire optimisé, mais dispose d'un ensemble de méthodes bien plus conséquent).

Fonctionnalités des Lists

- L'interface List
  - L'ordre est la caractéristique la plus importante d'une **List** ; elle garantit de maintenir les éléments dans un ordre particulier.
  - Les **List**s disposent de méthodes supplémentaires permettant l'insertion et la suppression d'éléments au sein d'une **List** (ceci n'est toutefois recommandé que pour une LinkedList).
  - Une **List** produit des **ListIterator**s, qui permettent de parcourir la **List** dans les deux directions, d'insérer et de supprimer des éléments au sein de la **List**.

Fonctionnalités des Lists

- La classe ArrayList
  - Une **List** implémentée avec un tableau.
  - Permet un accès aléatoire instantané aux éléments, mais se révèle inefficace lorsqu'on insère ou supprime un élément au milieu de la liste.
  - Le **ListIterator** ne devrait être utilisé que pour parcourir l'**ArrayList** dans les deux sens, et non pour l'insertion et la suppression d'éléments, opérations coûteuses comparées aux **LinkedLists**.

Fonctionnalités des Lists

- La classe LinkedList
  - Fournit un accès séquentiel optimal, avec des coûts d'insertion et de suppression d'éléments au sein de la **List** négligeables.
  - Relativement lente pour l'accès aléatoire (préférer une **ArrayList** pour cela).
  - Fournit aussi les méthodes **addFirst()**, **addLast()**, **getFirst()**, **getLast()**, **removeFirst()** et **removeLast()** (qui ne sont définies dans aucune interface ou classe de base) afin de pouvoir l'utiliser comme une pile, une file (une queue) ou une file double (queue à double entrée).

Réaliser une pile à partir d'une LinkedList

```
import java.util.*;

public class StackL {
  private LinkedList list = new LinkedList();
  public void push(Object v) {
    list.addFirst(v);
  }
  public Object top() { return list.getFirst(); }
  public Object pop() {
    return list.removeFirst();
  }
```

Réaliser une file à partir d'une LinkedList

```
import java.util.*;

public class Queue {
  private LinkedList list = new LinkedList();
  public void put(Object v) { list.addFirst(v); }
  public Object get() {
    return list.removeLast();
  }
  public boolean isEmpty() {
    return list.isEmpty();
  }
```

Fonctionnalités des Sets

- Les **Set**s ont exactement la même interface que les **Collection**s, et à l'inverse des deux différentes **List**s, ils ne proposent aucune fonctionnalité supplémentaire.
- Les **Set**s sont donc juste une **Collection** ayant un comportement particulier (implémenter un comportement différent constitue l'exemple type où il faut utiliser l'héritage et le polymorphisme).
- Un **Set** refuse de contenir plus d'une instance de chaque valeur d'un objet (savoir ce qu'est la « valeur » d'un objet est plus compliqué, comme nous allons le voir).

Fonctionnalités des Sets

- L'interface **Set**
  - Chaque élément ajouté au **Set** doit être unique ; sinon le **Set** n'ajoutera pas le doublon.
  - Les **Object**s ajoutés à un **Set** doivent définir la méthode **equals()** pour pouvoir établir l'unicité de l'objet.
  - Un **Set** possède la même interface qu'une **Collection**.
  - L'interface **Set** ne garantit pas que les éléments sont maintenus dans un ordre particulier.

Fonctionnalités des Sets

- La classe **HashSet**
  - Pour les **Set**s où le temps d'accès aux éléments est primordial.
  - Les **Object**s doivent définir la méthode **hashCode()**.
- La classe **TreeSet**
  - Un **Set** trié stocké dans un arbre. De cette manière, on peut extraire une séquence triée à partir du **Set**.

Sets triés : les SortedSets

- Un **SortedSet** (dont **TreeSet** est l'unique représentant) garantit que ses éléments seront stockés triés, ce qui permet de proposer de nouvelles fonctionnalités grâce aux méthodes supplémentaires de l'interface **SortedSet** suivantes :
  - **Comparator comparator()** : Renvoie le **Comparator** utilisé pour ce **Set**, ou **null** dans le cas d'un tri naturel.
  - **Object first()** : Renvoie le plus petit élément.
  - **Object last()** : Renvoie le plus grand élément.
  - **SortedSet subSet(fromElement, toElement)** : Renvoie une vue du **Set** contenant les éléments allant de **fromElement** inclus à **toElement** exclu.
  - **SortedSet headSet(toElement)** : Renvoie une vue du **Set** contenant les éléments inférieurs à **toElement**.
  - **SortedSet tailSet(fromElement)**: Renvoie une vue du **Set** contenant les éléments supérieurs ou égaux à **fromElement**.

Fonctionnalités des Maps

- Un *tableau associatif*, ou *map*, ou *dictionnaire* offre une fonctionnalité particulièrement puissante de « sélection dans une séquence ».
- Conceptuellement, cela ressemble à une **ArrayList**, mais au lieu de sélectionner un objet par un nombre, on le sélectionne en utilisant un *autre objet* !
- Ce fonctionnement est d'une valeur inestimable dans un programme.

Fonctionnalités des Maps

- Le concept est illustré dans Java via l'interface **Map**.
- La méthode **put(Object key, Object value)** ajoute une valeur (la chose qu'on veut stocker), et l'associe à une clef (la chose grâce à laquelle on va retrouver la valeur).
- La méthode **get(Object key)** renvoie la valeur associée à la clef correspondante.
- Il est aussi possible de tester une **Map** pour voir si elle contient une certaine clef ou une certaine valeur avec les méthodes **containsKey()** et **containsValue()**.

Fonctionnalités des Maps

- La bibliothèque Java standard propose deux types de **Map**s : **HashMap** et **TreeMap**.
- Les deux implémentations ont la même interface (puisqu'elles implémentent toutes les deux **Map**), mais diffèrent sur un point particulier : les performances.
- Au lieu d'effectuer une recherche lente sur la clef, un HashMap utilise une valeur spéciale appelée *code de hachage* (*hash code*).
- Le code de hachage est une façon d'extraire une partie de l'information de l'objet en question et de la convertir en un **int** « relativement unique ».
- Tous les objets Java peuvent produire un code de hachage, et **hashCode()** est une méthode de la classe racine **Object**. Un **HashMap** récupère le **hashCode()** de l'objet et l'utilise pour retrouver rapidement la clef. Le résultat en est une augmentation drastique des performances.

Fonctionnalités des Maps

- L'interface **Map**
  - Maintient des associations clef - valeur (des paires), afin de pouvoir accéder à une valeur en utilisant une clef.
- La classe **HashMap**
  - Implémentation basée sur une table de hachage (utilisez ceci à la place d'une **Hashtable**).
  - Fournit des performances constantes pour l'insertion et l'extraction de paires.
  - Les performances peuvent être ajustées via des constructeurs qui permettent de positionner la *capacité* et le *facteur de charge* de la table de hachage.

Fonctionnalités des Maps

- La classe **TreeMap**
  - Implémentation basée sur un arbre rouge-noir.
  - L'extraction des clefs ou des paires fournit une séquence triée (selon l'ordre spécifié par **Comparable** ou **Comparator**, comme nous le verrons plus loin).
  - Le point important dans un **TreeMap** est qu'on récupère les résultats dans l'ordre.
  - **TreeMap** est la seule **Map** disposant de la méthode **subMap()**, qui permet de renvoyer une portion de l'arbre.

Exemple d'utilisation d'une HashMap(1)

```
import java.util.*;

public class TestHashMap {

  public static void main(String[] args) {

    HashMap  hmap = new HashMap();
    hmap.put(new Integer(3), "donnees 3");
    hmap.put(new Integer(1), "donnees 1");
    hmap.put(new Integer(2), "donnees 2");

    System.out.println(hmap.get(new Integer(2)));

  }
}
```

Exemple d'utilisation d'une

```
import java.util.*;
class Counter {
  private int i = 1;
  public void plus(){i++;};
  public String toString() {
    return Integer.toString(i);
  }
}
public class Statistics {
  public static void main(String[] args) {
    HashMap hm = new HashMap();
    for(int i = 0; i < 10000; i++) {
       Integer r = new Integer((int)(Math.random() * 20));
      if(hm.containsKey(r))
        ((Counter)hm.get(r)).plus();
      else
        hm.put(r, new Counter());
    }
    System.out.println(hm);
  }
}
```

Maps triées : les SortedMaps

- Une **SortedMap** (dont **TreeMap** est l'unique représentant) garantit que ses éléments seront stockés triés selon leur clef, ce qui permet de proposer de nouvelles fonctionnalités grâce aux méthodes supplémentaires de l'interface **SortedMap** suivantes :
  - **Comparator comparator()** :
    - Renvoie le **Comparator** utilisé pour cette **Map**, ou **null** dans le cas d'un tri naturel.

Maps triées : les SortedMaps

- **Object firstKey()** :
  - Renvoie la plus petite clef.
- **Object lastKey()** :
  - Renvoie la plus grande clef.
- **SortedMap subMap(fromKey, toKey)** :
  - Renvoie une vue de la **Map** contenant les paires dont les clefs vont de **fromKey** inclus à **toKey** exclu.
- **SortedMap headMap(toKey)** :
  - Renvoie une vue de la **Map** contenant les paires dont la clef est inférieure à **toKey**.
- **SortedMap tailMap(fromKey)** :
  - Renvoie une vue de la **Map** contenant les paires dont la clef est supérieure ou égale à **fromKey**.

Implémentations ordonnées des Set et Map

- À partir du JDK 1.2 on dispose de la possibilité de maîtriser l'ordre dans lequel se présenteront les éléments d'un Set ou d'une Map lors de l'utilisation d'un itérateur.
- Ce sont les implémentations TreeSet de SortedSet et TreeMap de SortedMap qui fournissent cette fonctionnalité.
- Commençons par voir le cadre fournit par Java pour définir un ordre sur des objets.

Comment définir un ordre sur des objets

- Il y a deux manières de définir un ordre sur des objets, soit de façon externe, où on introduit un objet *comparateur* qui est capable de dire si deux objets sont dans l'ordre ; soit de façon interne où ce sont les objets eux-mêmes qui sont capables de se comparer entre-eux, on parle alors d'*ordre naturel*.
- La première manière est la plus souple puisqu'elle permet de définir plusieurs ordres différents pour un même type d'objet. Ceci est indispensable si on veut par exemple pouvoir parcourir une promotion d'étudiants par noms croissants et, séparément, par moyennes décroissantes.

Ordre défini par un Comparator

- Il s'agit de donner une implémentation de l'interface java.util.Comparator :

```
public interface Comparator {
  public int compare(Object o1, Object o2) ;
  public boolean equals(Object obj) ;
}
```

- La méthode principale est compare() qui renvoie un entier :
  - négatif, si o1 est strictement inférieur à o2
  - nul, si o1 est égal à o2
  - Positif, si o1 est strictement supérieur à o2

Ordre défini par un Comparator

- Il faut bien sûr que o1 et o2 soient comparables, c'est à dire qu'ils doivent être compatibles avec un type commun qui permet de définir la relation d'ordre recherchée. Si ce n'est pas le cas, il suffit de déclencher une exception ClassCastException. C'est très simple puisque cette exception est automatiquement déclenchée par une indication de type invalide.
- Cette technique permet d'ordonner la collection d'objets suivant plusieurs ordres : il suffit d'implémenter autant de fois que nécessaire cette interface.

Ordre défini par un Comparator

- Par exemple, on veut disposer de deux ordres différents sur des étudiants : un ordre par noms croissants et un ordre par moyennes décroissantes.
- Ici le type commun avec lequel doivent être compatibles les deux objets est tout simplement Etudiant.

Ordre défini par un Comparator

```
class NomsCroissants implements java.util.Comparator {
  public int compare (Object o1, Object o2) {
    return compare ((Etudiant) o1, (Etudiant) o2) ;
  }
  private int compare (Etudiant e1, Etudiant e2) {
    return e1.nom ().compareTo (e2.nom ()) ;
    // Ordre natuel de String
  }
}
```

Ordre défini par un Comparator

```
class MoyennesDecroissantes implements
java.util.Comparator {
    public int compare (Object o1, Object o2) {
        return compare ((Etudiant) o1, (Etudiant) o2) ;
    }
    private int compare (Etudiant e1, Etudiant e2) {
        int me1 = e1.moyenne () ;
        int me2 = e2.moyenne () ;
        return me1 <  me2 ? 1 : me1 == me2 ? 0 : -1 ;
    }
}
```

Définir un ordre naturel

- Tout objet qui implémente l'interface java.lang.Comparable est dit posséder un **ordre naturel**.

```
public interface Comparable {
    public int compareTo(Object o) ;
}
```

- La méthode compareTo() renvoie un entier :
  - négatif, si this est strictement inférieur à o
  - nul, si this est égal à o
  - positif, si this est strictement supérieur à o

Définir un ordre naturel

- Cette solution est moins souple que la précédente car :
  - Un objet possède au plus 1 ordre naturel,
  - Pour qu'un objet ait un ordre naturel, il faut que sa classe implémente Comparable (autrement dit, il faut le prévoir à l'avance).

Définir un ordre naturel

- Par exemple on peut donner un ordre naturel aux Etudiant grâce à une extension de Etudiant :

```
class EtudiantParNom extends Etudiant implements
Comparable {
  public EtudiantParNom (String nom) { super (nom) ; }
  public int compareTo (Object o) {
    return this.nom ().compareTo (((EtudiantParNom) o).nom ()) ;
  }
}
```

Utilisation des conteneurs ordonnés

- Si vous utilisez un conteneur ordonné, comme TreeSet ou TreeMap, vous devrez choisir entre l'ordre naturel soit l'ordre explicite (avec un Comparator).
- Les conteneurs ordonnés qui fonctionnent avec l'ordre naturel ne peuvent contenir que des objets implémentant l'interface Comparable
- Les conteneurs ordonnés qui fonctionnent avec l'ordre explicite devront être créés avec un constructeur prenant un Comparator en argument.

Consistance avec equals()

- La relation d'ordre doit toujours être consistante avec equals() :
  - Dans le cas d'un ordre naturel :
    - e1.compareTo((Object)e2) == 0 renvoie la même valeur que e1.equals((Object)e2)
  - Dans le cas d'un ordre explicite :
    - compare((Object)e1, (Object)e2)==0 renvoie la même valeur que e1.equals((Object)e2)

Consistance avec equals()

- Si cette règle n'est pas respectée, il peut y avoir des comportements incohérents.
- En effet, si on insère dans un SortedSet ordonné naturellement, deux clefs a et b telles que :
  - (!a.equals((Object)b) && a.compareTo((Object)b) == 0)
- Le deuxième opération add() retournera false et l'élément ne sera pas inséré.

Hachage et codes de hachage

```
import java.util.*;

class Groundhog {
  int ghNumber;
  Groundhog(int n) { ghNumber = n; }
}

class Prediction {
  boolean shadow = Math.random() > 0.5;
  public String toString() {
    if(shadow)
      return "Six more weeks of Winter!";
    else
      return "Early Spring!";
  }
}
```

Hachage et codes de hachage

```
public class SpringDetector {
  public static void main(String[] args) {
    HashMap hm = new HashMap();
    for(int i = 0; i < 10; i++)
      hm.put(new Groundhog(i), new Prediction());
    System.out.println("hm = " + hm + "\n");
    System.out.println(
      "Looking up prediction for Groundhog #3:");
    Groundhog gh = new Groundhog(3);
    if(hm.containsKey(gh))
      System.out.println((Prediction)hm.get(gh));
    else
      System.out.println("Key not found: " + gh);
  }
}
```

Hachage et codes de hachage

- Cet exemple ne marche pas.
- Le problème vient du fait que **Groundhog** hérite de la méthode **hashCode()** de **Object** qui est utilisée pour générer le code de hachage pour chaque objet, et par défaut, celle-ci renvoie juste l'adresse de cet objet.
- La première instance de **Groundhog(3)** ne renvoie donc pas le même code de hachage que celui de la seconde instance de **Groundhog(3)** que nous avons tenté d'utiliser comme clef d'extraction.

Hachage et codes de hachage

- On pourrait penser qu'il suffit de redéfinir **hashCode()**. Mais ceci ne fonctionnera toujours pas tant qu'on n'aura pas aussi redéfini la méthode **equals()**.
- Cette méthode est utilisée par le **HashMap** lorsqu'il essaie de déterminer si la clef est égale à l'une des autres clefs de la table. Et la méthode par défaut **Object.equals()** compare simplement les adresses des objets, ce qui fait qu'un objet **Groundhog(3)** est différent d'un autre **Groundhog(3)**.
- Pour utiliser un nouveau type comme clef dans un **HashMap**, il faut donc redéfinir les deux méthodes **hashCode()** et **equals().**

Hachage et codes de hachage

```
import java.util.*;
class Groundhog2 {
  int ghNumber;
  Groundhog2(int n) { ghNumber = n; }
  public int hashCode() { return ghNumber; }
  public boolean equals(Object o) {
    return (o instanceof Groundhog2)
      && (ghNumber == ((Groundhog2)o).ghNumber);
  }
}
public class SpringDetector2 {
  public static void main(String[] args) {
    HashMap hm = new HashMap();
    for(int i = 0; i < 10; i++)
      hm.put(new Groundhog2(i),new Prediction());
    System.out.println("hm = " + hm + "\n");
    System.out.println(
      "Looking up prediction for groundhog #3:");
    Groundhog2 gh = new Groundhog2(3);
    if(hm.containsKey(gh))
      System.out.println((Prediction)hm.get(gh));
  }
}
```

Les interfaces et classes obsolètes

- Beaucoup de code a été écrit en utilisant les conteneurs Java 1.0 / 1.1, et aujourd'hui encore ces classes continuent d'être utilisées dans du nouveau code.
- Bien qu'il faille éviter d'utiliser les anciens conteneurs lorsqu'on produit du code nouveau, il faut toutefois être conscient qu'ils existent.
- Cependant, les anciens conteneurs étaient plutôt limités, il n'y a donc pas tellement à en dire sur eux.

L'interface java.util.Enumeration

- L'interface Enumeration définit deux méthodes permettant de balayer une séquence d'éléments :

```
public interface Enumeration{
  boolean hasMoreElements();
  Object nextElement();
}
```

- En fait l'interface Enumeration est obsolète, on doit désormais lui préférer l'interface Iterator.

La classe java.util.Vector

- Le **Vector** était la seule séquence auto-redimensionnable dans Java 1.0 / 1.1, ce qui favorisa son utilisation.
- Il s'agit d'une **ArrayList** .
- Elle n'est abordée ici que parce que de nombreux programmes font et en feront encore usage et qu'on ne peut donc pas l'ignorer.

La classe java.util.Vector

- Pour mieux intégrer la vielle classe Vector, les auteurs du JDK en ont fait, à partir de la version 1.2, une implémentation de List. Ceci provoque un certain nombre d'inconvénients :
  - Opérations redondantes, par exemple add() et insertElementAt() font la même chose, ainsi que get(int) et elementAt(int), car Java conserve la compatibilité avec les versions antérieures. Dans la documentation les méthodes obsolètes sont qualifiées de *deprecated*.
  - Un Vector sait fournir à la fois une Enumeration de ces éléments (vestige de la première version) et un Iterator.
  - public Enumeration elements() // JDK 1.0 Renvoie une énumération des éléments du Vector, qui seront obtenus dans l'ordre des indices croissants.

La classe java.util.Vector

- Voici comment on peut utiliser une Enumeration pour balayer une liste :

```
public int moyenneAvecEnumeration () {
  int sommeNotes = 0 ;
  int sommeCoeff = 0 ;
  java.util.Enumeration e;
  for (e = l.elements() ; e.hasMoreElements () ;) {
    NoteCoefficientee n = (NoteCoefficientee) e.nextElement () ;
    sommeNotes += n.note () * n.coefficient () ;
    sommeCoeff += n.coefficient () ;
  }
  return sommeNotes / sommeCoeff ;
}
```

La classe java.util.Stack

- Stack est sous-classe de Vector et hérite des méthodes de Vector.
- Les nouvelles méthodes définies par Stack sont les opérations classiques d'une pile :
  - **Stack()** le seul constructeur.
  - **boolean empty()** sans commentaire !
  - **Object peek()** renvoie le sommet sans dépiler.
  - **Object push(Object item)** empile et renvoie item.
  - **Object pop()** dépile et renvoie le sommet.
  - **int search(Object o)**

La classe : java.util.Hashtable

- Hashtable est la version obsolète des HashMap.
- Elle n'est abordée ici que parce que de nombreux programmes font et en feront encore usage et qu'on ne peut donc pas l'ignorer.
- Une particularité des Hashtable est, qu'à l'origine, elles fournissaient non par des Iterator, mais des Enumeration.
- Les Enumeration sont obsolètes et remplacées par les Iterator.
- Cependant elles fournissent le même genre de méthodes avec des noms différents (hasMoreElements() au lieu de hasNext() et nextElement() au lieu de next()).

La classe : java.util.Hashtable

- Une Enumeration des clefs ou des valeurs pouvait être prise directement auprès d'une Hashtable :
  - public Enumeration keys() ;
    - Pour obtenir une énumération des clefs.
  - public Enumeration elements() ;
    - Pour obtenir une énumération des valeurs.

La classe : java.util.Hashtable

```
class Promotion {
  private java.util.Map m = new java.util.Hashtable () ;


  public void ajouter (Etudiant e) {
    m.put (e.nom (), e) ;
  }


  public Etudiant etudiant (String nom) {
    return (Etudiant) m.get (nom) ;
  }


  public int moyenne () {// en iterant sur les etudiants
    int cumul = 0 ;
    for (Enumeration i = m.elements () ; i.hasMoreElements () ;) {
      cumul += ((Etudiant) i.nextElement ()).moyenne () ;
    }
    return cumul / m.size () ;
  }
}
```

La classe : java.util.BitSet

- Un **BitSet** est utile si on veut stocker efficacement un grand nombre d'informations on-off.
- Cette classe implémente un vecteur de bits dont la taille augmente en fonction des besoins.
- Chaque élément du BitSet a une valeur booléenne (boolean).
- Les bits d'un BitSet sont indexés pas des entiers positifs.
- Par défaut, tous les bits sont initialisés à false.

La classe : java.util.BitSet

- La classe BitSet propose un ensemble de méthodes :
  - void and(BitSet set)
    - Réalise une opération ET logique avec l'argument
  - void or(BitSet set)
    - Réalise une opération OR logique avec l'argument
  - void xor(BitSet set)
    - Réalise une opération XOR logique avec l'argument
  - cardinality(), clear(), intersects(), isEmpty(), etc.

Choisir une implémentation

- Nous avons vu qu'il existe trois types de conteneurs : les **Map**s, les **List**s et les **Set**s, avec seulement deux ou trois implémentations pour chacune de ces interfaces.
- Si on décide d'utiliser les fonctionnalités offertes par une **interface** particulière, comment choisir l'implémentation qui conviendra le mieux ?

Choisir une implémentation

- Chaque implémentation dispose de ses propres fonctionnalités, forces et faiblesses.
- Par exemple, on peut voir dans le diagramme que les classes **Hashtable**, **Vector** et **Stack** sont des reliquats des versions précédentes de Java, ce vieux code testé et retesté n'est donc pas près d'être pris en défaut. D'un autre côté, il vaut mieux utiliser du code Java 2.

Choisir une implémentation

- La distinction entre les autres conteneurs se ramène la plupart du temps à leur « support sous-jacent » ; c'est à dire la structure de données qui implémente physiquement l'**interface** désirée.
- Par exemple, les **ArrayList**s et les **LinkedList**s implémentent toutes les deux l'interface **List**, donc un programme produira les mêmes résultats quelle que soit celle qui est choisie. Mais :
  - Une **LinkedList** est implémentée sous la forme d'une liste doublement chaînée. C'est donc le choix approprié si on souhaite effectuer de nombreuses insertions et suppressions au milieu de la liste.
  - Dans les autres cas, une **ArrayList** est typiquement plus rapide car implémentée sous forme de tableau permettant un accès direct rapide.

Choisir une implémentation

- Un **Set** peut être implémenté soit sous la forme d'un **TreeSet** ou d'un **HashSet**.
- Un **TreeSet** est supporté par un **TreeMap** et est conçu pour produire un **Set** constamment trié.
- Cependant, si le **Set** est destiné à stocker de grandes quantités d'objets, les performances en insertion du **TreeSet** vont se dégrader.
- Quand vous écrirez un programme nécessitant un **Set**, choisissez un **HashSet** par défaut, et changez pour un **TreeSet** s'il est plus important de disposer d'un **Set** constamment trié.

Les utilitaires

- Il existe des classes java.util.Arrays et java.util.Collections, qui fournissent un essemble de méthodes statiques bien pratiques pour manipuler, respectivement, les Array et les Collection.
- Ces classes fournissent entre autres des méthodes de tri.

Trier des tableaux

- La classe java.util.Arrays fournit une méthodes statiques sort() pour chaque type tableaux de type primitif, et deux méthodes statiques sort() pour les tableaux d'objets qui se distingue par le fait que l'une utilise l'ordre naturel des objets et l'autre utilise un Comparator :
  - public static void sort(Object[] a) qui ordonne le tableau sur l'ordre naturel des objets de a. Une exception est bien sûr déclenchée si les objets en question n'ont pas d'ordre naturel, ou bien s'ils ne sont pas comparables.
  - public static void sort(Object[] a, Comparator c) qui ordonne le tableau sur l'ordre défini par le Comparator c. Si les objets du tableau disposent d'un ordre naturel, celui-ci n'est bien sûr pas pris en compte. Il n'est pas nécessaire que les objets du tableau dispose d'un ordre naturel.

Trier des List

- La classe java.util.Collections fournit des méthodes statiques (Collections.sort()) permettant d'ordonner une java.util.List (par exemple LinkedList et ArrayList).
  - public static void sort(List list) utilise l'ordre naturel
  - public static void sort(List list, Comparator c) utilise l'ordre défini par c.

java.util.Arrays

- La classe Arrays fournie également :
  - La méthode List asList(Object[] a)
  - La méthode boolean equals(Object[] a, Object[] a2)
    - + les surcharges pour les types primitifs
  - Les méthodes int binarySearch(Object[] a, Object o) et int binarySearch(Object[] a, Object o, Comparator c)
    - + les surcharges pour les types primitifs

java.util.Collections

- La classes Collections fournie également :
  - Les méthodes int binarySearch(List list, Object o) et int binarySearch(List list, Object o, Comparator c)
  - La méthode void copy(List dest, List src)
  - La méthode void reverse(List list)
  - Les méthodes Object max(Collection coll) et Object min(Collection coll)
  - Les méthodes Object max(Collection coll, Comparator c) et Object min(Collection coll, Comparator c)

Résumé

- Un tableau associe des indices numériques à des objets. Il stocke des objets d'un type connu afin de ne pas avoir à transtyper le résultat quand on récupère un objet. Il peut être multidimensions, et peut stocker des scalaires. Cependant, sa taille ne peut être modifiée une fois créé.
- Une **Collection** stocke des éléments, alors qu'une **Map** stocke des paires d'éléments associés.
- Comme un tableau, une **List** associe aussi des indices numériques à des objets - on peut penser aux tableaux et aux **List**s comme à des conteneurs ordonnés. Une **List** se redimensionne automatiquement si on y ajoute des éléments. Mais une **List** ne peut stocker que des références sur des **Object**s, elle ne peut donc stocker des scalaires et il faut toujours transtyper le résultat quand on récupère une référence sur un **Object** du conteneur.

Résumé

- Utiliser une **ArrayList** si on doit effectuer un grand nombre d'accès aléatoires, et une **LinkedList** si on doit réaliser un grand nombre d'insertions et de suppressions au sein de la liste.
- Le comportement de files et piles est fourni via les **LinkedList**s.
- Une **Map** est une façon d'associer non pas des nombres, mais des *objets* à d'autres objets. La conception d'un **HashMap** est focalisée sur les temps d'accès, tandis qu'un **TreeMap** garde ses clefs ordonnées, et n'est donc pas aussi rapide qu'un **HashMap**.
- Un **Set** n'accepte qu'une instance de valeur de chaque objet. Les **HashSet**s fournissent des temps d'accès optimaux, alors que les **TreeSet**s gardent leurs éléments ordonnés.
- Il n'y a aucune raison d'utiliser les anciennes classes **Vector**, **Hashtable** et **Stack** dans du nouveau code.