Chapitre 12 : Collections et Maps

Interfaces

Implantations

Tables de hachage

Généricité

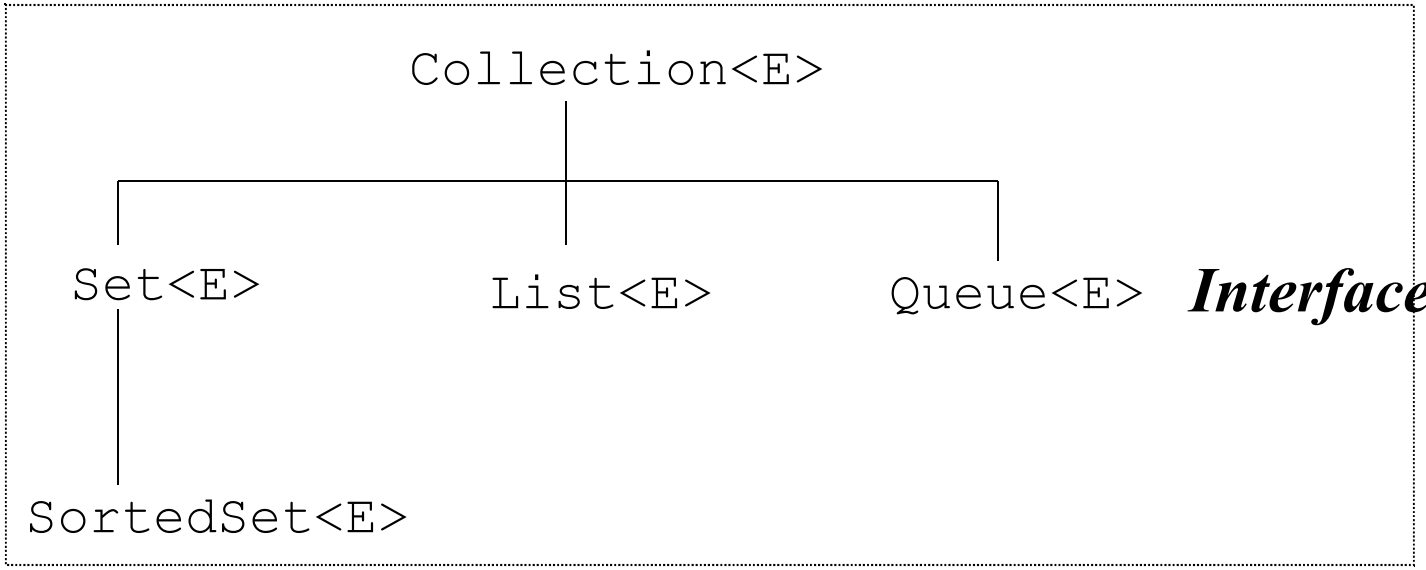
Les Collections : interfaces

- Définition
- Hiérarchie des interfaces de collections
- Interface Collection
- Les itérateurs

Collection : Définition

- Une collection est une structure de données qui rassemble des objets en une seule unité (objet)
- Il existe deux catégories de collections :
  - les `listes` (**`List`**) sont une **séquence ordonnée** d'objets. On peut trouver **plusieurs exemplaires** du même objet dans une liste. Chaque élément peut être accédé par un **index**.
  - les `ensembles` (**`Set`**) rassemblent des éléments **sans ordre** particulier. On ne peut pas y trouver plus d'un exemplaire d'un même objet. Les éléments ne sont pas accédés par un index.
- L'interface `Queue` représente les collections accessibles en mode FIFO

Collections : interfaces



Propriétés générales des collections

- Toute collection est de type `Collection<E>`
- Toute collection possède :
  - une interface (son Type Abstrait de Données)
  - une implantation (classe qui implante le TAD)
  - des algorithmes utiles sur ces structures de données(tris, recherche, etc…)
- Propriétés
  - toutes les collections possèdent la même interface
  - certaines collections possèdent en plus des opérations propres
  - Seule l'implantation des opérations change, elle dépend du type de la structure de données

Interface `Collection<E>` (1/2)

```
public interface Collection<E> {
 // Opérations de base
 int size();
 boolean isEmpty();
 boolean contains( Object element );
 Iterator<E> iterator();

 // autres Operations
 boolean containsAll( Collection<?> c );

 // Array Operations
 Object[] toArray();
 <T>T[] toArray( T[]a );
}
```

Interface `Collection<E>` (2/2)

Méthodes optionnelles : certaines implantations peuvent ne pas définir certaines méthodes.
Dans ce cas, si une telle méthode est invoquée, l'exception `UnsupportedOperationException` est levée.

```
boolean add(E o)                    // Optional
boolean remove(Object element)      // Optional

// autres Opérations
boolean addAll(Collection<? extends E> c) // Optional
boolean removeAll(Collection<?> c)        // Optional
boolean retainAll(Collection<?> c)        // Optional
void clear()                              // Optional
```

Collections : les itérateurs

- `Iterator<E>` est une interface qui propose les méthodes permettant de parcourir une collection
- Elle facilite la programmation de boucles
- Au cours d'une itération, certains éléments peuvent être supprimés de la collection

`boolean hasNext()` *Retourne true si l'itération a encore des éléments.*

`E next()` *Retourne l'élément suivant dans l'itération.*

`void remove()` *supprime le dernier élément retourné par l'itérateur (opération optionnelle).*

Exemple d'itérations

Itération non générique

```
void filtre( Collection c ){
        Iterator i = c.iterator();
        for ( ;i.hasNext(); ){
             if( !cond( i.next() )) i.remove();
       }
}
```

Itération générique

```
void filtre( Collection<String> c ){
        Iterator<String> i = c.iterator();
        for ( ;i.hasNext(); ){
             if( !cond( i.next() )) i.remove();
       }
```

Itération : nouvelle boucle for

Java 1.5 propose une nouvelle boucle **for** pour itérer sur les collections

```
void filtre( Collection c ){
      for ( Object i : c )
            if( !cond( i )) c.remove(i);
}
```

Collections : les listes

- Interface `List<E>`
- Propriétés des listes
- Polymorphisme
- Itérateurs

Interface `List<E>` (1/2)

- Une liste est une collection ordonnée (séquence).

```
public interface List<E> extends Collection<E>
```

- Elle peut contenir des éléments dupliqués
- En plus des opérations héritées de l'interface `Collection`, elle inclut des opérations propres :

```
E get( int index );
int indexOf( Object o );
int lastIndexOf( Object o );
ListIterator<E> listIterator();
ListIterator<E> listIterator( int index );
List<E> subList( int from, int to );
```

Interface `List<E>` (2/2)

L'interface `List<E>` inclut les opérations optionnelles :

```
E set( int index, E element );
void add( int index, E element );
E remove( int index );
boolean addAll(int index, Collection<? extends E> c);
```

Propriétés générales des listes

- On trouve une méthode pour accéder un élément par son index

```
list<String> l= new List<String>();
l.add(5,new String( "Java" ));
String s = l.get(5);
```

- Pour modifier une partie de liste

```
l.subList(3,8).clear();
// suppression des éléments de l'indice 3 à l'indice 7
```

- Un itérateur spécifique : `ListIterator<E>()` avec des méthodes supplémentaires

```
previous(), hasPrevious(), ...
```

Polymorphisme

```
static void swap( List<Integer> a,int i,int j ){
      Integer tmp = a.get( i );
      a.set( i, a.get(j) );
      a.set( j, tmp );
}
```

Cette méthode réalise une permutation de deux éléments d'une liste quelle que soit son type d'implantation

L'itérateur de `List<E>`

•En plus de l'`Iterator<E>` de l'interface `Collection<E>`, `List<E>` inclut un itérateur spécifique `ListIterator<E>`

•`ListIterator<E>` permet de parcourir une liste dans les deux directions, de modifier la liste pendant l'itération et d'obtenir l'index courant de l'itérateur.

```
boolean hasNext();

E next();

boolean hasPrevious();

E previous();

int nextIndex();

int previousIndex();

void remove(); // Optional

void set(E o); // Optional

void add(E o); // Optional
```

Exemple

Parcours d'une liste `l` de droite à gauche :

```
ListIterator i = l.listIterator( l.size());
while( i.hasPrevious() )
{
      Foo f = (Foo) i.previous(); ...
}
```

Collections : les ensembles

- Interface `Set<E>`
- Interface `SortedSet<E>`

Collections : les ensembles

L'interface `Set<E>`

- Un ensemble est une collection qui ne peut pas contenir d'élément en double
- Ses méthodes sont exactement celles de l'interface `Collection`
- La sémantique de certaines méthodes est différente :
  - La méthode `add` doit vérifier que l'élément ajouté n'est pas déjà présent dans l'ensemble

Ensembles triés

L'interface `SortedSet<E>`

`E` **`first`**`()`
retourne le plus petit élément

`SortedSet<E>` **`headSet`**`(E toElement)`
retourne les éléments strictement inférieurs à `toElement`.

`E` **`last`**`()`
retourne le plus grand élément

`SortedSet<E>` **`subSet`**`(E fromElement, E toElement)`
retourne les éléments compris entre `fromElement`, inclus, et `toElement`, exclus.

`SortedSet<E>` **`tailSet`**`(E fromElement)`
retourne les éléments plus grands ou égal à `fromElement`.

Collection : les files

- Interface `Queue<E>`

L'interface `Queue<E>`

- Elle dérive de `Collection<E>`
- Elle fournit de opérations supplémentaires :

`E element`

Retourne, sans supprimer, la tête de la file

`boolean offer( E o )`

Insère l'élément `o` dans la file, si possible

`E peek()`

Retourne, sans supprimer, la tête de la file, retourne `null` si la file est vide

`E poll()`

Retourne et supprime la tête de la file, ou `null` si la file est vide

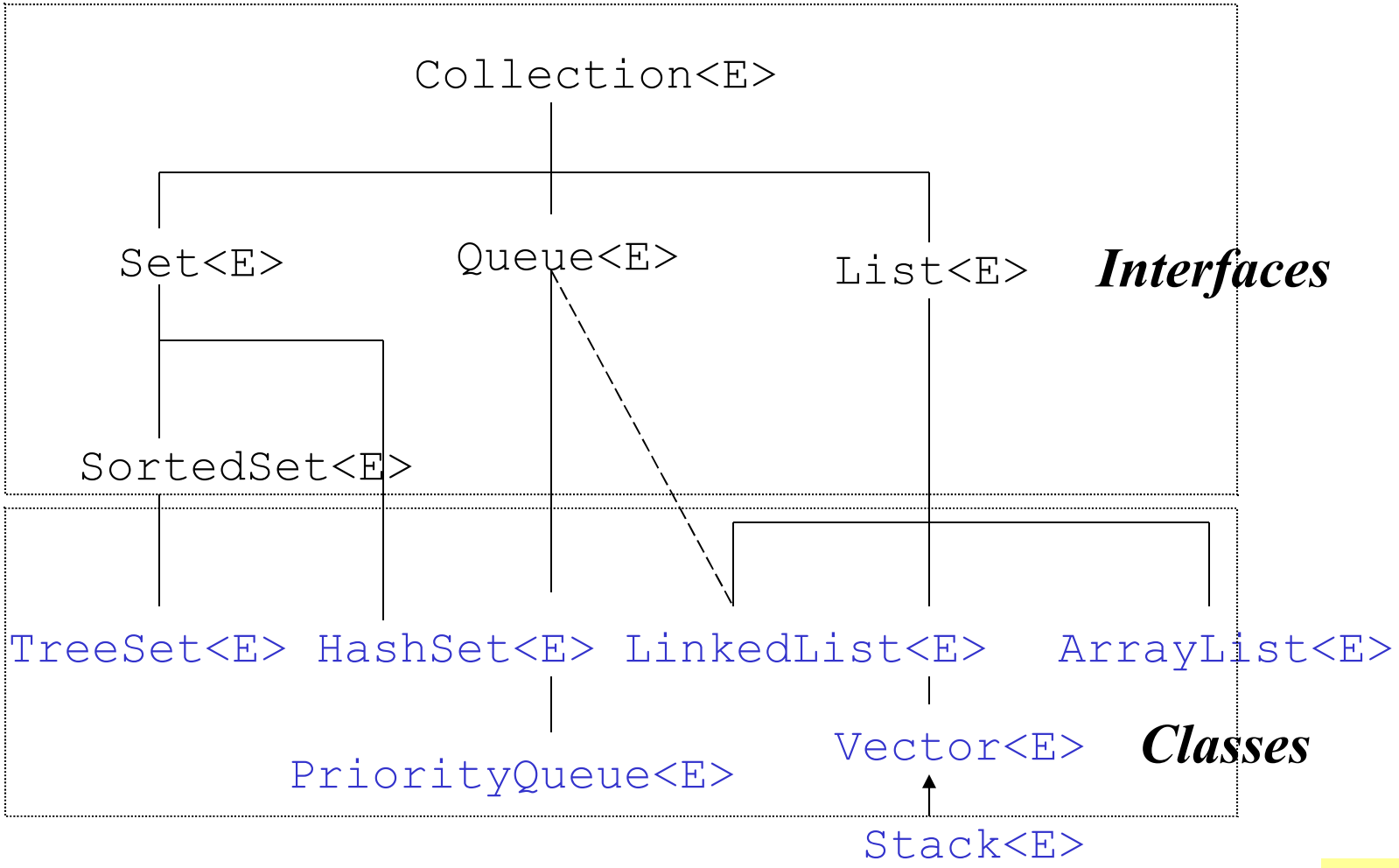
`E remove()`

Retourne et supprime la tête de la file

Collections : implantations

- les listes : `ArrayList,Vector,LinkedList`
- les ensembles : `HashSet,TreeSet`
- les files : `PriorityQueue,LinkedList`
- les piles : `Stack`

Implantation des collections : les classes



Classes et constructeurs

- Elles ont toutes 2 (au moins) constructeurs :

```
// Constructeur par défaut
HashSet<String> h = new HashSet<String>();
// création d'une collection par copie
// d'une autre collection
LinkedList<String> l = new LinkedList<String>( h );
```

Ou bien

```
ArrayList<String> a = new ArrayList<String>( h );
```

Ou encore

```
TreeSet<String> t = new TreeSet<String>( a );
```

Implantation des listes

- Représentation séquentielle
- Représentation chaînée

Listes : représentation séquentielle

- Les éléments de la séquence sont contigus en mémoire. La classe `ArrayList<E>` implante un tableau capable de s'accroître dynamiquement

- **L'insertion** ou la **suppression** d'un élément impose des manipulations lourdes sur le tableau pour maintenir la contiguïté des éléments (`O(n)` inefficace).

- Intérêt : Pour la **sélection** ou le **remplacement** d'un i-ème élément, on utilise directement l'indexation du tableau (`O(1)`).

La classe `Vector<E>`

- appartient au package `java.util`
- stocke des références à des objets

```
Vector<E> v = new Vector<E>();
```

- quelques méthodes

```
addElement( E o )
// ajoute o en queue et augmente la taille si nécessaire
removeElement( Object o )
boolean isEmpty()
boolean contains( Object )
```

`E firstElement()` et `E lastElement()`

```
int size() // nombre d'éléments courant
int capacity() // nombre d'éléments potentiels
```

```
import java.util.*;
public class VecteurClient{
  public static void main(String[] args){
    Integer i1 = new Integer(1),i2 = new Integer(2),
            i3 = new Integer(3);
    Vector<Integer> v = new Vector<Integer>();
    for (int i=0;i<5;i++)
      v.addElement( new Integer(i) );
    v.addElement(i1);v.addElement(i2);v.addElement(i3);
    if ( v.contains(i2) )
    System.out.println( "v contient i2" );
    if ( v.contains( new Integer(4) ) )
       System.out.println( "v contient 4" );
    else
       System.out.println( "v ne contient pas 4" );
    for( Integer i : v )
        System.out.print( i+"  " );
  }}
```

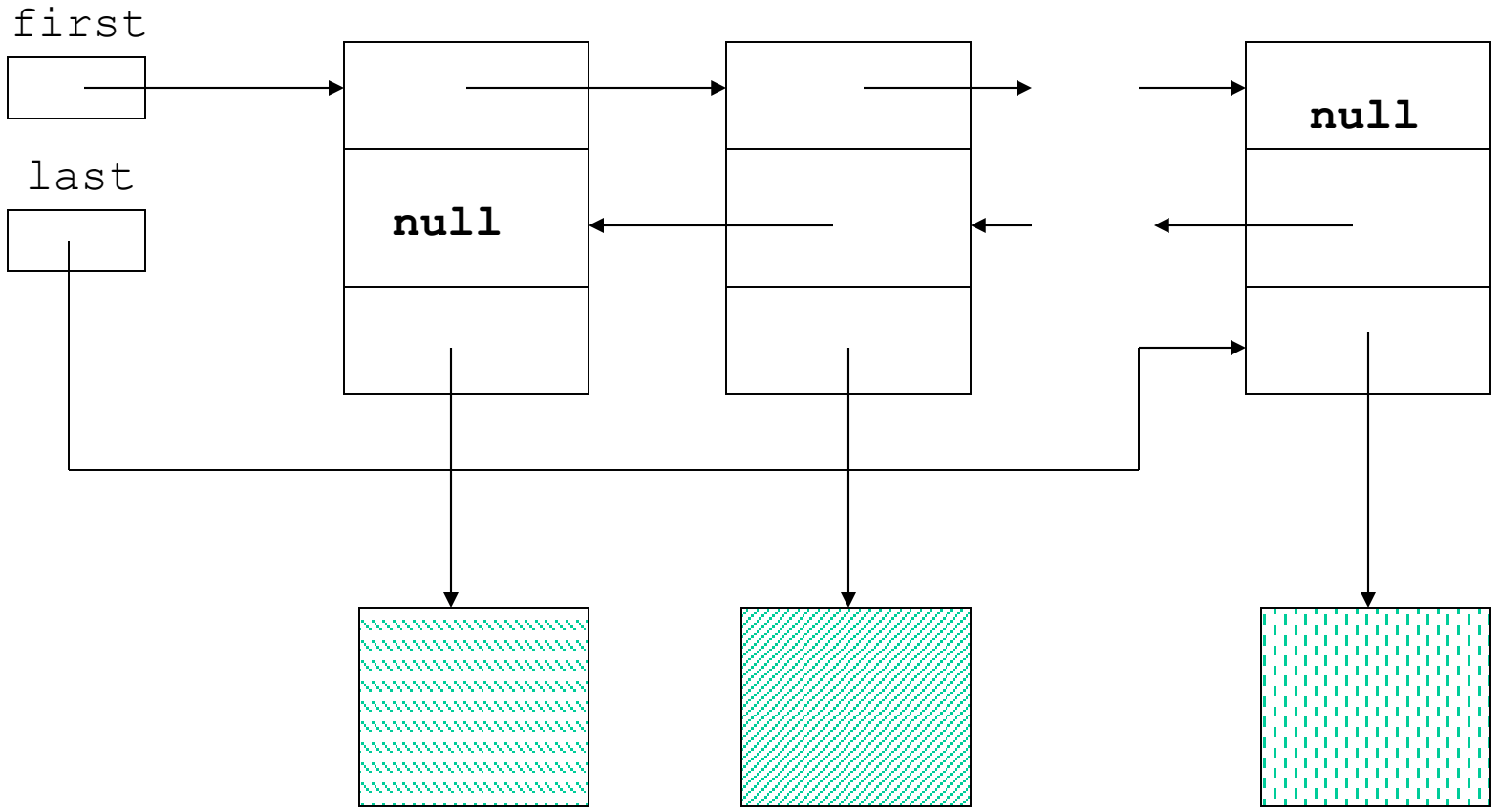
Résultats

v contient i2

v contient 4

0 1 2 3 4 1 2 3

Listes : représentation chaînée



Représentation chaînée

- En java, les listes chaînées sont implantées par la classe `LinkedList<E>`
- Inconvénient :
  - Accéder au i-ème élément suppose le parcours à partir de la tête de liste.
- Evaluation du coût :
  - Si on a autant de chances d'accéder à chacun des éléments, alors le nombre de noeuds visités est :

`(1+2+...+i)/n = (n+1)*n/2=n/2+1/2 => O(n)`

- En séquentiel, le coût est : `O(1)`

Exercice (1/4)

La classe `Color` (package `java.awt`) possède les constantes :

```
blue,yellow,cyan,orange,pink,black,green,red, ...
```

Ecrire un programme qui forme 2 listes chaînées contenant : l'une les 4 premières couleurs, l'autre les 4 suivantes.

Ce programme concatène la seconde liste à la première et l'affiche.

2 méthodes supplémentaires de la classe `LinkedList<E>` sont nécessaires :

```
boolean add(Object o)
boolean addAll(Collection c)
```

Exercice (2/4)

```
import java.util.*; import java.awt.*;
public class Exercice{

  private static final Color[]
   C1={Color.blue,Color.yellow,Color.cyan,Color.orange}
   ;
  private static final Color[]
   C2={Color.pink,Color.black,Color.green,Color.red};

  public Exercice(){
    LinkedList<Color> liste1 = new LinkedList<Color>();
    LinkedList<Color> liste2 = new LinkedList<Color>();
    for ( int i=0;i<C1.length;i++ ){
   liste1.add(C1[i]); liste2.add(C2[i]);
     }
        liste1.addAll(liste2);
        this.print(liste1);
    }
```

Exercice (3/4)

```
public void print( LinkedList<Color> liste ){
    for( Color obj : liste )
          System.out.println( obj+" ");
}

public static void main(String[] args){
    new Exercice(); }
}
```

Exercice : résultats (4/4)

```
java.awt.Color[r=0,g=0,b=255]
java.awt.Color[r=255,g=255,b=0]
java.awt.Color[r=0,g=255,b=255]
java.awt.Color[r=255,g=200,b=0]
java.awt.Color[r=255,g=175,b=175]
java.awt.Color[r=0,g=0,b=0]
java.awt.Color[r=0,g=255,b=0]
java.awt.Color[r=255,g=0,b=0]
```

Complexité des opérations

opérations	représentation séquentielle	représentation chaînée
calcul de la longueur	O(1)	O(n)
insertion d'un nouveau premier élément	O(n)	O(1)
suppression du dernier élément	O(1)	O(1)
remplacement	O(1)	O(n)
suppression du i-ème élément	O(1)	O(n)

Coût en espace

- Pour les listes chaînées, la représentation du pointeur est nécessaire. Sa taille devient significative lorsqu'elle est proche de la taille de l'élément.
- Exemple : 8 octets pour les pointeurs + 4 octets pour l'élément = 12 octets pour chaque noeud alors que la représentation séquentielle ne demande que 4 octets.
- Pour la représentation séquentielle, la place perdue pour la partie non utilisée peut être importante.
- Il s'agit de trouver un juste équilibre, soient :

`n` le nombre de noeuds , `x` la taille du pointeur , `y` la taille de l'élément et `t` la taille du tableau . D'où la formule : `(2*x+y)*n=y*t`

Pour : `x=4 et y=4, on a n=t/3`

- En d'autres termes, il vaut mieux utiliser un tableau si on en utilise plus que le tiers

Implantation des ensembles

- Ensembles triés
- Comparaison d'objets
- Ensembles non triés

Ensembles

- Comment choisir entre les 2 implantations `HashSet<E>` et `TreeSet<E>`
- `HashSet<E>` stocke ses éléments dans une table de hachage
- `TreeSet<E>` stocke ses éléments dans un arbre

Ensembles

- `HashSet` est plus efficace (opérations en temps constant) mais ne considère pas d'ordre sur les éléments

- Les opérations de `TreeSet` sont en temps logarithmique mais on peut itérer selon l'ordre ascendant des éléments

Exemple (1/2)

```
import java.util.*;
public class Ensembles{
  private static final String[] couleurs={"blue","yellow",
"cyan","orange","green","red","pink","black","green","red",
"cyan"};
  public static void printNonDupliquees(Collection<String> c){
       Set<String> ensemble= new HashSet<String>( c );
       Iterator<String> iter = ensemble.iterator();
       while( ;iter.hasNext(); )
              System.out.print( iter.next() + "  " );
        System.out.println();
  }
  public static void main(String[] args){
       List<String> liste =
              new ArrayList<String>(Arrays.asList(couleurs));
       printNonDupliquees( liste );  }
```

Exemple (2/2)

```
green  red  cyan  orange  pink  blue  yellow  black
```

Peux-t-on remplacer `HashSet` par `TreeSet` ?

```
black  blue  cyan  green  orange  pink  red  yellow
```

Comparaison d'objets (1/2)

- L'interface `Comparable` spécifie la méthode :

```
public int compareTo( Object obj )
```

- Les classes qui implante cette méthode se base sur un ordre naturel de leurs éléments :

```
Integer, Float, Double, …, String, Date
```

- Exemple : soit `x` et `y` deux instances de la classe `Date` ou `String` ou toute autre classe numérique

```
int i = x.compareTo( y );
   i<0 => x<y
   i=0 => x=y
   i>0 => x>y
```

Comparaison d'objets (2/2)

Il faut redéfinir la méthode `compareTo` pour tout autre Classe

Exemple :

```
class Rectangle implements Comparable{
   private double longueur;
   private double largeur;
   ...
   public int compareTo( Rectangle r){
      if (r.longueur*r.largeur>this.longueur*this.largeur)
         return -1;
      else if (r.longueur*r.largeur<this.longueur*this.largeur)
       return 1;
            else return 0;
   }
}
```

Ensembles triés

L'interface `SortedSet` implantée par la classe `TreeSet`

`E` **`first`**`()`
retourne le plus petit élément

`SortedSet<E>` **`headSet`**`(E toElement)`
retourne les éléments strictement inférieurs à `toElement`.

`E` **`last`**`()`
retourne le plus grand élément

`SortedSet<E>` **`subSet`**`(E fromElement, E toElement)`
retourne les éléments compris entre `fromElement`, inclus, et `toElement`, exclus.

`SortedSet<E>` **`tailSet`**`(E fromElement)`
retourne les éléments plus grands ou égal à `fromElement`.

```
import java.util.*;
public class Ensembles{
  private static final String[] couleurs={"blue","yellow","cyan",
"orange","green","red","pink","black","green","red","cyan"};

  public static  void printSet( Collection<String> c ){
   SortedSet<String> ensemble = new TreeSet<String>( c );
   Iterator<String> iter = ensemble.iterator();
   while( iter.hasNext() )
     System.out.print( iter.next() + "  " );
   System.out.println();
  }

  public static void main(String[] args){
   SortedSet<String> l=
             new TreeSet<String>(Arrays.asList(couleurs));
   printSet( l );
   System.out.println("avant \"orange\"");
   printSet( l.headSet("orange"));
   System.out.println("après \"orange\"");
   printSet( l.tailSet("orange"));
  }
}
```

Résultats affichés

```
black  blue  cyan  green  orange  pink  red  yellow
avant "orange"
black  blue  cyan  green
après "orange"
orange  pink  red  yellow
```

Implantation des piles

- La classe `Stack<E>`
- Exemples d'utilisation

Piles

- Une structure de pile diffère d'une structure de liste par ses opérations :
  - construire une pile vide
  - déterminer si une pile est vide
  - empiler un composant au sommet de la pile
  - dépiler un composant à partir du sommet de la pile
  - faire une copie du composant situé au sommet

la classe `Stack<E>`

- Sous-classe de la classe `Vector<E>`

`boolean empty()`
Teste si la pile est vide

`E peek()`
retourne l'objet situé au sommet de la pile sans l'enlever

`E pop()`
retourne et supprime l'objet situé au sommet de la pile `E`

`push(E item)`
place l'objet `item` au sommet de la pile

Piles : exemple 1 (1/2)

- Vérifier qu'une expression algébrique est correctement parenthésée

```
début
  construire une pile vide;
  faire
    analyser l'expression en :
      - empilant chaque parenthèse ouvrante
      - dépilant lorsque l'on trouve une parenthèse
        fermante;
  tant que la pile n'est pas vide ;
fin.
```

Piles : exemple 1 (2/2)

```
((4+x)*(y/2)-3) =>

empiler (;
empiler (;
dépiler ;
empiler (;
dépiler ;
dépiler;
pile vide
```

Question : aurait-on pu utiliser une file ? expliquer.

Piles : exemple 2 (1/2)

Vérifier qu'une expression algébrique est correctement parenthésée, en utilisant cette fois 3 types de parenthèses `(, [, {` pour faire apparaître clairement les sous-expressions

L'intérêt de cette modification est que l'on peut, cette fois, faire apparaître les erreurs de parenthésage : ([(x+y}*2]+{[z*(a+b)]-6})

Algorithme : en considérant que l'expression est bien formée

```
début
  Construire une pile vide;
faire
  Analyser l'expression en :
   .empilant chaque parenthèse ouvrante;
   .dépilant lorsque l'on trouve une parenthèse
  fermante de même type sinon erreur;
tant que la pile n'est pas vide ;
fin.
```

Piles : exemple 2 (2/2)

```
([(x+y)*2]+{[z*(a+b)]-6})=>
empiler (;
empiler [;
empiler (;
dépiler );
dépiler ];
empiler {;
empiler [;
empiler (;
dépiler );
dépiler ];
dépiler };
dépiler );
pile vide
```

Question : pourrait-on utiliser une file ? expliquer

Piles : exemple 3 (1/3)

Evaluation d'expressions algébriques

3 notations sont utilisables pour les expressions algébriques :

- infixe : `(x+y)*2`
- préfixe : `*+xy2`
- postfixe : `xy+2*`

Les parenthèses ne sont pas nécessaires dans les notations infixe et postfixe. La notation postfixée est la mieux adaptée pour une évaluation de l'expression

(la machine virtuelle Java utilise la notation postfixée pour évaluer les opérations arithmétiques : instruction `iadd`).

Piles : exemple 3 (2/3)

Evaluation de l'expression `xy+2*`

traitement du symbole x	empiler x
traitement du symbole y	empiler y
traitement du symbole +	dépiler le 1er opérande dépiler le second opérande sommer les 2 opérandes. empiler le résultat
traitement du symbole 2	empiler 2
traitement du symbole *	dépiler le 1er opérande dépiler le second opérande. sommer les 2 opérandes. empiler le résultat

Piles : exemple 3 (3/3)

- Lorsque tous les symboles de l'expression ont été traités, le résultat de l'évaluation figure au sommet de la pile.
- C'est le seul élément de la pile.
- **Question** : que se passe-t-il si l'expression est mal formée ?
- **Question** : l'évaluation est-elle possible avec une file ?
- **Question** : Quel est l'intérêt de la pile ?

Piles : implantation d'appels récursifs en java

- L'exécution d'instructions bytecode Java utilise une **pile d'exécution**.
- Chaque thread actif possède sa propre pile d'exécution.
- Chaque invocation de méthode provoque la construction d'un contexte d'exécution construit sur le modèle suivant :
  - variables locales à la méthode
  - paramètres effectifs
  - valeur de retour
  - adresse de retour (dans le bytecode)
  - pointeur sur l'élément précédent dans la pile de contexte

  Ce contexte est empilé sur la pile d'exécution.
- Ce mécanisme est bien adapté à l'exécution de méthodes récursives :
  - A chaque appel récursif, le contexte d'exécution est empilé
  - A chaque retour, le contexte est dépilé.

Application

Soit la fonction de calcul de la factorielle d'un nombre entier naturel :

```
// n : entier naturel, n>=0
int fact(int n)
{
  if (n==0)    return 1;
  else         return n*fact(n-1);
}
```

Considérons l'invocation : `fact(3)`

Question : que se passe-t-il si le programme boucle ?

CODE **ACTION** **PILE de CONTEXTES**

```
fact(3)
```

1:empiler contexte

11:remplacer appel fact(3)
par la valeur retournée => 6

12:dépiler contexte

10:mise à jour valeur retournée 6

```
n*fact(n-1)
3*fact(2)
```

2:empiler contexte

8:remplacer appel fact(3)
*par la valeur retournée => 3*2*

9:dépiler contexte

7:mise à jour valeur retournée 2

```
n*fact(n-1)
2*fact(1)
```

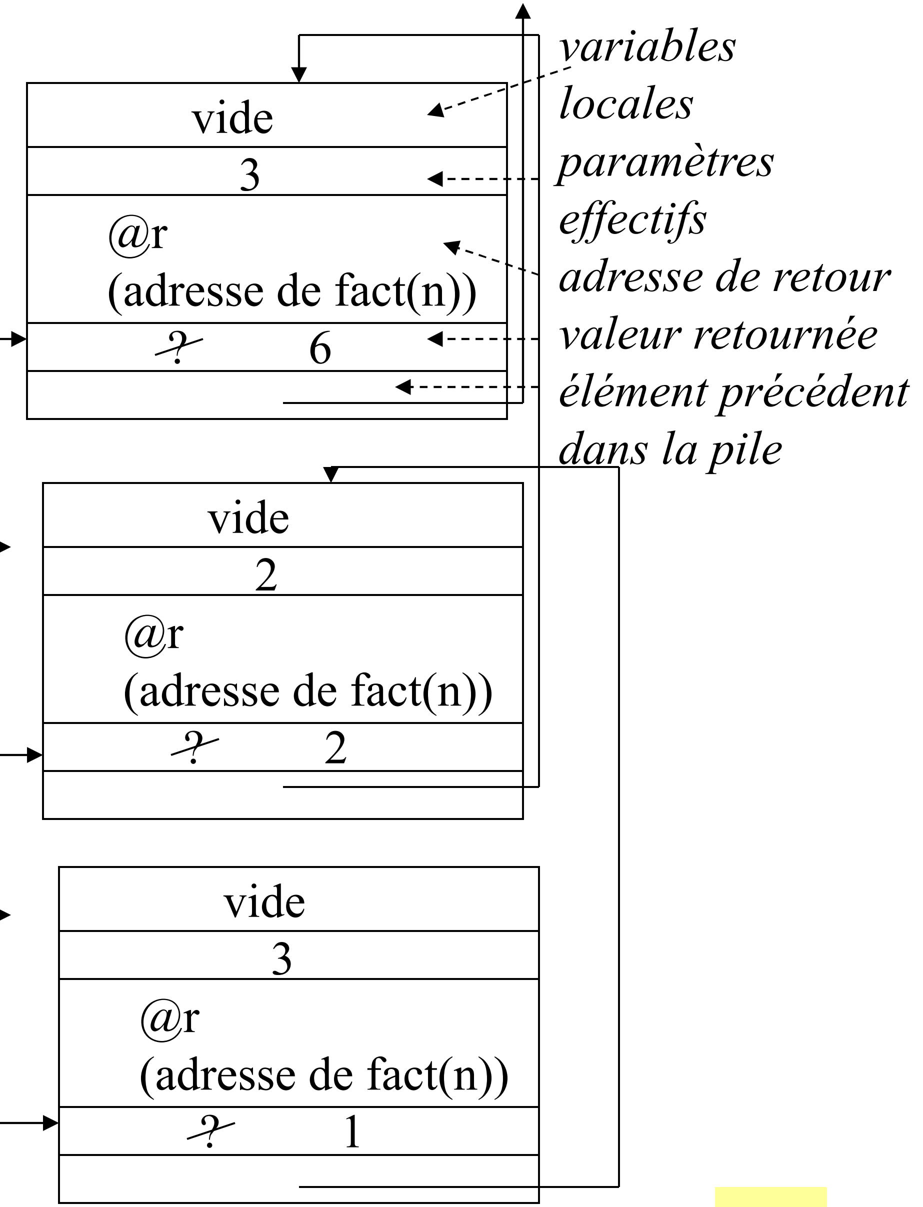
3:empiler contexte

5:remplacer appel fact(1)
*par la valeur retournée => 2*1*

6:dépiler contexte

```
n*fact(n-1)
1
```

4:mise à jour valeur retournée 1



Exemple

- Inversion d'une phrase d'une longueur quelconque, mot par mot
- L'utilisation d'une pile rend l'algorithme très simple

```
import java.io.*;import java.util.Stack;
import java.util.Scanner;import static java.lang.System.*;
public class Inversion {
 public static void main (String args []) throws IOException {
   new Inversion();
 }
 public Inversion () throws IOException {
  Scanner in = new Scanner ( System.in );
  Stack<String> pile = new Stack<String>();
  out.print("Taper une phrase, un mot par ligne ");
  out.println( "et terminer par stop" );
  out.println( "La phrase originale est: " );
  String mot = "";
  while ( !mot.equals( "stop" ) ) {
    mot = in.next();
    pile.push( mot );
  }
  out.println( "La phrase inversée est:" );
 while ( !pile.empty() )  out.print ( pile.pop() + " " );
 out.println();
}}
```

Implantation des files

- Avec `LinkedList<E>`
- Exemples d'utilisation

Les Files

- Opérations sur les files :
  - construire une file vide
  - déterminer si une file est vide
  - ajouter un composant en **queue** de file
  - retirer le composant de **tête** de la file (si elle n'est pas vide)

Implantation d'une file

- La classe `LinkedList<E>` fournit une implantation des méthodes propres aux files spécifiées dans l'interface `Queue<E>`.

- La méthode `offer` diffère de la méthode `add` (de `Collection`). Elle insère un élément si et seulement si c'est possible. Elle renvoie `false` sinon contrairement à la méthode `add` qui lève une exception dans ce cas.

- Comment utiliser une file d'entiers (par exemple)

```
Queue<Integer> file = new LinkedList<Integer>();
file.offer( 67 );
System.out.println(" tete="+file.peek() );
```

Files : application

Tampon de synchronisation

- Le buffer contient des lignes d'impression. Le processus CPU ajoute des lignes en fin de file
- Le processus de gestion de l'imprimante les consomme à partir de la tête de file
- Si la file est pleine, le CPU attend. Ainsi les 2 processus s'exécutent en concurrence. Lorsqu'un accès a lieu, l'autre processus est bloqué. A la fin d'un accès, les autres sont débloqués.
- Ce problème de synchronisation est appelé problème de producteur/consommateur.

- **Inconvénient** : le CPU attend si l'imprimante est lente.

Autres applications

- Serveur d'impression partagé (réseau)

Le fichier à imprimer est transféré par le réseau vers un serveur d'impression et est servi sur un mode FIFO (**`First In First Out`**)

Avantage : le CPU n'a pas à être bloqué en attente d'une fin d'impression

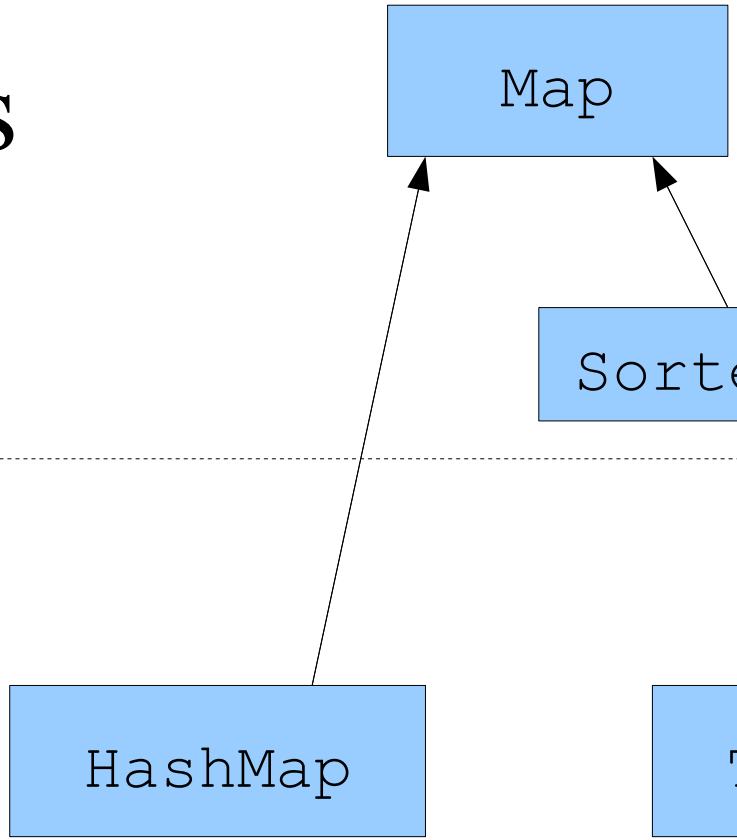
Gestion de la mémoire virtuelle dans un système d'exploitation (OS) à temps partagé

Les pages à échanger (swapper) sont rangées dans des files et transférées en mode FIFO de la mémoire centrale à la mémoire secondaire.

Les Maps

- Les techniques de hachage
- Hashtable, TreeMap
- HashMap

Maps



Clés triées

Interfaces

Classes

éléments enregistrés dans une table de hachage

éléments enregistrés dans un arbre

Maps

- Les Map associent des clés à des valeurs
- A chaque clé correspond une et une seule valeur
- Les Map diffèrent des Set qui ne contiennent que des valeurs

Techniques de hachage

- L'intérêt des techniques de hachage est de fournir des algorithmes de recherche en **temps constant**, c'est à dire indépendants du nombre d'éléments.

- Si l'on représente un ensemble d'éléments `(<clé,valeur>)` sous la forme d'un tableau, tout élément de l'ensemble sera enregistré à un indice du tableau que l'on pourrait calculer à partir de sa clé à condition de posséder une fonction :

```
h : Clé -> [1..n]
```

- L'idéal serait que pour toute clé `c` et `c'`,

```
h(c) ≠ h(c')
```

Exemple

- Les numéros de téléphones portables associés à leur position géographique. Ils constituent des clés uniques représentées sur 10 chiffres (5 tranches de 2 chiffres).

- Les 4 dernières tranches offrent un potentiel de 100 millions de **clés**. Le nombre d'abonnés réels tournant autour de 48 millions, le taux de clés réellement utilisées est de 48%.

- Si l'on représente cet ensemble d'éléments sous la forme d'un tableau, tout élément de l'ensemble sera enregistré à **un indice du tableau** que l'on pourrait calculer à partir de sa clé à condition de posséder une **fonction de hachage**

Collisions

- Le nombre de clés possibles étant largement supérieur au nombre d'adresses du tableau, on sera inévitablement confronté à des situations où deux clés `c` et `c'` seront telles que : `h(c) = h(c')` => **`collision`**

- Il existe une collection de clés qui correspondent à un même indice.
- On peut dire que l'ensemble des clés est **"haché".** La fonction `h` est appelée fonction de hachage.

Tables de hachage

- Pour imposer une adresse unique en correspondance à chaque clé il faut introduire une stratégie de résolution des collisions. Plusieurs stratégies existent.
- On désigne par "facteur de charge" (`fc`) le ratio entre le nombre d'entrées `N` effectivement occupées et la taille du tableau.

```
fc = N / S
```

- `fc` est toujours compris entre `0` et `1`. Plus `fc` se rapproche de `1` plus le tableau est proche de la saturation et plus les performances des méthodes de hachage se dégradent.

Exemple

- Soit un ensemble de clés

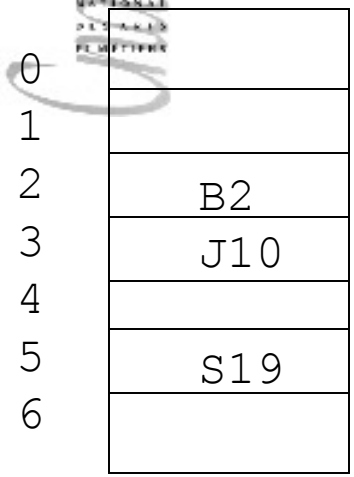
`Ec = {A1,B2,C3,...,R18,...,Z26}` où chaque lettre est indicée par sa position dans l'alphabet.A chacune de ces clés est associée une information utile dont nous ne tiendrons pas compte.

- Soit un tableau `T` de taille `7 (S=7)` implantant la table précédente.
- On choisit une fonction de hachage `h` telle que : quels que soit `x` dans `{A,B,C,...,R,...,Z}`, `n` dans `[1..26]`,

```
h(xn) = n % 7
```

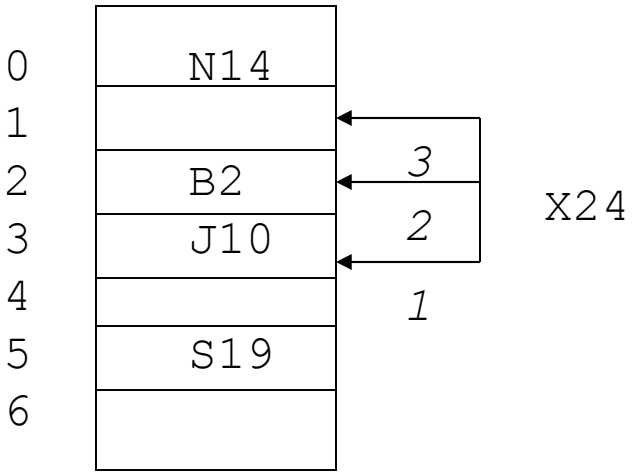
d'où,

```
h(J10) = 10 % 7 = 3
h(B2) = 2 % 7 = 2
h(S19) = 19 % 7 = 5
...
```

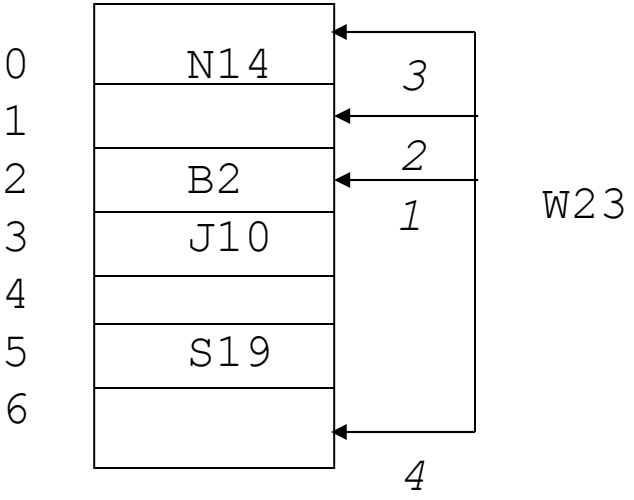


insertion de N14
h(N14)=14%7=0

h(X24)=24%7=3
collision



insertion de W23
h(W23)=23%7=2
collision



La classe `HashMap<K,V>`

- **`HashMap`** n'est pas synchronisée

  - **`HashMap`** Construit une HashMap vide avec une capacité initiale (16) et un facteur de charge par défaut (0.75).

  - **`HashMap`**`(int initialCapacity)` Construit une HashMap vide de capacité initiale (`initialCapacity`) et de facteur de charge par défaut (0.75).

  - **`HashMap`**`(int initialCapacity, float loadFactor)` Construit une HashMap vide de capacité initiale (`initialCapacity`) et de facteur de charge (`loadFactor`).

  - **`HashMap`**`(Map m)` Construit une HashMap à partir de la Map `m`

Quelques méthodes de `Hashtable<K,V>`

`Hashtable` est synchronisée

`boolean` **`contains`**`(Object value)`

teste s'il existe une clé qui correspond à la valeur

`boolean` **`containsKey`**`(Object key)`

teste si `key` est une clé pour la `Hashtable`

`Enumeration<V>` **`elements`**`()`

retourne une énumération des valeurs de la `Hashtable`

`V` **`get`**`(Object key)`

retourne la valeur associée à la clé

`Object` **`put`**`(Object key, Object value)`

ajoute un couple <clé, valeur> à la `Hashtable`

`V` **`remove`**`(Object key)`

supprime une entrée connaissant sa clé

`Enumeration<K>` **`keys`**`()`

retourne une énumération des clés de la `Hashtable`

Exercice 10 (1/4)

Ecrire un programme qui :

- insère dans une table de hachage les couples (clé, valeur) suivants :

(MEX,Mexico),(GLA,Glasgow),(NRT,Narita),(BRU,Brunei),
(JFK,Kennedy),(ORY,Orly)

- affiche les valeurs associées aux clés FRK et NRT si elles existent (un message d'erreur sinon)
- affiche la liste de toutes les valeurs de la table
- affiche la liste de toutes les clés
- remplace la valeur Orly par Orly sud et l'affiche
- supprime le couple (JFK,Kennedy)

Exercice 10 (2/4)

```
public class Exercice10{
   public static void main(String[] args){
        Hashtable<String,String> hash=new Hashtable<String,String>();
        hash.put("MEX","Mexico");
        hash.put("GLA","Glasgow");
        hash.put("NRT","Narita");
        hash.put("BRU","Brunei");
        hash.put("JFK","Kennedy");
        hash.put("ORY","Orly");
        if (hash.containsKey("FRK"))
                System.out.println(hash.get("FRK"));
        else System.out.println("FRK : clé inconnue");
        if (hash.containsKey("NRT"))
                System.out.println(hash.get("NRT"));
        else System.out.println("NRT : clé inconnue");
```

Exercice 10 (3/4)

```
Enumeration<String> e=hash.elements();
String s;
while(e.hasMoreElements()){
  s=e.nextElement();
  System.out.println(s);
 }
 e=hash.keys();
 while(e.hasMoreElements()){
   s=e.nextElement();
   System.out.println(s);
 }
 hash.put("ORY","Orly Sud");
 hash.remove("JFK");
 e=hash.elements();
 while(e.hasMoreElements()){
   s=e.nextElement();
   System.out.println(s);
  }}}
```

Exercice 10 (4/4)

```
FRK : clé inconnue
Narita
Orly
Glasgow
Brunei
Mexico
Narita
Kennedy
ORY
GLA
BRU
MEX
NRT
JFK
Orly Sud
Glasgow
Brunei
Mexico
Narita
```

Généricité

- Interfaces génériques
- Généricité et sous-typage
- Types joker
- Méthodes génériques

source : http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf

Paramétrage par les types

- En Java 1.4, utilisation du "cast"

```
List liste = new LinkedList();
liste.add( new Integer(0) );
Iterator iter = liste.iterator();
Integer x = (Integer)iter.next();
```

Problème :

- le compilateur ne garantit que le retour d'un élément de type `Object`
- pourtant le programmeur sait quel est le type de données ajouté à la liste

Version générique

- Idée : afficher l'intention du programmeur dès la déclaration de l'objet, ainsi le compilateur en aura connaissance et pourra effectuer la vérification de type

```
List<Integer> liste = new LinkedList<Integer>();
liste.add( new Integer(0) );
Iterator<Integer> iter = liste.iterator();
Integer x = iter.next();
```

`List` est une interface qui prend un type en paramètre

Interfaces génériques

- exemples :

```
public interface List<E>{
   void add( E x );
   Iterator<E> iterator();
}
public interface Iterator<E>{
   E next();
   boolean hasNext();
}
```

- `E` est un paramètre formel générique de type
- `List<Integer>` est un type paramétré
- `Integer` est un paramètre effectif de type

Généricité : C++ vs Java

- En C++, la généricité (templates) donne lieu à une expansion de code (code source pour chaque interface paramétrée)

```
public interface ListInteger{
    void add(Integer x);
    Iterator<Integer> iterator();
}
```

- En Java, le type générique est compilé une fois pour toute. Les paramètres génériques sont remplacés par les paramètres effectifs au moment de l'invocation d'une méthode. Ensuite, le corps de la méthode est évalué.

Généricité et sous-typage

- Le code suivant est-il légal ?

```
(1) List<String> ls = new ArrayList<String>();
(2) List<Object> lo = ls;
```

Question : une liste de "`String`"est-elle une liste d'"`Object`" ?

```
(3) lo.add( new Object() );
(4) String s = lo.get(0);
```

Commentaires

(1) type de `lo => List<Object>,` or on ajoute une instance d'`Object` => pas de problème

(1) `lo` possède un premier élément de type `Object` que l'on affecte à une `String.`

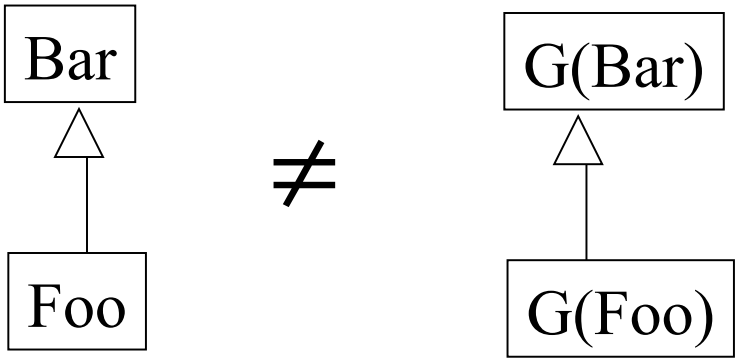
=>**Erreur!**

Généricité et sous-typage

- Le compilateur détectera cette erreur à la ligne

```
List<Object> lo = ls;
```

Conclusion



Type générique "joker"

- En Java 1.4

```
void printCollection(Collection c){
   Iterator i = c.iterator();
   for( int k=0;k<c.size();k++ )
       System.out.println(i.next());
}
```

- En utilisant la généricité et la nouvelle boucle for

```
void printCollection(Collection<Object> c){
   for( Object e : c )
       System.out.println( e );
}
```

Effort louable mais qui entraîne une erreur!

Type générique "joker"

Problème

`Collection<Object>`, à cause du problème de sous-typage entre type générique, n'est pas le super type de toutes les sortes de collections

⇒ On ne pourra donc pas appeler cette méthode avec n'importe quelle sorte de collection en paramètre

Solution

```
void printCollection(Collection<?> c){
   for( Object e : c )
        System.out.println( e );
}
```

Quel que soit le paramètre générique effectif, la collection contient des objets => donc la boucle for est valide

Utilisation du type Joker

```
Collection<?> c = new ArrayList<String>();
c.add( new Object() );
```

=> ERREUR de COMPILATION

Problème

La collection contient des objets de type inconnu, il est donc improbable que l'objet ajouté à `c` soit du même type ou d'un sous-type

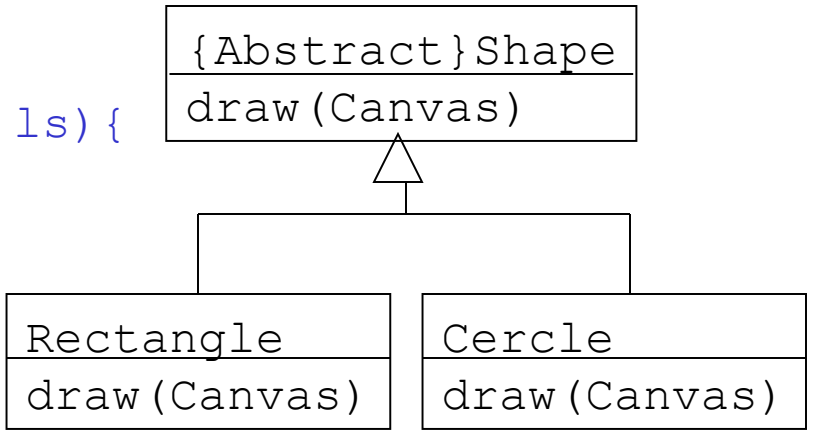
Mais

```
List<?> c = new ArrayList<String>();
Object o = c.get();
```

est valide car quel que soit le type des objets de la liste, ils sont aussi du type `Object`

Type joker contraint

```
public class Canvas{
 public void drawAll(List<Shape> ls){
    for( Shape s : ls )
       s.draw(this);
  }
}
```



Rappel : une `List<Shape>` n'est pas super classe de, par exemple, `List<Circle>`, `drawAll` ne considère donc que des listes constituées d'objets strictement du type `Shape`.

Solution

```
public void drawAll(List<? extends Shape> ls){...}
```

Type joker contraint : utilisation

```
public void addRectangle(List<? extends Shape> ls){
   ls.add(0,new Rectangle());
}
```

ATTENTION !

Le type du paramètre générique effectif associé à `?` est un sous type de `Shape`, on ne sait pas si `Rectangle` en est un sous-type

Méthodes génériques

```
void arrayToCollection(Object[] a,Collection<?> c){
   for( Object o : a )
        c.add(o);
}
```

ERREUR à la compilation

On ne peut pas placer des éléments de type `Object` dans une collection dont on ne connaît pas le type

Solution : méthodes génériques

```
<T>void arrayToCollection(T[] a,Collection<T> c){
   for( T o : a )
        c.add(o);
}
```

Inférence de type

- Le compilateur opère par inférence de type

Exemple 1 :

```
Object[] oa = new Object[20];
Collection<Object> oc = new ArrayList<Object>();
arrayToCollection(oa,oc)
```

=> `Object` est inféré à partir de `T`

Exemple 2 :

```
String[] oa = new String[20];
Collection<String> oc = new ArrayList<String>();
arrayToCollection(oa,oc)
```

=> `String` est inféré à partir de `T`