iUT ORSAY

Java et la généricité

Jean-Pierre Fournier

http://www.iut-orsay.fr/~fournier

Java est un langage objet

- d'origine, les principaux mécanismes
  - héritage,
  - encapsulation,
  - exceptions…
- et de nombreuses classes…
- mais pas vraiment de généricité jusqu'à la version 1.4 incluse

La généricité, c'est

- la possibilité d'écrire un algorithme dans lequel le type des objets manipulés est un paramètre (type abstrait de données)
- pas besoin de le refaire ou de le copier quand le même traitement doit être effectué sur un autre type de données

Exemple : Un algorithme de recherche

- doit déterminer si une valeur donnée est présente dans un tableau

```
static boolean estPresente(int valeur,
   int[] valeurs){
   for (int i=0; i<valeurs.length; i++)
      if (valeur==valeurs[i])    return true;
   return false;
   }
```

Le main et ses résultats :

```
/**
 * @param args 10 5 8 10 12 16 24
 */
public static void main(String[] args) {
    if (args.length<2)      return;
    int []valeurs = new int[args.length];
    for (int i=0; i<args.length; i++)
            valeurs[i]=Integer.parseInt(args[i]);
    int [] autres = new int[args.length-1];
    for (int i=0; i<autres.length; i++) autres[i]=valeurs[i+1];
    System.out.println("Le résultat de la recherche est
"+estPresente(valeurs[0], autres));
    System.out.println("Le résultat de la recherche est
"+estPresente(0, autres));
}
```

- Le résultat de la recherche est true
- Le résultat de la recherche est false

Si nous devons faire la même chose sur d'autres données

```
static boolean estPresente(float
  valeur, float[] valeurs){
  for (int i=0; i<valeurs.length;
  i++)
    if (valeur==valeurs[i])return
    true;
  return false;
  }
```

Sans généricité, risque de multiplication d'algorithmes identiques sur des données différentes

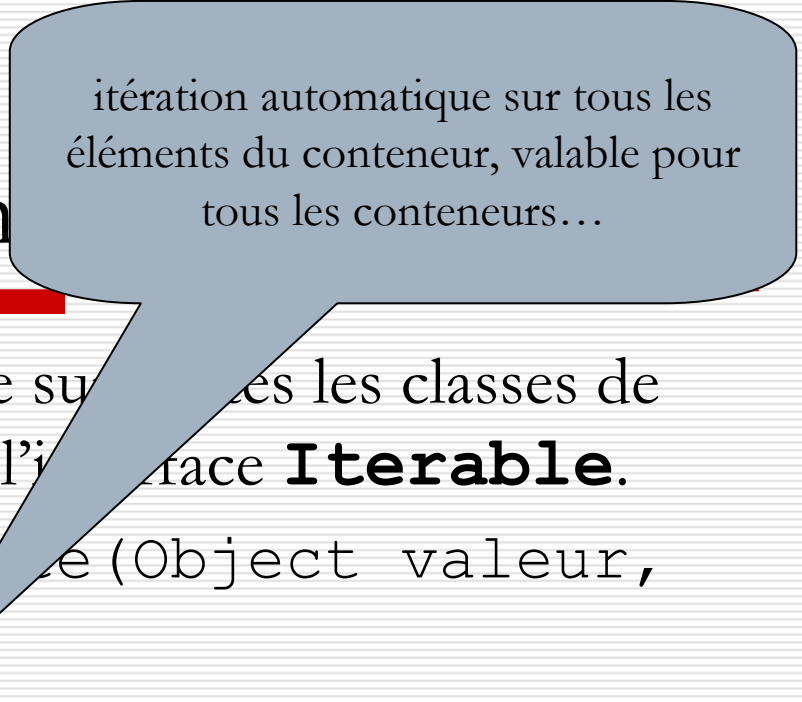
- l'approche Java (=> 1.4)
  - toutes les classes dérivent de la classe **`Object`**
  - si l'algorithme a pour données des instances de la classe **`Object`**, il sera générique

```
static boolean estPresente(Object valeur,
    Object[] valeurs){
    for (int i=0; i<valeurs.length; i++)
        if (valeur.equals(valeurs[i]))
            return true;
    return false;
    }
```

`equals` doit être surchargé

== compare les références d'objets
equals compare les valeurs

Une variante intéressan



- Cette possibilité est disponible su[illegible]es les classes de conteneurs qui implémentent l'i[illegible]face **`Iterable`**.

```
static boolean estPrese[illegible]e(Object valeur,
   Object[] valeurs){
   for (Object v : valeurs)
      if (valeur.equals(v))     return true;
   return false;
   }
```

- ne permet pas de travailler simultanément sur plusieurs collections

Mais le main devient plus complexe

```
/**
 * @param args 10 5 8 10 12 16 24
 */
public static void main(String[] args) {
    if (args.length<2)       return;
    float []valeurs = new float[args.length];
    for (int i=0; i<args.length; i++)
       valeurs[i]=Float.parseFloat(args[i]);
    Float [] autres = new Float[args.length-1];
    for (int i=0; i<autres.length; i++) autres[i]=new
    Float(valeurs[i+1]);
    System.out.println("Le résultat de la recherche est
    "+estPresente(new Float(valeurs[0]), autres));
    System.out.println("Le résultat de la recherche est
    "+estPresente(0, autres));
}
```

Et les conteneurs ?

- tous les conteneurs utiles
  - Vector, ArrayList, Stack, Set, Map…
  - sont définis comme contenant des Object
- ils peuvent contenir n'importe quoi
  - Vector v1=new Vector();
  - v1.add(new Integer(5));
  - v1.add(new Float(3.4));
  - v1.add(new Button("Coucou"));
- si on essaie : v1.get(0).intValue()
  - c'est réfusé, parce que v1.get(0) est un Object, la méthode intValue() étant inconnue sur les Object
- et si on essaie : ((Integer) v1.get(1)).intValue(), on produit une ClassCastException à l'exécution…
- par contre : if (v1.get(i) instanceof Integer) …permet de l'éviter.

Les autres langages « objet »

- C++, Ada…
- possèdent la généricité de manière plus ou moins native
- traitent cette généricité à la compilation, garantissant une exécution sans problème
- Java devait réagir, d'où la version 1.5

Java, version 5 (1.5)

- propose des conteneurs génériques
  - Vector<Integer> v = new Vector<Integer>();
  - v.get(i).intValue() est possible, puisque la méthode get renvoie maintenant un Integer
- les tableaux `[]` restent comme avant…
  - donc à proscrire des algorithmes génériques

```
static boolean estPresente(Integer valeur, Vector<Integer> valeurs){
    for (int i=0; i<valeurs.size(); i++)
        if (valeur.equals(valeurs.get(i)))      return true;
    return false;
    }
```

- ou aussi (foreach)

```
static boolean estPresente(Integer valeur, Vector<Integer> valeurs){
    for (Integer i : valeurs)
        if (valeur.equals(i))   return true;
    return false;
    }
```

Le main…

```
/**
 * @param args 10 5 8 10 12 16 24
 */
public static void main(String[] args) {
    if (args.length<2)       return;
    int []valeurs = new int[args.length];
    for (int i=0; i<args.length; i++)
       valeurs[i]=Integer.parseInt(args[i]);
    Vector<Integer> autres = new Vector<Integer>();
    for (int i=0; i<args.length-1; i++) autres.add(new
    Integer(valeurs[i+1]));
    System.out.println("Le résultat de la recherche est
    "+estPresente(new Integer(valeurs[0]), autres));
    System.out.println("Le résultat de la recherche est
    "+estPresente(0, autres));
       }
```

Généricité

- Nous avons maintenant des conteneurs spécialisés
- Nous aimerions que notre algorithme reste générique

```
static boolean estPresente(Object
   valeur, Vector<?> valeurs){
   for (int i=0; i<valeurs.size(); i++)
      if (valeur.equals(valeurs.get(i)))
         return true;
   return false;
   }
```

Généricité du second type

Le mot Vector est présent

- Dans les versions précédentes, le type des données manipulées peut changer, le type du conteneur reste figé

```
static boolean estPresente(Object
   valeur, Vector<?> valeurs){
   for (int i=0; i<valeurs.size(); i++)
      if (valeur.equals(valeurs.get(i)))
         return true;
   return false;
   }
```

on accède aux éléments par un indice

Pour généraliser

- beaucoup de conteneurs ne donnent pas accès à leurs éléments par des indices : listes chaînées, arbres, tables de correspondance, graphes…
- tous les conteneurs peuvent être parcourus avec des itérateurs
- les algorithmes génériques doivent donc employer des itérateurs au lieu des indices

```
static boolean estPresente(Object valeur,
   Vector<?> valeurs){
   for (Iterator<?> i=valeurs.iterator();
   i.hasNext();)
         if (valeur.equals(i.next()))
               return true;
   return false;
   }
```

Pour généraliser davantage

- pour « oublier » dans quel conteneur sont les données, il suffit de ne transmettre que l'itérateur

```
static boolean estPresente(Object valeur,
   Iterator<?> it){
   for (Iterator<?> i=it; i.hasNext();)
         if (valeur.equals(i.next())
               return true;
   return false;
   }
```

- ***ce sous-programme ne sait pas quelle structure de données il parcourt***

Et le main transmet l'itérateur

```
    /**
     * @param args 10 5 8 10 12 16 24
     */
public static void main(String[] args) {
    if (args.length<2)       return;
    int []valeurs = new int[args.length];
    for (int i=0; i<args.length; i++)
       valeurs[i]=Integer.parseInt(args[i]);
    Vector<Integer> autres = new Vector<Integer>();
    for (int i=0; i<autres.size(); i++) autres.add(new
    Integer(valeurs[i+1]));
    System.out.println("Le résultat de la recherche est
    "+estPresente(new Integer(valeurs[0]),
    autres.iterator()));
    System.out.println("Le résultat de la recherche est
    "+estPresente(0, autres.iterator()));
}
```

A suivre : les collections

- Extrait de la documentation : « The Collections Framework »
  - **réduit le travail** du programmeur en lui fournissant des classes, des structures de données et des méthodes qu'il n'a plus besoin de développer lui-même
  - **améliore les performances** en fournissant des implémentations de haut niveau des structures de données et algorithmes standards. Les implémentations étant interchangeables, les programmes peuvent être ajustés en changeant facilement d'implémentation.
  - **Permet l'interopérabilité entre des APIs distinctes** en établissant un langage commun pour parcourir les collections.
  - **Réduit les efforts nécessaires pour apprendre une API** puisqu'on n'a plus besoin d'apprendre plusieurs modes d'emploi.
  - **etc.**

Le choix des collections est important

- Celui qui élabore un algorithme commence par des choix déterminants pour la suite :
  - choix du mode de programmation (itératif, récursif)
  - choix de la structure de données
- Chaque mode de programmation a ses avantages et inconvénients, chaque structure de données aussi
- S'il fait un mauvais choix, il aboutira (peut-être) à une solution inélégante et probablement inefficace
  - exemple : utilisation d'un algorithme itératif sur une structure de données récursive (liste chaînée ou arbre)

Les principaux conteneurs Java

- Vector (java.util.Vector) : tableau dynamique
- ArrayList : implémentation dans un tableau d'une liste chaînée
- HashMap : implémentation en adressage dispersé (hash-coding) d'une table de correspondance (Map)
- LinkedList : implémentation d'une liste chaînée
- Stack : pile classique
- TreeMap : implémentation qui s'appuie sur un arbre bien équilibré d'une table de correspondance

Et quelques algorithmes puissants

- sort() pour trier une collection d'éléments,
- shuffle() pour obtenir une permutation aléatoire d'une collection d'éléments,
- reverse() pour retourner une liste,
- rotate() pour faire tourner une liste,
- min(), max(), frequency()
- disjoint() qui détermine si deux listes sont disjointes…
- copy(), etc…

Retour sur l'exemple précédent…

- un algorithme qui reçoit un objet, un conteneur, qui indique si cet objet est présent dans ce conteneur
- besoin simple => existe certainement…

```
/**
 * @param args 10 5 8 10 12 16 24
 */
public static void main(String[] args) {
    if (args.length<2)  return;
    int []valeurs = new int[args.length];
    for (int i=0; i<args.length; i++)
        valeurs[i]=Integer.parseInt(args[i]);
    Vector<Integer> autres = new Vector<Integer>();
    for (int i=0; i<args.length-1; i++) autres.add(new
    Integer(valeurs[i+1]));
    System.out.println("Le résultat de la recherche est
    "+autres.contains(new Integer(valeurs[0]));
    System.out.println("Le résultat de la recherche est
    "+autres.contains(0));
    }
```

adaptation automatique…

La Java Collection Framework

- prétend que ses classes et ses méthodes sont très efficaces…vérifions !
- méthode :
  - nous créons une classe ABR (arbre binaire de recherche) générique simple pour stocker des chaînes de caractères associées à des nombres entiers
  - nous y stockons 100'000 valeurs aléatoires, nous mesurons le temps mis à les stocker, puis à les rechercher toutes
  - nous employons la classe TreeMap pour stocker la même chose et nous mesurons les mêmes temps

Test de performance (1)

- la classe ABR

```
class ABR<E extends Comparable, T>{
     E valeur; T texte;
     ABR<E, T> sAG, sAD;

     public ABR(){
     valeur = null;
     }
     public boolean recherche(E i) {
     if (valeur==null)     return false;
     if (valeur.equals(i)) return true;
     if (valeur.compareTo(i)<0)       return (sAD==null)?false:sAD.recherche(i);
     return (sAG==null)?false:sAG.recherche(i);
     }
     public int size() {
     if (valeur ==  null)return 0;
     return 1+((sAG==null)?0:sAG.size())+((sAD==null)?0:sAD.size());
     }
     public int profondeur(){
     if (valeur == null)   return 0;
     return 1+Math.max(((sAD==null)?0:sAD.profondeur()),
     ((sAG==null)?0:sAG.profondeur()));
     }
```

Test de performance (2)

```
public void add(E i, T t){
    if (i==null)  return;
    if (valeur==null)    {valeur=i; texte = t; return;}
    if (valeur.compareTo(i)<0)  {
            if (sAD == null)     sAD = new ABR<E, T>();
            sAD.add(i, t);
            }
    else {
            if (sAG == null)     sAG = new ABR<E, T>();
            sAG.add(i, t);
            }
    }
}
```

Test de performance (3)

- le programme principal

```
public static void main(String[] args) {
Vector<Integer> v = new Vector<Integer>();
Random r = new Random(System.currentTimeMillis());
for(int i=0; i<100000; i++) v.add(r.nextInt());
ABR<Integer, String> abr = new ABR<Integer, String>();
long initial = System.currentTimeMillis();
for (Integer i : v)          abr.add(i, "abcd");
System.out.println("temps pour ajouter "+(System.currentTimeMillis()-
initial)+" ms.");
System.out.println("nombre de valeurs dans l'arbre "+abr.size()+" et
profondeur "+abr.profondeur()+" au lieu de "+(int)Math.log(v.size()));

long depart = System.currentTimeMillis();
for (Integer i : v){
        if (!abr.recherche(i))      System.err.println("Anomalie");
        }
long tempsCumule = System.currentTimeMillis()-depart;
System.out.println("temps pour tout trouver "+tempsCumule+" s.");
```

Test de performance (4)

```
TreeMap<Integer, String> tm = new TreeMap<Integer, String>();
    initial = System.currentTimeMillis();
    for (Integer i : v)     tm.put(i, "abcd");
    System.out.println("temps pour ajouter
    "+(System.currentTimeMillis()-initial)+" ms.");
    System.out.println("nombre de valeurs dans l'arbre
    "+tm.size());

    depart = System.currentTimeMillis();
    for (Integer i : v){
            if (!tm.containsKey(i))
    System.err.println("Anomalie");
            }
    tempsCumule = System.currentTimeMillis()-depart;
    System.out.println("temps pour tout trouver "+tempsCumule+"
    s.");
    }
```

Test de performance (5)

- les résultats des mesures

```
temps pour ajouter 271 ms.
nombre de valeurs dans l'arbre 100000
   et profondeur 41 au lieu de 11
temps pour tout trouver 210 s.
temps pour ajouter 330 ms.
nombre de valeurs dans l'arbre 99994
temps pour tout trouver 151 s.
```

arbre mal équilibré

plus efficace

Et toujours

- http://www.iut-orsay.fr/~fournier