

2i002 - Conteneurs Génériques

Vincent Guigue



PROBLÉMATIQUE GÉNÉRALE

- Construire une structure de données adaptée à différents types d'entrées
 - Listes : d'entier, de réels, de String...
 - Piles, Files, Tas...
 - ⇒ ne pas construire une classe par cas !!!

PROBLÉMATIQUE GÉNÉRALE

- Construire une structure de données adaptée à différents **types d'entrées**
 - Listes : d'**entier**, de **réels**, de **String**...
 - Piles, Files, Tas...
 - **⇒ ne pas construire une classe par cas!!!**
- Solution (avant Java 1.5) :

```
1 public class ListeGenOld {
2     private final static int TAILLE_MAX = 500;
3     private Object[] liste;
4     private int size;
5
6     public ListeGenOld(){
7         liste = new Object[TAILLE_MAX];
8         size = 0;
9     }
10
11     public void add(Object o){liste[size] = o; size ++;}
12     public Object get(int i){return liste[i];}
13     ... // remove, getSize...
14 }
```

LIMITES

- Obligation de faire des **cast** à chaque récupération d'objet
- **Sécurisation bof** : on peut mettre n'importe quoi dans la structure... + cast à la récupération
- Difficilement compatible avec des **algorithmes génériques** (tri, min, max...)

LE CAS ArrayList (USAGE)

La plupart des structures de données classiques existent déjà

- ArrayList, HashMap, HashSet, PriorityQueue

General-purpose Implementations

Interfaces	Hash table Implementations	Resizable array Implementations	Tree Implementations	Linked list Implementations	Hash table + Linked list Implementations
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Usage :

- ArrayList<E> = liste de E, est un type à part entière
 - déclaration, instantiation classique

```
1 // sur un exemple avec des Integer
2 ArrayList<Integer> maListe = new ArrayList<Integer>();
```

- get : retourne des Integer (pas besoin de cast)
- on ne peut mettre que des Integer dedans

CRÉATION D'UNE CLASSE GÉNÉRIQUE...

Exemple (très) utile : la Paire

La plupart des langages modernes gèrent des N-uplet... Mais pas Java. On peut créer une classe Paire pour retourner facilement plusieurs valeurs depuis une méthode.

CRÉATION D'UNE CLASSE GÉNÉRIQUE...

Exemple (très) utile : la Paire

La plupart des langages modernes gèrent des N-uplet... Mais pas Java. On peut créer une classe Paire pour retourner facilement plusieurs valeurs depuis une méthode.

```
1 public class Paire<A,B> {  
2     private A el1;  
3     private B el2;  
4     public Paire(A el1 , B el2) {  
5         super();  
6         this.el1 = el1;  
7         this.el2 = el2;  
8     }  
9     public A getEl1() {  
10        return el1;  
11    }  
12    public B getEl2() {  
13        return el2;  
14    }  
15 }
```

La syntaxe est celle des ArrayList... Vous la connaissez déjà !

```
1 public static void main(String [] args){  
2     Paire<Integer , String> p =  
3         new Paire<Integer , String >(2, "toto");  
4     System.out.println(p.getEl1()+"␣"+p.getEl2());  
5 }
```

- Le type est donc : Paire<Integer, String>
- Le type du contenant est passé en argument *spécial* entre <>

EXTENSION D'UNE CLASSE GÉNÉRIQUE (1)

Besoin d'une liste avec des méthodes spécifiques... Mais toujours générique

exemple récupération de la case du milieu

```
1 public class MaListeMilieu<E> extends ArrayList<E>{
2     // constructeur sans argument par défaut
3     // Les méthodes sont héritées: add, get, size...
4
5     // Méthode spécifique
6     public E getMilieu(){
7         return super.get(super.size()/2); // division entière
8     }
9
10 }
```

Usage client :

```
1 public static void main(String[] args){
2     MaListeMilieu<Double> li =
3         new MaListeMilieu<Double>();
4     li.add(2.0); li.add(1.4); li.add(3.7);
5     System.out.println(li.getMilieu());
6 }
```

EXTENSION D'UNE CLASSE GÉNÉRIQUE (2)

Besoin d'une liste de *quelque chose*

- ex. du TD : créer un aquarium = liste de Poisson
- Train = liste de wagon
- Population = liste de personne
- ...

```
1 public class Aquarium extends ArrayList<Poisson>{  
2 // bcp de méthodes héritées !!!  
3 }
```

Usage client : la liste ne gère que des poissons

```
1 public static void main(String [] args){  
2     Aquarium aqua = new Aquarium();  
3     aqua.add(new Thon());  
4     aqua.add(new Requin());  
5     ...  
6 }
```

CAST SUR LES OBJETS GÉNÉRIQUES <E>

- 1 Coté *contenu* : très agréable (et classique)
 - objets de types E et descendants de E
 - récupération d'objets dans des variables E

- 2 Coté *contenant* : non flexible

```
1 ArrayList<Personne> pop = new ArrayList<Personne>(); // OK
2 // Etudiant extends Personne
3 ArrayList<Personne> promo = new ArrayList<Etudiant>(); // KO
```

⇒ Une seule issue (en cas de besoin) : la syntaxe *wildcard*

Subsorption *contenu/contenant*

On a besoin de cette propriété pour définir des algorithmes génériques.

Syntaxe : `ArrayList<?>` ou `ArrayList<? extends Poisson>`

N'importe quelle liste, ou n'importe quelle liste d'objets dérivés de poissons :

```
1 ArrayList<? extends Poisson> li = new ArrayList<Thon>();
```

Exemple : comment proposer une technique de recherche de minimum dans une liste sans connaître le type de contenu ?

Subsorption *contenu/contenant*

On a besoin de cette propriété pour définir des algorithmes génériques.

Syntaxe : `ArrayList<?>` ou `ArrayList<? extends Poisson>`

N'importe quelle liste, ou n'importe quelle liste d'objets dérivés de poissons :

```
1 ArrayList<? extends Poisson> li = new ArrayList<Thon>();
```

Exemple : comment proposer une technique de recherche de minimum dans une liste sans connaître le type de contenu ?

- 1 Définir une propriété : *Comparable*
- 2 Définir un algorithme acceptant *n'importe quel conteneur d'objets comparables* en utilisant la syntaxe *wildcard*

ATTENTION : ce type de syntaxe empêche toute modification sur l'objet passé

① Définir une propriété : Comparable

```
1 public interface Comparable<E> {  
2     //retourne -1 si ref < obj, 0 si égalité, 1 sinon  
3     public int compareTo(E obj);  
4 }
```

Avec par exemple un Poisson répondant à la spécification :

```
1 public class Poisson implements Comparable<Poisson>{  
2     private double taille;  
3  
4     public Poisson(double taille) {  
5         super();  
6         this.taille = taille;  
7     }  
8     public double getTaille() {  
9         return taille;  
10    }  
11    public int compareTo(Poisson obj) {  
12        if(taille<obj.taille) return -1;  
13        else if(taille==obj.taille) return 0;  
14        return 1;  
15    }  
16 }
```

② Utiliser la propriété dans un algorithme générique :

```
1 public class GenericTools<E extends Comparable<E>> {
2
3     // constructeur par défaut
4     public E getMinimum(ArrayList<? extends E> liste){
5         E min = liste.get(0);
6         for (int i=1; i<liste.size(); i++){
7             // si : min > liste.get(i)
8             if(min.compareTo(liste.get(i)) == 1)
9                 min = liste.get(i);
10        }
11        return min;
12    }
13 }
```

Usage coté client :

```
1 public static void main(String [] args){
2     ArrayList<Poisson> aquarium = new ArrayList<Poisson>();
3     GenericTools<Poisson> tool = new GenericTools<Poisson>();
4     Poisson lePlusPetit = tool.getMinimum(aquarium);
5 }
```

Générique & static

La même chose en version static : il faut rendre une méthode paramétrique au lieu d'une classe

```

1 // syntaxe static au niveau d'une méthode
2 public static <T extends Object & Comparable<? super T>>
3     T getMinimumStatic(ArrayList<? extends T> liste){
4     T min = liste.get(0);
5     for (int i=1; i<liste.size(); i++){
6         // min > liste.get(i)
7         if(min.compareTo(liste.get(i)) == 1)
8             min = liste.get(i);
9     }
10    return min;
11 }

```

Usage coté client :

```

1 public static void main(String [] args){
2     ArrayList<Poisson> aquarium = new ArrayList<Poisson>();
3     Poisson lePlusPetitv2 =
4         GenericTools.getMinimumStatic(aquarium);
5 }

```


Classe algorithmique de gestion des listes générique

Collections

- min, max, sort, shuffle, indexOf, frequency...

Quelques exemples d'outils disponibles :

<code>static int</code>	<code>frequency(Collection<?> c, Object o)</code> Returns the number of elements in the specified collection equal to the specified object.
<code>static int</code>	<code>indexOfSubList(List<?> source, List<?> target)</code> Returns the starting position of the first occurrence of the specified target list within the specified source list, or -1 if there is no such occurrence.
<code>static <T extends Object & Comparable<? super T>> T</code>	<code>min(Collection<? extends T> coll)</code> Returns the minimum element of the given collection, according to the <i>natural ordering</i> of its elements.
<code>static void</code>	<code>shuffle(List<?> list)</code> Randomly permutes the specified list using a default source of randomness.
<code>static <T extends Comparable<? super T>> void</code>	<code>sort(List<T> list)</code> Sorts the specified list into ascending order, according to the <i>natural ordering</i> of its elements.
<code>static <T> void</code>	<code>sort(List<T> list, Comparator<? super T> c)</code> Sorts the specified list according to the order induced by the specified comparator.