Programmation Orientée Objet

4 – Généricité et Java

P. Berthomé

INSA Centre Val de Loire
Département STI — 3 ème année

5 décembre 2017

Principes de la généricité

Universalité de certains processus

- Le type des données importe peu
  - Parcours d'un tableau
  - affichage d'un tableau
- Quelques opérations doivent éventuellement être définies
  - Recherche du plus grand élément
  - Tri

Généricité

- Ne définir qu'une seule méthode généralisant un grand nombre
- Notion de *polymorphisme paramétrique*

Une première solution

Constat

- Toute classe dérive de la classe *Object*
- Il suffit alors de déclarer les objets de type *Object*
- L'instanciation se fait par polymorphisme

```
public class PaireObjet {
    protected Object premier, second;
    public PaireObjet(Object un, Object deux){
        premier = un;  second = deux;}
  public Object getPremier(){ return premier;}
  public Object getSecond(){ return second; }
  public String toString(){
        return "[" + premier + "," + second + "]";}
}
```


Exemple... suite et quelques problèmes

```
PaireObjet po = new PaireObjet("Hello", "Bonjour");

System.out.println(po);
```

```
String maC = (String) po.getPremier();
```

```
po = new PaireObjet("Hello", 2);
System.out.println(po + " " + maC);

int f = ((Integer) po.getSecond()).intValue();
System.out.println(f);

Integer I = (Integer) po.getSecond();
System.out.println(f);
```

Paramétrage de classe

classe générique

- Paramètres formels spécifiant la *variable de type*
- À définir lors de l'instanciation
- Souvent utilisé pour les *conteneurs*

```
public class Paire<T1, T2> {
    protected T1 premier;
    protected T2 deuxieme;
    public Paire(){premier = null; deuxieme = null; }
    public Paire(T1 un, T2 deux){premier = un;deuxieme = deux;}
    public String toString()
        {return("(" + premier + "," + deuxieme + ")");}
    public T1 getPremier(){ return premier; }
    public T2 getDeuxieme(){return deuxieme; }
}
```


Utilisation

```
Paire<String, String> ps = new Paire<String, String> ("Un", "Deux");
System.out.println(ps);

String un = ps.getPremier();
String deux = ps.getDeuxieme();
System.out.println(un + " " + deux);
```

```
Paire<Integer, String> pis = new Paire<Integer, String>(10, "dix");
System.out.println(pis);
```

Paramétrage de méthodes

Idée

- Utiliser des nouveaux types dans les méthodes

```
public <T> boolean memePremier(Paire<T1,T> pt1t){
      return (premier.equals(pt1t.getPremier()));}
```

```
String un = ps.getPremier();

Paire<String, Integer> psi = new Paire<String, Integer>(un, 1);
if (ps.memePremier(psi))
    System.out.println("Ils ont les mêmes éléments");

psi = new Paire<String, Integer>("Deux", 1);
if (ps.memePremier(psi))
    System.out.println("Ils ont les mêmes éléments");
```


Paramétrage d'interfaces

Interfaces

- Elles peuvent être paramétrées de la même manière

```
public interface IBidon <T>{
  public T valeur();
}
```

```
public class Paire<T1, T2> implements IBidon<T1>{
...
@Override
public T1 valeur() {
return premier;
}
```

```
System.out.println(psi.valeur());
```

Mécanisme

Instanciation

- C++ :
  - Copie du code
  - Remplacement des types formels
  - À la compilation
- Java
  - Effacement de type
  - Types remplacés par le type le plus général possible
  - Ajout de *cast*

```
public class Paire<T1, T2> implements IBidon<T1>{
    static public int nbInst = 0;
    ...
```

```
System.out.println(Paire.nbInst));
```


Sous-typage et spécialisation

Héritage

- Si *PaireDérivée<T1, T2>* hérite de *Paire<T1,T2>* :
- *Paire<String,String> = **new** PaireDérivée<String,String>()*
- *Paire<Object,Object> = **new** Paire<String,String>*

Ajout/Suppression de généricité

- ***class** Dico **extends** Paire<String,String>*
- ***class** MinMax<T> **extends** Paire<T, T>*
- ***class** DicoSpecial<T> **extends** Dico*

Paramétrage contraint

Exemple : classe MinMax

- Avoir deux éléments qui soient les bornes supérieures et inférieures d'ensembles
- Hérite de la classe `Paire`

```
public class MinMax<T> extends Paire<T, T> {
  public MinMax(T[] données){
    if (données == null) {premier = null;deuxieme = null;}
    else{
        premier = deuxieme = données[0];
        for (int i=1; i<données.length; i++){
          if (données[i].compareTo(premier)<0) premier = données[i];
          if (données[i].compareTo(deuxieme)>0) deuxieme =
                données[i];}
         }
}
```


Classe MinMax

Problème

- Pour que le constructeur fonctionne, il faut
  - La méthode *compareTo* soit définie pour `T`
  - Elle est définie dans l'interface `Comparable`
- L'interface `Comparable` est générique

Solution

- Contraindre le type générique à *implémenter* l'interface
- ***public class*** *MinMax<T* ***extends*** *Comparable<T>>* ***extends*** *Paire<T, T>*

Contrainte *extends*

extends

- ***class*** *monType<T* ***extends*** *TypeMinimal> ...*
- indique que le type générique doit dériver de ce type
  - classe
  - interface
- On pourra ensuite utiliser toutes les méthodes de *typeMinimal*

Contraintes multiples

- ***class*** *monType<T* ***extends*** *TypeMin1 & TypeMin2>*
- ***class*** *monType<T* ***extends*** *Comparable<T>>*

Jokers

Wildcards

- Permet de simplifier l'écriture de fonctions paramétrées où le type exact importe peu

```
public <T> boolean memePremier(Paire<T1,T> pt1t){
    return (premier.equals(pt1t.getPremier()));
}
```

```
public boolean mmPrem(Paire<T1, ?> pt1t){
return (premier.equals(pt1t.getPremier()));
}
```

Classes Génériques Java

Collections ou *conteneurs*

- Regroupement de plusieurs objets sous la même entité
- On peut alors facilement :
  - les stocker
  - les récupérer
  - les manipuler

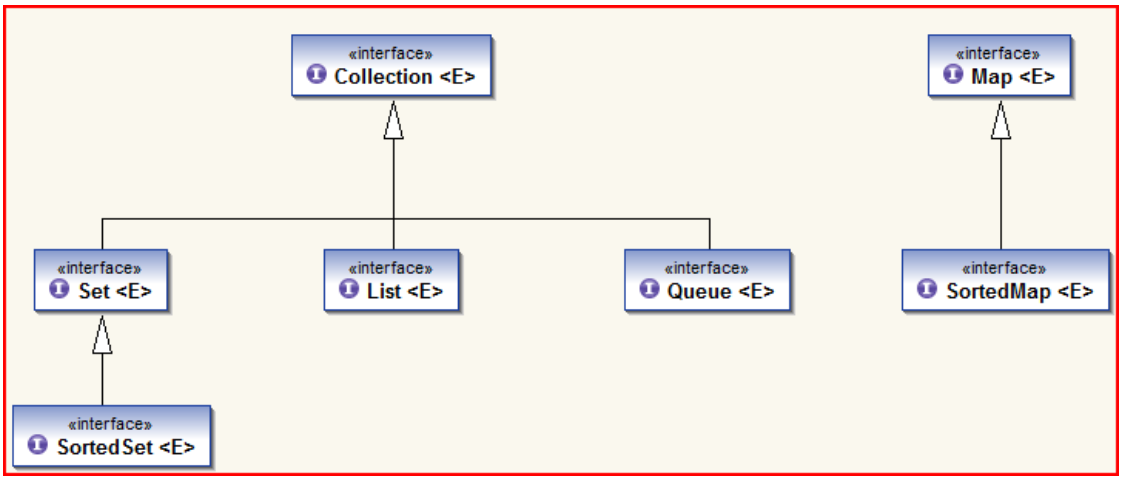
Framework des collections

interfaces : Notions abstraites autour de chacun des types de collection

Implémentations : la structure de données réelle

Algorithmes : actions pour manipuler les collections, le plus souvent sous format *polymorphe*

Hiérarchie Collection



Interface Collection<E>

Opérations

size() : nombre d'éléments

isEmpty() : est-ce vide ?

contains(Object El) :

add(E Elt) : ajout d'un élément

remove(Object Elt) : élimination d'un élément

iterator() : donne un moyen de parcourir efficacement la collection

Parcours d'une collection

for

```
Collection<String> cols = new Vector<String>();
cols.add("un"); cols.add("deux"); cols.add("trois");
for(String s:cols){ System.out.print(s + "\t");}
```

Interface `Iterator<E>`

hasNext() : a-t-on fini de parcourir la collection ?

E next() : le prochain élément de la collection

remove() : élimine l'élément courant

```
for(Iterator<String> it = cols.iterator(); it.hasNext() ;){
    System.out.print(it.next()+"\t");}
```

L'interface List

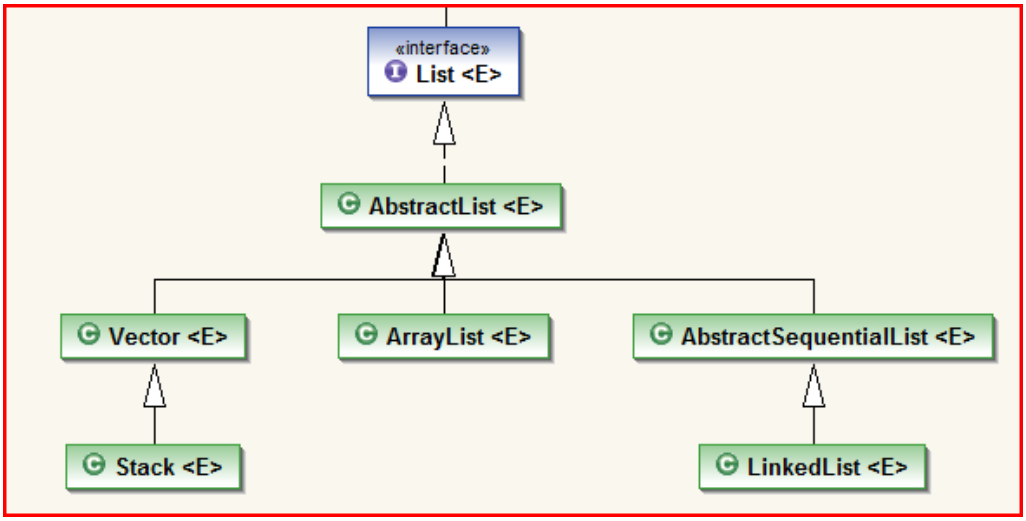
Les listes

- L'interface `List<E>` représente une `Collection` ordonnée (séquence)
- Opérations supplémentaires
  - accès direct :
    - *E get(**int** index)*
    - *E set(**int** index, E elt)*
  - Recherche d'éléments
    - ***int** indexOf(Object o)*
    - ***int** lastIndexOf(Object o)*
  - Itérateurs améliorés `ListIterator<E>`
  - Opérations sur des intervalles
    - *List<E> subList(**int** from, **int** to)*

Exemple

```
List<Integer> li = new Vector<Integer>();
Random rd = new Random();
for(int i = 0; i< 10; i++) li.add(new Integer(rd.nextInt(30)));
for(ListIterator<Integer> lit = li.listIterator();lit.hasNext();)
    System.out.print(lit.next() + "\t");
System.out.println();
for(ListIterator<Integer> lit = li.listIterator(li.size()); lit.hasPrevious();)
    System.out.print(lit.previous() + "\t");
System.out.println();
System.out.println(li.indexOf(5));
System.out.println(li.indexOf(new Integer(5)));
if (li.indexOf(5) != -1) {
  li = li.subList(li.indexOf(5), li.lastIndexOf(5));
  for (Iterator<Integer> lit = li.iterator(); lit.hasNext();)
     System.out.print(lit.next() + "\t");
  System.out.println();}
```

Hiérarchie simplifiée `List`



Classe Vector

Principe

- Un des plus anciens conteneurs
- Implémente un tableau d'objets
- L'accès se fait directement par les indices
- La taille du tableau se modifie dynamiquement en fonction des ajouts

Quelques méthodes supplémentaires

- `capacity()`
- `ElementAt(int)`
- `firstElement()`
- `get(int)`
- `insertElementAt(E, int)`
- `set(int, E)`

Classe `Stack`

Principe

- Dérive de `Vector`
- Met en œuvre une pile

Méthodes spécifiques

- ***boolean*** *empty()*
- *E peek()* regarde l'élément en haut de pile sans dépiler
- *E pop()* élimine le haut de pile
- *E push(E)*
- ***int*** *search(Object o)*

Classe `ArrayList`

Principe

- Meilleure mise en œuvre des vecteurs que `Vector`

```
ArrayList<String> als = new ArrayList<String>();
als.add("Un"); als.add("Deux"); als.add(1, "Un virgule cinq");
for(ListIterator<String> lit = als.listIterator(); lit.hasNext();)
    System.out.print(lit.next() + "\t");
System.out.println();
```

Classe `LinkedList`

Principe

- Mise en place de la liste doublement chainée
- Accès simplement par l'une des deux extrémités
- Parcours par des itérateurs de listes *ListIterator*

Quelques méthodes

- *addFirst(E)*
- *addLast(E)*
- *E getFirst()*
- *E getLast()*
- ***boolean** offer(E)*
- *E peek()*
- *E poll()*
- *E removeFirst()*

Interface `Queue`

Principe

- Interface ajoutant ce qui est nécessaire à une pile généralisée
- Interface mise en œuvre par *LinkedList*

```
public interface Queue<E> extends Collection<E> {
    E element();
    boolean offer(E e);
    E peek();
    E poll();
    E remove();
}
```

Ensembles

Interface Set

- Mêmes méthodes que pour l'interface `Collection`
- Assure que chaque élément est unique

```
Set<String> sets = new HashSet<String>();
int nbDoublons = 0;
for(String s: new String[]{"il", "fait", "beau", "il", "fait", "chaud"}){
    if(!sets.add(s)) nbDoublons++;
}
System.out.println("Nombre d'éléments dans mon ensemble : " +
    sets.size());
System.out.println("Nombre de doublons : " + nbDoublons);
```


Mise en œuvre de `Set`

Implémentations simples

AbstractSet	Squelette minimal des ensembles
HashSet	Mise en œuvre par des fonctions de dispersion
LinkedHashSet	Maintient en plus une liste doublement chaînée sur les éléments
TreeSet	

Associations

Map

- Mise en œuvre des fonctions partielles

```
public interface Map<K,V> {
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();
    public Set<K> keySet();
    public Collection<V> values();
...
}
```


Exemple

```
Map<String, String> defs = new HashMap<String, String>();
defs.put("Un", "Article indéfini singulier");
defs.put("Deux", "Nombre premier pair");
defs.put("Trois", "Premier nombre premier impair");
System.out.println(defs.get("Un"));
System.out.println(defs.containsKey("Deux"));
System.out.println(defs.containsValue("Un petit Cochon"));
defs.remove("Trois");
System.out.println(defs.containsKey("Trois"));
Set<String> sets2 = defs.keySet();
for(String s : sets2){
    System.out.print(s+ "\t");}
Set<Entry<String, String>> sets3 = defs.entrySet();
for(Entry<String, String> s : sets3){
    System.out.print(s+ "\t");}
```