Généricité (fin)
collections…

Chapitre VII

1. Principes généraux
2. Types génériques imbriqués
3. Types paramètres bornés
4. Méthodes génériques

Méthodes génériques

- Supposons que l'on veuille convertir en tableau une File de E
  - on a vu précédemment que l'on ne pouvait ni instancier un objet E ni créer un tableau de E
  - on peut cependant passer un tableau de la taille appropriée à une méthode qui retourne ce tableau:

enTableau1

```
public E[] enTableau1(E[] tab){
     Object[] tmp = tab;
     int i=0;
     for(Cellule<E> c= tete; c != null && i< tab.length;
     c=c.getSuivant())
          tab[i++] = c.getElement();
     return tab;
   }
```

- enTableau1 est une nouvelle méthode de File:

```
File<String> fs=new File<String>();
String[]  u;
u=fs.enTableau1(new String[fs.getTaille()]);
```

enTableau

- Mais,
  - il faut que le tableau passé en paramètre soit un tableau de E, alors qu'un tableau d'une super-classe de E devrait fonctionner (si F est une superclasse de E un tableau de F peut contenir des objets E).
  - avec une méthode générique:

enTableau

```
public <T> T[] enTableau(T[] tab){
    Object[] tmp = tab;
    int i=0;
    for(Cellule<E> c= tete; c != null && i< tab.length;
    c=c.getSuivant())
        tmp[i++] = c.getElement();
    return tab;
}
```

- la déclaration impose que le type du tableau retourné soit du type du tableau de l'argument
- Notons que `tmp` est un tableau d'`Object` ce qui est nécessaire pour le `getSuivant`
- Notons que normalement il faudrait que `T` soit une superclasse de `E` (à l'exécution il peut y avoir une erreur).
- Notons enfin que 'T' ne sert pas dans le corps de la méthode.

Remarque

```
public <T> T[] enTableaubis(T[] tab){
        int i=0;
        for(Cellule<E> c= tete;
              c != null && i< tab.length;
            c=c.getSuivant())
            tab[i++] = (T)c.getElement();
        return tab;
}
```

- provoque un warning `"Cellule.java uses unchecked or unsafe operations".`
- (l'"effacement" ne permet pas de vérifier le type)

Avec Reflection…

- Une autre solution peut être, si on veut créer un vrai tableau, d'utiliser Array.newInstance de la classe: java.lang.reflect

Exemple avec Reflection

```
public E[] enTableau2(Class<E> type){
        int taille = getTaille();
        E[] arr=(E[])Array.newInstance(type,taille);
        int i=0;
        for(Cellule<E> c= tete; c != null && i< taille;
        c=c.getSuivant())
            arr[i++] = c.getElement();
        return arr;
}
```

- on crée ainsi un tableau de "E"
- "unchecked warning": le cast (E[]) n'a pas le sens usuel
- pour fs déclaré comme précédemment on aura:

```
String[] u=fs.enTableau2(String.class); //ok
Object[] v=fs.enTableau2(Object.class); //non
```

- car le type doit être exact

Avec une méthode générique

```
public <T> T[] enTableau3(Class<T> type){
     int taille = getTaille();
     T[] arr=(T[])Array.newInstance(type,taille);
     int i=0;
     Object[] tmp=arr;
     for(Cellule<E> c= tete; c != null && i< taille;
     c=c.getSuivant())
         tmp[i++] = c.getElement();
     return arr;
 }
```

Inférence de type

- Comment invoquer une méthode générique?
- Exemple:

```
static <T> T identite(T obj){
        return obj;
   }
```

Invocations

- On peut explicitement préciser le type:

```
String s1="Bonjour";
String s2= Main.<String>identite(s1);
```

- Mais le compilateur peut, lui-même, trouver le type le plus spécifique:

```
String s1=identite("Bonjour");
```

- On aura:

```
Object o1=identite(s1);            //ok
Object o2=identite((Object)s1);    //ok
s2=identite((Object) s1);          //non!!!
s2=(String)identite((Object) s1);//ok
```

Comment ça marche?

- Mécanisme de l'effacement ("erasure")
- Pour chaque type générique il n'y a qu'une classe: Cellule<String> et Cellule<Integer> ont la même classe
- Effacement:
  - Cellule<String> -> Cellule
    - Cellule est un type *brut*
  - Pour une variable type:
    - <E> -> Object
    - <E extends Number> -> Number
- Le compilateur remplace chaque variable type par son effacement

Comment ça marche?

- Si le résultat de l'effacement du générique ne correspond pas à la variable type, le compilateur génère un cast:
  - par effacement le type variable de File<E> est Object
  - pour un "defiler" sur un objet File<String> le compilateur insère un cast sur String

Comment ça marche?

- A cause de l'effacement, rien de ce qui nécessite de connaître la valeur d'un argument type n'est autorisé. Par exemple:
  - on ne peut pas instancier un type en paramètre: pas de new T() ou de new T[]
  - on ne peut pas utiliser `instanceof` pour une instance de type paramétré
  - on ne peut pas créer de tableau sur un type paramétré sauf s'il est non borné new `List<String>[10]` est interdit mais new `List<?>[10]` est possible.

Comment ça marche?

- les "cast" sont possibles mais n'ont pas la même signification que pour les types non paramétrés:
  - le cast est remplacé par un cast vers le type obtenu par effacement et génère un "unchecked warning" à la compilation.
  - Exemple:
    - on peut caster paramètre File<?> vers un File<String> pour un enfiler (ce qui génère le warning)
    - A l'exécution si le type effectif est File<Number> cela passe… mais le defiler provoquera un ClassCastException.

Comment ça marche?

- Exemple:

```
List<String> l=new ArrayList<String>();
Object o=identite(l);
List<String> l1=(List<String>)o;// warning
List<String> l2=(List)o;//warning
List<?> l3=(List) o; //ok
List<?> l4=(List<?>)o; //ok
```

Application: surcharge

- avoir la même signature s'étend aux méthodes avec variables types
- même signature pour des variables types = même type et même borne (modulo bien sûr renommage!)
- signatures équivalentes par annulation: mêmes signatures où l'effacement des signatures sont identiques

Surcharge

```
class Base<T>{
    void m(int x){};
    void m(T t){};
    void m(String s){};
    <N extends Number> void m(N n){};
    void m(File<?> q){};
}
-----
m(int)
m(Object)
m(String)
m(Number)
m(File)
```

Et héritage…

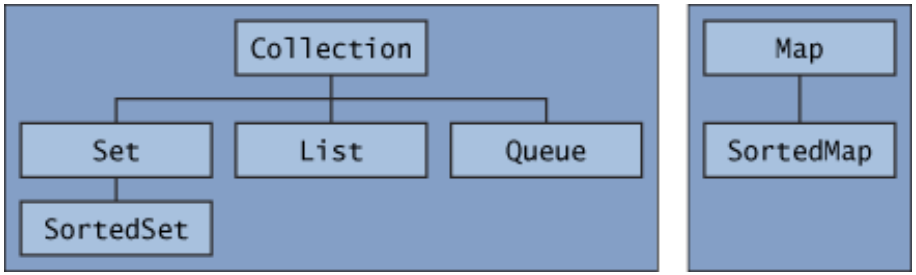
- exemple:

```
class D<T> extends Base<T>{
    void m(Integer i){} //nouveau
    void m(Object t){}  // redéfinit m(T t)
    void m(Number n){}   // redéfinit m(N n)
}
```

Collections

Collections

- types de données
  - interfaces
  - implémentations
  - algorithmes
- Interfaces:



Collections: les interfaces

Les collections sont des interfaces génériques

- Collection<E>: add, remove size toArray…
  - Set<E>: éléments sans duplication
    - SortedSet<E>: ensembles ordonnés
  - List<E>: des listes éléments non ordonnés et avec duplication
  - Queue<E>: files avec tête: peek, poll (défiler), offer (enfiler)
- Map<K,V>:association clés valeurs
- SortedMap<K,V> avec clés triées

Certaines méthodes sont *optionnelles* (si elles ne sont pas implémentées UnsupportedOperationException).

En plus:

- Iterator<E>: interface qui retourne successivement les éléments next(), hasNext(), remove()
- ListIterator<E>: itérateur pour des List, set(E) previous, add(E)

Collection

```
public interface Collection<E> extends Iterable<E> {
    // operations de base
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);          //optionnel
    boolean remove(Object element); //optionnel
    Iterator<E> iterator();

    // operations des collections
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c); //optionnel
    boolean removeAll(Collection<?> c);        //optionnel
    boolean retainAll(Collection<?> c);        //optionnel
    void clear();                              //optionnel

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

Collection

- Les collections sont *génériques*
- Parcours:
  - On peut parcourir les éléments par « for »:

```
for (Object o : collection)
    System.out.println(o);
```

  - Ou avec un Iterator:

```
static void filter(Collection<?> c) {
    for (Iterator<?> it = c.iterator(); it.hasNext();)
        if (!cond(it.next()))
            it.remove();
}
```

Collection

- On peut convertir une collection en tableau
  - En tableaux de Object
  - En tableaux d'objet du type paramètre de la collection
- Il existe aussi une *classe* Collections qui contient des méthodes statiques utiles

Set

- Interface pour contenir des objets différents
  - Opérations ensemblistes
  - SortedSet pour des ensembles ordonnés
- Implémentations:
  - HashSet par hachage (preformances)
  - TreeSet arbre rouge-noir
  - LinkedHashSet ordonnés par ordre d'insertion

Set

```
public interface Set<E> extends Collection<E> {
    // opérations de base
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);          //optionnel
    boolean remove(Object element); //optionnel
    Iterator<E> iterator();

    // autres
    boolean containsAll(Collection<?> c); // sous-ensemble
    boolean addAll(Collection<? extends E> c); //optionnel- union
    boolean removeAll(Collection<?> c);        //optionnel- différence
    boolean retainAll(Collection<?> c);        //optionnel- intersection
    void clear();                              //optionnel

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

Exemple:

```
public static void chercheDoublons(String ... st){
       Set<String> s = new HashSet<String>();
       for (String a : st)
           if (!s.add(a))
               System.out.println("Doublon: " + a);

       System.out.println("il y a "+s.size() + " mots différents: " + s);
}
public static void chercheDoublonsbis(String st[]){
       Set<String> s=new HashSet<String>();
       Set<String> sdup=new HashSet<String>();
       for(String a :st)
           if (!s.add(a))
               sdup.add(a);
       s.removeAll(sdup);
       System.out.println("Mots uniques:    " + s);
       System.out.println("Mots dupliqués: " + sdup);
 }
```

Lists

- En plus de Collection:
  - Accès par position de l 'élément
  - Recherche qui retourne la position de l'élément
  - Sous-liste entre deux positions
- Implémentations:
  - ArrayList
  - LinkedList

List

```
public interface List<E> extends Collection<E> {
    // accès par position
    E get(int index);
    E set(int index, E element);    //optional
    boolean add(E element);         //optional
    void add(int index, E element); //optional
    E remove(int index);            //optional
    boolean addAll(int index,
        Collection<? extends E> c); //optional

    // recherche
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteration
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    // sous-liste
    List<E> subList(int from, int to);
}
```

Itérateur pour listes

```
public interface ListIterator<E> extends Iterator<E> {
    boolean hasNext();
    E next();
    boolean hasPrevious();
    E previous();
    int nextIndex();
    int previousIndex();
    void remove(); //optional
    void set(E e); //optional
    void add(E e); //optional
}
```

Exemple

```
public static <E> void swap(List<E> a, int i, int j) {
        E tmp = a.get(i);
        a.set(i, a.get(j));
        a.set(j, tmp);
}
public static void melange(List<?> list, Random rnd) {
        for (int i = list.size(); i > 1; i--)
            swap(list, i - 1, rnd.nextInt(i));
}
```

Suite…

```
public static <E> List<E> uneMain(List<E> deck, int n) {
       int deckSize = deck.size();
       List<E> handView = deck.subList(deckSize - n, deckSize);
       List<E> hand = new ArrayList<E>(handView);
       handView.clear();
       return hand;
}
public static void distribuer(int nMains, int nCartes) {
       String[] couleurs = new String[]{"pique","coeur","carreau","trèfle"};
       String[] rank = new String[]
       {"as","2","3","4","5","6","7","8", "9","10","valet","dame","roi"};
       List<String> deck = new ArrayList<String>();
       for (int i = 0; i < couleurs.length; i++)
           for (int j = 0; j < rank.length; j++)
               deck.add(rank[j] + " de " + couleurs[i]);
       melange(deck,new Random());
       for (int i=0; i < nMains; i++)
           System.out.println(uneMain(deck,nCartes));
   }
```

Map

- Map associe des clés à des valeurs
  - Association injective: à une clé correspond exactement une valeur.
  - Trois implémentations, comme pour set
    - HashMap,
    - TreeMap,
    - LinkedHashMap
  - Remplace Hash

Map

```
public interface Map<K,V> {
    // Basic operations
    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();
    // Bulk operations
    void putAll(Map<? extends K, ? extends V> m);
    void clear();
    // Collection Views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();
    // Interface for entrySet elements
    public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

Exemples

```
 public static void mapFreq(String ... t) {
        Map<String, Integer> m = new HashMap<String,
                                              Integer>();
        for (String a : t) {
            Integer freq = m.get(a);
            m.put(a, (freq == null) ? 1 : freq + 1);
        }
        System.out.println("Il y a: " + m.size() +
            " mots différents:\n"+m);
    }
// ordre arbitraire
```

Exemples

```
public static void mapFreq(String ... t) {
        Map<String, Integer> m = new TreeMap<String,
                                             Integer>();
        for (String a : t) {
            Integer freq = m.get(a);
            m.put(a, (freq == null) ? 1 : freq + 1);
        }
        System.out.println("Il y a: " + m.size() +
            " mots différents:\n"+m);
    }
// ordre arbitraire
```

Exemples

```
public static void mapFreq(String ... t) {
        Map<String, Integer> m = new LinkedHashMap<String,
                                                  Integer>();
        for (String a : t) {
            Integer freq = m.get(a);
            m.put(a, (freq == null) ? 1 : freq + 1);
        }
        System.out.println("Il y a: " + m.size() +
            " mots différents:\n"+m);
    }
// ordre arbitraire
```

Queue

- Pour représenter une file (en principe FIFO):
  - Insertion: offer -add
  - Extraction: poll - remove
  - Pour voir: peek –element
  - (retourne une valeur - exception
- PriorityQueue implémentation pour une file à priorité

Interface Queue

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

Exemple

```
public static void compteur(int n)
            throws InterruptedException {
        Queue<Integer> file = new
                    LinkedList<Integer>();
        for (int i = n; i >= 0; i--)
            file.add(i);
        while (!file.isEmpty()) {
            System.out.println(file.remove());
            Thread.sleep(1000);
        }
    }
```

Exemple

```
static <E> List<E> heapSort(Collection<E> c) {
    Queue<E> queue = new PriorityQueue<E>(c);
    List<E> result = new ArrayList<E>();
    while (!queue.isEmpty())
        result.add(queue.remove());
    return result;
}
```

Des implémentations

- HashSet<E>: implémentation de Set comme table de hachage. Recherche/ ajout suppression en temps constant
- TreeSet<E>: SortedSet comme arbre binaire équilibré O(log(n))
- ArrayList<E>: liste implémentée par des tableaux à taille variable accès en O(1) ajout et suppression en O(n-i) (i position considérée)
- LinkedList<E>: liste doublement chaînée implémente List et Queue accès en O(i)
- HashMap<K,V>: implémentation de Map par table de hachage ajout suppression et recherche en O(1)
- TreeMap<K,V>: implémentation de SortedMap à partir d'arbres équilibrés ajout, suppression et recherche en O(log(n))
- WeakHashMap<K,V>: implémentation de Map par table de hachage
- PriorityQueue<E>: tas à priorité.

Comparaisons

- Interface Comparable<T> contient la méthode
  - public int compareTo(T e)
  - "ordre naturel"
- Interface Comparator<T> contient la méthode
  - public int compare(T o1, T o2)

Quelques autres packages

- System méthodes static pour le système:
  - entrée-sorties standard
  - manipulation des propriétés systèmes
  - utilitaires "Runtime" exit(), gc() …

Runtime, Process

- Runtime permet de créer des processus pour exécuter des commande: exec
- Process retourné par un exec méthodes
  - destroy()
  - exitValue()
  - getInputStream()
  - getOutputStream()
  - getErrorStream()

Exemple

- exécuter une commande (du système local)
  - associer l'entrée de la commande sur System.in
  - associer la sortie sur System.out.

Exemple

```
class plugTogether extends Thread {
    InputStream from;
    OutputStream to;
    plugTogether(OutputStream to, InputStream from ) {
        this.from = from; this.to = to;
    }
    public void run() {
        byte b;
        try {
            while ((b= (byte) from.read()) !=  -1) to.write(b);
        } catch (IOException e) {
            System.out.println(e);
        } finally {
            try {
                to.close();
                from.close();
            } catch (IOException e) {
                System.out.println(e);
            }
        }
    }
}
```

Exemple suite

```
public class Main {
    public static Process userProg(String cmd)
    throws IOException {
      Process proc = Runtime.getRuntime().exec(cmd);
      Thread thread1 = new plugTogether(proc.getOutputStream(), System.in);
      Thread thread2 = new plugTogether(System.out, proc.getInputStream());
      Thread thread3 = new plugTogether(System.err, proc.getErrorStream());
      thread1.start(); thread2.start(); thread3.start();
      try {proc.waitFor();} catch (InterruptedException e) {}
        return proc;
    }
    public static void main(String args[])
    throws IOException {
       String cmd = args[0];
       System.out.println("Execution de: "+cmd);
       Process proc = userProg(cmd);
    }
}
```