Le langage Java

Apprentissage en lien avec le langage UML

Le langage Java

Apprentissage en lien avec le langage UML

L'interet des collections

- Lors d'un développement, l'un des points importants est le choix de structures de données adaptées.
- Il est donc souvent nécessaire de développer les mêmes solutions pour
  - Stocker des objets, pour les parcourir et les retrouver (avec ou sans clé)
  - Sous la forme de tableaux, listes, arbres, tables de hachages, ...

Les classes historiques

- Les jdk 1.1 proposaient les classes : `Vector` et `HashTable`
- Ces classes ont été rendues obsolètes
- Elles existent toujours et peuvent être rencontrées dans des programmes existants

Les collections

- En java, une collection est un objet qui permet de contenir des références vers d'autres objets
- Nous avons déjà vu un type de collection : les tableaux
- Le jdk propose
  - Des Interfaces qui définissent les grands types de collections
  - Des classes abstraites qui implantent partiellement ces interfaces
  - Des classes concrètes qui précisent les implantations
- Elles sont définies dans le paquetage `java.util`
- Un mécanisme de parcours général est proposé : *les itérateurs*
- Et des outils pour trier et rechercher

Collections et indexation

- On distinguera :
  - Les collections au sens large décrites dans l'interface `Collection`
  - Et les collections indexées décrites dans l'interface `Map`
    - On associe à un objet une clé (un autre objet)
    - On peut alors indexer et rechercher par clé.
- Dans tous les cas, on pourra
  - ajouter un objet (`add()`)
  - parcourir la collection (voire obtenir un objet (`get()`))

Une hiérarchie d'Interfaces

- On distiguera parmis les collections (`Collection`)
  - Les ensembles (`Set`)
    - Les ensembles triés (`SortedSet`)
  - Les listes (`List`)
  - Les files (`Queue`)
- Et parmis les maps (`Map`) les maps triées (`SortedMap`)

Des classes abtraites

- Un ensemble de classes abstraites réalisent ces interfaces
  - Elles implantent les parties communes
    - `AbstractCollection`, `AbstractList`, `AbstractQueue`, `AbstractSequentialList`, `AbstractSet`, ...
    - `AbstractMap`
- Ces classes abstraites (et donc les interfaces correspondantes) sont réalisées par des classes concrètes
  - `ArrayList`, `TreeSet`, ...
  - `HashMap`, `TreeMap`, ...

Algorithmique et structure de données

- Les structures de données classiques sont disponibles et utilisées pour les implantations concrètes
- Elles sont organisées dans une hiérarchie de classes et implantent des interfaces communes
- Par exemple, il existe (parmis d'autres) :
  - Des tableaux de taille variable : `ArrayList`
    - héritent de `java.util.AbstractList` et de `AbstractCollection`
    - implantent les interfaces `Collection` et `List`
  - Des listes chaînées : `LinkedList`
  - Des tables de hachages : `HashSet` et `HashMap`
  - Des arbres à balance équilibrés : `TreeSet` et `TreeMap`

Administration des collections

- L'administration des instances des collections est assurée par des méthodes statiques des classes :
  - `Collections`
    - pour trier et convertir des collections
    - pour rechercher efficacement
  - `Arrays`

Un premier exemple

- Création d'une liste de chaînes de caractères sous forme d'un `ArrayList`
- Tri et affichage

```
package coursSSI3.exemples.collections;
import java.util.*;
public class PremierExemple {
        public static void main(String[] args) {
                List l = new ArrayList();
                l.add("Medor");
                l.add("Rex");
                l.add("Brutus");
                Collections.sort(l);
                System.out.println(l);
        }
}
```

Listing 1 – coursSSI3/exemples/collections/PremierExemple.java

L'interface Collection

- La racine de la hiérarchie des collections
  - `boolean add(E o)`
  - `boolean addAll(Collection<? extends E> c)`
  - `void clear()`
  - `boolean contains(Object o)`
  - `boolean containsAll(Collection<?> c)`
  - `boolean equals(Object o)`
  - `int hashCode()`
  - `boolean isEmpty()`
  - `Iterator<E> iterator()`
  - `boolean remove(Object o)`
  - `boolean removeAll(Collection<?> c)`
  - `boolean retainAll(Collection<?> c)`
  - `int size()`
  - `Object[] toArray()`
  - `<T> T[] toArray(T[] a)`

Les standards et les exceptions

- Les constructeurs sans paramètres et avec une `Collection` en paramètre sont “toujours” disponibles
  - Cela ne peut pas être garanti (pas de constructeur dans les interfaces)
  - Cela facilite les conversions entre les implantations différentes
- Toutes les méthodes de l'interface sont implantées (c'est obligatoire) mais :
  - Pour certaines spécialisations certaines méthodes n'ont pas de sens
    - Exemple : collection en lecture seule (`add()`, `put()`, ...)
  - L'implantation consiste alors à retourner une *exception* : `UnsupportedOperationException`

List et ArrayList

- L'interface `List`
  - Une collection ordonnées (une séquence)
  - L'utilisateur controle la position d'insertion des éléments
  - L'utilisateur accède à cet élément par son index (un entier)
- La classe `ArrayList`
  - Une implantation redimensionnable de l'interface `List`
  - Toutes les méthodes sont implantées
  - Il est possible de contrôler la taille du tableau utilisé en interne

Des collections “génériques” ?

- Les collections représentent des ensembles d'objets (cf. type de retour de l'interface)
- Un objet ou un ensemble d'objets extraits sont donc du type `Object`
- La conséquence est donc que les objets extraits doivent être transtypés
- De plus les fonctions prennent en paramètres des instances de `Object`, il est donc difficile de contrôler la consistance.

Un exemple de transtypage

```
package coursSSI3.exemples.collections;

import java.util.*;
import coursSSI3.exemples.animaux.*;
public class Transtypepage {
        public static void main(String[] args) {
                List l = new ArrayList();
                l.add(new Chien());
                l.add(new Chien());
                Chien c = (Chien)l.get(0);
                c.aboyer();
                ((Chien)l.get(1)).aboyer();

                // que faire ?
                List l2 = new ArrayList();
                l2.add(new Chien());
                l2.add(new Chat());
        }
}
```

Listing 2 – coursSSI3/exemples/collections/Transtypepage.java

Le parcours d'une collection (1/2)

- Les collections indexées (Listes, ...) peuvent être parcourues avec une boucle
  - Cette méthode n'est pas bonne pour l'évolutivité, elle supprime l'encapsulation des collections
- Java introduit la notion d'itérateur

Definition

Un itérateur est une instance de la classe `Iterator` qui permet d'énumérer les éléments d'une collection.

- Toutes les collections possèdent la méthode `iterator()` qui retourne un itérateur.
  - Cette itérateur peut être spécialisé en fonction des sous-classes de `Collection` (Ex : `ListIterator`).
- Un iterateur permet de modifier la collection en cours de parcours.

Le parcours d'une collection - Exemple (2/2)

```
package coursSSI3.exemples.collections;

import java.util.*;

import coursSSI3.exemples.animaux.*;

public class Parcours {
 public static void main(String[] args) {
  List listeDeChiens = new ArrayList();
  listeDeChiens.add(new Chien());listeDeChiens.add(new Chien());

  for (int i=0;i<listeDeChiens.size();i++)
   System.out.println((Chien)listeDeChiens.get(i));

  Iterator itDeChiens = listeDeChiens.iterator();
  while (itDeChiens.hasNext())
   ((Chien)itDeChiens.next()).aboyer();
  }
}
```

Listing 3 – coursSSI3/exemples/collections/Parcours.java

La conversion vers un tableau

- `toArray()` : **Attention au type de retour**
- Il est possible de passer un tableau en paramètre
  - **Il ne suffit pas de transtyper le résultat** (pourquoi ?)
  - Si celui-ci est trop petit, un tableau du même type est créé.
- Collection vide implique tableau vide (et non `null`)

```
package coursSSI3.exemples.collections;
import java.util.*;
import coursSSI3.exemples.animaux.*;
public class ToArray {
 public static void main(String[] args) {
  List listeDeChiens = new ArrayList();
  listeDeChiens.add(new Chien()); listeDeChiens.add(new Chien());
  Object[] tObjets = listeDeChiens.toArray();
  Chien[] desChiens = new Chien[listeDeChiens.size()];
  listeDeChiens.toArray(desChiens);
  for(Object c:tObjets){((Chien)c).aboyer();};
  for(Object c:listeDeChiens){((Chien)c).aboyer();};
  for(Chien c:desChiens){c.aboyer();};
 }
}
```

Listing 4 – coursSSI3/exemples/collections/ToArray.java

Des collections “génériques” avec Java 5 (1/2)

- La boucle *foreach* (cf. tableaux) s'applique aux collections

Definition

Un type générique (en anglais *generic* ou aussi type paramétré) permet de définir une classe ou une interface qui aura des types différents (grace à des types passés en parmètres) lors de l'instanciation.

`http://java.sun.com/j2se/1.5.0/docs/guide/language/generics.html`

- Définir des classes spécifiques au type indiqué. On peut donc définir une “collection de `Chiens`”.
- Vérifier les paramètres des méthodes et convertir les types de retour.
- Les collections ne traitent que des objets
  - Pour les types primitifs, il faut utiliser les classes enveloppantes
  - Java 5 propose l'*autoboxing* et l'*autounboxing* qui convertit automatiquement les primitifs.

Les “génériques” de Java 5 (2/3)

```
package coursSSI3.exemples.collections;

public class Couple<T1, T2> {
          final T1 e1;
          final T2 e2;

          public Couple(T1 p1, T2 p2) {
            e1 = p1; e2 = p2;
          }
        }
```

Listing 5 – coursSSI3/exemples/collections/Couple.java

```
package coursSSI3.exemples.collections;

import coursSSI3.exemples.animaux.*;
public class Generique {


        public static void main(String[] args) {
                Chien unChien = new Chien();
                Chat unChat = new Chat();
                Couple<Chien,Chat> c =
                        new Couple<Chien,Chat>(unChien, unChat);
        }

}
```

Listing 6 – coursSSI3/exemples/collections/Generique.java

Des collections “génériques” avec Java 5 - Exemple (2/3)

```
package coursSSI3.exemples.collections;

import java.util.*;

import coursSSI3.exemples.animaux.*;

public class Java5 {
 public static void main(String[] args) {
  List<Chien> listeDeChiens = new ArrayList<Chien>();
  listeDeChiens.add(new Chien());listeDeChiens.add(new Chien());
  // Provoque une erreur de compilation
     //listeDeChiens.add(new Chat());
  for(Chien c:listeDeChiens) c.aboyer();

  List<Integer> l = new ArrayList<Integer>();
  l.add(new Integer(3));
  int i = l.get(0);

  l.add(4);
  i = l.get(1);
  }
}
```

Listing 7 – coursSSI3/exemples/collections/Java5.java

L'utilisation des `Map`

- Une `Map` est utilisée pour associer une clé à une valeur
- Une clé est un objet qui est associé à une et une seule valeur
- Une clé doit être unique, plus précisément la valeur retournée par la méthode `equals()` de deux clés doit être différente
- En Java, il s'agit d'une interface implantée en particulier par :
  - `HashMap` : Une table de hachage (accès en $\theta(1)$)
  - `TreeMap` : Un arbre à balance équilibré qui ordonne les valeurs en fonction des clés :
    - L'accès est en $\theta(n)$ (n le nombre d'éléments)
    - Les objets doivent implanter l'interface `Comparable`
    - On peux aussi fournir un comparateur (Instance de `Comparator`) externe
- Utilisation classique :
  - ajouter et enlever des entrées
  - retrouver un objet par sa clé
  - retrouver la ou les clés associées à une valeur

L'interface Map<K,V>

- void clear()
- boolean containsKey(Object key)
- boolean containsValue(Object value)
- Set<Map.Entry<K,V>> entrySet()
- boolean equals(Object o)
- V get(Object key)
- int hashCode()
- boolean isEmpty()
- Set<K> keySet()
- V put(K key, V value)
- void putAll(Map<? extends K,? extends V> t)
- V remove(Object key)
- int size()
- Collection<V> values()

Fonctionnement de `Map`

- Par défaut une `Map` stocke des instances de `Object`
- Les génériques sont utilisables (et conseillés)
- Les couples clé/valeur sont manipulés *via* l'interface `Map.Entry`
  - Trois méthodes : `getKey()`, `getValue()`, `setValue()`
  - La méthode `entrySet()` de `Map` retourne un ensemble (`Set`) de `Map.Entry`
- **Attention**, pour garantir le bon fonctionnement une clé ne peut être remplacée que par une clé égale (cf. `equals()`)
  - Un conseil : faire un ajout et une supression.

Exemple Map

```
package coursSSI3.exemples.collections;
import java.util.*;
import coursSSI3.exemples.animaux.*;
public class MapSimple {
 public static void main(String[] args) {
  Map<String, Chien> m = new HashMap<String, Chien>();
  //Map<String, Chien> m = new TreeMap<String, Chien>();
  //SortedMap<String, Chien> m = new TreeMap<String, Chien>();

  m.put("Ch3",new Chien("Medor"));
  m.put("Ch1",new Chien("Rex"));
  m.put("Ch2",new Chien("Medor"));
  m.put("Ch2",new Chien("Brutus"));
  System.out.println("Le chien Ch2 est "+m.get("Ch2").nom);
 }

}
```

Listing 8 – coursSSI3/exemples/collections/MapSimple.java

Parcourir une Map

- Pour parcourir une Map
  - On récupère l'ensemble des clés (`keySet()`)
  - On parcours cet ensemble (*Itérateur* ou *for each*)
  - Pour chaque entrée on récupère éventuellement la clé (`getKey()`) et la référence vers la valeur (`getValue()`)
  - `values()` retourne la collection correspondante

```
package coursSSI3.exemples.collections;
import java.util.*;
import coursSSI3.exemples.animaux.*;
public class ParcoursMap {
 public static void main(String[] args) {
  Map<String, Chien> m = new HashMap<String, Chien>();
  m.put("Ch3",new Chien("Medor"));m.put("Ch1",new Chien("Rex"));
  m.put("Ch2",new Chien("Medor"));
  Set<Map.Entry<String, Chien>> setChiens = m.entrySet();
  for(Map.Entry<String, Chien> uneEntree: setChiens)
   { System.out.print(uneEntree.getKey()+ " :" );
          uneEntree.getValue().aboyer(); }
 }
}
```

Listing 9 – coursSSI3/exemples/collections/ParcoursMap.java

Les critères de choix de classe

- Un des objectifs des programmes Java est la réutilisabilité
  - Il faut choisir la classe (ou l'interface) la plus adaptée
  - Mais le programme doit être utilisable dans un nombre de cas le plus large possible
- On choisira donc plutot :
  1. De manipuler les objets via des interfaces les plus générales possibles pour les paramètres
  2. Et il faudrait choisir des types de retour des objets instances des classes les plus spécifiques possibles (offrant le plus de fonctions)
  3. Cependant, ce dernier choix provoquerait des problèmes en cas de changement d'implantation ; on choisira donc aussi des interfaces plus générales.

Utilitaires

- Les classes `Collections` et `Arrays` propose des méthodes de classe pour traitre des collections et des tableaux.
- Elles permettent en particulier de trier et de rechercher dans des listes, de faire des copies, ...
- Nous avons déjà vu que l'on pouvait trier une collection de `String`, en réalité pour qu'une collection puisse être triée, ses élements doivent implanter l'interface `Comparable`.
  - `int compareTo(T o)`
    Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.
- Attention à la consistance entre `Comparable()` et `equals()`.

Trier une collection (1/3)

```
package coursSSI3.exemples.animaux;

public class Animal implements Comparable<Animal> {
        public final String nom;
        public final int age;
        public final int poids;

        public Animal() {this("",-1,-1);}
        public Animal(String nom) {this(nom, -1, -1);}
        public Animal(String nom, int age, int poids)
        {this.nom=nom;this.age = age;this.poids=poids;}

        public int compareTo(Animal o) {
                        return (age - ((Animal) o).age);
        }
        public boolean equals(Animal o) {
                return age==o.age;
        }
        public String toString() {
                return nom+" "+age+" an(s)"+" "+poids+" kg.";}
}
```

Listing 10 – coursSSI3/exemples/animaux/Animal.java

Trier une collection (2/3)

```
package coursSSI3.exemples.collections;
import java.util.*;
import coursSSI3.exemples.animaux.*;
public class Utilitaire {
 public static void main(String[] args) {
  List<Animal> l = new ArrayList<Animal>();
  //Nom, age, poids
  l.add(new Chien("Medor",2,5));
  l.add(new Chat("Figaro",3,2));
  l.add(new Chien("Brutus",1,15));

  System.out.println(l);
  Collections.sort(l); //Trier par age
  System.out.println(l);

  //Trier par poids ?
  Collections.sort(l,new ComparateurPoidsAnimal());
  System.out.println(l);
 }
}
```

Listing 11 – coursSSI3/exemples/collections/Utilitaire.java

Trier une collection (3/3)

```
package coursSSI3.exemples.collections;

import java.util.*;
import coursSSI3.exemples.animaux.Animal;
public class ComparateurPoidsAnimal
  implements Comparator<Animal> {
        public int compare(Animal arg0, Animal arg1) {
                return arg0.poids-arg1.poids;
        }

}
```

Listing 12 – coursSSI3/exemples/collections/ComparateurPoidsAnimal.java

Recherche dans une collection

```
package coursSSI3.exemples.collections;

import java.util.*;
import coursSSI3.exemples.animaux.*;

public class Recherche {
 public static void main(String[] args) {
  List<Animal> l = new ArrayList<Animal>();
  //Nom, age, poids
  l.add(new Chien("Medor",2,5));
  Chat figaro;
  l.add(figaro=new Chat("Figaro",3,2));
  l.add(new Chien("Brutus",1,15));

  Collections.sort(l);
  int positionFigaroAge = Collections.binarySearch(l, figaro);

  Collections.sort(l, new ComparateurPoidsAnimal());
  int positionFigaroPoids = Collections.binarySearch(l, figaro,
    new ComparateurPoidsAnimal());
  System.out.println(positionFigaroAge+ " "+positionFigaroPoids);
 }
}
```

Listing 13 – coursSSI3/exemples/collections/Recherche.java

Arguments variables

- Avec Java 5, il n'est plus nécessaire d'utiliser une collection pour passer un ensemble d'arguments du même type à ne méthode

```
package coursSSI3.exemples.collections;

public class VarArgs {
        public static int somme(int ... entiers) {
                int total = 0;
                for (int e:entiers) total+=e;
                return total;
        }

        public static void main(String[] args) {
                System.out.println("1+2="+somme(1,2)+
                                "␣et␣1+2+3+4="+somme(1,2,3,4));
        }
}
```

Listing 14 – coursSSI3/exemples/collections/VarArgs.java

EnumSet

Special-purpose Set and Map implementations are provided for use with enums :

- `EnumSet` - a high-performance Set implementation backed by a bit-vector. All elements of each EnumSet instance must be elements of a single enum type.
- `EnumMap` - a high-performance Map implementation backed by an array. All keys in each EnumMap instance must be elements of a single enum type.