Collection et généricité

Victor Marsault
Aldric Degorre

CPOO 2015

Outline

Deux types d'égalité

==

- Compare les emplacements mémoires des instances
- si `(a==b)` alors
  - `a` et `b` font référence à la même instance;
  - toute modification de l'un modifie l'autre.

Deux types d'égalité

==

- Compare les emplacements mémoires des instances
- si `(a==b)` alors
  - `a` et `b` font référence à la même instance;
  - toute modification de l'un modifie l'autre.

`boolean equals(Object obj)`

- Est une méthode de `Object` qui est originalement:
  `{ return (this == obj); }`
- Peut-être redéfinie (comme toute méthode).
- La signification de `(a.equals(b))` n'est pas standard.
  Elle dépend des types de `a` et `b`.

Redéfinir equals


```
public class MonEntier {
    int valeur;

    @Override
    public boolean equals(Object obj) {
        //si obj est de type MonEntier
        if (obj instanceof MonEntier)
            // et a la meme valeur que this
            if (((MonEntier) obj).valeur == valeur )
                return true; // renvoie true

    return false; // dans tous les autres cas
                  // on renvoie false
    }
}
```

```
public static void main (String args) {
  MonEntier i = new MonEntier();
  MonEntier j = new MonEntier();
  i.valeur = 10;
  j.valeur = 10;

  System.out.println(i == j);       // false
  System.out.println(i.equals(j)); // true
}
```

Attention ! (2)


```
public class MonEntierDeux extends MonEntier {
  int deuxiemeVal;  // On rajoute une deuxième valeur

  @Override
  public boolean equals(Object obj) {
    if (obj instanceof MonEntierDeux)
      if ((((MonEntierDeux) obj).valeur == valeur )
            &&
          (((MonEntierDeux) obj).deuxiemeVal == deuxiemeVal))
        return true;

    return false;
  }
```

Attention ! (3)


```
public static void main (String args) {
    MonEntier i = new MonEntier();
    MonEntierDeux j = new MonEntierDeux();
    i.valeur = 10; j.valeur = 10;
    j.deuxiemeValeur = 10;

    System.out.println(i == j);      // false
    System.out.println(i.equals(j)); // true
    System.out.println(j.equals(i)); // false
}
```

Redéfinir `equals`

- change la façon dont Java traite l'égalité (cf section suivante);
- demande des précautions : non-réflexivité, etc;
- demande (en théorie) la ré-implémentation de `hashcode()`.

Outline

Rappel sur Iterator et Iterable


```
public interface Iterator<E>{
    boolean hasNext();
    E next() throws NoSuchElementException;
    void remove() throws UnsupportedOperationException,
                         IllegalStateException;
}

public interface Iterable<E> {
    Iterator<E> iterator()
}
```

Collection


```
public interface Collection<E> extends Iterable<E> {
    boolean add(E elem);
    boolean addAll(Collection<? extends E> collec);
```

Collection


```
public interface Collection<E> extends Iterable<E> {
    boolean add(E elem);
    boolean addAll(Collection<? extends E> collec);

    int size();
    boolean contains(Object obj);
    boolean containsAll(Collection<?> collec);
    boolean isEmpty();
```

Collection


```
public interface Collection<E> extends Iterable<E> {
    boolean add(E elem);
    boolean addAll(Collection<? extends E> collec);

    int size();
    boolean contains(Object obj);
    boolean containsAll(Collection<?> collec);
    boolean isEmpty();

    Iterator<E> iterator();   // hérité de Iterable<E>
```

Collection


```
public interface Collection<E> extends Iterable<E> {
    boolean add(E elem);
    boolean addAll(Collection<? extends E> collec);

    int size();
    boolean contains(Object obj);
    boolean containsAll(Collection<?> collec);
    boolean isEmpty();

    Iterator<E> iterator();   // hérité de Iterable<E>

    boolean remove(Object o);
    boolean removeAll(Collection<?> collec);
    boolean retainAll(Collection<?> collec);
    void clear();
```

Collection


```
public interface Collection<E> extends Iterable<E> {
    boolean add(E elem);
    boolean addAll(Collection<? extends E> collec);

    int size();
    boolean contains(Object obj);
    boolean containsAll(Collection<?> collec);
    boolean isEmpty();

    Iterator<E> iterator();   // hérité de Iterable<E>

    boolean remove(Object o);
    boolean removeAll(Collection<?> collec);
    boolean retainAll(Collection<?> collec);
    void clear();

    ...
}
```

Collection (2)

Structure de donnée

- Ajout: `add` et `addAll`
- Retrait: `remove` et `removeAll`
- Appartenance: `contains` et `containsAll`
- Parcours: `iterator`

Collection (2)

Structure de donnée

- Ajout: `add` et `addAll`
- Retrait: `remove` et `removeAll`
- Appartenance: `contains` et `containsAll`
- Parcours: `iterator`

Théorique

- Union: `add` et `addAll`
- Intersection: `retainAll`
- Différence: `remove` et `removeAll`

Set et List (2)


```
// Collection sans répétition
public interface Set<E> extends Collection<E> {
```

Set et List (2)


```
// Collection sans répétition
public interface Set<E> extends Collection<E> {
    // rien de plus que Collection...
}
```

Set et List (2)


```
// Collection sans répétition
public interface Set<E> extends Collection<E> {
    // rien de plus que Collection...
}

// Collection avec un ordre
public interface List<E> extends Collection<E> {
    void add(int i, E elem) // ajoute à l'indice i
    List<E> subList(int fromIndex, int toIndex)
    int indexOf(Object o)
    int LastIndexOf(Object o)
    E   get(int index)
    ...
}
```

Les collections permettent une interopérabilité!

- `ArrayList`, `LinkedList` et `Vector` implementent `List`
- `TreeSet` et `HashSet` implémentent `Set`

Les collections permettent une interopérabilité!

- `ArrayList`, `LinkedList` et `Vector` implementent `List`
- `TreeSet` et `HashSet` implémentent `Set`

```
MonEntier ent1, ent2, ent3;
...

ArrayList<MonEntier> liste1 = new ArrayList<MonEntier>();
liste1.add(ent1); liste1.add(ent2); liste1.add(ent3);
```

Les collections permettent une interopérabilité!

- `ArrayList`, `LinkedList` et `Vector` implementent `List`
- `TreeSet` et `HashSet` implémentent `Set`

```
MonEntier ent1, ent2, ent3;
...

ArrayList<MonEntier> liste1 = new ArrayList<MonEntier>();
liste1.add(ent1); liste1.add(ent2); liste1.add(ent3);

HashSet<MonEntier> ensemble1 = new HashSet<MonEntier>();
```

Les collections permettent une interopérabilité!

- `ArrayList`, `LinkedList` et `Vector` implementent `List`
- `TreeSet` et `HashSet` implémentent `Set`

```
MonEntier ent1, ent2, ent3;
...

ArrayList<MonEntier> liste1 = new ArrayList<MonEntier>();
liste1.add(ent1); liste1.add(ent2); liste1.add(ent3);

HashSet<MonEntier> ensemble1 = new HashSet<MonEntier>();
ensemble1.addAll(list1);
```

Les collections permettent une interopérabilité!

- ArrayList, LinkedList et Vector implementent List
- TreeSet et HashSet implémentent Set

```
MonEntier ent1, ent2, ent3;
...

ArrayList<MonEntier> liste1 = new ArrayList<MonEntier>();
liste1.add(ent1); liste1.add(ent2); liste1.add(ent3);

HashSet<MonEntier> ensemble1 = new HashSet<MonEntier>();
ensemble1.addAll(list1);
ensemble1.addAll(list1); // ne fait rien
list1.addAll(ensemble1); //dedouble tous les éléments
```

Utiliser les Collections – Importer


```
public class MaCelluleDEntier {
    private MonEntier valeur; private MaCelluleDEntier suiv
    ...
}
public class MaListeDentier {
    MaCelluleDEntier tete;
    ajoute(MonEntier i) { ... }
```

Utiliser les Collections – Importer

```
public class MaCelluleDEntier {
    private MonEntier valeur; private MaCelluleDEntier suiv
    ...
}
public class MaListeDentier {
    MaCelluleDEntier tete;
    ajoute(MonEntier i) { ... }
    public void ajouteCol(Collection<MonEntier> c) {
        for(MonEntier i: c)
            ajoute(i);
    }
}
//Dans un main
ArrayList<MonEntier> arraylist = ...;
... // ajout d'éléments dans arraylist
MaListeDentier l = new MaListeDentier();
l.ajouteCol(arraylist);
```

```
public class MaCelluleDEntier {
    private MonEntier valeur;
    private MaCelluleDEntier suivant;
    ...
}
public class MaListeDentier {
    MaCelluleDEntier tete;
    ajoute(MonEntier i) { ... }

    public void ajouteALaCol(Collection<MonEntier> c) {
        courante= tete;
        while (courante != null) {
            c.add(courante.getValeur());
            courante= courante.getSuivante();
        }
    }
}
```

Implémenter une Collection: AbstractCollection<E>


```
abstract class AbstractCollection<E>
implements Collection<E> {
    ...
    // Obligatoire
    abstract Iterator<E>  iterator();
    abstract int size();

    // "Optionnel"; pour faire fonctionner add et addAll
    boolean add(E e) throws UnsupportedOperationException

}
```

Pour que `remove` (et `removeAll`, etc.) fonctionnent, il faut que la méthode du même nom existe dans l'itérateur renvoyée par `iterator()`.
(Une autre solution serait de ré-implémenter directement ces méthodes.)

AbstractCollection utilise equals


```
public class MonEntier {
    int valeur;
    @Override public boolean equals(Object obj) {...}
}
public class MaCelluleDEntier {...}

public class MaListeDentier
  extends AbstractCollection<MonEntier> {...}
```

AbstractCollection utilise equals


```
public class MonEntier {
    int valeur;
    @Override public boolean equals(Object obj) {...}
}
public class MaCelluleDEntier {...}

public class MaListeDentier
  extends AbstractCollection<MonEntier> {...}

public static void main(String args) {
    MaListeDentier  liste = new MaListeDentier();
    MonEntier i = new MonEntier(); i.valeur = 10;
    MonEntier j = new MonEntier(); j.valeur = 10;
```

AbstractCollection utilise equals


```
public class MonEntier {
    int valeur;
    @Override public boolean equals(Object obj) {...}
}
public class MaCelluleDEntier {...}

public class MaListeDentier
  extends AbstractCollection<MonEntier> {...}

public static void main(String args) {
    MaListeDentier  liste = new MaListeDentier();
    MonEntier i = new MonEntier(); i.valeur = 10;
    MonEntier j = new MonEntier(); j.valeur = 10;

    liste.add(i);
    System.out.println(liste.contains(j)); //true
}
```

Outline

Principe de la généricité

- Permet de paramétrer une classe ou une méthode par un type: une Liste de `X`, un arbre de `X`, etc...
- Permet de créer une structure de donnée (Liste, Arbre, etc.) indépendamment des valeurs qu'elle porte (`X` qui peut être `Integer`, `String`, etc.)

```
public class MaClass<X,Y,Z,...> {
    X attribut1;
    Y attribut2;
    Z attribut3;
}
```

Premier exemple: Couple générique


```
public class Couple<E> {
    E un; E deux;
    Couple (E un, E deux) { this.un=un; this.deux=deux; }
```

Premier exemple: Couple générique


```
public class Couple<E> {
    E un; E deux;
    Couple (E un, E deux) { this.un=un; this.deux=deux; }

    public String toString() {
        return ( "(" +  un.toString()  + ", "
                     + deux.toString() + ")" ); }
```

Premier exemple: Couple générique


```
public class Couple<E> {
    E un; E deux;
    Couple (E un, E deux) { this.un=un; this.deux=deux; }

    public String toString() {
        return ( "(" +  un.toString()  + ", "
                     + deux.toString() + ")" ); }

    public boolean estRedondant() {
        return (un.equals(deux)); }
```

Premier exemple: Couple générique


```
public class Couple<E> {
    E un; E deux;
    Couple (E un, E deux) { this.un=un; this.deux=deux; }

    public String toString() {
        return ( "(" +  un.toString()  + ", "
                     + deux.toString() + ")" ); }

    public boolean estRedondant() {
        return (un.equals(deux)); }

    public static void main (String args) {
        Couple<String> c =
            new Couple<String> ("Hello", "World");
```

Premier exemple: Couple générique


```
public class Couple<E> {
    E un; E deux;
    Couple (E un, E deux) { this.un=un; this.deux=deux; }

    public String toString() {
        return ( "(" +  un.toString()  + ", "
                     + deux.toString() + ")" ); }

    public boolean estRedondant() {
        return (un.equals(deux)); }

    public static void main (String args) {
        Couple<String> c =
            new Couple<String> ("Hello", "World");

        System.out.println(c); // (Hello, World)
        System.out.println(c.estRedondant()); // false
}   }
```

Deuxième exemple: Pile générique


```
public class Pile<E> {
    private static class Cellule<F> {
        private F val;
        private Cellule<F> suivant;
```

Deuxième exemple: Pile générique


```
public class Pile<E> {
    private static class Cellule<F> {
        private F val;
        private Cellule<F> suivant;
        Cellule(F el, Cellule<F> c) {val=el; suivant=c;}
        public F getVal() {return val;}
        public Cellule<F>  getSuivant() {return suivant;}
    }
```

Deuxième exemple: Pile générique


```
public class Pile<E> {
    private static class Cellule<F> {
        private F val;
        private Cellule<F> suivant;
        Cellule(F el, Cellule<F> c) {val=el; suivant=c;}
        public F getVal() {return val;}
        public Cellule<F>  getSuivant() {return suivant;}
    }
    Cellule<E> hautDePile;
```

Deuxième exemple: Pile générique


```
public class Pile<E> {
    private static class Cellule<F> {
        private F val;
        private Cellule<F> suivant;
        Cellule(F el, Cellule<F> c) {val=el; suivant=c;}
        public F getVal() {return val;}
        public Cellule<F>  getSuivant() {return suivant;}
    }
    Cellule<E> hautDePile;
    public void add(E elem) {
        hautDePile = new Cellule<E>(elem, hautDePile); }
```

Deuxième exemple: Pile générique


```
public class Pile<E> {
    private static class Cellule<F> {
        private F val;
        private Cellule<F> suivant;
        Cellule(F el, Cellule<F> c) {val=el; suivant=c;}
        public F getVal() {return val;}
        public Cellule<F>  getSuivant() {return suivant;}
    }
    Cellule<E> hautDePile;
    public void add(E elem) {
        hautDePile = new Cellule<E>(elem, hautDePile); }
    public E depiler() {
        if (hautDePile ==  null) { return null; }
        E el= hautDePile.getValeur();
        hautDePile = hautDePile.getSuivant();
        return el;
    }
}
```

Deuxième exemple: Pile générique (2)


```
public static void main (String args) {
    Pile<String> pile =  new Pile<String>();
    pile.add("Hello");
    pile.add("World");
    System.out.println(pile.depiler()) // World;
    System.out.println(pile.depiler()) // Hello;
    System.out.println(pile.depiler()) // null;

    Pile<Integer> pile2 =  new Pile<Integer>();
    pile2.add(10)
    pile2.add(20)
    int i = pile2.depiler()+pile2.depiler(); // vaut 30
}
```

Deuxième exemple: `Pile` générique itérable


```
public class Pile<E> implements Iterable<E> {
    private static class Cellule<F> {
        private F val; private Cellule<F> suivant;
        ...
    }
    Cellule<E> hautDePile;
    ...
```

Deuxième exemple: `Pile` générique itérable


```
public class Pile<E> implements Iterable<E> {
    private static class Cellule<F> {
        private F val; private Cellule<F> suivant;
        ...
    }
    Cellule<E> hautDePile;
    ...
    public Iterator<E> iterator() {
```

Deuxième exemple: `Pile` générique itérable


```
public class Pile<E> implements Iterable<E> {
    private static class Cellule<F> {
        private F val; private Cellule<F> suivant;
        ...
    }
    Cellule<E> hautDePile;
    ...
    public Iterator<E> iterator() { return(new Iterator<E>(){
        Cellule<E> courant= hautDePile;
        public E next() { E el= courant.getVal();
                          courant= courant.getSuivant();
                          return el; }
        public boolean hasNext() { return(courant!=null); }
```

Deuxième exemple: Pile générique itérable


```
public class Pile<E> implements Iterable<E> {
    private static class Cellule<F> {
        private F val; private Cellule<F> suivant;
        ...
    }
    Cellule<E> hautDePile;
    ...
    public Iterator<E> iterator() { return(new Iterator<E>(){
        Cellule<E> courant= hautDePile;
        public E next() { E el= courant.getVal();
                          courant= courant.getSuivant();
                          return el; }
        public boolean hasNext() { return(courant!=null); }
        public void remove()
            { throw new UnsupportedOperationException(); }
    });}
}
```

Deuxième exemple: `Pile` générique itérable (2)


```
public static void main (String args) {
    Pile<String> pile =  new Pile<String>();
    pile.add("World");
    pile.add("Little");
    pile.add("Hello");
    for (String s: pile)
        System.out.print(s); // HelloLittleWorld
```

Deuxième exemple: Pile générique itérable (2)


```
public static void main (String args) {
    Pile<String> pile =  new Pile<String>();
    pile.add("World");
    pile.add("Little");
    pile.add("Hello");
    for (String s: pile)
        System.out.print(s); // HelloLittleWorld

    Pile<Integer> pile2 =  new Pile<Integer>();
    pile2.add(7);
    pile2.add(3);
    pile2.add(3);
    pile2.add(1);
    for (int i: pile2)
        System.out.print(i); // 1337
}
```

Deuxième exemple: `Pile-Collection` générique


```
public class Pile<E> extends AbstractList<E> {
    private static class Cellule<F> {
        private F val; private Cellule<F> suivant;
        ...
    }
    Cellule<E> hautDePile;
    ...
```

Deuxième exemple: Pile-Collection générique


```
public class Pile<E> extends AbstractList<E> {
    private static class Cellule<F> {
        private F val; private Cellule<F> suivant;
        ...
    }
    Cellule<E> hautDePile;
    ...
    public void add(E elem) {...}        //deja fait
```

Deuxième exemple: Pile-Collection générique


```
public class Pile<E> extends AbstractList<E> {
    private static class Cellule<F> {
        private F val; private Cellule<F> suivant;
        ...
    }
    Cellule<E> hautDePile;
    ...
    public void add(E elem) {...}          //deja fait
    public Iterator<E> iterator() {...} //deja fait
```

Deuxième exemple: Pile-Collection générique


```
public class Pile<E> extends AbstractList<E> {
    private static class Cellule<F> {
        private F val; private Cellule<F> suivant;
        ...
    }
    Cellule<E> hautDePile;
    ...
    public void add(E elem) {...}          //deja fait
    public Iterator<E> iterator() {...} //deja fait
    public int size() { ... } // a faire
}
```

Les éléments de Pile ne peuvent pas être retirés à l'aide des méthodes `remove`, `removeAll`, etc (qui vont renvoyer `UnsupportedOperationException`).
Ceci est en effet cohérent avec une `Pile`, dont on ne peut retirer que le sommet (avec `depiler`).

Généricité avec contrainte


```
MaClasse<E extends MonInterface>
ou
MaClasse<E extends MonAutreClasse>
```

- `MonInterface` ou `MonAutreClasse` définissent *ce dont on a besoin* pour fournir les services proposés par `MaClasse`.
- Permet de rester le plus général possible.

Généricité avec contrainte


```
MaClasse<E extends MonInterface>
ou
MaClasse<E extends MonAutreClasse>
```

- `MonInterface` ou `MonAutreClasse` définissent *ce dont on a besoin* pour fournir les services proposés par `MaClasse`.
- Permet de rester le plus général possible.

Exemple: on veut définir une classe `Max<E>` qui calcule le maximum d'une `Collection<E>`, que doit-on être capable de faire?

Généricité avec contrainte


```
MaClasse<E extends MonInterface>
ou
MaClasse<E extends MonAutreClasse>
```

- `MonInterface` ou `MonAutreClasse` définissent *ce dont on a besoin* pour fournir les services proposés par `MaClasse`.
- Permet de rester le plus général possible.

Exemple: on veut définir une classe `Max<E>` qui calcule le maximum d'une `Collection<E>`, que doit-on être capable de faire? Comparer deux instances de `E`!

Comparable


```
public interface Comparable<T> {
    public int compareTo(T obj);
      // Compare this avec obj, renvoie entier
      //   - negatif  si this < T
      //   - nul      si this et T sont egaux
      //   - positif  si this > T
}
```

Si une classe `C` implémente `Comparable<T>`, toute instance de `C` est *comparable* avec toute instance de `T`: on peut déterminer laquelle des deux est la plus grande.
Usuellement `C` implémente `Comparable<C>`: les instances de `C` sont ordonnées.

```
public class Max<E extends Comparable<E>> {
    E resultat;
    Max (Collection<E> collec) {
        for (E e: collec)
            if ( (resultat == null)
                  || (e.compareTo(resultat) > 0) ) {
                resultat = e;
            }
    }
}
```

Max<E> (2)


```
public static void main(String args) {
    ArrayList<Integer> collec = new ArrayList<Integer>();
    collec.add(0);
    collec.add(54);
    collec.add(1337);
    collec.add(2);
    Max<Integer> max = new Max<Integer>(collec);
    System.out.println(max.resultat); // 1337

    HashSet<String> collec2 = new HashSet<String>();
    collec2.add("0");
    collec2.add("54");
    collec2.add("1337");
    collec2.add("2");
    Max<String> max2 = new Max<String>(collec2);
    System.out.println(max2.resultat); // 54
}
```