Université Paris Cité

Compléments en Programmation Orientée Objet
TP/TD nº 7 - Collections, Généricité et *wildcards*
(Correction)

1 Généricité et *wildcards*

Exercice 1 : Paires

Qui n'a jamais voulu renvoyer deux objets différents avec la même fonction ?

1. Implémenter une classe (doublement) générique `Paire<X,Y>` qui a deux attributs publics `gauche` et `droite`, leurs getteurs et setteurs respectifs et un constructeur, prenant un paramètre pour chaque attribut.

Correction :

```java
public class Paire<X, Y> {
  public X gauche;
  public Y droite;
  public Paire(X gauche, Y droite) {
    this.gauche = gauche;
    this.droite = droite;
  }
  public X getGauche() {
    return gauche;
  }
  public Y getDroite() {
    return droite;
  }
  public void setGauche(X gauche) {
    this.gauche = gauche;
  }
  public void setDroite(Y droite) {
    this.droite = droite;
  }
}
```

Remarque : cette classe ne correspond pas aux pratiques habituelles. En général, les attributs seraient privés. Dans de rares cas (tuple utilisé localement, en privé), on simplifie en utilisant des attributs publics, mais dans ce cas on ne mettrait pas de getteurs et setteurs.
Ici, la classe `Paire<X, Y>` est programmée ainsi dans le seul but de pouvoir répondre aux questions d'après.

2. Application : programmez une méthode

```java
static <U extends Number, V extends Number> Paire<Double, Double> somme(List<Paire<U, V>> aSommer);
```

qui retourne une paire dont l'élément gauche est la somme des éléments gauches de `aSommer` et l'élément droit la somme de ses éléments droits (pour une raison technique, le résultat est typé `Paire<Double, Double>`, mais quelle est cette raison ?).

Correction :

```java
static <U extends Number, V extends Number> Paire<Double, Double> somme(List<Paire<U, V>>
    aSommer) {
  double sommeGauche = 0, sommeDroite = 0;
  for (Paire<U, V> p : aSommer) {
    sommeGauche += p.gauche.doubleValue();
    sommeDroite += p.droite.doubleValue();
  }
  return new Paire<>(sommeGauche, sommeDroite);
```

```
8 }
```

Le type de retour est `Paire<Double, Double>` au lieu de `Paire<U, V>` pour la raison suivante :

- soit on décide de faire les calculs intermédiaires dans des acculumateurs de type `U` et `V`, mais à ce moment-là, on ne sait pas comment initialiser les accumulateurs : via une méthode non statique, ce n'est pas possible car on ne connait a priori pas d'instance de `U` et `V` ; via une méthode statique (de `Number` ou de ses sous-classes), ce n'est pas possible car on ne peut pas savoir quelle méthode choisir : ce sont des méthodes différentes pour générer des valeurs chaque sous-type de `Number`, qu'il faudrait donc choisir à l'exécution en vérifiant une condition sur `U` et `V`, ce qui n'est pas possible à cause de l'effacement de type.
- soit on fait les calculs intermédiaires avec un type primitif (double par exemple), mais on est coincé quand on doit retransformer le résultat en `U` et `V` (pour les mêmes raisons que ci-dessus). On aurait pu essayer de se reposer sur l'autoboxing, mais ça ne marche que si le compilateur peut connaitre le type cible exact (ce n'est pas le cas vu que le type n'est qu'une variable).

Une façon de contourner serait de restreindre la m"thode aux listes non vides. À ce moment-là, il suffit d'initialiser les accumulateurs avec le premier élément.
Une autre façon consisterait à ajouter deux paramètres de types `U` et `V` destinés à recevoir les zéros des deux types et initialiser ainsi les deux accumulateurs.

3. – Écrivez la déclaration d'une variable à laquelle on peut affecter toute paire de nombres de type `Paire<Number, Number>` (contenant donc des instances de `Number` où d'un de ses sous-types).

   Correction : `Paire<Number, Number> p;` (en effet, on peut ensuite faire par exemple `p = new Paire<>(); p.setGauche(new Long(5)); p.setDroite(new Double(12.))`).

   – Écrivez la déclaration d'une variable à laquelle on peut affecter toute paire du type `Paire<M, N>` où `M <: Number` et `N <: Number`.

   Correction : `Paire<? extends Number, ? extends Number> p;`

   – Expliquez la différence entre les deux déclarations précédentes.

   Correction : La différence : dans le premier cas, une fois qu'on affecte un objet à `p`, cet objet peut se voir affecter à sa gauche ou à sa droite n'importe quelle instance de `Number`.
   Dans le deuxième cas, on peut affecter à `p` un objet paire dont les attributs gauche et droite ne peuvent contenir que des instances de 2 sous-types de `Number`, choisis lors de l'instanciation de cette paire.
   Pire, si on veut modifier cet objet via la variable `p`, on ne pourra jamais qu'affecter la valeur `null` à ses attributs : `null` est la seule valeur qui est garantie appartenir à tous les sous-types de `Number`. L'accès en lecture, lui, fonctionnera bien (on sait qu'on récupère des instances, directes ou indirectes, de jjNumber).

4. – Si on écrit `Paire<? extends Number, ? extends Number> p1 = new Paire<Integer, Integer>(15, 12)`, quelles méthodes de la classe `Paire` seront inutiles, appelées sur l'expression `p` ? Lesquelles seront utiles ? (discutez sur les signatures)

> **Correction :** Méthodes inutiles : les setteurs. En effet, on ne pourra que leur passer la valeur `null` (voir corrigé question précédente). De façon générale, toute méthode prenant `U` ou `V` en paramètre aura ce problème (utilisation en position contravariante, alors que les paramètres sont bornés par le haut).
> Méthodes utiles : les getteurs (et toute méthode retournant `U` ou `V`). En effet, on sait qu'ils retournent des instances d'un sous-type de `Number`... donc des instances de Number.

– Si on écrit `Paire<? super Integer, ? super Integer> p2 = new Paire<Number, Number>(15, 12)`, quelles méthodes de la classe `Paire` seront inutiles, appelées sur l'expression `p` ? Lesquelles seront utiles ?

> **Correction :** Méthodes inutiles : les getteurs. En effet, on aura aucune information sur leur type de retour (le seul type contenant tous les supertypes de `Integer` est `Object`). De façon générale, toute méthode retournant `U` ou `V` aura ce problème (utilisation en position covariante, alors que les paramètres sont bornés par le bas).
> Méthodes utiles : les setteurs (et toute méthode prenant `U` ou `V` en paramètre). En effet, on sait qu'ils prennent des instances d'un supertype de `Integer` (même si on ne sait pas lequel), donc, en particulier, toute instance de `Integer` est acceptée.

– Dans les 2 cas précédents, peut-on, sans `cast`, accéder aux attributs de `p1` ou `p2` en lecture (essayez de copier leurs valeurs dans une variable déclarée avec un type de nombre quelconque) ? et en écriture (essayez de leur affecter une valeur autre que `null`) ?

> **Correction :** Du point de vue des vérifications de type, les accès aux attributs en lecture se comportent comme les getteurs, et ceux en écriture comme les setteurs. Ainsi, dans le premier cas (extends), seuls les accès en lecture fonctionnent bien, alors que dans le second cas, ce sont les accès en écriture (super).

– Du coup, supposons qu'on écrive une version immuable de `Paire` (ou n'importe quelle classe générique immuable), et qu'on veuille en affecter une instance à une variable (`Paire<XXX, XXX> p = new Paire<A, B>();`). Pour que cette variable soit utile, doit-elle plutôt être déclarée avec un type comme celui de `p1` ou comme celui de `p2` ?

> **Correction :** Les accès à un objet de type immuable se font en lecture seule. Or dans le cas de la paire, les paramètres de type servent à typer le contenu de la paire, donc ils ne vont apparaître qu'en position covariante (retour de méthode). Donc une déclaration utile pour une variable générique de paire immuable utilisera des wildcards bornés par le haut (mot-clé `extends`), comme `p1`.

Exercice 2 : Une classe générique simple, les « optionnels »

Quand une fonction peut renvoyer soit quelque chose de type `T` soit rien, permettre de retourner `null` pour « rien » peut provoquer des erreurs. On voudrait plutôt retourner un type ayant une instance réservée pour la valeur « rien » (les autres instances encapsulant une « vraie » valeur). C'est ce qu'on propose avec la classe générique `Optionnel<T>`[1], à programmer dans l'exercice.

1. Programmez une telle classe. Cette classe aura un unique attribut de type `T`, sa nullité sera considéré comme une valeur « vide ». Mettez-y un constructeur et les méthodes suivantes :
   – `boolean estVide()` : retourne `true` si l'objet ne contient pas d'élément, `false` sinon.
   – `T get()` : retourne l'élément, lance `NoSuchElementException` (package `java.util`) si l'optionnel est vide.

1. L'API de java propose justement `Optional<T>` à cet effet.

- `T ouSinon(T sinon)` : retourne l'élément s'il existe, `sinon` sinon.

2. Toilettage : Ajoutez des fabriques statiques `Optionnel<T> de(T elt)` (pour `elt` non `null`, sinon on lance `IllegalArgumenException`) et `Optionnel<T> vide()` qui retourne un objet « vide », puis rendez le(s) constructeur(s) privé(s).

Correction :

```java
public class Optionnel<T>{
    private final T val;

    private Optionnel(T val){
        this.val = val;
    }

    public static<T> Optionnel<T> de(T elt){
        if (elt == null)
            throw new IllegalArgumentException();
        return new Optionnel<T>(elt);
    }

    public static<T> Optionnel<T> vide(){
        return new Optionnel<T>(null);
    }

    public boolean estVide(){
        return val == null;
    }

    public T get(){
        if (val == null)
            throw new NoSuchElementException();
        return val;
    }

    T ouSinon (T sinon){
        return val != null ? val : sinon;
    }
}
```

3. Amélioration plus difficile : Afin d'optimiser, faites en sorte que `Optionnel<T> vide()` retourne toujours la même instance `VIDE` : Il faudra créer `VIDE` sans paramètre générique : `private static Optionnel VIDE = new Optionnel<>(null);` et faire un cast approprié dans le code de `vide()`. Vous pourrez ensuite supprimer les warnings de `javac` en mettant `@SuppressWarnings("unchecked")` avant la méthode et `@SuppressWarnings("rawtypes")` devant la déclaration de `VIDE`.

Correction :

```java
//Ce qui change uniquement
    @SuppressWarnings("rawtypes")
    private static final Optionnel VIDE = new Optionnel<>(null);

   @SuppressWarnings("unchecked")
    public static<T> Optionnel<T> vide(){
        return (Optionnel<T>)VIDE;
    }
```

4. Application : écrivez et testez une méthode qui cherche le premier entier pair d'une liste d'entiers et retourne un optionnel le contenant, si elle le trouve, ou l'optionnel vide sinon.

Correction :

```java
public static Optionnel<Integer> cherchePremierPair(List<Integer> liste){
```

```
    for(Integer i : liste){
        if(i != null && i %2 == 0)
               return Optionnel.de(i);
    }
    return  Optionnel.vide();
}
```

Exercice 3 :

Soit le code suivant :

```
class Base { }
class Derive extends Base { }
class G<T extends Base, U> { public T a; public U b; }
```

Ci-dessous, plusieurs spécialisations du type `G`.

1. Certaines ne peuvent exister, dites lesquelles.
2. Des conversions sont autorisées entre les types restants. Quelles sont-elles ? Donnez-les sous forme d'un diagramme.

Voici les types :

1. `G<Object, Object>`
2. `G<Object, Base>`
3. `G<Base, Object>`
4. `G<Derive, Object>`
5. `G<? extends Object, ? extends Object>`
6. `G<? extends Object, ? extends Base>`
7. `G<?, ?>`
8. `G<? extends Derive, ? extends Object>`
9. `G<? extends Base, ? extends Object>`
10. `G<? extends Base, ? extends Derive>`
11. `G<? super Object, ? super Object>`
12. `G<? super Object, ? super Base>`
13. `G<? super Base, ? super Object>`
14. `G<? super Base, ? super Derive>`

Correction : `? extends T` : le type argument de la référence étend `T` et `? super T` : le type argument de la référence est étendu par `T`. À partir de là on en déduit les solutions.

1. Les types interdits sont `G<Object, Object>`, `G<Object, Base>`, `G<? super Object, ? super Object>` et `G<? super Object, ? super Base>`
2. – 5 et 3 représentent un même type.
   – Les conversions possibles sont :
   — $1 \to 8 \to 4 \to 3, 5$
   — $8 \to 7 \to 3, 5$
   — $1 \to 9 \to 10 \to 3, 5$
   – Remarque :
   — 9 est le type `G<? super Base, Object>`
   — 4 est le type `G<?, ? extends Base>`
   — 6 est le type `G<? extends Derive, ?>`

2 Utilisation avancée de collections

Le but de cet exercice est d'implémenter un petit système de base de données en mémoire. Dans le model que nous allons adopter :

– Une base de données (`BaseDeDonnees`) contient plusieurs tableaux.

- Chaque tableau (`Tableau`) est définit par son nom et un ensemble de colonnes.
- Une colonne (`Colonne`) Tous les tableaux ont une colonne `"id"` qui permet d'identifier chaque entré/ligne dans le tableau.

Votre implémentation doit être utilisable avec le code suivant :

```java
public class Main {
  public static void main(String[] args) {
    // creation de la base de donnees
    BaseDeDonnees db = new BaseDeDonnees("UFR Informatique");
    // definition de tableau etudiants
    Tableau etudiants = db.ajouterTableau("etudiants");
    etudiants.ajouterColonne("nom", TypeDonnee.STRING);
    etudiants.ajouterColonne("prenom", TypeDonnee.STRING);
    etudiants.ajouterColonne("groupe", TypeDonnee.INT);
    // insertion de donnees
    db.inserer("etudiants", List.of("nom", "prenom", 1));
    db.inserer("etudiants", List.of("Martin", "Marie", 5));
    db.inserer("etudiants", List.of("Laurent", "Jean", 1));
    db.inserer("etudiants", List.of("Simon", "Pierre", 5));
    // recherche de donnees
    List<Ligne> resultats = db.chercher("etudiants", "groupe", 5);
    for (Ligne ligne : resultats) {
      System.out.println(ligne.get("nom") + " " + ligne.get("prenom"));
    }
  }
}
```

Exercice 4 : Implémentation simple

1. Définir `TypeDonnee` qui permet d'identifier les différents types de données (`INT`, `STRING`...) qu'on peut stocker dans la base de données.

   Correction :

   ```java
   public enum TypeDonnee {
       INT,
       STRING
   }
   ```

2. Premièrement on implémentera les entrés/lignes comme des dictionnaires (`Map`) :
   - Implémenter la classe `Ligne` tel que elle encapsule un dictionnaire passé au constructeur.
   - Ajouter une méthode `Object get(String nom)` qui retourne la valeur associé à au nom passé en paramètre.

   Correction :

   ```java
   public class Ligne {
     private final Map<String, Object> valeurs;

     public Ligne(Map<String, Object> valeurs) {
       this.valeurs = valeurs;
     }

     public Object get(String colonne) {
         return valeurs.get(colonne);
     }
   }
   ```

3. Implémenter la classe `Colonne` définit par un nom et un type de donnée `TypeDonnee`.

   Correction :

   ```java
   public class Colonne {
     private final String nom;
   ```

```
  private final TypeDonnee type;

  public Colonne(String nom, TypeDonnee type) {
    this.nom = nom;
    this.type = type;
  }

  public String getNom() {
    return this.nom;
  }

  public TypeDonnee getType() {
    return this.type;
  }
}
```

4. Implémenter la classe `Tableau` :
   - Ajouter une méthode `void ajouterColonne(String nom, TypeDonnee type)` qui ajoute une nouvelle colonne au tableau.
   - Assurer que tous les tableaux ont par défaut une colonne nommé `"id"` de type `TypeDonnee.INT`.

Correction :

```
public class Tableau {
  private final String nom;
  private final ArrayList<Colonne> colonnes;

  public Tableau(String nom) {
    this.nom = nom;
    this.colonnes = new ArrayList<Colonne>();
    this.ajouterColonne("id", TypeDonnee.INT);
  }

  public String getNom() {
    return this.nom;
  }

  public void ajouterColonne(String nom, TypeDonnee type) {
    this.colonnes.add(new Colonne(nom, type));
  }
}
```

5. Créer la classe `BaseDeDonnees` avec les méthodes :
   - `Tableau ajouterTableau(String nom)` qui crée et ajoute un tableau avec le nom donné à la base de données, et elle retourne l'instance de tableau créé.
   - `Tableau getTableau(String nom)` qui retourne le tableau avec nom s'il existe dans la base de donnée, sinon elle retourne `null`.

Correction :

```
public class BaseDeDonnees {
  private final String nom;
  private final ArrayList<Tableau> tableaux;

  public BaseDeDonnees(String nom) {
    this.nom = nom;
    this.tableaux = new ArrayList<Tableau>();
  }

  public String getNom() {
    return nom;
  }

  public Tableau ajouterTableau(String nom) {
    Tableau tableau = new Tableau(nom);
```

```
    this.tableaux.add(tableau);
    return tableau;
  }

  public Tableau getTableau(String nom) {
    for(Tableau tableau : tableaux) {
      if(tableau.getNom().equals(nom)) {
        return tableau;
      }
    }
    return null;
  }
}
```

6. La méthode statique `List.of(...)` permet de créer une liste de type `List<Object>` qui contient les éléments passés en argument.
   Écrire un méthode `Map<String, Object> preparer(List<Object> elements)` dans la classe `Tableau` qui associe à chaque nom de colonne (les colonnes ordonnées par ordre d'insertion) une valeur dans le tableau `elements` passé en arguments. La méthode `preparer` doit aussi associer à la clé `"id"` une valeur unique.

   Correction : Ajouter à la classe `Tableau` :

```
private int conteur = 0;
public Map<String, Object> preparer(List<Object> elements) {
  Map<String, Object> valeurs = new HashMap<String, Object>();
  valeurs.put("id", conteur++);
  for(int i = 1; i  < this.colonnes.size(); i++) {
    valeurs.put(this.colonnes.get(i).getNom(), elements.get(i - 1));
    // elements.get(i - 1) parce que la valeur de la colonne "id" n'est pas donne.
  }
  return valeurs;
}
```

7. Les lignes insérées doivent être stockées au niveau de la classe `BaseDeDonnees` (la classe `Tableau` stocke seulement les informations relatives au tableau).
   Ajouter la méthode `void inserer(String nom_tableau, List<Object> elements)` à `BaseDeDonnees` qui permet de créer une ligne à partir de `elements` (utiliser la fonction `preparer` de la classe `Tableau`) et la stocker à la base de données.

   Correction : Ajouter à la classe `BaseDeDonnees` l'attribut suivante :

```
private final Map<String, List<Ligne>> lignes;
```

   et l'initialise au niveau de constructeur :

```
  this.lignes = new HashMap<String, List<Ligne>>();
```

   finalement, la méthode est définie par :

```
public void inserer(String nom_tableau, List<Object> elements) {
  Tableau tableau = this.getTableau(nom_tableau);
  Ligne ligne = new Ligne(tableau.prepare(elements));
  if(!this.lignes.containsKey(nom_tableau)) {
    this.lignes.put(nom_tableau, new LinkedList<Ligne>());
  }
  this.lignes.get(nom_tableau).add(ligne);
}
```

8. Définir la méthode `List<Ligne> chercher(String nom_tableau, String nom_col, Object valeur)` qui permet de retrouver les lignes dans le tableau dont la valeur de la colonne `nom_col` est égale à `valeur`.

Correction :

```java
public List<Ligne> chercher(String nom_tableau, String nom_col, Object valeur) {
  List<Ligne> resultats = new LinkedList<Ligne>();
  for(Ligne ligne : this.lignes.get(nom_tableau)) {
    if(ligne.get(nom_col).equals(valeur)) {
      resultats.add(ligne);
    }
  }
  return resultats;
}
```

9. Essayer votre implémentation avec le code donné en dessus.

Exercice 5 : Optimisation de choix de collections

La bibliothèque standard de Java vient avec plusieurs implémentation différentes pour les collections `List` (`ArrayList`, `LinkedList`...) et `Map` (`HashMap`, `LinkedHashMap`, `TreeMap`,...). Dans votre implémentation de l'exercice précédent, vous avez utilisé certaines de ces collections. Les tableaux suivants présentent les complexités [2] des certaines méthodes des collections les plus utilisés.

	add	remove	get	contains
ArrayList	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
LinkedList	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$

	get	contains
HashSet	$\mathcal{O}(1)$	$\mathcal{O}(1)$
LinkedHashSet	$\mathcal{O}(1)$	$\mathcal{O}(1)$
TreeSet	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

	get	containsKey
HashMap	$\mathcal{O}(1)$	$\mathcal{O}(1)$
LinkedHashMap	$\mathcal{O}(1)$	$\mathcal{O}(1)$
TreeMap	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

Sachant qu'en pratique :

- La création de nouveaux tableaux n'est pas assez fréquente et la plupart des tableaux sont créés juste après la création de base de données.
- Le nombre des lignes d'un tableau est plus grand que le nombres de ces colonnes.
- Les opérations d'insertions et de recherche sont largement utilisés.

Revisiter votre implémentation et améliorer le choix de collections que vous avez fait. Expliquer vos choix.

Correction : Par rapport à l'implémentation donnée en correction :

- Dans la plupart des cas avec des `Map`, il faut juste utiliser `HashMap` ou `LinkedHashMap` (généralement `LinkedHashMap` est mieux que `HashMap` puisqu'il n'est pas affecté par la capacité des tableaux de collision).
- Si on utilise une `Map` (`HashMap`) au lieu d'une `List` pour stocker les tableaux dans `BaseDeDonnees`, la complexité de `getTableau` devient $\mathcal{O}(1)$ au lieu de $\mathcal{O}(n)$. Ce qui va améliorer les performances des autres méthodes qui l'utilisent (`inserer` et `chercher`).

2. Rappel : $\mathcal{O}(2^n) > \mathcal{O}(n^3) > \mathcal{O}(n^2) > \mathcal{O}(n \log n) > \mathcal{O}(n) > \mathcal{O}(\sqrt{n}) > \mathcal{O}(\log n) > \mathcal{O}(1)$