Université Paris Cité

Compléments en Programmation Orientée Objet
TP/TD n° 7 - Collections, Généricité et *wildcards*

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.
2. Application : programmez une méthode

```
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 ?).
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).
   – Écrivez la déclaration d'une variable à laquelle on peut affecter toute paire du type `Paire<M, N>` où `M <: Number` et `N <: Number`.
   – Expliquez la différence entre les deux déclarations précédentes.
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)
   – 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 ?
   – 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`) ?
   – 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` ?

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.

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

  - `T get()` : retourne l'élément, lance `NoSuchElementException` (package `java.util`) si l'optionnel est vide.
  - `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).
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`.
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.

Exercice 3 :

Soit le code suivant :

```
1 class Base { }
2 class Derive extends Base { }
3 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>`

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

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