TP 3 : Surcharge, Masquage, Réécriture

©2022 Ghiles Ziat
ghiles.ziat@epita.fr

Dans les exercices qui suivent, vous êtes libres de répondre aux questions en utilisant le langage de votre choix parmi Java et C++, sauf lorsqu'il est explicitement demandé d'utiliser un des deux langages ou les deux.

Exercice I : Surcharge & Réécriture

Q1 – Déclarez une classe `Rational` avec deux champs publics : un numérateur `num:int` et un dénominateur `den:int`.

Q2 – Pour pouvoir afficher des rationnels sur la sortie standard :

- (Java) Réécrivez la méthode `toString()`
- (C++) : Surchargez l'opérateur d'insertion pour reconnaître un objet `ostream` à gauche et un objet `Rational` à droite pour que `cout` accepte un objet `Rational` après l'opérateur d'insertion

Q3 – Ajoutez à votre classe une méthode `Rational mul (int coef)` qui crée une nouvelle instance de `Rational` ou le numérateur et le dénominateur de l'instance appelante ont été multipliés par `coef`.

Q4 – On veut à présent trier un tableau de rationnels dans l'ordre croissant. Pour cela on peut utiliser les fonctions de la bibliothèque standard : `Arrays.sort()` en Java et `std::sort()` en C++. Créez un main et triez le tableau $\{\frac{1}{2}, \frac{1}{3}, \frac{1}{1}, \frac{5}{3}, \frac{7}{8}\}$. Que remarquez vous ?

Q5 – Pour pouvoir utiliser ces méthodes, il nous faut spécifier un ordre sur nos rationnels. Comparer deux rationnels est équivalent à les réduire au même dénominateur et comparer les numérateurs. Vous pourrez réutiliser la méthode `mul`.

- (Java) Implémentez l'interface `Comparable<Rational>` et réécrivez la méthode `public int compareTo(Rational that)` pour qu'elle implémente la comparaison décrite ci-dessus.
- (C++) : Surchargez l'opérateur de comparaison `bool operator < (const Rational& that)const` pour pouvoir comparer l'instance appelante avec l'objet passé en paramètre.

Q6 – Vérifiez votre implémentation en affichant le tableau avant et après le tri.

Exercice II : Polymorphisme paramétrique

Le polymorphisme paramétrique est un mécanisme qui permet d'écrire un code de manière générique afin qu'il puisse gérer les valeurs de manière identique sans dépendre de leurs types. Ce concept est implémenté avec les Génériques [1] en Java, et avec les Patrons [2] (communément appelés Template) en C++.

Q1 – Écrivez une méthode `swap`, paramétrée par un type `T`, qui prend un tableau d'éléments de type `T` ainsi que deux indices entiers et qui échange les valeurs à ces deux indices.

Q2 – Écrivez une méthode `int smallest(T[] arr, int start, int end)`, paramétrée par un type `T`, qui prend un tableau d'éléments de type `T` ainsi que deux indices entiers `start` et `end` et qui retourne l'indice de la plus petite valeur entre les deux indices `start` et `end`. En java, en ajoutera la contrainte que le paramètre de type `T` doit étendre l'interface `Comparable` pour pouvoir appeler la méthode `compareTo`.

Q3 – Le tri par sélection [3] consiste à rechercher le plus petit élément du tableau, l'échanger avec l'élément d'indice 0 ; rechercher le plus petit élément du sous-tableau qui commence à l'indice 1, et l'échanger avec l'élément d'indice 1 ; continuer de cette façon jusqu'à ce que le tableau soit entièrement trié. Implémentez un tri par sélection générique en utilisant les deux méthodes définies précédemment.

Q4 – Testez votre fonction et affichez son résultat sur un exemple de tableau.

Exercice III : Surcharge

On veut implémenter un évaluateur d'expression arithmétiques écrites en notation polonaise inverse (NPI). Cette notation permet d'écrire de façon non ambiguë les formules arithmétiques sans utiliser de parenthèses. Par exemple, l'expression $(3 + 2) * 4$ peut s'écrire en NPI sous la forme $2\ 3 + 4\ *$. Les évaluateurs NPI utilisent une pile, où les opérandes sont disposées au sommet de la pile et dépilées au fur et à mesure que les opérateurs s'appliquent.

Q1 – Créez une classe `Pol` avec comme attribut une pile d'entiers [4].

Q2 – Ajoutez à votre classe une méthode `push(int x)` qui place un entier au sommet de la pile.

Q3 – Ajoutez à votre classe une méthode `push(String op)` qui dépile le bon nombre d'argument correspondant à l'opérateur `op`, applique l'opération à ces argument et rempile le résultat.

Q4 – Créez un `main` ou vous testerez que l'évaluation de l'expression $2\ 3\ +\ 4\ *$ donnera bien 20.

Q5 – Lors du calcul $(2 * 3 + 4)/(15 - (2 * 3 + 4))$ on peut remarquer que l'expression $(2 * 3 + 4)$

1. https://fr.wikibooks.org/wiki/Programmation_Java/Types_g%C3%A9n%C3%A9riques
2. https://fr.wikipedia.org/wiki/Template_(programmation)
3. https://fr.wikipedia.org/wiki/Tri_par_s%C3%A9lection
4. On pourra utiliser la classe `java.util.Stack` en java, et la class `stack` en c++ (sans oublier d'inclure le *header* `#include <stack>`)

apparaît deux fois. En NPI, ce calcul peut s'écrire : 2 3 ∗ 4 + 15 2 3 ∗ 4 + − /. Cependant, si on ajoute à notre jeu d'instructions deux opérateurs spéciaux ":" qui permet de dupliquer la tête de la pile, et "<>" qui permet d'intervertir les deux premières valeurs de la pile, on peut faire un calcul plus court avec : 2 3 ∗ 4 + : 15 <> − /. Ajoutez à votre implémentation ces deux opérateurs.

Q6 – Vérifiez que les deux calculs

— 2 3 ∗ 4 + 15 2 3 ∗ 4 + − /

— 2 3 ∗ 4 + : 15 <> − /

donnent bien 2.

Exercice IV : Polymorphisme paramétrique (Bonus, Java avancé)

Q1 – **Memoïsation.** La mémoïsation est la mise en cache des valeurs de retour d'une fonction selon ses valeurs d'entrée. Ce type de cache est habituellement implémenté en utilisant une table de hachage.

Q2 – Créez une classe `Memo`, dotée d'un attribut `HashMap<In, Out> cache` et d'une méthode privée `private Function<In, Out> doMemoize(Function<In, Out> function)` qui prend une fonction `f` en paramètre (voir la classe Function [5]) et qui retourne une lambda expression ayant le même comportement que `f` mais qui met en cache les paires entrées-sorties de façon à ne pas refaire les calculs qui ont déjà été faits pour une entrée.

Q3 – Ajoutez à la classe `Memo` une méthode
`<T, U> Function<T, U> memoize(Function<T, U> function)` capable mémoïser n'importe quelle fonction/méthode. On s'assurera qu'une nouvelle table de hachage est créée à chaque appel (en fabriquant une nouvelle instance de la classe `Memo` par exemple.)

Q4 – la classe `Calculation` ci-dessous simule un long calcul (méthode `slow`). Dans une classe `Main`, appelez la méthode `slow` et affichez son temps d'exécution. Puis, fabriquez une version mémoïsée de cette fonction et mesurez les temps d'exécution de celle-ci sur la même entrée deux fois de suite.

```
1 class Calculation{
2     Integer slow(Integer x) {
3         try {
4             Thread.sleep(2_000); // simulates a long calculation
5         } catch (InterruptedException ignored) {}
6         return 2*x;
7     }
8 }
```

5. https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html