TP 3 : Overloading, Masking, Overriding

©2022 Ghiles Ziat
ghiles.ziat@epita.fr

In the following exercises, you are free to answer the questions using the language of your choice among Java and C++, except when it is explicitly requested to use one of the two languages or both.

Exercice I : Overload & Rewrite

Q1 – Declare a class `Rational` with two fields a numerator `num:int` and a denominator `den:int`.

Q2 – To be able to display rationals on the standard output :

- (Java) Rewrite the `toString()` method
- (C++) : Override the insertion operator to recognize a object `ostream` on the left and an object `Rational` on the right so that `cout` accepts a `Rational` object after insertion operator

Q3 – Add to your class a method `Rational mul (int coef)` which creates a new instance of `Rational` or the numerator and denominator of the calling instance have been multiplied by `coef`.

Q4 – We now want to sort an array of rational numbers in order increasing. For this you can use the functions of the library standard : `Arrays.sort()` in Java and `std::sort()` in C++. Create a main and sort the array $\{\frac{1}{2}, \frac{1}{3}, \frac{1}{1}, \frac{5}{3}, \frac{7}{8}\}$. What do you notice ?

Q5 – To be able to use these methods, we need to specify a order on our rationales. Comparing two rationals is equivalent to reduce them to the same denominator and compare the numerators. You you can reuse the `mul` method.

- (Java) Implement the interface `Comparable<Rational>` and rewrite the method `public int compareTo(Rational that)` so that it implements the comparison described above.
- (C++) : Overload the comparison operator `bool operator < (const Rational& that)const` to be able to compare the calling instance with the object passed as a parameter.

Q6 – Check your implementation by printing the array before and after sorting.

Exercice II : Parametric polymorphism

Parametric polymorphism is a mechanism for writing a code generically so that it can handle the values of identical way without depending on their types. This concept is implemented with Generics[1] in Java, and with Patterns[2] (commonly called Template) in C++.

Q1 – Write a method `swap`, parameterized by a type `T`, which takes an array of elements of type `T` as well as two integer indices and which exchanges the values at these two clues.

Q2 – Write a method `int smallest(T[] arr, int start, int end)`, parameterized by a type `T`, which takes an array of elements of type `T` as well as two integer indices `start` and `end` and which returns the index of the smallest value between the two indices `start` and `end`. In java, will add the constraint that the `T` type parameter must extend the interface `Comparable` to be able to call the `compareTo` method.

Q3 – Selection sort[3] consist in finding the smallest element of an array, swap it with the element of index 0 ; find the smallest element of the subarray that starts at index 1, and swap it with the element with index 1 ; continue in this way until the array is fully sorted. Implement a generic selection sort by using the two methods defined above.

Q4 – Test your function and display its result on an example.

Exercice III : Overload

We want to implement an evaluator of written arithmetic expressions in Reverse Polish Notation (NPI). This notation allows to write unambiguously the arithmetic formulas without using parentheses. For example, the expression $(3 + 2) * 4$ can be written as NPI in the form $23 + 4*$. NPI evaluators use a stack, where the operands laid out at the top of the stack and popped as they go as operators apply.

Q1 – Create a class `Pol` with a stack as an attribute `s` integers.

Q2 – Add to the class `Pol` a method `push(int x)` which places an integer on top of the stack.

Q3 – Add to the class `Pol` a method `push(String op)` which pops the right number of arguments corresponding to the operator `op`, applies the operation to these arguments and stacks the result.

Q4 – Create a `main` where you will test that the evaluation of the expression $2\ 3\ +\ 4\ *$ will give 20.

Q5 – When calculating $(2*3+4)/(15-(2*3+4))$ we can notice that the expression $(2*3+4)$ appears twice. In NPI, this calculation can be written : $2\ 3\ *\ 4\ +\ 15\ 2\ 3\ *\ 4\ +\ -\ /$. However, if we add to our instruction set two special operators ":" which allows to duplicate the head of the stack, and "<>" which allows to invert the first two values of the stack, we can do a shorter calculation with : $2\ 3\ *\ 4 +\ :\ 15\ <>\ -\ /$. Add to your implementation these two operators.

Q6 – Verify that both calculations

1. `https://fr.wikibooks.org/wiki/Java_Programming/Types_g%C3%A9n%C3%A9riques`
2. `https://en.wikipedia.org/wiki/Template_(programming)`
3. `https://en.wikipedia.org/wiki/Selection_sort`

— 2 3 $*$ 4 + 15 2 3 $*$ 4 + − /

— 2 3 $*$ 4 + : 15 <> − /

give 2.

Exercice IV : Parametric polymorphism (Bonus, advanced Java)

Q1 – **Memoization.** Memoization is the caching of return values from a function according to its input values. This type of cache is usually implemented using a hash table.

Q2 – Create a class `Memo`, with an attribute `HashMap<In, Out> cache` and a private method `private Function<In, Out> doMemoize(Function<In, Out> function)` which takes a function `f` as a parameter (see the class Function[4]) and which returns a lambda expression with the same behavior as `f` but which caches the input-output pairs of so as not to redo the calculations that have already been made for a hall.

Q3 – Add to the class `Memo` a method `<T, U> Function<T, U> memoize(Function<T, U> function)` able to memoize any function/method. We will make sure that a new hash table is created on each call (in making a new instance of the class `Memo` by example.)

Q4 – the `Calculation` class below simulates a long calculation (`slow` method). In a class `Main` call the `slow` method and display its execution time. Then, make a memoized version of this function and measure the execution time of the latter on the same input twice in a row.

```java
class Calculation{
   Integer slow(Integer x) {
      try {
         Thread.sleep(2_000); // simulates a long calculation
      } catch (InterruptedException ignored) {}
      return 2*x;
   }
}
```

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