Programmazione ad Oggetti
Mod. 2

5/9/2023

Studente _____ Matricola _____

1. Si implementi un metodo statico generico in Java 8+ che, dato un iteratore ed una funzione, produce un nuovo iteratore che in maniera *asincrona* applica la funzione ad ogni elemento dell'iteratore originale. Ciò significa che un thread diverso deve processare ciascun elemento.

   (a) [10 punti] Si implementi tutto ciò che è necessario dello snippet seguente.

   ```
   static <A, B> Iterator<Supplier<B>> asyncIterator(Iterator<A> it, Function<A, B> f) {
       return new Iterator<>() {
           @Override
           public boolean hasNext() { /* da implemetare */ }

           private class Future implements Supplier<B> {
               public Future(Supplier<B> f) { /* da implementare */ }

               /* da completare/implementare */
           }

           @Override
           public Supplier<B> next() { /* da implementare */ }
       };
   }
   ```

   Si badi che la classe innestata `Future` ha lo scopo di semplificare l'implementazione della `next()`. Essa rappresenta una computazione in corso che non è ancora terminata. Essendo di fatto un `Supplier`, sarà possibile conoscere il risultato invocando il metodo `get()`. Naturalmente questo risultato sarà possibile ottenerlo solamente dopo aver atteso che il relativo thread abbia finito di computare il `Supplier` passato in costruzione.

   (b) [7 punti] Si implementi il seguente metodo statico in modo che utilizzi il metodo statico `asyncIterator()` di cui all'esercizio precedente per scorrere l'argomento iterabile e ordinare in maniera asincrona i suoi elementi[1].

   ```
   static <T extends Comparable<? super T>> void sortLists(Iterable<List<T>> c)
   ```

2. Si scriva in C++ (specificando quale revisione del linguaggio si intende adottare) una classe generica `matrix` che rappresenta matrici bidimensionali di valori di tipo `T`, dove `T` è un tipo templatizzato. Tale classe deve comportarsi come un *valore*, implementando lo stile della *value-oriented programming* e della *generic programming*. Inoltre deve aderire al concept denominato *Container* da STL.

   Si implementino dunque:

   (a) [1 punti] costruttore di default;

   (b) [1 punti] costruttore per copia;

   (c) [1 punti] costruttore con 2 parametri + 1 opzionale: numero di righe, numero di colonne e valore iniziale di tipo `T`;

[1] Si ricordi che il JDK fornisce un metodo `Collections.sort()` per ordinare liste di oggetti confrontabili.

(d) [1 punti] distruttore (se necessario);

(e) [2 punti] operatore di assegnamento;

(f) [3 punti] member type `iterator`, `const_iterator`, `value_type`, `reference`, `const_reference` e `pointer`;

(g) [2 punti] operatore di accesso tramite riga e colonna: implementare 2 overload (const e non-const) di `operator()`;

(h) [3 punti] metodi `begin()` ed `end()`: 2 overload (const e non-const) per poter iterare tutta la matrice come un unico container lineare dall'angolo superiore sinistro all'angolo inferiore destro come se fosse piatta;

(i) [2 punti] altri metodi a piacere che si ritengono utili.

L'implementazione può utilizzare STL liberamente.

Si prenda a riferimento il seguente snippet per la specifica dei requisiti del tipo `matrix`:

```
matrix<double> m1;           // non inizializzata
matrix<double> m2(10, 20);   // 10 X 20 inizializzata col default constructor di double
matrix<double> m3(m2);       // costruita per copia
m1 = m2;                     // assegnamento
m3(3, 1) = 11.23;            // operatore di accesso come left-value
for (typename matrix<double>::iterator it = m1.begin(); it != m1.end(); ++it) {
    typename matrix<double>::value_type& x = *it;  // de-reference non-const
    x = m2(0, 2);  // operatore di accesso come right-value
}
matrix<string> ms(5, 4, "ciao"); // 5 X 4 inizializzata col la stringa passata come terzo argomento
for (typename matrix<string>::const_iterator it = ms.begin(); it != ms.end(); ++it)
    cout << *it;   // de-reference const
```

Question:	1	2	Total
Points:	17	16	33
Bonus Points:	0	0	0
Score:			