Programmazione ad Oggetti
Mod. 2

13/1/2023

Studente _____ Matricola _____

1. Si prenda in considerazione la seguente interfaccia Java:

```
public interface Pool<R> {
    R acquire() throws InterruptedException;  // acquisisce una risorsa
    void release(R x);                        // rilascia una risorsa e la rimette nella pool
}
```

Una pool è un container di oggetti che si comporta come una coda bloccante: è possibile ottenere una risorsa con `acquire()` per poi restituirla alla pool tramite il metodo `release()`.

Il generic `R` astrae il tipo della risorsa fornita dalla `acquire()` e restituita alla pool tramite la `release()`.

Quando la coda è vuota e nessun oggetto è disponibile, il metodo `acquire()` deve essere bloccante. Al fine di semplificare l'implementazione, si utilizzi un oggetto di tipo `LinkedBlockingQueue` come struttura dati d'appoggio per la coda bloccante all'interno della pool. La classe `LinkedBlockingQueue` è fornita dal JDK ed i suoi metodi più importanti sono:

```
class LinkedBlockingQueue<E> implements BlockingQueue<E> {
    LinkedBlockingQueue();                  // costruisce una coda bloccante vuota
    void add(E x);                          // aggiunge un elemento alla coda
    E take() throws InterruptedException;   // (bloccante) estrae la testa, se la coda è vuota
                                            //    aspetta che ci sia un elemento
    E peek();                               // (non bloccante) ritorna la testa senza rimuoverla,
                                            //    oppure ritorna null se non c'è nulla
    int size();                             // numero di elementi nella coda
}
```

(a) [5 punti] Si implementi una classe generica `SimplePool` che implementa l'interfaccia `Pool` e realizza una semplice pool usando `LinkedBlockingQueue` internamente come coda bloccante. Naturalmente deve esserci un modo per pre-popolare la pool: si progetti un qualche meccanismo ragionevole per farlo senza fare assunzioni troppo strette sul type parameter.

(b) [7 punti] Si implementi una classe `AutoPool` parametrica su un tipo `T` che implementa l'interfaccia `Pool<Resource<T>>`. Gli oggetti di tipo `T` all'interno della pool vengano messi in una `LinkedBlockingQueue<T>`. Anche qui deve esistere un modo per pre-popolare la pool, come nell'esercizio precedente. Si badi che la pool memorizza al suo interno oggetti di tipo `T` ma fornisce oggetti di tipo `Resource<T>`, che fungono da *proxy* per i veri oggetti di tipo `T`. L'interfaccia generica `Resource` è definita come segue:

```
public interface Resource<T> {
    T get();           // getter dell'oggetto di tipo T
    void autorelease(); // rilascia questo oggetto e lo rimette nella pool da cui proviene
}
```

Se `p` è una `AutoPool<T>`, l'obiettivo è fare in modo che una risorsa `x` di tipo `Resource<T>` acquisita tramite la chiamata `p.acquire()` non debba essere esplicitamente riconsegnata alla pool chiamando `p.release(x)`, ma sia possibile rilasciarla invocando semplicemente `x.autorelease()`.

(c) [5 punti] Si automatizzi il meccanismo di cui sopra facendo in modo che un oggetto di tipo `Resource<T>` si auto-rilasci quando non esistono più riferimenti ad esso.

Suggerimento: la superclasse `Object` definisce un metodo `finalize()` che viene invocato nel momento in cui un oggetto viene cancellato dal garbage collector.

2. Si prenda in considerazione la seguente classe C++14 che rappresenta curve definite tramite funzioni unarie $\mathbb{R} \to \mathbb{R}$ in un certo intervallo di dominio $[a, b] \in \mathbb{R}$.

```
#include <functional>
#include <iostream>
#include <utility>

using namespace std;
using real = double;
using unary_fun = function<real(const real&)>;

#define RESOLUTION (1000)   // risoluzione dell'intervallo [a, b]

class curve
{
private:
    real a, b;
    unary_fun f;
public:
    curve(const real& a_, const real& b_, const unary_fun& f_) : f(f_), a(a_), b(b_) {}
    curve(const real& c) = default;

    real get_dx() const { return (b - a) / RESOLUTION; }
    pair<real, real> interval() const { return pair<real, real>(a, b); }
    real operator()(const real& x) const { return f(x); }

    curve derivative() const { /* DA IMPLEMENTARE */ }
    curve primitive() const { /* DA IMPLEMENTARE */ }
    real integral() const { /* DA IMPLEMENTARE */ }

    class iterator
    {
    private:
        const curve& c;
        real x;
    public:
        iterator(const curve& c_, const real& x_) : c(c_), x(x_) {}
        iterator(const iterator& c) = default;

        pair<real, real> operator*() const { /* DA IMPLEMENTARE */ }
        iterator operator++() { /* DA IMPLEMENTARE */ }
        iterator operator++(int) { /* DA IMPLEMENTARE */ }
        bool operator!=(const iterator& it) const { /* DA IMPLEMENTARE */ }
    };

    iterator begin() const { /* DA IMPLEMENTARE */ }
    iterator end() const { /* DA IMPLEMENTARE */ }
};
```

(a) [4 punti] Si implementi il metodo `derivative()` che calcola la derivata, cioè la curva con una lambda che calcola il rapporto differenziale $\frac{f(x+dx)-f(x)}{dx}$.

(b) [4 punti] Si implementi il metodo `primitive()` che calcola la primitiva, ovvero la curva che rappresenta

l'integrale indefinito con una lambda che calcola il prodotto differenziale $f(x) \cdot dx$.

(c) 3 punti Si implementi il metodo `integral()` che calcola l'integrale definito $\int_a^b f(x)dx$, ovvero la sommatoria dei prodotti differenziali calcolati dalla primitiva nell'intervallo $[a, b]$.

(d) 5 punti Si implementino i metodi relativi all'iteratore tenendo presente che:

- il metodo `begin()` computa un iteratore all'inizio dell'intervallo $[a, b]$;
- il metodo `end()` computa un iteratore oltre la fine dell'intervallo $[a, b]$, includendo l'estremo b nell'iterazione;
- l'operatore di de-reference dell'iteratore produce una coppia di reali che rappresentano un punto della curva sul piano cartesiano;
- gli operatori di pre e post incremento incrementano l'ascissa corrente x della quantità dx.

Il seguente snippet di test può servire da riferimento per capire il comportamento dell'iteratore:

```
curve c(-10., 10., [](const real& x) { return x * x - 2 * x + 1; });
for (curve::iterator it = c.begin(); it != c.end(); ++it)
{
    const pair<real, real>& p = *it;
    const real& x = p.first, & y = p.second;
    cout << "c(" << x << ") = " << y << endl;
}
```

Total for Question 2: 16

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