Programmazione ad Oggetti mod. 2

22/1/2019

Studente _____ Matricola _____

1. Si implementi il seguente sistema di iteratori e trasformazioni tra iteratori.
   (a) i. [1 punti] Si definisca una *interfaccia funzionale* simile a quella definita in `java.util.Function` parametrica sia sul tipo dell'argomento che sul tipo di ritorno.
   ii. [4 punti] Si definisca un metodo statico `mapIterator` generico su due tipi `A` e `B` che, dato un iteratore su `A` ed una funzione da `A` a `B`, produce un nuovo iteratore che si comporta come un *wrapper* di quello in input applicando la funzione ad ogni elemento iterato. Si utilizzi il tipo funzione definito nel punto precedente.
   (b) i. [1 punti] Si definisca un tipo per la coppia eterogenea, ovvero una classe `Pair` parametrica su due tipi distinti `A` e `B` che rappresentano rispettivamente il tipo del primo e del secondo elemento della coppia.
   ii. [6 punti] Si definisca un metodo statico `pairIterator` generico su un tipo `A` che, dato un iteratore su `A`, ritorna un nuovo iteratore che itera su coppie di elementi, prendendone due a due dall'iteratore passato come argomento. Quando non sono disponibili 2 elementi, si consideri la sequenza come terminata - in altre parole, una sequenza di lunghezza dispari viene trattata come una sequenza di coppie fino al penultimo elemento, scartando l'ultimo. Si utilizzi il tipo `Pair` definito nel punto precedente.
   (c) [5 punti] Si scriva un blocco di codice Java che utilizza i metodi di cui sopra per trasformare un iteratore di cateti in un iteratore di ipotenuse che applica il teorema di pitagora in tempo reale. In particolare, si scriva un blocco di codice che:
      1. crea una collection iterabile a vostra scelta e la popola con numeri interi casuali usando la classe `Random` del JDK[1];
      2. produce un iteratore su coppie di interi, avente tipo `Iterator<Pair<Integer, Integer>>`, passando l'iteratore originale della collection come argomento al metodo `pairIterator` definito nell'esercizio precedente;
      3. trasforma l'iteratore su coppie di interi appena prodotto in un altro iteratore che itera su `Double` chiamando `mapIterator` con una callback opportunamente definita. Tale callback calcola l'ipotenusa di un triangolo rettangolo data una coppia di cateti. Si badi che i cateti sono interi e le ipotenuse sono `double`.

Total for Question 1: 17

2. Si prenda in considerazione la seguente interfaccia Java:

```
public interface Pool<T, R> {
    void add(T x);                              // popola la pool con un nuovo elemento
    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 `T` astrae il tipo degli oggetti contenuti internamente nella pool, mentre `R` è il tipo della risorsa restituita dalla `acquire()`: il motivo per cui sono due generic differenti è per consentire alle classi che implementano questa

[1] Ricordiamo che la classe `java.util.Random` offre un costruttore senza parametri per inizializzare il PRNG e dei metodi `nextInt()` e `nextDouble()` per generare, rispettivamente, un intero o un double compresi tra 0 e il valore massimo.

interfaccia di rappresentare in modo diverso, se necessario, gli elementi conservati all'interno e le risorse restituite dalla `acquire()`.

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 campo interno. Riportiamo un estratto della classe `LinkedBlockingQueue` definita dal JDK con i metodi pubblici più significativi:

```
class LinkedBlockingQueue<E> implements BlockingQueue<E> {
    LinkedBlockingQueue();                  // costruttore
    void add(E x);                          // aggiunge un elemento
    E take() throws InterruptedException;   // estrae la testa (bloccante)
    E peek();                               // ritorna la testa senza rimuoverla, oppure null se vuota
    int size();                             // numero di elementi
    // etc...
}
```

(a) [3 punti] Si definisca una interfaccia `BasicPool` che estende l'interfaccia `Pool` e che ha un solo generic per rappresentare sia il tipo degli elementi interni sia il tipo delle risorse restituite dalla `acquire()`.

(b) [4 punti] Si implementi una classe `SimplePool` che implementa l'interfaccia `BasicPool` e realizza una semplice coda bloccante.

(c) [7 punti] Si implementi una classe `AutoPool` che implementa l'interfaccia `Pool` realizzando un meccanismo di *auto-release* delle risorse. Se `p` è una pool, l'obiettivo è fare in modo che una risorsa `x` acquisita tramite la chiamata `p.acquire()` non debba essere esplicitamente riconsegnata alla pool chiamando `p.release(x)`, ma sia possibile rilasciarla invocando `x.release()`.

Suggerimento: si definisca un nuovo tipo parametrico `Resource` che si comporta come un *proxy* per l'oggetto contenuto al suo interno e che implementa la logica di *auto-release* rilasciando `this`.

(d) [3 punti] Si automatizzi il meccanismo di cui sopra facendo in modo che un oggetto di tipo `Resource` si rilasci *autonomamente* quando non esistono più riferimenti ad esso.

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

Total for Question 2: 17