Programmazione ad Oggetti mod. 2

4/9/2018

Studente _____ Matricola _____

1. Si prenda in considerazione questa semplificazione della classe `java.util.Random` del JDK: essenzialmente essa offre un costruttore senza parametri, un secondo costruttore con il seed per inizializzare il PRNG ed alcuni metodi per generare valori numerici di tipo differente:

```
public class Random {
    public Random() { ... }
    public Random(int seed) { ... }
    public boolean nextBoolean() { ... }
    public int nextInt() { ... }
    public double nextDouble() { ... }
}
```

   (a) [2 punti] I metodi che essa presenta hanno nomi diversi: sarebbe stato possibile definirli tutti con lo stesso nome (ad esempio `next()`) mantenendo inalterate le firme e la semantica?

   - ◯ Sì: l'overloading sarebbe possibile e permetterebbe anche l'invocazione polimorfa di `next()` con risoluzione dipendente dal contesto;
   - ◯ Non è necessario: Java offre una forma speciale di risoluzione dipendente dal contesto per metodi aventi nome simile e firma differente solo per il tipo di ritorno;
   - ◯ No: l'overloading non è possibile tra metodi che differiscono solamente per il tipo di ritorno;
   - ◯ Sì: l'overloading sarebbe possibile, tuttavia non darebbe alcun beneficio pratico poiché Java ad oggi non consente la risoluzione dell'overloading dipendente dal contesto.

   (b) [6 punti] Si scriva un *wrapper* della classe `Random` che si comporta come un **singleton**, facendo attenzione a riprodurre ogni aspetto dell'originale.

   (c) [6 punti] Si definisca una classe `RandomIterator` che implementa l'interfaccia `java.util.Iterator<Integer>` del JDK e che si comporta come un iteratore su interi, generando un numero casuale ad ogni invocazione del metodo `next()` anziché scorrendo una vera collection, fino ad esaurire la sequenza di lunghezza specificata in costruzione. Si implementino opportunamente il costruttore ed i metodi richiesti dall'interfaccia.

Total for Question 1: 14

2. [6 punti] Si scriva un metodo statico e generico `compareMany` che, dati due parametri di tipo `java.util.Collection` generici su due tipi differenti, confronti ogni elemento di tipo `A` della prima con il corrispettivo elemento di tipo `B` della seconda. Il confronto tra elementi va implementato chiamando il metodo `compareTo` opportunamente; qualsiasi elemento differente rende le collection differenti, così come una lunghezza diversa. Il risultato del metodo `compareMany` è di tipo `int` e deve rispettare la semantica del confronto *a tre vie* di Java, da reinterpretare in modo ragionevole per il caso specifico del confronto tra container.

Total for Question 2: 6

3. Si implementi una sottoclasse generica di `java.util.ArrayList` di nome `SkippableArrayList` che estende la superclasse con un iteratore in grado di discriminare gli elementi secondo un predicato booleano. Gli elementi che soddisfano il predicato vengono processati da una certa funzione di trasformazione[1]; gli altri vengono passati ad una seconda callback (non una funzione di trasformazione).

[1] Una funzione di trasformazione è una funzione in cui dominio è uguale al codominio, per esempio una funzione $f : \tau \to \tau$ è una funzione di trasformazione sull'insieme τ.

(a) [2 punti] Si definisca una *interfaccia funzionale* di nome `Predicate` specializzando l'interfaccia generica `java.util.Function` del JDK in modo che il tipo del parametro dell metodo `apply` sia generico ed il tipo di ritorno sia `Boolean`.

(b) [2 punti] Si definisca una interfaccia di nome `Either` parametrica su un tipo `T` che include due metodi di nome diverso: il primo metodo, `onSuccess`, è una funzione di trasformazione che viene chiamata dall'iteratore quando il predicato ha successo; il secondo metodo, `onFailure`, viene invocato invece quando il predicato fallisce, prende un argomento di tipo `T` e non produce alcun risultato, tuttavia può lanciare una eccezione di tipo `Exception`.

(c) [6 punti] Si definisca la sottoclasse `SkippableArrayList` parametrica su un tipo `E` e si implementi un metodo pubblico avente firma `Iterator<E> iterator(Predicate<E> p, Either<E> f)` che crea un iteratore con le caratteristiche accennate sopra. In particolare:

- l'iteratore parte sempre dall'inizio della collezione ed arriva alla fine, andando avanti di un elemento alla volta normalmente;
- ad ogni passo l'iteratore applica il predicato `p` all'elemento corrente: se il predicato `p` viene soddisfatto allora viene invocato il metodo `onSuccess` di `f` e passato l'elemento corrente come argomento; altrimenti viene invocato il metodo `onFailure` e passato l'elemento corrente come argomento a quest'ultimo;
- l'invocazione di `onFailure` deve essere racchiusa dentro un blocco che assicura il *trapping* delle eccezioni - in altre parole, una eccezione proveniente dall'invocazione di `onFailure` non deve interrompere lo scorrimento della collection da parte dell'iteratore;
- quando viene invocato `onSuccess`, il suo risultato viene restituito come elemento corrente dall'iteratore;
- quando viene invocato `onFailure`, l'iteratore ritorna l'elemento originale che ha fatto fallire il predicato.

(d) [4 punti] Si scriva un esempio di codice `main` che:

- costruisce una `ArrayList` di interi vuota;
- costruisce una `SkippableArrayList` di interi;
- popola quest'ultima con numeri casuali compresi tra 0 e 10, inclusi gli estremi[2];
- invocando **solamente una volta** il metodo `iterator(Predicate<E>, Either<E>)` della `SkippableArrayList` con gli argomenti opportuni, somma 1 a tutti gli elementi maggiori di 5 e appende all'`ArrayList` quelli minori o uguali a 5.

Total for Question 3: 14

[2] Si utilizzi la classe `Random` del JDK: il costruttore non ha parametri ed il metodo per generare un intero tra 0 ed `n` (esclusivo) ha firma `nextInt(int n)`.