Lab Lesson 08

Giacomo Bergami

March 13, 2018

Esercizi

Per controllare se effettivamente avete svolto bene gli esercizi, questa volta fornisco anche dell'ulteriore codice, che controlla la correttezza delle funzioni da voi implementate. Solo nel terzo esercizi non sono presenti file `.java` pre-editati.

1. Dato un ArrayList di numeri interi, modificare il metodo `filter` in modo che aggiorni la lista ls, mantenendo tutti quei numeri x al suo interno che sono o minori (`<`), o maggiori (`>`), o uguali (`==`), o diversi (`!=`), o minori uguali (`<=`) o maggiori uguali (`>=`) ad un numero val dato. Utilizzare la ricorsione tail recursive. Gli operatori aritmetici di confronto verranno passati all'argomento `op`.

2. Dato un ArrayList di numeri interi, modificare i metodi `summate` e `product` in modo che rispettivamente forniscano la sommatoria e la produttoria di tutti gli interi all'interno della lista. Definire queste funzioni in modo non ricorsivo, ma in modo che invochino tutte e due una stessa funzione ricorsiva. **Suggerimento**: guardare alla definizione di tail recursion.

   Curiosità. Alla fine dell'esercitazione, dare un'occhiata alla soluzione alternativa proposta in `GenericFold.java` (non richiesto dal programma). Questo codice fornisce una implementazione generica della tail recursion sulle ArrayList, indipendentemente dalla funzione di accumulazione e dai tipi di dati specifici della lista. In questo modo, con un'unica funzione, riesco ad implementare append di stringhe, sommatorie, produttorie, congiunzioni e disgiunzioni.

3. Si vuole modellare un sistema di voti ed insegnamenti per l'anno accademico in corso. Ogni insegnante, che ha nome e cognome, può insegnare alcune materie. Ognuna di queste, identificate da un nome, ha associata una lista di voti. Ogni voto di ogni materia, espresso in trentesimi, è associato ad uno studente. Ogni studente ha un nome e cognome, ed è identificato da un numero di matricola.

   Ogni insegnante può aggiungere una materia a quelle già insegnate, fornendone un nome. Dato uno studente, un voto ed un nome di materia, ogni insegnante può verbalizzare un voto d'esame. L'insegnate vuole inoltre conoscere, data la materia,

qual è la media dei voti registrati (ovvero, degli esami registrati con almeno 18) e quanti studenti non hanno passato l'esame.

Ogni studente può conoscere l'esito dell'esame senza interpellare il docente ma consultando la materia: in particolare, può sapere se il voto è stato verbalizzato; se tale voto è verbalizzato, può conoscere se è stato promosso o bocciato e, se promosso, può sapere il voto che gli è stato registrato, o il voto con cui è stato bocciato.

Tenendo presente che un docente non può verbalizzare due volte un esame per studente e materia, si svolga interamente l'esercizio.

4. Si riscriva l'esercizio del `Ristorante` spiegato nella lezione suppletiva, in modo da utilizzare `ArrayList` invece di array estendibili, e definire una classe `Dish` che consenta di associare immediatamente il nome dei piatti al suo prezzo, senza ricorrere a due `ArrayList` o due array disgiunti.
   - La classe `Dish` dovrà esibire un costruttore `public Dish(double price, String name)`.
   - La classe `BillItem` esibisce il costruttore `public BillItem(String dish, int no)`.
   - La classe `Restaurant` dovrà esibire i seguenti metodi:
     a) **findPrice**: dato una stringa rappresentante il nome del piatto, restituirne il prezzo associato. Restituire zero qualora il piatto non esista.
     b) **addToMenu**: dato un `Dish d`, aggiorna o inserisce il prezzo associato al nome del piatto.
     c) **bill**: data una `ArrayList` di `BillItem`, che rappresenta un'associazione tra nome del piatto ed il numero di piatti serviti, valutare il prezzo finale del conto.

   Completare le classi `Dish`, `BillItem` e `Restaurant` parzialmente fornite, creando gli opportuni Getter e Setter. Usare `TopChefJava` per valutare il risultato finale.

   Osserva. Per facilitare il processo di compilazione, sono forniti i seguenti script:
   - `compile_all.sh`: compila tutti i file java nell'ordine richiesto. Funzionerà solamente quando tutti i metodi di `BillItem` e `Dish` verranno dichiarati.
   - `clean.sh`: rimuove tutti i file class.
   - `driver.sh`: esegue il driver che utilizza le classi implementate.
   - `test.sh`: esegue il test per verificare la correttezza del codice.

5. Si vuole creare una struttura dati detta *Albero Binario Di Ricerca*, senza utilizzare ereditarietà o polimorfismo. Un albero binario di ricerca possiede un valore r; tale albero può contenere due rami, detti (sotto)albero sinistro e (sotto)albero destro. Se un albero binario di ricerca non ha nessun sottoalbero, allora tale albero si dice *foglia*. Tutti valori presenti nel sottoalbero sinistro devono essere strettamente minori di r, così come i sottoalberi destri sono strettamente maggiori di r.

- **setValue**: dato un albero ed un intero, setta il valore all'interno nel nodo radice corrente. Non restituisce alcun valore. **Osserva:** questa operazione può cambiare il valore settato inizialmente.
- **addUniqueValue**: dato un albero, inserisce un elemento. Se questo già esiste, incrementare il numero di istanze associate a quel determinato valore.
- **countNumbers**: conta il numero delle istanze associate a ciascun valore. Non accetta nessun argomento.
- **sum**: sommo tutti i valori interi presenti nell'albero, e li restituisco. Non accetta argomenti.
- **isLeaf**: dato un albero, restituisce un valore booleano che mi dice se l'albero è una foglia o meno. Non accetta argomenti.
- **toString**: serializza un albero come una lista di interi separati da virgola. Ogni elemento ripetuto è rappresentato una volta sola.

Scrivere una classe `Albero` che sia eseguibile con il codice di verifica `Testing`.