Collection Classes in Java

Progetto del corso di
Programmazione ad Oggetti

A.A. 2006/07

Problema

Si tratta di implementare una gerarchia (semplificata) di classi di strutture dati, analoghe alle *Collection Classes* di Smalltalk. Analoghe classi esistono in realtà anche in Java o C++, ma sarà necessario implementare alcune particolarità del linguaggio Smalltalk.
In particolare, sarà necessario simulare degli oggetti particolari di Smalltalk, i *blocchi*, che sono oggetti che contengono codice e che rispondono a un metodo `valuta()` (eventualmente con parametri) che mette in esecuzione il codice del blocco. Un possibile modo di simulare questi oggetti sarà quello di definire delle interfacce `Blocco` (eventualmente parametrizzate rispetto a tipi generici[1]) e ogni qualvolta necessitiamo di un blocco si crea un oggetto di una classe anonima. Siccome Java è un linguaggio tipato dovremmo distinguere diversi tipi di blocco, a seconda che la funzione `valuta` sia unaria, binaria o senza parametri. Ogni qualvolta un metodo di una collezione richiede un blocco, occorrerà passare come parametro un oggetto creato da una classe anonima che implementa un blocco di tipo opportuno.
Le collezioni dovranno poter contenere oggetti arbitrari, dovranno essere quindi *polimorfe* rispetto agli elementi contenuti.

Faranno parte della valutazione:

- le scelte implementative;
- le scelte sulla struttura della gerarchia;
- la scelta dei problemi e delle soluzioni per testare il corretto funzionamento delle strutture dati implementate;
- utilizzo appropriato delle caratteristiche del linguaggio Java, in particolare delle eccezioni.

Collection Classes

In questa sezione descriviamo le classi da implementare e i metodi richiesti (lo studente è libero di arricchire a suo gusto la gerarchia e l'insieme dei metodi definiti per ciascuna classe).

[1] vedere ad esempio l'implementazione delle liste generiche presentato a lezione e reperibile nella pagina del Diario delle Lezioni

`Collezione<T>` La classe `Collezione` è una classe astratta che definisce il protocollo generale delle classi collezioni ed è la root class della gerarchia.
Tutte le altre classi erediteranno (o ridefiniranno in modo opportuno) tutti i metodi della classe `Collezione`.

Vanno definiti i metodi (eventualmente lasciati astratti):

`boolean vuota()` risponde `true` se la collezione è vuota, `false` altrimenti;

`int dim()` restituisce il numero di elementi della collezione;

`int occorrezeDI(T t)` restituisce il numero di occorrenze dell'elemento `t` nella collezione;

`boolean contiene(T t)` risponde `true` se `t` appartiene alla collezione, `false` altrimenti.

`void aggTutta(Collezione C)` aggiunge alla collezione tutti gli elementi della collezione passata come parametro.

`void fai(BloccoVoid b)` esegue la computazione del blocco `b` su tutti gli elementi della collezione.

`Collezione<S> fai(Blocco<T,S> b)` come fai, solo che questo metodo crea una nuova collezione con la stessa struttura (ma con elementi di tipo `S`), in cui ciascun elemento è il risultato dell'applicazione della valutazione del blocco `b` a un elemento della vecchia collezione;

`Collezione<T> seleziona(BloccoBool b)` genera una nuova collezione con tutti gli elementi che soddisfano la condizione logica del blocco `b`.

`Collezione<T> rifiuta(BloccoBool b)` genera una nuova collezione con tutti gli elementi che **non** soddisfano la condizione logica del blocco `b`.

`<S>inietta(Blocco<S,T>)` inietta un blocco che contiene una funziona `valuta(S f, T g)` binaria: essa restituisce `g` sulla collezione vuota, e calcola iterativamente `f`. Ad esempio iniettare la funzione somma e 0 su una collezione di interi produce come risultato la sommatoria degli elementi della collezione.

Insiemi Un insieme è una collezione di oggetti che non ammette ripetizioni. Gli oggetti di ogni insieme dovranno rispondere a un metodo `equals()`. Implementare le tipiche operazioni sugli insiemi (unione, intersezione, differenza, differenza simmetrica, test di appartenenza). Fare in modo che queste operazioni restituiscano una *nuova collezione*, diversa da quelle originarie. Implementare inoltre le funzioni:

`Insieme<Insieme<T>> parti()` che restituisce l'insieme delle parti dell'insieme che riceve il metodo.

`Insieme<Insieme<T>> parti(int n)` che restituisce tutti i sottoinsiemi di cardinalità `n`.

InsiemiOggetti come gli insiemi, ma i duplicati vengono verificati usando l'identità degli oggetti e non l'uguaglianza. Ricordo che due oggetti sono identici se sono lo stesso oggetto.

Busta Una busta è un insieme in cui sono ammesse le ripetizioni. Matematicamente corrisponde ai cosidetti *multiinsiemi*, in cui ciascun elemento appartiene all'insieme con un certa molteplicità. Rispetto agli insiemi avrà un metodo: `void aggNcopie(int n)`.

`CollezioneOrdinata<T>` Le collezioni ordinate contengono una serie di elementi in un certo ordine. L'ordine dipende dalla sequenza e dal tipo degli inserimenti (non da un ordine definito sugli elementi della collezione). Gli oggetti della classe `Collezioni Ordinate`, risponde ai metodi:

`void aggPrimo(T t)` aggiunge un elemento all'inizio.

`void aggUltimo(T t` aggiunge un elemento alla fine.

`T rimPrimo(T t)` rimuove il primo elemento, restituendolo al chiamante.

`T rimUltimo(T t` rimuove l'ultimo elemento, restituendolo al chiamante.

`void aggPrima(T t1, T t2)` aggiunge l'elemento `t2` **prima** dell'elemento `t1` (se presente, altrimenti ...)

`void aggDopo(T t1, T t2)` aggiunge l'elemento `t2` **dopo** dell'elemento `t1` (se presente, altrimenti ...)

L'esistenza della classe `CollezioneOrdinata` rende banale l'implementazione di code o pile. Nelle code si aggiunge in testa e si rimuove in coda, mentre nelle pile sia inserzioni che rimozioni avvengono in testa.
É corretto definire le classi `Pila` e `Coda` come sottoclassi di `CollezioniOrdinate`? Implementare le classi `Pila<T>` e `Coda<T>` motivando le scelte dell'implementazione.

`CollezioneSortata<T>` Le collezioni sortate sono collezioni ordinate rispetto a una funzione di ordinamento (il cui prototipo è del tipo `boolean sort(T t, T t)`: siccome la collezione è parametrica rispetto a una funzione di ordinamento supporre che la una `CollezioneSortata` abbia un campo che mantiene un blocco che definisce una funzione binaria a valori booleani. Osservare che la funzione di ordinamento può essere modificata, e che la sostituzione della funzione di ordinamento causa necessariamente il riordinamento di tutta la collezione. Implementare opportunamente il metodo:

`void cambiaOrdine(BloccoOrd b)`

Ridefinire opportunamente i metodi delle `CollezioneOrdinata`

`Vettore<T>` Le collezioni vettoriali hanno la particolarità che ciascun elemento è accessibile mediante un indice. Inoltre hanno una dimensione fissata all'atto della creazione. (a tal proposito vedere come eventualemte ridefinire metodi come `rimuovi, agg`... ecc.) Implementare i metodi:

`T at(int n)` che restituisce l'elemento in posizione `n`

`void aggiorna(int n, T t)` che mette l'elemento `t` nella posizione `n`

`VettoreEstensibile<T>` come per vettore, con la differenza che la dimensione può variare dinamicamente.

`Matrice<T>` Le matrici sono particolari collezioni vettoriali, in cui gli elementi sono accessibili specificando due indici. Implementare anche metodi che estraggono come vettore una riga, una diagonale o una colonna (ricordare di considerare due tipi di diagonale una *ascendente* caratterizzata da somma degli indici costante (le potete numerare da 0 a m+n-2, dove m, n sono le dimensioni della matrice) e quelle *discendenti*, caratterizzate dal fatto che la differenza degli indici sono costanti (e le potete numerare da -n-1 a m-1). La classe Matrici sarà una classe astratta con due estensioni concrete: `MatriciDense` e `MatriciSparse`: le matrici dense avranno l'usuale implementazione bidimensionale, mentre le matrici sparse si suppongono diverse da zero solo in relativamente pochi punti e si dovranno memorizzare solo i punti in cui la matrice è diversa da zero. Il cliente del tipo `Matrice` tuttavia non dovrà notare alcuna differenza tra le due implementazioni.

`dimC()` restituisce il numero di colonne della matrice;

`dimR()` restituisce il numero di righe della matrice;

`T at(int n, int m)` che restituisce l'elemento in posizione `n, m`

`void aggiorna(int n, int m, T t)` che mette l'elemento `t` nella posizione `n, m`

`Vettore<T> riga(int n)` restituisce il vettore corrispondente alla riga `n`.

`Vettore<T> col(int n)` restituisce il vettore corrispondente alla colonna `n`.

`Vettore<T> atDA(int n)` restituisce il vettore della diagonale ascendente in cui la somma degli indici è `n`.

`Vettore<T> atDD(int n)` restituisce il vettore della diagonale discendente in cui la differenza degli indici è `n`.

Conversioni e funzioni comuni

Ogni classe collezione dovrebbe avere dei metodi per convertirla in un altro tipo di collezione[2]: ad esempio nella collezione `Busta<T>` definire un metodo `Insieme<T> comeInsieme<T>()` che genera un insieme con gli stessi elementi dell'oggetto `Busta` ricevemente (eliminando ovviamente i duplicati!).

Tutte le classi dovranno inoltre implementare convenientemente:

`String toString()` che trasforma il contenuto di una collezione in una stringa;

`boolean equals(Collezione<T> C)` che verifica se due collezioni sono uguali (osservare che possono essere uguali solo collezioni dello stesso tipo!)

[2]probabilmente non tutte le conversioni hanno un senso preciso: in ogni caso prendere opportune decisioni su come trasformare una collezione in un'altra.

`boolean lessOrEqual(Collezione<T> C)` che definisce una relazione di minore o uguale tra diverse collezioni (scegliere un ordinamento a scelta: per gli insiemi potrebbe essere l'inclusione, per gli array l'estensione lessicografica dell'ordine definito sugli elementi etc.)

Testing

Inventare e risolvere dei piccoli problemini che si risolvono elegantemente grazie alle classi Collezione definite. La scelta dei problemi farà parte della valutazione. E' obbligatoria l'implementazione dei seguenti:

1. definire il prodotto righe per colonne tra matrici, usando solo gli iteratori, definendo prima il prodotto scalare tra due vettori e poi sfruttando i metodi che permettono di estrarre righe e colonne da una matrice.
2. definire un metodo che preso in input un numero `n` restituisce l'insieme di tutte le buste di numeri la cui somma è `n`: ad esempio, avendo come input 4, si dovrebbe costruire l'insieme di buste: {`<1,1,1,1>, <1,1,2>, <1,3>, <2,2>, <4>`}[3].
3. stampare le permutazioni di un vettore. Trattare opportunamente il caso in cui ci siano elementi ripetuti, evitando di stampare più volte la stessa permutazione.
4. testare opportunamente il comportamento di pile e/o code, ad esempio implementando una visita per livelli di un albero binario.

Cosa Consegnare

Lo studente dovrà consegnare un file (in formato `.tar` o `.zip` (mi raccomando: non .rar) contenente:

- I sorgenti Java (che dovranno contenere anche i programmi che testano le classi collezione).
- Una descrizione sintetica (2-4 pagine) del progetto, con descrizione delle scelte implementative più significative.

Gli elaborati andranno inviati via e-mail a `salvo@di.uniroma1.it`, specificando nel subject Progetto PO 06/07. I progetti potranno essere sviluppati in gruppi preferibilmente di due persone.

Buon lavoro!

[3]Osservare che nelle buste non è rilevante l'ordine con cui appaiono gli elementi, cioè la busta `<1,2>` è da ritenersi uguale alla busta `<2,1>`