

22 - Strutture Dati (Java Collections Framework)

Programmazione e analisi di dati
Modulo A: Programmazione in Java

Paolo Milazzo

Dipartimento di Informatica, Università di Pisa
<http://pages.di.unipi.it/milazzo>
milazzo@di.unipi.it

Corso di Laurea Magistrale in Informatica Umanistica
A.A. 2016/2017

Sommario

1 Collezioni: insiemi e liste

2 Iteratori

Collezioni di oggetti (1)

La libreria standard di Java fornisce una serie di classi che consentono di lavorare con **gruppi di oggetti (collezioni)**

Tali classi della libreria standard prendono il nome di **Java Collections Framework**

Sono tutte classi contenute nel package **java.util** della libreria standard di Java

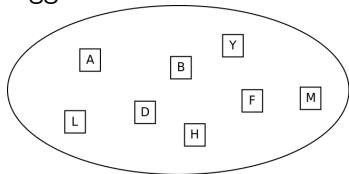
La classe **Vector**, che consente di lavorare con sequenze di oggetti indicizzate, fa parte del Java Collections Framework

Collezioni di oggetti (2)

Il Java Collections Framework prevede due tipologie principali di collezione: **insiemi** e **liste**

Un **insieme** corrisponde a un gruppo di oggetti **tutti distinti** tra loro

- gli elementi possono (facoltativamente) essere mantenuti secondo qualche ordinamento

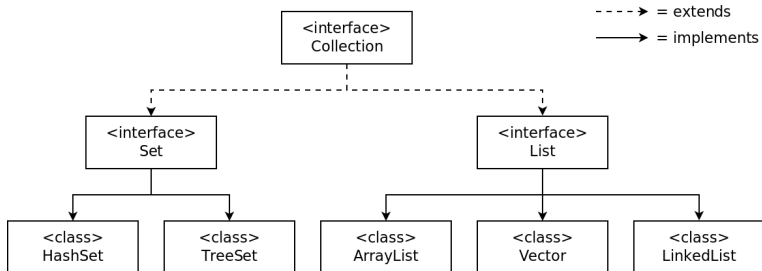


Una **lista** corrisponde a un gruppo di oggetti in cui possiamo avere **elementi ripetuti**

- in questo caso gli oggetti sono generalmente mantenuti in ordine di inserimento



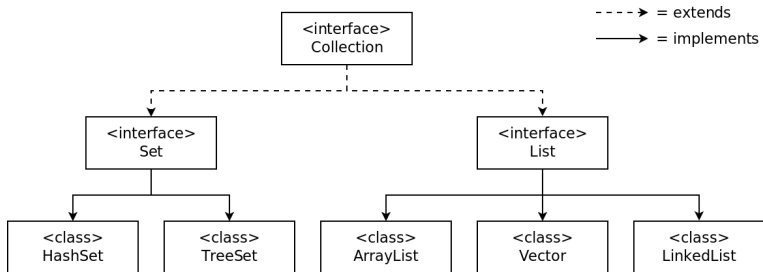
Interfacce e Classi del Java Collections Framework (1)



Il Java Collections Framework descrive le caratteristiche generali delle principali tipologie di collezioni tramite opportune interfacce

- **Collection** è l'interfaccia che descrive genericamente le funzionalità di una collezione
- **Set** è l'interfaccia (estensione di `Collection`) che descrive le funzionalità di un insieme
- **List** è l'interfaccia (estensione di `Collection`) che descrive le funzionalità di una lista

Interfacce e Classi del Java Collections Framework (2)

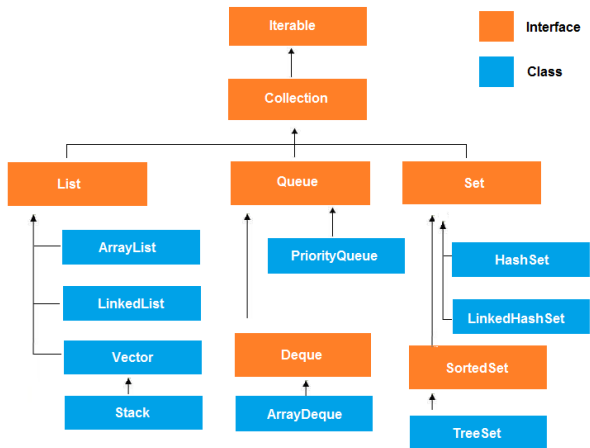


Inoltre, il framework fornisce una serie di implementazioni di queste interfacce, tra cui:

- **HashSet** implementa l'interfaccia `Set` non mantenendo gli elementi in nessun ordine particolare
- **TreeSet** implementa l'interfaccia `Set` mantenendo gli elementi secondo un loro ordine naturale (ordinamento numerico, lessicografico, ecc...)
- **Vector** e **ArrayList** implementano l'interfaccia `List` tramite array dinamici (ridimensionabili)
- **LinkedList** implementa l'interfaccia `List` tramite liste concatenate

Interfacce e Classi del Java Collections Framework (3)

In realtà la gerarchia di classi è anche più articolata:



Per gli scopi di questa lezione è sufficiente lo schema semplificato visto prima...

Attenzione!

Nel resto della lezione verranno descritte le principali classi del Java
Collection Framework

La descrizione si baserà su concetti ed esempi. Per i dettagli sui metodi implementati nella varie classi si veda la documentazione della libreria standard di Java (Java API).

L'Interfaccia Collection

L'interfaccia `Collection` indica i metodi che devono essere presenti in una classe che descrive una generica collezione di oggetti

- **non si fanno assunzioni** sulla natura di tale collezione (elementi ripetuti o meno, ordinamento degli elementi, ecc...)
- perciò, l'interfaccia è volutamente generale e prevede **metodi** per:
 - assicurarsi che un elemento sia nella collezione `add`
 - rimuovere un elemento dalla collezione `remove`
 - verificare se un elemento è nella collezione `contains`
 - verificare se la collezione è vuota `isEmpty`
 - sapere il numero di elementi della collezione `size`
 - generare un array con gli elementi della collezione `toArray`
 - verificare se due collezioni sono "uguali" `equals`
 - ... e altri ... (si veda la doc. della libreria di Java)

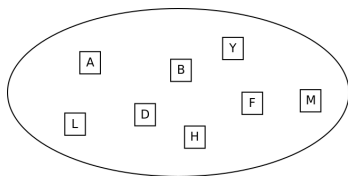
I metodi `add` e `remove` prendono come parametro l'elemento da aggiungere/rimuovere, non la sua posizione.

Abbiamo già visto molti di questi metodi quando abbiamo parlato di `Vector` (che implementa `Collection`).

L'Interfaccia Set

L'interfaccia Set estende Collection introducendo l'idea di **insieme** di elementi

- in quanto insieme, **non ammette elementi duplicati** e non ha una nozione di sequenza o di posizione
- non aggiunge **nessun metodo in più** rispetto a Collection. Pone però dei **vincoli** sull'uso dei metodi:
 - ▶ add aggiunge un elemento solo se esso non è già presente
 - ▶ equals restituisce true se i due Set confrontati contengono gli stessi elementi indipendentemente dall'ordine



L'Interfaccia List

L'interfaccia `List` estende `Collection` introducendo l'idea di **sequenza** di elementi

- tipicamente **ammette elementi duplicati**
- in quanto sequenza, ha una nozione di **posizione**
- pone però dei **vincoli** sull'uso dei metodi di `Collection`, e aggiunge ad essi alcuni **nuovi metodi** per l'accesso posizionale
 - ▶ `add` aggiunge un elemento in ultima posizione (`append`)
 - ▶ `equals` restituisce `true` se i due `List` confrontati contengono gli stessi elementi nelle stesse posizioni
 - ▶ nuovi metodi `add`, `remove`, `get` che agiscono sulla posizione passata loro come parametro



La classe HashSet

La classe HashSet è l'implementazione di Set che viene utilizzata più comunemente

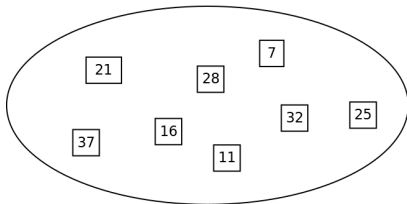
La classe HashSet è implementata in modo da rendere particolarmente efficienti le operazioni di inserimento, ricerca e cancellazione di un elemento

- Utilizza al suo interno una **tabella hash**
- Le operazioni possono essere eseguite in tempo costante (ossia indipendente dal numero di elementi contenuti nell'insieme)

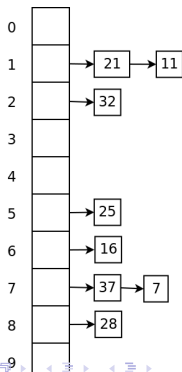
In breve: le tabelle hash

Una **tabella hash** rappresenta un insieme come un array di liste

- usa una **funzione hash** che, dato un elemento, restituisce un numero
- dato un elemento e dell'insieme, la lista che lo conterrà è quella che si trova in posizione $hash(e)$ dell'array
- se la funzione hash riesce a “spalmare” bene gli elementi nelle posizioni, le operazioni di ricerca, inserimento e cancellazione di un elemento diventano veloci (accesso diretto alla posizione giusta dell'array e scansione di una breve lista)



$$hash(n) = n \% 10$$



Esempio di HashSet

Programma che chiede all'utente di inserire 10 parole, e stampa un messaggio ogni volta che si inserisce una parola già inserita in precedenza.

- sfrutta il fatto che il metodo `add` di `Set` restituisce `false` se l'elemento non era presente

```
import java.util.HashSet;
import java.util.Scanner;

public class ParoleRipetute {
    public static void main(String[] args) {

        Scanner input = new Scanner(System.in);
        HashSet<String> set = new HashSet<String>();

        for (int i=0; i<10; i++) {
            String s = input.nextLine();
            // cerca di inserire s in set, se già presente stampa un msg.
            if (!set.add(s))
                System.out.println("Parola ripetuta: " + s);
        }
    }
}
```

Altro esempio di HashSet

Programma che simula il gioco della tombola, ma in cui i numeri vengono rimessi nel sacchetto dopo l'estrazione. Quante estrazioni ci vogliono prima di vedere tutti e 90 i numeri?

```
import java.util.HashSet;
import java.util.Random;

public class TombolaConRipetizioni {
    public static void main(String[] args) {

        HashSet<Integer> numeriEstratti = new HashSet<Integer>();
        Random generator = new Random();

        int cont=0;
        while (numeriEstratti.size()!=90) {
            cont++;
            int numero = 1+generator.nextInt(90);
            numeriEstratti.add(numero);
            System.out.println(cont+": "+numero+" (" +numeriEstratti.size()+")");
        }
    }
}
```

La classe TreeSet

La classe TreeSet è l'implementazione di Set che viene utilizzata quando è utile mantenere gli elementi ordinati (secondo il loro ordine naturale)

Esempi di situazioni in cui è conveniente mantenere un insieme ordinato:

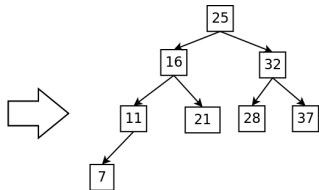
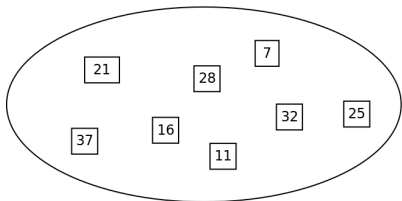
- quando si deve frequentemente visualizzare o rappresentare gli elementi dell'insieme in modo ordinato
- quando si devono frequentemente confrontare due insiemi (confrontare due insiemi ordinati è più facile che confrontare due insiemi disordinati)

La classe TreeSet è implementata in modo da rendere per quanto possibile efficienti le operazioni di inserimento, ricerca e cancellazione di un elemento

- Utilizza al suo interno una rappresentazione ad **albero**
- Le operazioni possono essere eseguite in tempo logaritmico rispetto al numero di elementi dell'insieme

In breve: la rappresentazione ad albero

Un insieme ordinato può essere rappresentato da un albero binario



Per vedere se un elemento è presente

- si parte dalla radice dell'albero
- si scende a destra o a sinistra a seconda che l'elemento cercato sia minore o maggiore di quello corrente
- si ripete il procedimento finchè non si trova l'elemento e non si raggiunge una foglia

Questo procedimento di ricerca richiede di fare circa $\log_2(n)$ passi, se l'insieme contiene n elementi.

Esempio di TreeSet

Programma che inserisce 10 stringhe in un HashSet e in un TreeSet e stampa il contenuto delle due strutture dati.

```
import java.util.*;

public class TestSet {
    public static void main(String[] args) {
        HashSet s1 = new HashSet<String>();
        riempiSet(s1);
        System.out.println("s1 = " + s1);

        TreeSet s2 = new TreeSet<String>();
        riempiSet(s2);
        System.out.println("s2 = " + s2);
    }

    // l'uso dell'interfaccia Set per il parametro consente di usare
    // il metodo ausiliario sia con oggetti HashSet che TreeSet
    private static void riempiSet(Set<String> s) {
        for (int i=0; i<10; i++)
            s.add("str" + i);
    }
}
```

Esecuzione:

```
s1 = [str7, str8, str5, str6, str3, str4, str1, str2, str0, str9]
s2 = [str0, str1, str2, str3, str4, str5, str6, str7, str8, str9]
```

Altro esempio con HashSet e TreeSet

Programma che legge il testo della Divina Commedia, ne conta le parole (con ripetizioni) e le parole distinte.

Si vedano i programmi allegati:

- `ParoleDivinaCommedia.java` — usa `Vector` per raccogliere le parole e `HashSet` per raccogliere le parole distinte
- `ParoleDivinaCommedia2.java` — usa `Vector` per entrambi gli scopi
- `ParoleDivinaCommedia3.java` — usa `Vector` e `TreeSet`

I tre programmi usano il file `divina_commedia.txt` (allegato) contenente il testo dell'opera, e visualizzano il tempo impiegato per completare l'operazione.

Le classi ArrayList e Vector

Le classi ArrayList e Vector (molto simili tra loro) sono le implementazioni di List che si usano più comunemente.

Le classi ArrayList e Vector sono implementate tramite **array dinamici**

- Memorizzano gli oggetti della lista in un array
- Quando l'array è pieno, ne creano uno più grande, ci copiano gli elementi e lo sostituiscono al precedente
- Le operazioni di lettura di un elemento tramite la posizione possono essere eseguite rapidamente (accesso diretto all'array tramite indice)
- La ricerca di un elemento richiede un tempo proporzionale al numero di elementi nella lista (scansione sequenziale dell'array, tempo lineare)
- L'inserimento di un elemento nel mezzo della lista richiede di spostare in avanti tutti gli elementi che seguono (anche qui tempo lineare)

La classe `LinkedList`

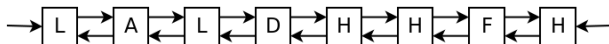
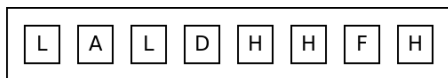
La classe `LinkedList` è l'implementazione di `List` da preferirsi quando si deve frequentemente inserire/rimuovere elementi nel mezzo della lista.

La classe `LinkedList` è implementata tramite una **lista doppiamente puntata**

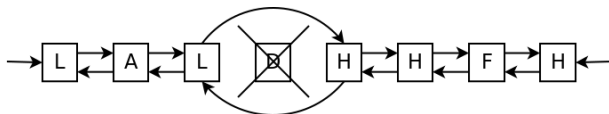
- Ad ogni elemento è aggiunto un riferimento all'elemento successivo e un riferimento al precedente nella lista
- La ricerca di un elemento richiede di scandire la lista da un'estremità (tempo lineare, come per `ArrayList` e `Vector`)
- Anche le operazioni di accesso tramite la posizione richiedono di scandire la lista...
- L'inserimento nel mezzo della lista (una volta trovata la posizione) richiede solo di aggiustare i riferimenti al precedente/successivo nel punto in cui si inserisce il nuovo elemento
- Un `ArrayList` o `Vector`, invece, deve anche spostare tutti gli elementi seguenti avanti o indietro di una posizione (memorizzazione contigua degli elementi)

In breve: la lista doppiamente puntata

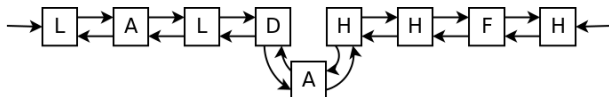
Rappresentazione:



Rimozione dal mezzo della lista:



Inserimento nel mezzo della lista:



Esempio di confronto tra Vector e LinkedList

Di esempi con Vector ne abbiamo già visti in passato

Si veda il programma allegato: `TestList.java`

Questo programma esegue delle operazioni di inserimento e lettura (tramite posizione) in liste implementate tramite Vector e LinkedList, confrontando i tempi di esecuzione.

Sommario

1 Collezioni: insiemi e liste

2 Iteratori

Iteratori (1)

Un **iteratore** è un oggetto di supporto usato per accedere agli elementi di una collezione, uno alla volta e in sequenza

Un iteratore è sempre associato ad un oggetto collezione (lavora su uno specifico insieme o lista)

Per funzionare, un oggetto iteratore deve conoscere (e poter accedere) alla rappresentazione interna della classe che implementa la collezione (tabella hash, albero, array, lista puntata, ecc...)

Iteratori (2)

Un iteratore è un oggetto di una classe che implementa l'interfaccia `Iterator`

L'interfaccia `Collection` contiene il metodo `iterator()` che restituisce un iteratore per la collezione

- le varie implementazioni di `Collection` quali `HashSet`, `Vector`, ecc... implementano il metodo `iterator()` restituendo un oggetto iteratore specifico per quel tipo di collezione (ossia, una specifica implementazione di `Iterator`)

L'interfaccia `Iterator` prevede già tutti i metodi necessari per usare un iteratore. Non è necessario conoscere alcun dettaglio implementativo.

L'interfaccia Iterator

L'interfaccia **Iterator** prevede i seguenti metodi:

- `next()` che restituisce il prossimo elemento della collezione e contemporaneamente sposta il “cursore” dell'iteratore all'elemento successivo
- `hasNext()` che verifica se c'è un elemento successivo da fornire o se invece si è raggiunto la fine della collezione
- `remove()` che elimina l'elemento restituito dall'ultima invocazione di `next()`

Come si usa un iteratore (1)

Un iteratore si usa come nel seguente schema

```
// collezione di oggetti di tipo T che vogliamo scandire
Collection<T> c = ....
...

// iteratore specifico per la collezione c
Iterator<T> it = c.iterator()

// finche'non abbiamo raggiunto l'ultimo elemento
while (it.hasNext()) {
    // ottieni un riferimento all'oggetto corrente, ed avanza
    T e = it.next();
    ....      // usa l'oggetto corrente (anche rimuovendolo)
}
```

Come si usa un iteratore (2)

Un esempio concreto su un insieme di interi (rimuove i pari e stampa di dispari)

```
HashSet<Integer> set = new HashSet<Integer>();  
  
.....  
  
Iterator<Integer> it = set.iterator()  
  
while (it.hasNext()) {  
    Integer i = it.next();  
    if (i % 2 == 0)  
        it.remove();  
    else  
        System.out.println(i);  
}
```

Come si usa un iteratore (3)

Si noti che l'iteratore non ha alcuna funzione che lo “resetti”:

- una volta iniziata la scansione, non si può fare tornare indietro l'iteratore
- una volta finita la scansione, l'iteratore non è più utilizzabile (bisogna crearne uno nuovo)

Gli iteratori sono in realtà il meccanismo usato da Java per realizzare i cicli `for-each`

- scrivere `for(String s : v)` corrisponde in realtà a creare implicitamente un iteratore per `v`
- usare gli iteratori in maniera esplicita consente di fare più cose
 - ▶ suddividere (sospendere) la scansione della collezione in due cicli successivi che usano lo stesso iteratore
 - ▶ rimuovere elementi dalla collezione mentre si itera su essa (proibito con il `for-each`)
 - ▶ interrompere il ciclo prima di aver scandito tutta la collezione

Attenzione!

Nel mentre che si sta usando un iteratore su una collezione, è bene **non apportare modifiche** alla collezione stessa (aggiungere/rimuovere elementi) se non tramite il metodo `remove()` di `Iterator`

Il comportamento dell'iteratore dopo una modifica della collezione è in generale imprevedibile!

Altre classi spesso usate del Java Collection Framework

- **Collections** (notare la 's' finale) Contiene parecchi metodi utili per l'elaborazione di collezioni (di qualunque tipo)
 - ▶ ordinamento di collezioni
 - ▶ calcolo di massimo e minimo
 - ▶ rovesciamento, permutazione, riempimento di una collezione
 - ▶ confronto tra collezioni (elementi in comune, sottocollezioni, ...)
 - ▶
- **HashMap** (implementazione di Map) non è un'implementazione di Collection, ma è comunque una struttura dati molto usata del Java Collection Framework
 - ▶ realizza una struttura dati "dizionario" che associa termini chiave (univoci) a valori
- **Queue, Deque** collezioni specializzate nella gestione di code FIFO (First-In First-Out) o in cui si possono inserire/togliere gli elementi da entrambe le estremità