UNIVERSITÀ DEGLI STUDI DI PARMA
Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea in Informatica

Classi collezione in Java e JSetL: confronti ed implementazioni

Relatore:
Chiar.mo Prof. Gianfranco Rossi

Candidato:
Federica Belli

Anno Accademico 2013/2014

Ai miei genitori,
Daniela e Gabriele.
Ai miei fratelli,
Diego e Elena.

Ringraziamenti

Ringrazio i miei genitori per la pazienza e la generosità infinite.

Ringrazio la mia nonna, che mi è sempre stata vicina.

Ringrazio Diego e Elena per il prezioso aiuto.

Ringrazio il Professor Gianfranco Rossi per la grandissima disponibilità dimostrata nel seguirmi con tranquillità e pazienza nonostante i numerosi impegni.

Ringrazio di cuore i miei compagni e amici Fabio, Gianni e Lorenzo per aver condiviso con me questo percorso, aiutandomi e sostenendomi in ogni momento.

Infine, ringrazio Monica e Andrea, che non hanno mai smesso di credere in me.

Indice

Introduzione

Java offre diverse strutture dati di tipo "collezione", definite come classi e interfacce nella libreria java.util. In particolare, List e Set, che sono molto utili nella pratica.

JSetL è una libreria Java, sviluppata presso il Dipartimento di Matematica e Informatica, pensata principalmente per il supporto alla programmazione dichiarativa a vincoli.

Anche JSetL offre diverse strutture dati di tipo collezione; in particolare, insiemi e liste, definite come altrettante classi (classi LSet e LList).

Una differenza importante tra le collezioni Java e quelle JSetL è che queste ultime possono essere parzialmente specificate.

Questo significa che i valori di alcuni elementi, o di tutta una parte della collezione, possono non essere noti.

Inoltre, questi valori possono essere eventualmente ristretti tramite vincoli posti sulle variabili che li rappresentano.

Questa caratteristica delle collezioni JSetL e' ottenuta utilizzando la nozione di variabile logica (classe LVar) e permettendo ad una collezione JSetL (insieme o lista) di contenere variabili logiche non inizializzate come loro elementi o al posto di parte della collezione stessa.

Ad esempio, se x e y sono variabili logiche non inizializzate, allora, usando una notazione astratta, l'insieme { x,y } rappresenta l'insieme di al piu' due elementi con valore non noto: se poi, successivamente, x e y assumono lo stesso valore v, allora l'insieme diventera' l'insieme singoletto { v }.

Analogamente, sempre usando una notazione astratta, l'insieme { 1|S }, dove S e' un insieme logico non inizializzato, rappresenta un insieme contenente l'elemento 1 e un resto non specificato (e cioe' { 1 } U S).

Per queste loro caratteristiche, queste collezioni vengono indicate come insiemi e liste logiche.

Insiemi e liste logiche costituiscono un utile strumento di supporto alla programmazione dichiarativa, ma possono venir utilizzati anche come collezioni "normali", in modo analogo agli insiemi e liste di java.util.

Quando sono usati in questo modo, pero', insiemi e liste logiche presentano grosse limitazioni prestazionali, dovute essenzialmente al modo in cui queste strutture dati sono implementate all'interno di JSetL.

L'implementazione di insiemi e liste logiche in JSetL infatti, si basa su una struttura dati mista che prevede di memorizzare i dati della collezione in parte in un Vector (campo dElems), e quindi in una struttura dati ad accesso diretto, e in parte in una lista concatenata (campo rElems), e quindi in una struttura dati ad accesso sequenziale.
Quali elementi di una insieme/lista logica vengano memorizzati nel Vector e quali nella lista concatenata dipende dal modo in cui l'utente costruisce l'insieme/lista nel suo programma.
Se l'insieme/lista e' costruito fornendo tutti gli elementi in un colpo solo, ad esempio tramite una insAll, allora gli elementi verranno tutti memorizzati nel Vector; se invece gli elementi vengono forniti uno alla volta, allora verranno memorizzati in una lista formata da piu' insiemi/liste logiche concatenate tramite i loro campi rElems.

Ovviamente l'implementazione di insiemi e liste logiche garantisce che il comportamento funzionale esterno di queste strutture dati sia indipendente dal modo in cui gli elementi vengono memorizzati.

Quando pero' insiemi e liste logiche vengono utilizzati come collezioni "normali", ad esempio per contenere quantita' relativamente grandi di dati, questa loro diversa memorizzazione, in particolare la presenza di liste concatenate, puo' portare a comportamenti fortemente negativi in termini di prestazioni.

Ad esempio, se si crea una lista logica di 1000 elementi tramite inserimenti ripetuti e poi si prova ad eseguire 100 get sull'ultimo elemento inserito (che nella struttura dati concreta si trova in fondo alla lista concatenata rElems) i tempi di esecuzione saranno molto alti e comunque di gran lunga superiori rispetto alle stesse operazioni effettuate su una lista Java (o anche su una lista JSetL, ma costruita fornendo tutti gli elementi in un colpo solo, ad esempio con una insAll).

In questo lavoro di tesi vogliamo analizzare l'attuale implementazione di insiemi e liste logiche in JSetL e quindi proporre alcune implementazioni alternative che permettano di realizzare operazioni di base su queste strutture dati che risultino:

- piu' "pulite" dal punto di vista della struttura interna dell'implementazione (ad esempio, con l'uso di iteratori per scandire le strutture dati interne piuttosto che semplici cicli ad-hoc);

- piu' "complete", dal punto di vista delle operazioni fornite, in modo che siano presenti le analoghe di tutte le operazioni previste per Set e List di Java;
- piu' "controllate", con l'uso dei generics di Java piuttosto che collezioni di Object;
- ultimo, ma non meno importante, piu' efficienti per quanto riguarda i tempi di esecuzione delle principali operazioni.

L'approccio che intendiamo seguire prevede di realizzare due classi, che indicheremo con LSet0 e LList0, che conterranno le strutture dati di base di insiemi e liste logiche e tutte e sole le operazioni di base su insiemi e liste logiche completamente specificate. Le classi LSet0 e LList0 verranno poi usate come classi base delle classi "standard" di JSetL, LSet e LList.

Queste ultime si limiteranno ad estendere LSet0 e LList0 fornendo tutte le altre funzionalita' necessarie per la gestione di collezioni parzialmente specificate, come ad esempio la possibilita' che nelle operazioni su insiemi/liste uno degli oggetti coinvolti sia non inizializzato (con il conseguente lancio dell'eccezione NotInitilizedVar), oppure i constraint su insiemi e liste.

Il fatto che le classi LSet0/LList0 trattino soltanto collezioni completamente specificate significa che le operazioni da esse fornite non terranno conto di elementi non noti ne' di eventuali parti della collezione non specificate.

In particolare, l'uguaglianza tra insiemi/liste sarà un'uguaglianza "sintattica" e non l'unificazione tra esse. Ad esempio, usando sempre una notazione astratta, dati due insiemi r={ 1,x }, dove x e' una variabile logica non inizializzata e s={ 1,2 }, la r.equals(s) restituira' false mentre l'unificazione tra r ed s, r.eq(s), restituirebbe true con x inizializzato a 1.

Le classi LSet0 e LList0 saranno definite utilizzando i generics di Java, nell'ipotesi che anche le classi LSet e LList possano essere definite in questo modo in futuro. In realta', come gia' previsto nell'attuale JSetL, le classi LSet0 e LList0 saranno definite come sottoclassi di una classe comune, la classe LCollection0, che quindi fornisce un super-tipo comune ad entrambe.

In questa prima implementazione di insiemi e liste logiche non si terra' tanto conto dell'ultimo dei punti citati sopra, ovvero gli aspetti di efficienza.

Da questo punto di vista la soluzione proposta potrebbe risultare non migliore di quella attuale e quindi peggiore di quella fornita dalle classe List e Set di java.util.

Gli aspetti prestazionali verranno quindi presi in considerazione in una nuova implementazione delle classi LSet0 e LList0, denominate rispettiamente LSet0n e LList0n (la "n" sta per "normalizzate").

In questa nuova implementazione si cerchera' di sfruttare il piu' possibile l'efficienza delle collezioni di java.util nell'implementazione delle nostre liste e insiemi logici.

A questo scopo, i campi dElems di LSet0n e LList0n verranno definiti rispettivamente come Set e List di java.util piuttosto che come semplici Vector come previsto per gli LSet0 e LList0 (e per gli LSet e LList della implementazione attuale).

Inoltre la maggior parte delle operazioni su insiemi e liste logiche effettueranno, se necessario, una fase preliminare di "normalizzazione" della collezione, che consiste nel "recuperare" tutti gli elementi della collezione eventualmente memorizzati nella lista concatenata realizzata attraverso i campi rElems, per andarli a memorizzare nei rispettivi Set/List contenuti nei campi rElems della collezione.

Questo al fine di permettere successivi accessi agli elementi della collezione logica in modo significativamente piu' efficiente, cosi' come garantito dall'uso delle collezioni.

I capitoli sono strutturati nel seguente modo:

Il capitolo 1 è dedicato alla descrizione del Java Collection Framework. In questo capitolo vengono descritte le caratteristiche principali delle Java Collection; ci soffermeremo, in particolare, sulle interfacce List e Set.

Il capitolo 2 è dedicato alla descrizione di insiemi logichi e liste logiche nella libreria JSetL.

Nel capitolo 3 si parlerà dei tipi generici in Java, descrivendone la sintassi, le tipologie, l'utilizzo e i vantaggi che la loro introduzione all'interno del linguaggio ha apportato.

Il capitolo 4 descriverà la prima implementazione alternativa a quella di insiemi e liste logiche, evidenziando modifiche e aggiunte apportate rispetto a JSetL.

Nel capitolo 5 verranno esposti i problemi evidenziati nella prima delle implemetazioni proposte, i quali hanno quindi portato alla realizzazione di una seconda implementazione, della quale verranno descritte le principali caratteristiche.

Nel capitolo 6 verrà analizzata l'efficienza delle operazioni di insiemi e liste logiche da noi realizzati, confrontando le due implementazioni tra loro e con le Java Collection.

Capitolo 1

Insiemi e Liste in Java

Il package java.util mette a disposizione il Java Collection Framework (JCF), una libreria standard dedicata alle collezioni, intese come classi deputate a contenere altri oggetti.

Questa libreria offre strutture dati di supporto, molto utili alla programmazione, come liste, array di dimensione dinamica, insiemi, mappe associative (anche chiamate dizionari) e code.

In questo capitolo ci concentreremo, in particolare, sulla descrizione delle due interfacce che estendono l'interfaccia Collection, utilizzate per rappresentare e gestire insiemi e liste: Set e List.

1.1 Breve panoramica sulle Java Collection

1.1.1 Cos'è una collezione

Una collezione (chiamata anche contenitore o container) è un oggetto che raggruppa più elementi in una singola unità.

Il Java Collection Framework, come detto sopra, è una libreria volta a manipolare e gestire gli elementi contenuti in una collezione ed è costituito da:

- interfacce: consentono di utilizzare un tipo di collezione senza doverne conoscere l'implementazione;
- classi: implementazioni concrete dei tipi di collezione definiti dalle interfacce;
- algoritmi: metodi che implementano funzioni sulle collezioni come ricerca e ordinamento.

Una collezione Java è una struttura di livello più alto rispetto agli array:

- Non è necessario stabilire a priori il numero massimo degli elementi che può contenere una collezione (negli array dev'essere fissato al momento dell'allocazione);
- E' possibile inserire o cancellare un oggetto in qualsiasi posizione senza dover necessariamente spostare gli altri oggetti;
- E' possibile mantenere un ordinamento tra gli oggetti;
- E' possibile avere diverse politiche di accesso (es: FIFO, LIFO).

1.1.2 Gerarchia e struttura dell'interfaccia Collection

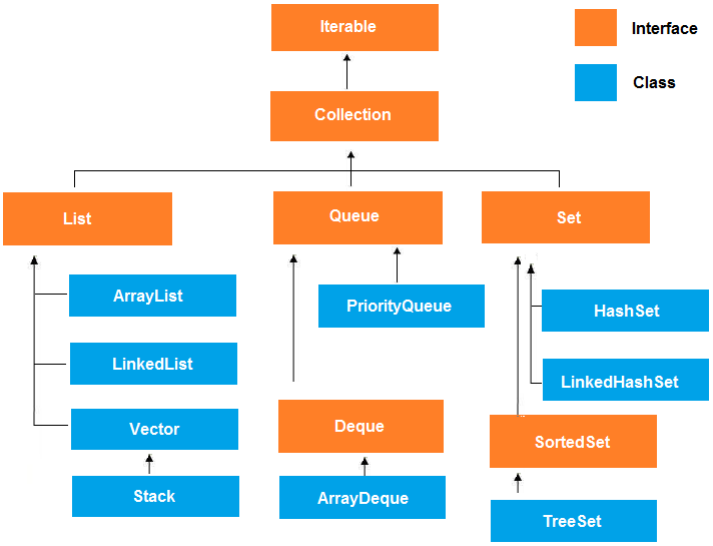

Figura 1.1: Java Collection Hierarchy

L'interfaccia Collection estende l'interfaccia Iterable, di cui però ci occuperemo in modo approfondito in un paragrafo successivo.

Dalla versione 1.5 sono stati introdotti i tipi generici (generics); anche di questo argomento parleremo approfonditamente in un capitolo successivo.

In questo capitolo, considereremo, comunque, la versione generica della varie interfacce.

Qui di seguito riportiamo la struttura dell'interfaccia Collection:

```
public interface Collection <E> {
```

```
        int size();
        boolean isEmpty();
        boolean add(<E> element);
        boolean contains(Object element);
        boolean remove(Object element);
        Iterator <E> iterator();
        boolean equals(Object o);
        int hashCode();

        boolean containsAll(Collection <?> c);
        boolean addAll(Collection<? extends E> c);
        boolean removeAll(Collection <?> c);
        boolean retainAll(Collection <?> c);
        void clear(); // Optional

        Object[] toArray();
        <T> T[] toArray(T[] A);
}
```

I metodi principali dell'interfaccia sono i seguenti:

- int size(): restituisce il numero di oggetti contenuti nella collezione;
- boolean isEmpty(): restituisce vero se e solo se la collezione è vuota;
- boolean add(E x): aggiunge x alla collezione, se possibile; restituisce vero se e solo se x è stato aggiunto con successo;
- boolean contains(Object x) restituisce vero se e solo se la collezione contiene un oggetto uguale (nel senso di equals) ad x;
- boolean remove(Object x) rimuove l'oggetto x (o un oggetto uguale ad x secondo equals) dalla collezione; restituisce vero se e solo se un tale elemento era presente nella collezione.

L'interfaccia Collection è estesa dalle interfacce Set, List e Queue.

Nei paragrafi successivi ci concentreremo sulla descrizione delle prime due.

1.2 L'interfaccia Set

Set rappresenta il tipo di dato insieme, inteso come insieme matematico: gli elementi compaiono senza ripetizioni, e non sono organizzati in un ordine prefissato.

Nella collezione Set gli oggetti non hanno una posizione (non sono ordinati), ma sono organizzati in strutture dati che consentono la loro reperibilità in maniera più efficiente.

Set non aggiunge metodi a Collection, ma ne rende più specifici alcuni.

Un tipico esempio di quanto abbiamo appena affermato riguarda il metodo boolean add(<E> element); infatti, la specifica di un Set rispetto ad una Collection in generale è che esso non può contenere duplicati, quindi se si tenta di aggiungere un elemento già presente usando il metodo add, quest'ultimo restituisce false e l'elemento non viene aggiunto all'insieme che, quindi, non viene modificato.

Set è implementato dalle classi HashSet, LinkedHashSet ed esteso dall'interfaccia SortedSet, la quale, a sua volta è implementata dalla classe TreeSet.

Vediamole nel dettaglio.

La classe HashSet

La classe HashSet è un Set implementato internamente come una tabella hash.

Una tabella hash rappresenta un array associativo suddiviso in bucket. Un bucket è una casella dell'array di HashSet, in cui ogni qualvolta si vuole aggiungere un oggetto viene calcolato un numero intero univoco (con il metodo hashCode) per selezionare il bucket in cui posizionare l'elemento. Ogni casella (bucket) dell'HashSet avrà un certo numero di oggetti collegati tra una lista concatenata.

E' fondamentale la coerenza tra i metodi equals e hashCode; qualora venissero inseriti nell'hashSet due elementi uguali per equals ma che hanno ottenuto due hashCode differenti, si avrebbe un duplicato in HashSet, quindi sarebbero state violate le condizioni del contratto.
Infatti,
"Per ogni coppia di elementi x e y: x.equals(y) == true allora x.hashCode() == y.hashCode()" Se due oggetti risultano uguali per equals allora devono avere lo stesso hashCode.

Dato che la codifica hash sparge gli elementi nella tabella, questo tipo di dato è utile se mantenere l'ordine degli elementi non è importante per la nostra applicazione.

Il vantaggio principale dell'utilizzo degli HashSet è il fatto che permettono di trovare gli elementi molto velocemente.

L'aspetto negativo è che essi non permettono di controllare l'ordine con il quale gli elementi vengono presentati.

La classe TreeSet

La classe TreeSet è implementata internamente come albero di ricerca bilanciato.

E' simile ad un HashSet, ma con un miglioramento aggiuntivo. Un TreeSet è una collezione ordinata. Gli elementi vengono inseriti nella collezione in un ordine qualsiasi, ma quando si itera la collezione gli elementi vengono automaticamente presentati secondo un ordine prestabilito.

L'interfaccia SortedSet implementata da TreeSet rappresenta un insieme sui cui elementi è definita una relazione d'ordine (totale).

L'iteratore di un SortedSet garantisce che gli elementi saranno visitati in ordine, dal più piccolo al più grande.

Inoltre, un tale insieme dispone di due metodi extra:

- first: restituisce l'elemento minimo tra quelli presenti nella collezione
- last restituisce l'elemento massimo.

Questi due metodi non modificano la collezione.

Gli elementi devono essere dotati di una relazione d'ordine, in uno dei seguenti modi:

- gli elementi devono implementare l'interfaccia Comparable; in questo caso, si può utilizzare il costruttore di TreeSet senza argomenti;

  oppure

- bisogna passare al costruttore di TreeSet un opportuno oggetto Comparator.

TreeSet, per confrontare gli elementi, utilizza un ordinamento, fornito da Comparable o Comparator (entrambe sono interfacce che permettono di gestire il confronto, se necessario, fra oggetti di qualsiasi tipo), per decidere dove inserire gli elementi all'interno dell'albero.

Se l'ordinamento non fosse coerente con l'uguaglianza definita da equals, la struttura dati potrebbe comportarsi in maniera imprevedibile.

In genere, la coerenza tra equals e l'ordinamento è consigliata, ma nel caso di TreeSet è obbligatoria!

Vediamo il caso di Comparable:
Per ogni coppia di elementi x e y, in aggiunta alla normale proprietà di equals e compareTo, deve valere la seguente condizione: x.equals(y) == true se e solo se x.compareTo(y) == 0.
Una condizione analoga vale per Comparator.

I LinkedHashSet

Dalla versione 1.4 sono stati introdotti i LinkedHashSet.
Essi sono logicamente uguali agli HashSet, con la sola differenza che, in essi, viene preservato l'ordine di inserimento degli elementi nell'insieme.

Questa classe differisce da HashSet in quanto mantiene una lista doppiamente concatenata che definisce l'ordine di iterazione, che corrisponde all'ordine con cui gli elementi sono stati inseriti nell'insieme. In tal modo, l'iteratore che scorre l'insieme ritornerà gli elementi ordinati in base al loro inserimento nell'insieme.
Questa implementazione evita l'ordinamento 'casuale' tipico degli HashSet, ed ha un'implementazione interna meno costosa di quella di un TreeSet.

I LinkedHashSet, non solo mantengono gli elementi ordinati in base all'inserimento ma, fanno sì che, questo ordine venga mantenuto anche a seguito di eventuali modifiche; questa caratteristica li rende particolarmente adatti in tutti quei casi in cui si necessita di una copia esatta di un Set in cui gli elementi devono essere memorizzati esattamente nell'ordine in cui si trovano nell'oggetto di cui si richiede la copia.

1.3 L'interfaccia List

List estende e specializza Collection introducendo l'idea di sequenza di elementi; questa Collection ammette duplicati; inoltre, viene introdotta la nozione di posizione.

A differenza di Set, l'interfaccia List aggiunge nuovi metodi a Collection, tra cui due metodi posizionali:

- get(int i): restituisce l'elemento i-esimo della sequenza, solleva un'eccezione se la sequenza è vuota oppure se l'indice i è maggiore o uguale a size();

- set(int i, E elem): sostituisce l'elemento che si trova all'indice i-esimo con l'elemento elem; restituisce l'elemento sostituito oppure solleva un'eccezione in caso di indice scorretto.

L'interfaccia List è implementata dalle classi ArrayList, LinkedList e Vector.

Gli ArrayList

ArrayList è un'implementazione di List, realizzata internamente con un array di dimensione dinamica. Ovvero, quando l'array sottostante è pieno, esso viene riallocato con una dimensione maggiore, e i vecchi dati vengono copiati nel nuovo array; questa operazione avviene in modo trasparente per l'utente.

Il metodo size() restituisce il numero di elementi effettivamente presenti nella lista, non la dimensione dell'array sottostante.

Il ridimensionamento avviene in modo che l'operazione di inserimento (add) abbia complessità ammortizzata costante.

In ArrayList, ogni operazione di accesso posizionale richiede tempo costante.

Se l'applicazione richiede l'accesso posizionale, è opportuno utilizzare un semplice array, oppure la classe ArrayList.

Il fatto che l'accesso posizionale sia particolarmente indicato per ArrayList è anche segnalato dal fatto che, delle due, solo ArrayList implementa l'interfaccia RandomAccess.

I LinkedList

Questa classe rappresenta una lista doppiamente concatenata.

Vediamo alcuni metodi tipici della classe LinkedList.

- public void addFirst(Object x): aggiunge x in testa alla lista;
- public void addLast(Object x): equivalente ad add(x), ma senza valore restituito;
- public Object removeFirst(): rimuove e restituisce la testa della lista; solleva NoSuchElementException se la lista è vuota;
- public Object removeLast(): rimuove e restituisce la coda della lista;solleva NoSuchElementException se la lista è vuota;
- public void addFirst(E elem): aggiunge x in testa alla lista;
- public void addLast(E elem): equivalente ad add(x), ma senza valore restituito;

- public E removeFirst(): rimuove e restituisce la testa della lista;
- public E removeLast(): rimuove e restituisce la coda della lista.

Gli ultimi quattro metodi elencati permettono di utilizzare una LinkedList sia come stack sia come coda.

Per ottenere il comportamento di uno stack (detto LIFO: last in first out), inseriremo ed estrarremo gli elementi dalla stessa estremità della lista, ad esempio, inserendo con addLast (o con add) ed estraendo con removeLast.

Per ottenere, invece, il comportamento di una coda (FIFO: first in first out), inseriremo ed estrarremo da due estremità opposte.

In LinkedList, ciascuna operazione di accesso posizionale può richiedere un tempo proporzionale alla lunghezza della lista (complessità lineare); infatti, per accedere all'elemento di posto n è necessario scorrere la lista, a partire dalla testa, o dalla coda, fino a raggiungere la posizione desiderata. Pertanto, è fortemente sconsigliato utilizzare l'accesso posizionale su LinkedList.

Vector

Analogamente alla classe ArrayList, la classe Vector fornisce le funzionalità tipiche di una struttura dati di tipo array che è in grado di ridimensionarsi dinamicamente.

Infatti, se quando si instanzia la lista non si indica la grandezza, ArrayList e Vector cambiano dinamicamente la grandezza del loro Array interno e giustamente lo fanno solo quando non c'è più spazio per il nuovo elemento che si desidera inserire

La differenza sostanziale tra ArrayList e Vector, in questo caso, sta nella nuova grandezza che l'array interno assume: ArrayList lo aumenta del 50%, Vector lo raddoppia.

Si sconsiglia, quindi, di instanziare una lista e incominciare ad aggiungere gli elementi in maniera brutale.

E' preferibile infatti settare una grandezza massima alla lista, tale da rientrare con gli inserimenti ed evitare di pagare il ridimensionamento dinamico dell'array interno.

Vector ha però un vantaggio: se si conosce il rate con cui crescono gli inserimenti è possibile settare il valore di incremento dell'array interno.

1.4 Gli iteratori

1.4.1 Cos' è un iteratore

In informatica, un iteratore è un oggetto che consente di visitare tutti gli elementi contenuti in un altro oggetto, tipicamente un contenitore, senza

doversi preoccupare dei dettagli di una specifica implementazione.

Un iteratore può essere considerato un tipo di puntatore specializzato che fornisce un punto di accesso sequenziale agli elementi di un oggetto che contiene un numero finito di oggetti più semplici, detto aggregato.

L'iteratore offre due operazioni fondamentali:

- Accesso all'elemento dell'aggregato attualmente puntato;
- Aggiornamento del puntatore così che punti all'elemento successivo nella sequenza

Queste semplici operazioni permettono di accedere agli elementi di un aggregato in modo uniforme e indipendente dalla struttura interna dell'aggregato, che può essere ben più complessa delle sequenze lineari implementate da array e liste.

Oltre all'accesso e all'aggiornamento, un iteratore deve fornire come minimo due operazioni:

- Inizializzazione o ripristino dello stato iniziale, in cui l'elemento puntato dall'iteratore è il primo della sequenza;
- Verifica se l'iteratore ha esaurito tutti gli elementi dell'aggregato, cioè se è stato aggiornato oltre l'ultimo elemento della sequenza.

A seconda del linguaggio e delle necessità, gli iteratori possono fornire operazioni aggiuntive o esibire comportamenti diversi.

Un esempio di iteratori specializzati è offerto dagli iteratori bidirezionali, che permettono di visitare l'insieme degli elementi di un aggregato partendo dall'ultimo elemento e procedendo verso il primo.

Un altro esempio è offerto dagli iteratori filtranti, che consentono di visitare soltanto il sottoinsieme degli elementi di un aggregato che soddisfa a condizioni pre-impostate all'interno dell'iteratore.

Una classe iteratore viene solitamente progettata in stretta coordinazione con la corrispondente classe contenitore.
Solitamente il contenitore fornisce i metodi per creare iteratori su di esso.

Lo scopo primario di un iteratore è di consentire al codice che ne fruisce di visitare ogni elemento di un contenitore senza dipendere dalla struttura interna e dai dettagli di implementazione del contenitore stesso.

Questo permette di riutilizzare, con variazioni minime, il codice che accede ai dati. È possibile cioè modificare o sostituire un contenitore con uno di struttura diversa senza compromettere la correttezza del codice che visita i suoi elementi.

Nei linguaggi procedurali è comune usare indici interi per scorrere gli elementi di un array.

Sebbene con alcuni contenitori orientati agli oggetti possano essere usati anche indici interi, l'uso degli iteratori ha i seguenti vantaggi:

- I cicli di conteggio non sono adatti per tutte le strutture dati; in particolare, per le strutture dati in cui l'accesso diretto è lento o assente, come le liste e gli alberi. -
- Gli iteratori possono fornire un modo coerente di iterare sulle strutture dati di ogni categoria, e perciò rendono il codice più leggibile, riusabile, e meno sensibile ai cambiamenti nella struttura dati.
- Un iteratore può imporre restrizioni di accesso aggiuntive, come assicurare non si saltino degli elementi o che non si visiti più volte lo stesso elemento.
- Un iteratore può consentire all'oggetto contenitore di essere modificato senza invalidare l'iteratore. Per esempio, dopo che un iteratore è avanzato oltre il primo elemento può essere possibile inserire elementi aggiuntivi all'inizio del contenitore con risultati predicibili. Con l'uso di indici, questo è problematico dal momento che gli indici devono cambiare.

1.4.2 Iteratori in Java

L'interfaccia Iterator

Gli iteratori in Java sono oggetti che estendono l'interfaccia Iterator.

Vediamola nella sua versione parametrica:

```
public interface Iterator<E> {
        public E next();
        public boolean hasNext();
        public void remove();
}
```

Quest'interfaccia dichiara tre metodi: next(), hasNext() e remove().

Un iteratore, quando viene creato, punta al primo elemento della struttura dati.

Con il metodo hasNext(), si chiede all'iteratore se sono presenti altri elementi da iterare. Se la struttura dati non è vuota, questo restituirà true.

E' possibile accedere all'elemento puntato tramite il metodo next(), che porta l'iteratore a puntare all'elemento successivo. Questo significa che, la seconda volta che verrà chiamato, il metodo next() restituirà il secondo elemento e così via.

Se non ci sono altri elementi, questo metodo solleva un'eccezione, più precisamente una NoSuchElementException.

Proprio per questo motivo è sempre consigliabile, per non dire obbligatorio, usare il metodo hasNext() prima dell'eventuale chiamata di next(). Il metodo next() non può, quindi, essere chiamato prima di questo metodo, altrimenti restituirà una IllegalStateException.

Il metodo remove() rimuove l'elemento che è stato restituito dall'ultima chiamata del metodo next(). Ciò può avere senso in molte situazioni, quando è necessario vedere l'elemento prima di decidere che si tratti proprio dell'elemento da rimuovere.

Se, invece, si vuole rimuovere un elemento che si trova in una posizione particolare, è comunque necessario passare oltre l'elemento.

Iterare sulle Java Collection

L'ordine con il quale un iteratore scorre gli elementi di una Java Collection dipende dal tipo di collezione. Se l'iterazione riguarda, per esempio, un ArrayList, l'iteratore inizia con indice 0 e incrementa l'indice ad ogni passaggio, mentre se si scorrono gli elementi in un HashSet, lo scorrimento avviene sostanzialmente seguendo un ordine casuale. Si può essere sicuri di incontrare tutti gli elementi della collezione nel corso dell'iterazione, ma non si possono fare previsioni in merito all'ordine con il quale vengono incontrati. Di solito, questo non è un problema, in quanto l'ordine non ha nulla a che fare con l'elaborazione.

C'è una differenza importante da un punto di vista concettuale tra gli iteratori della libreria delle collezioni Java e gli iteratori delle altre librerie.

Nelle librerie delle collezioni tradizionali, gli iteratori vengono modellati in base agli array. Dato un tale iteratore, si ricerca l'elemento memorizzato in una data posizione, analogamente a come si ricerca l'elemento i-esimo in un array.

Indipendentemente dalla ricerca, si può avanzare l'iteratore sulla posizione successiva. Questa operazione è simile all'avanzamento dell'indice di un array che si effettua chiamando i++, senza eseguire una ricerca. Tuttavia, gli iteratori Java non funzionano in questo modo.

La ricerca e la modifica della posizione sono operazioni strettamente accoppiate tra loro. L'unico modo per cercare un elemento consiste nel chiamare il metodo next, ma questa ricerca fa avanzare l'iteratore alla posizione successiva.

Conviene allora pensare agli iteratori di Java come a un segnaposto tra due elementi. Quando si chiama next(), l'iteratore salta sull'elemento successivo e restituisce un riferimento all'elemento che è stato passato. Abbiamo detto che un iteratore ci consente di navigare tra gli elementi di una struttura dati in modo sequenziale.

L'interfaccia Iterable Per creare un iteratore su una collezione una possibilità è quella di definire una classe che implementi l'interfaccia Iterator, ma esiste un metodo molto più semplice.

Tutte le Collection, infatti, implementano l'interfaccia Iterable:

```
public interface Iterable{

        Iterator iterator();

}
```

Come si può notare, l'interfaccia dichiara un solo metodo: iterator(), che restituisce un iteratore sulla collezione stessa. Invocando il metodo iterator() sulla collezione che vogliamo iterare, questo restituirà un oggetto rappresentante proprio un iteratore per la collezione stessa.

Vediamo un esempio di creazione e utilizzo di un iteratore su una una LinkedList<String>:

```
List<String> l=new LinkedList<String>();
l.add('rosso');
l.add('verde');
Iterator<String> it=l.iterator();
while(it.hasNext()){
        String stringa = it.next();
        System.out.println(stringa);
}
```

Vediamo cosa abbiamo fatto in questo esempio.

Per prima cosa otteniamo un oggetto iteratore con il metodo iterator() chiamato sulla lista l (come le Collection, anche le interfacce Iterator e Iterable sono parametriche). Chiamiamo il metodo *hasNext*() come condizione del costrutto while. In questo metodo entreremo nel ciclo solo a patto che siano presenti elementi in l. Dentro il ciclo, sfruttando il metodo *next*(), otteniamo un riferimento al prossimo elemento di l e ci spostiamo su quello successivo.

La List l del nostro esempio contiene tre elementi, quindi il ciclo funzionerà come segue:

- hasNext() restituise true perchè sono presenti altri due elementi in l;
- next() restituisce la String 'rosso';
- hasNext() restituisce true perchè c'è ancora un elemento in l;
- next() restituisce la String 'verde';
- hasNext() restituisce false perchè non sono presenti altri elementi in l, quindi il ciclo termina.

Vediamo un ulteriore esempio in cui utilizziamo anche il metodo remove() dell'interfaccia Iterator (utilizzeremo ancora la lista l dell'esempio precedente).

```
Iterator<String> it=l.iterator();
it.next(); // si passa oltre il primo elemento;
it.remove; //ora viene rimosso;
```

E' molto importante ricordare che c'è un legame tra le chiamate del metodo next() e il metodo remove().

Non è ammesso chiamare remove() se questa chiamata non è preceduta da una chiamata di next(), diversamente verrebbe sollevata una IllegalStateException.

Se vogliamo rimuovere due elementi adiacenti, non è possibilie chiamare semplicemente:

```
it.remove();
it.remove(); // errore!
```

In questo caso si deve prima chiamare *next*() per superare l'elemento che si vuole rimuovere:

```
it.remove();
it.next();
it.remove(); //ok
```

Il ciclo for-each

A partire da Java 1.5, è stato introdotto un nuovo tipo di ciclo for, detto "for-each". Facendo sempre riferimento alla List lista dell'esempio precedente, possiamo adesso scrivere il seguente codice:

```
for(String stringa : lista){

        System.out.println(stringa);
}
```

Questo blocco di codice è equivalente al precedente. Questa nuova forma di ciclo permette di iterare su un array senza dover esplicitamente utilizzare un indice, quindi senza il rischio di sbagliare gli estremi dell'iterazione.

Il ciclo for-each è molto più sintetico e riduce il rischio di scrivere codice errato, ma, per contro, dà meno libertà e può essere utilizzato solo per leggere dati da una struttura, non per rimuoverli o fare altre modifiche.

Capitolo 2

Insiemi e liste in JSetL

2.1 Breve panoramica su JSetL

JSetL è una libreria Java, sviluppata presso il Dipartimento di Matematica e Informatica, pensata principalmente per il supporto alla programmazione dichiarativa a vincoli.

Anche JSetL offre diverse strutture dati di tipo collezione; in particolare, insiemi e liste, definite come altrettante classi (classi LSet e LList).

Una differenza importante tra le collezioni Java e quelle JSetL è che queste ultime possono essere parzialmente specificate.

Questo significa che i valori di alcuni elementi, o di tutta una parte, della collezione, possono non essere noti. Inoltre, questi valori possono essere eventualmente ristretti tramite vincoli posti sulle variabili che li rappresentano.

Questa caratteristica delle collezioni JSetL e' ottenuta utilizzando la nozione di variabile logica (classe LVar) e permettendo ad una collezione JSetL (insieme o lista) di contenere variabili logiche non inizializzate come loro elementi o al posto di parte della collezione stessa.

Ad esempio, se x e y sono variabili logiche non inizializzate, allora, usando una notazione astratta, l'insieme x,y rappresenta l'insieme di al piu' due elementi con valore non noto: se poi, successivamente, x e y assumono lo stesso valore v, allora l'insieme diventera' l'insieme singoletto v.

Analogamente, sempre usando una notazione astratta, l'insieme { 1|S }, dove S e' un insieme logico non inizializzato, rappresenta un insieme contenente l'elemento 1 e un resto non specificato (e cioe' { 1 } U S).

Per queste loro caratteristiche, queste collezioni vengono indicate come insiemi e liste logiche.

Vediamo la struttura di insiemi e liste logiche:

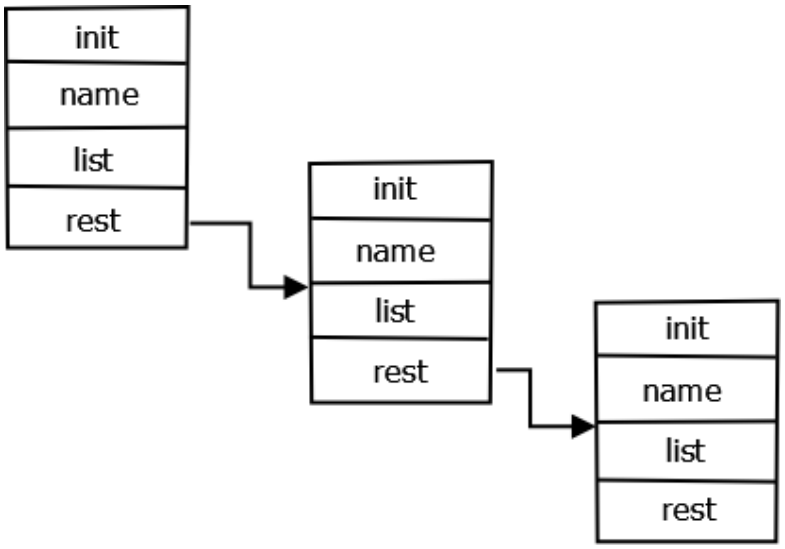

Figura 2.1: memorizzaione in JSetL

Quando pero' insiemi e liste logiche vengono utilizzati come collezioni "normali", ad esempio per contenere quantita' relativamente grandi di dati, questa loro diversa memorizzazione, in particolare la presenza di liste concatenate, puo' portare a comportamenti fortemente negativi in termini di prestazioni.

Ad esempio, se si crea una lista logica di 1000 elementi tramite inserimenti ripetuti e poi si prova ad eseguire 100 get sull'ultimo elemento inserito (che nella struttura dati concreta si trova in fondo alla lista concatenata rest) i tempi di esecuzione saranno molto alti e comunque di gran lunga superiori rispetto alle stesse operazioni effettuate su una lista Java (o anche su una lista JSetL, ma costruita fornendo tutti gli elementi in un colpo solo, ad esempio con una insAll).

2.2 Cos'è una variabile logica

Le variabili logiche rappresentano delle incognite.

In JSetL, una variabile logica è un'istanza della classe LVar. Questa classe fornisce costruttori per creare variabili logiche e dei semplici metodi per testarle e manipolarle, oltre ad alcuni vincoli di base.

Possiamo associare ad una LVar un valore relativo (valRel) ed un nome esterno (extName), entrambi opzionali.

Il valore relativo di una variabile logica può essere specificato al momento della costruzione ma in seguito non pu'o essere modificato direttamente con metodi della classe Lvar.

Quando ad una variabile logica viene associato un valore, essa viene detta bound (inizializzata), altrimenti viene detta unbound (non inizializzata).

2.3 Liste Logiche: la classe LList

Una lista logica è un particolare tipo di variabile logica il cui valore è una coppia di elementi $\langle elems, rest \rangle$, dove elems è una lista [e_0,...,e_n], con n$\geq$ 0 oggetti di tipo arbitrario e rest (detto anche coda) è una lista vuota o unbound.

Sia l una lista logica, quando rest è una lista non inizializzata, possiamo dire che la lista logica l è una lista aperta e usiamo la notazione [e_0,...,e_n | r] per denotarla; diversamente, quando rest è una lista logica vuota, possiamo dire che l rappresenta una lista chiusa e possiamo usare la notazione astratta [e_0,...,e_n].

Quando elems è vuoto e rest è null, usiamo la notazione [] per denotare la lista logica in questione.

Una lista logica contenente oggetti logici (variabili logiche o insiemi logici) non inizializzati, rappresenta una lista parzialmente specificata.

In JSetL una lista logica è un'istanza della classe LList che estende la classe LCollection; in particolare, la parte elems della lista logica è un'istanza della classe ArrayList, la quale implementa l'interfaccia java.util.List.

La classe LList fornisce metodi per creare nuove l-list, la possibilità di creare liste nuove da liste esistenti e altri metodi che riguardano la manipolazione degli elementi contenuti in una lista logica.

Inoltre, fornisce metodi legati alla gestione dei vincoli, di cui noi, però, non ci occuperemo.

Vediamo i metodi principali della classe LList.

- Costruttori:
  - LList()
  - LList(String extName)
    Crea una LList non inizializzata (unbound); extName, se specificato, viene assegnato come nome esterno alla variabile, in caso contario le viene assegnato automaticamente il nome esterno di default "?".

  - LList(List<?> l)
  - LList(String extName, List<?> l)

    Crea una LList (bound) dove il nome esterno è inizializzato a extName se specificato e value è uguale a l.
  - LList(LList ll)
  - LList(String extName, LList ll)

    Crea una LList, dove il nome esterno è inizializzato con extName e il suo valore è inizializzato con ll.

    Vediamo alcuni esempi

    1 Creazione di una LList unbound a.
    LList a = new LList();
    2 Creazione di una LList avente nome esterno "b" e valore [1,2,3].
    List l = new ArrayList(); l.add(1); l.add(2); l.add(3); LList b = new LList(b, l);
    3 Creazione di una LList equivalente alla LList a.
    LList c = new LList(a);

- Creazione di nuove LList (bound)
  - static LList empty();

    Ritorna una LList vuota.
  - LList ins(Object o);

    Ritorna una nuova lista logica avente o come primo elemento della lista e la lista di invocazione come resto; la nuova lista ritornata sarà inizializzata o non inizializzata a seconda della lista di invocazione.
  - LList insn(Object o);

    Come ins, ma o è aggiunto come ultimo elemento della lista.
  - LList insAll(Object[] c);
  - LList insAll(Collection c);

    Ritorna una nuova LList avente come valore tutti gli elementi di c e come resto la lista di invocazione; la nuova lista ritornata sarà inizializzata o non inizializzata a seconda della lista di invocazione.

    Vediamo alcuni esempi.

    1 Creazione di una lista d inizializzata a lista vuota.
    LList d = LList.empty();

2 Creazione di una lista avente come valore ['c','b','a'].
LList e = LList.empty().ins('a').ins('b').ins('c');
oppure
char[] elems = 'a','b','c'; LList e = LList.empty().insAll(elems);

3 Creazione di una lista l parzialmente specificata avente come valore [1,x], dove x è una variabile logica unbound.
LVar x = new LVar(); LList f = LList.empty().insn(1).insn(x);

4 Creazione di un LList aperta g avente come valore [2|r], dove r è una LList unbound.
LVar z = new LVar(2); LList r = new LList(); LList g = r.insn(1).insn(z);

5 Creazione di una lista h avente il valore [[],['c','b','a'],[1,x]] (cioè una lista di liste).
LList e = LList.empty().ins(f).ins(e).ins(d);
dove d, e, e f sono liste create precedentemente.

- Metodi generali
  - LList clone();

    Crea e ritorna una copia della lista di invocazione.
  - boolean equals(LList ll);

    Ritorna vero se la lista di invocazione è uguale a ll; ritorna falso altrimenti.
  - boolean equals(Object o);

    Ritorna vero se la lista di invocazione è uguale a o; ritorna falso altrimenti.
  - String getName();

    Ritorna il nome esterno della lista di invocazione.
  - Object getValue();

    Ritorna vero se la lista di invocazione è bound; ritorna falso altrimenti.
  - void output();

    Stampa il nome esterno della lista, seguito da '=', seguito poi dal valore della lista di invocazione se è bound, oppure da "unknown" se è unbound.
  - LList setName(String extName);

    Inizializza il nome esterno della variabile di invocazione con extName.

- String toString();
  Ritorna la stringa corrispondente all'elenco degli elementi della lista di invocazione se questa è bound; ritorna il nome esterno della lista altrimenti.
- Object get(int i);
  Se la variabile è bound, ritorna l'elemento i-esimo della collezione; se questo non esiste viene sollevata una NotInitVarException.
  Ritorna vero se la lista di invocazione ha una parte resto vuota, ritorna falso altrimenti
- boolean isEmpty();
  Se la lista di invocazione è bound ritorna vero se essa è vuota, se la lista è unbound solleva una NotInitVarException.
- void printElems(char sep);
  Se la lista di invocazione è bound, stampa tutti i suoi elementi separati dal carattere 'sep' (non racchiusi tra parentesi quadre); se la lista di invocazione è unbound, stampa il suo extName.
- Vector toVector(); Ritorna un Vector contenente tutti gli elementi della lista di invocazione; se la lista è unbound, viene sollevata una NotInitVarException.
- int getSize();
  Se la lista è bound, ritorna il numero dei suoi elementi. Altrimenti, viene sollevata una NotInitVarException. Sia l una LList inizializzata, allora l.getSize() è equivalente a l.getValue().size().
- Iterator iterator(); Se la lista è inizializzata, ritorna un iteratore sui suoi elementi; altrimenti viene sollevata una NotInitVarException. Sia l una LList inizializzata, allora l.iterator() è equivalente a l.getValue(). iterator().
- boolean testContains(Object o);
  Se la lista di invocazione è inizializzata, il metodo restituisce true se la lista d'invocazione contiene o; si noti che o può essere un oggetto logico (o una variabile logica o un insieme logico) e, in tal caso, viene considerato il valore ad esso associato ai fini di stabilire l'appartenenza di o alla lista di invocazione. Se la lista di invocazione è non inizializzata, viene sollevata una NotInitVarEXception.
  Sia l una LList inizializzata, allora l.testContains(o) equivale a l.getValue().contains(o.getValue()) oppure a l.getValue().contains(o).

Si noti che tutti questi metodi tengono conto solo della parte value della lista logica, mentre la parte rest è sempre ignorata.

La classe LList fornisce altri metodi legati alla gestione dei vincoli, di cui, però noi non ci occuperemo.

2.4 Gli insiemi Logici: la classe LSet

Un insieme logico è un particolare tipo di variabile logica il cui valore è una coppia di elementi $\langle elems, rest \rangle$, dove elems è un insieme [e_0,...,e_n], con n di oggetti di tipo arbitrario (n$\geq$ 0) e rest (detto anche coda) è un insieme vuoto o unbound rappresentante il resto dell'insieme.

Quando rest è un insieme non inizializzato, possiamo dire che l'insieme logico s è un insieme aperto e per denotarlo usiamo la notazione [e_0,...,e_n | r]; diversamente, quando rest è un insieme logico vuoto, possiamo dire che s rappresenta un insieme chiuso e usiamo la notazione [e_0,...,e_n].

Quando elems è vuoto e rest è null, usiamo la notazione [] per denotare il nostro insieme logico s.

Gli insiemi logici sono molto simili alle liste logiche per molti aspetti.

In particolare, come le liste logiche, anche gli insiemi logici possono rappresentare delle collezioni parzialmente specificate.

La differenza principale con le l-list è che l'ordine degli elementi e le loro ripetizioni non sono importanti, mentre in una l-list contano.

Da notare che, a differenza delle l-list, la cardinalità di un insieme parzialmente specificato non è determinata in modo univoco (anche se l'insieme è chiuso).Per esempio, la cardinalità dell'insieme { 1,x }, dove x è una variabile logica non inizializzata, può essere 1 o 2 a seconda che x sarà inizializzata ad un valore uguale a 1 o diverso da 1, rispettivamente.

In JSetL un insieme logico è un'istanza della classe LSet che estende la classe LCollection; in particolare, la parte elems della lista logica è un'istanza della classe HashSet, la quale implementa l'interfaccia java.util.Set.

La classe LSet fornisce metodi per creare l-set, la possibilità di creare nuovi l-set da l-set esistenti e altri metodi che riguardano la manipolazione degli elementi contenuti in un insieme logico.

Inoltre, fornisce metodi legati alla gestione dei vincoli, di cui noi, però, non ci occuperemo.

Vediamo i metodi principali della classe LSet considerati in questo lavoro.

- Costruttori
  - LSet()

- LSet(String extName)

  Crea un LSet non inizializzato (unbound); extName, se specificato, viene assegnato come nome esterno dell'insieme, in caso contario gli viene assegnato automaticamente il nome esterno di default "?".
- LSet(Set<?> s)
- LSet(String extName, Set<?> s)

  Crea un LSet (bound) dove il nome esterno è inizializzato a extName se specificato e value è uguale a s.
- LSet(LSet ls)
- LSet(String extName, LSet ls)

  Crea un LSet, dove il nome esterno è inizializzato con extName e il suo valore è inizializzato con ls.

  Vediamo alcuni esempi.

  1 Creazione di un LSet unbound a.
  LSet a = new LSet();
  2 Creazione di una LSet avente nome esterno "b" e valore [1,2,3].
  Set s = new HashSet(); s.add(1); s.add(2); s.add(3); LSet b = new LSet(b, s);
  3 Creazione di un LSet c equivalente al LSet a.
  LSet c = new LSet(a);

- Creazione di nuovi LSet (bound)
  - static LSet empty();

    Ritorna un LSet vuoto.
  - LSet ins(Object o);

    Ritorna un nuovo l-set avente o come primo elemento dell'insieme e l'insieme di invocazione come resto; il nuovo insieme ritornato sarà inizializzato o non inizializzato a seconda dell'insieme di invocazione.
  - LSet insn(Object o);

    Come ins, ma o è aggiunto come ultimo elemento dell'insieme.
  - LSet insAll(Object[] c);
  - LSet insAll(Collection c);

    Ritorna un nuov LSet avente come valore tutti gli elementi di c e come resto l'insieme di invocazione; il nuovo insieme ritornato sarà inizializzato o non inizializzato a seconda dell'insieme di invocazione.

- static LSet mkList(int n);
- static LSet mkList(String extName, int n);
  Ritorna un nuovo LSet il cui nome esterno sarà extName, se specificato, e il suo valore sarà un insieme di n variabili logiche unbound.

  Vediamo alcuni esempi.

  1 Creazione di un insieme inizializzato a insieme vuoto.
  LSet d = LSet.empty();
  2 Creazione di un insieme avente come valore ['c','b','a'].
  LSet e = LSet.empty().ins('a').ins('b').ins('c');
  oppure
  char[] elems = 'a','b','c'; LSet e = LSet.empty().insAll(elems);
  3 Creazione di un insieme s parzialmente specificato avente come valore [1,x], dove x è una variabile logica unbound.
  LVar x = new LVar(); LList f = LList.empty().insn(1).insn(x);
  4 Creazione di un LSet aperto g avente come valore [2|r], dove r è un LSet unbound.
  LVar z = new LVar(2); LSet r = new LSet(); LSet g = r.insn(1).insn(z);
  5 Creazione di un insieme h avente il valore [[],['c','b','a'],[1,x]] (cioè un insieme di insiemi).
  LSet e = LSet.empty().ins(f).ins(e).ins(d);
  dove d, e, e f sono insiemi creati precedentemente.

- Metodi Generali

  I seguenti metodi sono gli stessi delle classi LVar e LList ma adattati ad oggetti LSet.

  - LSet clone(); Crea e ritorna una copia dell'insieme di invocazione.
  - boolean equals(LSet ls); Ritorna true se l'insieme di invocazione è uguale a ls.
  - boolean equals(Object o); Riotrna true se l'insieme di invocazione è uguale a o; se o è un oggetto logico, si tiene conto dei valori ad esso associati per stabilire l'uguaglianza con l'insieme di invocazione.
  - String getName(); Ritorna il nome esterno dell'insieme di invocazione.
  - Object getValue(); Ritorna il valore dell'insieme di invocazione se quest'ultimo è inizializzato; ritorna null altrimenti.

- boolean isBound(); Ritorna true se l'insieme è inizializzato; ritorna false altrimenti.
- void output(); Stampa il nome esterno dell'insieme, seguito da '=', seguito poi dal valore dell'insieme di invocazione se è bound, oppure da "unknown" se è unbound.
- LList setName(String extName); Imposta il nome esterno dell'insieme di invocazione con extName.
- String toString(); Ritorna la stringa corrispondente all'elenco degli elementi dell'insieme di invocazione se questo è bound; ritorna il nome esterno dell'insieme altrimenti.
- Object get(int i);
- LSet getRest();
- boolean isClosed();
- boolean isEmpty();
- boolean isGround();
- void printElems(char sep);
- Vector toVector();
- int getSize();
- Iterator iterator();
- boolean testContains(Object o);

Tutti i metodi sopra elencati funzionano come quelli di LList con alcune differenze legate proprio alle princiaple differenza che c'è tra un insieme logico e una lista logica, infatti, un insieme può contenere più occorrenze dello stesso elemento, mentre una lista no. Quindi, per esempio, se l'insieme s contiene due occorrenze dell'elemento 2, il metodo getSize() chiamato su s ritornerà 1; ciò non accadrebbe nel caso di una lista logica, sulla cui chiamata di getSize() otteremmo come valore ritornato 2.

Inoltre, l'implementazione di get(int i) per un insieme logico garantisce solo che diversi valori di i corrispondano a diversi elementi di un insieme logico perchè su un insieme non ha senso parlare di posizione i-esima di un elemento, sempre per il fatto che ordine e duplicati non contano.

La classe LSet fornisce altri metodi legati alla gestione dei vincoli, di cui, però noi non ci occuperemo.

Ci limitiamo solo a dire che alcuni di questi metodi realizzano le operazioni insiemistiche di unione, differenza e intersezione.

Capitolo 3

Tipi Generici Java

Con JDK 1.5 sono state introdotte nuove funzionalità nel linguaggio Java: una delle più importanti è l'introduzione dei tipi generici, più comunemente chiamati Generics.

I Generics aggiungono al linguaggio Java la possibilità di 'parametrizzare' i tipi di dati gestiti ad esempio nei contenitori (Collection, Map, Set etc..).

Questo approccio fornisce un meccanismo per comunicare al compilatore il tipo di dato che un contenitore può ricevere, in modo che possa essere controllato in fase di compilazione.

Una volta che il compilatore conosce il tipo dell'elemento del contenitore, esso può verificare se questo viene utilizzato in modo coerente.

3.1 Tipologie di generics

I Generics permettono di creare classi e interfacce parametrizzate rispetto ai tipi.

Possiamo immaginare un parametro di tipo come un segnaposto. La funzione di questo nuovo tipo è quella di poter costruire un programma che non sia vincolato al tipo effettivo che il riferimento "segnaposto" avrà nel programma vero e proprio, bensì parametrico inrispetto ad esso.

Tale programma godrà comunque dell'univocità di questo tipo, quasi fosse un tipo creato localmente alla classe.

A seguito riportiamo le tipiche configurazioni delle dichiarazioni generiche, confrontate con codice equivalente non generico.

3.1.1 Forma classica <A>

La prima forma è quella più classica, laddove prima si scriveva:

```
List l = new ArrayList();
```

ora si potrà scrivere, in forma parametrica:

```
List<Integer> l = new ArrayList<Integer>();
```

dove, l'interfaccia List sarà:

```
public interface List<A> {
        ...
        public void add(A element);
        public A get(int index);
        ...
}
```

Si nota che laddove prima avevamo le dichiarazioni di Object ora è presente il segnaposto A, a cui il compilatore provvederà ad assegnare un oggetto sempre uguale in tutta l'interfaccia, affinché vi si possa passare un qualsiasi oggetto in modo trasparente ed adattabile in ogni estensione.

Vediamo un altro esempio:

```
Coppia c=new Coppia();
```

Adesso potremo scrivere in forma parametrica:

```
Coppia<Integer,Double>=new Coppia<Integer,Double>();
```

dove la classe Coppia sarà così definita:

```
public class Coppia<A,B>{
        A first;
        B second;
        public Coppia(A f, B s){
                this.first=f;
                this.second=s;
        }
        ...
        ...
        ...
}
```

3.1.2 Wildcards

Un importante aspetto da tenere presente quando si definiscono e utilizzano classi generiche è la mancata relazione di sottotipo tra i tipi generici.

Per meglio capire ciò, aiutiamoci con alcuni esempi.

Supponiamo di avere le seguenti definizioni di classi:

```
class G<T>{};
class Bar{};
class Foo extends Bar{};
```

possiamo asserire che , mentre è vero che Foo è un sottotipo di Bar e possiamo assegnare ad un riferimento di tipo Bar un riferimento di tipo Foo, non è altrettanto vero che G<Foo> è un sottotipo di G<Bar> e pertanto l'assegnamento di un oggetto di tipo G<Foo> non è consentiito a un riferimento di tipo G<Bar>.

Nello stesso modo, possiamo asserire che un ArrayList<Number> non è mai una superclasse di ArrayList<Integer> anche se un Integer è un sottotipo di Number e lo si può assegnare ad esso. Ciò significa che, se volessimo scrivere un metodo che sommi tutti gli elementi di un ArryList<Integer> e poi di un ArrayList<Double>, non potremmo scrivere il metodo medesimo con una segnatura che avesse come parametro un ArrayList<Number> perchè, ripetiamo, la relazione di ereditarietà di un ArrayList di Integer o di un Double verso un ArrayList di tipo Number non è ammessa, mentre, se lo fosse, potremmo assegnare, per esempio, senza la generazione di errori a compile-time, un riferimento di tipo ArrayList<Integer> a un riferimento di tipo ArrayList<Number> e poi tramite quest'ultimo aggiungere alla lista anche numeri di tipo double.

Tuttavia, a runtime avremmo un'eccezione di tipo ClassCastException perchè abbiamo tentato di aggiungere dei numeri in virgola mobile ad una lista che poteva contenere solo numeri interi.

L'esempio precedente ha mostrato che non ha senso cercare una compatibilità generale fra tipi parametrici, perché non può esistere.

Ha senso invece cercare compatibilità fra casi specifici e precisamente fra tipi di parametri di singoli metodi.

Perciò, alla normale notazione dei tipi generici List<T>, usata per creare oggetti, si affianca una nuova notazione, pensata esplicitamente per esprimere i tipi accettabili come parametri in singoli metodi.

Si parla quindi di tipi parametrici varianti, in Java più brevemente detti WILDCARD.

Vediamo le tre notazioni wildcard.

Posto che la notazione List<T> indica il normale tipo generico, abbiamo:

- il tipo covariante <? extends T>

  Cattura le proprietà dei List<X> in cui X estende T; si usa per specificare tipi che possono essere solo letti;

- il tipo controvariante <? super T>

  Cattura le proprietà dei List<X> in cui X è esteso da T; si usa per specificare tipi che possono essere solo scritti;

- il tipo bivariante <?>

  Cattura tutti i List<T> senza distinzione; si usa per specificare tipi che non consentono né letture né scritture, ma che possono servire comunque per altri fini.

Invarianza, Covarianza, Controvarianza

Il tipo wildcard consente di aggirare elegantemente la limitazione descritta precedentemente di mancanza di relazione di ereditarietà tra i tipi generici, definita come invarianza.

Esempi di utilizzo di wildcard

- ```
  public class MyList<T> {
          private T head;
          private MyList<T> tail;
          public T getHead(){ return head; }
          public <E extends T> void setHead(E element){
                  head=element; }
  }

  MyList<Number> list1 = new MyList<Number>();
  MyList<Integer> list2 = new MyList<Integer>();
  list1.setHead( new Double(1.4) ); // OK!
  list1.setHead( list2.getHead() ); // OK!
  ```

  Poiché head è di tipo T, come element va bene qualunque tipo E più specifico di T.

  Nel caso di list1, che è lista di Number, si può quindi mettere nella head qualunque tipo più specifico di Number.

- ```
  public class MyList<T> {
          private T head;
          private MyList<T> tail;
          public T getHead(){ return head; }
          public <E extends T> void setHead(E element)
               {...}
          public void setTail(MyList<T> l){ tail=l; }
  ```

```
        public MyList<? extends T> getTail(){
                return tail; }
}

MyList<? extends Number> list3 = list1.getTail();
MyList<? extends Number> list4 = list2.getTail();
MyList<? extends Integer> list5 = list2.getTail();
```

Il metodo getTail restituisce una lista di elementi di tipo almeno T; nel codice d'esempio le prime due chiamate del metodo getTail restituiscono una lista di Number, che è compatibile col tipo lista di qualcosa che estenda Number, l aterza chiamata di getTail restituisce una lista di Integer, che è compatibile col tipo lista di qualcosa che estenda Integer.

3.2 Vantaggi dei generics

L'introduzione dei generics ha portato con sè due importanti benefici:

- Type Safety.

  Come già detto, i generici nascono appunto per tentare di garantire già a compile time la verifica di vincoli sui tipi.

  Infatti conoscendo i vincoli di tipo richiesti ad una variabile il compilatore è capace di verificare (ed in un certo senso forzare) le assunzioni fatte, offrendo un controllo ulteriore sulla situazione a runtime. L'aver spostato il controllo dei tipi dal runtime al compile-time ha permesso di poter individuare errori molto più facilmente ed in modo più affidabile.

- Eliminazione dei cast.

  Avendo già le garanzie sul tipo ritornato si riduce drasticamente l'utilizzo dei casting espliciti ad opera dell'utente.

  In realtà i casting permangono, ma sono sicuri, poiché fatti dal compilatore a seguito della verifica dei tipi.

  Si possono anche effettuare casting o analisi di tipo(instanceof), ma solo nel caso in cui i vincoli a compile-time possano garantire la completa corrispondenza a runtime del tipo con quanto dichiarato.

Capitolo 4

Insiemi e liste logiche come collezioni standard

In questo lavoro di tesi vogliamo analizzare l'attuale implementazione di insiemi e liste logiche in JSetL e quindi proporre alcune implementazioni alternative che permettano di realizzare operazioni di base su queste strutture dati che risultino:

- piu' "pulite" dal punto di vista della struttura interna dell'implementazione (ad esempio, con l'uso di iteratori per scandire le strutture dati interne piuttosto che semplici cicli ad-hoc);
- piu' "complete", dal punto di vista delle operazioni fornite, in modo che siano presenti le analoghe di tutte le operazioni previste per Set e List di Java;
- piu' "controllate", con l'uso dei generics di Java piuttosto che collezioni di Object;
- ultimo, ma non meno importante, piu' efficienti per quanto riguarda i tempi di esecuzione delle principali operazioni.

L'approccio che intendiamo seguire prevede di realizzare due classi, che indicheremo con LSet0 e LList0, che conterranno le strutture dati di base di insiemi e liste logiche e tutte e sole le operazioni di base su insiemi e liste logiche completamente specificate. Le classi LSet0 e LList0 verranno poi usate come classi base delle classi "standard" di JSetL, LSet e LList.

Queste ultime si limiteranno ad estendere LSet0 e LList0 fornendo tutte le altre funzionalita' necessarie per la gestione di collezioni parzialmente specificate, come ad esempio la possibilita' che nelle operazione su insiemi/liste uno degli oggetti coinvolti sia non inizializzato (con il conseguente lancio dell'eccezione NotInitilizedVar), oppure i constraint su insiemi e liste.

Le classi LSet0 e LList0 saranno definite utilizzando i generics di Java, nell'ipotesi che anche le classi LSet e LList possano essere definite in questo modo in futuro. In realta', come gia' previsto nell'attuale JSetL, le classi LSet0 e LList0 saranno definite come sottoclassi di una classe comune, la classe LCollection0, che quindi fornisce un super-tipo comune ad entrambe.

Chiameremo JSetL0 la versione della libreria JSetL che contiene le nuove classi LCollection0, LList0 e LSet0.

Vediamo la sua struttura gerarchica:

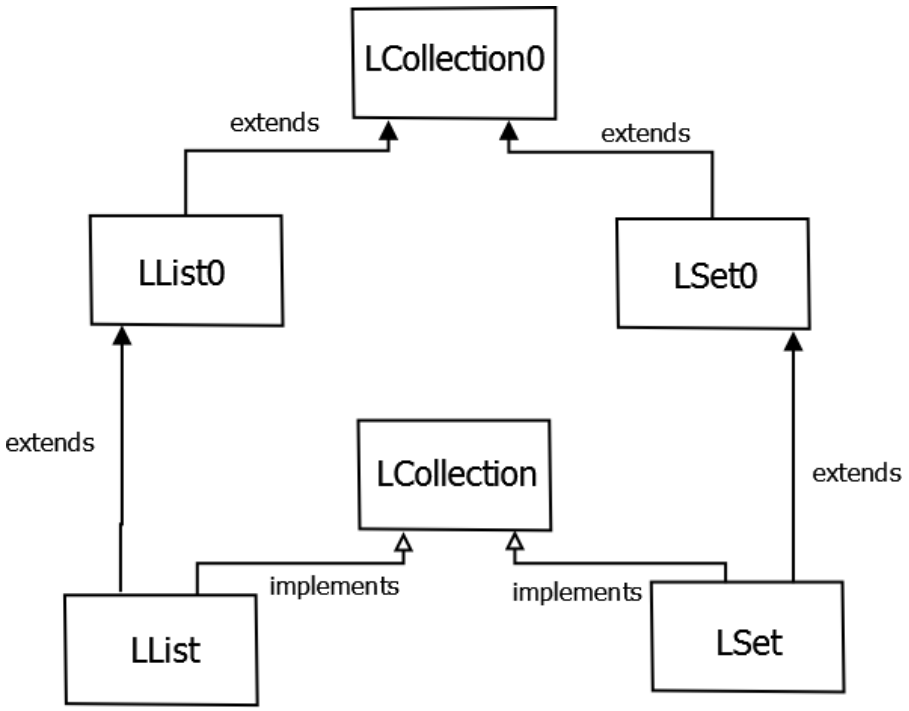

Figura 4.1: JSetL0 hierarchy

4.1 LCollection0<T>: una classe generica

Una delle più significative modifiche che la nostra classe LCollection0 ha apportato alle LCollection di JSetL è la possibilità di parametrizzare i tipi gestiti nella collezione, come abbiamo anche potuto notare dagli esempi della sezione precedente.

Infatti, LCollection0 è una classe parametrica (o generica) e quelli che implementa sono metodi generici.

Quindi, laddove prima (in JSetL) avevamo le dichiarazioni:

```
LList l = new LList();
LSet s = new LSet();
```

adesso (in JSetL0) possiamo dichiarare:

```
LList0<String> l = new LList0<String>();
LSet0<Integer> s = new LSet0<Integer>();
```

In questo modo, viene garantita la verifica sui tipi degli elementi gestiti nelle nostre collezioni già a tempo di compilazione.

Con le LCollection di JSetL non era possibile verificare a tempo di compilazione i tipi di dato inseriti in una collezione, con la conseguente possibilità di ritrovarsi a gestire collezioni completamente disomogenee.

Prendiamo, per esempio, il metodo ins, nelle collezioni di JSetL:

```
LList0 l1=new LList0();
l1.ins(1).ins(2).ins(3.4).ins('a');
```

Questo insieme è disomogeneo, non vi è alcun controllo a compile-time sui tipi di elementi inseriti nella lista l1.

Con la versione generica di LCollection0, invece, verifico l'omogeneità degli elementi già a tempo di compilazione, infatti, scrivendo il codice:

```
LList0<Integer> l1=new LList0<Integer>();
l1.ins(1).ins(2).ins(3.4).ins('a'); // Errore!
```

questo genererà un errore proprio in corrispondenza del tentativo di inserire dei double e dei char in una lista che si aspetta solo degli interi.

In questo modo, avendo già delle garanzie sui tipi, viene ridotto l'utilizzo dei casting ad opera dell'utente; in realtà, i casting permangono ma sono sicuri perchè fatti dal compilatore a seguito della verifica sui tipi.

Il metodo statico empty() come metodo generico E' molto interessante notare il comportamente dei metodi statici nella loro versione generica.

Prendiamo, come esempio, proprio il metodo empty() presente sia in JSetL che in JSetL0 con cui è possibile creare una collezione inizializzata vuota.

Vediamolo nella nostra LList0:

```
public static <T> LList0<T> empty() {
        LList0<T> emptyLList0 = new LList0<T>();
        emptyLList0.init = true;
        emptyLList0.name = "empty";
        return emptyLList0;
}
```

Per creare una lista vuota in JSetL0, la sintassi da utilizzare è la seguente:

```
LList0<Integer> l=LList0.<Integer>empty();
```

Questa particolare sintassi sta a significare che, in Java, il tipo generico della classe generica (nel nostro caso LCollection0) e i tipi generici dei suoi metodi statici non sono la stessa cosa.

Se provassimo a creare una lista vuota in JSetL0 nel seguente modo:

```
LList0<?> l=LList0.empty(); // Creo una lista l
    inizializzata vuota
```

non avremmo alcun problema, in quanto il metodo empty restituisce un LList0<Object> che si può "castare" a LList0<?>.

Se, invece, scrivessimo:

```
LList0<Integer> l=LList0.empty(); // Errore!
```

questo codice genererebbe un errore proprio dovuto al fatto che il tipo LList0<Object> restituito dal metodo empty() non è "castabile" nel tipo LList0<Integer>.

4.2 LCollection0: struttura

L'implementazione di insiemi e liste logiche in JSetL si basa su una struttura dati mista che prevede di memorizzare i dati della collezione in parte in un Vector (campo dElems), e quindi in una struttura dati ad accesso diretto, e in parte in una lista concatenata (campo rElems), e quindi in una struttura dati ad accesso sequenziale. Questa struttura è implementata anche in LCollection0 (e quindi anche nelle sue sottoclassi LSet0 e LList0).

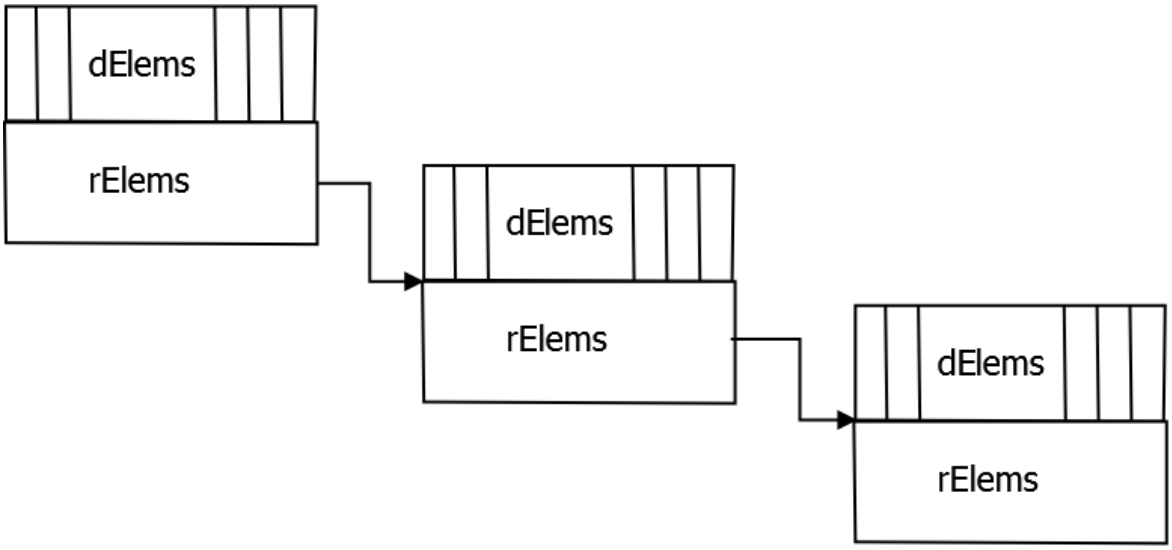

Figura 4.2: memorizzaione dei dati in JSetL0

Vediamo ora le caratteristiche principali della classe LCollection0.

4.2.1 Dati membro

- Una sequenza di elementi contenuti nella collezione che è un'instanza della classe Vector<T> inizialmente inizializzata a null:

```
protected Vector<T> dElems=null;
```

- Una parte resto che è un'istanza della classe LCollection0<T>:

```
protected LCOollection0<T> rElems;
```

4.2.2 Creazione di nuove LCollection0

E' possibile creare nuove istanze della classe LCollection0 tramite i seguenti costruttori:

```
public LCollection0<T>(){
                this.dElems=null;
                this.rElems=null;

        }

        public LCollection0<T>(LCollection0<T> lc){
                this.dElems=lc.dElems;
                this.rElems=lc.eElems;
        }
```

Il primo crea un nuovo oggetto che rappresenta la collezione vuota, il secondo crea un oggetto avente i campi uguali ai corrispettivi campi dell'oggetto passato come parametro.

E' anche possibile creare nuovi oggetti partendo da oggetti esistenti, utilizzando i vari metodi di inserimento, simili a quelli di JSetL.

Essi permettono di creare un nuovo oggetto partendo da un oggetto già creato e inserendo uno (metodo ins(element)) o più elementi (metodo insAll(a)). In questi casi, il campo rElems del nuovo oggetto creato conterrà un puntatore all'oggetto di partenza.

Vediamo alcuni esempi di possibili creazioni di nuove liste e insiemi:

- ```
  LList0<Integer> l1=new LList0<Integer>();
  l1=l1.ins(1);
  LList0<Integer> l2=l1.ins(2);
  ```

  In questo modo, creiamo una lista vuota l1; inseriamo, quindi, al suo interno l'elemento 1 cosicchè, dopo tale inserimento, l1 sarà inizializzato e conterrà il valore 1 nel suo campo dElems, mentre il suo campo rElems punterà ad un insieme vuoto.

  L'oggetto l2, a seguito della creazione e dell'inserimento sopra esposti, conterrà il valore 5 nel suo campo dElems, mentre il suo campo rElems punterà a l1.

- ```
  LList0<Integer> l1=LList0.<Integer>empty().ins(1);
  LList0<Integer> l2=l1.ins(2);
  ```

In questo caso, l1, a seguito della sua creazione tramite il metodo statico empty() e del successivo inserimento dell'elemento 1, conterrà nel campo dElems il valore 1 e il suo campo rElems punterà ad un oggetto vuoto; l2,creato a partire da l1, conterrà il valore 2 nel suo campo dElems e il suo campo rElems punterà a l1.

- Sia a=[1,2,3] un array creato precedentemente.

```
LList0<Integer> l1=new LList0<Integer>();
LList0<Integer> l2=l1.insAll(a);
```

In questo caso, nella lista l2 tutti gli elementi saranno inseriti in un colpo solo nel campo dElems tramite il metodo insAll al quale viene passato l'array a; il campo dElems di l2 conterrà quindi gli elementi 1, 2 e 3 e il suo campo rElems punterà alla lista vuota l1.

Dunque, come visto per LSet e LList nel capitolo 2.5, quali elementi di una insieme/lista logica vengano memorizzati nel Vector e quali nella lista concatenata dipende dal modo in cui l'utente costruisce l'insieme/lista nel suo programma.

Se l'insieme/lista e' costruito fornendo tutti gli elementi in un colpo solo, ad esempio tramite una insAll (che vedremo in uno dei successivi esempi), allora gli elementi verranno tutti memorizzati nel Vector; se invece gli elementi vengono forniti uno alla volta, allora verranno memorizzati in una lista formata da piu' insiemi/liste logiche concatenate tramite i loro campi rElems.

Ovviamente l'implementazione di insiemi e liste logiche garantisce che il comportamento funzionale esterno di queste strutture dati sia indipendente dal modo in cui gli elementi vengono memorizzati.

4.3 L'iteratore di LCollection0: la classe LCollection0Iterator

Uno degli obiettivi del nostro progetto è anche quello di arrivare ad avere strutture dati piu' "pulite" dal punto di vista della struttura interna dell'implementazione.

Tale risultato è ottenibile, ad esempio, con l'uso di iteratori per scandire le strutture dati interne piuttosto che semplici cicli ad-hoc.

All'interno della nostra classe LCollection0, infatti, abbiamo implementato la classe LCollection0Iterator, che costituisce proprio un iteratore per la collezione.

LCollection0 implementa la versione parametrica dell'interfaccia Iterable.

L'iteratore per LCollection0 è ottenuto chiamando il metodo iterator() sulla nostra collezione, il quale restituirà un oggetto di tipo LCollection0Iterator.

La classe LCollection0Iterator è una classe interna a LCollection0 e, a sua volta, implementa la versione parametrica dell'interfaccia Iterator.

Vediamola nel dettaglio:

```
public class LCollection0<T> implements Cloneable, Iterable<
    T> {
/*** DATA MEMBERS ***/
...
...
...

/*** CONSTRUCTORS ***/
...
...
...

/*** PUBLIC METHODS ***/
...
public Iterator<T> iterator(){
        LCollection0Iterator i=new
        LCollection0Iterator();
        return i;
}
...

/***PROTECTED/PRIVATE METHODS***/

//Classe LSet0Iterator
private class LCollection0Iterator implements Iterator<T>{
private int current=0;
private LCollection0<T> s;

public LCollection0Iterator(){
s=LCollection0.this;
}

public boolean hasNext(){
if(s.dElems == null && s.rElems==null)
        return false;
else
```

```
        if(s.dElems.size()>current)
                return true;
                        else
                                if(s.rElems==null)
                                        return false;
                                else
                                        if(s.rElems.dElems
                                            == null)
                                                return false
                                                    ;
                                        else
                                                return true;
}

public T next() throws NoSuchElementException{
        if(!hasNext())
                throw new NoSuchElementException();
        T element =null;
        if(current == s.dElems.size()){
                current=0;
                        s=s.rElems;
        }
        element = (T)s.dElems.get(current);
        current++;
        return element;
        }


public void remove() throws UnsupportedOperationException{
        throw new UnsupportedOperationException();
}

public LCollection0<T> getLast(){
        return (LCollection0<T>)s.rElems;
}
}
}
```

Dati membro

I dati membro di questa classe sono rappresentati da current, che è un intero e da s, un'istanza della classe LCollection0.

Dato che, come sappiamo, la nostra collezione può essere costituita da una concatenazione di insiemi o liste, s rappresenta proprio l'oggetto logico (insieme o lista) corrente su cui siamo posizionati in un dato momento dell'iterazione; esso, al momento della chiamata di iterator(), sarà inizializzato al primo oggetto della concatenazione.

L'intero current indica la posizione corrente (ovvero la posizione in un dato momento dell'iterazione) all'interno del campo dElems dell'oggetto logico identificato da s. Al momento della chiamata di iterator(), il campo current viene inizializzato a 0, ad ogni avanzamento dell'iteratore (cioè ad ogni chiamata del metodo next(), che descriveremo fra poco) sarà incrementato di 1 e, ad ogni nuovo assegnamento di s, cioè ad ogni 'spostamento' dell'iteratore in un nuovo oggetto della collezione, ripartirà da 0.

Capiremo meglio il meccanismo di assegnamento a s e a current tra poco, cioè quando descriveremo i metodi hasNext() e next().

Costruttore

La classe LCollection0Iterator prevede un unico costruttore che andrà ad inizializzare il campo s col primo oggetto logico della concatenazione.

Metodi pubblici

- hasNext() Ritorna un booleano che indica se, rispetto alla posizione in cui ci trovimo in un preciso momento dell'iterazione sulla collezione logica, sono presenti altri elementi; ritorna true se è presente un altro elemento, ritorna false altrimenti.

  Vediamo i casi in cui questo metodo restituirebbe false:

  - L'oggetto corrente della concatenazione è non inizializzato;
  - L'oggetto corrente della concatenazione è inizializzato ma sia il suo campo dElems che il suo campo rElems sono uguali a null e cioè è l'insieme vuoto;
  - La dimensione del campo dElems dell'oggetto corrente è maggiore di current e, contemporaneamente, il suo campo rElems è uguale a null;
  - La dimensione del campo dElems dell'oggetto corrente è maggiore di current, il suo campo rElems è diverso da null ma il campo dElems dell'oggetto puntato da rElems è uguale a null.

  In tutti gli altri casi, il metodo restituisce true.

- next() Ritorna l'elemento appena passato dall'iterazione e si posiziona su quello successivo, se presente. E' strettamente correlato al metodo hasNext(), infatti, nel caso hasNext() restituisse false (quindi non ci sarebbero più elementi nella collezione), una successiva chiamata di next() solleverebbe un'eccezione del tipo NoSuchElementException.

  La prima istruzione del metodo next() è proprio la verifica della presenza di ulteriori elementi e quindi dell'eventuale sollevamente della suddetta eccezione.

Una volta verificato che la collezione contiene altri elementi non ancora iterati, il metodo next() controlla se è stata scandito tutto il campo dElems dell'oggetto corrente; per verificare questo, next() non fa altro che confrontare il valore di current con la dimensione del campo dElems dell'oggetto corrente.

Se questo confronto dà esito positivo (ovvero se abbiamo scandito tutto il Vector), current viene nuovamente inizializzato a 0 e s viene inizializzato all'oggetto puntato dal campo rElems dell'oggetto appena scandito.

E' fondamentale notare il fatto che i metodi hasNext() e next() sono fortemente correlati e vanno usati in modo 'coerente', ovvero è necessario verificare con hasNext() la presenza di ulteriori elementi prima di ogni chiamata di next().

- getLast() Ritorna l'ultimo oggetto della concatenazione.

Vediamo un semplice esempio per meglio capire l'utilizzo dei metodi sopra descritti.

```
LSet0<Integer> s1 =

LSet0.<Integer>empty().ins(1).ins(2).ins(3);
LSet0<Integer> s2 = s1.ins(4).ins(5).ins(6);
Iterator it = s2.iterator();
while(it.hasNext())
        Integer element=it.next();
```

In questo esempio abbiamo creato due LSet, s1 e s2; s1 contiene gli elementi 1,2 e 3, s2 è creato partendo da s1 (che quindi rappresenterà il suo resto) e inserendo gli elementi 4,5, e 6.

Creiamo un iteratore su s2 ed entriamo in un ciclo while che terminerà solo quando saranno stati scanditi tutti gli elementi di s2.

Alla primo iterazione di questo ciclo, s sarà inizializzato con s2 e current a 0; l'iteratore sarà posizionato sul primo elemento (6) e la condizione di uscita non sarà verificata i quanto ci sono ancora altri 5 elementi da iterare. Viene, quindi, chiamato il metodo next() che restituirà l'elemento 6 e si posizionerà sull'elemento 5; current viene incrementato di 1 e s rimane associato a s2.

Si prosegue nello stesso modo fino a quando il metodo next() non verificherà che è stato scandito tutto il Vector di s2 e, bisognerà quindi, passare a scandire s1: il campo current dell'iteratore viene azzerato e al campo s viene assegnato il valore contenuto nel campo rElems di s2, cioè s1.

Una volta giunti all'elemento 1, cioè l'ultimo, il metodo hasNext() restituirà false, quindi sarà stata ottenuta la condizione di uscita dal while e quindi non verrà fatta un'ulteriore chiamata di next().

4.4 Reimplementazione di metodi di JSetL

In LCollection0 vengono reimplementati alcuni metodi di JSetL con lo scopo di migliorarne l'efficienza in termini di memoria.

Queste implementazioni si basano proprio sull'utilizzo dell'iteratore sopra descritto e i metodi in cui questa situazione viene meglio evidenziata sono get, contains e equals.

Il fatto che le classi LSet0/LList0 trattino soltanto collezioni completamente specificate significa che le operazioni da esse fornite non terranno conto di elementi non noti ne' di eventuali parti della collezione non specificate.

In particolare, l'uguaglianza tra insiemi/liste sarà un'uguaglianza "sintattica" e non l'unificazione tra esse. Ad esempio, usando sempre una notazione astratta, dati due insiemi r={ 1,x }, dove x e' una variabile logica non inizializzata e s={ 1,2 }, la r.equals(s) restituira' false mentre l'unificazione tra r ed s, r.eq(s), restituirebbe true con x inizializzato a 1.

Insiemi e liste logiche costituiscono un utile strumento di supporto alla programmazione dichiarativa, ma possono venir utilizzati anche come collezioni "normali", in modo analogo agli insiemi e liste di java.util.

Il metodo get Negli insiemi non ha senso parlare di posizione dell'elemento all'interno della struttura.

Il metodo get, infatti, è implementato solo per le liste dato che restituisce l'elemento che si trova alla posizione i-esima.

Come per il metodo equals(), anche get in JSetL passa per un Vector, ovvero, viene creato un Vector inizializzato col valore ritornato dal metodo toVector chiamato sulla lista di invocazione e, viene sfrutatto il metodo get della classe Vector.

Vediamolo nel dettaglio.

```
public T get(int i) {
        if (i+1 > this.getSize() || this.dElems==null)
        throw new IndexOutOfLimits();

    if (this.rElems.isEmpty())
        return this.dElems.get(i);

    else{
                T element = null;
                int n = this.dElems.size();
                if (i < n){
                    element=this.dElems.get(i);
```

```
                }
                else{
                    i = i-n;

                    element=((LList0<T>)(this.rElems)).get(i
                        );
                }
                return element;
    }

}
```

Nella nostra implementazione, dopo aver verificato che la lista contenga almeno un elemento e che l'indice i sia minore della dimensione della lista, scorriamo gli elementi lungo la concatenazione fino a raggiungere la posizione desiderata.

Il raggiungimento della posizione i avviene nel seguente modo: se i è minore della dimensione del campo dElems dell'oggetto corrente, sfruttiamo il metodo get delle liste Java per estrapolare l'elemento cercato, altrimenti, sottraiamo ad i il valore della dimensione del campo dElems dell'oggetto corrente e l'iteratore si sposterà sul successivo oggetto della concatenazione e ripeterà il tutto.

Il metodo equals Il metodo equals ritorna un booleano che indica se la collezione passata come parametro e la collezione di invocazione sono uguali (true se lo sono, false altrimenti).

Naturalmente, questo metodo sarà implementato sia per le liste che per gli insiemi dato che, il criterio per stabilire l'uguaglianza tra liste e l'uguaglianza tra insiemi è differente.

In JSetL, equals viene implementato memorizzando il contenuto dei due oggetti da confrontare in due Vector chiamando il metodo toVector.

In JSetL0, per stabilire l'uguaglianza tra due collezioni, sfruttiamo il nostro iteratore anzichè chiamare il metodo toVector.

Vediamo l'implementazione del metodo equals in LList0<T> per le liste:

```
public boolean equals(LList0<T> l){
                if(this.dElems== null && l.dElems==null)
                        return true;
                else{
                    boolean b=true;
                    Iterator<T> i = this.iterator();
                    Iterator<T> k= l.iterator();
                    int j=0;
```

```
                    if (this.getSize() != l.getSize())
                        return false;
                    else{
                            while(i.hasNext() && b){
                                if(!(this.get(j).equals(l.
                                    get(j))))
                                        b=false;
                                i.next();
                                k.next();
                                ++j;
                            }
                    }
                    return b;
        }
}
```

In questo modo, iteriamo le nostre liste solo dopo aver verificato che queste siano non vuote e di dimensione uguale, condizioni per le quali il metodo ritornerebbe immediatamente false.

Durante l'iterazione di entrambe (che non solleverà mai un'eccezione dato che abbiamo verificato prima l'uguaglianza delle dimensioni), ogni elemento di una viene confrontato con il corrispondente elemento dell'altra.

Vediamo il metodo equals per gli LSet0<T>:

```
public boolean equals(LSet0<T> r){
        Iterator<T> is= this.iterator();
        Iterator<T> ir= r.iterator();
        if( r instanceof Set)
                return this.getValue().equals(r);
        else
                if (r instanceof LSet0){
                        if (!is.hasNext() && !ir.hasNext())
                                return true;
                        else
                                return (this.subset(r) && r.
                                    subset(this));
                        }
                        else
                                return false;
}
```

Nel caso degli insiemi, sfruttiamo il metodo subset() per verificare che ognuno dei due insiemi sia un sottoinsieme dell'altro. In caso affermativo, essi sono uguali.

Con queste implementazioni del metodo equals() in JSetL0, salvaguardiamo la memoria evitando la creazione di strutture come i Vector.

Il metodo contains Come per i metodi precedenti, anche il metodo per verificare se un elemento è presente o meno nell'insieme, in JSetL utilizza dei Vector in cui viene memorizzato il valore ritornato dal metodo toVector.

Il corrispettivo metodo in JSetL0 non fa uso di queste strutture dati, bensì utilizza l'iteratore realizzato per LCollection0:

```
public boolean contains (Object o){
        Iterator<T> i = this.iterator();
        boolean f=false;
        while (i.hasNext() && !f){
                if (i.next().equals(o))
                        f=true;
        }
        return f;
}
```

Con il nostro metodo contains, creiamo un iteratore sull'insieme di invocazione col quale lo andremo a scorrere. Sfruttando il metodo equals andremo a verificare se l'elemento passato come parametro è presente oppure no.

4.5 Metodi di List e Set

Come detto in precedenza, le liste e gli insiemi logici possono essere usati anche come collezioni 'normali' e, in quanto tali, devono prevedere le operazioni tipiche delle collezioni del JCF.

Esponiamo, dunque, tali metodi, i quali rappresentano un aggiunta rispetto alle LCollection di JSetL.

Il metodo remove(int i) Il metodo remove(int i) è implementato solo per le liste in quanto restituisce un oggetto contenente i medesimi elementi di quello di invocazione a differenza dell'elemento corrispondente alla posizione i-esima.

Vediamolo nel dettaglio.

```
public LList0<T> remove(int i){
        if (i > this.getSize()-1)
                return this;
        int count=0;
        boolean b=false;
        Vector<T> v = new Vector<T>();
        for(Iterator<T> iter=this.iterator();iter.hasNext()
             ;){
                T elem=iter.next();
                if(count != i)
                        v.add(elem);
                else
                        b=true;
                count++;
    }
        if(!b)
                throw new IndexOutOfLimits();
        LList0<T> l= LList0.<T>empty().insAll(v);
        return l;
        }
```

Il metodo remove(T element) Il metodo remove che prende come parametro un elemento della lista, ritornerà una lista contenente i medesimi elementi di quella di invocazione eccetto la prima occorrenza dell'elemento element.

```
public LList0<T> remove(T element){
                LList0<T> l=new LList0<T>();
                Vector<T> v=new Vector<T>();
                Iterator<T> it =this.iterator();
                for(;it.hasNext();){
                        T elem=it.next();
                        if(elem != element)
                                v.add(elem);
                }

                l.dElems=v;
                return l;
        }
```

Il metodo removeAll() Ritorna una lista vuota in quanto rimuove tutti gli elementi della lista di invocazione.

```
public LList0<T> removeAll(){
        return new LList0<T>();
                }
```

Il metodo sublist(int i, int j) Ritorna una lista contenente tutti gli elementi di quella di invocazione, eccetto gli elementi dall (i+1)-esimo al (j-1)-esimo.

```
public LList0<T> subList(int i, int j){
                LList0<T> ll=new LList0<T>();
                if(i> this.getSize() || j>this.getSize())
                        throw new IndexOutOfLimits();
                else{
                        Vector<T> v=new Vector<T>();
                        for(int k=i;k<j;k++)
                                v.add(this.get(k));
                        ll.dElems=v;
                }

                return ll;

}
```

Il metodo lastIndexOf(T element) Restituisce l'indice corrispondente all'ultima occorrenza dell'elemento element nella lista di invocazione.

```
public int lastIndexOf(T e){
    Vector<T> v = this.toVector();
    boolean b=false;
    int i=v.size()-1;
    while(i>0 && !b){
        if(v.get(i)==e)
                b=true;
        if(!b)
                i--;
    }
    if(!b)
        throw new IndexOutOfLimits();
    else
        return i;
}
```

Capitolo 5

Insiemi e liste normalizzate

5.1 LCollection0: problemi riscontrati

Nel capitolo precedente abbiamo descritto dettagliatamente la classe LCollection0 e le sue sottoclassi LList0 e LSet0 realizzate in questo lavoro di tesi.

La reimplementazione di molti metodi di queste classi rispetto a JSetL aveva anche lo scopo di ottenere una migliore efficienza in termini di memoria.

Infatti, come detto nel capitolo precedente, numerosi metodi di JSetL, si "appoggiano" a strutture dati, quali i Vector, incidendo negativamente sulle prestazioni della memoria.

Basta ricordare, per esempio, il metodo contains di LSet, il quale, prima di provvedere alla ricerca dell'elemento richiesto, memorizzava il contenuto dell'oggetto di invocazione in Vector tramite la chiamata sullo stesso del metodo toVector, con un conseguente spreco della memoria allocata.

Lo stesso valeva per i metodi get delle LList o equals delle LCollection.

Da una successiva analisi sull'efficienza della nostra LCollection0 abbiamo però constatato che questa implementazione ha portato, da una parte, ad un miglioramento delle prestazioni in termini di memoria utilizzata, ma dall'altra ad un possibile peggioramento dell'efficienza dei metodi in termini di tempo e comunque a tempi di esecuzione decisamente inferiori a quelle delle corrispondenti collezioni di java.util.

Questo ci ha portati a realizzare un'implementazione alternativa della classe LCollection0, con lo scopo di rendere più veloce l'esecuzione delle varie operazioni su tali oggetti.

Abbiamo definito queste collezioni, delle collezioni normalizzate proprio per l'operazione preliminare di normalizzazione che viene fatta sull'oggetto

prima di eseguire ogni metodo, come sarà spiegato più chiaramente nelle successive sezioni.

5.2 La classe LCollection0n

La classe LCollection0n è una classe generica che implementa l'interfaccia Clonable ed è, a sua volta, estesa dalle classi derivate LList0n e LSet0n.

In questa nuova implementazione si cerchera' di sfruttare il piu' possibile l'efficienza delle collezioni di java.util nell'implementazione delle nostre liste e insiemi logici.

A questo scopo, i campi dElems di LSet0n e LList0n verranno definiti rispettivamente come Set e List di java.util piuttosto che come semplici Vector come previsto per gli LSet0 e LList0 (e per gli LSet e LList della implementazione attuale).

Inoltre la maggior parte delle operazioni su insiemi e liste logiche effettueranno, se necessario, una fase preliminare di "normalizzazione" della collezione, che consiste nel "recuperare" tutti gli elementi della collezione eventualmente memorizzati nella lista concatenata realizzata attraverso i campi rest, per andarli a memorizzare nei rispettivi Set/List contenuti nei campi dElems della collezione.
Questo al fine di permettere successivi accessi agli elementi della collezione logica in modo significativamente piu' efficiente, cosi' come garantito dall'uso delle collezioni Java.

Vediamo subito alcuni esempi.

L'inserimento di nuovi elementi Come per JSetL0, anche in JSetL0n è possibile creare nuovi oggetti partendo da oggetti esistenti.

I metodi di creazione e inserimento sono gli stessi presenti in JSetL0, anche se il loro 'comportamento' presenta un'importante differenza rispetto alle collezione in JSetL0.

Prendiamo, come esempio, il metodo insAll che prende in ingresso un array di elementi e ritorna un nuovo oggetto partendo dall'oggetto di invocazione ed inserendo nel campo dElems tutti gli elementi dell'array passato.

- Per la classe LList0n:

```
public LList0n<T> insAll(T[] a){
        LList0n<T> c = new LList0n<T>();
    c.init=true;
    c.list=new ArrayList<T>();
    for(int i=0;i<a.length;i++){
        c.list.add(a[i]);
    }
    c.rest=this;
    return c;
}
```

- Per la classe LSet0n:

```
public LSet0n<T> insAll(T[] a){
        ...
        c.list=new HashSet<T>();
        ...
}
```

Per quanto riguarda LList0n, ogni elemento presente nell'array passato verrà aggiunto a prescindere dal numero di occorrenze dello stesso elemento già presenti nella lista proprio perchè il campo dElems è una List di Java e in quanto tale rispetta la definizione di lista; per LSet0n, ogni occorenza di un elemento già presente, verrà ignorata proprio perchè il suo campo dElems è un Java Set e, in quanto tale, non ammette duplicati.

La scelta di usare un Set per il campo dElems di LSet0 e un List per quello di LList fornisce vantaggi soprattutto nella gestione della classe LSet0n perchè garantisce che il campo dElems di ogni elemento della 'concatenazione' non avrà elementi ripetuti e questo è già un piccolo passo verso un miglioramento in termini di tempo di tutti quei metodi che richiedono la scansione di tutta la struttura dati.

In JSetL0 questo non accadeva in quanto il campo dElems era in entrambi i casi (LList0 e LSet0) un Vector, nel quale le ripetizioni sono ammesse.

Con la distinzione del campo dElems nelle due classi derivate da LCollection0n, anche la classe che realizza l'iteratore deve essere implementata in ognuna di esse proprio perchè, al suo interno, viene utilizzo il campo dElems, avente tipo diverso a seconda dell'oggetto logico di appartenenza.

5.3 La normalizzazione delle collezioni

La particolarità delle LCollecetion0n è proprio il fatto che esse vengano 'normalizzate' prima di effettuare molte delle operazioni eseguibili su di esse.

Con il termine 'collezione normalizzata' indichiamo una collezione i cui elementi sono tutti contenuti nel suo campo dElems.

In questo modo è possibile sfruttare tutti i metodi tipici delle Java Collection direttamente sul campo dElems dell'oggetto, con un notevole guadagno in termini di efficienza.
Vediamo nel dettaglio il metodo chiamato per ritornare una collezione fatta proprio nel modo appena descritto.

5.3.1 Il metodo normalize()

Il metodo normalize, implementato sia in LList0n che in LSet0n, restituisce una collezione logica contenente nel suo campo dElems tutti gli elementi della collezione di invocazione.

Se esso viene chiamato su un oggetto inizializzato vuoto, ritorna un oggetto inizializzato vuoto. Se, invece, viene chiamato su un oggetto non vuoto ritorna un oggetto contenente tutti gli elementi dell'oggetto di invoazione nel campo dElems dell'oggetto ritornato.

Vediamo l'implementazione del metodo normalize.

- in LList0n:

```
protected LList0n<T> normalize(){

                if (this.isEmpty())
                        return this;
                LList0n<T> ll=new LList0n<T>();
                ll.dElems=new ArrayList<T>();
                Iterator<T> iter=this.iterator();
                while(iter.hasNext())
                          ll.dElems.add(iter.next());
                ll.rElems=((LList0nIterator)iter).
                     getLast();
                return ll;
}
```

- in LSet0n:

```
public LSet0n<T> normalize(){
        if(this.isEmpty())
                return this;
        LSet0n<T> ls=new LSet0n<T>();
        ls.dElems=new HashSet<T>();
        Iterator<T> iter=this.iterator();
        while(iter.hasNext())
                ls.dElems.add(iter.next());
        ls.rElems=((LSet0nIterator)iter).getLast();
        return ls;

}
```

Il metodo normalize scandisce (tramite l'iteratore descritto dettagliatamente nel capitolo precedente) tutti gli elementi della struttura dati aggiungendo, di volta in volta, l'elemento al campo dElems dell'oggetto ritornato.

Notiamo che, per quanto riguarda gli LSet0n, avendo dichiarato come Set il loro campo dElems, il metodo add dei Java Set invocato su dElems ignorerà ogni ulteriore occorrenza di un elemento già presente in esso.

I metodi che utilizzano normalize()

La prima istruzione di molti dei metodi implementati nelle nostre collezioni è proprio la chiamata del metodo normalize sull'oggetto di invocazione del metodo stesso che crea una nuova collezione normalizzata su cui è possibile poi utilizzare i corrispondenti metodi del Java Collection Framework e migliorare nettamente l'efficienza delle operazioni.

Vediamo alcuni esempi in LList0n.

- Il metodo insn:

```
public LSet0n<T> insn(T o){
        LSet0n<T> ls=new LSet0n<T>();
    ls.dElems=this.normalize().dElems;
    ls.dElems.add(o);
    ls.init=true;
    return ls;
}
```

Questo metodo ritorna un oggetto partendo dall'oggetto di invocazione contenente ; viene subito chiamato il metodo normalize in modo tale da poter sfruttare il metodo add dei Java Set per aggiungere l'elemento o al campo dElems nel quale erano già

contenuti tutti gli elementi nell'oggetto di invocazione grazie alla precedente chiamata di normalize().

- Il metodo intersection di LSet0n

```
public LSet0n<T> intersection(LSet0n<T> r){
        if(!this.init || !r.init)
        throw new NotInitVarException();
    if(this.isEmpty() || r.isEmpty())
        return LSet0n.<T>empty();
    LSet0n<T> hs1=this.normalize();
    LSet0n<T> ls=new LSet0n<T>();
    ls.dElems=new HashSet<T>();
    for(Iterator<T> iter=hs1.iterator();iter.hasNext()
        ;){
        T element=iter.next();
        if(r.contains(element))
                ls.dElems.add(element);
    }
    return ls;
}
```

Questo metodo ritorna un nuovo insieme che rappresenta l'intersezione tra l'insieme di invocazione e l'insieme passato come parametro; viene normalizzato l'insieme di invocazione, dopodichè questo viene scandito utilizzando l'iteratore dei Java Set.

Talvolta può capitare che un oggetto sul quale invochiamo un determinato metodo abbia già tutti i suoi elementi raggruppati nel campo dElems, o perchè già normalizzato precedentemente o perchè creato in modo tale che gli elementi fossero così memorizzati.

A tale scopo, in vari metodi di LList0n e LSet0n viene effettuato un controllo prima dell'eventuale invocazione del metodo normalize in modo tale da verificare che l'oggetto di invocazione venga normalizzato 'inutilmente' con un conseguente spreco di tempo e spazio.

Vediamo, come esempio di quanto appena detto, il metodo get di LList0n.

```
public T get(int i) {
        if (this.dElems == null || i+1>this.getSize())
                throw new IndexOutOfLimits();
        if (this.rElems.isEmpty())
```

```
                return ((ArrayList<T>)this.dElems).get(i);
                        // NEW - this e' gia' normalizzata
        else {
                LList0n<T> c = this.normalize();
            return ((ArrayList<T>)c.dElems).get(i);
        }
}
```

Notiamo che:

1 vi è un controllo per verifiare che la lista sia inizializzata e, in caso contrario, viene sollevata un'eccezione;

2 vi è un controllo per verificare che l'indice passato non sia eccessivo rispetto alla dimensione della lista e, in caso contrario viene sollevata un'eccezione;

3 vi è un controllo per verificare che la lista di invocazione non sia già normalizzata, cioè che tutti i suoi elementi siano raggruppati nel campo dElems e, in caso contrario, viene chiamato direttamente il metodo get delle Java List sul campo dElems dell'oggetto di invocazione.

Capitolo 6

Efficienza delle collezioni in Java, JSetL0 e JSetL0n

6.1 Panoramica sull'efficienza di Java Set e Java List

6.1.1 Java Set: HashSet vs TreeSet vs LinkedHashSet

In base al comportamento che desideriamo ottenere, possiamo scegliere fra TreeSet, HashSet e LinkedHashSet.

Le prestazioni di HashSet sono generalmente superiori a quelle di TreeSet, specialmente per quanto riguarda l'aggiunta e la ricerca di elementi, due fra le operazioni più importanti.

TreeSet ha la caratteristica di mantenere in ordine gli elementi e di mantenere tale ordine nel tempo anche a seguito di eventuali modifiche, quindi è consigliabile utilizzare questa struttura dati solo quando abbiamo la necessità di gestire un Set ordinato oppure se, per qualche ragione, desideriamo accedere direttamente al primo o all'ultimo elemento dell'insieme.

Conviene, invece, usare un HashSet se si hanno prevalentemente accessi diretti agli elementi e il fatto di mantenerli in ordine non è fondamentale ai fini del nostro problema.
Infatti, usando gli HashSet il tempo di accesso ad un elemento è inferiore (costante) rispetto ai TreeSet (logaritmico).

Considerando la struttura interna necessaria a mantenere l'ordinamento e poichè l'iterazione è generalmente l'eleaborazione più ricorrente, questa operazione di norma è più veloce in un TreeSet che in un HashSet.

Riportiamo qui di seguito una tabella dove elenchiamo i principali metodi di HashSet e TreeSet con l'indicazione delle relative complessità.

	Set	
	HashSet	TreeSet
size	O(1)	O(1)
isEmpty	O(1)	O(1)
add	O(1)	O(logn)
contains	O(1)	O(logn)
remove	O(1)	O(logn)

Per quanto riguarda i LinkedHashSet, le operazioni di inserimento sono più onerose rispetto ad un HashSet, tenuto conto del maggiore lavoro sostenuto per supportare la lista collegata al contenitore con hash.

Un LinkedHashSet rappresenta una sorta di via di mezzo tra un HashSet e un TreeSet, evitando l'ordine casuale degli elementi che caratterizza la prima struttura dati, ma evitando anche le scarse prestazioni dei TreeSet.

Vediamo alcuni semplici programmi di test dove verranno messe a confronto le prestazioni di alcune tra le principali operazioni delle tre tipologie di Set sopra descritte.

- Inserimento di un elemento(metodo add(T element))

```
import java.util.*;

public class SetPerformanceAdd {

        public static void main(String[] args) {
                Random r = new Random();

                HashSet<Integer> hashSet = new HashSet<
                    Integer>();
                TreeSet<Integer> treeSet = new TreeSet<
                    Integer>();
                LinkedHashSet<Integer> linkedSet = new
                    LinkedHashSet<Integer>();

                System.out.println("TESTO L'EFFICIENZA
                    DEL METODO add SU HashSet, TreeSet
                    e LinkedHashSet");
                System.out.println();
                long startTime = System.nanoTime();
```

```
                for (int i = 0; i < 1000; i++) {
                        int x = r.nextInt(1000 - 10) +
                            10;
                        hashSet.add(x);
                }

                long endTime = System.nanoTime();
                long duration = endTime - startTime;
                System.out.println("HashSet: " +
                    duration);


                startTime = System.nanoTime();
                for (int i = 0; i < 1000; i++) {
                        int x = r.nextInt(1000 - 10) +
                            10;
                        treeSet.add(x);
                }

                endTime = System.nanoTime();
                duration = endTime - startTime;
                System.out.println("TreeSet: " +
                    duration);


                startTime = System.nanoTime();
                for (int i = 0; i < 1000; i++) {
                        int x = r.nextInt(1000 - 10) +
                            10;
                        linkedSet.add(x);
                }

                endTime = System.nanoTime();
                duration = endTime - startTime;
                System.out.println("LinkedHashSet: " +
                    duration);


        }

}
```

L'output di questo blocco di codice è il seguente:

TESTO L'EFFICIENZA DEL METODO add SU HashSet, TreeSet e LinkedHashSet:

HashSet: 2244768

TreeSet: 3549314

LinkedHashSet: 2263320

Da questo esempio possiamo notare che, per quanto riguarda l'operazione di inserimento, gli HashSet sono in assoluto i più veloci tra le varie tipologie di Set, mentre i TreeSet, all'aumentare del numero degli elementi divengono sempre più lenti pur offrendo, come già detto, altri vantaggi.

- Iterare sugli elementi (metodo iterator())

```
import java.util.*;


public class SetPerformanceIter {

        public static void main(String[] args) {
                Random r = new Random();

                System.out.println("TESTO L'EFFICIENZA
                    DELL'ITERATORE DI HashSet, TreeSet
                    e LinkedHashSet");
                System.out.println();
                Set<Integer> hs=new HashSet<Integer>();
                Set<Integer> ts=new TreeSet<Integer>();
                Set<Integer> lhs=new LinkedHashSet<
                    Integer>();

                //Inserisco degli elementi

                for (int i = 0; i < 1000; i++) {
                        int x = r.nextInt(1000 - 10) +
                            10;
                        hs.add(x);
                        lhs.add(x);
                        ts.add(x);
                }

                //Itero sugli elementi

                long startTime = System.nanoTime();
                Iterator<Integer> iterHs=hs.iterator();
                long endTime = System.nanoTime();
                long duration=endTime-startTime;
```

```
                System.out.println("HashSet: " +
                    duration);


                startTime = System.nanoTime();
                Iterator<Integer> iterTs=ts.iterator();
                endTime = System.nanoTime();
                duration=endTime-startTime;
                System.out.println("TreeSet: " +
                    duration);

                startTime = System.nanoTime();
                Iterator<Integer> iterLhs=ts.iterator()
                    ;
                endTime = System.nanoTime();
                duration=endTime-startTime;
                System.out.println("LinkedHashSet: " +
                    duration);


        }

}
```

L'output prodotto è il seguente:

```
TEST SULL'EFFICIENZA DELL'ITERATORE DI HashSet,
    TreeSet e LinkedHashSet

HashSet: 523058
TreeSet: 460503
LinkedHashSet: 2887
```

Questo codice evidenzia, dunque, quanto detto sopra, ovvero che iterare su un LinkedHashSet è decisamente meno costoso che iterare su un HashSet o su un TreeSet.

6.1.2 Java List: ArrayList vs LinkedList vs Vector

L'implementazione di una lista con ArrayList consente, rispetto all'implementazione tramite una LinkedList, di aumentare manualmente la capacità di contenimento minima dell'array e di impostare una capacità pari al numero di elementi presenti.

In ogni caso, a parte la differenza appena indicata, la decisione se scegliere l'una o l'altra implementazione dipende dalle performance e dalle operazioni più frequentemente utilizzate dall'applicazione sviluppata.

Per esempio, mentre una LinkedList è più efficiente rispetto ad un ArrayList nei casi di inserimento e cancellazione degli elementi, dato che la lista viene modificata agendo solo sul riordinamento dei collegamenti fra i nodi, un ArrayList lo è nelle operazioni di accesso e ottenimento degli elementi tramite l'indicizzazione perchè si procede in modo arbitrario (direttamente alla posizione indicata) e non sequenziale.

Il metodo get(i) di un ArrayList ()che restuituisce l'elemento che si trova alla posizione i) ha costo costante; questo, invece, non vale per la rimozione di un elemento dato che essa comporta la riorganizzazione di tutti gli elementi rimasti all'interno della lista.

Con le LinkedList, invece, per accedere ad un elemento è necessario iterare la lista, con un conseguente costo pari ad O(n).

Riportiamo qui di seguito una tabella dove elenchiamo i principali metodi di ArrayList e LinkedList con l'indicazione delle relative complessità.

	List	
	ArrayList	LinkedList
add	O(1)	O(1)
contains	O(n)	O(n)
remove	O(n)	O(n)
get,set	O(1)	O(n)
addFirst,addLast,removeFirts,removeLast	O(1)	

Per quanto riguarda i Vector, l'inserimento di un elemento aggiuntivo in un Vector la cui dimensione è inferiore alla sua capacità costituisce un'operazione relativamente rapida, mentre l'inserimento di un elemento in un Vector che deve ingrandirsi per poter contenere questo nuovo elemento costituisce un'operazione relativamente lenta.

La strategia di raddoppiare la capacità di un Vector è molto più efficiente rispetto a far crescere ogni volta un oggetto Vector dello spazio di cui necessita per contenere un singolo elemento; ovviamente, però, in tal modo può andare sprecato dello spazio.

Vediamo anche per le List due semplici programmi di test per comparare l'efficienza dei metodi add e get in base alla tipologia di List utilizzata.

Per quanto riguarda il programma di test dell'inserimento, ci limitiamo a esporne l'output dato che il codice è identico a quello mostrato per i Set,

se non, ovviamente, nell'utilizzo di strutture differenti, quali ArrayList, LinkedList e Vector.

- Inserimento di elementi(metodo add(element))

```
TEST SULL'INSERIMENTO IN ArrayList, LinkedList e Vector

ArrayList: 1242925
LinkedList: 785309
Vector: 384955
```

Possiamo ben notare il tempo nettamente inferiore che il metodo add impiega in un LinkedList rispetto ad un ArrayList, come spiegato sopra.

- Ricerca e ottenimento di un elemento in una data posizione (metodo get(i))

```
import java.util.List;
import java.util.Random;
import java.util.Vector;

public class ListPerformanceGet {

        public static void main(String[] args) {
                Random r = new Random();

                System.out.println("Testo il metodo get
                    (i) sulle List");
                List<Integer> al = new ArrayList<
                    Integer>();
                List<Integer> ll = new LinkedList<
                    Integer>();
                List<Integer> v = new Vector<Integer>()
                    ;


                for (int i = 0; i < 1000; i++) {
                        int x = r.nextInt(1000 - 10) +
                            10;
                        al.add(x);
                        ll.add(x);
                        v.add(x);
                }

                long startTime = System.nanoTime();
```

```
                for(int i=0; i<1000; i++){
                        int x = r.nextInt(1000 - 10) +
                             10;
                        al.get(x);
                }
                long endTime = System.nanoTime();
                long duration = endTime - startTime;
                System.out.println("ArrayList: " +
                     duration);

                startTime = System.nanoTime();
                for(int i=0; i<1000; i++){
                        int x = r.nextInt(1000 - 10) +
                             10;
                        ll.get(x);
                }
                endTime = System.nanoTime();
                duration = endTime - startTime;
                System.out.println("LinkedList: " +
                     duration);

                startTime = System.nanoTime();
                for(int i=0; i<1000; i++){
                        int x = r.nextInt(1000 - 10) +
                             10;
                        ll.get(x);
                }
                endTime = System.nanoTime();
                duration = endTime - startTime;
                System.out.println("Vector: " +
                     duration);
        }
}
```

L'output prodotto è:

```
TEST DEL METODO GET SU ArrayList, LinkedList e Vector

ArrayList: 694363
LinkedList: 3847146
Vector: 2257762
```

Come detto sopra, abbiamo mostrato anche tramite un semplice esempio come la ricerca di un elemento in un LinkedList sia decisamente più costosa rispetto a quella in un ArrayList.

6.2 Efficienza: Java Collection, LCollection0 e LCollection0n a confronto

In questa sezione metteremo a confronto, tramite semplici programmi di test, l'efficienza di JSetL0 con JSetL0n e con le Collezioni del Java Collection Framework.

Come abbiamo detto in un capitolo precedente, l'esigenza di reimplementare alcuni metodi delle collezioni di JSetL nasce da questioni di inefficienza in termini di memoria, legata soprattutto al frequente utilizzo di strutture dati come i Vector in cui memorizzare una copia degli elementi della collezione.

Abbiamo così realizzato la libreria JSetL0, la quale, però ha evidenziato sì un miglioramento delle prestazioni in termini di memoria, ma dall'altra parte anche delle problematiche in termini di costo inteso come tempo di esecuzione di varie operazioni.

6.2.1 Inserimento di nuovi elementi

Vediamo un semplice esempio di codice che testa l'inserimento di nuovi elementi in insiemi e liste in JSetL0, JSet0ln e nelle Java Collection

E' importante ricordare che l'inserimento (metodo ins) di nuovi elementi in collezioni logiche non comporta la modifica dell'oggetto di invocazione, mentre nelle Java Collection (metodo add) comporta l'aggiunta dei nuovi elementi nella collezione di invocazione.

In ogni caso, nell'esempio che segue mettiamo a confronto l'efficienza di questi due metodi.

```
import java.util.*;


public class TestIns {

        public static void main(String[] args) {

                LList0<Integer> ll0=new LList0<Integer>();
                LList0n<Integer> ll0n=new LList0n<Integer>()
                    ;
                List<Integer> l=new ArrayList<Integer>();
                LSet0<Integer> ls0=new LSet0<Integer>();
                LSet0n<Integer> ls0n=new LSet0n<Integer>();
                Set<Integer> s=new HashSet();
```

```
System.out.println("INSERIMENTO IN LSet0,
    LSet0n e JAVA SET: CONFRONTO");
System.out.println();
System.out.println("List:");

long startTime=System.nanoTime();
for (int i=0; i<1000; i++){
        ll0=ll0.ins(i);
}
long duration=System.nanoTime()-startTime;
System.out.println("List non normalizzate: "
     + duration + " ns");

startTime=System.nanoTime();
for (int i=0; i<1000; i++){
        ll0n=ll0n.ins(i);
}
duration=System.nanoTime()-startTime;
System.out.println("List normalizzate: " +
    duration + " ns");

startTime=System.nanoTime();
for (int i=0; i<1000; i++){
        l.add(i);
}
duration=System.nanoTime()-startTime;
System.out.println("Java List: " + duration
    + " ns");

System.out.println();
System.out.println("Set:");

startTime=System.nanoTime();
for (int i=0; i<1000; i++){
        ls0=ls0.ins(i);
}
duration=System.nanoTime()-startTime;
System.out.println("LSet non normalizzati: "
     + duration + " ns");

startTime=System.nanoTime();
for (int i=0; i<1000; i++){
        ls0n=ls0n.ins(i);
}
duration=System.nanoTime()-startTime;
System.out.println("LSet normalizzati: " +
    duration + " ns");

startTime=System.nanoTime();
```

```
                for (int i=0; i<1000; i++){
                        s.add(i);
                }
                duration=System.nanoTime()-startTime;
                System.out.println("Java Set: " + duration +
                    " ns");
        }
}
```

L'output prodotto è:

```
INSERIMENTO IN LSet0, LSet0n e JAVA SET: CONFRONTO

List:
List non normalizzate: 1981094 ns
List normalizzate: 2499827 ns
Java List: 357049 ns

Set:

LSet non normalizzati: 485529 ns
LSet normalizzati: 4316833 ns
Java Set: 1555234 ns
```

Da questo test possiamo fare alcune riflessioni: la prima è che, come c'aspettavamo, il metodo add delle Java Collection è il più veloce tra tutti e tre.

Inoltre, possiamo notare che, l'inserimento in una lista non normalizzata è più efficiente rispetto a quello in una lista normalizzata, anche se comunque, la differenza è lieve.

6.2.2 Ricerca e ottenimento di un elemento

Si può parlare di ricerca e ottenimento di un elemento in una data posizione della struttura solo per le liste, in quanto, per esse, l'ordine e le ripetizioni contano, cosa che non si può invece dire per gli insiemi.

Il metodo incaricato a svolgere questo compito è il metodo get(i) implementato sia per LList0 che per LList0n: esso ritorna l'elemento che si trova nella posizione i-esima della lista e, nel caso in cui la lista fosse non inizializzata oppure il valore di i fosse 'inappropriato' (troppo grande rispetto alla dimensione della lista), verrebbe sollevata un'eccezione.

Vediamo un semplice esempio in cui testiamo l'efficienza del metodo get(i) per le liste in JSetL0, JSetL0n e per le Java List.

```
import java.util.*;

public class TestGet {

        public static void main(String[] args) {

                Random r=new Random();

                LList0<Integer> ll0=new LList0<Integer>();
                LList0n<Integer> ll0n=new LList0n<Integer>()
                    ;
                List<Integer> l=new ArrayList();

                System.out.println("TEST SUL METODO get(i)")
                    ;
                System.out.println();


                for (int i=0; i<1000; i++){
                        ll0=ll0.ins(i);
                        ll0n=ll0n.ins(i);
                        l.add(i);
                }

                long startTime=System.nanoTime();
                for(int i=0; i<1000; i++){
                        int x=r.nextInt(1000 - 10) + 10;
                        ll0.get(x);
                }
                long duration=System.nanoTime()-startTime;
                System.out.println("List Non Normalizzate: "
                     + duration);

                startTime=System.nanoTime();
                for(int i=0; i<1000; i++){
                        int x=r.nextInt(1000 - 10) + 10;
                        ll0n.get(x);
                }
                duration=System.nanoTime()-startTime;
                System.out.println("List Normalizzate: " +
                    duration);

                startTime=System.nanoTime();
                for(int i=0; i<1000; i++){
                        int x=r.nextInt(1000 - 10) + 10;
                        l.get(x);
                }
                duration=System.nanoTime()-startTime;
                System.out.println("Java List: " + duration)
```

```
                    ;
        }

}
```

Questo codice produce il seguente output:

```
TEST SUL METODO get(i)

List Non Normalizzate: 44504191240
List Normalizzate: 226931820
Java List: 172269
```

Questo esempio mostra chiaramente il grande miglioramento in termini di efficienza delle liste normalizzate rispetto a quelle non normalizzate; anche in tal caso, comunque, le strutture più efficienti rimangono sempre le Java List.

6.2.3 Uguaglianza fra insiemi e fra liste

Vediamo, ora un test sull'efficienza del metodo equals sia per le liste che per gli insiemi.

Prendiamo come esempio, coppie di oggetti contenenti 1000 elementi e uguali, in modo tale che si renda necessario il confronto fra ogni elemento delle strutture senza interrompere prima la scansione.

Mostriamo solamente l'output del programma di test:

```
TEST SUL METODO EQUALS

List Non Normalizzate: 7061624 ns
List Normalizzate: 303877421 ns
Java List: 47639 ns
Set Non Normalizzati: 131350789699 ns
Set Normalizzati: 559896522 ns
Java Set: 68811 ns
```

Iniziamo col distinguere le liste dagli insiemi.

L'uguaglianza tra due liste ha complessità minore rispetto a quella tra insiemi, dato che basta confrontare ogni elemento della prima col corrispondente elemento della seconda.

Negli insiemi, è invece necessario verificare che ogni elemento del primo sia contenuto nel secondo e che ogni elemento del secondo sia contenuto nel

primo: questo comporta una scansione dell'intera struttura per ogni verifica di contenimento di un elemento.

Di conseguenza, la complessità di equals per gli insiemi logici è quadratica, mentre per le liste è lineare.

Dall'esempio notiamo che, per le liste, il metodo equals è più efficiente in JSetL0 rispetto a JSetL0n: ciò è spiegato dal fatto che, normalizzare le due liste prima di compararle (come avviene appunto in JSetL0n) costa molto di più che iterare una lista e confrontare mammano ogni elemento con quello corrispondente dell'altra lista (come avviene in JSetL0).

Per gli insiemi il discorso è opposto proprio alla luce di quanto spiegato prima: è molto più efficiente normalizzare i due oggetti e sfruttare il metodo equals di HashSet (come avviene in JSetL0n) piuttosto che iterare il primo oggetto e, per ogni elemento contenuto, si itera il secondo oggetto per verificare che l'elemento stesso sia anche qui contenuto (come avviene in JSetL0).

6.2.4 Verifica della presenza di un elemento all'interno della struttura

L'ultimo test che mostriamo è quello sul metodo contains.

Esso ritorna un booleano che indica se l'elemento passato come parametro è presente nell'oggetto di invocazione.

Mostriamone solo l'output:

```
TEST SUL METODO CONTAINS

Set Non Normalizzati: 143989540 ns
Set Normalizzati: 625558 ns
Java Set: 16842 ns
```

Anche in questo caso, notiamo un netto miglioramento di efficienza per gli insiemi normalizzati rispetto a quelli non normalizzati dato che viene sfruttato il corrispondente metodo contains(element) dei Java Set chiamato sull'insieme normalizzato.

6.3 Riflessioni conclusive

Dai nostri test, abbiamo verificato che l'implementazione alternativa delle collezioni logiche basata sulla normalizzazione ha apportato numerosi benefici in termini di prestazioni relativamente a molte operazioni ripetto

alla prima versione, ma per altre si sono comunque rivelati migliori le collezioni non normalizzate.

Una possibile soluzione per massimizzare le prestazioni potrebbe essere quella di realizzare una terza versione di LCollection nella quale, ogni metodo, viene implementato come in JSetL0 o come in JSetL0n, a seconda dei risultati ottenuti dai test di efficienza.

Bibliografia

[1] Gianfranco Rossi, Elio Panegai, Elisabetta Poleo
JSetL: a Java library for supporting declarative programming in Java
Software Practice & Experience 2007; 37:115-149.

[2] F. Bergenti, L. Chiarabini, G. Rossi
Programming with Partially Specified Aggregates in Java
Computer Languages, Systems & Structures, Elsevier, vol. 37/4, 178-192, 2011.

[3] Gianfranco Rossi e Roberto Amadini
JSetL User's Manual
`http://cmt.math.unipr.it/jsetl/JSetLUserManual-v.2.3.pdf`

[4] Stefano Iacinti
Java Generics

[5] JSetL Home Page
`http://cmt.math.unipr.it/jsetl.html`

[6] Bruce Eckel
Thinking in Java 4th Edition Vol. 1-3
Prentice-Hall, 2006.

[7] Pellegrino Principe
Java 7 Guida Completa
Apogeo, 1995.