UNIVERSITÀ DEGLI STUDI DI PARMA

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea in Informatica

Gestione di insiemi ed operazioni insiemistiche in Java tramite l'integrazione tra la libreria JSetL e l'interfaccia Set di Java.

Relatore:
Prof. Gianfranco Rossi

Candidata:
Delia Di Giorgio

ANNO ACCADEMICO 2005-2006

Indice

1 JSetL **3**

1.1 Introduzione . . . 3

1.2 Caratteristiche principali di JSetL . . . 4

1.3 Definizione e uso di variabili logiche e insiemi . . . 5

1.4 I vincoli in JSetL . . . 7

1.5 Un esempio di JSetL . . . 8

2 L'interfaccia `Set` di Java **11**

2.1 Ereditarietà, classi concrete, classi astratte e interfacce in Java 11

2.2 La gerarchia di `Collection` . . . 20

2.3 L'interfaccia `java.util.Set` . . . 21

2.4 Implementazione di `Set` . . . 25

2.4.1 HashSet . . . 25

2.4.2 TreeSet . . . 26

3 Integrazione **28**

3.1 Motivazioni . . . 28

3.2 Architettura . . . 29

3.3 L'interfaccia `LSet` . . . 31

3.4 L'interfaccia `LSetProtected` . . . 32

3.5 L'interfaccia `MutableLSet` . . . 34

4 Implementazione: la classe `ConcreteLSet` **35**

4.1 Attributi . . . 35

4.2 Costruttori 36
4.3 Implementazione dei metodi di `LSet` e `LSetProtected` 40
4.3.1 Metodi di tipo set e get 40
4.3.2 Metodi di vario utilizzo 40
4.3.3 Metodi di generazione di vincoli 42

5 Implementazione: la classe `ConcreteMutableLSet` 45
5.1 Costruttori 46
5.2 Implementazione dei metodi di `LSet` e di `LSetProtected` . . 48
5.3 Implementazione dei metodi di `java.util.Set` 48
5.3.1 Inserimento e unione: `add` e `addAll` 50
5.3.2 Appartenenza e inclusione: `contains` e `containsAll` . 56
5.3.3 Estrazione e differenza: `remove` e `removeAll` 60
5.3.4 Intersezione: `retainAll` 64
5.3.5 Uguaglianza: `equals` 68
5.3.6 `iterator` 71
5.3.7 `size` 74
5.3.8 `isEmpty` 75
5.3.9 `clear` 76

6 Esempi 78
6.1 Utilizzo degli insiemi di JSetL 78
6.1.1 La classe `Coloring` 78
6.1.2 Le modifiche a `Coloring` 80
6.2 Utilizzo degli insiemi di `java.util.Set` 81
6.2.1 La classe `Max` 81
6.3 Un esempio ibrido 82

7 Conclusioni e lavori futuri 85

Capitolo 1

JSetL

1.1 Introduzione

JSetL [1, 2] è una libreria Java che offre alcune funzionalità per il supporto alla programmazione dichiarativa. Questo paradigma di programmazione consiste nello specificare *che cosa* il programma deve fare, piuttosto che *come* farlo, tipico invece della programmazione imperativa. La soluzione viene quindi trovata attraverso un insieme di condizioni e di vincoli e non con una lista di istruzioni da seguire in ordine specifico.
In particolare JSetL fornisce funzionalità tipiche della:

- programmazione logica a vincoli (CLP [4]) tra cui: unificazione, variabili logiche, strutture dati ricorsive, soluzione di vincoli, non determinismo;

- programmazione logica a vincoli con insiemi (ad esempio CLP($\mathcal{SET}$)[5]), tra cui la possibilità di creare insiemi anche specificati parzialmente e di operare su essi attraverso vincoli.

Nei linguaggi logici il programma è costituito da una sequenza di asserzioni (fatti) e regole; l'esecuzione del programma consiste nella dimostrazione della verità di una formula di partenza (il goal). Nella ricerca di tale dimostrazione assumono importanza meccanismi quali il pattern matching (in

italiano potremo tradurre in "combaciamento di forme") e il backtracking (se il sistema durante la ricerca entra in un vicolo cieco, ritorna alla scelta fatta più recentemente e prova ad applicare la regola o il fatto seguente).

I principali vantaggi della programmazione dichiarativa sono: la facilità di sviluppo del programma, la comprensibilità, la riusabilità, la concisione e il parallelismo implicito.

1.2 Caratteristiche principali di JSetL

Vediamo brevemente quali sono le principali caratteristiche della libreria JSetL:

- **Variabili logiche:** hanno lo stesso significato che possiedono nei linguaggi di programmazione logica e funzionale. Possono essere inizializzate o non inizializzate, il loro valore può essere di ogni tipo e può essere determinato come risultato di vincoli che coinvolgono le variabili stesse.

- **Liste e insiemi:** sono strutture dati la cui principale differenza consiste nel fatto che nelle *liste* è importante l'ordine e la ripetizione degli elementi, mentre negli *insiemi* l'ordine e la ripetizione non hanno importanza. Entrambe le strutture dati possono essere parzialmente specificate, cioè possono contenere variabili logiche non inizializzate sia come elementi che come parte della struttura dati.

- **Unificazione:** l'unificazione tra due oggetti può essere usata per testare l'equivalenza tra essi o per assegnare valori alle variabili logiche non inizializzate in essi eventualmente contenute.

- **Vincoli:** uguaglianza, disuguaglianza ed operazioni insiemistiche di base, come differenza, unione, intersezione, sono trattati come vincoli in JSetL. I vincoli vengono in primo luogo aggiunti al *constraint store* e poi risolti tramite il *constraint solver*.

- **Non determinismo:** i vincoli in JSetL vengono risolti in maniera non deterministica, sia perché l'ordine in cui sono risolti non è importante, sia perché nella loro risoluzione si utilizzano i punti di scelta e il backtracking.

- **Vincoli definiti dall'utente:** attraverso la classe *NewConstraints* JSetL permette all'utente di definire nuovi vincoli.

1.3 Definizione e uso di variabili logiche e insiemi

Variabili logiche

Definizione 1.3.1 *Una **variabile logica** è un'istanza della classe `Lvar`, creata dalla dichiarazione*

```
Lvar nomeVar = new Lvar(NomeVarExt, ValoreVar);
```

dove `nomeVar` è il nome della variabile, `NomeVarExt` è un nome esterno (parametro opzionale), `ValoreVar` è un valore (anch'esso opzionale) di inizializzazione della variabile stessa.

Definizione 1.3.2 *Una variabile logica che non ha un valore associato con essa è detta **non inizializzata** (o incognita). Altrimenti la variabile è detta **inizializzata**.*

Il valore di una variabile logica può essere specificato o quando la variabile è creata o come il risultato di elaborazione di vincoli che la coinvolgono, in particolare, vincoli di uguaglianza.
Oltre ai vincoli, la classe `Lvar` fornisce metodi che permettono di leggere e scrivere il valore della variabile logica, di conoscere se la variabile è inizializzata o no, di ottenere il suo nome esterno, e cosi via.

Insiemi

Definizione 1.3.3 *Un **insieme** è una collezione finita di valori (**elementi** dell'insieme). In JSetL un insieme è un'istanza della classe `Set`, creata dalla dichiarazione*

```
Set nomeSet = new Set(NomeSetExt, ValoreSetElem);
```

dove `nomeSet` *è il nome dell'insieme,* `NomeSetExt` *è un nome esterno (paramentro opzionale),* `ValoreSetElem` *è una parte opzionale che è usata per specificare gli elementi dell'insieme e può essere un array di elementi* $c_1, \ldots, c_n$ *di un qualsiasi tipo, o i limiti l e u di un intervallo [l,u] di numeri interi. L'insieme vuoto è denotato dalla costante* `Set.empty`*.*

Gli insiemi possono essere inizializzati o non inizializzati e gli elementi possono essere di qualsiasi tipo. Il valore di un insieme può essere specificato elencando i suoi elementi in un array o, implicitamente, fornendo i limiti di un intervallo di numeri interi, o attraverso un altro insieme.

Esempi di utilizzo di `Lvar` e `Set`

Esempio 1.3.1 *Dichiarazioni di oggetti di* `Lvar` *e* `Set`*.*

```
Lvar x = new Lvar();              //variabile logica non inizializzata
Lvar y = new Lvar("y",'a');       //var. l. inizializzata
                                  //con nome esterno "y" e valore 'a'
Lvar t = new Lvar(x);             //var. l. non inizializzata (come x)
Set z = new Set("z");             //insieme non inizializzato con nome esterno "z"
Set i = new Set(4,4+3);           //insieme inizializzato con valore {4,5,6,7}
int[] s_elems = {2,4,8,3};
Set s = new Set("s",s_elems);     //insieme inizializzato con nome
                                  //esterno "s" e valore {2,4,8,3}
Lvar r = new Lvar(new Set());     //var. l. non inizializzata, il cui valore è
                                  //un insieme non inizializzato
```

Definizione 1.3.4 *Un insieme che contiene alcuni elementi che non sono inizializzati è detto* ***insieme parzialmente specificato****.*

Definizione 1.3.5 *Gli elementi di un insieme che sono essi stessi insiemi sono detti* ***insiemi annidati****.*

Esempio 1.3.2 *Insiemi parzialmente specificati e annidati.*

```
Lvar x = new Lvar();
Object[] pl_elems = {new Integer(1),x};
Set ps = new Set(pl_elems);                 //l'insieme {1,x}
Set[] ns_elems = {i,s,r};                   //i, s e r sono insiemi
                                            //definiti nell'esempio 1.3.1
Set ns = new Set(ns_elems);                 //l'insieme {{4,5,6,7},{2,3,8,3},r}
```

Definizione 1.3.6 *Un insieme è detto **illimitato** se contiene un certo numero di elementi (noti o meno) $e_1, \ldots, e_n$ e un "resto" r, rappresentato da un insieme non inizializzato. La notazione usata è la seguente:*

$$\{e_1, \ldots, e_n | r\}$$

Un insieme illimitato viene costruito tramite i metodi di inserimento `ins` e `insAll`. Vediamo qualche esempio di utilizzo di questi metodi:

Esempio 1.3.3 *Insiemi completamente specificati, parzialmente specificati e illimitati.*

```
Set s1 = Set.empty.ins(1);              // l'insieme {1}
Lvar x = new Lvar();
Set s2 = s1.ins(3).ins(x);              // insieme parzialm. specif. {3,1,x}
int[] s3_elems = {1,2,3};
Set s3 = Set.empty.insAll(s3_elems);    //insieme completam. specif. {1,2,3}
Set r = new Set("r");                   //insieme non inizializzato, r = unknown
Set us = r.ins(1);                      //insieme illimitato {1|r}
```

1.4 I vincoli in JSetL

JSetL fornisce dei vincoli per specificare condizioni su variabili logiche e insiemi. I vincoli attivi di un programma vengono memorizzati nel *constraint store* del *constraint solver S*, un'istanza della classe `SolverClass`. JSetL fornisce i metodi attraverso i quali è possibile aggiungere nuovi vincoli al

constraint store, visualizzarne il contenuto e rimuovere tutti i vincoli presenti. L'introduzione di un nuovo vincolo avviene attraverso il metodo `add` della classe `SolverClass`:

```
S.add(C);
```

che aggiunge il vincolo `C` al constraint store di `S`. Quando tutti i vincoli sono stati aggiunti al constraint store, possiamo risolverli invocando il metodo `solve` (o `boolSolve`) della classe `SolverClass`:

```
S.solve();
```

Questo metodo ricerca, in modo non-deterministico, una soluzione che soddisfi tutti i vincoli introdotti. Se non esiste alcuna soluzione, viene generata un'eccezione `Failure` e la computazione termina. Il metodo `solve` tiene traccia delle alternative rimaste inesplorate durante la ricerca e in caso di backtracking torna ad un punto di scelta precedente che abbia ancora delle alternative aperte e continua la ricerca da quel punto fino a trovare, se esiste, una soluzione. Il metodo `boolSolve` si comporta in modo analogo, ma ritorna `true` se il vincolo è stato risolto con successo e `false` se il risolutore di vincoli non ha trovato nessuna soluzione (a differenza del metodo `solve` non lancia un'eccezione).

1.5 Un esempio di JSetL

Vediamo ora un esempio di utilizzo della libreria JSetL in cui si usano in particolare insiemi e variabili logiche. L'obiettivo del seguente programma è di trovare, in un insieme di interi, quello che ha valore massimo. La soluzione viene trovata in modo dichiarativo, ovvero *un elemento x di s è il massimo di s se per ogni elemento y di s si ha $y \leqslant x$.*

```
class Max {
    static SolverClass Solver = new SolverClass();
    public static Lvar max(Set s) throws Failure {
```

```
        Lvar x = new Lvar();
        Lvar y = new Lvar();
        Solver.add(x.in(s));
        Solver.forall(y, s, y.le(x));
        Solver.solve();
        return x;
    }

    public static void main (String[] args) throws Failure {
        int[] a = {1,6,4,8,10,5};
        Set s = new Set(a);      // S = {1,6,4,8,10,5}
        System.out.print("Max = ");
        max(s).print();
    }
}
```

La prima dichiarazione nella classe `Max` crea un oggetto della classe `SolverClass`, chiamato `Solver`. Il metodo `max` prende un `Set s` come parametro e definisce due variabili logiche `x` e `y` non inizializzate. L'oggetto `Solver` invoca il metodo `add` che aggiunge il vincolo `x.in(s)` ($x \in s$) all'attuale constraint store. Il vincolo restituisce `true` se `s` è un insieme e `x` appartiene ad `s`. Se `x` è non inizializzato quando l'espressione è valutata questo equivale ad assegnare in modo non deterministico un elemento di `s` ad `x`. L'invocazione del metodo `forall` ci permette di aggiungere al constraint store il nuovo vincolo `y.le(x)` ($y \leqslant x$) per ogni `y` appartenente a `s`. Non appena il metodo `solve` è invocato, il constraint solver controlla se l'attuale collezione di vincoli nel constraint store è soddisfacente o no. Se lo è, l'invocazione del metodo `solve` termina restituendo il risultato di `max`. Se, al contrario, uno dei vincoli nel constraint store è valutato `false`, si ha il backtracking e il calcolo torna indietro al punto di scelta più vicino. In questo caso il più vicino e l'unico punto di scelta è quello creato dal vincolo `x.in(s)`. La sua esecuzione legherà in modo non deterministico `x` con ogni

elemento di `s`, uno dopo l'altro. Se tutti i valori di `s` sono stati confrontati, il calcolo di `max` termina sollevando l'eccezione `Failure`. Se l'eccezione non è catturata con un'istruzione `catch`, l'intero programma termina segnalando un fallimento. Eseguendo il programma con il semplice insieme di interi dichiarati nel metodo `main` si ha il risultato `Max = 10`.

Capitolo 2

L'interfaccia `Set` di Java

In questo capitolo illustriamo le caratteristiche principali della classe `Set` di Java e del contesto in cui essa si inserisce. Prima di questo però è opportuno vedere brevemente come funziona l'ereditarietà in Java e cosa sono le classi concrete, le classi astratte e le interfacce.

2.1 Ereditarietà, classi concrete, classi astratte e interfacce in Java

Classi concrete

Una *classe* è un raggruppamento di dati (*variabili membro*) e di funzioni (*metodi*) che operano su questi dati. La definizione di classe in Java si realizza nella seguente forma:

```
[public] class NomeClasse {
    // definizioni di variabili membro
    // costruttori
    // definizioni di metodi
}
```

dove la parola `public` è opzionale: se la si omette, di default la classe sarà visibile solo dalle altre classi del package, altrimenti sarà accessibile da qual-

siasi altra classe. Una classe *concreta* non contiene metodi `abstract` (sezione 2.1) e può essere usata per istanziare oggetti. Un *oggetto* è un esemplare concreto di una classe. In Java esso viene creato attraverso l'operatore `new` seguito da un costruttore della classe. Ad esempio, le istruzioni

```
 NomeClasse oggetto1 = new NomeClasse();
 NomeClasse oggetto2 = new NomeClasse(arg1);
 NomeClasse oggetto3 = new NomeClasse(arg1, arg2);
```

creano tre oggetti di classe `NomeClasse`. Nel primo caso viene richiamato un costruttore senza parametri della classe `NomeClasse` che normalmente crea un oggetto i cui campi dato hanno un valore indefinito, negli altri due casi l'oggetto viene costruito con determinati valori. Per richiamare un metodo della classe si utilizza l'operatore '.' applicato ad un oggetto della classe (*oggetto di invocazione*). Ad esempio:

```
oggetto1.nomeFunzione();
```

richiama il metodo di nome `nomeFunzione`, senza parametri, sull'oggetto `oggetto1`. Vediamo un esempio più concreto:

```
// Definizione della classe A
public class A {
    private int a; //variabile
    public A() {a = 0;} //costruttore senza parametri
    public A(int x) {a = x;} //costruttore con un parametro
    public int getA() {return a;} //metodo
}

// Creazione di due oggetti di tipo A
 A ogg1 = new A(); //la variabile 'a' vale 0
 A ogg2 = new A(3); //la variabile 'a' vale 3

// Chiamata del metodo getA() dall'oggetto ogg2
```

```
int y = ogg2.getA(); // in y viene memorizzato il valore di a,
                     // che in questo caso è 3
```

Ereditarietà

L'ereditarietà è una forma di riutilizzo del software, in quanto le nuove classi vengono create "assorbendo" gli attributi e i metodi di una classe esistente, e sovrascrivendo e migliorando le caratteristiche di quest'ultima in base alle nuove necessità. Quando si definisce una nuova classe, anziché dichiarare per esteso metodi e variabili membro completamente nuovi, si può decidere che la nuova classe *erediti* i membri di una classe (che prende il nome di *superclasse*) già definita. La nuova classe prende il nome di *sottoclasse*. Una sottoclasse (che può a sua volta diventare una superclasse di una classe futura), aggiunge normalmente i propri metodi e variabili di istanza a quelli ereditati dalla superclasse; quindi, una sottoclasse è più specifica rispetto alla sua superclasse e rappresenta un insieme di oggetti più piccolo. Tipicamente, la sottoclasse ha i comportamenti della sua superclasse, ma aggiunge comportamenti specifici.
La *superclasse diretta* è la superclasse da cui la sottoclasse eredita in modo esplicito. In Java l'ereditarietà (diretta) viene espressa nel seguente modo:

```
[public] class NomeSottoclasse extends NomeSuperclasse {
    // corpo della classe
}
```

Una *superclasse indiretta* è invece ereditata da due o più livelli superiori della *gerarchia delle classi*, che definisce le relazioni di ereditarietà tra le classi. In Java, la gerarchia delle classi inizia con la classe `Object` (nel package `java.lang`), da cui ogni classe Java, direttamente o indirettamente, eredita.
Nell'*ereditarietà singola*, una nuova classe può essere derivata da una sola sottoclasse. A differenza del linguaggio C++, Java non supporta l'*ereditarietà multipla* (in cui una classe viene derivata da più superclassi dirette), ma

supporta la nozione di *interfaccia*, che vedremo tra poco. Le interfacce aiutano Java a sfruttare molti dei vantaggi dell'ereditarietà multipla. Vediamo più in concreto come funziona l'ereditarietà in Java con un esempio in cui una classe `Studente` è definita tramite l'ereditarietà diretta da una classe `Persona`:

```
public class Persona {
    private String nome;
    private String cognome;
    public Persona() {...}
    public Persona(String n, String c) {...}
    public setNome(String n) {...}
    public getNome() {...}
    ...
}

public class CurriculumUniversitario {...}

public class Studente extends Persona {
    // Eredita le variabili membro e i metodi della classe Persona.
    // Aggiunge caratteristiche specifiche della classe Studente:
    private int matricola;
    private CurriculumUnivesitario cu;
    public setMatricola(int m) {...}
    public getMatricola() {...}
    ...
}
```

È importante saper distinguere la relazione "è un" dalla relazione "ha un". "È un" rappresenta l'ereditarietà, ovvero un oggetto di una sottoclasse può essere trattato anche come un oggetto della superclasse. Per esempio, "uno Studente *è una* Persona". Invece, la relazione "ha un"

identifica la composizione, cioè un oggetto di una classe ha come membri uno o più oggetti di altre classi. Per esempio, "uno studente *hà un* CurriculumUniversitario".

Classi astratte

In Java una classe *astratta* viene dichiarata tale premettendo la parola chiave `abstract` alla dichiarazione della classe. La caratteristica principale di una classe astratta è che da essa non è possibile creare un'istanza, a differenza delle classi concrete. Una classe astratta ha dei metodi astratti (anch'essi preceduti dalla parola `abstract`), cioè dei metodi che non hanno implementazione; in tal modo si obbliga le classi derivate a fornire un'implementazione adeguata del metodo. Se una classe ha anche solo un metodo `abstract` è obbligatorio che la classe sia `abstract`. In ogni sottoclasse questo metodo dovrà essere ridefinito o dovrà essere a sua volta dichiarato come `abstract` (il metodo e la sottoclasse). Una classe `abstract` può avere metodi che non sono astratti. Anche se non si possono creare oggetti di questa classe, le sue sottoclassi erediteranno il metodo pronto per essere utilizzato. La forma sintattica generale di una classe astratta in Java è la seguente:

```
[public] abstract class NomeClasse {
    // metodi astratti
    abstract tipoDiRitorno1 metodo1(arg1);
     ...
    abstract tipoDiRitornoN metodoM(argN);
    // [metodi non astratti]
    [tipoDiRitorno1 metodo1(arg1) {//corpo del metodo}
     ...
     tipoDiRitornoM metodoM(argM) {//corpo del metodo}]
}
```

Dato che i metodi `static` non possono essere ridefiniti, un metodo astratto non può essere statico.

Vediamo un esempio più concreto:

```
public abstract class A {
    public abstract int f();
    public abstract int g(int x);
    public void h() {...}
    ...
}

public class B extends A {
    int f() {...} //implementazione del metodo f
    int g(int x) {...} //implementazione del metodo g
    // resto della classe
    ...
}
```

Nell'esempio la classe `B` estende `A` implementando i suoi metodi astratti: in questo modo è possibile creare un'istanza di `B`.

Interfacce

Un'interfaccia è un insieme di dichiarazioni di metodi senza la definizione, ovvero senza il corpo che ne specifica l'implementazione. Questi metodi definiscono un tipo di comportamento; tutte le classi che implementano una determinata interfaccia sono obbligate a dare una definizione dei metodi dell'interfaccia e in questo senso acquistano un comportamento o un modo di funzionamento.

In Java la dichiarazione di un'interfaccia è del tutto analoga a quella di una classe, basta sostituire la parola `class` con la parola `interface`; la sintassi è la seguente:

```
[public] interface NomeInterfaccia
                    [extends Interfaccia1,...,InterfacciaN] {
   [static final tipo1 var1 = valore1;
```

```
    ...
    static final tipoN varN = valoreN;]
    tipoDiRitorno1 metodo1(arg1);
    ...
    tipoDiRitornoM metodoM(argM);
}
```

Come si può vedere un'interfaccia specifica solo metodi che devono essere implementati; in Java tutte le interfacce sono pubbliche e tutti i suoi metodi sono implicitamente pubblici ed astratti. Sono anche permessi attributi `public static final` (le costanti), che devono essere necessariamente inizializzate, perché le variabili dell'interfaccia non possono essere modificate. Un'interfaccia può estendere i metodi di un'altra interfaccia, esattamente come accade per le classi.
Una classe può *implementare* una o più interfacce dichiarandole esplicitamente e implementando i metodi dichiarati nelle interfacce stesse. In tal caso, gli oggetti di questa classe saranno riconosciuti anche come oggetti che implementano l'interfaccia. La dichiarazione di implementazione all'interno di una classe si ottiene per mezzo della parola chiave `implements` seguita dal nome di una o più interfacce separate da virgole:

```
class NomeClasse implements Interfaccccia1,...,InterfacciaN {
    //implementazione dei metodi delle interfacce
    //più eventuali altri metodi specifici
}
```

Ad esempio:

```
 public interface I1 {
    public void f();
 }
  public interface I2 {
    public int g();
```

```
}
public class C implements I1,I2 {
    public void f() {...}     //implementazione metodo di I1
    public int g() {...}      //implementazione metodo di I2
    public string s() {...}   //implementazione metodo proprio di C
}
```

Ovviamente però non si può creare un'istanza di un'interfaccia con l'istruzione `new`, per cui quando un oggetto è di tipo corrispondente a un'interfaccia significa che appartiene a una classe che *implementa* quell'interfaccia. Ad esempio:

```
// Creazione di un oggetto di tipo interfaccia (1);
I x1 = new C();
x1.f(); // ok
x1.g(); // errore! Dentro a I non esiste g()

// Creazione di un oggetto di tipo classe concreta (2):
C x2 = new C();
x2.f(); // ok
x2.g(); // ok
```

È importante capire che quando un oggetto è di tipo corrispondente a un'interfaccia, si possono utilizzare solo i metodi dell'interfaccia e non eventuali metodi aggiunti nella classe concreta; per poterli utilizzare tutti bisogna creare l'oggetto come nell'esempio (2).

Una classe può avere solo una superclasse, mentre può implementare un qualsiasi numero di interfacce. In questo modo in Java viene supportata l'ereditarietà multipla. Vediamo un esempio:

```
public interface I1 {
    public static final int a = 10;
    public int f(double x);
```

```
    public void g();
    ...
}

public interface I2 {
    public int p();
}

public interface I3 extends I1 {
    public String h(String s);
    ...
}

public class C1 {...}

public class C2 extends C1 implements I2, I3 {
    public int p() {...}
    public int f(double x) {...}
    public void g() {...}
    public String h(String s) {...}
    // Resto della classe
    ...
}
```

In questo esempio la classe `C2` eredita i membri e i metodi di `C1` e fornisce un'implementazione di tutti metodi delle interfacce `I2` e `I3` e anche dell'interfaccia `I1`, in quanto essa viene ereditata da `I3`.

Differenze tra interfaccia e classe astratta

Un'interfaccia e una classe astratta hanno in comune che possono contenere dichiarazioni di metodi senza fornirne l'implementazione. Nonostante questa

somiglianza, che fa si che in alcune occasioni si posso sostituire una con l'altra, esistono anche alcune differenze importanti:

- Una classe non può ereditare da due classi astratte, pero può ereditare da una classe astratta e implementare una o più interfacce.
- Una classe non può ereditare metodi da un'interfaccia.
- Grazie alla possibilità di implementare più di una interfaccia, queste permettono molta più flessibilità per ottenere che due classi abbiano lo stesso comportamento, indipendentemente dalla posizione nella gerarchia delle classi di Java.
- Le interfacce permettono di "rendere pubblico" il comportamento di una classe dando il minimo di informazioni.
- Le interfacce hanno una gerarchia propria, indipendente e più flessibile delle classi, in quanto è permessa l'ereditarietà multipla.

2.2 La gerarchia di `Collection`

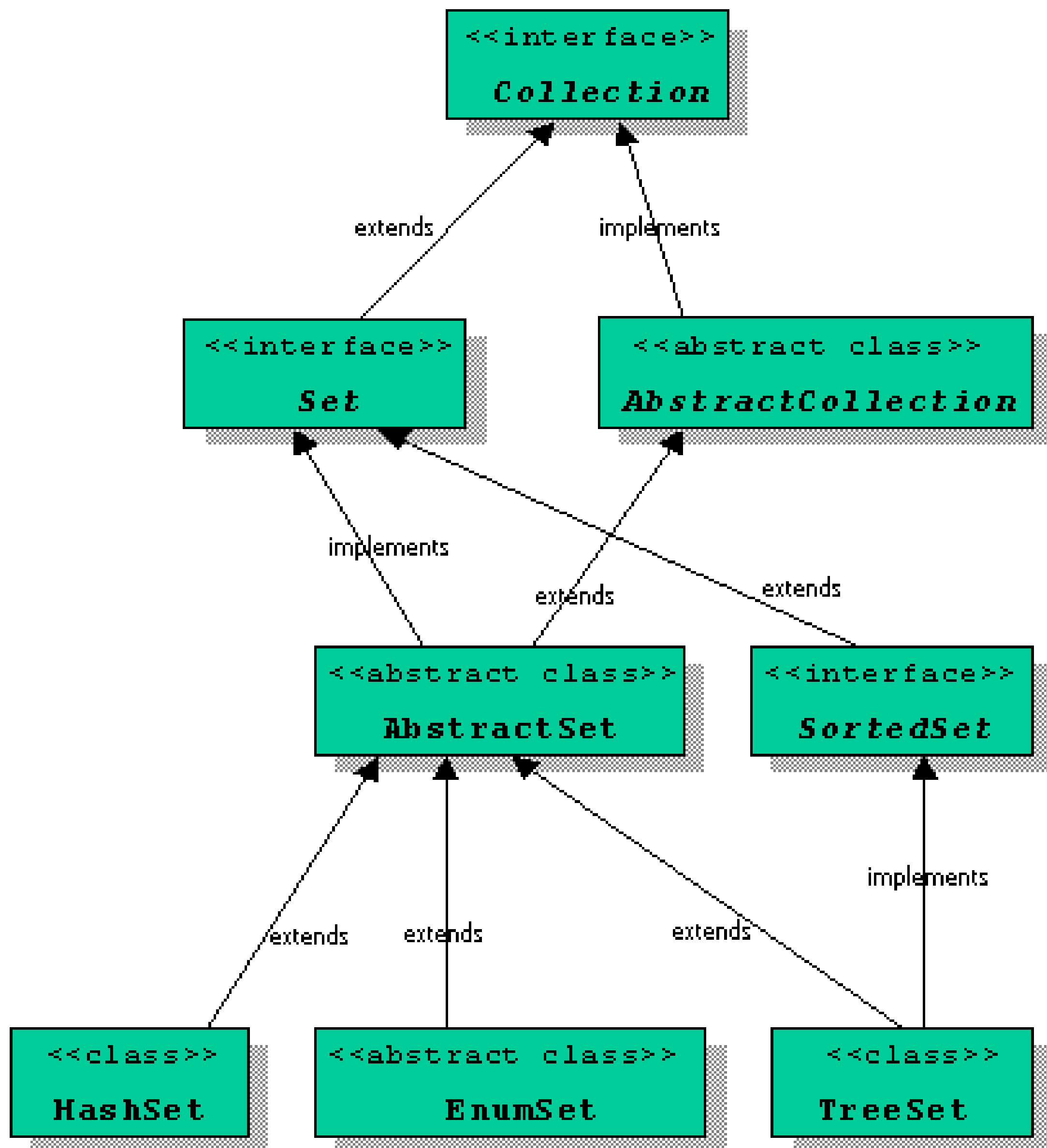
Vediamo ora com'è strutturata la gerarchia di `Collection`, in quanto faremo riferimento a questa parte della libreria Java per inserire la classe `Set` di `JSetL`.

Una collezione rappresenta un gruppo di oggetti (*elementi* della collezione).
Le collezioni si possono suddividere in:

- collezioni che permettono elementi duplicati (come le liste) e collezioni che non lo permettono (come gli insiemi);
- collezioni in cui è importante l'ordine (come le liste) e quelle in cui l'ordine non importa (come gli insiemi).

Java non offre nessuna implementazione concreta dell'interfaccia `Collection`: viene fornita l'implementazione di sottointerfacce più specifiche, come appunto `Set` e `List`. L'interfaccia `Collection` è tipicamente usata per manipolare le collezioni dove è richiesta la massima generalità.

Per capire meglio la gerarchia osserviamo il seguente diagramma di classi che rappresenta il sottoalbero di `Collection` riguardante gli insiemi.



L'interfaccia `Collection` è la radice della gerarchia. `Collection` viene ereditata dall'interfaccia `Set` e implementata dalla classe astratta `AbstractCollection`, che fornisce un'implementazione ridotta di `Collection`, implementando al-

cuni metodi e dichiarando `abstract` i metodi `iterator()` e `size()`.
La classe astratta `AbstractSet` implementa `Set` ed estende `AbstractCollection`: il processo di implementazione è lo stesso che c'è tra `Collection` e `AbstractCollection`, con l'unica differenza che tutti i metodi devono osservare un vincolo aggiuntivo imposto dall'interfaccia `Set`, ovvero rispettare l'unicità degli elementi nell'insieme. In particolare `AbstractSet` sovrascrive il metodo `removeAll(Collection c)` di `AbstractCollection` e aggiunge l'implementazione di `equals()` e `hashCode()`.
L'interfaccia `SortedSet` eredita da `Set` e aggiunge metodi che garantiscono che i suoi elementi si mantegano ordinati secondo l'ordine naturale o secondo un criterio stabilito.
Infine le classi `HashSet` (implementata mediante tabelle hash), `TreeSet` (implementata mediante un albero binario ordinata) e la classe astratta `EnumSet` estendono `AbstractSet`, in particolare `TreeSet` implementa anche `SortedSet`.

2.3 L'interfaccia `java.util.Set`

In generale, un insieme è una collezione in cui:

1. gli elementi duplicati non contano, ad esempio gli insiemi $\{1, 1, 2\}$, $\{1, 2, 2\}$ sono uguali e identificano entrambi l'insieme $\{1, 2\}$;

2. non importa l'ordine degli elementi, ad esempio gli insiemi $\{1, 2\}$ e $\{2, 1\}$ si equivalgono.

L'interfaccia `Set`, che come suggerisce il nome modella l'astrazione matematica di insieme, estende l'interfaccia `Collection` e offre tutti e soli i suoi metodi, con la restrizione che le classi che implementano `Set` si "impegnano" a non ammettere la presenza di elementi duplicati.
Vediamo quali sono i metodi, la loro funzione e un breve esempio per ciascuno di essi:

- `boolean add(Object o)`: aggiunge all'insieme l'oggetto passato come parametro; ritorna `true` se l'insieme è cambiato dopo la chiamata a questo metodo. Ad esempio: $o \in s$

```
Set s1 = new HashSet();
s1.add(1);  // true: s1 = [1]
s1.add(2);  // true: s1 = [1,2]
s1.add(1);  // false: s1 = [1,2]
```

  Si noti che in Java gli insiemi sono rappresentati con le parentesi quadre.

- `boolean addAll(Collection c)`: aggiunge all'insieme tutti gli elementi della collezione passata come parametro (implementando così una forma di *unione*: `i.addAll(c)` significa $i \leftarrow i \cup c$, dove `i` è l'oggetto di invocazione e `c` la collezione passata come parametro); ritorna `true` se l'insieme è cambiato dopo l'invocazione di questo metodo. Ad esempio:

```
Set s2 = new HashSet();
s2.addAll(s1);  // true: s2 = [1,2]
s2.addAll(s1);  // false: s2 = [1,2]
```

- `void clear()`: rimuove tutti gli elementi dall'insieme. Ad esempio:

```
s2.clear();   // s2 = []
```

- `boolean contains(Object o)`: ritorna `true` se l'insieme contiene un elemento uguale all'oggetto passato come parametro. Ad esempio:

```
s1.contains(1); // true
s1.contains(5); // false
```

- `boolean containsAll(Collection c)`: ritorna `true` se l'insieme contiene tutti gli elementi della collezione passata come parametro. Ad esempio:

```
s2.addAll(s1);       // s2 = s1 = [1,2]
s1.containsAll(s2);  // true
s2.add(3);           // s2 = [1,2,3]
s1.containsAll(s2);  // false
```

- `boolean equals(Object o)`: verifica l'uguaglianza tra l'insieme e l'oggetto `o`. Ad esempio:

```
s1.equals(s2); // false (s1 = [1,2] e s2 = [1,2,3])
s1.add(3);     // s1 = [1,2,3]
s1.equals(s2); // true
```

- `boolean isEmpty()`: ritorna `true` se l'insieme è vuoto. Ad esempio:

```
s2.isEmpty();  // false
s2.clear();
s2.isEmpty();  // true
```

- `Iterator iterator()`: restituisce un oggetto Iterator, per iterare sugli elementi dell'insieme. Ad esempio:

```
Iterator it = s1.iterator();           // s1 = [1,2,3]
while(it.hasNext)
   System.out.println(it.next + " ");  // 3 2 1
```

- `int size()`: ritorna il numero di elementi presenti nell'insieme. Ad esempio:

```
s1.size();  // s1 = [1,2,3], size = 3
```

- `boolean remove(Object o)`: rimuove dall'insieme gli elementi uguali all'oggetto passato come parametro. Ritorna `true` se l'insieme è cambiato dopo l'invocazione del metodo. Ad esempio:

```
s1.remove(1);   // true: s1 = [2, 3]
s1.remove(5);   // false
```

- `boolean removeAll(Collection c)`: rimuove dall'insieme tutti gli elementi uguali a quelli che sono contenuti nella collezione passata come parametro (implementando così una forma di *differenza*: `i.removeAll(c)` significa $i \leftarrow i \setminus c$, dove `i` è l'oggetto di invocazione e `c` la collezione passata come parametro); ritorna `true` se l'insieme è cambiato dopo l'invocazione di questo metodo. Ad esempio:

```
s1.removeAll(s2);  // false: s1 = [2,3] e s2 = []
s2.addAll(s1);     // s1 = s2 = [2,3]
s2.add(4);         // s2 = [2,3,4]
s2.removeAll(s1);  // true: s2 = [4]
```

- `boolean retainAll(Collection c)`: rimuove dall'insieme tutti gli elementi che non sono presenti nella collezione passata come parametro (implementando così una forma di *intersezione*: `i.retainAll(c)` significa $i \leftarrow i \cap c$, dove `i` è l'oggetto di invocazione e `c` la collezione passata come parametro); ritorna `true` se l'insieme è cambiato dopo l'invocazione di questo metodo. Ad esempio:

```
s1.retainAll(s2);  // true: s1 = []
s1.addAll(s2);     // s1 =  s2 = [4]
s1.retainAll(s2);  // false: s1 = s2 = [4]
```

2.4 Implementazione di `Set`

L'interfaccia `Set` viene implementata dalla classe astratta `AbstractSet`, che è superclasse delle sottoclassi `HashSet`, `TreeSet`, `EnumSet`. Vediamo in particolare le prime due classi.

2.4.1 HashSet

```
public class HashSet extends AbstractSet implements Set
```

Questa classe implementa l'interfaccia `Set` e gestisce un insieme non ordinato mediante tabelle hash con liste concatenate. Gli oggetti vengono inseriti in posizioni della tabella che dipendono dalla funzione di hash adottata.

Una *funzione di hash* è una funzione che calcola un numero intero, codice hash, a partire dai dati di un oggetto, in modo che sia molto probabile che oggetti non uguali abbiano codici diversi. Due o più oggetti possono avere lo stesso codice di hash: questa situazione genera una collisione. Una buona funzione di hash deve minimizzare le collisioni. Se i codici hash sono diversi allora gli elementi non sono uguali; il contrario non è necessariamente vero. Le funzioni di hash vengono usate per creare collezioni (mappe e insiemi) su cui poter operare in modo molto efficiente. L'idea è quella di avere un array i cui indici siano i codici hash degli elementi. Questa idea però ha due problemi:

1. La dimensione dell'array
2. Le collisioni

Per ovviare al primo problema si deve scegliere una funzione di hash che generi codici in un range ragionevolmente ridotto; per ovviare al secondo problema si usano liste concatenate. Se la funzione di hash è ben definita (range piccolo e poche collisioni) le operazioni di ricerca, verifica di appartenenza, rimozione, inserimento in una collezione hanno costo costante; le operazioni insiemistiche hanno complessità lineare.

In Java il codice hash di un oggetto deve essere restituito dal metodo `hashCode()`. Se vogliamo usare l'implementazione `HashSet` di `Set`, gli oggetti della collezione devono avere i metodi `hashCode()` e `equals()` definiti opportunamente. In sintesi, per verificare la presenza di duplicati `HashSet` usa `equals()` in combinazione con il valore restituito dal metodo `hashCode()`. In particolare, per verificare se due oggetti sono uguali, `HashSet` prima verifica se il loro codice hash è identico: solo in caso affermativo (collisione) viene invocato il metodo `equals()`.

Il metodo `remove()` senza argomento si riferisce all'interfaccia `Iterator`. Quest'ultimo metodo evita modifiche concorrenti che possano interferire con altre istanze di iteratori mediante contatori di modifiche. `HashSet` non dà garanzie sull'ordine di iterazione dell'insieme; in particolare, non garantisce che l'ordine rimarrà costante nel tempo. Questa classe permette gli elementi nulli.

2.4.2 TreeSet

```
public class TreeSet extends AbstractSet implements SortedSet
```

La classe `TreeSet` implementa l'interfaccia `Set` e gestisce gli insiemi ordinati mediante alberi binari di ricerca non bilanciati. L'implementazione di `TreeSet` garantisce (oltre all'assenza di duplicati) che gli elementi siano ordinati in accordo con:

- l'ordinamento naturale interno, oppure
- un ordinamento esterno stabilito da un comparatore e noto all'insieme stesso (ad esempio perché ricevuto al momento della sua creazione attraverso uno dei suoi molteplici costruttori).

Il criterio di equivalenza tra elementi si basa sul metodo `compareTo()` o `compare()` (di una classe esterna). Gli oggetti dell'insieme devono fornire il metodo `compareTo()` dell'interfaccia standard `Comparable` per mantenere le chiavi ordinate. L'eccezione `IllegalArgumentException` viene generata se gli oggetti non sono tutti `Comparable`.

È possibile, infine, creare un iteratore `Iterator` che scandisce gli oggetti in maniera crescente secondo l'ordine specificato. Il metodo `remove()` senza argomento si riferisce all'interfaccia `Iterator`. Quest'ultimo metodo evita modifiche concorrenti che possano interferire con altre istanze di iteratori mediante contatori di modifiche.

L'inserzione, la ricerca e la cancellazione hanno complessità logaritmica rispetto al numero di elementi; le operazioni insiemistiche hanno complessità lineare.

Capitolo 3

Integrazione

In questo capitolo descriviamo l'architettura complessiva del nuovo package JSetL e le interfacce Java create. Le classi che implementano le interfacce sono descritte nei successivi capitoli.

3.1 Motivazioni

Lo scopo del lavoro è di creare un'unica astrazione di insieme su cui poter operare sia con i metodi forniti da `java.util.Set` che con quelli di JSetL, limitando al minimo le modifiche al `package` JSetL. Il nuovo `package` JSetL deve fornire quindi anche tutte le funzionalità degli insiemi di Java, mantenendo la piena compatibilità con applicazioni scritte usando le precedenti librerie (a meno di nomi).

Per gli insiemi completamente specificati bisogna garantire che la semantica dele operazioni sugli insiemi di Java sia la stessa prevista dall'interfaccia `java.util.Set`. Questo deve essere vero anche se l'insieme è costruito a partire da un "logical set" contenente variabili logiche inizializzate; ad esempio se sull'insieme s così definito:

$$s = \{1, x, 4\}, \text{ con } x \text{ variabile logica con valore } 3$$

viene richiamato il metodo `contains` di `java.util.Set`

```
s.contains(3);
```

deve essere restituito `true` come risultato.

I metodi di utilità previsti in JSetL con comportamento "extra logico" verranno rimpiazzati, quando possibile, dai corrispondenti metodi di `java.util.Set`. Ad esempio:

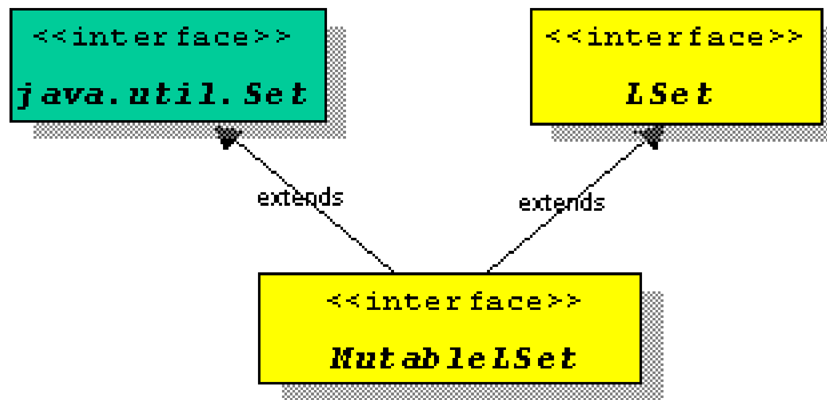
- Il metodo `concat` può essere sostituito dal metodo `addAll`;
- Il metodo `sub` può essere sostituito dal metodo `remove`.

Mentre i metodi `isEmpty`, `size` e `clear` presenti sia in `java.util.Set` che in JSetL, con semantica sostanzialmente identica, possono essere fusi insieme.
Su insiemi qualsiasi (completamente e parzialmente specificati, limitati e illimitati) deve rimanere possibile operare con le operazioni previste da JSetL, in particolare con i vincoli insiemistici forniti da JSetL.

3.2 Architettura

Si vuole aggiungere a JSetL una classe che implementi i metodi dell'interfaccia `java.util.Set` e nello stesso tempo possieda tutte le funzioni di `JSetL.Set`. Per fare questo definiamo due interfacce: `LSet` ("Logical Set") e `MutableLSet`. La prima contiene tutti i metodi `public` di `JSetL.Set`; la seconda estende le interfacce `LSet` e `java.util.Set`, ereditandone tutti i metodi.

Vediamo la gerarchia di queste tre interfacce:

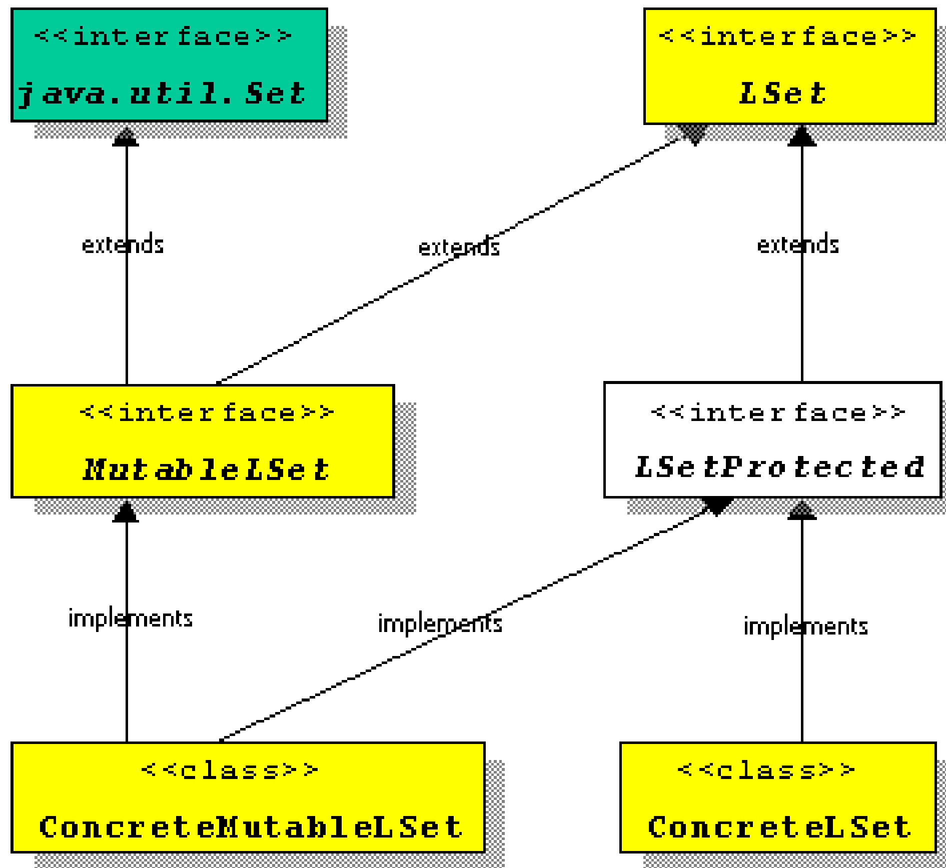


Un insieme in `LSet` è considerato un oggetto immutabile, ovvero qualsiasi operazione su di esso non modifica mai l'oggetto, ma piuttosto ne viene fatta una copia. `MutableLSet` invece viene chiamata *mutabile* proprio perché si prevede che l'oggetto chiamante possa essere modificato, in quanto i metodi di `java.util.Set` modificano l'oggetto di invocazione.

Per mantere non visibili i metodi `proteceted` al di fuori del `package` aggiungiamo anche l'interfaccia `LSetProtected`, con visibilità `package`, che estende `LSet`.

Le classi concrete sono due: `ConcreteLSet`, che implementa l'interfaccia `LSetProtected` ed è sostanzialmente la classe `JSetL.Set` originale cambiata di nome e con alcune modifiche; `ConcreteMutableLSet`, che implementa `LSetProtected` e `MutableLSet`.

Vediamo il diagramma di classi dove vengono rappresentate le interfacce e le classi che le implementano:



Questa architettura permette di creare, attraverso l'interfaccia `MutableLSet` un oggetto che possa utilizzare sia i metodi di `java.util.Set`, che quelli di `LSet`:

```
MutableLSet s1 = new ConcreteMutableLSet();
```

Questo è il modo più generico per costruire un insieme, in quanto `s1` fornisce sia le funzionalità degli insiemi in Java che quelle degli insiemi in JSetL e può essere inserito senza problemi di compatibilità in entrambi i contesti.
Inoltre è possibile creare oggetti che possano utilizzare rispettivamente i metodi delle due interfacce:

- `LSet s2 = new ConcreteLSet();`

  In questo modo viene creato un insieme `s2` con le stesse potenzialità e

gli stessi metodi della classe originale `JSetL.Set` (equivale all'originale `Set s2 = new Set();`)

- `java.util.Set s3 = new ConcreteMutableLSet();`
  L'insieme `s3` può utilizzare tutti e soli i metodi dell'interfaccia `Set` di Java, che vengono implementati nella classe `ConcreteMutableLSet`. Questa classe può essere considerata un'ulteriore implementazione di `java.util.Set`, come `HashSet` e `TreeSet`.

3.3 L'interfaccia LSet

L'interfaccia `LSet` è pubblica e contiene tutti i metodi `public` dell'originale classe `JSetL.Set`.

```
public interface LSet {
   // dichiarazione dei metodi public di JSetL.Set
}
```

Vediamo in particolare quali attributi e metodi contiene.

Attributi

Come abbiamo visto nella sezione 2.1, un'interfaccia in Java può contenere solo attributi costanti. `LSet` contiene un solo attributo costante pubblico:

```
public static final ConcreteLSet empty =
                new ConcreteLSet("emptySet", null, null);
```

L'attributo `empty` è un `ConcreteLSet` e richiama un costruttore di `ConcreteLSet` per creare l'insieme vuoto.

Metodi

I metodi dell'interfaccia `LSet` si possono suddividere in 2 categorie:

1. **metodi di vario utilizzo**: `ins`, `insAll`, `clone`, `isEmpty`, `equals`, `isBound`, `toVector`, `size`, `concat`, `output`, `print`, `isGround`, `known`, `read`, `normalizeSet`, `toString`, `first`, `setName`, `getName`, `getValue`, `get`.

2. **metodi di generazione di vincoli**: sono metodi che permettono di creare vincoli che svolgono operazioni insiemistiche, tra cui: uguaglianza (`eq`) e disuguaglianza (`neq`), appartenenza (`in`) e non appartenenza (`nin`), inclusione (`subset`) e non inclusione (`nsubset`), unione (`union`) e non unione (`nunion`), intersezione (`inters`) e non intersezione (`ninters`), disgiunzione (`disj`) e non disgiunzione (`ndisj`), differenza (`differ`) e non differenza (`ndiffer`), sottrazione (`less`, `less1`).

I metodi, che nella dichiarazione hanno un insieme come tipo di ritorno o come parametro, utilizzano l'interfaccia `LSet`, in quanto è visibile anche dall'esterno del `package`. Vediamo ad esempio i metodi `ins` e `in` come vengono dichiarati:

```
public LSet ins(Object o);
public Constraint in (LSet set);
```

3.4 L'interfaccia `LSetProtected`

L'interfaccia `LSetProtected` è stata aggiunta in quanto necessaria per mantere non visibili al di fuori del `package` i metodi `protected` di `JSetL.Set`. In un'interfaccia tutti i metodi devono essere `public`, ma la loro visibilità dipende dalla visibilità dell'interfaccia stessa. `LSetProtected` è stata definita con visibilità `package`, ovvero non anteponendo niente alla parola `interface`:

```
interface LSetProtected extends LSet {
   // dichiarazione dei metodi protected
}
```

In questo modo `LSetProtected` è visibile e pertanto utilizzabile solo all'interno del `package`. Questa interfaccia estende `LSet` e viene a sua volta implementata da `ConcreteMutableLSet` e `ConcreteLSet`.

Attributi

L'interfaccia `LSetProtected` contiene gli attributi constanti `private` originali della classe `JSetL.Set`:

```
static String prefix = "LSet_";
static Vector nonInizSet = new Vector();
static SolverClass solver = new SolverClass();
```

- `prefix` è una stringa e viene inizializzata con "`LSet_`", a cui viene concatenato un numero sempre crescente, per dare un nome univoco a ogni `LSet` che viene creato;
- `nonInizSet` è un vettore che viene creato vuoto e in cui viene inserito l'oggetto di invocazione se questo è non inizializzato (`if(!iniz) nonInizSet.add(this)`);
- `solver` è un oggetto della classe `SolverClass` e viene utilizzato per risolvere i vincoli.

Metodi

Anche i metodi di `LSetProtected` si possono suddividere in 2 categorie:

- **metodi di tipo set e get**: i metodi set e get rispettivamente impostano e restituiscono un attributo non costante della classe concreta che implementa l'interfaccia `LSet`, tra cui: `iniz`, `lista`, `resto`, `equ`, `isInt`, `inter`, `id`, `counter`.
- **metodi di vario utilizzo non visibili fuori dal package**: `sub`, `subfirst`, `sost`, `appendGround`, `append`, `occurs`, `contains`, `isEmptySL`, `ultimoResto`, `value`.

I metodi, che nella dichiarazione hanno un insieme come tipo di ritorno o come parametro, utilizzano l'interfaccia `LSetProtected`, in quanto è visibile solo all'interno del `package`. Vediamo ad esempio come viene dichiarato il metodo `append`:

```
public LSetProtected append(LSetProtected set);
```

L'interfaccia `LSetProtected` viene nascosta all'esterno e viene solo utilizzata per l'implementazione interna.

3.5 L'interfaccia `MutableLSet`

L'interfaccia `MutableLSet` viene chiamata così in quanto la sua implementazione crea degli oggetti che sono *mutabili*, cioè che attraverso una chiamata a un metodo l'oggetto di invocazione può essere modificato, mentre con `LSet` viene creato un oggetto nuovo. Questa interfaccia eredita i metodi delle interfacce `java.util.Set` e `LSet`, senza aggiungerne di nuovi. La sua definizione è la seguente:

```
public interface MutableLSet extends java.util.Set, LSet
{ }
```

In quanto `MutableLSet` è un interfaccia, può estendere più interfacce, cosa che non è possibile tra le classi, ovvero una classe può ereditare al più da un'altra classe.

Capitolo 4

Implementazione: la classe ConcreteLSet

La classe `ConcreteLSet` implementa l'interfaccia `LSetProtected` (e per ereditarietà anche l'interfaccia `LSet`), ed è stata creata utilizzando la classe `Set` originale di JSetL, modificandola in alcune sue parti e cambiandole il nome per renderlo più esplicativo: "`Concrete`" perché è una classe concreta e "`LSet`" (*Logical Set*) per sottolineare che è un insieme logico.

```
public class ConcreteLSet implements java.io.Serializable, LSetProtected {
    // attributi
    // costruttori
    // implementazione metodi di LSetProtected e LSet
}
```

Vediamo più nel particolare la classe `ConcreteLSet`.

4.1 Attributi

Gli attributi costanti della classe originale (`prefix`, `nonInizSet`, `solver` ed `empty`) sono stati, inseriti e inizializzati nelle interfacce `LSet` e `LSetProtected`.
Gli altri attributi sono invece rimasti invariati, e sono:

```
protected boolean iniz = false;
protected Vector lista;
protected LSet resto;
protected LSet equ = null;
protected Int inter = null;
protected boolean isInt = false;
protected String name = null;
protected int id;
private static int counter = 0;
```

L'attributo `iniz` indica se l'insieme è inizializzato o no: di default è non inizializzato (`false`). I campi `lista` e `resto` servono per memorizzare il valore dell'insieme. `lista` è un riferimento ad un vettore e contiene gli elementi dell'insieme; `resto` è un riferimento ad un oggetto `LSet` e contiene il resto non specificato dell'insieme. Il campo `equ`, inizializzato a `null`, è un riferimento ad un altro oggetto `LSet`. Questo attributo è stato introdotto per rendere più efficiente la propagazione dei vincoli. Il campo `inter` è un oggetto di tipo `Int` (classe di JSetL che implementa l'astrazione degli intervalli) ed è `null` di default, altrimenti gli viene assegnato l'intervallo passato nel costruttore corrispondente. Per verificare che l'insieme sia formato da un intervallo si utilizza l'attributo booleano `isInt` che ha valore `true` se l'insieme è un intervallo, `false` altrimenti. L'attributo `name` è una stringa che memorizza il nome dell'insieme: si può assegnare il nome passandolo come parametro quando viene costruito un oggetto, oppure gli viene assegnato il nome di default "`LSet_`" seguito dal numero univoco `id`, un intero a cui viene assegnato il valore di `counter`, attributo statico che viene inizializzato a 0 e incrementato ogni volta che viene creato un oggetto.

4.2 Costruttori

Vediamo quali sono i costruttori per creare un oggetto di tipo `ConcreteLSet`. Innanzi tutto commentiamo il costruttore senza parametri:

```
public ConcreteLSet() {
   this.lista = null;
   this.resto = null;
   this.id = counter++;
   nonInizSet.add(this);
   this.name = prefix.concat((new Integer(this.id)).toString());
}
```

Gli attributi `lista` e `resto` vengono inizializzati a `null`, in quanto l'insieme non contiene nessun elemento. All'`id` viene assegnato il valore di `counter` incrementato di uno. Al vettore `nonInizSet` viene aggiunto l'oggetto non inizializzato. Infine il nome sarà la concatenazione tra `prefix` (“`LSet_`”) e l'`id`.

In secondo luogo vediamo i costruttori con un parametro. Vediamo ad esempio la definizione di un costruttore che prende come parametro un array:

```
public ConcreteLSet(tipo[] arr) {...}
```

dove per “`tipo`” si intendono `Object` e i tipi primitivi `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`. Vediamo come si comporta uno di questi costruttori, ad esempio quello che prende un array di interi come parametro:

```
private Vector wrapperSet() {
   Vector v   = new Vector();
   this.iniz  = true;
   this.resto = ConcreteLSet.empty;
   this.id    = counter++;
   return v;
}

public ConcreteLSet(int[] arr) {
   Vector v = this.wrapperSet();
```

```
    for(int i = 0; i < arr.length; i++) {
        Integer ii = new Integer(arr[i]);
        v.add(ii);
    }
    this.lista = v;
    this.name  = prefix.concat((new Integer(this.id)).toString());
}
```

La prima cosa che il costruttore fa è creare un vettore richiamando il metodo privato `wrapperSet`, che crea un vettore vuoto, imposta la variabile `iniz` a `true` per indicare che l'insieme è inizializzato, l'attributo `resto` viene anch'esso inizializzato e creato vuoto, la variabile `id` assume il suo valore da `counter` e infine restituisce un vettore. In secondo luogo il costruttore scorre gli elementi dell'array, crea un oggetto della classe corrispondente al tipo primitivo utilizzato (a parte nel caso di `Object`) e aggiunge ogni elemento al vettore. Infine all'attributo `lista` viene assegnato il valore del vettore e a `name` il nome dell'oggetto.

Gli altri tipi di costruttori con un parametro della classe `ConcreteLSet` sono:

```
public ConcreteLSet(String n) {...}
public ConcreteLSet(Int I) {...}
public ConcreteLSet(LSet s) {...}
public ConcreteLSet(Lst l) {...}
public ConcreteLSet(Lvar l) {...}
```

Nel primo viene richiamato il costruttore vuoto con `this()` e all'attributo `name` viene assegnata la stringa passata come parametro. Nel secondo caso l'insieme è inizializzato (`iniz = true`), alla variabile `isInt` viene assegnato il valore `true`, a `inter` viene assegnato l'intervallo passato, `lista` e `resto` sono nulli e `id` e `name` vengono inizializzati come sempre. Il terzo costruttore assegna il valore degli attributi dell'insieme passato agli attributi dell'oggetto chiamante, in particolare `equ` viene uguagliato all'insieme `s`. Negli ultimi

due costruttori viene creato un vettore richiamando il metodo `wrappedSet` e ad esso viene aggiunta la variabile logica o la lista (`v.add(l)`) e viene creato il nome.

I costruttori con due parametri creano tutti gli oggetti già creati dai costruttori con un parametro e in più viene assegnato loro un nome. In generale l'implementazione è la seguente:

```
public ConcreteLSet(String n, tipo t) {
   this(t);
   this.name = n;
}
```

Vi sono altri due costruttori con due parametri con i rispettivi costruttori con tre parametri che in più prendono una stringa, che rappresenta il nome dell'insieme. Essi sono:

```
public ConcreteLSet(int p, int q) {...}
public ConcreteLSet(int p, Lvar q) {...}

public ConcreteLSet(String n, int p, int q) {...}
public ConcreteLSet(String n, int p, Lvar q) {...}
```

Entrambi creano un insieme con gli elementi di un intervallo, con la differenza che nel primo e nel terzo vengono passati due interi estremi dell'intervallo, mentre nel secondo e nel quarto si può passare una variabile logica il cui valore rappresenta il limite superiore dell'insieme.

Infine vi è un costruttore protetto che prende come parametri un vettore e un insieme (vi è anche il costruttore corrispondente con in più una stringa come parametro per il nome):

```
protected ConcreteLSet(Vector v, LSet s) {...}
protected ConcreteLSet(String n, Vector v, LSet s) {...}
```

Questo costruttore prima di tutto controlla se il vettore e l'insieme sono nulli, in tal caso `iniz` è inizializzata a `true` e `lista` e `resto` a `null`; altrimenti alla variabile `lista` viene assegnato il vettore `v` e a `resto` l'insieme `s`. Inoltre se la dimensione della lista è positiva, allora l'attributo `iniz` diventa `true`, altrimenti gli viene assegnato il valore di `iniz` dell'insieme `s`. Infine si controlla ancora se `iniz` è uguale a `true` e se non lo è l'oggetto chiamante viene inserito nel vettore `nonInizSet`; `id` e `name` vengono inizializzati nel solito modo.

4.3 Implementazione dei metodi di `LSet` e `LSetProtected`

4.3.1 Metodi di tipo set e get

I metodi di tipo `set` e `get` non esistono nella classe `JSetL.Set` originale e sono stati aggiunti per dare la possibilità anche alla nuova classe `ConcreteMutableLSet` di poter richiamare e impostare gli attributi di `ConcreteLSet`. Vediamo ad esempio i metodi di tipo `set` e `get` per l'attributo `lista`:

```
public Vector getLista() {
       return lista;
}
public void setLista(Vector lista) {
       this.lista = lista;
}
```

Il metodo `getLista` restituisce l'attributo `lista`, mentre il metodo `setLista` assegna all'attributo `lista` il valore passato come parametro.

4.3.2 Metodi di vario utilizzo

L'implementazione di questi metodi è quella prevista in JSetL. In particolare vediamo alcuni dettagli sull'implementazione dei metodi di inserimento e di estrazione.

I metodi di inserimento di elementi in un insieme sono due:

```
public LSet ins(tipo n) {...}
public LSet insAll(tipo[] arr) {...}
```

Il primo prende come parametro un tipo, che può essere un tipo primitivo, oppure un `Object`. Il metodo `ins` controlla se la variabile `equ` è uguale a `null` e in tal caso costruisce un vettore vuoto, vi aggiunge il parametro `n` con il proprio metodo `add`, crea un `LSet` tramite il costruttore protetto della classe concreta `ConcreteLSet` che prende un vettore e un `LSet` e infine restituisce l'oggetto creato. Altrimenti, se l'attributo `equ` non è `null`, viene richiamato il metodo stesso su `equ` (`return this.equ.ins(n)`).

Il metodo `insAll` ha come parametro un array, che può essere ancora un qualsiasi tipo primitivo o un `Object`. L'implementazione è molto simile alla precedente, con la differenza che si scorre l'array con un ciclo `for` e viene inserito nel vettore creato un elemento alla volta.

Il metodo di estrazione di elementi da un insieme è il seguente:

```
public LSet sub() {...}
```

Il metodo `sub` restituisce un `LSet` privato del primo elemento dell'insieme. Questo metodo, dal comportamento "extra logico", può essere sostituito dal nuovo metodo `addAll` di `java.util.Set`, come viene spiegato più avanti.

Ai metodi già presenti in JSetL è stato aggiunto il metodo booleano `isBound`, che verifica se l'insieme è illimitato.

```
public boolean isBound() {
    if(getIniz() || ultimoResto().equals(ConcreteLSet.empty))
       return true;
    else return false;
}
```

Questo metodo controlla se l'insieme è inizializzato o se la sua "coda" (data dal metodo booleano `ultimoResto`) è vuota: se una delle due condizioni sono verificate, l'insieme è limitato e inizializzato.

4.3.3 Metodi di generazione di vincoli

Nella classe `ConcreteLSet` è possibile costruire dei vincoli che verranno poi inseriti nel Constraint Store e saranno risolti tramite il metodo `Solve` della classe `SolverClass`. Vediamo quali sono i metodi della classe `ConcreteLSet` che generano vincoli e il loro significato applicato a un `LSet s`:

Uguaglianza e disuguaglianza:

utilizzo	metodo	vincolo generato
`s.eq(o)`	`public Constraint eq(Object o)`	s = o
`s.neq(o)`	`public Constraint neq(Object o)`	s $\neq$ o

Less:

utilizzo	metodo	vincolo generato
`s.less(l,ls)`	`Constraint less(Lvar l,Lvar ls)`	s $\in$ ls $\wedge$
	`Constraint less(Lvar l,LSet ls)`	ls = s $\setminus$ {l}

Appartenenza e non appartenenza:

utilizzo	metodo	vincolo generato
`s.in(ls)`	`Constraint in(LSet ls)`	s $\in$ ls
	`Constraint in(Lvar ls)`	
`s.nin(ls)`	`Constraint nin(LSet ls)`	s $\notin$ ls
	`Constraint nin(Lvar ls)`	

Disgiunzione e non disgiunzione:

utilizzo	metodo	vincolo generato
`s.disj(ls)`	`Constraint disj(LSet ls)`	s $\cap$ ls $\neq \emptyset$
	`Constraint disj(Lvar ls)`	
`s.ndisj(ls)`	`Constraint ndisj(LSet ls)`	s $\cap$ ls = $\emptyset$
	`Constraint ndisj(Lvar ls)`	

Inclusione e non inclusione:

utilizzo	metodo	vincolo generato
`s.subset(ls)`	`Constraint subset(LSet ls)`	$s \subseteq ls$
	`Constraint subset(Lvar ls)`	
`s.nsubset(ls)`	`Constraint nsubset(LSet ls)`	$s \nsubseteq ls$
	`Constraint nsubset(Lvar ls)`	

Unione e non unione:

utilizzo	metodo	vincolo generato
`s.union(ls1,ls2)`	`Constraint union(LSet ls1,LSet ls2)`	$s = ls1 \cup ls2$
	`Constraint union(Lvar ls1,Lvar ls2)`	
	`Constraint union(LSet ls1,Lvar ls2)`	
	`Constraint union(Lvar ls1,LSet ls2)`	
`s.nunion(ls1,ls2)`	`Constraint nunion(LSet ls1,LSet ls2)`	$s \neq ls1 \cup ls2$
	`Constraint nunion(Lvar ls1,Lvar ls2)`	
	`Constraint nunion(LSet ls1,Lvar ls2)`	
	`Constraint nunion(Lvar ls1,LSet ls2)`	

Intersezione e non intersezione:

utilizzo	metodo	vincolo generato
`s.inters(ls1,ls2)`	`Constraint inters(LSet ls1,LSet ls2)`	$s = ls1 \cap ls2$
	`Constraint inters(Lvar ls1,Lvar ls2)`	
	`Constraint inters(LSet ls1,Lvar ls2)`	
	`Constraint inters(Lvar ls1,LSet ls2)`	
`s.ninters(ls1,ls2)`	`Constraint ninters(LSet ls1,LSet ls2)`	$s \neq ls1 \cap ls2$
	`Constraint ninters(Lvar ls1,Lvar ls2)`	
	`Constraint ninters(LSet ls1,Lvar ls2)`	
	`Constraint ninters(Lvar ls1,LSet ls2)`	

Differenza e non differenza:

utilizzo	metodo	vincolo generato
`s.differ(ls1,ls2)`	`Constraint differ(LSet ls1,LSet ls2)`	`s = ls1` $\setminus$ `ls2`
	`Constraint differ(Lvar ls1,Lvar ls2)`	
	`Constraint differ(LSet ls1,Lvar ls2)`	
	`Constraint differ(Lvar ls1,LSet ls2)`	
`s.ndiffer(ls1,ls2)`	`Constraint ndiffer(LSet ls1,LSet ls2)`	`s` $\neq$ `ls1` $\setminus$ `ls2`
	`Constraint ndiffer(Lvar ls1,Lvar ls2)`	
	`Constraint ndiffer(LSet ls1,Lvar ls2)`	
	`Constraint ndiffer(Lvar ls1,LSet ls2)`	

L'implementazione di tutti questi metodi è invariata rispetto a JSetL originale.

Capitolo 5

Implementazione: la classe ConcreteMutableLSet

La classe ConcreteMutableLSet implementa l'interfaccia MutableLSet, ovvero tutti i metodi di java.util.Set e di JSetL.LSet e in più implementa anche l'interfaccia LSetProtected. Viene chiamata concrete perché è una classe concreta e pertanto può istanziare oggetti, e mutable perché può cambiare l'oggetto di invocazione

Questa classe eredita gli attributi costanti delle interfacce LSet e LSetProtected e in più ha al suo interno un oggetto protetto della classe ConcreteLSet chiamato set, che serve sia per richiamare su di esso i metodi di ConcreteLSet, sia per realizzare i metodi di java.util.Set modificando l'oggetto.
Vediamo la dichiarazione della classe e del campo set.

```
public class ConcreteMutableLSet implements MutableLSet, LSetProtected {
    protected ConcreteLSet set;
    ...
}
```

L'oggetto set è dichiarato protected, ovvero può essere modificato solo all'interno del package o da una eventuale sottoclasse diretta di ConcreteMutableLSet.

5.1 Costruttori

In generale i costruttori della classe `ConcreteMutableLSet` sono gli stessi della classe `ConcreteLSet` (tutti quelli `public`). Viene richiamato il costruttore corrispondente di `ConcreteLSet` attraverso l'oggetto `set`:

```
public ConcreteMutableLset(tipo t1,...) {
   set = new ConcreteLSet(t1,...);
}
```

È stata fatta una modifica al costruttore senza parametri ed è stato aggiunto un altro costruttore. In JSetL, quando viene creato un oggetto con il costruttore senza parametri

```
LSet s = new ConcreteLSet();
```

l'insieme `s` è non inizializzato, invece nella semantica di `java.util.Set` l'oggetto creato è vuoto, ovvero il metodo `isEmpty` ritorna `true`. Per fare questo il costruttore senza parametri di `ConcreteMutableLSet` è stato implementato costruendo un insieme inizializzato, ma senza elementi:

```
public ConcreteMutableLSet() {
    set = MutableLSet.empty;
}
```

In questo modo `set` è un insieme vuoto. Però, per non perdere la semantica del costruttore senza parametri di JSetL, è stato implementato un nuovo costruttore (vi è anche il costruttore corrispondente con in più una stringa come parametro per il nome):

```
public ConcreteMutableLSet(boolean iniz) {
   if (iniz) set = MutableLSet.empty;
   else set = new ConcreteLSet();
}
```

```
public ConcreteMutableLSet(String n, boolean iniz) {
   this(iniz);
   setName(n);
}
```

Il primo costruttore prende come parametro un booleano e se gli viene passato `true` crea un insieme vuoto; altrimenti richiama il costruttore senza parametri di `ConcreteLSet` che crea un oggetto non inizializzato. Il secondo costruttore richiama il primo con `this(iniz)` e imposta il nome a `n`. Ad esempio:

```
MutableLSet s1 = new ConcreteMutableLset(false);
MutableLSet s2 = new ConcreteMutableLset("s2",false);
```

`s1` e `s2` sono due insiemi non inizializzati e hanno la stessa semantica delle istruzioni:

```
LSet s1 = new ConcreteLSet();
LSet s2 = new ConcreteLSet("s2");
```

Se si vuole creare un insieme illimitato bisogna utilizzare questo costruttore:

```
MutableLSet s1 = new ConcreteMutableLSet("s1",false);
MutableLSet s2 = new ConcreteMutableLSet("s2",s1.ins(1));
s2.output();

/*
 * s2 = {1|s1}
 */
```

Se invece si vuole creare un insieme vuoto basta utilizzare il costruttore senza parametri.

5.2 Implementazione dei metodi di `LSet` e di `LSetProtected`

Tutti i metodi di `LSet` e di `LSetProtected` vengono implementati richiamando i metodi di `ConcreteLSet` attraverso l'oggetto `set`. La forma generale è la seguente:

```
public tipoRitorno funzione(Param p) {
   return set.fuzione(p);
}
```

In questo modo vengono reindirizzati alla classe `ConcreteLSet` che li gestisce.
Vediamo ad esempio il metodo `ins`:

```
public LSet ins(LSet s) {
    return set.ins(s);
}
```

In questo modo, nel momento in cui un oggetto di tipo `MutableLSet` utilizza il metodo `ins`:

```
MutableLSet s = new ConcreteMutableLSet("s",MutableLSet.empty.ins(1).ins(3));
s.output();


/*
 * s = {3,1}
 */
```

nell'implementazione viene richiamato il metodo `ins` di `ConcreteLSet` attraverso l'oggetto `set`.

5.3 Implementazione dei metodi di `java.util.Set`

Abbiamo già visto i metodi più importanti di `java.util.Set` e le loro funzioni nel paragrafo 2.3, ora vediamo come vengono implementati nella

classe `ConcreteMutableLSet`. Come in parte specificato, alcuni metodi di `java.util.Set` rappresentano delle operazioni insiemistiche, fornite anche dall'interfaccia `LSet`. Nella seguente tabella possiamo vedere le corrispondenze:

`java.util.Set`	OPERAZIONI INSIEMISTICHE	`LSet`
`add`	inserimento	`ins`
`addAll`	unione	`union`
`contains`	appartenenza	`in`
`containsAll`	inclusione	`subset`
`remove`	estrazione	`less`
`removeAll`	differenza	`differ`
`retainsAll`	intersezione	`inters`
`equals`	uguaglianza	`eq`

La differenza fondamentale tra i metodi implementati da `java.util.Set` e quelli insiemistici di `LSet` è che i metodi di `java.util.Set` modificano l'oggetto chiamante, tramite un assegnamento; invece i metodi di `LSet` non modificano l'oggetto, ma ne fanno una copia. Ne segue che i metodi di `java.util.Set` non sono utilizzabili nell'ambito della programmazione dichiarativa, invece quelli di `LSet` sono adatti.

Inoltre vi sono altri metodi di JSetL che hanno un'analogia con quelli di `java.util.Set`, e sono:

`LSet`	`java.util.Set`
`size`	`size`
`isEmpty`	`isEmpty`
`clear`	`clear`
`concat`	`addAll`
`sub`	`remove`

I primi tre metodi, oltre ad avere lo stesso nome, hanno anche la stessa semantica, quindi i metodi di `java.util.Set` vengono implementati richiamando il metodo omonimo di `ConcreteLSet`. Il metodo `concat` concatena due insiemi, funzione fornita anche dal metodo `addAll`, che può sostituirlo. Lo stesso vale per il metodo `sub` che rimuove un elemento, così come `remove`.

I metodi di `java.util.Set` sono stati implementati solo per gli insiemi limitati e inizializzati, pertanto, all'inizio di ogni metodo, viene fatto un controllo sull'insieme attraverso il metodo booleano `isBound()` e se ritorna `false` viene lanciata l'eccezione `UnboundedSetException`.
Vediamo più nel particolare come sono stati implementati i metodi di `java.util.Set`.

5.3.1 Inserimento e unione: `add` e `addAll`

`Inserimento`

Il metodo `add` inserisce nell'insieme un `Object`.

```
public boolean add(Object o) {
   if(isBound()) {
       int n = set.size();
       set = set.ins(o);
       return (set.size() > n);
   }
   else throw new UnboundedSetException();
}
```

Il metodo `add` prima di tutto verifica che l'insieme sia limitato e inizializzato, altrimenti lancia l'eccezione `UnboundedSetException`. Se l'insieme è limitato, viene memorizzata la dimensione dell'insieme nella variabile `n`, l'oggetto viene poi inserito attraverso il metodo `ins` e infine viene controllato se la dimensione è aumentata. Restituisce `true` se la dimensione è cambiata e quindi è stato aggiunto l'oggetto, altrimenti ritorna `false`.
Vediamo qualche esempio di utilizzo del metodo `add`:

Inserimento di oggetti e di variabili logiche inizializzate (e non) in un insieme completamente specificato:

```
Lvar x = new Lvar("x",3);
Object[] s_elems = {1,2,x};
MutableLSet s = new ConcreteMutableLSet("s",s_elems);
s.output();
Lvar y = new Lvar("y",5);
Lvar z = new Lvar("z");
s.add(y);
s.add(z);
s.add(4);
s.output();

/* OUTPUT:
 * s = {1,2,3}
 * s = {4,5,_z,1,2,3}
 */
```

L'insieme `s` contiene gli interi `1` e `2` e la variabile logica `x` inizializzata a `3` e con il metodo `add` viene aggiunta un'altra variabile logica inizializzata (`y`), una non inizializzata (`z`) e un intero che modifica l'insieme `s`.

Inserimento di oggetti e di variabili logiche inizializzate (e non) in un insieme non completamente specificato:

```
Lvar x = new Lvar("x");
MutableLSet s =
    new ConcreteMutableLSet("s",MutableLSet.empty.ins(x)); // s = {_x}
s.output();
s.add(1);
Lvar y = new Lvar("y",2);
```

```
s.add(y);
Lvar z = new Lvar("z");
s.add(z);
s.output();

/* OUTPUT:
 * s = {_x}
 * s = {_z,2,1,_x}
 */
```

L'insieme `s` contiene la variabile logica non inizializzata `x`, ovvero non è completamente specificato. Il metodo `add` aggiunge ad `s` l'intero `1` e due variabili logiche, una inizializzata (`y`) e una no (`z`).

Inserimento di elementi in un insieme illimitato o non inizializzato:

```
MutableLSet s = new ConcreteMutableLSet("s",false);
MutableLSet s2 = new ConcreteMutableLSet("s2",s.ins(2));
s2.output();
s2.add(3);

/* OUTPUT:
 * s2 = {2|s}
 * Exception in thread "main" JSetL.UnboundedSetException
 */
```

Viene lanciata l'eccezione `UnboundedSetException` perché l'insieme `s` è illimitato.

`Unione`

Il metodo `addAll` inserisce nell'insieme tutti gli elementi della collezione. In questo modo implementa l'unione tra l'oggetto di invocazione e la collezione passata come parametro:

```
public boolean addAll(Collection c) {
  if(set.isBound()) {
   if (c instanceof LSet) {
       ConcreteMutableLSet tmp = new ConcreteMutableLSet(false);
       boolean result =
           solver.boolSolve(tmp.union(((ConcreteMutableLSet) c),set));
       set = tmp.set;
       return result;
   }
   else {
       Iterator it = c.iterator();
       boolean result = true;
       while (it.hasNext())
           result = result && add(it.next());
       return result;
   }
  }
  else throw new UnboundedSetException();
}
```

Questo medoto funziona solo per insiemi limitati e inizializzati e si può dividere in due parti:

1. la prima implementa l'unione nel caso in cui la collezione passata come parametro sia un `LSet`: viene creato un oggetto temporaneo `tmp` della classe stessa non inizializzato, che prende il valore dell'unione tra l'oggetto di invocazione e la collezione passata come parametro. Questo risultato viene poi assegnato all'attributo `set` dell'oggetto chiamante, modificandolo. Infine viene restituito il valore ottenuto dal metodo `boolSolve`, richiamato dall'oggetto `solver` della classe `SolverClass`, che ritorna `true` se la `union` è andata a buon fine, `false` altrimenti.

2. la seconda implementa l'unione tra un `LSet` e una qualsiasi collezione:

viene utilizzato l'iteratore della collezione per scorrere tutti gli elementi e aggiungerli uno per uno all'insieme attraverso il metodo `add`. Restituisce `true` se l'insieme ha cambiato dimensione, `false` altrimenti.

Vediamo qualche esempio di utilizzo del metodo `addAll`:

Unione di due insiemi non completamente specificati:

```
Lvar x1 = new Lvar("x",3);
Lvar x2 = new Lvar("x2");
Object[] s1_elems = {1,2,x1,x2};
MutableLSet s1 = new ConcreteMutableLSet("s1",s1_elems);
s1.output();
Lvar y1 = new Lvar("y1",5);
Lvar y2 = new Lvar("y2");
Object[] s2_elems = {4,y1,y2};
MutableLSet s2 = new ConcreteMutableLSet("s2",s2_elems);
s2.output();
s2.addAll(s1);
s2.output();

/* OUTPUT:
 * s1 = {1,2,3,_x2}
 * s2 = {4,5,_y2}
 * s2 = {1,2,3,_x2,4,5,_y2}
 */
```

Gli insiemi `s1` e `s2` contengono variabili logiche non inizializzate e il metodo `addAll` aggiunge tutti gli elementi di `s1` a `s2`.

Unione di due insiemi di cui l'oggetto di invocazione è non inizializzato:

```
//s1 come nell'esempio precedente, s1 = {1,2,3,_x2}
```

```
MutableLSet s2 = new ConcreteMutableLSet(false);
s2.addAll(s1);

/* OUTPUT:
 * Exception in thread "main" JSetL.UnboundedSetException
 */
```

Se l'oggetto di invocazione, in questo caso `s2`, non è inizializzato, viene lanciata l'eccezione `UnboundedSetException`. Si noti che il metodo `addAll` non fa controlli sull'oggetto passato per parametro, quindi se nell'esempio precedente `s1` fosse illimitato e `s2` specificato, il risultato di `s2.addAll(s1)` sarebbe `true` e `s2` diventerebbe illimitato.

Unione di un insieme con una collezione, ad esempio con un oggetto della classe `Vector`:

```
Vector v = new Vector();
for(int i = 1; i < 5; i++)
   v.add(i);   // v = [1,2,3,4]
Lvar x = new Lvar("x",7);
Lvar y = new Lvar("y");
Object[] s_elems = {10,x,y};
MutableLSet s = new ConcreteMutableLSet("s",s_elems);
s.addAll(v);
s.output();

/* OUTPUT:
 * s = {4,3,2,1,10,7,_y}
 */
```

In questo esempio si vede che si può fare l'unione anche tra un insieme, `s`, e un oggetto di una qualsiasi classe concreta che implementa l'interaccia `Collection`, ad esempio un vettore.

5.3.2 Appartenenza e inclusione: `contains` e `containsAll`

`Appartenenza`

Il metodo `contains` verifica se l'insieme contiene (almeno) un elemento uguale all'oggetto passato come parametro.

```
public boolean contains(Object o) {
    if (set.isBound()) {
        Lvar lv = new Lvar(o);
        return solver.boolSolve(lv.in(this.set));
    }
    else throw new UnboundedSetException();
}
```

Il metodo `contains` ritorna `true` se l'insieme contiene l'oggetto `o`, `false` altrimenti. In questo metodo si controlla se l'insieme è limitato e inizializzato: se `isBound` ritorna `true` viene costruita una variabile logica a cui viene dato il valore dell'oggetto `o` e viene restituito il valore booleano dato dalla `boolSolve`, che risolve il vincolo insiemistico `in`; se `isBound` ritorna `false` viene lanciata l'eccezione `UnboundedSetException`.
Vediamo qualche esempio di utilizzo del metodo `contains`:

Appartenenza di un elemento ad un insieme non completamente specificato:

```
Lvar x = new Lvar("x");
Object[] s_elems = {1,x};
MutableLSet s = new ConcreteMutableLSet("s",s_elems);
System.out.println("s.contains(2): " + s.contains(2));
s.output();
x.output();

/* OUTPUT:
```

```
 * s.contains(2): true
 * s = {1,2}
 * x = 2
 */
```

L'insieme `s` è formato dall'elemento `1` e dalla variabile logica non inizializzata `x`. Il metodo `contains` lega il valore `2` alla variabile logica `x` non inizializzata e ritorna `true`.

Appartenenza di un elemento ad un insieme non inizializzato:

```
MutableLSet s1 = new ConcreteMutableLSet(false);
System.out.println("s1.contains(1): " + s1.contains(1));

/* OUTPUT:
 * s1 = {1|LSet_0}
 * Exception in thread "main" JSetL.UnboundedSetException
 */
```

L'insieme `s1` è illimitato, pertanto il metodo `contains` lancia l'eccezione. Lo stesso succede con un insieme non inizializzato.

`Inclusione`

Il metodo `containsAll` verifica se questa lista contiene tutti gli elementi contenuti nella collezione `c`. In questo modo implementa l'inclusione nell'insieme della collezione passata come parametro:

```
public boolean containsAll(Collection c) {
    if (set.isBound()) {
        if (c instanceof LSet)
            return solver.boolSolve(((ConcreteMutableLSet) c).subset(set));
        else {
            boolean result = true;
```

```
            Iterator it = c.iterator();
            while (it.hasNext())
                result = result && contains(it.next());
            return result;
        }
    }
    else throw new UnboundedSetException();
}
```

Anche il metodo `containsAll` controlla se l'insieme è limitato e inizializzato e in caso contrario lancia un'eccezione. Se invece l'insieme è limitato il metodo si può dividere in due parti come `addAll`: la prima parte risolve il vincolo dell'inclusione (`subset`) se la collezione passata come parametro è un `LSet`, la seconda itera gli elementi della collezione e per ognuno di essi richiama la `contains`. Ritorna `true` se l'insieme contiene tutti gli oggetti della collezione, `false` altrimenti.
Vediamo qualche esempio di utilizzo del metodo `containsAll`:

Inclusione di un insieme contenente variabili logiche inizializzate:

```
Lvar x = new Lvar("x",4);
Lvar y = new Lvar("y",6);
Object[] c1_elems = {2,x,6};
Object[] c2_elems = {2,4,y,8,10};
MutableLSet c1 = new ConcreteMutableLSet("c1",c1_elems);
MutableLSet c2 = new ConcreteMutableLSet("c2",c2_elems);
c1.output();
c2.output();
System.out.println("c2.containsAll(c1): " + c2.containsAll(c1));

/* OUTPUT:
 * c1 = {2,4,6}
 * c2 = {2,4,6,8,10}
```

```
 * c2.containsAll(c1): true
 */
```

L'insieme `c1` contiene (oltre a dei numeri interi) la variabile logica `x` inizializziata a `4` e l'insieme `c2` contiente la variabile logica `y` inizializzata a `6`. Quando viene richiamato il metodo `containsAll` su `c2`, viene verificato che l'insieme è limitato e ritorna `true`, in quanto `c1` è incluso in `c2`.

\# Inclusione di una collezione in un insieme parzialmente specificato:

```
 Vector v = new Vector();
 v.add(1);
 v.add(2);
 Lvar y = new Lvar("y");
 Object[] c1_elems = {y,2,4,6};
 MutableLSet c1 = new ConcreteMutableLSet("c1",c1_elems);
 System.out.println("v: " + v);
 c1.output();
 System.out.println("c1.containsAll(v): " + c1.containsAll(v));
 y.output();

/* OUTPUT:
 * v: [1, 2]
 * c1 = {_y,2,4,6}
 * c1.containsAll(v): true
 * y = 1
 */
```

L'insieme `c1` contiene una variabile logica non inizializzata. Tramite il metodo `containsAll` gli viene legato il valore `1` e il metodo restituisce `true`.

5.3.3 Estrazione e differenza: `remove` e `removeAll`

`Estrazione`

Il metodo `remove` rimuove dall'insieme l'oggetto passato come parametro, se presente.

```
public boolean remove(Object o) {
    int n = set.size();
    if (set.isBound()) {
        ConcreteMutableLSet tmp = new ConcreteMutableLSet(false);
        Lvar lv = new Lvar(o);
        try {
            boolean result = solver.boolSolve(set.less(lv,(tmp)));
            if (result == true)
                set = tmp.set;
            return result;
        }
        catch(Failure f) {
            System.out.println("Failure catturata");
        }
        return (set.size() < n);
    }
    else throw new UnboundedSetException();
}
```

Il metodo ritorna `true` se la dimensione dell'insieme è cambiata dopo l'invocazione del metodo, `false` altrimenti. Anche questo metodo funziona solo con insiemi limitati e inizializzati. Viene creato un oggetto temporaneo `tmp` non inizializzato di classe `ConcreteMutableLSet` e una variabile logica con il valore dell'oggetto `o` passato come parametro. Il metodo `less` viene chiamato nel blocco `try` in quanto nella sua dichiarazione viene lanciata l'eccezione `Failure` e deve essere catturata. Dopo la chiamata al metodo `less`, se il valore della variabile è presente nell'insieme `set`, l'oggetto `tmp` contiene tutti

gli elementi dell'insieme eccetto l'elemento `o`. Se la `boolSolve` ritorna `true`, a `set` viene assegnato il valore del campo `set` di `tmp` e viene restituito il risultato.
Vediamo qualche esempio di utilizzo del metodo `remove`:

\# Rimozione di un elemento da un insieme contenente variabili logiche inizializzate:

```
 Lvar x = new Lvar("x");
 Lvar y = new Lvar("y",5);
 Object[] s_elems = {1,y,x,3,4};
 MutableLSet s = new ConcreteMutableLSet("s",s_elems);
 s.output();
 System.out.println("s.remove(2): " + s.remove(2));
 s.output();
 System.out.println("s.remove(y): " + s.remove(y));
 s.output();
 System.out.println("s.remove(5): " + s.remove(5));
 s.output();
 x.output();

/* OUTPUT:
 * s = {1,5,_x,3,4}
 * s.remove(2): true
 * s = {1,5,3,4}
 * s.remove(y): true
 * s = {1,3,4}
 * s.remove(5): false
 * s = {1,3,4}
 * x = 2
 */
```

L'insieme `s` non è completamente specificato in quanto contiene la

variabile `x` non inizializzata. Inoltre contiene anche una varibile logica `y` inizializzata a `5`. La prima volta che si chiama il metodo `remove` si vuole eliminare l'elemento `2` che non è presente nell'insieme, quindi viene legato alla variabile logica `x` che viene eliminata. Poi `remove` viene richiamato per togliere `y`, infatti l'insieme viene privato dell'elemento `5`, valore a cui la variabile era legata. Infine, tentando di togliere ancora il valore `5`, il metodo ritorna `false`, però l'elemento non esiste.

`Differenza`

Il metodo `removeAll` rimuove dall'insieme tutti gli elementi contenuti nella collezione `c`. In questo modo implementa la differenza tra l'insieme e la collezione passata come parametro:

```
public boolean removeAll(Collection c) {
   if (set.isBound()) {
     if (c instanceof LSet) {
        ConcreteMutableLSet tmp = new ConcreteMutableLSet(false);
        boolean result =
           solver.boolSolve(tmp.set.differ(set,(ConcreteMutableLSet) c));
        if (result == true)
           set = tmp.set;
        return result;
     }
     else {
         int n = set.size();
         boolean result = false;
         Iterator it = c.iterator();
         while (it.hasNext()) remove(it.next());
         return (set.size() < n);
     }
 }
 else throw new UnboundedSetException();
```

```
}
```

Il metodo `removeAll` rimuove tutti gli elementi della collezione passata come parametro: ritorna `true` se è cambiata la dimensione dell'insieme, ovvero se è stato tolto almeno un elemento. Se la collezione è un `LSet` viene utilizzato il metodo `differ`, altrimenti si itera la collezione e si richiama il metodo `remove` per togliere ogni elemento.
Vediamo qualche esempio di utilizzo del metodo `removeAll`:

Rimozione di un insieme da un insieme contenente variabili logiche inizializzate:

```
Lvar x = new Lvar("x",2);
Object[] c1_elems = {1,x,3,4};
MutableLSet c1 = new ConcreteMutableLSet("c1",c1_elems);
Lvar y = new Lvar("y",5);
c1.output();
Object[] c2_elems = {1,y};
MutableLSet c2 = new ConcreteMutableLSet("c2",c2_elems);
c2.output();
System.out.println("c1.removeAll(c2): " + c1.removeAll(c2));
c1.output();

/* OUTPUT:
 * c1 = {1,2,3,4}
 * c2 = {1,5}
 * c1.removeAll(c2): true
 * c1 = {2,3,4}
 */
```

L'insieme `c1` contiene la variabile logica `x` inizializzata a `2` e l'insieme `c2` contiente la variabile logica `y` inizializzata a `5`. Quando viene richiamato il metodo `removeAll` su `c2`, viene verificato che l'insieme

è limitato e ritorna `true`, in quanto è stato eliminato l'elemento `1`. Si noti che non devono essere necessariamente rimossi tutti gli elementi affinchè il risultato sia `true`.

Rimozione di una collezione da un insieme:

```
Vector v = new Vector();
v.add(2);
v.add(4);
System.out.println("v: " + v);
Lvar x = new Lvar("x",2);
Object[] c1_elems = {1,x,3,4};
MutableLSet c1 = new ConcreteMutableLSet("c1",c1_elems);
c1.output();
System.out.println("c1.removeAll(v): " + c1.removeAll(v));
c1.output();


/* OUTPUT:
 * v = [2,4]
 * c1 = {1,2,3,4}
 * c1.removeAll(v): true
 * c1 = {1,3}
 */
```

Il vettore `v` di elementi `2` e `4` viene rimosso dall'insieme `c1`.

5.3.4 Intersezione: `retainAll`

Il metodo `retainAll` rimuove dall'insieme tutti gli elementi che non sono contenuti nella collezione `c`. In questo modo implementa l'intersezione tra l'insieme e la collezione passata come parametro:

```
public boolean retainAll(Collection c) {
```

```
  if(set.isEmptySL() && c.isEmpty()) return false;
  if (set.isBound()) {
      if (c instanceof LSet) {
          ConcreteMutableLSet tmp = new ConcreteMutableLSet(false);
          boolean result =
            solver.boolSolve(tmp.inters(set,(ConcreteMutableLSet) c));
          if (result == true)
            set = tmp.set;
          return result;
      }
      else {
          Iterator it = c.iterator();
          ConcreteMutableLSet tmp = new ConcreteMutableLSet();
          while (it.hasNext()) {
              Object tmpob = it.next();
              if (contains(tmpob))
                  tmp.add(tmpob);
          }
          set = tmp.set;
          return true;
      }
  }
  else throw new UnboundedSetException();
 }
```

Questo metodo ritorna `false` quando sia l'oggetto d'invocazione che la collezione passata come parametro sono vuoti. Altrimenti controlla che l'insieme sia limitato e inizializzato e se la collezione è un `LSet` utilizza il metodo `inters`; per ogni altra collezione crea un iteratore e un oggetto di classe `ConcreteMutableLSet` temporaneo in cui vengono inseriti gli elementi in comune tra l'oggetto di invocazione e la collezione. Infine l'attributo `set` di `tmp` viene copiato nel campo `set` dell'oggetto chiamante.

Vediamo qualche esempio di utilizzo del metodo `retainAll`:

\# Intersezione non vuota tra due insiemi completamente specificati:

```
Lvar x = new Lvar("x",5);
Object[] s1_elems = {1,2,x};
MutableLSet s1 = new ConcreteMutableLSet("s1",s1_elems);
Lvar y = new Lvar("y",2);
Object[] s2_elems = {1,3,y};
MutableLSet s2 = new ConcreteMutableLSet("s2",s2_elems);
s1.output();
s2.output();
System.out.println("s1.retainAll(s2): " + s1.retainAll(s2));
s1.output();

/* OUTPUT:
 * s1 = {1,2,5}
 * s2 = {1,3,2}
 * s1.retainAll(s2): true
 * s1 = {1,2}
 */
```

I due insiemi `s1` e `s2` hanno elementi in comune, quindi il metodo `retainAll` restutuisce `true`. Se entrambi gli insiemi fossero vuoti, il metodo restituirebbe `false`.

\# Intersezione tra due insiemi di cui uno non completamente specificato:

```
 Lvar x = new Lvar("x");
Object[] s1_elems = {1,2,x};
MutableLSet s1 = new ConcreteMutableLSet("s1",s1_elems);
Lvar y = new Lvar("y",2);
Object[] s2_elems = {1,3,y};
MutableLSet s2 = new ConcreteMutableLSet("s2",s2_elems);
```

```
s1.output();
s2.output();
System.out.println("s1.retainAll(s2): " + s1.retainAll(s2));
s1.output();
x.output();

/* OUTPUT:
 * s1 = {1,2,_x}
 * s2 = {1,3,2}
 * s1.retainAll(s2): true
 * s1 = {1,2}
 * x = unknown
 */
```

L'insieme `s1`, su cui è richiamato il metodo `retainAll`, contiene una variabile logica non inizializzata che non viene legata a nessun valore. Dopo la chiamata al metodo, `s1` contiene l'intersezione degli elementi noti.

\# Intersezione con un insieme non inizializzato:

```
Lvar x = new Lvar("x",5);
Object[] s1_elems = {1,2,x};
MutableLSet s1 = new ConcreteMutableLSet("s1",s1_elems);
MutableLSet s2 = new ConcreteMutableLSet(false);
System.out.println("s1.retainAll(s2): " + s1.retainAll(s2));

/* OUTPUT:
 * Exception in thread "main" JSetL.UnboundedSetException
 */
```

In questo caso `s1`, oggetto di invocazione, è non inizializzato, quindi viene lanciata l'eccezione `UnboundedSetException`.

Intersezione tra un insieme e una collezione:

```
Vector v = new Vector();
v.add(5);
v.add(3);
v.add(2);
System.out.println("v: " + v);
Lvar y = new Lvar("y",7);
Object[] s_elems = {1,3,5,y};
MutableLSet s = new ConcreteMutableLSet("s",s_elems);
s.output();
System.out.println("s.retainAll(v): " + s.retainAll(v));
s.output();

/* OUTPUT:
 * v: [5, 3, 2]
 * s = {1,3,5,7}
 * s.retainAll(v): true
 * s = {3,5}
 */
```

Dopo la chiamata al metodo `retainAll`, l'insieme `s` contiene gli elementi in comune (l'intersersezione) tra l'insieme stesso e il vettore `v`.

5.3.5 Uguaglianza: `equals`

Bisogna porre particolare attenzione al metodo `equals`, in quanto esiste sia nell'interfaccia `LSet` che nell'iterfaccia `java.util.Set`, con la differenza che il primo prende come parametro un `LSet`, mentre il secondo prende un `Object`. Nella classe `ConcreteMutableLSet` vengono implementati entrambi, ovvero viene fatto l'overloading del metodo `equals`: quando l'oggetto passato come parametro è un `LSet` viene richiamato il metodo `equals(LSet`

s), in tutti gli altri casi viene richiamato il metodo `equals(Object o)`.
Vediamo l'implementazione del metodo `equals` di `LSet`:

```
public boolean equals(LSet s) {
   if(set.isBound())
       return solver.boolSolve(set.eq(s));
   else throw new UnboundedSetException();
}
```

Questo metodo viene chiamato solo quando il parametro passato è un `LSet`: si controlla che l'insieme sia limitato e inizializzato e si risolve il vincolo `eq`.
Vediamo qualche esempio di utilizzo del metodo `equals(LSet s)`:

Equivalenza tra insiemi di `MutableLSet`:

```
Lvar x = new Lvar("x");
Object[] s1_elems = {x,2,3};
MutableLSet s1 = new ConcreteMutableLSet("s1",s1_elems);
Lvar y = new Lvar("y",3);
Object[] s2_elems = {2,1,y};
MutableLSet s2 = new ConcreteMutableLSet("s2",s2_elems);
s1.output();
s2.output();
System.out.println("s1.equals(s2): " + s1.equals(s2));
x.output();
MutableLSet s3 =
    new ConcreteMutableLSet("s3",MutableLSet.empty.ins(1).ins(y));
s3.output();
System.out.println("s1.equals(s3): " + s1.equals(s3));

/* OUTPUT:
 * s1 = {1,2,3}
 * s2 = {2,1,3}
```

```
* s1.equals(s2): true
* x = 1
* s3 = {3,1}
* s1.equals(s3): false
*/
```

I due insiemi `s1` e `s2` contengono rispettivamente la variabile logica `x` non inizializzata e `y` inizializzata a `3`. Il metodo `equals` lega la variabile `x` a `1` e in questo modo i due insiemi sono uguali. Invece viene restituito `false` per gli insiemi `s1` e `s3`.

Più generale è invece il metodo `equals(Object o)` in quanto può confrontare l'insieme `LSet` con un qualsiasi insieme di `java.util.Set`.

```
public boolean equals(Object o) {
    if (!(o instanceof Set))
        return false;
    Collection c = (Collection) o;
    if (c.size() != size())
        return false;
    return containsAll(c);
}
```

Se l'oggetto non è un `Set`, allora ritorna `false`; altrimenti l'oggetto `o` viene trasformato in una `Collection`: se la dimensione della collezione è diversa da quella dell'oggetto di invocazione ritorna `false`, altrimenti restituisce il valore del metodo `containsAll`.
Vediamo qualche esempio di utilizzo del metodo `equals(Object o)`:

Equivalenza tra insiemi di `MutableLSet` e di `java.util.Set`:

```
// s1 come nell'esempio precedente (con x = 1)
// s3 come nell'esempio precedente
s1.output();
```

```
s3.output();
HashSet hset = new HashSet();
hset.addAll(s1);
System.out.println("hset: " + hset);
System.out.println("hset.equals(s1): " + hset.equals(s1));
System.out.println("s1.equals(hset): " + s1.equals(hset));
System.out.println("hset.equals(s3): " + hset.equals(s3));

/* OUTPUT:
 * s1 = {1,2,3}
 * s3 = {3,1}
 * hset: [2, 1, 3]
 * s.equals(s1): true
 * s1.equals(s): true
 * hset.equals(s3): false
 */
```

Gli insiemi `s1` e `s3` sono gli stessi dell'esempio precedente, in più viene creato un insieme `HashSet` aggiungendogli tutti gli elementi di `s1`. I due insiemi, `hset` e `s1` vengono confrontati sia con l'`equals` di `HashSet` che con quello di `MutableLSet`. In entrambi i casi i due insiemi sono uguali, quindi i metodi restituiscono `true`. Viene poi confrontato l'insieme `hset` con `s3` e il metodo ritorna `false`, in quanto i due insiemi sono diversi.

Si noti che questo metodo permette di poter confrontare un `LSet` completamente specificato con una delle classi che implementano l'interfaccia `Set`, come `HashSet` e `TreeSet`.

5.3.6 `iterator`

Il metodo `iterator` restituisce un iteratore sugli elementi dell'insieme, creando un oggetto della classe `LSetIterator`.

```
public Iterator iterator() {
   if (set.isBound()) return new LSetIterator(set);
   else throw new UnboundedSetException();
}
```

Il metodo controlla che l'insieme sia limitato e inizializzato e se lo è crea un oggetto di tipo `LSetIterator`, altrimenti lancia un'eccezione.

La classe `LSetIterator` implementa l'interfaccia `Iterator`, la quale fornisce la dichiarazione di tre metodi: un metodo `next()`, che avanza l'iteratore e restituisce il valore puntato; un metodo `hasNext()`, che determina se tutti gli elementi dell'insieme sono stati visitati ritornando un valore booleano; e può opzionalmente supportare un metodo `remove()`, che toglie dall'insieme l'elemento visitato per ultimo. Nella classe `LSetIteratore` vengono implementati solo i primi due metodi.

```
public class LSetIterator implements Iterator {
    private LSet set;
    private int n = 0;
    public LSetIterator(LSet s) {
        set = s;
        set = set.normalizeSet();
    }
    public Object next() {
        Object ob = set.get(n++);
        if(ob instanceof Lvar && ((Lvar)ob).iniz))
            ob = ((Lvar)ob).val;
        return ob;
    }
    public boolean hasNext() {
        return (n < set.size());
    }
```

```
    public void remove() {}
}
```

La classe `LSetIterator` ha due attributi: un oggetto `LSet` (`set`) e un intero (`n`) che viene incrementato per scorrere gli elementi. Il costruttore prende come parametro un `LSet` che assegna al proprio attributo e su di esso richiama il metodo `normalizeSet` per togliere evenutali elementi duplicati memorizzati nell'insieme. Il metodo `next` memorizza in un oggetto `ob` l'elemento successivo dell'insieme richiamando il metodo `get(n++)` sull'attributo `set`; viene controllato se l'oggetto `ob` è un'istanza della classe `Lvar` e se è inizializzato, in tal caso viene memorizzato solo il suo valore; infine si restituisce `ob`. Il metodo `hasNext` restituisce un booleano, in particolare ritorna `true` se `n` è minore della dimensione dell'insieme, `false` altrimenti. Il metodo `remove` non viene implementato.
Il seguente esempio mostra un semplice uso degli iteratori:

Iterazione di un insieme non completamente noto:

```
Lvar x = new Lvar("x");
Lvar y = new Lvar("y",4);
Object[] ob_elems = {2,3,x,y};
MutableLSet s = new ConcreteMutableLSet(ob_elems);
Iterator it = s.iterator();
while (it.hasNext())
   System.out.println(it.next());

/* OUTPUT:
 * 4
 * Lvar: x, id: 0, is NOT initialized.
 * 3
 * 2
 */
```

Viene creata una variabile logica non inizializzata `x` e una variabile `y` con valore `4`. Viene poi costruito un array di oggetti `ob_elems` formato dagli interi `2` e `3` e dalle varibili logiche `x` e `y`. L'insieme `s` viene inzializzato con il valore di `ob_elems`; con l'istruzione `Iterator it = s.iterator()` viene costruito un iteratore di `s`. Il ciclo `while` stampa tutti gli oggetti dell'insieme: finché esiste un elemento successivo (`it.hasNext == true`), avanza al successivo e lo stampa.

Appena dopo essere stato creato, un iteratore punta a un valore speciale che precede il primo elemento, cosicché il primo elemento si ottiene con la prima chiamata a `next()`.

5.3.7 `size`

Il metodo `size` resituisce la dimensione dell'insieme, solo se questo è limitato, altrimenti lancia una eccezione.

```
public int size() {
   if(set.isBound())
       return set.size();
   else throw new UnboundedSetException();
}
```

Il metodo `size` controlla che l'insieme sia limitato e inizializzato con il metodo `isBound` e in tal caso richiama il metodo `size` dell'attributo `set`, altrimenti lancia un'eccezione.
Vediamo qualche esempio di utilizzo del metodo `size`:

\# Dimensione di un insieme:

```
MutableLSet s1 = new ConcreteMutableLSet("s1");
s1.output();
System.out.println("s1.size(): " + s1.size());
s1.add(1);
```

```
Lvar x = new Lvar("x",3);
s1.add(x);
s1.output();
System.out.println("s1.size(): " + s1.size());
Lvar y = new Lvar("y");
s1.add(y);
s1.output();
System.out.println("s1.size(): " + s1.size());
MutableLSet s2 = new ConcreteMutableLSet(false);
s2.size();

/* OUTPUT:
 * emptySet = {}
 * s1.size(): 0
 * s1 = {3,1}
 * s1.size(): 2
 * s1 = {_y,3,1}
 * s1.size(): 3
 * Exception in thread "main" JSetL.UnboundedSetException
 */
```

L'insieme `s1` viene creato vuoto, quindi la `size` restituisce il valore `0`. Dopo vengono inseriti nell'insieme una variabile inizializzata e un intero, quindi la dimensione dell'insieme diventa `2`. Infine si inserisce una variabile logica non inizializzata e la `size` restituisce il valore `3`. Con un insieme non inizializzato o illimitato, il metodo `size` lancia l'eccezione `UnboundedSetException`.

5.3.8 isEmpty

Il metodo `isEmpty` verifica se l'insieme è vuoto.

```
public boolean isEmpty() {
```

```
    return set.isEmpty();
 }
```

Questo metodo semplicemente richiama il metodo omonimo sull'attributo `set`.

Vediamo qualche esempio di utilizzo del metodo `isEmpty`:

Controllo se un insieme è vuoto:

```
MutableLSet s1 = new ConcreteMutableLSet("s1");
s1.output();
System.out.println("s1.isEmpty(): " + s1.isEmpty());
Lvar x = new Lvar("x",3);
s1.add(x);
s1.output();
System.out.println("s1.isEmpty(): " + s1.isEmpty());

/* OUTPUT:
 * emptySet = {}
 * s1.isEmpty(): true
 * s1 = {3}
 * s1.isEmpty(): false
 */
```

L'insieme `s1` nel momento in cui viene creato è vuoto, viene poi aggiunta una variabile logica inizializzata e il metodo `isEmpty` ritorna `false`.

5.3.9 `clear`

Il metodo `clear` rimuove dall'insieme tutti gli elementi.

```
 public void clear() {
    set = ConcreteLSet.empty;
 }
```

Il metodo `clear` pone `set` uguale all'insieme vuoto.

Vediamo qualche esempio di utilizzo del metodo `clear`:

Rimozione di tutti gli elementi da un insieme:

```
Lvar x = new Lvar("x");
Lvar y = new Lvar("y",3);
MutableLSet s1 =
    new ConcreteMutableLSet(MutableLSet.empty.ins(1).ins(x).ins(y));
s1.output();
s1.clear();
s1.output();

/* OUTPUT:
 * LSet_3 = {3,_x,1}
 * emptySet = {}
 */
```

L'insieme `s1` viene creato con l'intero `1`, la variabile logica `x` inizializzata a `0` e la varibile logica `y` non inizializzata. Dopo l'utilizzo del metodo `clear`, l'insieme è vuoto.

Capitolo 6

Esempi

In questo capitolo si vuole mostrare come l'interfaccia `MutableLSet` e la classe `ConcreteMutableLSet`, che la implementa, possano essere usate sfruttando i metodi di `LSet`, di `java.util.Set` e anche in modo ibrido, cioè utilizzando metodi di entrambe le interfacce nella stessa applicazione. Per far questo vediamo tre programmi che utilizzano l'interfaccia `MutableLSet` e la classe `ConcreteMutableLSet`: due che utilizzano separatamente i metodi delle due interfacce `LSet` e `java.util.Set`; uno che utilizza metodi di entrambe le interfacce.

6.1 Utilizzo degli insiemi di JSetL

Come primo esempio vediamo un programma in cui si usano gli insiemi e le relative operazione previste in JSetL.

6.1.1 La classe `Coloring`

Data una mappa di n regioni $R_1, \ldots, R_n$ e un insieme di m colori $c_1, \ldots, c_m$, il problema consiste nel trovare un assegnamento di colori per le regioni della mappa tale che le regioni vicine abbiano colori differenti.
La classe `Coloring` ha un metodo `coloring` che prende come parametri tre insiemi chiamati `reg` (regioni), `map` (mappa) e `col` (colori).

```
public static void coloring(MutableLSet reg, MutableLSet map, MutableLSet col)
throws Failure  {
    Lvar Y = new Lvar();
    Solver.add(reg.eq(col));
    Solver.forall(Y, col, MutableLSet.empty.ins(Y).nin(map));
    Solver.solve();
    return;
}
```

Nel metodo viene viene creata una variabile logica `Y` non inizializzata, viene aggiunto al constraint store il vincolo che i colori e le regioni devono essere uguali (attraverso il metodo `eq`), tramite il metodo `forall` della classe `SolverClass` si introduce il vincolo che per ogni elemento `Y` appartenente all'insieme dei colori, `Y` non appartenga (`nin`) a `map`. Infine viene chiamata la `solve` per risolvere tutti i vincoli.

Un possibile `main` che utilizza il metodo `coloring` è il seguente:

```
public static void main (String[] args) throws Failure {
    Lvar R1 = new Lvar("R1");
    Lvar R2 = new Lvar("R2");
    Lvar R3 = new Lvar("R3");
    Lvar[] n1 = {R1,R2};
    Lvar[] n2 = {R2,R3};
    Lvar[] regions = {R1,R2,R3};
    String[] colors = {"red", "blue"};
    MutableLSet m1 = new ConcreteMutableLSet(n1);
    MutableLSet m2 = new ConcreteMutableLSet(n2);
    MutableLSet Regions = new ConcreteMutableLSet(regions);
    MutableLSet Colors  = new ConcreteMutableLSet(colors);
    MutableLSet Map =
        new ConcreteMutableLSet("Map", MutableLSet.empty.ins(m2).ins(m1));
    coloring(Regions, Map, Colors);
```

```
R1.output();
R2.output();
R3.output();
Map.output();
```

Nel `main` viene creato l'insieme `Regions` che contiene un array di tre variabili logiche non inizializzate (`Regions` $= \{R_1, R_2, R_3\}$), viene costruito l'insieme `Map` che a sua volta contiene due insiemi (`Map` $= \{\{R_1, R_2\}, \{R_2, R_3\}\}$), viene creato l'insieme `Colors` tramite un array di `String` (`Colors` $= \{''red'','' blue''\}$. A questo punto viene richiamato il metodo `coloring` sui tre insiemi creati e viene stampata una possibile soluzione, ad esempio:

```
R1 = red
R2 = blue
R3 = red
Map = {{red,blue},{blue,red}}
```

6.1.2 Le modifiche a `Coloring`

Nella classe `Coloring` sono state fatte alcune modifiche riguardanti gli insiemi: non viene più usata la classe `Set` originale di JSetL, ma vengono utilizzate l'interfaccia `MutableLSet` e la classe `ConcreteMutableLSet`. Vediamo in particolare quali modifiche sono state fatte:

- Nel metodo `coloring` gli insiemi passati come parametro vengono modificati da `Set` a `MutableLSet`. In generale in tutti i metodi che prendono come parametro un insieme si deve fare questa modifica;

- Il terzo parametro del metodo `forall` non è più `Set.empty.ins(x).nin(map)`, ma `MutableLSet.empty.ins(x).nin(map)`. Ogni volta che si trova la variabile statica `empty` bisogna modificare da `Set` a `MutableLSet`;

- Nel `main` tutti gli insiemi originariamente creati nel seguente modo

  ```
  Set s = new Set(parametri);
  ```

vengono cosi modificati:

```
MutableLSet s = new ConcreteMutableLSet(parametri);
```

6.2 Utilizzo degli insiemi di `java.util.Set`

Come esempio vediamo un algoritmo che, dato un insieme di numeri interi, restituisce l'elemento con valore massimo, utilizzando solo i metodi dell'interfaccia `java.util.Set`.

6.2.1 La classe `Max`

La classe `Max` calcola il massimo di un insieme di interi, attraverso il proprio metodo `maxInt`.

```
public static int maxInt(MutableLSet s) {
   int max = 0;
   if(s.isEmpty()) return 0;
   else{
       Iterator it = s.iterator();
       while (it.hasNext()) {
       Object ob = it.next();
       if(ob instanceof Integer)
           if((Integer)ob > max)
               max = (Integer)ob;
       }
       return max;
   }
}
```

Il metodo `maxInt` prende come parametro un `MutableLSet` e restituisce un intero. Al suo interno crea una variabile intera `max` che inizializza a `0`. Se l'insieme è vuoto, allora ritorna il valore `0`; altrimenti crea un iteratore e confronta ogni elemento dell'insieme con la variabile `max`: se l'elemento

confrontato è maggiore, allora a `max` viene assegnato il suo valore. Infine si restituisce la variabile `max`. Nel ciclo `while` viene fatto un cast esplicito, in quanto il metodo `next` dell'iteratore restituisce un `Object`, ma su questo non è possibile fare confronti.
Vediamo ora il `main` in cui viene costruito l'insieme:

```
public static void main (String[] args) {
    MutableLSet s = new ConcreteMutableLSet();
    int[] a = {1,6,4,8,10,5};
    for(int i = 0; i < a.length; i++)
        s.add(a[i]);
    System.out.print("Max = " + maxInt(s));
}
```

L'insieme `s` viene creato con l'interfaccia `MutableLSet` e la classe concreta `ConcreteMutableLSet`. Viene poi costruito un array di interi, che viene inserito nell'insieme `s` scorrendo gli elemento dell'array con un ciclo `for` e aggiungendo ad `s` ogni elemento con il metodo `add`. Infine viene stampato il risultato richiamando il metodo `maxInt` sull'insieme `s` e il cui risultato è `0`.

In questo semplice esempio si può notare che l'utilizzo dei metodi `isEmpty`, `iterator` e `add` di `ConcreteMutableLSet` ha la stessa sintassi e semantica richiesta dall'interfaccia `java.util.Set`.
Si noti, inoltre, che la medesima dichiarazione del metodo `maxInt` può essere utilizzata anche per un'implementazione dichiarativa, senza dover modificare nulla nel programma chiamante e viceversa.

6.3 Un esempio ibrido

Vediamo un esempio in cui si utilizzano sia metodi di `java.util.Set` che metodi di `LSet`, ovvero degli insiemi di JSetL originale.

La classe `Sum` calcola la somma degli elementi di un insieme, attraverso il proprio metodo `sum`.

```
SolverClass Solver = new SolverClass();
public static void sum(MutableLSet s, Lvar sSum) throws Failure {
   if(s.isEmpty()) Solver.add(sSum.eq(0));
   else{
       Iterator it = s.iterator();
       int sumElem = 0;
       while (it.hasNext()) {
           Object ob = it.next();
           if(ob instanceof Integer)
               sumElem += (Integer)ob;
       }
       Solver.add(sSum.eq(sumElem));
   }
   Solver.solve();
}
```

Il metodo `sum` prende come parametri un `MutableLSet s`, che contiene l'insieme di cui si fa la somma degli elementi, e un `Lvar sSum`, in cui viene memorizzata la somma. Se l'insieme è vuoto, viene aggiunto nel constraint store il vincolo che la variabile logica `sSum` vale `0` (`sSum.equ(0)`), tramite l'oggetto `Solver` della classe `SolverClass`. Altrimenti, se l'insieme non è vuoto, viene creato un iteratore dell'insieme e una variabile intera `sumElem` inizializzata a `0`. Nel ciclo `while` vengono iterati gli elementi dell'insieme e, attraverso un cast a `Integer` dell'oggetto restituito dal metodo `next` dell'iteratore, viene memorizzato e sommato ogni valore nella variabile `sumElem`. Fuori dal ciclo viene aggiunto al constraint store il vincolo che la variabile `sSum` sia uguale al valore di `sumElem`. Infine viene chiamata la `solve` sull'oggetto `Solver`.

In `sum` si vede come vengono integrati i metodi dell'interfaccia `java.util.Set` con i metodi della libreria JSetL: della prima si utilizzano gli iteratori per scorrere l'insieme, della seconda si utilizzano i vincoli e la loro risoluzione tramite il metodo `solve`.

Vediamo il `main`, dove viene costruito l'insieme:

```
public static void main(String[] args) throws Failure {
   Lvar a = new Lvar(6);
   Lvar x = new Lvar("sum");
   Object[] ob = {4,3,2,5,a};
   MutableLSet S = new ConcreteMutableLSet("S",ob);  // S = {4,3,2,5,6}
   sum(S,x);
   S.output();
   x.output();
}
```

Vengono create due variabile logiche: `a`, inizializzata con valore `6` e che viene poi inserita tra gli elementi dell'insieme; `x`, non inizializzata, a cui viene dato il nome "`sum`" e che conterrà la somma degli elementi. Viene poi creato un array di `Object`, `ob`, (si noti che non è un array di interi, in quanto si è voluto generalizzare l'esempio inserendo nell'insieme anche una variabile logica inizializzata) e un insieme `S` della classe `ConcreteMutableLSet`, inizializzato con gli elementi di `ob` e con nome "`S`". Viene chiamato il metodo `sum` con parametri `S` e `x` e infine viene stampato l'insieme e la somma dei suoi elementi:

```
// OUTPUT:
 S = {4,3,2,5,6}
 sum = 20
```

Capitolo 7

Conclusioni e lavori futuri

È stata creata un'unica astrazione di insieme su cui poter operare sia con i metodi forniti da `java.util.Set` che con quelli di JSetL, limitando al minimo le modifiche al `package` JSetL. Il nuovo `package` JSetL fornisce quindi anche tutte le funzionalità degli insiemi di Java, mantenendo la piena compatibilità con applicazioni scritte usando le precedenti librerie (a meno di nomi).

In futuro si può migliorare l'efficienza creando una nuova classe che implementa l'interfaccia `MutableLSet` e che al suo interno utilizza, oltre all'oggetto di classe `ConcreteLSet`, un oggetto della classe `HashSet`, per richiamare i metodi dell'interfaccia `java.util.Set` quando l'insieme è completamente specificato.

Inoltre, si potrebbe integrare la classe `Lst` di JSetL e l'interfaccia `java.util.List` in modo analogo a quanto fatto per gli insiemi.

Bibliografia

[1] Gianfranco Rossi, Elio Panegai, Elisabetta Poleo.
JSetL: A Java Library for Supporting Declarative Programming in Java.
Software-Practice & Experience, 37, 2007, p 115-149.

[2] Elio Panegai, Elisabetta Poleo, Gianfranco Rossi.
JSetL User's Manual. Version 1.0. Novembre 2004.

[3] Harvey M. Deitel, Paul J. Deitel.
Java, Fondamenti di programmazione. Apogeo, 2003.

[4] J. Jaffer, M. J. Maher.
Constraint Logic Programming: a Survey.
In Journal Logic Programming 19, 20: 503 - 581, 1994.

[5] Agostino Dovier, Carla Piazza, Enrico Pontelli, Gianfranco Rossi.
Set and Constraint Logic Programming.
ACM Toplas, 22(5) 2000, 861-931.

[6] http://java.sun.com/j2se/1.5.0/docs/api/.