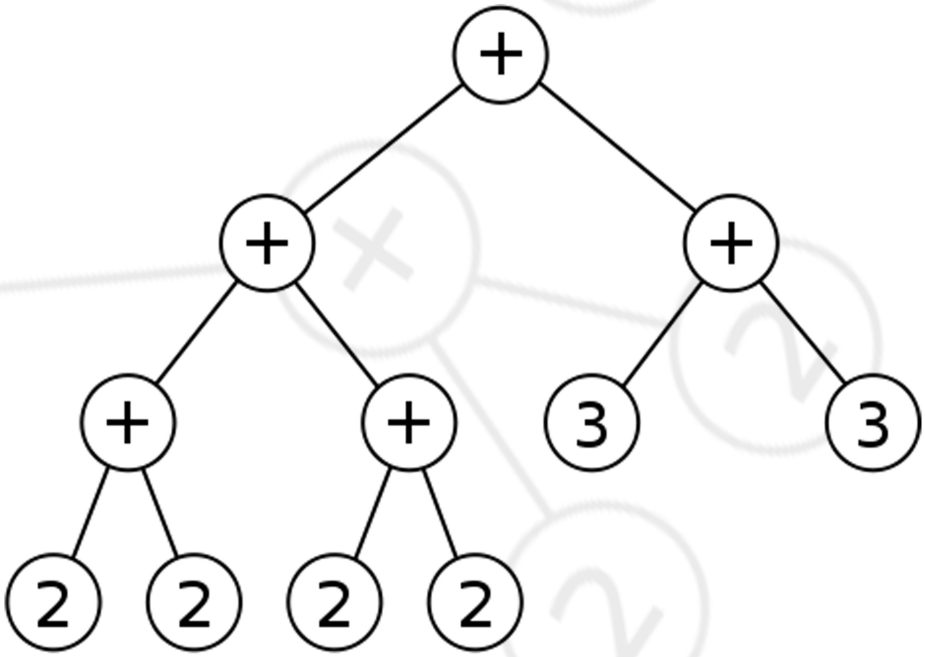
UNIVERSITÀ DELLA CALABRIA
Facoltà di INGEGNERIA

PROGETTO

ALBERO DI UN'ESPRESSIONE ARITMETICA

Giuseppe Sottile \ Mat: 142880

DM 270 Anno 2013



Java

Docente: **Libero Nigro**

CORSO DM 270
Programmazione Orientata agli oggetti in Java

Indice

Introduzione

Lo scopo di queste pagine è quello di presentare la realizzazione di uno dei progetti di fine corso di - “Programmazione orientata agli oggetti in Java”, tenutosi dal Prof. Libero Nigro nell'anno accademico 2013 – DM 270 Unical. Il progetto di programmazione, prevede la gestione di un albero di un'espressione aritmetica “semplice”, che comprende gli usuali operatori binari “+-*/% exp(^)”.

1 Corso di Programmazione Orientata agli Oggetti
Progetto: Albero di un'espressione aritmetica

Si considerano espressioni aritmetiche intere con gli operatori +,-,*,/,%,^ (^ denota l'esponenziazione) e le usuali priorità della matematica: P(^)>P(*,/,%)>P(+,-). A parità di priorità, si assume l'associatività a sinistra. Eventualmente, si possono usare le parentesi per alterare le priorità intrinseche: un'espressione in parentesi () viene valutata prima. Come studiato a lezione, una tale espressione aritmetica può essere convenientemente rappresentata mediante un albero binario. Di seguito si suggerisce l'algoritmo da utilizzare per costruire un albero binario di un'espressione.

Si usano due stack: il primo è uno stack di alberi operandi, il secondo è uno stack di caratteri operatori. Quando arriva un operando, si costruisce un albero con solo l'operando e lo si inserisce in cima allo stack di alberi.

Quando arriva un operatore, sia esso opc (operatore corrente), si procede come segue:
A) se opc è più prioritario dell'operatore affiorante dallo stack di operatori o tale stack è vuoto, si inserisce opc in cima allo stack operatori.

B) Se opc non è più prioritario rispetto alla cima dello stack operatori, si preleva l'operatore al top dello stack operatori, quindi si prelevano due operandi a2 (top) e a1 (top-1) dallo stack di operandi (in caso di eccezioni, l'espressione è malformata). Si crea un nodo operatore con l'operatore prelevato, e gli si legano rispettivamente a1 come figlio sinistro e a2 come figlio destro. Il nuovo albero è inserito in cima allo stack operandi. Si continua ad eseguire il passo B) se opc risulta ancora non più prioritario dell'operatore affiorante dallo stack operatori. Dopo questo, o perché opc è più prioritario dell'operatore in cima allo stack operatori o perché lo stack è vuoto, si applica il caso A. Quando l'espressione è terminata, se lo stack operatori non è vuoto, si estraggono uno alla volta gli operatori dallo stack operatori e si costruiscono alberi con i rispettivi due operandi prelevati dallo stack operandi, secondo le stesse modalità (operando sinistro e destro) spiegate sopra, e si inseriscono tali alberi sullo stack operandi. Si nota che certamente vengono considerati prima gli operatori più prioritari. Quando lo stack operatori è vuoto, allora lo stack operandi dovrebbe contenere un solo elemento che è l'albero dell'espressione. Ogni altra situazione (stack operandi vuoto o con più di un elemento) denota una espressione malformata. Cosa succede se ci sono parentesi ?Quando si incontra una parentesi aperta '(' si invoca ricorsivamente la procedura di costruzione (diciamolabuildEspressione()). Quando si incontra una parentesi chiusa ')', si ritorna l'albero in cima allo stack operandi (un solo elemento o l'espressione è malformata).

Materialmente, il metodo buildEspressione(), potrebbe introdurre i due stack come variabili locali. Quando termina, ritorna l'unico elemento (o la sotto espressione è malformata) in cima allo stack operandi locale. Il metodo buildEspressone() potrebbe ricevere come parametro uno stringtokenizer inizialmente aperto sulla stringa espressione. Il processo di costruzione parte con il metodo build() che riceve una stringa espressione e quindi sub-appalta il lavoro a buildEspressione() che restituisce l'albero dell'espressione. Utilizzare un'espressione regolare per scoprire subito che un'espressione aritmetica non è malformata (condizione necessaria).

L'applicazione dovrebbe essere munita di GUI al fine di consentire la scelta di tutte le operazioni via menu. L'effetto di una qualsiasi operazione (incluso l'iterazione) dovrebbe essere visualizzato graficamente (eventualmente si può rimpiazzare la grafica con una JTextArea usata come console).

2 Progettazione

2.1 Analisi e progettazione (OOAD)

Nella progettazione delle applicazioni non è di buon uso adottare un'idea di stesura per così dire - “diretta”, in cui si procede a priori nel codificare “just-in time”, le idee astratte di progetto in codice ad alto livello. Una buona metodologia di progettazione, prevede in primo luogo una cosiddetta analisi dettagliata di quello che si intende realizzare, cioè di uno studio delle “problematiche” cui deve rispondere l'applicativo; tale fase di progettazione costituisce il momento più importante di progetto e viene identificato come (OOAD) (Object Oriented Analysis Design). E' in questa fase che si prevede cosa un'eventuale classe deve incapsulare internamente, se una informazione piuttosto che un'altra, della idea di interfaccia ADT (Abstract Data Type) che interagisce con l'esterno e della ricerca dei nomi adatti a ruoli-metodi o adatti a dati-membro o attributi. Sostanzialmente in questa fase, si prevede il lavoro più impegnativo, mentre la scrittura in codice, costituisce per così dire una fase di chiusura di un progetto di programmazione. L'approccio ad oggetti (OOP), oramai affermatosi nelle comunità dei programmatori di tutto il mondo dell'informatica, si è rivelato uno strumento molto efficace nella realizzazione di sistemi software complessi. Nella programmazione ad oggetti l'unità fondamentale è la “classe”. In effetti si privilegia il dato piuttosto che l'azione(funzione), rispetto ai linguaggi tipo C, Pascal ecc; il programmatore non deve più preoccuparsi di progettare funzioni o procedure che manipolano dati – ma oggetti di classi che hanno degli attributi e possono svolgere delle azioni e comunicare tra di loro mediante dei messaggi, è questo il potere della o.o.p ed è per questo motivo che linguaggi come Java, C++ o C# per citarne alcuni, hanno avuto un grande successo.

2.2 Diagramma delle classi UML

Descriviamo anzitutto le classi utilizzate nel progetto iniziando la nostra analisi dai diagrammi UML (di classe). Il linguaggio UML è un linguaggio di modellazione grafica e rappresenta lo standard più utilizzato per il design software. Il linguaggio prevede svariati diagrammi, di flusso (o di attività), casi d'uso, azioni, ecc; tutto questo per analizzare e quindi descrivere dai diversi “punti di vista” un'applicazione. Noi faremo uso dei diagrammi di classe per mettere in evidenza quelle che sono le gerarchie di classi adoperate, compresi i metodi e gli attributi incapsulati, come mostrato in figura 2.0.

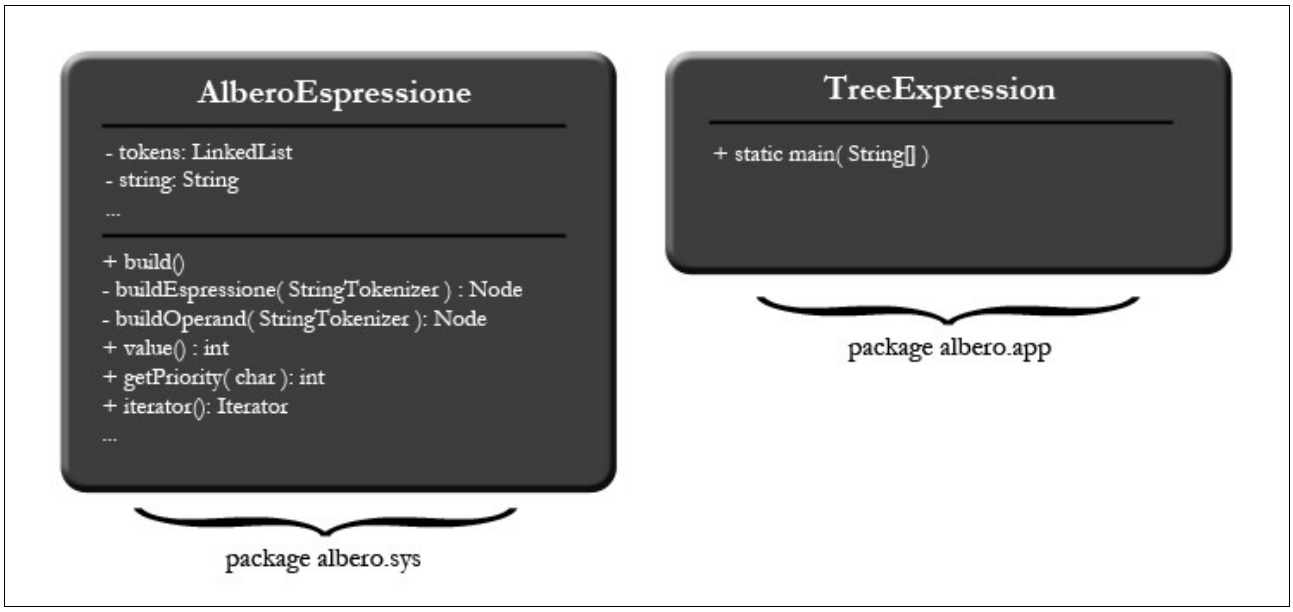


*FIG 2.1– Diagrammi di classe UML

I diagrammi in figura 2.0 riportano le classi utilizzate nel progetto insieme ai metodi e agli attributi. Per questioni di "pulizia" si è evitato di riportare l'intera lista dei metodi nei diagrammi, riportando solo quelli più importanti dal punto di vista dell'elaborazione.

2.3 Gerarchia di classe Node

Una prima scelta di progetto è quella di gestire i nodi dell'albero mediante una gerarchia di ereditarietà. Il motivo e che l'albero deve, di momento in momento, contenere diversi tipi di nodo (operando ed operatore) come suggerito nella traccia, quindi, si imposta una soluzione gerarchica facendo uso dell'ereditarietà. La figura 2.3 mostra la gerarchia delle classi Node.

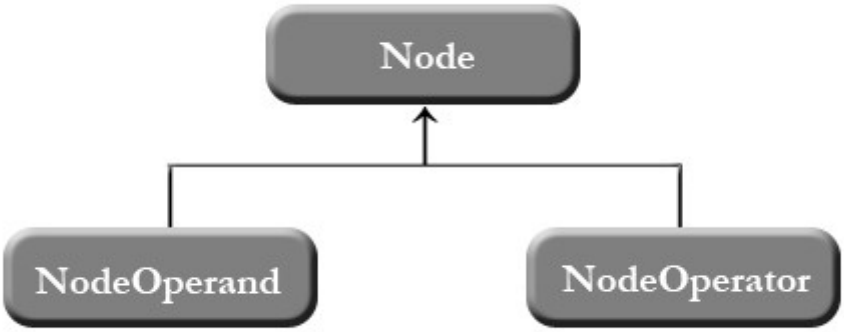


*FIG 2.2 – Gerarchia di classe Node

La relazione che lega le classi è di tipo "is-a" (è un) – relazione di ereditarietà fra Nodi. L'idea è che la struttura ad albero essendo una di struttura di tipo Node, potrà ospitare oggetti di tipo Node, NodeOperand e NodeOperator.

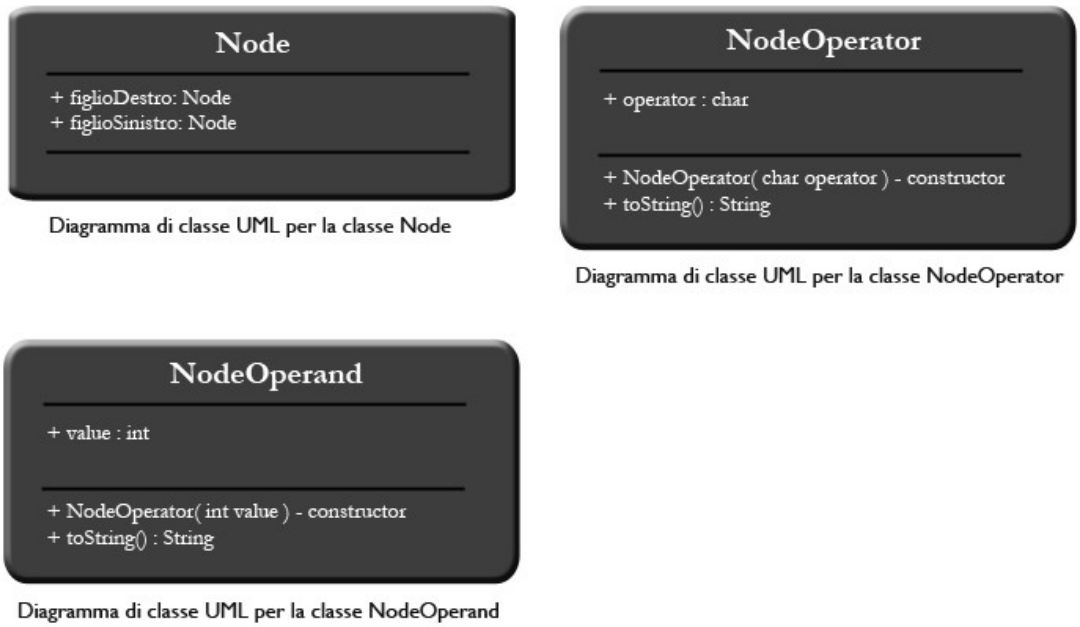


**FIG 2.3 – Diagramma delle classi UML, per le classi Node*

La gerarchia di classe Node è gestita, nel progetto mediante il concetto di Inner-class di Java. Invece di implementare i concetti di Nodo in classi esterne separate in file diversi, si è optato per una implementazione differente, a (inner-class) o classi interne (innestate). E' bene dire anzitutto, che la scelta di "battezzare" queste classi come *private* è una scelta guidata dall'ingegneria del software, in quanto i concetti di Nodo, utilizzati nel progetto, sono dei dettagli di implementazione che andrebbero nascosti per evitare delle "dipendenze" dall'esterno, perciò si fa uso dell'importante concetto della o.o.p e cioè dell'incapsulamento dei dati) e l'intera gerarchia, comprese le classi NodeOperand e NodeOperator, è *private*, inoltre si è fatto uso della parola chiave *static* per eliminare quel famoso collegamento (puntatore) tra inner-class e outer-class per semplici motivi di efficienza. La figura 2.3 Mostra i diagrammi di classe UML per le classi di tipo Node. La classe Node contiene solo due attributi è presenta una definizione per così dire "ricorsiva", in quanto i dati membro sono riferimenti dello stesso tipo della classe in cui sono contenuti è cioè Node. Le classi eredi NodeOperand e NodeOperator sono più specifiche; esse aggiungono dei dettagli tipo, il valore nel caso di un nodo operando ed il carattere operatore nel caso del nodo operatore appunto. Entrambe le classi sono munite di un costruttore con un solo parametro e del metodo toString indispensabile nel caso in cui si preferisce avere una "stampa a video" dei medesimi nodi.

I prossimi paragrafi mostrano il funzionamento dei metodi e delle classi adoperate, in più viene mostrato qualche estratto di codice sorgente e ne vengono discussi i dettagli implementativi. Per una visualizzazione completa del codice si può passare all'appendice o consultare il CD allegato.

3 La classe AlberoEspressione

Di seguito si mostra una parte della definizione della classe `AlberoEspressione`.

```
package albero.sys;
import java.util.*;

import javax.swing.*;

public class AlberoEspressione implements Iterable<String> {
   private LinkedList<String> tokens = new LinkedList<>();
   private String string;

   private static class Node {...} // Node

   private static class NodeOperator extends Node {...} // NodeOperator

   private static class NodeOperand extends Node {...} // NodeOperand

   // nodo radice dell'albero
   private Node root = null;

   private boolean flag = false;

   // ...
   // ..
   public Iterator<String> iterator() {
      return new TreeIterator<String>();
   }

   private class TreeIterator implements Iterator<String> {...}

} // AlberoEspressione
```

**FIG 3.1 – classe AlberoEspressione*

La classe `AlberoEspressione` è dichiarata all'interno del package `Albero.sys` questo package contiene solo la classe AlberoEspressione per motivi di ordine. La classe implementa l'interfaccia *Iterable<T>* per mettere a disposizione il meccanismo dell'iterazione e quindi ridefinisce (Overriding) il metodo iterator(). I campi private *tokens* e *string* rappresentano il primo una *LinkedLisk* di *String* utilizzata per gestire l'iterazione "a stampa" cfr 3.5, il secondo una stringa utilizzata per le visite "a stampa" sull'albero cfr 3.3. La classe dichiara inoltre una variabile di tipo Node, utilizzata come puntatore (radice) dell'albero e la variabile di stato, *boolean flag*, utilizzata per individuare l'apertura e la chiusura esatta delle parentesi tonde.

3.1 I metodi build() e buildEspressione()

Il processo di elaborazione viene iniziato dal metodo pubblico *build()*, che riceve una stringa espressione *"expression"* e crea un oggetto di classe *StringTokenizer* per frammentare l'espressione nei singoli operatori ed operandi.

```
public void build( String expression ) {

   StringTokenizer stk = new StringTokenizer( expression, "(+-*/%^)", true );
   root = buildEspressione( stk );
}
```

**FIG 3.2 – Il metodo build*

Lo *StringTokenizer* viene impostato in maniera tale da restituire sia i token (in questo caso i numeri interi che compongono l'espressione), sia i delimitatori "(+-*/%^)", che ne costituiscono i singoli operatori matematici, impostando il terzo parametro al valore *true*. Il metodo *build*, sub-appalta il compito ad un altro metodo interno dichiarato private *buildEspressione* il quale riceve lo *StringTokenizer* impostato sulla stringa espressione e restituisce a fine processo, la radice dell'albero costruito.

Analizziamo quindi il metodo più importante di tutto il progetto, il metodo *buildEspressione* che costruisce l'albero dell'intera espressione aritmetica.

```
private Node buildEspressione( StringTokenizer stk ) {
      // variabili locali al metodo
      Stack<Node> stackOperand = new Stack<>();
      Stack<Character> stackOperator = new Stack<>();
      char token;

      stackOperand.push( buildOperand( stk ) );

      while( stk.hasMoreTokens() ) {
            token = stk.nextToken().charAt(0);
            if( token == '(')
                  throw new RuntimeException("Espressione malformata");

            if( token == ')' && flag ) {
                  while( !stackOperator.isEmpty() ) {
                        NodeOperator np = new NodeOperator( stackOperator.pop() );
                        np.figlioSinistro  = stackOperand.pop();
                        np.figlioDestro = stackOperand.pop();
                        stackOperand.push(np);
                  }
                  return stackOperand.pop();
            }

            while( !stackOperator.isEmpty() && getPriority(stackOperator.peek()) >=
                  getPriority(token) ) {
                  NodeOperator np = new NodeOperator( stackOperator.pop() );
                  np.figlioSinistro = stackOperand.pop();
                  np.figlioDestro = stackOperand.pop();
                  stackOperand.push(np);
            }
```

```
	stackOperand.push( buildOperand( stk ) );
		stackOperator.push(token);
	}

	while( !stackOperator.isEmpty() ) {
		NodeOperator np = new NodeOperator( stackOperator.pop() );
		np.figlioSinistro = stackOperand.pop();
		np.figlioDestro = stackOperand.pop();
		stackOperand.push(np);
	}

	return stackOperand.pop();
}
```

FIG 3.2 – Il metodo buildEspressione

Il corpo del metodo dichiara due variabili locali di tipo *Stack*, il primo uno stack di nodi operandi, il secondo uno stack di caratteri operatori, la variabile di tipo *char* token è utilizzata per contenere il prossimo token durante la computazione. La prima istruzione, aggiunge in cima allo stack degli operandi il primo operando dell'espressione che viene restituito dal metodo *buildOperand*. (Ci si aspetta che in una espressione "ben formata" il primo carattere è "(" o un valore numerico).

3.1.1 Il metodo buildOperand()

Il metodo *buildOperand* costruisce un nuovo nodo di tipo (*NodeOperand*). Ad ogni chiamata, la stringa op contiene il prossimo token, dalla quale successivamente se ne estrae il primo carattere mediante il metodo *charAt(int)*. Il metodo in questione, invoca ricorsivamente il metodo *buildEspressione* nel caso in cui si incontra una parentesi tonda ")" chiusa. Se quindi si entra nell'if a causa di una parentesi di chiusura, la variabile flag è impostata a true in modo tale da intercettare eventuali parentesi non chiuse dopo una parentesi aperta.

```
private Node buildOperand( StringTokenizer stk ) {
	String op = stk.nextToken();

	if(op.charAt(0) == '(') {
		flag = true;
		return buildEspressione( stk );
	}

	NodeOperand nodeOperand = new NodeOperand( Integer.parseInt(op) );
	return nodeOperand;
}
```

FIG 3.3 – Il metodo buildEspressione

In caso contrario il nodo viene costruito e restituito al chiamante. I campi figlioDestro e figlioSinistro di NodeOperand, sono impostati automaticamente a null dalle inizializzazioni di default, il parametro al costruttore di NodeOperand, rappresenta il contenuto "operando" in veste intera, che verrà incapsulato nel nodo.
Proseguendo quindi con l'elaborazione nel metodo buildEspressione, il primo ciclo while, quello più esterno, viene eseguito fintanto che ci sono token nell'espressione. Se non vi sono più

token nella stringa dell'espressione, il ciclo non viene eseguito e si passa quindi al ciclo di iterazione successivo, utilizzato in ogni caso per "svuotare lo stack degli operatori e quindi per costruire l'albero, che in questo caso degenera in un solo nodo, contenente un unico operando numerico. Nel caso in cui si entra nel ciclo di while più esterno(e questo è il caso in cui l'espressione non degenera in un singolo operando) , la prima istruzione interna al ciclo memorizza il prossimo token che può essere sia un operatore, sia un operando. Una parentesi tonta aperta costituirebbe un caso di espressione "malformata", e quindi l'istruzione if successiva lancia un'eccezione nel caso ciò avvenga. Se il prossimo carattere, rappresenta una parentesi tonda chiusa e se la variabile flag vale true (cio vuol dire che precedentemente è già stata aperta una parentesi tonda e stiamo eventualmente elaborando ricorsivamente una espressione in parentesi "annidate"), si svuota nuovamente lo stack degli operatori, si costruisce l'albero dell'espressione e si impila in cima allo stack operandi. Nel caso in cui il prossimo token non rappresenta una parentesi tonda chiusa(non dobbiamo quindi ritornare un processo ricorsivo), fintanto che lo stack degli operatori non è vuoto e la priorità dell'operatore affiorante sullo stack è minore di quella dell'operatore corrente, si costruisce un nuovo nodo operatore con il campo figlioDestro uguale alla cima dello stack operatori(top) e figlioSinistro uguale alla cima dello stack operatori (top-1). Ad ogni iterazione viene aggiunto alla cima dello stack degli operatori il prossimo operatore ed alla cima dello stack operandi il prossimo operando. Il processo termina quando non vi sono più token da elaborare.

3.2 Il metodo getPriority()

Il compito della gestione delle priorità degli operatori è affidato ad un metodo private(quindi inaccessibile dall'esterno) getPriority il cui codice è riportato in figura 3.4.

```
private int getPriority( char op ) {
      switch( op ) {
         case '+':
         case '-':
            return 1;
         case '*':
         case '/':
         case '%':
            return 2;
         case '^':
            return 3;
         default:
            return -1;
       }
}
```

**FIG 3.4 – Il metodo getPriority*

Il metodo fa uso dell' istruzione condizionale multipla – *switch* per gestire le diverse tipologie di operatori cui associa un valore numerico di tipo *int*. La convenzione adottata è la seguente: - ad un valore numerico maggiore corrisponde una maggiore priorità, perciò agli operatori additivi "+" e "-" vi si associa un 1 , agli operatori moltiplicativi e di resto(euclideo) "*", "/" e "%", vi si associa un 2, per l'operatore potenza "^" un 3. Il caso di per così dire di "fondo" in gergo di *default* restituisce un -1.

3.3 Le visite sull'albero

Una volta costruito, l'albero di una espressione aritmetica, permette di svolgere qualsiasi operazione sull'espressione. Una delle più comuni è più logiche è sicuramente la sua valutazione, ovvero il calcolo effettivo – il valore dell'espressione - Cfr 3.4. E' interessante però anche un'altra operazione sugli alberi, ed in particolare su un albero di un'espressione aritmetica è cioè la visita. Esistono tre tipi di visite in un albero binario e nel nostro caso in una albero di un'espressione(con operatori binari). La visita simmetrica o (in-order) , la visita anticipata o (pre-order) e la visita posticipata (post-order). Di seguito si presentano i metodi di visita dell'applicazione (il solo codice della visita simmetrica) per gli altri metodi, nell'appendice e/o nel CD, è riportato l'intero codice di tutto il progetto.

3.3.1 la visita simmetrica

Nella visita simmetrica la regola consiste nel visitare prima il sotto-albero sinistro, poi la radice quindi il sotto-albero destro (ricorsivamente). In un albero di una espressione la visita simmetrica riproduce nuovamente l'espressione di partenza. Il metodo *public inOrder()* come è consuetudine, delega il compito ad un altro metodo private in veste però ricorsiva, il compito di ottenere una visita simmetrica dell'albero appena computato.

```
private void inOrder( Node root ) {
      if( root != null ) {
            if( root instanceof NodeOperator ) {
                  string += "(";
             }
            inOrder( root.figlioDestro );
            tokens.add(root.toString());
            string += root.toString();
            inOrder( root.figlioSinistro);
            if( root instanceof NodeOperator ) {
            string += ")";
            }
      }
}
```

**FIG 3.5 – Il metodo private inOrder()*

Il metodo riceve un parametro di tipo *Node root*, inteso come radice dell'albero. La semantica del metodo consiste nell'effettuare ricorsivamente una visita a sinistra re-invocando lo stesso metodo e passandogli ora il parametro “figlioSinistro” poi si visita la radice, concatenando il valore ad una stringa ed aggiungendo lo stesso valore ad una lista(per le stampe a video); infine si effettua una visita a destra. Per ragioni di estetica e soprattutto di ordine e di precedenza nell'espressione, si è preferito inserire ad ogni chiamata del metodo una coppia di parentesi tonde “(“ e “)”. La stringa è utilizzata nel metodo toString, la lista invece è utilizzata nel costrutto di iterazione.

3.3.2 la visita anticipata

La visita anticipata procede allo stesso modo tranne ovviamente nelle chiamate ricorsive, questa

volta viene prima letto il valore della radice e poi si invoca ricorsivamente lo stesso metodo con il sotto-albero sinistro e poi con il sotto-albero destro. Il pregio di una visita anticipata è che è possibile ottenere l'espressione in veste di notazione polacca (e nel caso della visita posticipata, polacca inversa RPN) e questo permette di eliminare le parentesi per l'ordine delle precedenze.

3.3.3 la visita posticipata

per la visita posticipata si invoca il metodo con “figlioSinistro” come parametro, poi “figlioDestro” ed infine si visita il valore del nodo radice.

3.4 La valutazione dell'espressione

Siamo giunti ora alla valutazione dell'espressione aritmetica. Il compito è affidato ad un metodo *value()* il cui scopo è quello di scorrere l'albero, intercettare gli operatori, gli operandi, risiedenti – sempre sulle foglie e quindi effettuare i calcoli rispettando le precedenze, imposte eventualmente dalle parentesi tonde. Il codice seguente definisce il metodo *value()*.

```
private int value( Node root ) {
	if ( root == null )
		throw new RuntimeException("error");
	if( root instanceof NodeOperand )
		return ((NodeOperand)root).value;
	else {
		int a = value( root.figlioSinistro );
		int b = value( root.figlioDestro );

		switch( ((NodeOperator)root).operator ) {
		case '+':
			return a+b;
		case '-':
			return b-a;
		case '*':
			return a*b;
		case '/':
			return a/b;
		case '%':
			return a%b;
		case '^':
			return (int)Math.pow(b, a);
			default:
				throw new RuntimeException("error");
		}
	}
}
```

**FIG 3.6 – Il metodo value()*

Il risultato del metodo di tipo *int* rappresenta il valore dell'espressione.

3.5 Il costrutto di iterazione

La gestione dell'iterazione si basa su una struttura dati di tipo *LinkedList*, che contiene, a seconda che, sia stata eseguita una visita simmetrica, anticipata o posticipata, la successione degli elementi da iterare; questo perché ad ogni chiamata di un metodo di visita alla lista, chiamata nel codice con l'identificatore “tokens”, vi si aggiungono progressivamente tutti i nodi dell'albero. Il risultato è che ogni qualvolta si itera un AlberoEspressione, in realtà l'iterazione si effettua su una lista che contiene la successione dei nodi.

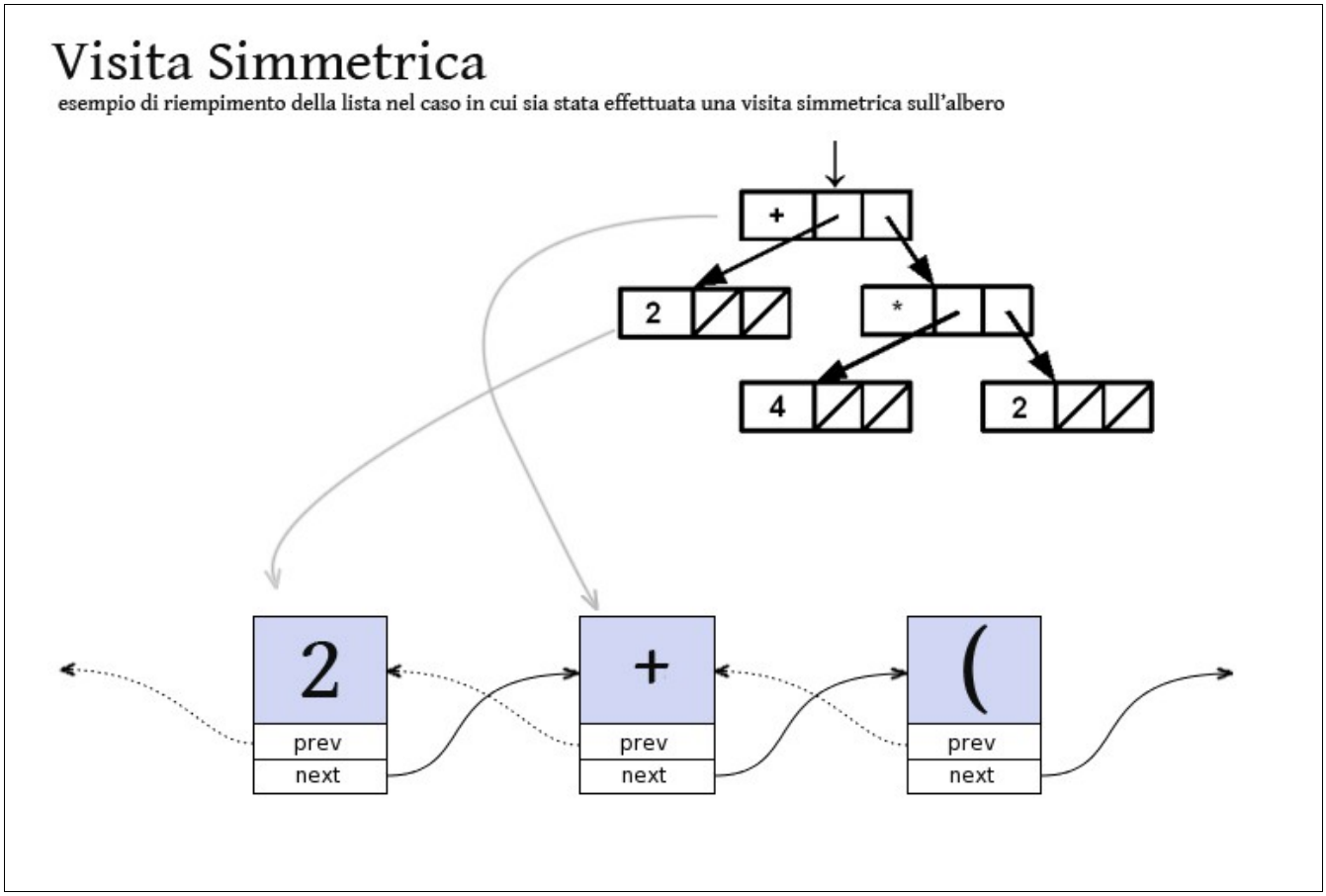


**FIG 3.7 – esempio di creazione della lista per l'iterazione (in-Order)*

4 Design GUI

4.1 Il progetto dell'interfaccia grafica

Il progetto dell'interfaccia grafica consiste essenzialmente, nel realizzare gli elementi, o meglio l'elemento per così dire “terminale” - (front-end), una sorta di apparato “virtuale”, con il quale l'utente interagisce per effettuare le operazioni più svariate con il programma, in modo amichevole e semplificato, attraverso finestre, menus, pulsanti ecc. Nel progetto si fa uso del framework di Java “Swing”, che mette a disposizione centinaia di classi per gestire la grafica nelle applicazioni. La figura 4.1 mostra l'idea grafica adottata nel progetto.

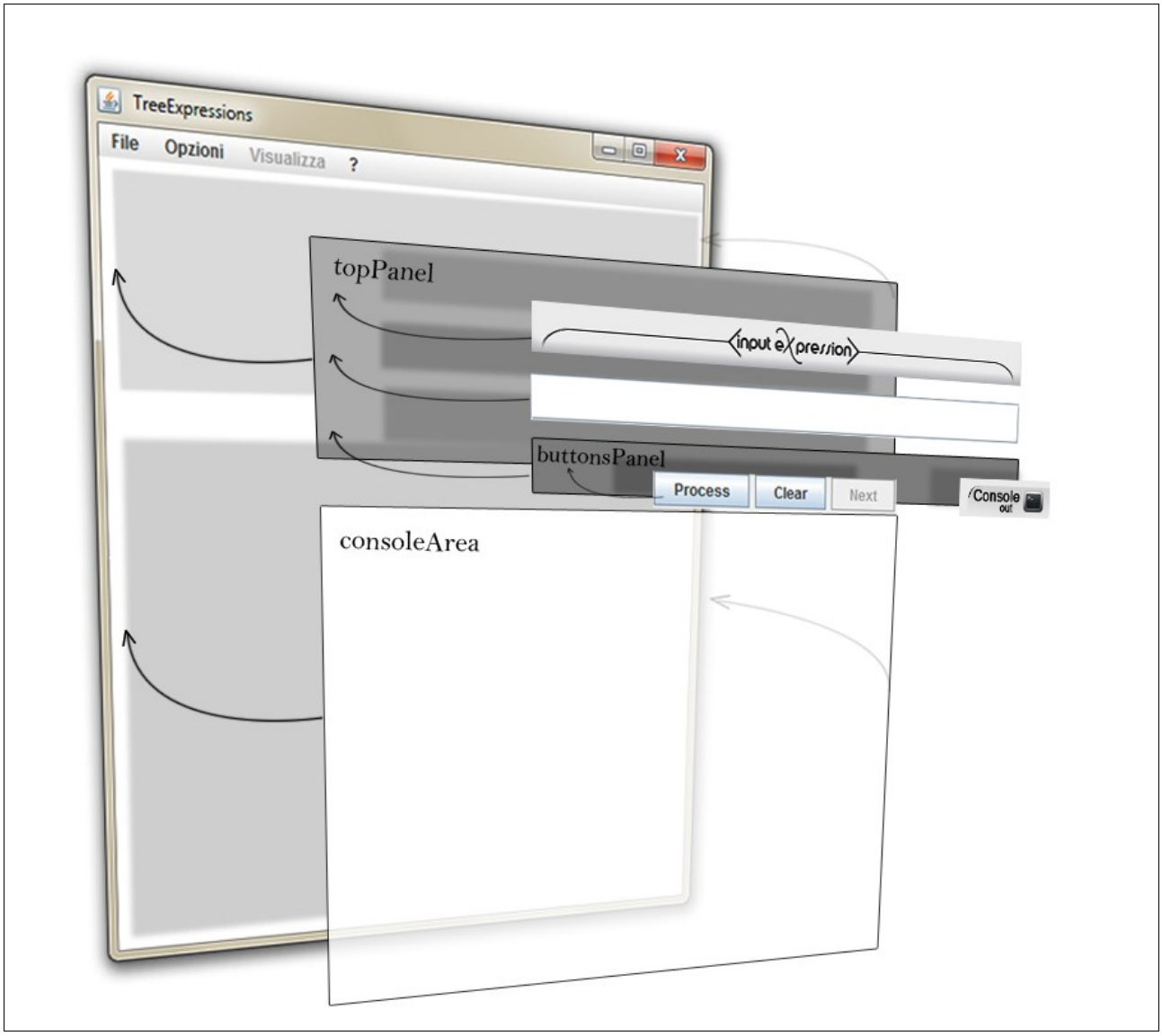


FIG 4.1 – GUI del progetto

La classe per la grafica *“WinMain”* si trova nel *package “albero.graphic”*. Vengono utilizzati due pannelli. Il primo chiamato *buttonsPanel* dove sono situati(secondo la gestione *FlowLayout*) i pulsanti principali; il secondo pannello prevede una gestione di tipo *GridLayout*, esso è il

"topPanel" nel quale vi sono inseriti i componenti *"Jlabel"* (con l'immagine), *"JTextField"* ed il *buttonsPanel* con i pulsanti già agganciati. Infine il primo pannello *(topPanel)* è aggiunto al *JFrame* principale a *"NORTH"* mentre il componente *"JTextArea"* è aggiunto al *JFrame* a *"SOUTH"*.

4.2 Il menu principale

Per quanto riguarda la scelta dei menu non c'è molto da dire. L'interfaccia è dotata di tre menu. Il primo è il menu "File", con le informazioni principali ed il tasto di uscita("Quit"). Il secondo menu è il menu "Visualizza" che consente di effettuare le operazioni principali sull'albero quali le visite simmetriche, pre, post ecc, la valutazione dell'espressione(nel menu la voce è "valore") e la gestione dell'iterazione, mediante un sotto-menu "Iterazione". Infine sulla destra non può mancare il menu "?" dove vi sono le informazioni (about), autore, guida.

*il presente progetto è stato realizzato nel mese di giugno 2013

Appendice

Codice Sorgente Java

Di seguito sono riportati i listati del codice di tutto il progetto java “Albero Espressione Aritmetica” - per una eventuale consultazione; tutto il progetto, esportato sotto forma di “filexy.jar” si trova nel CD allegato insieme alla cartella delle immagini ed alla medesima relazione in formato pdf.

```
package albero.app;

import albero.graphic.*;

public class TreeExpressions {
        public static void main( String[] args )
        {
                WinMain winMain = new WinMain();
                winMain.setVisible( true );
        }
}
```

```
package albero.sys;
import java.util.*;

import javax.swing.*;

public class AlberoEspressione implements Iterable<String>{
        private LinkedList<String> tokens = new LinkedList<>();
        private String string;

        // la classe Node modella il concetto di Nodo
        @SuppressWarnings("unused")
        private static class Node {
                Node figlioDestro;
                Node figlioSinistro;
        } // Node

        // la classe NodeOperator esetende il concetto di nodo
        // specificandone la tipologia in Nodo operatore
        @SuppressWarnings("unused")
        private static class NodeOperator extends Node {
                public char operator;

                public NodeOperator( char operator ) {
                        this.operator = operator;
                }

                public String toString() {
                        return "" + operator;
                }

                public boolean equals( Object o ) {
                        if( !(o instanceof NodeOperator ) ) return false;
                        if( o == this ) return true;

                        NodeOperator n = (NodeOperator)o;
                        return operator == n.operator;
                }

        } // NodeOperator
```

```
// la classe NodeOperator esetende il concetto di nodo
// specificandone la tipologia in Nodo operando
@SuppressWarnings("unused")
private static class NodeOperand extends Node
{
        public int value;

        public NodeOperand( int value ) {
                this.value = value;
        }

        public String toString() {
                return "" + value;
        }

        public boolean equals( Object o ) {
                if( !(o instanceof NodeOperand ) ) return false;
                if( o == this ) return true;

                NodeOperand n = (NodeOperand)o;
                return value == n.value;
        }
} // NodeOperand

// nodo radice dell'albero
private Node root = null;

// variabile di stato
// utilizzata per intercettare eventuali coppie di parentesi tonde
// aperte ma non chiuse
private boolean flag = false;

/*
 * il metodo build inizia il processo di elaborazione
 * della stringa che rappresenta l'espressione
 * aritmetica
 */
public void build( String expression ) {
        StringTokenizer stk = new StringTokenizer( expression, "(+-*/%^)", true );
        root = buildEspressione( stk );
}

// metodo buildEspressione
private Node buildEspressione( StringTokenizer stk ) {

        // variabili locali al metodo
        Stack<Node> stackOperand = new Stack<>();
        Stack<Character> stackOperator = new Stack<>();
        char token;

        // si presume che all'inizio ci sia un operando nell'espressione
        // tale operando viene inserito sulla cima dello stack operandi (top)
        stackOperand.push( buildOperand( stk ) );


        while( stk.hasMoreTokens() ) {
                // prossimo carattere dell'espressione
                token = stk.nextToken().charAt(0);

                // parentesi aperta improbabile a questo punto
                // ci si aspetta un operatore
                if( token == '(')
                        throw new RuntimeException("Espressione malformata");

                // se la parentesi e' chiusa si ritorna eventualmente il sotto-albero
                // che puo' corrispondere eventualmente all'albero generico
                if( token == ')' && flag )
                {
                        /* fintantoche' lo stack degli operatori
                         * non è vuoto costruisci un nuovo nodo contenente
                         * come figlio sinistro e figlio destro duo sottoalberi
                         * contentenenti top e top-1 dello stack operandi
                         */
                        while( !stackOperator.isEmpty() ) {
                                NodeOperator np = new NodeOperator( stackOperator.pop() );
                                np.figlioSinistro  = stackOperand.pop();
                                np.figlioDestro = stackOperand.pop();
                                stackOperand.push(np);
                        }
```

```
                                // ritorna al chiamante il sottoalbero appena creato
                                return stackOperand.pop();
                        }

                        while( !stackOperator.isEmpty() && getPriority(stackOperator.peek()) >=
getPriority(token) ) {
                                NodeOperator np = new NodeOperator( stackOperator.pop() );
                                np.figlioSinistro = stackOperand.pop();
                                np.figlioDestro = stackOperand.pop();
                                stackOperand.push(np);
                        }

                        // se lo stack degli operatori e' vuoto
                        // o la priorita dell'operatore affiorante sulolo stack operatori
                        // e' < della priorita dell'operatore corrente
                        stackOperand.push( buildOperand( stk ) );

                        stackOperator.push(token);

                }

                while( !stackOperator.isEmpty() ) {
                        NodeOperator np = new NodeOperator( stackOperator.pop() );
                        np.figlioSinistro = stackOperand.pop();
                        np.figlioDestro = stackOperand.pop();
                        stackOperand.push(np);
                }

                return stackOperand.pop();
        }

        // metodo buildOperand
        private Node buildOperand( StringTokenizer stk ) {
                // memorizza in op il prossimo operando
                String op = stk.nextToken();

                // se il prossimo carattere corrisponde a '('
                // si invoca nuovamente il metodo builEspressione
                if(op.charAt(0) == '(') {
                        flag = true;
                        return buildEspressione( stk );
                }

                /*
                 * se si è arrivati sin qui vuol dire che il carattere
                 * è un operando valido percio,
                 * si costruisce un nodo con solo l'operando ed
                 * il reiferimento a tale nodo viene "ritornato" al chiamante
                 */
                NodeOperand nodeOperand = new NodeOperand( Integer.parseInt(op) );
                return nodeOperand;

        }

        // visita simmetrica
        public String inOrder() {
                string = "";
                tokens.clear();
                inOrder( root );
                return string;
        }

        private void inOrder( Node root ) {
                if( root != null ) {
                        if( root instanceof NodeOperator ) {
                                string += "(";
                        }
                        inOrder( root.figlioDestro );
                        tokens.add(root.toString());
                        string += root.toString();
                        inOrder( root.figlioSinistro);
                        if( root instanceof NodeOperator ) {
                                string += ")";
                        }
                }
        }
```

```
// visita anticipata
public String preOrder() {
        tokens.clear();
        string = "";
        preOrder( root );
        return string;
}

private void preOrder( Node root ) {
        if( root != null ) {
                string += root.toString();
                tokens.add(root.toString());
                preOrder( root.figlioSinistro );
                preOrder( root.figlioDestro );
        }
}

// visita posticipata
public String postOrder() {
        tokens.clear();
        string = "";
        postOrder( root );
        return string;
}

private void postOrder( Node root ) {
        if( root != null ) {
                preOrder( root.figlioSinistro );
                preOrder( root.figlioDestro );
                string += root.toString();
                tokens.add(root.toString());
        }
}

// metodo getPriority per la gestione delle priorità degli operatori
private int getPriority( char op ) {
        switch( op ) {
        case '+':
        case '-':
                return 1;
        case '*':
        case '/':
        case '%':
                return 2;
        case '^':
                return 3;
                default:
                        return -1;
        }
}

// il metodo ritorna una lista di tipo String
// che contiene una eventuale visita sull'albero
public LinkedList<String> getList() {
        return tokens;
}

// metodo di valutazione pubblico
public String value()
{
        return "" + value( root );
}

// metodo privato per la valutazione dell'espressione
private int value( Node root ) {
        if ( root == null )
                throw new RuntimeException("error");
        if( root instanceof NodeOperand )
                return ((NodeOperand)root).value;
        else {
                int a = value( root.figlioSinistro );
                int b = value( root.figlioDestro );

                switch( ((NodeOperator)root).operator ) {
                case '+':
                        return a+b;
                case '-':
                        return b-a;
                case '*':
```

```
                        return a*b;
                case '/':
                        return a/b;
                case '%':
                        return a%b;
                case '^':
                        return (int)Math.pow(b, a);
                        default:
                                throw new RuntimeException("error");
                }
        }
    }

    // costrutto di iterazione
    // l'iterazione è affidata ad una lista che contiene la vistita dell'albero
    public Iterator iterator() {
        return new TreeIterator();
    }

    private class TreeIterator implements Iterator<String> {
        Iterator<String> it = tokens.iterator();

        public boolean hasNext() {
                return it.hasNext();
        }

        public String next() {
                if( !hasNext() ) throw new NoSuchElementException();
                        return it.next();
        }

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

    // metodo toString
    public String toString() {
        return inOrder() + " = " + value();
    }

    // hashCode
    public int hashCode() {
        final int  MOLT = 163;
        return inOrder().hashCode() * MOLT;
    }

    // equals
    public boolean equals(Object o) {

        if (!(o instanceof AlberoEspressione)) return false;
        if ( o == this ) return true;

        AlberoEspressione nb = (AlberoEspressione)o;

        String a = inOrder();
        String b = nb.inOrder();

        return a.equals(b);
    }

    public static void main( String [] args )
    {
        AlberoEspressione a = new AlberoEspressione();

        a.build("2");
        System.out.println(a.inOrder());

    }
}
```

```
package albero.graphic;

import javax.swing.*;
import javax.swing.event.*;
```

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import albero.sys.*;

public class WinMain extends JFrame implements ActionListener{
        // elementi menu
        private JMenuBar menuBar;
                private JMenu fileMenu;
                        private JMenuItem infoItem;
                        private JMenuItem exitItem;

                private JMenu optionsMenu;
                        private JMenuItem fontItem;

                private JMenu viewMenu;
                        private JMenuItem inItem;
                        private JMenuItem preItem;
                        private JMenuItem postItem;
                        private JMenuItem valueItem;
                        private JMenuItem stackItem;
                        private JMenuItem treeItem;

                private JMenu iterMenu;
                        private JMenuItem preIterItem;
                        private JMenuItem postIterItem;
                        private JMenuItem orderIterItem;

                private JMenu aboutMenu;
                        private JMenuItem authorItem;
                        private JMenuItem helpItem;
                        private JMenuItem javaItem;

        // elementi area input
        private JPanel topPanel;
                private JLabel imageInLabel;
                private JTextField inputField;

        // elementi area pulsanti
        private JPanel buttonsPanel;
                private JButton processButton;
                private JButton clearButton;
                private JButton stackButton;
                private JButton treeButton;
                private JLabel imageOutLabel;
                private JLabel consoleImage;
                private JButton nextButton;

        // pannello per grafica dell'albero
        private TreeDialog tr;

        JTextArea consoleArea;
        AlberoEspressione aeX;
        Iterator<String>  it;

        public WinMain()
        {
                setTitle(" TreeExpressions" );
                setLocation( 200, 150 );
                setSize( 500, 600 );
                setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

                topPanel = new JPanel();
                topPanel.setLayout( new GridLayout(3, 1) );
                        inputField = new JTextField();
                        inputField.setToolTipText("Inserire un'espressione matematica");
                        inputField.setFont( new Font("Arial", Font.BOLD, 20) );
                        imageInLabel = new JLabel( new ImageIcon("images\\inputImage.png") );
                    topPanel.add( imageInLabel );
                        topPanel.add(inputField);

                buttonsPanel = new JPanel();
                        processButton = new JButton("Process");
                        processButton.setToolTipText("Costruisce l'albero dell'espressione");
                        processButton.addActionListener( this );
                        buttonsPanel.add(processButton);
                        clearButton = new JButton("Clear");
                        clearButton.setToolTipText("Svuota l'albero");
                        clearButton.addActionListener( this );
                        buttonsPanel.add(clearButton);
```

```
                nextButton = new JButton("Next");
                nextButton.setToolTipText("prossimo elemento");
                nextButton.addActionListener( this );
                nextButton.setEnabled(false);
                buttonsPanel.add(nextButton);
                treeButton = new JButton("Tree View");
                treeButton.setToolTipText("Visualizza l'albero");
                treeButton.addActionListener( this );
                treeButton.setEnabled( false );
                buttonsPanel.add(treeButton);
                consoleImage = new JLabel( new ImageIcon("images\\consoleImage.png") );
                buttonsPanel.add(consoleImage);


        topPanel.add(buttonsPanel);
        consoleArea = new JTextArea();
        consoleArea.setFont( new Font("Tahoma", Font.BOLD, 25));
        consoleArea.setToolTipText("Console Output");
        consoleArea.setText("\n ");
        consoleArea.setEditable(false);

        add( topPanel, BorderLayout.NORTH );
        add( consoleArea, BorderLayout.CENTER );

        menuBar = new JMenuBar();
        this.setJMenuBar(menuBar);

        initActions();
        initMenus();

        viewMenu.setEnabled( false );

}

private void initActions()
{
        infoItem = new JMenuItem("Informazioni");
        infoItem.addActionListener( this );
        exitItem = new JMenuItem("Quit");
        exitItem.addActionListener( this );

        fontItem = new JMenuItem("Font");

        inItem = new JMenuItem("espressione ");
        inItem.addActionListener( this );
        preItem = new JMenuItem("prefissa");
        preItem.addActionListener( this );
        postItem = new JMenuItem("infissa");
        postItem.addActionListener( this );
        valueItem = new JMenuItem("Valore");
        valueItem.addActionListener( this );
        stackItem = new JMenuItem("Stack Grafico");
        treeItem = new JMenuItem("Albero Grafico");
        treeItem.addActionListener( this );

        orderIterItem = new JMenuItem("In-order");
        orderIterItem.addActionListener( this );
        preIterItem = new JMenuItem("Pre-order");
        preIterItem.addActionListener( this );
        postIterItem = new JMenuItem("Post-order");
        postIterItem.addActionListener( this );

        authorItem = new JMenuItem("Autore (C)");
        javaItem = new JMenuItem("Java (C)");
        helpItem = new JMenuItem("Guida");

}

private void initMenus()
{
        fileMenu = new JMenu(" File ");
                fileMenu.add(infoItem);
                fileMenu.add(exitItem);
        menuBar.add(fileMenu);

        optionsMenu = new JMenu(" Opzioni ");
                optionsMenu.add(fontItem);
        menuBar.add(optionsMenu);
```

```
                viewMenu = new JMenu(" Visualizza ");
                        viewMenu.add(inItem);
                        viewMenu.add(preItem);
                        viewMenu.add(postItem);
                        viewMenu.addSeparator();
                        viewMenu.add(valueItem);
                        viewMenu.addSeparator();
                        //viewMenu.add(stackItem);
                        //viewMenu.add(treeItem);
                menuBar.add(viewMenu);

                iterMenu = new JMenu("Iterazione");
                        iterMenu.add(orderIterItem);
                        iterMenu.add(preIterItem);
                        iterMenu.add(postIterItem);
                        viewMenu.add(iterMenu);

                aboutMenu = new JMenu(" ? ");
                        aboutMenu.add(authorItem);
                        aboutMenu.add(javaItem);
                        aboutMenu.add(helpItem);
                menuBar.add(aboutMenu);
        }

        private void falseInitActions()
        {
                stackItem.setEnabled(false);
                treeItem.setEnabled(false);
                //processButton.setEnabled(false);
                inItem.setEnabled( false );
                preItem.setEnabled( false );
                postItem.setEnabled( false );
                valueItem.setEnabled( false );
        }

        public void actionPerformed( ActionEvent ae )
        {
                if( ae.getSource() == treeButton || ae.getSource() == treeItem ) {
                        tr = new TreeDialog(aeX.getList());
                        tr.setVisible(true);
                }
                else if( ae.getSource() == processButton ) {
                        aeX = new AlberoEspressione();
                        String in = inputField.getText();

                        try {
                                aeX.build( in );
                                consoleArea.setText( " " + in + " = " + aeX.value() + "\n " );
                        } catch ( Exception e ){
                                String er = "Errore! Inserire correttamente l'espressione \n" +
                                        "non sono consentiti caratteri speciali eccetto cifre e + - * / % ^";
                                JOptionPane.showMessageDialog(this, er, "Espressione malformata o
inesistente", JOptionPane.ERROR_MESSAGE);
                        }

                        viewMenu.setEnabled( true );
                        treeButton.setEnabled( true );

                }
                else if( ae.getSource() == inItem ) {
                        consoleArea.append( " # " + aeX.inOrder() + "\n " );
                }
                else if( ae.getSource() == preItem ) {
                        consoleArea.append( " # " + aeX.preOrder() + "\n " );
                }
                else if( ae.getSource() == postItem ) {
                        aeX.postOrder();
                        consoleArea.append( " # " + aeX.postOrder() + "\n " );
                }
                else if( ae.getSource() == valueItem ) {
                        try {
                        consoleArea.append( " = " + aeX.value() +"\n " );
                        }
                        catch ( Exception e ) {
                                String er = "Errore! Inserire correttamente l'espressione \n" +
                                "non sono consentiti caratteri speciali eccetto cifre e + - * / % ^";
                        JOptionPane.showMessageDialog(this, er, "Espressione malformata o inesistente",
JOptionPane.ERROR_MESSAGE);
                        }
                }
```

```
                else if( ae.getSource() == exitItem ) {
                        System.exit(0);
                }
                else if( ae.getSource() == infoItem ) {
                        JOptionPane.showMessageDialog(this, "Albero di un'espressione aritmetica\nby Giuseppe
Sottile 142880", "TreeExpression", JOptionPane.INFORMATION_MESSAGE);
                }
                else if( ae.getSource() == clearButton ) {
                        consoleArea.setText("");
                }
                else if( ae.getSource() == orderIterItem ) {
                        consoleArea.setText("");
                        aeX.inOrder();
                        it = aeX.iterator();
                        nextButton.setEnabled( true );
                }
                else if( ae.getSource() == preIterItem ) {
                        consoleArea.setText("");
                        aeX.preOrder();
                        it = aeX.iterator();
                        nextButton.setEnabled( true );
                }
                else if( ae.getSource() == postIterItem ) {
                        consoleArea.setText("");
                        aeX.postOrder();
                        it = aeX.iterator();
                        nextButton.setEnabled( true );
                }
                else if( ae.getSource() == nextButton ) {
                        try {
                                consoleArea.append(" -> [ " + it.next() + " ]");
                        } catch( Exception e ) {
                                consoleArea.append("\n");
                                nextButton.setEnabled(false);
                                JOptionPane.showMessageDialog(this, "NoSuchElementException");
                        }
                }
        }
}
```