



Alma Mater Studiorum-Università di Bologna
Scuola di Ingegneria

ESERCITAZIONE AUTONOMA

Espressioni, operandi, operatori in Java

Corso di Laurea in Ingegneria Informatica
Anno accademico 2021/2022

Prof. ENRICO DENTI

Dipartimento di Informatica – Scienza e Ingegneria (DISI)



VALUTARE ESPRESSIONI

- Abbiamo già visto come valutare, tramite uno stack, una semplice espressione aritmetica di interi
 - quell'implementazione era cablata nel codice → switch
 - oltre tutto, era specifica per gli interi
- Se volessimo generalizzare...?
- Per fare le cose «fatte bene», bisogna, come sempre, partire dall'analisi del problema
- Entità in gioco: un'espressione è composta di:
 - **operandi** di un certo *tipo*
 - **operatori** che agiscono su tali operandi



ESPRESSIONI «QUALSIASI»

- Operandi

- per essere il più generale possibile, *interfaccia*
- già ma.. quali caratteristiche ha un «qualsiasi» operando?

- Operatori

- anche qui si potrebbe scegliere una interfaccia, ma..
- ...gli operatori definiscono le operazioni, che di solito sono *note* e in *quantità prestabilita* → utile anche un *enumerativo*
- per semplicità, consideriamo solo operatori *binari*

```
interface Operand {  
    ...  
}
```

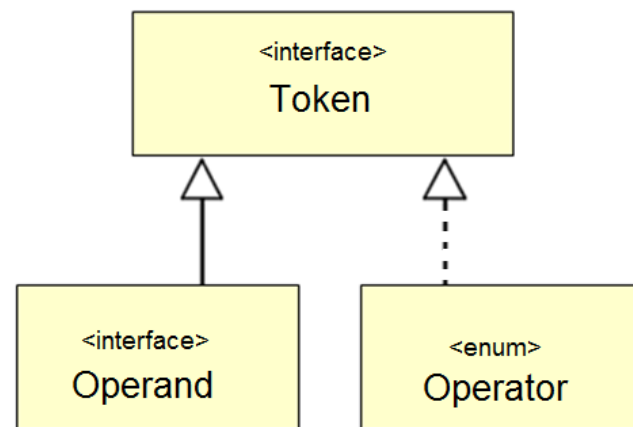
Java

```
enum Operator {  
    ...  
}
```

Java

OPERANDI & OPERATORI

- Ma per poter rappresentare l'espressione come albero, serve un tipo uniforme per il tipo T del nodo
 - ergo, operandi e operatori devono avere un *sovra-tipo comune*
 - MA uno è un'interfaccia, l'altro un enumerativo...!
- **IDEA: introdurre un'interfaccia-base **Token****
 - l'interfaccia **Operand** la *estenderà*
 - l'enumerativo **Operator** la *implementerà*
- Alcune domande
 - quali metodi comuni per **Token**?
 - quali metodi ulteriori per **Operand**?
 - quale organizzazione per **Operator**?





OPERANDI & OPERATORI

- Possibili risposte
 - un **Token** può esprimersi come stringa
 - un **Operand** è caratterizzato anche da un valore (intero e/o reale) e da una serie di possibili operazioni (è un campo!)
 - **Operator** dichiara gli operatori possibili

```
interface Token {  
    String getValueAsString();  
}
```

Java

```
interface Operand extends Token {  
    int getValueAsInteger();  
    double getValueAsReal();  
    Operand sum(Operand arg);  
    ...  
}
```

Java

```
enum Operator implements Token {  
    PLUS("+"), MINUS("-"),  
    TIMES("x"), DIVBY(":");  
    ...  
}
```

Java



OPERANDI & OPERATORI

```
interface Token {  
    String getValueAsString();  
}
```

Java

```
interface Operand  
    extends Token {  
    int getValueAsInteger();  
    double getValueAsReal();  
    Operand sum(Operand arg);  
    Operand mul(Operand arg);  
    Operand sub(Operand arg);  
    Operand div(Operand arg);  
}
```

Java

```
enum Operator implements Token {  
    PLUS("+"), MINUS("-"),  
    TIMES("x"), DIVBY(":");  
    private Operator(String rep) {  
        this.rep = rep; }  
    private String rep;  
  
    @Override  
    public String toString() {  
        return rep; }  
  
    @Override  
    public String getValueAsString() {  
        return toString(); }  
}
```

Java



OPERANDI CONCRETI

- Quali operandi concreti?
 - il grande classico: **Real**
 - ma perché non anche un... **Colore** ?

```
class Real implements Operand {  
    private double v;  
    public Real(double v) { this.v = v; }  
    public String toString() { return String.valueOf(v); }  
    @Override public int getValueAsInteger() {  
        return (int) Math.round(v); }  
    @Override public double getValueAsReal() { return v; }  
    @Override public String getValueAsString() {  
        return String.valueOf(v); }  
    @Override public Operand sum(Operand that) {  
        return new Real(this.getValueAsReal + that.getValueAsReal()); }  
    ...  
}
```

Java

OPERANDI CONCRETI

- Il colore e le sue operazioni (fantasia al potere!)
 - ogni colore è associato a un valore reale, ma ha anche un ordinal
 - le operazioni sfruttano l'ordinal per sintetizzare un risultato

```
enum Color implements Operand {  
  
    BLU(0.2), GIALLO(1.1), VERDE(0.8), ROSSO(1.4), ARANCIO(1.6);  
  
    private Color(double v) { this.v = v; }  
    private double v;  
  
    @Override public int getValueAsInteger() { return ordinal(); }  
    @Override public double getValueAsReal() { return v; }  
    @Override public String getValueAsString() { return toString(); }  
    @Override public Operand sum(Operand that) {  
        return Color.values()[ this.getValueAsInteger() +  
                                that.getValueAsInteger() ]; }  
  
    ...  
}
```

Java



ALBERO E VALUTAZIONE

- Primo passo:
l'algoritmo resta quello già visto, ma si generalizza
 - la funzione **calc** manipola un **TreeItem<Token>** ma continua a restituire un **Integer**
 - al suo interno ricava gli interi associati agli operandi e li usa per fare i calcoli *esattamente come prima*
 - si usa **instanceof** per distinguere operatori e operandi
 - lo **switch** che distingue i diversi operatori non discrimina più stringhe, ma costanti di tipo **Operator**



ALBERO E VALUTAZIONE

```
private static Integer calc(TreeItem<Token> root) {  
    Stack<Integer> stack = new Stack<Integer>();  
    List<TreeItem<Token>> list = new ArrayList<>();  
    postorderEnumeration(root, list);  
    for (TreeItem<Token> item : list) {  
        if (item.getValue() instanceof Operand) {  
            Integer i = ( (Operand)item.getValue() ).getValueAsInteger();  
            stack.push(i);  
        } else {  
            Operator op = (Operator)item.getValue();  
            Integer v2 = stack.pop(), v1 = stack.pop();  
            switch (op) {  
                case PLUS: stack.push(v1 + v2); break;  
                case MINUS: stack.push(v1 - v2); break;  
                case TIMES: stack.push(v1 * v2); break;  
                case DIVBY: stack.push(v1 / v2); break;  
            }  
        }  
    }  
    return stack.pop();  
}
```

Java

Cast sicuri (protetti da instanceof)



ALBERO E VALUTAZIONE

```
public static void main(String[] args){  
    System.out.println(calcExp(mkTestExp1()));  
    System.out.println(calcExp(mkTestExp2()));  
    System.out.println(calcExp(mkTestExp3()));  
    System.out.println(calcExp(mkTestExp4()));  
}  
private static String calcExp(TreeItem<Token> exp) {  
    StringBuilder sb = new StringBuilder();  
    postorder(exp, sb, " ");  
    return "Il risultato di " + sb + " è " + calc(exp);  
}
```

Java

Test coi reali

Test coi colori

```
Il risultato di 3.0 4.0 5.0 x + è 23  
Il risultato di 3.0 4.0 + 5.0 x è 35  
Il risultato di VERDE GIALLO + è 3  
Il risultato di VERDE GIALLO x è 2
```

Buono

Migliorabile...



ALBERO E VALUTAZIONE

test coi reali

Java

```
public static TreeItem<Token> mkTestExp1() {
    TreeItem<Token> root = new TreeItem<>(Operator.PLUS);
    TreeItem<Token> l = new TreeItem<>(new Real(3));
    TreeItem<Token> r = new TreeItem<>(Operator.TIMES);
    TreeItem<Token> rl = new TreeItem<>(new Real(4));
    TreeItem<Token> rr = new TreeItem<>(new Real(5));
    r.getChildren().add(rl);    r.getChildren().add(rr);
    root.getChildren().add(l);  root.getChildren().add(r);
    return root;
}

public static TreeItem<Token> mkTestExp2() {
    TreeItem<Token> root = new TreeItem<>(Operator.TIMES);
    TreeItem<Token> l = new TreeItem<>(Operator.PLUS);
    TreeItem<Token> r = new TreeItem<>(new Real(5));
    TreeItem<Token> ll = new TreeItem<>(new Real(3));
    TreeItem<Token> lr = new TreeItem<>(new Real(4));
    l.getChildren().add(ll);    l.getChildren().add(lr);
    root.getChildren().add(l);  root.getChildren().add(r);
    return root;
}
```



ALBERO E VALUTAZIONE

test coi colori

```
public static TreeItem<Token> mkTestExp3() {
    TreeItem<Token> root = new TreeItem<>(Operator.PLUS);
    TreeItem<Token> l = new TreeItem<>(Color.VERDE);
    TreeItem<Token> r = new TreeItem<>(Color.GIALLO);
    root.getChildren().add(l);    root.getChildren().add(r);
    return root;
}

public static TreeItem<Token> mkTestExp4() {
    TreeItem<Token> root = new TreeItem<>(Operator.TIMES);
    TreeItem<Token> l = new TreeItem<>(Color.VERDE);
    TreeItem<Token> r = new TreeItem<>(Color.GIALLO);
    root.getChildren().add(l);    root.getChildren().add(r);
    return root;
}
```

Java

BLU (0.2) ,
GIALLO (1.1) ,
VERDE (0.8) ,
ROSSO (1.4) ,
ARANCIO (1.6) ;

Il risultato di VERDE GIALLO + è 3
Il risultato di VERDE GIALLO x è 2

Giallo.ordinal()=1
Verde.ordinal() =2



ARCHITETTURA ALTERNATIVA

- Carino, ma.. non ancora soddisfacente!
 - l'enumerativo non ha davvero cambiato le cose
 - c'è sempre lo switch, di fatto ha solo sostituito la stringa..
- L'obiettivo vero sarebbe *astrarre completamente*
 1. ogni operatore dovrebbe «inglobare» la «sua» operazione
 - se così fosse, non servirebbe più alcuno switch! 😊 😊
 2. la funzione **calc** non dovrebbe MAI manipolare interi
 - così, potrebbe essere *davvero* generica! 😊 😊
- IDEA: **Operator** dichiarare una *operazione astratta doOp*
 - ogni costante dell'enum definisca *la sua versione* di **doOp**



ARCHITETTURA ALTERNATIVA

Operator revised

- Inseriamo un metodo astratto `doOp`
 - ogni costante enumerativa lo implementa tramite *classe anonima*

```
public enum Operator implements Token {  
    PLUS("+") { @Override public Operand doOp(Operand arg1, Operand arg2) {  
        return arg1.sum(arg2); } },  
    MINUS("-") { @Override public Operand doOp(Operand arg1, Operand arg2) {  
        return arg1.sub(arg2); } },  
    TIMES("x") { @Override public Operand doOp(Operand arg1, Operand arg2) {  
        return arg1.mul(arg2); } },  
    DIVBY(":") { @Override public Operand doOp(Operand arg1, Operand arg2) {  
        return arg1.div(arg2); } };  
    private Operator(String rep){ this.rep = rep; }  
    private String rep;  
    public abstract Operand doOp(Operand arg1, Operand arg2);  
    @Override public String toString() { return rep; }  
    @Override public String getValueAsString() { return toString(); }  
}
```

Java

Classi anonime






NB: il metodo
astratto va
dichiarato *dopo*
le costanti



ARCHITETTURA ALTERNATIVA

Operator revised

- E infatti, guardando il compilato

 Operator\$1.class	01/05/2020 11:58	File CLASS
 Operator\$2.class	01/05/2020 11:58	File CLASS
 Operator\$3.class	01/05/2020 11:58	File CLASS
 Operator\$4.class	01/05/2020 11:58	File CLASS
 Operator.class	01/05/2020 12:13	File CLASS

Classi
anonime

- Conseguentemente, in `calc` non occorre più lo switch

```
else {  
    Operator operator = (Operator)item.getValue();  
    Operand v2 = stack.pop();  
    Operand v1 = stack.pop();  
    stack.push(operator.doOp(v1,v2));  
}
```

Java

Il calcolo è generalizzato!
Si chiede all'operatore si
svolgere «la sua» operazione



- Come «bonus», il risultato è visualizzato in modo coerente



ARCHITETTURA ALTERNATIVA

TEST FINALE

```
public static void main(String[] args){  
    System.out.println(calcExp(mkTestExp1()));  
    System.out.println(calcExp(mkTestExp2()));  
    System.out.println(calcExp(mkTestExp3()));  
    System.out.println(calcExp(mkTestExp4()));  
}  
  
private static String calcExp(TreeItem<Token> exp) {  
    StringBuilder sb = new StringBuilder();  
    postorder(exp, sb, " ");  
    return "Il risultato di " + sb + " è " + calc(exp);  
}
```

Java

BLU(0.2) ,
GIALLO(1.1) ,
VERDE(0.8) ,
ROSSO(1.4) ,
ARANCIO(1.6) ;

```
Il risultato di 3.0 4.0 5.0 x + è 12.0  
Il risultato di 3.0 4.0 + 5.0 x è 12.0  
Il risultato di VERDE GIALLO + è ROSSO  
Il risultato di VERDE GIALLO x è VERDE
```

Niente più
numeri! 😊

Giallo.ordinal()=1
Verde.ordinal() =2

RECAP

- Bisogna sempre pensare *bene* all'architettura software
- Una scelta oculata di interfacce, classi, enumerativi *permette di scrivere software estendibile e robusto*
- Il software «ben fatto» si riconosce perché «non contiene pasticci», non ha «toppe»
 - tutto viene *naturale*, chiaro e *semplice*
 - non ci sono *forzature*
 - il codice è chiaro e *naturalmente corto* perché non ci sono i «pasticci» per «girare intorno» alle cose pasticciate

