Java: Basilari di Programmazione in Piccolo

Sommario: 27 Aprile, 2015

- Overloading di Metodi
- Eccezioni: Definizione, Sollevamento, Mascheramento
- ADT e Modificatori
- Interfacce: Segnature, Gerarchia di Classi e Interfacce
- Polimorfismo e Generalizzazione dei tipi coinvolti.

Overloading

- **Overloading** Metodi statici e non, ereditati e non, che hanno stesso nome ma sono, a coppie, differenti:
  - per numero di argomenti, o per tipo di un argomento, oppure
  - se uno è ereditato, ha tipo calcolato che non è supertipo del tipo calcolato dell'altro metodo
- In caso di invocazione di metodo overloaded, a compile time è scelto quello tra gli applicabili più prossimo al tipo atteso.

```
class A{}
class B extends A{}
class C extends B{}

class E{
    void over(A x, A y){//overloaded
        System.out.println("sono overAA di E");
    }
    void over(A x, B y){//overloaded
        System.out.println("sono overAB di E");
    }
    void over(B x, C y){//overloaded
            System.out.println("sono overBC di E");
    }
}
class Main{
    public static void main(String[] argv){
        A x = new A();
        B y1 = new B(); B y2 = new B();
        E w = new E();
        new E().over(y1,y2);
    }
}
```

Overloading

- Cautela. Errori nella definizione, possono rendere overloaded un metodo che nelle intenzioni deve essere overridden.

```
import java.io.*;
import java.util.*;

class Point{
    double x;
    Point(double n1){
        x = n1;
    }
    public double distance (Point p){
        return x-p.x;
    }
}
class D2Point extends Point{
    double y;
    D2Point(double n1, double n2){
        super(n1);
        y = n2;
    }
    public double distance (D2Point p){
        D2Point u = (D2Point) p;
        double d = super.distance(p);
        return Math.sqrt(d*d+u.y*u.y);
    }
}
class Main{
    public static void main(String[] argv){
        Point p = new Point(3);
        D2Point u = new D2Point(0,0);
        System.out.println("come si comporta la valutazione
                di u.distance(p) "+u.distance(p));
    }
}
```

Gestione delle Eccezioni

Definition

Un eccezione è un evento, che occorre durante l'esecuzione, e interrompe il normale flusso del programma

- **Meccanismo delle Eccezioni** Si compone di 3 parti
  - **Definizione** delle eccezioni gestite dal programma
  - **Sollevamento** e generazione di un'eccezione
  - **Gestione** e possibile risoluzione dell'eccezione

Meccanismo delle Eccezioni: Definizione

- **Definizione** delle eccezioni gestite dal programma
  - Eccezioni = Oggetti che Java associa a eventi che interrompono il flusso de programma

    `if (factorial(x) > y) {...`

    quando factorial non calcola un intero positivo, il flusso è interrotto
  - **Sottoclassi** di `Java.lang.Exception`
    - Java.lang.Exception contiene 5 costruttori con argomenti per raccogliere informazioni sullo stato
    - useremo solo `Exception()` e `Exception(String)`

```
class IllegalArgsException extends Exception{
    public IllegalArgsException(String s){super(s);}
}
class EmptyValueException extends Exception{
}
```

    - IllegalArgException permette di specificare il metodo invocato;
    - EmptyValueException permette di esprimere eccezioni senza altre informazioni
    - La scelta dipende dal trattamento che faremo di tale eccezione

Meccanismo delle Eccezioni: Sollevamento

- **Sollevamento** di un'eccezione
  - Per generare un'eccezione di una classe A si usa il costrutto:

    `throw new A(...)`

    - Genera un'eccezione della classe A
    - Termina l'invocazione corrente del metodo con tale "sollevamento"

```
public void swap(myIntMutSeq q)throws IllegalArgsException{
    /* element swapping di this.val con q.val */
    if (q==null) throw new IllegalArgsException("swap");
    {
        int temp = val();
        valUpdate(q.val());
        q.valUpdate(temp);
    }
}
```

  - Le classi di eccezioni sollevabili devono essere dichiarate nell'intestazione del metodo, utilizzando:

    `throws A1,...,An`

Meccanismo delle Eccezioni: Gestione

- **Gestione** di un'eccezione è trattata dal metodo metodo invocante in 3 forme possibili:
  - **ri-sollevamento** quando non è in grado di trattarla
  - **mascheramento** quando è in grado di risolverla completamente
  - **mista** quando è in grado di trattare solo alcuni aspetti. È una combinazione delle due.
- Esaminiamo i 3 casi nella seguente struttura generale:

Meccanismo delle Eccezioni: Gestione - Caso Generale

- Esaminiamo i 3 casi nella seguente struttura generale:

```
class E extends Exception {
    public E(String s){super(s)};
}
class E1 extends E {
    public E1(String s){super(s)};
}
class E2 extends E {
    public E2(String s){super(s)};
}
class A {
    ...
    void callee(C x) throws E{
        ...
        ... throw new E1("M1: caso semplice");
        ...
        ... throw new E2("M1: caso complesso");
        ...
    }
    //...
}
class B {
    ...
    void caller() {
        ...//altri statements
        ... o.callee(v) ...//statement in cui occorre l'invocazione
        ...//altri statements
    }
}
```

dove abbiamo:

- una gerarchia di eccezioni: `E > E1,E2`
- un metodo `callee` che genera e solleva eccezioni
- un metodo `caller` che invoca `callee`

Meccanismo delle Eccezioni: Gestione - Risollevamento

- Ri-sollevamento. Consiste nel:
  - Dichiarare il metodo `caller` sollevante tali eccezioni
  - Lo statement di `o.callee(v)` rimane invariato
  - Quando `o.callee(v)` solleva un'eccezione `E`, `caller` termina e solleva tale eccezione

```
class A {
    ...
    void callee(C x) throws E{
        ...
        ... throw new E1("M1: caso semplice");
        ...
        ... throw new E2("M1: caso complesso");
        ...
    }
    //...
}
class B {
    ...
    void caller() throws E {
        ...//altri statements
        ... o.callee(v) ...//statement in cui occorre l'invocazione
        ...//altri statements
    }
}
```

Meccanismo delle Eccezioni: Gestione - Mascheramento

- Mascheramento. Usa uno specifico costrutto:

```
try{regione di codice mascherata}
catch(eccezione intercettata){trattamento dell'eccezione}
  ...
catch(eccezione intercettata){trattamento dell'eccezione}
```

  formato da 2 sezioni:

  - Regione di Codice Mascherata:
    - con cui racchiude il codice "critico"
  - Lista Casi Trattati:
    - discrimina i casi possibili
    - fornisce un trattamento risolutivo

Meccanismo delle Eccezioni: Gestione - Mascheramento/2

- Mascheramento. Usa uno specifico costrutto:
  - racchiude il codice "critico"
  - discrimina i casi possibili
  - fornisce un trattamento risolutivo

```
class A {
    ...
    void callee(C x) throws E{
        ...
        ... throw new E1("callee: caso semplice");
        ...
        ... throw new E2("callee: caso complesso");
        ...
    }
    //...
}
class B {
    ...
    void caller() throws E {
        ...//altri statements
        try {
            ... o.callee(v) ...
        }
        catch (E1 e1) {
            System.out.println("generata E1 in "+e.getMessage()+" risolta in caller");
            //...codice per il trattamento
        }
        catch (E2 e2) {
            System.out.println("generata E2 in "+e.getMessage()+" risolta in caller");
            //...codice per il trattamento
        }
        ...//altri statements
    }
}
```

Meccanismo delle Eccezioni: Mista

- Mista. Combinazione di Risollevamento e Mascheramento
- Un caso concreto.

```
public class eccezioni1{
//OVERVIEW: collezione di metodi stand_alone

public static Object access(Object a_vector, Object a_integer)
        throws NoVectorException, NoIntegerException, OutOfVectorBounds {
    Vector V;
    try {V = (Vector) a_vector;}
    catch (Exception e) { throw new NoVectorException("access:arg1"); }
    int i;
    try {i = ((Integer) a_integer).intValue();}
    catch(Exception e) { throw new NoIntegerException("access:arg2"); }
    if (V.size() > i) return V.elementAt(i);
    throw new OutOfVectorBounds();
}

// Compilate e provate l'esecuzione con il seguente main.

public static void main(String[] args) throws Exception {
    Vector My = new Vector();
    My.addElement("la Prova ha successo");
    My.addElement("Una seconda stringa");
    My.addElement(new Integer(5));
    System.out.println(access(My,My));
}

}
```

Interfaccia

Definition

Una interfaccia è la segnatura di una collezione di metodi correlati

- Ha struttura sintattica simile a quella di classe ma contiene solo: Intestazioni di metodi di istanza.

```
public interface DLinkSeq {
            public DLinkSeq pred();
            public DLinkSeq succ();
            public int val();
            public void valUpd(int v);
}
```

- Fondamentale Struttura nella Gerarchia delle classi e nella nozione di sottotipo.
- Una classe può essere sottoclasse di una o più interfacce
- Una classe è sottoclasse di un interfaccia se ...

Gerarcha di classi e interfacce

- Una classe può essere sottoclasse di una o più interfacce
- Una classe è sottoclasse di un interfaccia se ...
  - Dichiara di "implementare" l'interfaccia
  - Fornisce un **overriding** per ogni metodo dell'interfaccia
  - Un caso concreto.

```
interface sortable{
    public boolean ord(sortable s);
}

class sortedName implements sortable{
    private String name;
    public sortedName(String n){
        name = n;
    }
    public boolean ord(sortable s){
        return ((sortedName)s).name.startsWith(this.name);
    }
}
```

Example

ord in sortedName è riflessivo, e transitivo. Estenderlo in un preordine totale ovvero in modo tale che per ogni u,v, u.ord(v) oppure v.ord(u).

Modificatori di Accesso e ADT

La portata di un identificatore (classe, metodo, costruttore, ~~variabile~~, campo) in Java non è sufficiente per la sua accessibilità.

- Modificatori di Accesso: stabiliscono l'accessibilità di un identificatore all'interno della sua portata
- 4 tipi di accesso:
  - default (omesso): package. La più piccola regione tra la portata e il package in cui è dichiarato
  - **public**: mondo.
  - **private**: classe.
  - **protected**: package e sottoclassi.

- L'uso dei soli `private` e `public` permette di definire classi che si comportano come ADT.

ADT: Implementazione

- **private**: Ogni campo della classe (stato dei valori) e ogni metodo non nella segnatura dell'ADT
- **public**: I soli metodi della segnatura dell'ADT
- Un esempio concreto è lo ADT delle doubly-linked sequence, sotto

```
public class intMutSeq {
/* ----------- Stato oggetti intMutSeq ------------------------ */
    private intMutSeq Pred;
    private intMutSeq Succ;
    private int Val;
/* ------------ Operazioni: Costruttore ------------------------ */
/* intMutSeq() è alias di empty() ------------------------------- */
/* --------- cambieremo quando sapremo usare le eccezioni ------ */

    public intMutSeq(intMutSeq p, int v){//alias di add
        /* inserisce dopo p */
        Pred = p;
        Val = v;
        if (p!=null){...}
    }
/* ------------ Metodi per OPERAZIONI -------------------------- */
/* ------------ OSSERVATORI PER ACCESSO COMPONENTI ------------- */

    public intMutSeq pred(){...};
    public intMutSeq succ(){...};
    public int val(){...};
/* ------------ MODIDICATORI COMPONENTI ------------------------ */

    public void valUpdate(int v){...};
}
```

ADT: Segnatura

- La segnatura di un ADT può essere definita con un'interfaccia
- Implementata da una classe che implementata l'interfaccia

```
interface DLinkSeq{
    public DLinkSeq pred();
    public DLinkSeq succ();
    public int val();
    public void valUpd(int v);
}

public class intMutSeq implements DLinkSeq {
/* ------------ rappresentazione interna ----------------- */
    private intMutSeq Pred;
    private intMutSeq Succ;
    private int Val;
    public intMutSeq(intMutSeq p, int v){
        Pred = p;
        Val = v;
        if (p!=null){
            Succ = p.Succ;
            p.Succ = this;
            if (Succ!=null) Succ.Pred = this;
        }
    }
/* ------------ Implementazione Metodi ------------------- */

    public intMutSeq pred(){
        return Pred;
    };
    public intMutSeq succ(){
        return Succ;
    };
    public int val(){
        return Val;
    };
    public void valUpd(int v){
        Val = v;
    };
```

Polimorfismo Generico

- Java contiene varie forme di polimorfismo (Object, Generic, Subtype)
- Qui consideriamo il polimorfismo generico presente anche in altri linguaggi con lo scopo di:
- Generalizzare i tipi ai quali sono applicabili le definizioni introdotte nel programma.

```
class coppia {A left; B right;}
```

- Permette di riusare una stessa definizione invece di doverne definire una per ogni specifico tipo.

```
class coppia {A left; B right;}
class coppia {B left; A right;}
class coppia {C left; D right;}
```

Polimorfismo Generico: Variabile di Tipo e Parametro

- Il Polimorfismo Generico richiede di:
  - Estendere il sistema dei tipi, includendovi **variabili di tipo**
  - Permettere definizioni che sono parametriche su variabili di tipo

```
class coppia<T1,T2> {T1 left; T2 right;}
```

- Permette di riusare una classe parametrica, `coppia<T1,T2>`, come classe:

```
coppia<A,B> x = new coppia<A,B>()
coppia<B,A> y = ...
coppia<C,D> y = ...
```

- per introdurre oggetti, o variabili di tipo diverso e inconfrontabile

Polimorfismo Generico: Esempi

- Vediamo un primo esempio di coppia generica:

```
class pair <A,B> {
    A left;
    B right;
}
class Main{
    public static void main(String[] argv){
        pair<Integer,Integer> x = new pair<Integer,Integer>();
        x.left = new Integer(5);
        x.right = new Integer(7);
    }
}
```

- Una diversa definizione di coppia generica per usare conversioni int-Integer fornite dal compilatore

```
class pair <A,B> {
    A left;
    B right;
    pair(A x, B y){
        left = x;
        right = y;
    }
}
class Main{
    public static void main(String[] argv){
        pair<Integer,Integer> x = new pair<Integer,Integer>(5,7);
    }
}
```