Java: Additional Features

Sommario: 16 Maggio, 2017

- ADT: Ancora una condizione
- Collection: Vector e LinkedList
- Uguaglianza di valori: ==, equals
- Duplicazione di valori: clone
- Presentazione di valori: toString
- ADT per valori strutturati: elements.
- Enhanced for (o for each)

ADT: Ancora una condizione

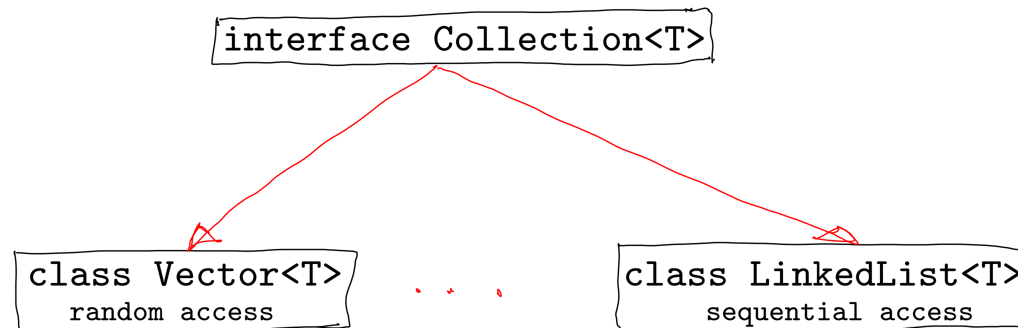
- ADT emulati in Java mediante classi e modificatori
- 3 condizioni:
  - **Stato Privato**
    Implementazione dei valori Inaccessibile
  - **Segnatura Pubblica**
    Uniche operazioni usabili dall'esterno della classe
  - **Esposizione Stato**
    Parametri trasmessi e Valori Calcolati delle operazioni pubbliche non devono mostrare parti dello stato.
    Esempio. `WrongStackImm2ADT` nell'allegato `stack`

Definition (Condizione di Non Esposizione dello Stato)

La stato della rappresentazione concreta non deve essere esposto in nessuna parte nè attraverso parametri nè attraverso il valore calcolato di un metodo pubblico. Quando la condizione è soddisfatta, l'ipotesi induttiva I(c) può essere assunta su c prima dell'invocazione di un metodo se provata vera sui costruttori di c.

Dati Strutturati

- **Collection** Interfaccia per gestire gruppi di oggetti
- 2 sottoclassi: Vector<T> e LinkedList<T>



Vector e LinkedList

- Vector<T>: Le operazioni che useremo (Vedi allegati)
- LinkedList<T>: Le operazioni che useremo (Vedi allegati)
- Vector<T>. Usiamo in:
  - Definizioni di tipi concreti
  - Definizioni di ADT: Stato Concreto
- LinkedList<T>. Usiamo in:
  - Definizioni di ADT: Supporto per ispezione contenuto valori astratti strutturati (vedi additionals)

Equivalenza di tipi e Assegnamento

- Java è un Linguaggio "fortemente tipato":

Definition (Strongly Typed, ST)

Ad ogni costruzione (espressione o comando) c, di ogni program- ma (legale) possiamo associare (a compile time) un **tipo unico** (possibilmente, un supertipo del tipo effettivo): $(\exists!\, \mathtt{T})\mathtt{c} : \mathtt{T}$

- I tipi di un Linguaggio ST hanno relazioni di equivalenza:
  - Strutturale e/o;
  - Nominale
- I tipi di Java hanno relazioni di equivalenza Nominale:

  `T1 equivalente T2 sse T1 è T2`

- Assegnamento:

  $(\mathtt{x} = \mathtt{e}) : \mathtt{T1}\ \ \mathtt{sse}\ \ (\mathtt{x}{:}\mathtt{T1} \wedge \mathtt{e}{:}\mathtt{T2} \wedge \mathtt{T1} > \mathtt{T2})$

Equivalenza di valori: ==, equals

- Categorie di Valori/Oggetti in Java:
  2 in accordo al **comportamento atteso**
  - **Modificabili (Mutable)**
    - `Stato oggetto può cambiare`
  - **NonModificabili (Immutable)**
    - `Stato oggetto non può cambia`
- Per l'equivalenza di valori Java possiede:
  - operatore ==
    - `v1==v2 sse stesso` **reference** `in memoria`
    - Corretto solo per Mutable e valori scalari (int, char, ...)
  - metodo **equals**:
    - Definito in Object ed ereditato da tutte le classi

      ```
      public boolean equals(Object o)
      ```
    - Ereditato: corretto solo per Mutable
    - Overridden: **obbligatoriamente** da tipi Immutable

ImPairADT e StackImm2ADTPE con equals

```
public class ImPairADTX <A,B> implements Cloneable{
    private final A left;//da nascondere
    private final B right;//da nascondere
    public ImPairADTX (A x, B y) {
        left = x;
        right = y;
    }
    public A getLeft(){...}
    public B getRight(){...}
    public boolean equals(Object o){//override equals
        ImPairADTX<?,?> ok;
        try{ok = (ImPairADTX<?,?>)o;}
        catch(Exception e){return false;}
        return (left.equals(ok.left) && right.equals(ok.right));
    }
```

Un'altro esempio è `StackImm2ADTPE` in Stack, allegato, dove vediamo un main per un suo uso

Duplicazione di Oggetti: clone

- metodo **clone**:
  - Definito in Object per tutte le classi (Cloneable)

```
protected Object clone()
                    throws CloneNotSupportedException
```

  - Crea una differente copia dell'oggetto: `x.clone()!=x`
  - Overridden: No
    obbligatoriamente da tutti i tipi Cloneable
    - **Immutable**: vedi file imPairADTX.java accluso
      - overriding pubblico di clone di Object
    - **Mutable**: vedi file muPairADTX.java accluso
      - interfaccia MuCloneable con overriding pubblico di clone
      - overriding pubblico di clone di MuCloneable
      - Uso di operatore: `instanceof`

ImPairADT con equals e clone

No : Rivisto

```
public class ImPairADTX <A,B> implements Cloneable{
    private final A left;//da nascondere
    private final B right;//da nascondere
    public ImPairADTX (A x, B y) {
        left = x;
        right = y;
    }
    public A getLeft(){...}
    public B getRight(){...}
    public boolean equals(Object o){...}
    protected ImPairADTX<A,B> clone() throws CloneNotSupportedException{
        return (new ImPairADTX<A,B>(left,right));
    }
    public String toString(){...}
}
```

Presentazione dei valori: toString

- metodo **toString**:
  - Definito in Object per tutte le classi
    ```
    public String toString()
    ```
  - Crea una stringa che rappresenta l'oggetto in modo testuale
  - Overidden: per fornire una presentazione dei valori

ImPairADT con equals, clone, toString

```
public class ImPairADTX <A,B> implements Cloneable{
    private final A left;//da nascondere
    private final B right;//da nascondere
    public ImPairADTX (A x, B y) {
        left = x;
        right = y;
    }
    public A getLeft(){...}
    public B getRight(){...}
    public boolean equals(Object o){...}
    protected ImPairADTX<A,B> clone() throws CloneNotSupportedException{...}
    public String toString(){
        return ("("+left.toString()+","+right.toString()+")");
    }
}
```

Un'altro esempio è `StackImm2ADTPE2` in Stack, allegato, dove vediamo un main per un suo uso

ImPairADTX con equals, clone, toString e Caso di uso

```
public class ImPairADTX <A,B> implements Cloneable{
    private final A left;//da nascondere
    private final B right;//da nascondere
    public ImPairADTX (A x, B y) {
        left = x;
        right = y;
    }
    public A getLeft(){...}
    public B getRight(){...}
    public boolean equals(Object o){//override equals
        ImPairADTX<?,?> ok;
        try{ok = (ImPairADTX<?,?>)o;}
        catch(Exception e){return false;}
        return (left.equals(ok.left) && right.equals(ok.right));
    }
    protected ImPairADTX<A,B> clone() throws CloneNotSupportedException{
        return (new ImPairADTX<A,B>(left,right));
    }
    public String toString(){
        return ("("+left.toString()+","+right.toString()+")");
    }
}

/...*/

/* (c) */
class main{
    public static void main(String args[])throws CloneNotSupportedException{
        ImPairADTX <Integer,String> myPlayCard = new ImPairADTX<Integer,String>(3,"fiori");
        System.out.println("il valore della carta è " + myPlayCard.getLeft());
        ImPairADTX <Integer,String> myPlayCard2 = new ImPairADTX<Integer,String>(3,"fiori");
        System.out.println("una copia della stessa carta? " + myPlayCard.equals(myPlayCard2));
        ImPairADTX <Integer,String> myPlayCard3 = myPlayCard.clone();
        System.out.println("una copia della stessa carta? " + myPlayCard.equals(myPlayCard3));
        System.out.println("una presentazione della carta è " + myPlayCard.toString());
    }
}
```

Collezione degli elementi: elements

- metodo **elements**:
  - Da definire in classi di oggetti strutturati: Liste, Alberi, Code, Insiemi...
  - Fornisce un valore `Collection` dei valori contenuti nel valore (astratto):
    - tutti gli elementi della lista
    - tutti i nodi dell'albero (oppure, tutti gli archi)
    - tutti gli oggetti nella coda

    ...
  - Lo esprimeremo con:

```
public LinkedList<T> elements()
```

Collezione degli elementi: elements /2

- metodo **elements**:
  - Permette di aumentare l'**usabilità** dei valori astratti proteggendone l'**integrità**

    vedi caso di uso in class Main di file `muSetADTX.java` più avanti (in collection e enhanced for)

Definition (Integrità di Valore o Dato)

Indica l'assenza di alterazioni non previste durante l'intera vita del valore

Valori Collection e enhanced for

- Interfaccia Collection è per valori che esprimono:
  - Collezioni di valori
  - Sono superTipi di classi importanti tra cui: `Vector<T>, LinkedList<T>`
- Utilizzabili nell'iterazione mediante *enhanced for*:
  - Sia `Coll<T>` una collection di valori di tipo T.
  - Sia `C` un oggetto di tipo `Coll<T>`.
  - Sia `code(x)` un codice nella variabile (libera) `x` di tipo `T`

    for`(T x:  C) code(x)`
  - Itera `code(x)` su ogni valore `u` di tipo `T` che sia in `C`;
  - Ad ogni iterazione `x` è legato ad un diverso `u` in `C`;
  - Ordine dei legami è ignoto e deve essere inessenziale per il programma.
- vedi caso di uso in file `muSetADTX.java` accluso.

elements e enhanced for

```
import java.lang.*;
import java.util.*;

interface Elements<T>{
    public LinkedList<T> elements();
}

class MuSetADTX<T> implements MuCloneable,Elements<T>{
    private boolean empty;
    private T elem;
    private MuSetADTX<T> rest;
//metodi
    public MuSetADTX(){...}
    public void add (T x) {...}
    public void remove (T x) {...}
    public boolean isEmpty () {...}
    public boolean isIn (T x) {...}
    public int size () {...}
//additionals
    ...
    public LinkedList<T> elements() {...}
}
class Main{
    public static void main(String args[]){
        MuSetADTX<Integer> aSet = new MuSetADTX<T>();
        aSet.add(3);
        ...
        if (!aSet.isEmpty()){
            int maxaSet = 0;
            for(Integer n: aSet.elements()){
                if(n>maxaSet)maxaSet=n;
            }
            ...
```