Java: Additional Features

Sommario: 16 Maggio, 2018

- 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.
- Collection: 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 della invocazione di un metodo se provata vera sui soli costruttori.

ADT: Esposizione dello stato

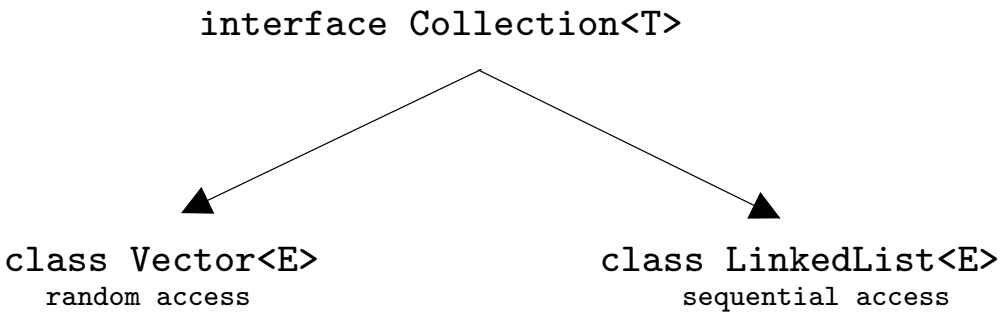
Definition (Condizione di Non Esposizione dello Stato)

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 della invocazione di un metodo se provata vera sui soli costruttori.

```
interface APISTKImm{
    public APISTKImm push(int k);
    public int top();
    public APISTKImm pop();
    public boolean isEmpty();
    public Vector<Integer> show();
        // added for content inspection
}
public class WrongStackImm2ADT implements APISTKImm{
    private Vector<Integer> p;
    //AF and I
    public WrongStackImm2ADT(){...}
    public WrongStackImm2ADT push(int k){...}
    public int top(){...}
    public WrongStackImm2ADT pop(){...}
    public boolean isEmpty(){...}
    public Vector<Integer> show(){
        return p;
    }
}
```

Dati Strutturati

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



`Vector<E>` . Le Operazioni che Useremo:

- `Vector<E>()` — costruttore 0-arity
- `boolean add(E e)` — modificatore
- `void add(int i, E e)` — modificatore
- `Object clone()` — produttore
- `boolean contains(Object o)` — osservatore
- `boolean equals(Object o)` — osservatore
- `E get(int i)` — osservatore
- `E remove(int i)` — modificatore
- `boolean remove(Object o)` — modificatore
- `E set(int i, E e)` — modificatore
- `int size()` — osservatore
- `String toString()` — osservatore

`LinkedList<E>`. Le Operazioni che Useremo:

- `LinkedList<E>()` — costruttore 0-arity
- `void add(int i, E e)` — modificatore
- `Object clone()` — produttore
- `boolean contains(Object o)` — osservatore
- `boolean equals(Object o)` — osservatore
- `E get(int i)` — osservatore
- `E remove(int i)` — modificatore
- `E set(int i, E e)` — modificatore
- `int size()` — osservatore
- `String toString()` — osservatore

Equivalenza di Tipi e Assegnamento

Definition (Strongly Typed, ST, Programming Language)

Ad ogni costruzione (espressione o comando) c, di ogni programma (legale) possiamo associare (a compile time) un **tipo unico** (un SuperTipo del tipo effettivo):

$$(\forall\, \mathtt{c})(\exists!\, \mathtt{T})\ \mathtt{c} : \mathtt{T}$$

- Java è un Linguaggio ST ("fortemente tipato"):
- I Tipi di un Linguaggio ST hanno relazioni di equivalenza:
  - Strutturale e/o;
  - Nominale
- I Tipi di Java hanno relazione di equivalenza Nominale:

  `T1 equivalente T2 sse T1 è T2`
- Assegnamento:

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

Equivalenza di valori: ==, equals

- Categorie di Oggetti in Java: Due Categorie in accordo al **comportamento atteso**
  - **Modificabili (Mutable)**
    - `Stato dell'oggetto può cambiare`
  - **NonModificabili (Immutable)**
    - `Stato dell'oggetto non può cambiare`
- 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 ed Overriding

- Metodo **equals**
  - Definito in Object ed ereditato da tutte le classi

    ```
    public boolean equals(Object o)
    ```
  - Su oggetti calcola come "==";
- **Mutable**
  - Uguali solo se sono lo stesso oggetto
  - equals **ereditata** calcola già correttamente
  - Overriding: NO, usare l'ereditata
- **Immutable**
  - Uguali se tutti i field corrispondenti sono uguali
  - equals ereditata non è adatta
  - Overriding: Si, per controllare i valori dei field corrispondenti

ImPairADT: Overriding di equals

- Come si esprime un overriding di **equals** quando usiamo polimorfismo generico?
- **equals** ha segnatura con polimorfismo Object

  `public boolean equals(Object o)`
- Usare cast da Tipo Object a Tipi generici ma a variabili anonime: "?"
- Non è necessario controllare che `"(ImPairADTX<?,?>)o"` abbia proprio `A` e `B` come variabili generiche.

```
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));
    }
```

StackImm2ADTPE: Overriding di equals

- **equals** ha segnatura con polimorfismo Object

  `public boolean equals(Object o)`
- Usare cast da Tipo Object a Tipi generici ma anonimi: "?"

```
interface APISTKImmPoly<A>{
    public APISTKImmPoly<A> push(A k);
    public A top();
    public APISTKImmPoly<A> pop();
    public boolean empty();
}
public class StackImm2ADTPE<A> implements APISTKImmPoly<A>{
    private Vector<A> p;
    //AF & I
    public StackImm2ADTPE(){...}
    public StackImm2ADTPE<A> push(A k){...}
    public A top(){...}
    public StackImm2ADTPE<A> pop(){...}
    public boolean empty(){...}
    public boolean equals(Object o){
        StackImm2ADTPE<?> ok;
        try {ok = (StackImm2ADTPE<?>)o;}
        catch (Exception e) {return false;}
        if (p.size()!=ok.p.size()) return false;
        for (int i=0;i<p.size()-1;i++){
            if (!(p.get(i).equals(ok.p.get(i))))
            return false;
        }
        return true;
    }
```

Duplicazione di Oggetti: "shallow", "deep", o "mista"?

- **Scopo**: Creare una copia `o.clone()` distinta di un oggetto `o`.
- **Copia distinta** significa soddisfare 3 proprietà:
  1) Identica Classe: `o.getClass() = o.clone().getClass()`[1]
  2) Oggetti Distinti: `o != o.clone()`
  3) Oggetti Uguali (in comportamento): `o.equals(o.clone())`
- **Soluzioni Diverse** soddisfano le 3 proprietà: Sia `o` un oggetto della classe `A`.
  - **Copia Shallow di** `o`. È un oggetto di `A` i cui campi contengono i valori dei corrispondenti campi di `o`.
  - **Copia Deep di** `o`. È un oggetto di `A` i cui campi contengono una Copia Deep dei corrispondenti campi di `o`.
  - **Copia Intermediate di** `o`. ... i cui campi contengono una Copia Intermediate dei corrispondenti campi di `o`.

[1] `getClass()` metodo di Object fornisce identificativo unico della classe del target a cui è applicata.

Duplicazione di Oggetti: clone() e il suo Meccanismo

Introduzione ed uso di un metodo **clone**: Poggia su un Meccanismo in 5 punti.

- **Meccanismo** realizzato mediante:
  - Definizione di un metodo **clone()** in **Object** con segnatura:
    `protected Object clone() throws CloneNotSupportedException`
  - Tutte le classi ereditano **clone()** da **Object**;
  - Definizione di interfaccia **Cloneable**;
  - Overriding di **clone()** ammesso solo per le classi che implementano **Cloneable**;
  - **Object** non implementa **Cloneable**. Al pari di ogni classe che estende Object, per definizione, ma non implementa **Cloneable**.

Duplicazione di Oggetti: Come Operare

- **clone()**
  - Signature:
    > `protected Object clone() throws CloneNotSupportedException`
  - Definizione e Comportamento:
    > `Definita in Object ed ereditata da tutte le classi;`
    > `Comportamento di o.clone():`
    dove: `o` di classe `A` e `clone()` metodo ereditato da Object
    - **solleva** `CloneNotSupportedException` se `A` non è `Cloneable`
    - **crea una Copia shallow** di `o` altrimenti

- **Cloneable**. Tutte le classi in cui si voglia **clone()**

Duplicazione di Oggetti: IMMUTABLE

- Overriding Obbligatorio per rendere `clone`:
  - > public
  - > calcolante duplicati "cast" sul Tipo della classe
  - > usabile all'esterno della classe

- Va bene il duplicato calcolato da `clone()` di Object:
  `super.clone()`

```
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){...}
    public ImPairADTX<A,B> clone() throws CloneNotSupportedException{
         return ((ImPairADTX<A,B>) super.clone());
    }
    public String toString(){...}
}
```

Duplicazione di Oggetti: MUTABLE

- Overriding Obbligatorio per rendere `clone`:
  > public, cast ed usabile (come negli Immutable)
- Una Copia Shallow potrebbe lasciare "accessi allo stato" condivisi da target e duplicato
- Il duplicato calcolato da `clone()` di Object non va bene
- Una Copia Deep: Uso di operatore `instanceof`

```
class MuPairADTX <A,B> implements Cloneable{
    private A left;//da nascondere
    private B right;//da nascondere
    public MuPairADTX(A x, B y) throws invalidArgException{
        if(x == null || y==null) throw new invalidArgException();
        left = x; right = y;
    }
    public A getLeft(){...}
    public B getRight(){...}
    public void setLeft(A x)throws invalidArgException{...}
    public void setRight(B y)throws invalidArgException{...}
    //public boolean equals(Object o) ereditata
    public MuPairADTX<A,B> clone() throws CloneNotSupportedException{
        MuPairADTX<A,B> res;
        try{
            A newLeft  =left;
            B newRight = right;
            if (left instanceof Cloneable) newLeft = (A)((Cloneable)left).clone();
            if (right instanceof Cloneable) newRight = (B)((Cloneable)right).clone();
            res = new MuPairADTX<A,B>(newLeft,newRight);
        } catch(invalidArgException e){return null;}
        return res;
    }
    public String toString(){
        return ("<"+left.toString()+","+right.toString()+">");
    }
}
```

Duplicazione di Oggetti: MUTABLE -

- Uso di Shallow

```
package MuPairPack;

//import java.io.*;
import java.util.*;

public class MyMPair implements Cloneable{
    private Vector<Integer> rep;
    private MyMPair(){};
    public MyMPair(int n1, int n2){
        rep = new Vector<Integer>();
        rep.add(0,n1); rep.add(1,n2);
    }
    public int left(){return rep.get(0);}
    public int right(){return rep.get(0);}
    public String toString(){return "<" + rep.get(0).toString() + ...}
    public void updL(int n){rep.set(0,n);}
    protected MyMPair clone() throws CloneNotSupportedException{
        //Copia Shallow
        return (MyMPair) super.clone();
    }
}
```

Duplicazione di Oggetti: MUTABLE -

- Uso di Shallow: equivalente

```
package MuPairPack;

//import java.io.*;
import java.util.*;

public class MyMbisPair implements Cloneable{
    private Vector<Integer> rep;
    private MyMbisPair(){};
    public MyMbisPair(int n1, int n2){...}
    public int left(){...}
    public int right(){...}
    public String toString(){....}
    public void updL(int n){...}
    protected MyMbisPair clone() throws CloneNotSupportedException{
        //return (MyMPair) super.clone();
        MyMbisPair res = new MyMbisPair();
        res.rep = rep;
        return res;
    }
}
```

Duplicazione di Oggetti: MUTABLE -

- Uso di Deep

```
package MuPairPack;

//import java.io.*;
import java.util.*;

public class MyMuPair implements Cloneable{
    private Vector<Integer> rep;
    public MyMuPair(int n1, int n2){...}
    public int left(){...}
    public int right(){...}
    public String toString(){...}
    public void updL(int n){...}
    protected MyMuPair clone() throws CloneNotSupportedException{
        MyMuPair res = (MyMuPair) super.clone();
        if (res.rep instanceof Cloneable)
            res.rep = (Vector<Integer>)res.rep.clone();
        return res;
    }
}
```

Duplicazione di Oggetti: MUTABLE -

- Tests

```
package MuPairPack;

import java.io.*;
//import java.util.*;

class Test1{
    public static void main(String args[]) throws CloneNotSupportedException{
        MyMPair v1 = new MyMPair(3,15);
        MyMPair v1Clone = v1.clone();
        System.out.println("v1 = " + v1.toString() + ";  v1Clone = " + v1Clone);
        v1.updL(7);
        System.out.println("v1 = " + v1.toString() + ";  v1Clone = " + v1Clone);
    }
}
class Test2{
    public static void main(String args[]) throws CloneNotSupportedException{
        MyMbisPair v1 = new MyMbisPair(3,15);
        MyMbisPair v1Clone = v1.clone();
        System.out.println("v1 = " + v1.toString() + ";  v1Clone = " + v1Clone);
        v1.updL(7);
        System.out.println("v1 = " + v1.toString() + ";  v1Clone = " + v1Clone);
    }
}
class Test3{
    public static void main(String args[]) throws CloneNotSupportedException{
        MyMuPair v1 = new MyMuPair(3,15);
        MyMuPair v1Clone = v1.clone();
        System.out.println("v1 = " + v1.toString() + ";  v1Clone = " + v1Clone);
        v1.updL(7);
        System.out.println("v1 = " + v1.toString() + ";  v1Clone = " + v1Clone);
    }
}
```

Duplicazione di Oggetti: MUTABLE -

- Comportamenti

```
MarcoBelliasAir:PairPack marcob$ java MuPairPack/Test1
v1 = <3,15>;  v1Clone = <3,15>
v1 = <7,15>;  v1Clone = <7,15>
MarcoBelliasAir:PairPack marcob$ java MuPairPack/Test2
v1 = <3,15>;  v1Clone = <3,15>
v1 = <7,15>;  v1Clone = <7,15>
MarcoBelliasAir:PairPack marcob$ java MuPairPack/Test3
v1 = <3,15>;  v1Clone = <3,15>
v1 = <7,15>;  v1Clone = <3,15>
MarcoBelliasAir:PairPack marcob$
```

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

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;
            }
            ...
```