Completamenti e Tipi Astratti:

Sommario: 22 Maggio, 2018

- **Completamenti:**
  > Metodo elements e Iteratori (Lezione17)

- **Il Tipo di Dato `Tree` in Java**
  > Stato: Struttura MultiComponente: AF & I
  > Interfaccia Label e Polimorfismo Object
  > Costruttori: Scelta
  > Eccezioni e Operazioni: `getLabel`, `getSons`, `add`, `remove`
  > Gli Additional: `equals`, `toString`, `clone`
  > clone: Valori dei Componenti e Struttura dello Stato
  > clone: Copia Shallow, Deep o Intermediate

- **Una Versione con Polimorfismo Generico:** `Tree<T>`
- **Il Tipo Astratto con Polimorfismo Generico:** `API<T>`
  > Usiamo `Tree<T>` e scriviamo una `API<T>` adatta
  > É adatta? Verifichiamolo con il compilatore
  > Problemi: Perchè?
  > Soluzioni: Perchè?
- **Esercizi**

Metodo `elements` e Iteratori

- 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

```
public LinkedList<A> elements(){
    //lista Valori Label
    if (label == null)return new LinkedList<A>();
    LinkedList<A> ret = sons.elements();
    ret.add(label);
    return ret;
}
```

Tipo Astratto `API<A>` in `TREES/MuGAETREE/Tree.java`

- **v.elements()**: Fornisce una `Collection` dei valori contenuti nel valore strutturato `v`
  - Permette di aumentare l'**usabilità** dei Tipi Astratti proteggendone l'**integrità**
  - Migliora l'**usabilità** dei Tipi di Dato indicando un'operazione preposta allo scopo

Definition (Integrità di Valore o Dato)

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

Valori Collection e iterazione enhanced for

- Interfaccia Collection è per valori che esprimono:
  - Collezioni di valori
  - È superTipo 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

- Enhance for è analogo a `List.fold_left`, in OCaml, dove la collection è la lista e `code(x)` è la funzione da iterare.

elements e enhanced for

```
class Test6{//elements
    public static void main(String[] args) throws CloneNotSupportedException{
        Tree<String> t1 = new Tree<String>("A",null);
        Tree<String> t2 = new Tree<String>("B",null);
        t1.add(1,t2);
        Tree<String> t8 = t1.clone();
        Scanner StdIn = new Scanner(System.in);
        String search = StdIn.next();
        System.out.println("le etichette dei nodi di " + t8 + " sono " + t8.elements());
        for(String x: t8.elements()){
            if (x.equals(search)){
                System.out.println(search + " etichetta un nodo di " + t8);
                return;
            }
        }
      System.out.println(search + " non etichetta alcun nodo di " + t8);
    }
}
```

Tipo Astratto `API<A>` in TREES/MuGAETREE/Test.java

Il Tipo di Dato `Tree` in Java: Stato Concreto e AF & I

- **Stato Concreto:**AF & I

```
public class Tree implements Cloneable {
    private Label label;
    private Vector<Tree> sons;
    /*
     AF(c) = < > iff c.label==null
     AF(c) = <a - t1 ... tn> iff (c.label == a)&&(c.sons.size() == n)&&
                                 (forall i\in[1..n], c.sons.get(i-1) == ti))
     I(c) = (c.label == null)||(c.label != null && c.sons != null)
    */
```

> La forma `< a - [t1, ..., tn]>` della presentazione si desume dalla definizione di `toString()`, data nel testo.

Interfaccia Label e Polimorfismo Object

- **Polimorfismo Object: I Tree sono etichettati su Label che è un'Interfaccia**.

```
public interface Label extends Cloneable{
}
```

> Riuso del codice `Tree` su ogni classe `A` che "implementi" `Label`

```
class A implements Label{String v; ...}
class B implements Label{Integer v;...}
...
class Z implements Label{...}

class Test6{//Polimorfismo Object
    public static void main(String[] args) {
        Tree t1 = new Tree(new B(100),null);
        Vector<Tree> sons1 = new Vector<Tree>();
        sons1.add(t1);
        Tree t2 = new Tree(new A("A"),sons1);
        System.out.println("t1 = " + t1);
        System.out.println("t2 = " + t2);
    }
}
```

> Esecuzione:

```
etruria-wifi-217-207:MuObjTREE marcob$ java TreePack/Test6
t1 = <100 - >
t2 = <A - [<100 - >]>
etruria-wifi-217-207:MuObjTREE marcob$
```

- In questo corso non approfondiamo oltre questa forma di Polimorfismo interessante ma con proprietà non del tutto chiare.

Costruttori: Scelta

- il Testo indicava la signature di 2 costruttori da definire:

```
public Tree(){
}
public Tree(Label label, Vector<Tree> sons){
    ...
}
```

  > Il primo crea alberi vuoti (e lo useremo per definire clone in forma appropriata allo stato utilizzato in Tree)
  > Il primo ha corpo `super()` di default

- Entrambi devono rispettare `AF&I`:

```
public Tree(){
}
public Tree(Label label, Vector<Tree> sons){
    this.label = label;
    this.sons = new Vector<Tree>();
    if (sons == null) return;
    for (int i=0; i<sons.size(); i++){//copiare il contenuto
        this.sons.add(i,sons.get(i)); //per evitare esposizione
    }                                 //della rappresentazione
}
```

Eccezioni e Operazioni: `getLabel`, `getSons`, `add`, `remove`

- Eccezioni: Potremmo anche non introdurne di nuove
  - NullPointerException, OutOfBoundsException, IllegalArgumentException: Sono tutte `RuntimeException`.
  - Sono tutte sollevale da `Vector.add` e Vector.remove, quando invocate da `add` e `remove`, definite nel testo.

- getLabel: Banale, restituisce il valore del componente label

```
public Label getLabel(){
    return label;
}
```

- getSons: Restituisce una copia della struttura con i valori

```
public Vector<Tree> getSons(){
    Vector<Tree> ret = new Vector<Tree>();//copiare il contenuto
    for (int i=0; i<sons.size(); i++){    //per evitare esposizione
        ret.add(i,sons.get(i));           //rappresentazione
    }
    return ret;
}
```

  - La struttura Vector dei valori deve essere nuova
  - I valori (Immutable o no) devono essere gli stessi contenuti nella struttura

Additional: Equals

- I Tree sono Valori MUTABLE
- Object.equals() è definita per calcolare correttamente con i MUTABLE
- Object.equals() è ereditata ed usabile se non ridefinita in Tree.
- ERRORE fornire in Tree una nuova definizione di `equals`.

Additional: clone

- Deve essere sempre ridefinita per fornire come tipo calcolato la classe invece di Object
- Dobbiamo creare una copia dell'oggetto con value reference differente e stato:
  - Copia Shallow
  - Copia Deep
  - Copia Intermediate
- Diamo la definizione per ciascuno dei 3 e vediamone il comportamento su un test di uso.

Test di uso

- Diamo la definizione per ciascuno dei 3 e ne vediamo il comportamento sul seguente test di uso.

```
class Test5{//clone a Copia ...
    public static void main(String[] args) throws CloneNotSupportedException{
        Tree t1 = new Tree(new A("A"),null);
        Tree t2 = new Tree(new A("B"),null);
        t1.add(1,t2);
        Tree t8 = t1.clone();
        System.out.print("t1 = " + t1);
        System.out.println(", e t8 = " + t8);
        System.out.println("t1 e t8 stesso valore in questo istante,\nMa t1 uguale t8 in ogni
            possibile stato del calcolo? --- " + (t1.equals(t8)));
        //Il requisito su uguaglianza tra oggetto e suo clone solo per IMMUTABLE
        //-- Le etichette di un nodo non possono cambiare.
        //   Domandiamoci se i due alberi hanno radice etichettata nello stesso modo
        System.out.println("t1 e il suo clone hanno stessa eticetta? --- " + (t1.getLabel().
            equals(t8.getLabel())));
        //-- I figli possono cambiare e vorremmo che cambiassero in modo indipenente
        //   Domandiamoci se i due alberi sono indipendenti sui figli.
        t1.add(2,t8); //aggiunto 1 figlio a t8
        System.out.println("t1 ha figli: " + t1.getSons());
        System.out.println("t8 ha figli: " + t8.getSons());
    }
}
```

- Definiamo e proviamo su `Test5` ciascuna delle 3 Copie.

Additional: clone a Copia Shallow

```
public Tree clone()throws CloneNotSupportedException{
    return (Tree) super.clone(); //overriding a Copia Shallow
}//Meglio Copia Deep? Discutere
```

```
class Test5{//clone a Copia ...
    public static void main(String[] args) throws CloneNotSupportedException{
        Tree t1 = new Tree(new A("A"),null);
        Tree t2 = new Tree(new A("B"),null);
        t1.add(1,t2);
        Tree t8 = t1.clone();
        System.out.print("t1 = " + t1);
        System.out.println(", e t8 = " + t8);
        System.out.println("t1 e t8 stesso valore in questo istante,\nMa t1 uguale t8 in ogni
            possibile stato del calcolo? --- " + (t1.equals(t8)));
        //Il requisito su uguaglianza tra oggetto e suo clone solo per IMMUTABLE
        //-- Le etichette di un nodo non possono cambiare.
        //   Domandiamoci se i due alberi hanno radice etichettata nello stesso modo
        System.out.println("t1 e il suo clone hanno stessa eticetta? --- " + (t1.getLabel().
            equals(t8.getLabel())));
        //-- I figli possono cambiare e vorremmo che cambiassero in modo indipenente
        //   Domandiamoci se i due alberi sono indipendenti sui figli.
        t1.add(2,t8); //aggiunto 1 figlio a t8
        System.out.println("t1 ha figli: " + t1.getSons());
        System.out.println("t8 ha figli: " + t8.getSons());
    }
}
```

```
MarcoBelliasAir:MuTREE marcob$ javac Label.java Tree.java Test.java -d .
MarcoBelliasAir:MuTREE marcob$ java TreePack/Test5
t1 = <A - [<B - >]>, e t8 = <A - [<B - >]>
t1 e t8 stesso valore in questo istante,
Ma t1 = t2 in ogni possibile stato del calcolo? --- false
t1 e il suo clone hanno stessa eticetta? --- true
Exception in thread "main" java.lang.StackOverflowError
...
```

Additional: clone a Copia Deep

```
public Tree clone()throws CloneNotSupportedException{
    Tree ret = (Tree) super.clone(); //overriding a Copia Deep
    if (label instanceof Cloneable)ret.label = (Label)((A)label).clone();
    if (sons instanceof Cloneable) ret.sons = (Vector<Tree>)sons.clone();
    return ret;
}//Meglio Copia Shallow? (vedi WrongSol)
```

```
class Test5{//clone a Copia ...
    public static void main(String[] args) throws CloneNotSupportedException{
        Tree t1 = new Tree(new A("A"),null);
        Tree t2 = new Tree(new A("B"),null);
        t1.add(1,t2);
        Tree t8 = t1.clone();
        System.out.print("t1 = " + t1);
        System.out.println(", e t8 = " + t8);
        System.out.println("t1 e t8 stesso valore in questo istante,\nMa t1 uguale t8 in ogni
            possibile stato del calcolo? --- " + (t1.equals(t8)));
        //Il requisito su uguaglianza tra oggetto e suo clone solo per IMMUTABLE
        //-- Le etichette di un nodo non possono cambiare.
        //   Domandiamoci se i due alberi hanno radice etichettata nello stesso modo
        System.out.println("t1 e il suo clone hanno stessa eticetta? --- " + (t1.getLabel().
            equals(t8.getLabel())));
        //-- I figli possono cambiare e vorremmo che cambiassero in modo indipenente
        //   Domandiamoci se i due alberi sono indipendenti sui figli.
        t1.add(2,t8); //aggiunto 1 figlio a t8
        System.out.println("t1 ha figli: " + t1.getSons());
        System.out.println("t8 ha figli: " + t8.getSons());
    }
}
```

```
etruria-wifi-217-207:MuObjTREE marcob$ javac Label.java Tree.java Test.java -d .
Note: Tree.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
MarcoBelliasAir:MuTREE marcob$ java TreePack/Test5
t1 = <A - [<B - >]>, e t8 = <A - [<B - >]>
t1 e t8 stesso valore in questo istante,
Ma t1 = t2 in ogni possibile stato del calcolo? --- false
t1 e il suo clone hanno stessa eticetta? --- false
t1 ha figli: [<B - >, <A - [<B - >]>]
t8 ha figli: [<B - >]
```

Additional: clone a Copia Intermediate

```
public Tree clone()throws CloneNotSupportedException{
    Tree ret = (Tree) super.clone(); //overriding a Copia Intermediate
    if (sons instanceof Cloneable)//Clone solo sulla struttura della
        ret.sons = (Vector<Tree>)sons.clone(); //rappresentazione
    return ret;
}//Meglio Copia Shallow? o Copia Deep (vedi WrongSol)
class Test5{//clone a Copia ...
    public static void main(String[] args) throws CloneNotSupportedException{
        Tree t1 = new Tree(new A("A"),null);
        Tree t2 = new Tree(new A("B"),null);
        t1.add(1,t2);
        Tree t8 = t1.clone();
        System.out.print("t1 = " + t1);
        System.out.println(", e t8 = " + t8);
        System.out.println("t1 e t8 stesso valore in questo istante,\nMa t1 uguale t8 in ogni
            possibile stato del calcolo? --- " + (t1.equals(t8)));
        //Il requisito su uguaglianza tra oggetto e suo clone solo per IMMUTABLE
        //-- Le etichette di un nodo non possono cambiare.
        //   Domandiamoci se i due alberi hanno radice etichettata nello stesso modo
        System.out.println("t1 e il suo clone hanno stessa eticetta? --- " + (t1.getLabel().
            equals(t8.getLabel())));
        //-- I figli possono cambiare e vorremmo che cambiassero in modo indipenente
        //   Domandiamoci se i due alberi sono indipendenti sui figli.
        t1.add(2,t8); //aggiunto 1 figlio a t8
        System.out.println("t1 ha figli: " + t1.getSons());
        System.out.println("t8 ha figli: " + t8.getSons());
    }
}
```

```
etruria-wifi-217-207:MuObjTREE marcob$ javac Label.java Tree.java Test.java -d .
Note: Tree.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
etruria-wifi-217-207:MuObjTREE marcob$ java TreePack/Test5
t1 = <A - [<B - >]>, e t8 = <A - [<B - >]>
t1 e t8 stesso valore in questo istante,
Ma t1 uguale t8 in ogni possibile stato del calcolo? --- false
t1 e il suo clone hanno stessa eticetta? --- true
t1 ha figli: [<B - >, <A - [<B - >]>]
t8 ha figli: [<B - >]
```

clone: da Polim. `Object` a Polim. Generico

- clone a Copia Intermediate è la soluzione giusta

```
public Tree clone()throws CloneNotSupportedException{
    Tree ret = (Tree) super.clone(); //overriding a Copia Intermediate
    if (sons instanceof Cloneable)//Clone solo sulla struttura della
        ret.sons = (Vector<Tree>)sons.clone(); //rappresentazione
    return ret;
}//Meglio Copia Shallow? o Copia Deep (vedi WrongSol)
```

- Ma la compilazione della definizione ha `warnings di cast`

```
etruria-wifi-217-207:MuObjTREE marcob$ javac Label.java Tree.java Test.java -Xlint:unchecked -d .
Tree.java:76: warning: [unchecked] unchecked cast
            ret.sons = (Vector<Tree>)sons.clone(); //rappresentazione
                                     ^
  required: Vector<Tree>
  found:    Object
1 warning
etruria-wifi-217-207:MuObjTREE marcob$
```

- Vediamo in dettaglio i problemi di cast rilevati e risolviamoli
- `sons.clone()` calcola un `Object` (vedi Vector.clone())
- Ma Java non è in grado di controllare cast da Tipi a Polimorfismo `Object` a Tipi a Polimorfismo Generico.

clone: Una diversa definizione in `Tree.java`

- clone a Copia Intermediate è la soluzione giusta
- Ma non dobbiamo usare `Vector.clone()` in `sons.clone()`
- Provvediamo a generare una copia ad Hoc del Vector `sons`

```
public Tree clone()throws CloneNotSupportedException{
    Tree ret = new Tree();//overriding a Copia Intermediate
    ret.label = label;
    ret.sons = new Vector<Tree>();//Clone solo sulla struttura della
    for(int i = 0; i < sons.size(); i++){
        ret.sons.add(i,sons.get(i));
    }
    return ret;
}//soluzione corretta
```

- Ed ecco compilazione e comportamento nel test di uso

```
etruria-wifi-217-207:MuObjTREE marcob$ javac Label.java Tree.java Test.java -Xlint:unchecked -d .
etruria-wifi-217-207:MuObjTREE marcob$ java TreePack/Test5
t1 = <A - [<B - >]>, e t8 = <A - [<B - >]>
t1 e t8 stesso valore in questo istante,
Ma t1 uguale t8 in ogni possibile stato del calcolo? --- false
t1 e il suo clone hanno stessa eticetta? --- true
t1 ha figli: [<B - >, <A - [<B - >]>]
t8 ha figli: [<B - >]
etruria-wifi-217-207:MuObjTREE marcob$
```

class Tree (con Polimorfismo Object): Operazioni

```
public class Tree implements Cloneable {
    private Label label;
    private Vector<Tree> sons;
    /*
     AF(c) = < > iff c.label==null
     AF(c) = <a - t1 ... tn> iff (c.label.equals(a))&&(c.sons.size()==n)&&
                                 (forall i\in[1..n], c.sons.get(i-1).equals(t1))
     I(c) = (c.label==null)||(c.label!=null && c.sons!=null)
    */
    public Tree(){
    }
    public Tree(Label label, Vector<Tree> sons){
        this.label = label;
        this.sons = new Vector<Tree>();
        if (sons == null) return;
        for (int i=0; i<sons.size(); i++){//copiare il contenuto
            this.sons.add(i,sons.get(i)); //per evitare esposizione
        }                                 //della rappresentazione
    }
    public Label getLabel(){
        return label;
    }
    public Vector<Tree> getSons(){
        Vector<Tree> ret = new Vector<Tree>();//copiare il contenuto
        for (int i=0; i<sons.size(); i++){    //per evitare esposizione
            ret.add(i,sons.get(i));           //rappresentazione
        }
        return ret;
    }
    public void add(int i, Tree t){
        sons.add(i-1,(Tree)t);
    }
    public void remove(int i){
        sons.remove(i-1);
    }
```

class Tree (con Polimorfismo Object): Additionals

```
    //Overriding di equals inutile: L'ereditata e' adeguata
    public Tree clone()throws CloneNotSupportedException{
        Tree ret = new Tree();//overriding a Copia Intermediate
        ret.label = label;
        ret.sons = new Vector<Tree>();//Clone solo sulla struttura della
        for(int i = 0; i < sons.size(); i++){
            ret.sons.add(i,sons.get(i));
        }
        return ret;
    }//soluzione corretta
    public String toString(){
        if (label == null) return ("< >");
        if (sons.size() == 0) return ("<" + label.toString() + " - >");
        return ("<" + label.toString() + " - " + sons.toString() + ">");
    }
}
```

class Tree<T> con Polimorfismo Generico: Operazioni

```
public class Tree<A> implements Cloneable {
    private A label;
    private Vector<Tree<A>> sons;
    /*
     AF(c) = < > iff c.label==null
     AF(c) = <a - t1 ... tn> iff (c.label.equals(a))&&(c.sons.size()==n)&&
                                 (forall i\in[1..n], c.sons.get(i-1).equals(t1))
     I(c) = (c.label==null)||(c.label!=null && c.sons!=null)
    */
    public Tree(){
    }
    public Tree(A label, Vector<Tree<A>> sons){
        this.label = label;
        this.sons = new Vector<Tree<A>>();
        if (sons == null) return;
        for (int i=0; i<sons.size(); i++){//copiare il contenuto
            this.sons.add(i,sons.get(i)); //per evitare esposizione
        }                                 //della rappresentazione
    }
    public A getLabel(){
        return label;
    }
    public Vector<Tree<A>> getSons(){
        Vector<Tree<A>> ret = new Vector<Tree<A>>();//copiare il contenuto
        for (int i=0; i<sons.size(); i++){   //per evitare esposizione rapp.
            ret.add(i,sons.get(i));
        }
        return ret;
    }
    public void add(int i, Tree<A> t){
        sons.add(i-1,(Tree<A>)t);
    }
    public void remove(int i){
        sons.remove(i-1);
    }
```

class Tree<T> con Polimorfismo Generico:Additionals

Errori nell'uso del Polimorfismo: Dove e Perchè?

```
    //Overriding di equals inutile: L'ereditata e' adeguata
    public Tree<A> clone()throws CloneNotSupportedException{
        //Copia Intermediate
        Tree<A> ret = (Tree<A>)super.clone();
        ret.sons = (Vector<Tree<A>>)sons.clone();
        return ret;
    }
    public String toString(){
        if (label == null) return ("< >");
        if (sons.size() == 0) return ("<" + label.toString() + " - >");
        return ("<" + label.toString() + " - " + sons.toString() + ">");
    }
}
```

Il compilatore segnale ben 2 warnings.

class Tree<T> con Polimorfismo Generico:Additionals

In questa versione gli errori sono stati rimossi:
Riscrivendo il codice che faceva uso di `Object.clone` e di `Vector.clone`, entrambi calcolanti tipo `Object`

```
    //Overriding di equals inutile: L'ereditata e' adeguata
    public Tree<A> clone()throws CloneNotSupportedException{
        //Copia Intermediate
        Tree<A> ret = new Tree<A>();
        if (label == null) return ret;
        ret.label = label;
        ret.sons = new Vector<Tree<A>>();
        for (int i = 0; i < sons.size(); i++){
            ret.sons.add(i,sons.get(i));
        }
        return ret;
    }
    public String toString(){
        if (label == null) return ("< >");
        if (sons.size() == 0) return ("<" + label.toString() + " - >");
        return ("<" + label.toString() + " - " + sons.toString() + ">");
    }
}
```

Un Tipo Astratto con `Tree<T>` come ADT

- **API**: Come scriverne una adatta per `Tree<T>` come ADT?.
- **API**: Scriviamo quella più ovvia.

```
public interface API<A> {
    public A getLabel();
    public Vector<API<A>> getSons();
    public void add(int n, API<A> t);
    public void remove(int n);
}
```

- Compiliamo e vediamo se supera l'Analisi dei Tipi

```
MarcoBelliasAir:MuGAETREE marcob$ javac API.java Tree.java -d .
Tree.java:35: error: Tree is not abstract and does not override abstract method getSons() in API
public class Tree<A> implements API<A>, Cloneable {
       ^
  where A is a type-variable:
    A extends Object declared in class Tree
Tree.java:57: error: getSons() in Tree cannot implement getSons() in API
    public Vector<Tree<A>> getSons(){
                           ^
  return type Vector<Tree<A#1>> is not compatible with Vector<API<A#1>>
  where A#1,A#2 are type-variables:
    A#1 extends Object declared in class Tree
    A#2 extends Object declared in interface API
2 errors
MarcoBelliasAir:MuGAETREE marcob$
```

Overriding di `getSons`: Errori nei Tipi e Cause

- Gli errori sono causati da `getSons`: Ma non sono di dell'API o dell'ADT, piuttosto sono di entrambi.

- **Signature** `getSons` in `API<A>` e in `Tree<A>` incompatibili.
  `public Vector<API<A>> getSons() in API<A>`
  `public Vector<Tree<A>> getSons() in Tree<A>`

- **Sottotipo?**:
  `Vector<Tree<A>>` non è sottotipo di `Vector<API<A>>`.

Osservazione (Tra i due Tipi non intercorre alcuna relazione)

Sia `class<T>` *un tipo polimorfo di nome* `class` *nella variabile generica* `T`. *Siano* `D1 > D2` *due tipi con* `D1` *supertipo di* `D2`. *i tipi* `class<D1>` *e* `class<D2>` *non hanno alcuna relazione nel sistema dei tipi di Java: In particolare* `class<D2>` *non è sottotipo di* `class<D1>`

Errori nei Tipi: Soluzioni

- **Soluzione** Visto che non possiamo avere una relazione tra il tipo nell'interfaccia e quello nell'overriding, l'unica soluzione possibile è usare esattamente lo stesso tipo nelle due signature

- **Signature** Quale signature possiamo usare in entrambe: Questa segnatura deve essere:
  - `esprimibile nell'interfaccia API<A>`
  - `compatibile con il codice scritto in Tree<A>`

Soluzione: Perchè

- **API**: Dichiara un tipo Vector di oggetti sottotipi di `API<A>`.

```
public interface API<A> {
    public A getLabel();
    public Vector<? extends API<A>> getSons();
    public void add(int n, API<A> t);
    public void remove(int n);
}
```

- Calcola valori di tipo contenuto in `Vector<? extends API<A>>`

```
public Vector<? extends API<A>> getSons(){
    Vector<Tree<A>> ret = new Vector<Tree<A>>();//copiare il contenuto
    for (int i=0; i<sons.size(); i++){   //per evitare esposizione rapp.
        ret.add(i,sons.get(i));
    }
    return ret;
}
```