```
/*
    Doppia implementazione: Una classe per non-empty ed una per empty.
    La non empty è estesa per relazioni simmetriche
*/
import java.io.*;
import java.util.*;

interface RelazioneAPI <T1,T2>{ //One Java API for the type Relazione Immutable
    public boolean isIn1 (T1 e);
//  public boolean isIn2 (T2 e);
    public Vector<T2> getAll2(T1 e);
//  public Vector<T1> getAll1(T2 e);
    public RelazioneAPI<T1,T2> add(T1 x, T2 y);
}

class RelazioneADT<T1,T2> implements RelazioneAPI<T1,T2>{
    //only non-empty relations
    private T1 left;
    private T2 right;
    private RelazioneAPI<T1,T2> rem;

    public RelazioneADT(T1 x, T2 y, RelazioneAPI<T1,T2> r){
        left = x; right = y; rem = r;
    }
    public boolean isIn1 (T1 e){
        return (left.equals(e)||rem.isIn1(e));
    }
    public Vector<T2> getAll2(T1 e){
        Vector<T2> r = rem.getAll2(e);
        if (left.equals(e)) r.add(right);
        return r;
    }
    public RelazioneAPI<T1,T2> add(T1 x, T2 y){
        return new RelazioneADT<T1,T2>(x,y,this);
    }
}
class EmptyRelazioneADT<T1,T2> implements RelazioneAPI<T1,T2>{
    //only empty relations
    public boolean isIn1 (T1 e){
        return false;
    }
    public Vector<T2> getAll2(T1 e){
        return new Vector<T2>();
    }
    public RelazioneAPI<T1,T2> add(T1 x, T2 y){
        return new RelazioneADT<T1,T2>(x,y,this);
    }
}

/*
1) Estendere RelazioneADT in un ADT contenente anche l'operazione size che calcola il numer
   coppie (anche duplicate) inserite;
*/

class RelProj<T1,T2> extends RelazioneADT<T1,T2>{
    ...
}
/*
2) Estendere RelazioneADT in un ADT contenente anche l'operazione size che calcola il numer
   coppie differenti contenute nella relazione;
```

```
  -- facile
*/

 class RelProjSingle<T1,T2> extends RelazioneADT<T1,T2>{
    ...
 }

/*
3) Estendere RelazioneADT in un ADT contenente anche l'operazione dom1 che calcola un Vecto
   che fornisce la proiezione, sul primo componente, della relazione.
*/

class RelProjD1<T1,T2> extends RelazioneADT<T1,T2>{
    ...
 }
/*
 3.1) Si dica perchè non occorre estendere la classe EmptyRelazioneADT<T1,T2>
 3.2) Si dica se questa soluzione può essere applicata solo a relazioni non modificabili, o
 potrebbe essere utilizzata per tutte le strutture dinamiche a componenti non modificabili.
 3.3) Si potrebbe usare quest'approccio per relazioni modificabili? E in tal caso risultere
 vantaggioso?

/*
 4) Estendere RelazioneADT (completa delle operazioni isIn2 e getAll1) in un ADT contenente
    un metodo booleano che calcola true se this è una relazione simmetrica.
*/

/*
5) Estendere RelazioneADT in un ADT per relazioni simmetriche.
*/
class SymmRelazione<T> extends RelazioneADT<T,T>{
    ....
}
/*
 6) Siano RelazioneADT1 e RelazioneADT2 due diverse implementazioni dell'API RelazioneAPI n
    modificabile. È possibile costruire una relazione avente elementi costruiti con Relazio
    e altri con RelazioneADT2, im modo del tutto trasparente? (ovvero, indistinguibili per
    li usa). Se si, mostrare un esempio concreto con la relazione (2,3)(4,7).
*/

class Main{
    public static void main(String[] args){
        Integer n1 = new Integer(4);
        Integer n2 = new Integer(5);
        Integer n3 = new Integer(7);
        String s1 = new String("abba");
        String s2 = new String("Rolling");
        String s3 = new String("abba");
        System.out.println(s1==s3);
        System.out.println(s1.equals(s3));
        RelazioneAPI<String,Integer> r1 = new EmptyRelazioneADT<String,Integer>().add(s1,n3
        RelazioneAPI<String,Integer> r2 = r1.add(s1,n2);
        r1 = r2.add(s2,n1);
        System.out.println("il numero di s1 in r1 è: " +r1.getAll2(s1).size());
        r2 = r2.add(s2,n1);
        System.out.println(r1==r2);
        System.out.println(r1.equals(r2));
        //test for ex. 1 ----------------------------------
        //test for ex. 2 ----------------------------------
        //test for ex. 3 ----------------------------------
```

```
        ...
    }
}
```