Correzione Secondo Esonero del 12/6/14

Ivano Salvo
Sapienza Università di Roma
email: salvo@di.uniroma1.it

Anno Accademico 2013-14

Esercizio 1.1

Questo esercizio era praticamente un calcio di rigore, visto che la classe `Coppia` era già nominata nel primo esonero e presente nel codice relativo alle liste polimorfe. Occorreva solo aggiungere qualche metodo.

C'era un'unica insidia sulla definizione del metodo `equals`. Il modo corretto di agire era definire un metodo `equals` con parametro `Object` e verificare a tempo di esecuzione con `instanceof` se l'oggetto passato era di tipo `Coppia`. Come sappiamo, in Java, una classe generica, benchè parametrica rispetto a dei tipi è un'*unica* classe. Quindi occorreva chiedersi se il parametro è istanza di `Coppia<?,?>`. Dopodichè era necessario un downcast per invocare i metodi `fst()` e `snd()`. Anche qui, è necessario fare un downcast usando il tipo `Coppia<?,?>`, speculando che poi sarà il metodo `equals` sui tipi `A` e `B` a trattare correttamente il caso in cui i tipi delle due coppie fossero diversi.

Infine, la soluzione più elegante era quella di scrivere due versioni del metodo come illustrato nel codice qui sotto, ma era possibile anche scrivere un'unica versione (ovviamente con parametro `Object` per fare overriding, come richiesto, del metodo ereditato dalla classe `Object`).

```
class Coppia<A,B>{
    public A a;
    public B b;
    public Coppia(A x, B y){a=x; b=y;}
    public A fst(){return a;}
    public B snd(){return b;}
    public boolean equals(Object o){
        if (! (o instanceof Coppia<?,?>)) return false;
        return equals((Coppia<?,?>) o);
        }
```

```
    public boolean equals(Coppia<?,?> c){
        return a.equals(c.a) && b.equals(c.b);
        }
    public String toString(){
        return "("+a.toString()+", "+ b.toString()+")";
        }
```

Osservazioni ed Errori Tipici

L'errore più tipico è stato non implementare `equals` coi tipi giusti dimodochè faccia effettivamente overriding del metodo `equals` ereditato da `Object`.

Esercizio 1.2

La soluzione corretta di questo esercizio prevedeva di arricchire le liste viste a lezione con un metodo `zip`. La soluzione corretta, prevedeva quindi di scrivere la segnatura di `zip` nell'interfaccia `Lista<A>` e poi fornire l'implementazione nella classe `ListaVuota<A>` e nella classe `ListaNonVuota<A>`. Ecco una possibile soluzione (osservate che occorre "astrarre" sul tipo `B` contenuto nel tipo del risultato ritornato da `zip` e nel tipo del parametro. Altrimenti ci sarebbe una variabile di tipo *libera*):

```
interface Lista<A>{
   ...
   <B>List<Coppia<A,B>> zip(List<B> l);
   ...
   }
class ListaVuota<A> implements Lista<A>{
   ...
   public <B> Lista<Coppia<A,B>> zip(Lista<B> l){
       Lista<Coppia<A,B>> m;
       if (!l.isEmpty()) throw new LunghezzeDifferentiException();
       m = new ListaVuota<Coppia<A,B>>();
       return m;
       }
   ...
   }
class ListaNonVuota<A> implements Lista<A>{
   ...
   public <B>Lista<Coppia<A,B>> zip(List<B> l){
       Lista<Coppia<A,B>> m;
       if (l.isEmpty()) throw new LunghezzeDifferentiException();
       m = new ListaNonVuota<Coppia<A,B>>(new Coppia<A,B>(head,l.head()), tail.zip(l.tail()));
       return m;
       }
   ...
   }
```

Osservazioni ed Errori Tipici

Un discreto numero di studenti ha risolto questo esercizio senza effettivamente fare quanto richiesto, usando delle "liste" con metodi tipo `get(i)` (che restituisce l'ennesimo elemento di una lista). Qualora l'implementazione fosse discreta, ciò ha comunque fato totalizzare loro un congruo numero di punti.

Gli *orrori* più grossolani (non è una svista, ho scritto proprio orrori!) sono stati vedere operazioni tipo `this istanceof ListaVuota` (il tipo dinamico di `this` infatti è sempre uguale alla classe che riceve l'invocazione del metodo) oppure un doppio ciclo for annidato (che genera una lista di lunghezza n^2, posto n la lunghezza delle liste da "zippare").

Esercizio 1.3

Il metodo `f` semplicemente solleva l'eccezione `En` (se il parametro `n` è maggiore di 0) oppure l'eccezione `E0` (se il parametro `n` è 0). il metodo chiamante `g` cattura entrambe le eccezioni. Quando cattura `En` si richiama ricorsivamente e quindi richiama la funzione `f` con `n-1`. Quando cattura l'eccezione `E0`, termina sollevando l'eccezione `Eg` caricandovi il valore della variabile `x`. L'eccezione `Eg` viene catturata dal main che stampa il contenuto dell'eccezione `Eg`. Quindi sostanzialmente, la funzione `g` invocata con parametri $n, 0, 1$ calcola nel parametro `d` i primi n numeri dispari e ne accumula la somma nella variabile `x`. Come dovrebbe essere noto, la somma dei primi n numeri dispari vale n^2.

Il `return` a fine del metodo `g` è necessario perchè JAVA controlla che tutti i metodi che restituiscono un tipo diverso da `void` eseguano come ultima istruzione un return. In realtà il metodo `g` termina sempre in modo "anomalo" sollevando un'eccezione, quindi, di fatto, tale `return` non viene mai eseguito. Quindi, se il compilatore `Java` non fosse così scrupoloso a fare controlli così sofisticati (come i compilatori C, per esempio), il codice prodotto funzionerebbe esattamente come quello ottenuto togliendo il `return`.

Osservazioni ed Errori Tipici

La maggior parte degli studenti non ha individuato la funzione (in funzione del parametro n) calcolata dalla funzione `g`.

Esercizio 2.1

L'esercizio richiedeva di implementare un iteratore per il tipo `Lista` e poi di usarlo per definire un metodo statico `zip` analogo a quello dell'esercizio precedente. Avendo fretta, definisco un iteratore solo per la classe `ListaNonVuota`:

```
public Iterator<A> iterator(){ return new GenLista<A>(this);}

private static class GenLista<A> implements Iterator<A>{
    Lista<A> l;
    public GenList(Lista<A> l){this.l=l;}
    public boolean hasNext(){return !l.isEmpty();}
    public A next(){
       A x = l.head();
       l=l.tail();
       return x;
    }
    public void remove(){}
}
```

Poi il codice della funzione `zip` era qualcosa del tipo:

```
public static <A,B> Lista<Coppia<A,B>> zip(Lista<A> l, Lista<B> m){
    Lista<Coppia<A,B>> res=new EmptyList<Coppia<A,B>>();
    if (l.isEmpty() && m.isEmpty()) return res;
    if (l.isEmpty() || m.isEmpty()) throw new DifferentLengthException();
    Iterator<A> iteL = ((NonEmptyList<A>) l).iterator();
    Iterator<B> iteM = ((NonEmptyList<B>) m).iterator();
    while (iteL.hasNext()) {
       if (!iteM.hasNext()) throw new DifferentLengthException();
       res=res.addLast(new Coppia<A,B>(iteL.next(), iteM.next()));
    }
    if (iteM.hasNext()) throw new DifferentLengthException();
    return res;
}
```

dove `addLast` aggiunge un elemento in coda (altrimenti il risultato viene rovesciato!).