

9

Soluzioni del capitolo 4

Contenuto

9.1 *Dai tipi di dato astratti agli oggetti*

9.2 *Eredità e polimorfismo*

I matematici sono come i francesi: se dici loro qualcosa, lo traducono nella propria lingua e, oplà!, diventa una cosa completamente diversa

Goethe

9.1 Dai tipi di dato astratti agli oggetti

9.1.1 Definire e usare oggetti

Esercizio 4.1 È sufficiente copiare il codice eliminando gli `static` e i parametri `CoppiaInt` dei vari metodi:

```
class OggettoCoppiaInt {
    private int x;
    private int y;

    public OggettoCoppiaInt(int a, int b) {
        x = a;
        y = b;
    }

    public int getPrimo() {
        return x;
    }
    public int getSecondo() {
        return y;
    }
    public void setPrimo(int primo) {
        x = primo;
    }
    public void setSecondo(int secondo) {
```

```

    y = secondo;
  }
}

```

Notate anche che dove nell'approccio TDA era scritto `c.x` ora è scritto semplicemente `x`.

Esercizio 4.2 Un metodo d'istanza non deve essere `static`. Inoltre il numero di parametri in questo caso è zero perché la coppia di cui dobbiamo scambiare le componenti è implicitamente determinata dal fatto di essere quella di cui si invoca il metodo `scambia` (o, come dicono gli esperti di OO, ossia *Object Oriented*, quella a cui si manda il messaggio `scambia`). Se, ad esempio, si scriverà `a.scambia()`, si intenderà scambiare le componenti della coppia `a`, se si scriverà `b.scambia()`, si intenderà scambiare le componenti della coppia `b`, e così via. Il metodo `scambia` è:

```

public void scambia() {
    int temporanea = x;
    x = y;
    y = temporanea;
}

```

Esercizio 4.3

```

class UsoOggettoCoppiaInt {
    public static void main(String[] args) {
        OggettoCoppiaInt a = new OggettoCoppiaInt(Leggi.unInt(),
                                                    Leggi.unInt());
        OggettoCoppiaInt b = new OggettoCoppiaInt(Leggi.unInt(),
                                                    Leggi.unInt());

        System.out.println(a.getPrimo());
        System.out.println(b.getSecondo());
    }
}

```

Nota Osserviamo che con l'approccio TDA avremmo dovuto scrivere `CoppiaInt.getPrimo(a)` invece di `a.getPrimo()`: la notazione a oggetti è più sintetica. Inoltre, l'istruzione `a.getPrimo()` può essere letta come “mando il *messaggio* `getPrimo()` all'oggetto `a`”: è per questo che si parla di *scambio messaggi*.

Esercizio 4.4 Vi sono due errori: il primo è che `getPrimo` è stato definito senza parametri (vedi la soluzione dell'esercizio 4.1) e quindi va invocato senza parametri; il secondo è che `getPrimo` non è un metodo `static` e quindi non può essere invocato come metodo di classe, cioè premettendo il nome della classe, ma deve essere

invocato come metodo d'istanza, cioè premettendo il nome dell'istanza, ad esempio `a.getPrimo()`. ■

Esercizio 4.6

```
public String toString() {
    return "[" + x + "|" + y + "];"
}
```

Esercizio 4.7 `toString` è un metodo particolare che viene invocato automaticamente, o implicitamente, dal metodo `System.out.println` (in realtà, ogni volta che “c'è bisogno di una rappresentazione stampabile dell'istanza). Per questo è sufficiente aggiungere:

```
System.out.println(a);
```

appunto senza nessuna invocazione esplicita di `toString`. ■

Esercizio 4.8 Implementiamo la classe `Portafogli` usando un'unica variabile privata `somma` di tipo `float` che rappresenterà la somma (in euro) contenuta nel portafogli. Scriviamo la classe `Portafogli` un po' alla volta. Cominciamo dalla variabile privata e dal costruttore:

```
public class Portafogli {
    /**Rappresenta il contenuto del portafogli espresso in euro*/
    private float somma;

    public Portafogli() {
        somma = 0;
    }
}
```

Il metodo `aggiungiEuro` avrà un unico parametro di tipo `float` (perché si può aggiungere un numero di euro con la virgola). Quando ad un oggetto `Portafogli` viene mandato un messaggio `aggiungiEuro`, l'unica cosa che deve fare è incrementare la variabile privata della quantità indicata dal parametro del metodo. Il codice sarà quindi:

```
public void aggiungiEuro(float euro) {
    somma += euro;
}
```

Nota Diversamente da quanto facevamo nel caso dei TDA, il metodo *non* è `static`.

Notate che nel caso TDA avremmo dovuto usare due parametri: uno di tipo `Portafogli` per dire a quale portafogli aggiungere euro ed il secondo di tipo `float`. Nell'approccio a oggetti il primo parametro non è più necessario perché implicito. La

riga di codice `somma += euro`; viene interpretata da Java come “aggiungi il contenuto del parametro `euro` alla variabile `somma` di questo oggetto”. Avremmo potuto scrivere esplicitamente questo riferimento così (parleremo più a fondo del `this` nel paragrafo 9.1.2):

```
public void aggiungiEuro(float euro) {
    this.somma += euro;
}
```

Nell’approccio con i TDA avremmo scritto invece:

```
public static void aggiungiEuro(Portafogli p, float euro) {
    p.somma += euro;
}
```

Per quanto riguarda il metodo `aggiungiLire` abbiamo un solo parametro di tipo `int` e potremmo scrivere, copiando la soluzione di prima:

```
public void aggiungiLire(int lire) {
    somma += (lire/1936.27f);
}
```

Notate che in questo caso abbiamo dovuto convertire le lire in euro prima di sommare la cifra alla variabile `somma`. Notate anche che per ottenere la divisione di tipo `float` abbiamo dovuto usare usare il letterale `float 1936.27f` invece di quello `double 1936.27`.

Un altro modo di implementare il metodo `aggiungiLire` ci è suggerito dal fatto che aggiungere tot lire al portafogli è lo stesso di aggiungere `tot/1936.27` euro. Quindi possiamo riscrivere il metodo in questo modo:

```
public void aggiungiLire(int lire) {
    aggiungiEuro(lire/1936.27f);
}
```

Implementare `aggiungiLire` in questo secondo modo ha un pro e un contro. Il contro è che rendiamo il codice leggermente meno efficiente perché obblighiamo Java ad una chiamata di metodo in più che non sarebbe necessaria. Il pro è che tutto il codice (e gli eventuali errori!) sta in un solo metodo e questo rende più facile la manutenzione e le successive modifiche. Se dovessimo operare delle modifiche, basterà farle una sola volta e tutti e due i metodi rimarranno corretti. Con la prima soluzione, invece, ogni volta che si fa una modifica in uno dei metodi si deve stare ben attenti a modificare in modo consistente anche l’altro.

In questo caso la perdita di efficienza è trascurabile ed è sicuramente preferibile la seconda versione. Il codice della classe `Portafogli`, dopo aver anche sostituito l’occorrenza del letterale `1936.27f` con la costante `CAMBIO_LIRA_EURO`, dunque diventa:

```
public class Portafogli {
    /**Rappresenta il contenuto del portafogli espresso in euro*/
```

```
private float somma;
public final static float CAMBIO_LIRA_EURO = 1936.27f;

public Portafogli() {
    somma = 0;
}

public void aggiungiEuro(float euro) {
    somma += euro;
}

public void aggiungiLire(int lire) {
    aggiungiEuro(lire/CAMBIO_LIRA_EURO);
}
}
```

Il metodo `togliEuro` sarà analogo ad `aggiungiEuro`, con l'unica eccezione che la variabile `somma` non può diventare negativa. Possiamo fare in modo che se non ci sono abbastanza soldi venga anche scritto un messaggio di errore:¹

```
public void toglieuro(float euro) {
    if (euro > somma) {
        System.out.println("Non ci sono abbastanza soldi");
        somma = 0;
    } else
        somma -= euro;
}
```

Come abbiamo fatto per `aggiungiLire` possiamo implementare `toglieLire` usando `toglieEuro`:

```
public void toglieLire(int lire) {
    toglieEuro(lire/CAMBIO_LIRA_EURO);
}
```

Grazie alla scelta di mantenere la somma contenuta nel portafogli memorizzata nella variabile `somma`, il metodo `quantiSoldi` diventa molto semplice:

```
public float quantiSoldi() {
    return somma;
}
```

Anche il metodo `alVerde` è semplice: se i soldi sono zero ritorna `true`, altrimenti `false`. Sappiamo che ogni volta che troviamo lo schema

se *condizione* allora ritorna `true` altrimenti ritorna `false`

possiamo sempre sostituirlo con l'istruzione che restituisce la condizione del `se` (`return` "i soldi sono zero" in questo caso):

¹Sarebbe più corretto usare il meccanismo delle eccezioni, che incontreremo più avanti.

```
public boolean alVerde() {  
    return (somma == 0);  
}
```

