Azzolini Riccardo 2018-12-07

`equals`, adombramento, `final` ed eccezioni

1 Il metodo `equals`

Nella classe `Frazione` è definito il metodo `equals`:

```
public boolean equals(Frazione f);
```

Anche la classe `Object`, superclasse di `Frazione`, definisce un metodo `equals`, ma con una segnatura diversa:

```
public boolean equals(Object o);
```

Poiché il tipo dell'argomento dei due metodi è diversi, si ha overloading, e *non* overriding. Di conseguenza, il confronto tra oggetti `Frazione` funziona correttamente solo se entrambi i riferimenti sono di tipo `Frazione` (e non `Object`):

```
Frazione f = new Frazione(1, 2);
Frazione g = new Frazione(1, 2);
Object fo = f;
Object go = g;

f.equals(g);   // true
fo.equals(g);  // false!
f.equals(go);  // false!
fo.equals(go); // false!
```

Per risolvere questo problema, cioè per *specializzare* il metodo equals di `Object` alle frazioni, bisogna ridefinirlo, quindi è necessario adottare la stessa segnatura.

Siccome il parametro deve essere di tipo `Object`, all'interno del metodo si utilizzano un controllo `instanceof` e un cast.

```
public boolean equals(Object altro) {
    if (altro instanceof Frazione) {
        Frazione a = (Frazione) altro;
        return this.num == a.num && this.den == a.den;
    } else {
        return false;
```

```
    }
}
```

Questo metodo può *sostituire* la versione `equals(Frazione f)`. In alternativa, una soluzione più elegante consiste nel mantenere entrambe le versioni e sfruttare `equals(Frazione f)` per implementare `equals(Object altro)`:

```
public boolean equals(Object altro) {
    if (altro instanceof Frazione) {
        return this.equals((Frazione) altro);
    } else {
        return false;
    }
}

public boolean equals(Frazione f) {
    return f != null && this.num == f.num && this.den == f.den;
}
```

Se si scrivono entrambi i metodi, è necessario modificare `equals(Frazione f)`, aggiungendo il controllo `f != null`, in modo che funzioni correttamente anche nell'invocazione `f.equals(null)`: in tal caso, infatti, il compilatore seleziona questa segnatura, dato che è più specifica, e il confronto con `null` deve restituire `false` (senza l'aggiunta del controllo, causerebbe invece un errore).

1.1 Altre osservazioni su `equals`

- È possibile utilizzare, invece di `instanceof`, il metodo `getClass`, il quale restituisce un riferimento a un oggetto speciale che rappresenta una classe.

  ```
  public boolean equals(Object altro) {
      if (altro == null) {
          return false;
      } else if (this.getClass() != altro.getClass()) {
          return false;
      } else {
          Frazione f = (Frazione) altro;
          return this.num == f.num && this.den == f.den;
      }
  }
  ```

  A differenza di `instanceof`, però, il confronto tra i risultati di `getClass` risulta `false` se i due oggetti non sono istanze della stessa identica classe (ad esempio, se uno dei due appartiene a una sottoclasse).

- Se in una classe si riscrive il metodo `equals`, si dovrebbe sempre riscrivere anche `hashCode`.

2 Variabili e adombramento

Una classe e una sua sottoclasse possono avere campi con lo stesso nome:

```
public class Sopra {
    int k = 1;

    public String toString() {
        return String.valueOf(k);
    }
}

public class Sotto extends Sopra {
    int k = 2;

    public String toString() {
        return k + ", " + super.toString();
    }
}
```

In questo caso, le istanze di `Sotto` hanno due campi chiamati `k`:

- quello ereditato da `Sopra`, inizializzato a 1
- e quello dichiarato in `Sotto`, inizializzato a 2

Per determinare quale campo sarà utilizzato da un'istruzione, è sufficiente *osservare il contesto* in cui si trova l'istruzione nel testo della classe:

- l'istruzione `return String.valueOf(k);` del metodo `toString` di `Sopra` utilizza sempre il campo `k` di `Sopra`, anche se eseguito da un'istanza di `Sotto`
- l'istruzione `return k + ", " + super.toString();` nella classe `Sotto` utilizza sempre il campo `k` di `Sotto`

Si dice quindi che, nella classe `Sotto`, la variabile `k` **adombra** la variabile `k` della classe `Sopra`.

Dalla sottoclasse è comunque possibile accedere ai campi della superclasse, purché non siano `private`, mediante il riferimento `super`. Ad esempio, il metodo `toString` di `Sotto` si può riscrivere come:

```java
public String toString() {
    return k + ", " + super.k;
}
```

L'adombramento si può verificare anche in altre situazioni. Ad esempio, in un metodo o un costruttore, un parametro o una variabile locale possono adombrare un campo.

In generale, se esistono più variabili con lo stesso nome, è effettivamente visibile quella che è stata dichiarata "più vicino" all'istruzione che la utilizza.

È sempre possibile accedere ai campi, anche se sono adombrati, utilizzando i riferimenti `this` e `super`.

3 Modificatore di visibilità `protected`

Un membro di una classe `A` dichiarato `protected` è visibile:

- ovunque all'interno del package (come il livello *amichevole*)
- nelle classi esterne al package che estendono (anche indirettamente) `A`

In particolare, all'esterno del package:

- l'unico modo per invocare un costruttore `protected` è utilizzando `super` nei costruttori delle sottoclassi *dirette*
- i campi e i metodi `protected` di una classe `A` sono accessibili solo nel corpo delle classi `B` che la estendono (anche indirettamente) ed esclusivamente tramite riferimenti di tipo non superiore a `B` (quindi neanche di tipo `A`)

Un metodo `protected` di una superclasse può essere ridefinito `protected` o `public`.

4 Modificatore `final`

`final` si può applicare a variabili, metodi e classi, con significati diversi, ma in generale stabilisce che l'identificatore a cui è applicato non può essere modificato.

4.1 Variabili `final`

A una variabile `final` è possibile assegnare un valore una sola volta, altrimenti si ha un errore in fase di compilazione. In altre parole, è una costante. Convenzionalmente, per le costanti si usano nomi in maiuscolo.

Nel caso delle variabili riferimento, è appunti il riferimento a essere immodificabile, quindi è possibile modificare l'oggetto riferito ma non assegnarne un altro.

`final` si può applicare a tutti i tipi di variabile: campi (anche statici), variabili locali e parametri.

- Per le variabili locali, il valore può essere assegnato in fase di dichiarazione oppure in seguito:

  ```
  final int X;
  final int Y = 2;
  X = Y + 1;
  ```

- Per i parametri, il valore viene assegnato al momento dell'invocazione del metodo/costruttore.
- Per i campi statici, il valore può essere assegnato solo in fase di dichiarazione.
- I campi non statici possono essere inizializzati esplicitamente in fase di dichiarazione, altrimenti si dicono *blank-final* e devono obbligatoriamente essere inizializzati all'interno di tutti i costruttori della classe (altrimenti il compilatore segnala un errore).

4.2 Metodi `final`

Un metodo `final` non può essere ridefinito nelle sottoclassi. Ciò e utile principalmente per impedire alle sottoclassi di cambiarne il significato, potenzialmente violandone il contratto.

Inoltre, i metodi `final` hanno anche una maggiore efficienza (dato che il compilatore li può trattare in modo diverso).

4.3 Classi `final`

Una classe `final` non può essere estesa.

Come nel caso dei metodi, dichiarare una classe `final` può essere utile per motivi di correttezza ed efficienza.

5 Definizione di un'eccezione

Per definire una nuova eccezione bisogna estendere `Exception` o una delle sue sottoclassi. La classe da estendere deve essere scelta con cura: in particolare, è necessario decidere con attenzione se definire un'eccezione *controllata* o *non controllata*.

In genere, i comportamenti delle istanze di un'eccezione sono molto semplici: spesso è sufficiente poter associare all'oggetto un messaggio, che verrà poi visualizzato dal metodo `toString`. Tale funzionalità si può implementare sfruttando il costruttore della superclasse.

Gli altri comportamenti, compreso il metodo `toString`, sono ereditati, quindi non è necessario definirli.

5.1 Esempio: `FrazioneException`

```
public class FrazioneException extends ArithmeticException {
    public FrazioneException(String msg) {
        super(msg);
    }
}
```

6 Ridefinizione di metodi ed eccezioni

Quando, in una sottoclasse, si ridefinisce un metodo che dichiara di delegare alcune eccezioni controllate, l'insieme di eccezioni delegate dal metodo della sottoclasse non può essere più ampio di quello del metodo della superclasse.

Se il metodo della superclasse dichiara di delegare le eccezioni `E1, ..., En`, allora il metodo della sottoclasse può:

- non delegare alcuna eccezione
- delegare eccezioni di alcuni dei tipi `E1, ..., En` (anche tutti) o di dei loro sottotipi

In questo modo, un blocco `try-catch` che invoca il metodo tramite un riferimento della superclasse è in grado di gestire eventuali eccezioni anche se, in fase di esecuzione, l'oggetto riferito è istanza di una sottoclasse. È comunque possibile aggiungere blocchi `catch` specifici per le eccezioni dei metodi ridefiniti: in questo caso, il blocco eseguito potrà quindi dipendere dal tipo dell'oggetto.

Questa regola vale anche per l'implementazione dei metodi astratti (sia delle classi astratte che delle interfacce), che come quelli concreti possono dichiarare di delegare delle eccezioni.