Lezione 13

Classi interne

Questa lezione chiude la parte del corso più squisitamente dedicata alla programmazione orientata agli oggetti con Java. Sarà esaminato un nuovo tipo di classi, finora taciuto, ma non per questo meno proficuo o importante: le classi interne (*inner-class*). Sostanzialmente, le classi interne sono classi definite all'interno del corpo di altre classi.

13.1 - Classi interne

Le classi interne si dividono in quattro differenti gruppi:

- Classi (ed interfacce) statiche innestate.
- Classi membro.
- Classi locali.
- Classi anonime.

I prossimi paragrafi prenderanno in esame le quattro tipologie elencate.

13.2 - Classi ed interfacce statiche innestate

Le classi e le interfacce statiche innestate sono definite all'interno del corpo di una classe o di un'interfaccia di tipo comune, precedute da uno specificatore d'accesso (opzionale) e dalla parola chiave *static*. Qualcosa come:

```
public class Persona {

  public static class NomeAnagrafico {

    private String nome;
    private String cognome;

    public NomeAnagrafico(String n, String c) {
      nome = n;
      cognome = c;
    }

    public String getNome() {
      return nome;
    }

    public String getCognome() {
      return cognome;
    }

  }

  private NomeAnagrafico nomeAnagrafico;

  public Persona(String n, String c) {
    nomeAnagrafico = new NomeAnagrafico(n, c);
  }

  public NomeAnagrafico getNomeAnagrafico() {
    return nomeAnagrafico;
  }

}
```

La classe *NomeAnagrafico* è interna, definita nel corpo della classe di primo livello Persona. La struttura è usata per mantenere salda la coppia di dati *nome* e *cognome*. La compilazione del sorgente *Persona.java* porta alla nascita di due distinti bytecode: *Persona.class*, come di consueto, e *Persona$NomeAnagrafico.class*. La norma che fa corrispondere un bytecode ad ogni classe o interfaccia, dunque, non è infranta. In ogni caso, *NomeAnagrafico* è una classe speciale, poiché statica ed interna a *Persona*.
Il codice presente nel corpo della classe *Persona* può creare un oggetto di tipo *NomeAnagrafico* come di consueto:

```
NomeAnagrafico na = new NomeAnagrafico(...);
```

Dall'esterno di *Persona*, tuttavia, le cose sono differenti. Lo dimostra il seguente test:

```
public class Test {

  public static void main(String[] args) {
    Persona.NomeAnagrafico na = new Persona.NomeAnagrafico("Mario", "Rossi");
    System.out.println(na.getNome());
    System.out.println(na.getCognome());
  }

}
```

Il nome completo della classe interna escogitata è:

```
Persona.NomeAnagrafico
```

Non è detto che si possa sempre accedere alle classi interne di questo tipo, stando all'esterno della struttura che le definisce. Come le proprietà ed i metodi, infatti, le classi statiche innestate possono fare uso dei quattro specificatori d'accesso validi in Java. Dunque, una classe statica innestata può essere:

- *public*, quindi visibile ovunque, mediante la notazione:

  ```
  ClasseContenitrice.ClasseInterna
  ```

- *private*, quindi accessibile solo dal codice della classe che la contiene.
- *protected*, quindi accessibile dal codice della classe che la contiene e delle sue eventuali classi figlie, oltre che dal codice delle classi riposte nello stesso pacchetto.
- priva di specificatore, quindi disponibile solo all'interno del pacchetto che ospita la classe che la contiene.

Le classi interne del tipo in esame soffrono delle stesse limitazioni dei metodi statici: non possono invocare membri non statici della classe che li contiene.
I discorsi svolti sinora nei riguardi delle classi innestate statiche valgono anche nei confronti delle interfacce: usando la parola *interface* è possibile avere delle interfacce innestate statiche.

13.3 - Classi membro

Le classi membro differiscono dalle innestate statiche per l'assenza della parola *static*.

Dunque, sono membri non statici della classe che le contiene. Ecco un esempio:

```
public class Persona {

  public class NomeAnagrafico {

    private String nome;
    private String cognome;

    public NomeAnagrafico(String n, String c) {
      nome = n;
      cognome = c;
    }

    public String getNome() {
      return nome;
    }

    public String getCognome() {
      return cognome;
    }

  }

  private NomeAnagrafico nomeAnagrafico;

  public Persona(String n, String c) {
    nomeAnagrafico = new NomeAnagrafico(n, c);
  }

  public NomeAnagrafico getNomeAnagrafico() {
    return nomeAnagrafico;
  }

}
```

Una classe membro, essendo non statica, non è associata alla classe che la contiene, ma alle sue istanze. In parole semplici, non è più possibile fare:

```
Persona.NomeAnagrafico na = new Persona.NomeAnagrafico("Mario", "Rossi");
```

In tal caso si otterrebbe un errore del tipo:

```
not an enclosing class: Persona
    Persona.NomeAnagrafico na = new Persona.NomeAnagrafico("Mario", "Rossi");
                                ^
```

Tuttavia, una classe membro può essere restituita ad un codice esterno, purché il suo specificatore d'accesso ne decreti la possibilità:

```
public class Test {

  public static void main(String[] args) {
    Persona p = new Persona("Mario", "Rossi");
    Persona.NomeAnagrafico na = p.getNomeAnagrafico();
    System.out.println(na.getNome());
    System.out.println(na.getCognome());
  }
```

```
}
```

Le classi membro non differiscono dai comuni metodi non statici. Grazie alla loro dipendenza da un'istanza realmente esistente, possono appellarsi a tutti i membri dell'oggetto di appartenenza, compresi quelli non statici.
Non esistono interfacce membro: un'interfaccia interna è sempre ed automaticamente statica, anche quando si omette la parola chiave *static*. Un'interfaccia, infatti, non contiene codice eseguibile, e pertanto non avrebbe comunque modo di richiamare altri membri della classe che la contiene, siano essi statici oppure no. Non c'è, quindi, necessità di distinzione.
Le classi membro non possono definire proprietà, metodi o ulteriori classi interne di tipo statico. Si ritiene che i membri statici debbano sempre apparire al livello superiore di visibilità, direttamente nella classe contenitrice o eventualmente all'interno di classi statiche innestate. Si osservi che le interfacce interne, come detto poche righe sopra, sono sempre ed automaticamente innestate statiche. Quindi, stando all'ultima norma esaminata, le classi membro non possono definire interfacce interne.

13.4 - Classi locali

Le classi locali sono classi definite all'interno di un blocco di codice Java eseguibile. Sono strutture valide localmente, all'interno dell'ambito che le definisce, proprio come le variabili:

```
public class Test {

  public static void main(String[] args) {

    class Rettangolo {

       private double width;
       private double height;

       public Rettangolo(double w, double h) {
         width = w;
         height = h;
       }

       public double getWidth() {
         return width;
       }

       public double getHeight() {
         return height;
       }

    }

    Rettangolo r = new Rettangolo(7.5, 3.2);
    System.out.println(r.getWidth());
    System.out.println(r.getHeight());

  }

}
```

La classe *Rettangolo*, nell'esempio appena mostrato, è una struttura ausiliaria dichiarata all'interno del blocco eseguibile associato al metodo *main()* della classe *Test*. Può essere sfruttata come una classe qualsiasi, ma solo all'interno del suo blocco di appartenenza. Al

di fuori di esso, la struttura è inesistente. Naturalmente, non c'è alcuna utilità nell'associare uno specificatore di accesso a questa classe: il suo utilizzo sarà comunque privato e ristretto al solo ambito di appartenenza.
Una classe locale può richiamare ogni membro della sua classe contenitrice, come avviene con le classi membro. In più, una classe locale può fare uso delle variabili valide nel suo stesso blocco di appartenenza, purché esse siano *final*:

```
public class Test {

  public static void main(String[] args) {

    final int i = 5;

    class ClasseLocale {

      public ClasseLocale() {
        System.out.println(i);
      }

    }

    ClasseLocale l = new ClasseLocale();

  }

}
```

Omettendo la parola *final* davanti alla variabile locale *i*, si ottiene un errore di compilazione:

```
local variable i is accessed from within inner class; needs to be declared final
        System.out.println(i);
                           ^
```

Le classi locali non possono dichiarare membri statici, secondo le stesse norme valide per le classi membro.

13.5 - Classi anonime

Una classe anonima è una classe locale priva di nome. Le classi anonime rappresentano una scorciatoia per dichiarare e contemporaneamente istanziare una classe che dovrà essere usata per generare un solo oggetto. Le classi anonime usano il concetto di ereditarietà. Si può creare una classe anonima solo estendendo un'altra classe, ridefinendone uno o più metodi:

```
// Animale.java
public class Animale {

  public void mangia() {
    System.out.println("GNAM!");
  }

  public void faiVerso() {
    System.out.println("BOH!");
  }

}
```

```
// Test.java
public class Test {

  public static void main(String[] args) {
    Animale a = new Animale() {
      public void faiVerso() {
        System.out.println("MIAO!");
      }
    };
    a.mangia();
    a.faiVerso();
  }

}
```

L'istruzione

```
Animale a = new Animale() {
  public void faiVerso() {
    System.out.println("MIAO!");
  }
};
```

dichiara e istanzia una classe anonima. Questa sintassi abbreviata esegue i seguenti passaggi:

- Crea una classe locale, priva di nome, che estende *Animale*.
- All'interno di tale classe locale viene ridefinito il metodo *faiVerso()*.
- La classe locale appena creata viene istanziata.
- Il riferimento all'oggetto creato viene mantenuto in una variabile del tipo della superclasse *Animale*.

Teoricamente, è possibile dotare una classe anonima di nuovi membri che non siano pura ridefinizione di quelli ereditati. Tuttavia, la pratica non è conveniente, giacché il riferimento restituito, che è del tipo associato alla superclasse, non permetterebbe di richiamarli. Siccome le classi anonime non hanno nome, non è possibile ottenere dei riferimenti del loro esatto tipo. Ogni loro membro che non sia una ridefinizione, pertanto, rimarrà invisibile.

Anziché estendere una superclasse, sia essa astratta o meno, una classe anonima può implementare un'interfaccia, dando definizione ad ogni metodo da essa richiesto:

```
// Animale.java
public interface Animale {

  public void faiVerso();

}

// Test.java
public class Test {

  public static void main(String[] args) {
    Animale a = new Animale() {
      public void faiVerso() {
        System.out.println("MIAO!");
```

```
        }
      };
      a.faiVerso();
    }

}
```

Le classi anonime soffrono le stesse limitazioni delle classi locali: non possono fare uso di membri statici, siano essi proprietà, metodi o ulteriori classi interne. Inoltre, non possono definire interfacce interne, poiché queste ultime sono sempre ed automaticamente innestate statiche. Infine, le classi anonime possono usare le variabili locali valide nel loro medesimo ambito di visibilità, purché queste siano *final*.