

Programmazione III

Classi Inestate (e classi locali)

Liliana Ardissono

Classi e interfacce innestate

Le classi possono essere dichiarate:

- **all'interno di altre classi in qualità di membri (classi interne: possono essere statiche o di istanza)**
- **all'interno di blocchi di codice (classi interne locali).**

La definizione di tipi innestati è utile per:

- Definire tipi strutturati e resi visibili all'interno di gruppi correlati logicamente
- Connettere in modo semplice e efficace oggetti correlati esplicitamente: un tipo innestato è considerato parte del tipo in cui è racchiuso e con il quale condivide una relazione di fiducia per cui ognuno dei due può accedere a tutti i membri dell'altro (anche a quelli privati)

Classi interne – Inner classes

(tipi innestati non statici)

Una classe innestata, dichiarata membro della classe che la racchiude, si comporta come una qualsiasi classe ma il suo nome e grado di accessibilità dipendono dalla classe contenitore:

- **La visibilità della classe innestata è per default la stessa di quella del contenitore** (a meno che la classe inner venga dichiarata *private*, nel qual caso è solo visibile all'interno del contenitore)
- **Il nome della classe innestata è così composto:**
NomeContenitore.NomeClasseInnestata

Classi interne – Inner class: esempio (classe di istanza) - I

```
class ClasseEsterna {
    private int val;
    private ClasseInterna ci;

    public ClasseEsterna(int v) {
        val = v; ci = new ClasseInterna(v); }

    public int getVal() { return val; }

    public ClasseInterna getCi() { return ci; }

    public void setVal(int val) {
        this.val = val;
        ci.valInterno = val; // la classe esterna accede ai componenti dell'interna
    }
    class ClasseInterna {
        private int valInterno;

        public ClasseInterna(int v) { valInterno = v; }

        public int getValInterno() { return valInterno; }
    }
}
```

Classi interne – Inner class: esempio (classe di istanza) - II

```
public class Test1 {
```

```
    public static void main(String[] args) {
```

```
        ClasseEsterna ce = new ClasseEsterna(10);
```

```
        System.out.println("Valore del dato di oggetto ClasseEsterna: " + ce.getVal());
```

```
        ClasseEsterna.ClasseInterna ci = ce.getCi();
```

```
        System.out.println("Valore del dato di oggetto ClasseInterna: "
                               + ci.getValInterno());
```

```
        ce.setVal(30);
```

```
        System.out.println("Valore del dato di oggetto ClasseInterna " +
                               "dopo la modifica "+ ci.getValInterno());
```

```
    }
```

```
}
```

Output - Progr3NestedClasses (run) X



run:



Valore del dato di oggetto ClasseEsterna: 10



Valore del dato di oggetto ClasseInterna: 10



Valore del dato di oggetto ClasseInterna dopo la modifica 30

BUILD SUCCESSFUL (total time: 0 seconds)

Classi interne – Inner classes - outerThis

Gli oggetti di una classe interna hanno un riferimento *outerThis* agli oggetti contenitori. Quindi essi possono accedere alle variabili di istanza dei contenitori, facendovi riferimento come se fossero locali. Es:

```
class Esterna {  
    private int val; private Interna ci;  
    public Esterna(int v) {  
        val = v; ci = new Interna(); }  
    ... Getter, setter, etc.  
  
    class Interna {  
        private int valInterno;  
        public Interna() {  
            valInterno = val; } // legge il valore di val definito nella classe esterna  
        public int getValInterno() { return valInterno; }  
    }  
}
```

Esempio: BankAccount - I

Nella classe BankAccount, definita nel prossimo lucido, la classe Operation non ha motivo di essere dichiarata come normale classe (serve solo in BankAccount) → può essere una classe innestata: il vantaggio è che BankAccount può accedere direttamente a tutti i membri di Operation e viceversa, anche a quelli privati.

Inoltre definiamo Operation come classe inner *private* (interna locale) per cui visibile solo all'interno di BankAccount. In questo modo non esportiamo la struttura dati delle Operation sui conti correnti (incapsulamento)

Esempio: BankAccount - II

```
public class BankAccount {  
    private long number;  
    private long balance;  
    private Vector<Operation> history;  
    private class Operation {  
        private String op;  
        private long amount;  
        private Operation(String o, long q) { op = o; amount = q; }  
        public String toString() {  
            return number + ": " + op + " " + amount; }  
         } // end Operation  
    public BankAccount(int n) {  
        number = n; history = new Vector<Operation>();  
    }  
    public void deposit(Person p, long amount) {  
        Operation op = new Operation("deposit", amount);  
        balance = balance+amount;  
        history.add(op);  
    }  
}
```


Tipi innestati statici (o classi interne statiche)

Se la classe innestata non ha bisogno di far riferimento ai membri della classe contenitrice ma si vogliono correlare i tipi (e magari rendere privata la dichiarazione della classe innestata) si può creare una classe interna statica.

Le classi statiche **non** mantengono il riferimento *outerThis*.

Es. in BankAccount la lista dei permessi forniti ai clienti potrebbe essere definita da una classe interna statica Permissions che specifica, per ogni persona, i diritti (deposito, prelievamento, etc.).

Esempio: BankAccount - III

```
public class BankAccount {  
    private long number;  
    private long balance;  
    private Vector<Permissions> grantedPerms;  
    private static class Permissions {  
        private Person pers;  
        private boolean canDeposit;  
        private boolean canWithdraw;  
        private Permissions(Person p, boolean d, boolean w) {  
            pers = p; canDeposit = d; canWithdraw = w;  
        }  
    }  
    public BankAccount(int n) {  
        number = n; grantedPerms = new Vector<Permissions>();  
    }  
    public void grantPermissions(Person p, boolean d, boolean w) {  
        Permissions perm = new Permissions(p, d, w);  
        grantedPerms.add(perm);  
    }  
}
```

Per leggibilità
ometto Operation

Classi innestate in interfacce

Se noi vogliamo definire una Interface I corredata di implementazione di default possiamo innestare l'implementazione nella Interface come classe interna statica.

Le classi che implementano I possono

- estendere l'implementazione di default (→ usare direttamente l'implementazione di default), oppure
- modificare l'implementazione di default, eventualmente riutilizzandone delle parti.

Classi innestate in interfacce – esempio - I

```
public interface Message {  
    public String getText();  
    public String getDest();
```

```
    static class MsgImpl {  
        protected int destinatario;  
        protected String txt;
```

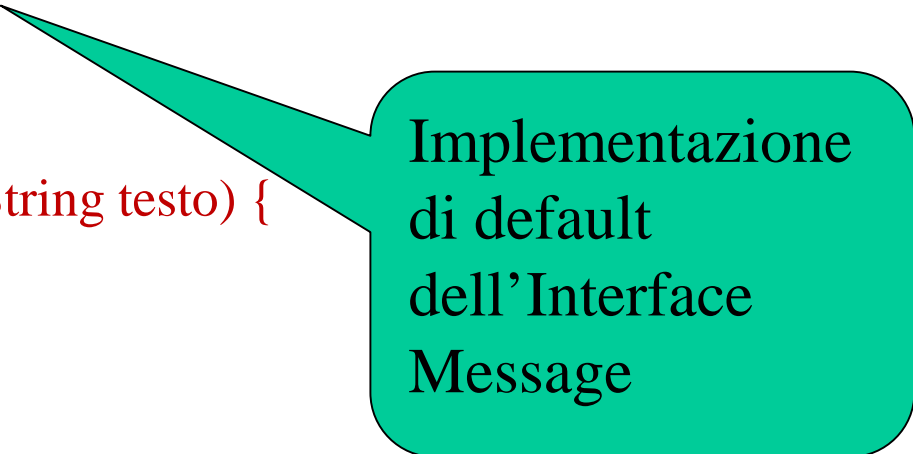
```
        public MsgImpl(int destinatario, String testo) {  
            this.destinatario = destinatario;  
            txt = testo;  
        }
```

```
        public String getText(){  
            return txt;  
        }
```

```
        public String getDest() {  
            return Integer.toString(destinatario);  
        }
```

```
    }
```

```
}
```



Implementazione
di default
dell'Interface
Message

Classi innestate in interfacce – esempio - II

```
public class SMSusaDefaultImpl extends Message.MsgImpl  
                                implements Message {  
  
    public SMSusaDefaultImpl(int destinatario, String testo) {  
        super(destinatario, testo);  
    }  
}
```

Usa implementaz di default
→ non fornisce una sua
implementazione
dell'Interface Message

Classi innestate in interfacce – esempio - III

```
public class SMSusaComponentsImpl implements Message {  
    private Message.MsgImpl msg;
```

```
    public SMSusaComponentsImpl(int dest, String txt) {  
        msg = new Message.MsgImpl(dest, txt);  
    }
```

```
    public String getText() {  
        return ">> " + msg.getText();  
    }
```

```
    public String getDest() {  
        return msg.getDest();  
    }
```

```
}
```

Incapsula un MsgImpl e usa i suoi metodi per reimplementare l'interface Message → potrei aggiungere istruzioni ai metodi per modificare il comportamento rispetto a quello di default di MsgImpl

Classi e interfacce innestate anonime

Se non serve dare un nome alle classi innestate (perché usate in un solo punto del codice della classe contenitrice) le si può definire come anonime, per compattezza. Però *per questioni di leggibilità si consiglia di definire classi anonime solo se hanno poche linee di codice.*

Vediamo come esempio **un'implementazione dell'interfaccia `Iterator`** che restituisce un iteratore su una collezione (qui implementata come array di `Object`).

NB: la classe interna è qui definita all'interno di un metodo

Classi e interfacce innestate – non anonime

```
interface Iteratore {  
    boolean hasNext();  
    Object next();  
}  
  
class Collezione {  
    private Object[] array;  
    public Collezione(Object[] elenco) { array = elenco; }  
    public Iteratore getIteratore() {  
        class Iter implements Iteratore {  
            private int pos = 0;  
            public boolean hasNext() { return (pos < array.length); }  
            public Object next() {  
                if (pos < array.length) {  
                    pos++;  
                    return array[pos - 1];  
                } else return null;  
            } } // end Iter  
        return new Iter();  
    }  
} // end Collezione
```

La definizione della class Iter, con nome, è verbosa (la classe non è usata altrove). Ma poiché restituisco un Iter, non può neppure essere una classe privata

Classi e interfacce innestate - anonime

```
class Collezione {  
    private Object[] array;  
    public Collezione(Object[] elenco) { array = elenco; }  
    public Iteratore getIteratore() {  
        return new Iteratore() {  
            private int pos = 0;  
            public boolean hasNext() {  
                return (pos < array.length); }  
            public Object next() {  
                if (pos < array.length) {  
                    pos++;  
                    return array[pos - 1];  
                } else return null;  
            } };  
    } } // end Collezione
```

Più sintetico del precedente.
NB: il risultato deve essere di tipo Iteratore perché non c'è un nome di classe da usare nel return