



Programmazione III

Prof.ssa Liliana Ardissono
Dipartimento di Informatica
Università di Torino

Classi innestate (e classi locali), Lambda expressions



Questa presentazione è distribuita sotto licenza Creative Commons CC BY ND

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 (variabili, metodi) dell'altro (anche a quelli privati)
- **Information hiding di tipi di dati** (classi all'interno di altre)

Classi interne – Inner class: esempio

```
class ClasseEsterna {
```

(classe di istanza) - I

```
    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; }    // ci serve per sperimentare...
```

```
    public void setVal(int val) {                  // anche se viola information hiding  
        this.val = val;  
        ci.valInterno = val;    // la classe esterna accede ai  
                                // componenti della classe interna  
    }
```

```
class ClasseInterna {  
    private int valInterno;  
  
    public ClasseInterna(int v) { valInterno = v; }  
    public int getValInterno() { return valInterno; }  
}  
}
```



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 il suo 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

```
class ClasseEsterna {
```

(classe di istanza) - I

```
    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; } // ci serve per sperimentare...
```

```
    public void setVal(int val) { // anche se viola information hiding  
        this.val = val;  
        ci.valInterno = val; // la classe esterna accede ai  
                               // componenti della classe interna
```

```
    }  
    class ClasseInterna {  
        private int valInterno;
```

```
        public ClasseInterna(int v) { valInterno = v; }  
        public int getValInterno() { return valInterno; }
```

```
    }  
}
```



Classi interne–Inner class: es. (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:

```
public class Test2 {  
    class Esterna {  
        private int val; 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;}
    }
    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, prelevamento, 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);  
    }  
}
```

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 {      // Implementazione di default di Message  
        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);  
        }  
    }  
}
```

Classi innestate in interfacce – esempio - II



```
public class SMSusaDefaultImpl extends Message.MsgImpl  
    implements Message {
```

```
    public SMSusaDefaultImpl(int destinatario, String testo) {  
        super(destinatario, testo);  
    }  
}
```

SMSusaDefaultImpl usa l'implementazione di default di Message → 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(); }
```

SMSusaComponentsImpl incapsula un **MsgImpl** e usa i suoi metodi per reimplementare l'interface **Message** → io 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 Iteratore** 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;  
            }  
        }  
        return new Iter(); }  
    }
```



La definizione di Iter, con nome, è verbosa (la classe non è usata altrove). Ma poiché restituisco un Iter, non può 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;  
            } };  
    }  
}
```

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

Lambda expressions



Le lambda expressions sono utili per implementare interface che offrono un solo metodo → non si vuole usare la sintassi verbosa delle classi anonime.

- Possono avere/non avere parametri;
 - Possono restituire un risultato o essere di tipo void;
- Tutto dipende dalla specifica del metodo nell'interface.

Esempio: consideriamo l'Interface Comparator:

```
public interface Comparator<T>
```

che offre il metodo


```
public int compare(T o1, T o2)
```


Esempio con classe anonima innestata



```
import java.util.Comparator;
public class EsempioSenzaLambda {
    public static void main(String[] args) {
        // con uso di classe anonima innestata
        Comparator<String> c = new Comparator<String>() {
            public int compare(String s1, String s2) {
                return s1.compareTo(s2);
            }
        };
        int ris = c.compare("ciao", "ciao");
        System.out.println(ris);
        System.out.println(c.compare("ciao1", "ciao2"));
    }
}
```

Esempio con uso di lambda expression – v0

```
import java.util.Comparator;
public class EsempioLambda0 {
    public static void main(String[] args) {
         Comparator<String> c = // con lambda, versione base
                           (String s1, String s2) -> {return s1.compareTo(s2);};
        int ris = c.compare("ciao", "ciao");
        System.out.println(ris);
        System.out.println(c.compare("ciao1", "ciao2"));
    }
}
```




- lo ho eliminato la **new Comparator<String>** perché c ha tale tipo, quindi si inferisce automaticamente
- lo ho eliminato la **definizione del metodo compare()**, lascio solo il body

Notate la compattezza della definizione rispetto a usare la classe anonima innestata

Esempio con uso di lambda expression – v1

```
public class EsempioLambda1 {  
    public static void main(String[] args) {  
        // senza tipo dei parametri del metodo  
        Comparator<String> c =  
            (s1, s2) -> {return s1.compareTo(s2); };  
        int ris = c.compare("ciao", "ciao");  
        System.out.println(ris);  
        System.out.println(c.compare("ciao1", "ciao2"));  
    }  
}
```



lo posso omettere il tipo dei parametri del metodo perché è definito nell'interface Comparator e stretto dal tipo di c

Esempio con uso di lambda expression – v2

```
public class EsempioLambda2 {  
    public static void main(String[] args) {  
        // senza tipo dei parametri  
        Comparator<String> c =  
            // con body semplificato  
            (s1, s2) -> s1.compareTo(s2);  
        int ris = c.compare("ciao", "ciao");  
        System.out.println(ris);  
        System.out.println(c.compare("ciao1", "ciao2"));  
    }  
}
```



- Solo se il body del metodo implementato ha una sola istruzione e restituisce un valore (non è void) io posso omettere {} e anche il return
- NB: questo è sbagliato: (s1, s2) -> return 1;

Esempio con uso di lambda expression – v3



```
interface Prova {  
    public void stampa();  
}
```



```
public class EsempioLambda3 {  
    public static void main(String[] args) {  
        Prova p = () -> {System.out.println("CIAO");};  
        p.stampa();  
    }  
}
```

- Come già specificato, questa lambda expression deve avere {} perché non restituisce un valore (non è una funzione)
- Se il metodo dell'interface non ha parametri io devo comunque specificare () nella lambda expression

Esempio con uso di lambda expression – v4

```
interface Prova {  
    public void stampa(String s);  
}
```



```
public class EsempioLambda4 {  
    public static void main(String[] args) {  
        Prova p =  
            messaggio -> {System.out.println(messaggio);};  
        p.stampa("CIAO!!");  
    }  
}
```

- Se il metodo dell'interface ha un solo parametro io posso omettere le () della dichiarazione del parametro
- NB: se il metodo ha più di un parametro le () della lista dei parametri non possono essere omesse

Le lambda expressions sono oggetti → posso passarle come parametri



```
interface Prova {  
    public void stampa(String s);  
}  
  
public class EsempioLambda5 {  
    public static void main(String[] args) {  
        Prova p = messaggio -> {System.out.println(messaggio);};  
        MyClass m = new MyClass();  
        m.metodo(p);  
    }  
}  
  
class MyClass {  
    void metodo(Prova p) {  
        System.out.println("Sto per richiamare il metodo stampa");  
        p.stampa("Metodo invocato!");  
    }  
}
```

