

Linguaggi

Corso M-Z - Laurea in Ingegneria Informatica
A.A. 2007-2008

Alessandro Longheu

<http://www.diit.unict.it/users/alongheu>

alessandro.longheu@diit.unict.it

- lezione 08 -

Classi astratte ed Interfacce in Java

1

A. Longheu – Linguaggi M-Z – Ing. Inf. 2007-2008

Classi astratte

- L'ereditarietà porta a riflettere sul rapporto fra progetto e struttura:
- Una classe può limitarsi a definire solo l'interfaccia, lasciando "in bianco" uno o più metodi, ...che verranno poi implementati dalle classi derivate
- Questa è una classe astratta
- Una classe astratta fattorizza, dichiarandole, operazioni comuni a tutte le sue sottoclassi, ma non le definisce (implementa)
- In effetti, non viene creata per definire istanze (che non saprebbero come rispondere ai metodi "lasciati in bianco"), ma per derivarne altre classi, che dettaglieranno i metodi qui solo dichiarati.

2

Classi astratte

- Una classe astratta e' simile a una classe regolare: può avere attributi (tipi primitivi, classi (static), istanze di oggetti), può avere metodi, è caratterizzata dalla parola chiave `abstract`, ed ha tuttavia solitamente almeno un metodo che è dichiarato ma non implementato (modificatore `abstract`)
- Una classe astratta può quindi definire dei metodi la cui implementazione è demandata alle sottoclassi; impone quindi l'esistenza del metodo in una classe derivata
- Moltissime entità che usiamo per descrivere il mondo non sono reali sono pure categorie concettuali, ma sono comunque molto utili!
- Una classe astratta può anche non avere metodi astratti; in tal caso è definita astratta per non essere implementata, e costituire semplicemente una categoria concettuale.
- Se comunque almeno un metodo è `abstract`, la classe deve esserlo anche lei, pena errore di compilazione

3

Classi astratte - Esempio

- Esempio: gli animali


```
public abstract class Animale {
    public abstract String verso();
    public abstract String si_muove();
    public abstract String vive();
    ...
}
```
- parlare di "animali" ci è molto utile, però non esiste "il generico animale", nella realtà esistono solo animali specifici; la categoria concettuale "animale" tuttavia ci fa molto comodo per "parlare del mondo", e per fattorizzare gli aspetti comuni
- Una classe che estende una astratta può fornire tutte le implementazioni necessarie; qualora ciò non avvenga, resta astratta anche se i suoi metodi non sono `abstract`, ad esempio:


```
public abstract class AnimaleTerrestre extends Animale {
    public String vive() { // era abstract
        return "sulla terraferma"; }
}
```

4

Classi astratte - Esempio

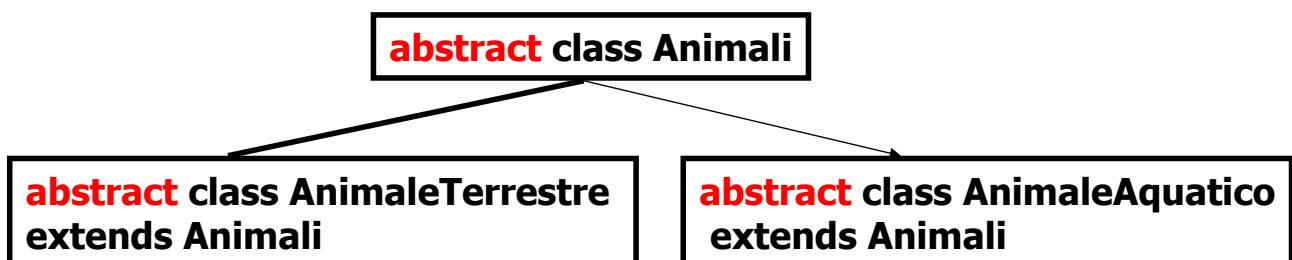
- Ipotesi:
 - ogni animale ha un metodo `chi_sei()` che restituisce una stringa descrittiva
 - ogni animale ha un metodo `mostra()` che lo stampa a video e che è indipendente dallo specifico animale (si appoggia sugli altri metodi)
 - tutti gli animali si rappresentano nello stesso modo, tramite il loro nome e il verso che fanno.

```
public abstract class Animale {
    private String nome;
    protected String verso;
    public Animale(String s) { nome=s; }
    public abstract String si_muove();
    public abstract String vive();
    public abstract String chi_sei();
    public void mostra() { System.out.println(nome + ", " +
        chi_sei() + ", " + verso + ", si muove " +
        si_muove() + " e vive " + vive() ); }
}
```

5

Classi astratte - Esempio

Una possibile classificazione:



Sono ancora classi astratte:

- nulla si sa del movimento
- quindi è impossibile definire il metodo `si_muove()`

6

Classi astratte - Esempio

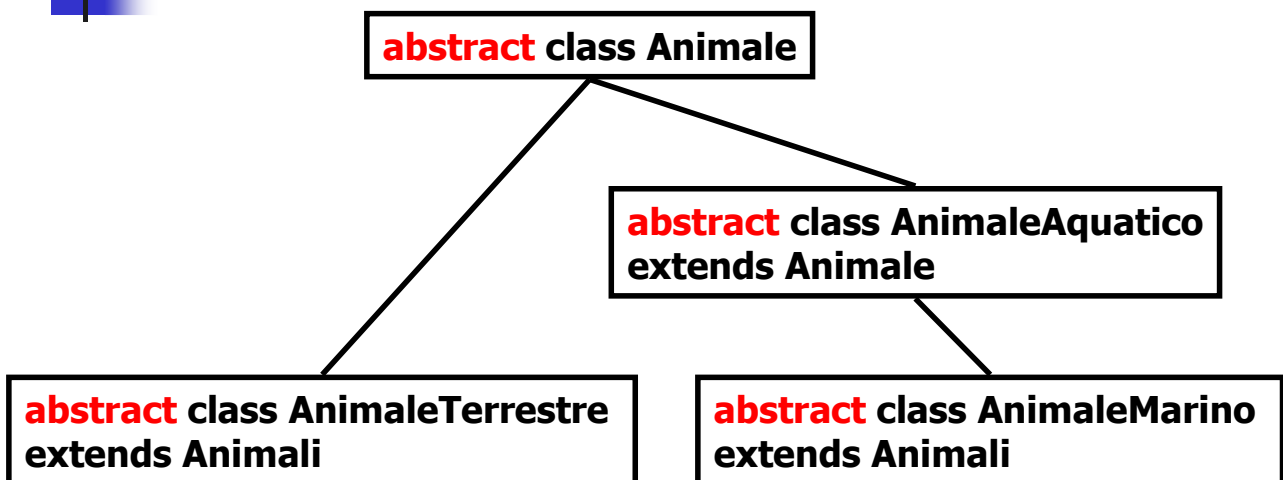
```
public abstract class AnimaleTerrestre extends Animale {
    public AnimaleTerrestre(String s) {super(s); }
    public String vive() {return "sulla terraferma"; }
    public String chi_sei()
        {return "un animale terrestre"; }
}
```

```
public abstract class AnimaleAcquatico extends Animale {
    public AnimaleAcquatico(String s) {super(s); }
    public String vive() {return "nell'acqua"; }
    public String chi_sei() {
        return "un animale acquatico"; }
}
```

- Due metodi astratti su tre sono ridefiniti, ma uno è ancora astratto quindi la classe è ancora astratta

7

Classi astratte - Esempio



- Una possibile specializzazione:
 - Perché introdurre l'animale marino?
 - non è correlato a verso, movimento, ma rispecchia semplicemente una realtà.

8



Classi astratte - Esempio

```
public abstract class AnimaleMarino extends AnimaleAcquatico {  
    public AnimaleMarino(String s) {super(s); }  
    public String vive() {return "in mare"; }  
    public String chi_sei() {return "un animale marino"; }  
}
```

- Specializza i metodi vive() e chi_sei(), ma non definisce l'altro e quindi è ancora astratta



Classi astratte - Esempio

La Tassonomia Completa:

- Animale
 - Animale_terrestre
 - Quadrupede Bipede Uccello
 - Uomo Cavallo
 - Corvo Pinguino
 - Pesce
 - Animale_acquatico
 - Animale_marino
 - Tonno

Classi astratte - Esempio

```
public class PesceDiMare extends AnimaleMarino {
    public PesceDiMare(String s)
        {super(s);
         verso = "non fa versi"; }
    public String chi_sei()
        {return "un pesce (di mare)"; }
    public String si_muove() {
        return "nuotando"; }
    }
}
```

- Definisce l'ultimo metodo astratto rimasto si_muove(). La classe non è più astratta

11

Classi astratte - Esempio

```
public class Uccello extends AnimaleTerrestre {
    public Uccello(String s) {super(s);verso="cinguetta"; }
    public String si_muove() { return "volando"; }
    public String chi_sei() { return "un uccello"; }
    public String vive() { return "in un nido su un albero"; }
}
public class Bipedede extends AnimaleTerrestre {
    public Bipedede(String s) { super(s); }
    public String si_muove() { return "avanzando su 2 zampe"; }
    public String chi_sei() { return "un animale con due zampe"; }
}
public class Quadrupede extends AnimaleTerrestre {
    public Quadrupede(String s) { super(s); }
    public String si_muove() { return "avanzando su 4 zampe"; }
    public String chi_sei() { return "un animale con 4 zampe"; }
}
public class Cavallo extends Quadrupede {
    public Cavallo(String s) { super(s); verso = "nitrisce"; }
    public String chi_sei() { return "un cavallo"; } }
}
```

12

Classi astratte - Esempio

```

public class Corvo extends Uccello {
    public Corvo(String s) {super(s); verso = "gracchia"; }
    public String chi_sei() {return "un corvo"; }
}

public class Uomo extends Bipedo {
    public Uomo(String s) {super(s); verso = "parla"; }
    public String si_muove() {return "camminando su 2 gambe"; }
    public String chi_sei() {return "un homo sapiens"; }
    public String vive() {return "in condominio"; }
}

public class Pinguino extends Uccello {
    public Pinguino(String s) { super(s); verso = "non fa versi"; }
    public String chi_sei() { return "un pinguino"; }
    public String si_muove() { return "ma non sa volare"; }
}

public class Tonno extends PesceDiMare {
    public Tonno(String s) { super(s); }
    public String chi_sei() { return "un tonno"; }}

```

13

Classi astratte - Esempio

```

UN MAIN... "mondo di animali"
public class MondoAnimale {
    public static void main(String args[]) {
        Cavallo c = new Cavallo("Furia");
        Uomo h = new Uomo("John");
        Corvo w = new Corvo("Pippo");
        Tonno t = new Tonno("Giorgio");
        Uccello u = new Uccello("Gabbiano");
        Pinguino p = new Pinguino("Tweety");
        c.mostra(); h.mostra();
        w.mostra(); t.mostra();
        u.mostra(); p.mostra();
    }
}

```

14

Interfacce

- Nella sua parte non statica, una classe fornisce la definizione di un ADT, ossia la parte visibile esternamente (public), ed anche però la relativa implementazione (dati e metodi privati, protetti, o visibili nel package)
- In questo modo, però, diventa impossibile dare una pura specifica di interazione di un ADT senza definirlo.
- Alcuni linguaggi permettono poi l'ereditarietà multipla:



- L'ereditarietà multipla fornisce una grande flessibilità ...
- ma anche una grande confusione, perchè possono essere presenti metodi con lo stesso nome ereditati da classi distinte, e quindi con semantica probabilmente differente

15

Interfacce

- Date quindi le due esigenze di:
 - potere definire un ADT senza doverlo implementare
 - potere supportare una forma di ereditarietà multipla senza conflitti
- Può essere utile disporre di un nuovo costrutto simile alla (parte non statica di una) classe, nel senso di consentire la definizione del "modo di interagire" di un'entità...
 - ma non tenuto a fornire implementazioni...
 - ... né legato alla gerarchia di ereditarietà delle classi, con i relativi vincoli.
- Nasce quindi **l'interfaccia**
- Un interfaccia è una collezione di metodi headers senza la loro definizione. Un interfaccia può anche dichiarare costanti.
- La scopo è quello di definire un protocollo del comportamento che deve essere fornito ad una classe. Una qualsiasi classe che implementa una data interfaccia è quindi obbligata a fornire l'implementazione di tutti i metodi elencati nell'interfaccia.
- Ad esempio, esistono le interfacce *Cloneable*, *Runnable*, *Serializable*

16



Interfacce

- Una interfaccia costituisce una pura specifica di interazione
 - contiene solo dichiarazioni di metodi, ed eventualmente costanti
 - ma non variabili né implementazioni di metodi
- Praticamente, una interfaccia è strutturalmente analoga a una classe, ma è introdotta dalla parola chiave `interface` anziché `class`
- Le interfacce inducono un diverso modo di concepire il progetto:
 - prima si definiscono le interfacce delle entità che costituiscono il sistema
 - in questa fase si considerano scelte di progetto (pulizia concettuale)
 - poi si realizzeranno le classi che implementeranno tali interfacce
 - in questa fase entreranno in gioco scelte implementative (efficienza ed efficacia)

17



Interfacce

- La dichiarazione di un'interfaccia adotta la sintassi:


```
<modificatore> interface <nome> { <membri> }
```
- il modificatore può essere:
 - annotazione
 - `public` (alternativamente, non si mette nulla e sarà accessibile a livello di package)
 - `strictfp`, che impone l'aritmetica stretta per l'inizializzazione delle costanti poste come membri; non viene invece propagato alcun vincolo sull'essere `strictfp` per i metodi, essendo essi astratti
- i membri possono essere costanti, metodi, e classi o interfacce innestate

18



Interfacce

- le costanti sono implicitamente `static`, `final` e `public`, e devono necessariamente essere inizializzati (inizializzazioni ritardate non sono ammesse); sono identificabili tramite la notazione puntata con il nome dell'interfaccia
- i metodi possono solo avere annotazioni, e nessun altro modificatore di accesso; sono implicitamente `public`, `abstract`, non possono essere statici (perché `static` e `abstract` sono incompatibili), né `final`, né prevedono `synchronized`, `strictfp`, `native` (tutti modificatori influenti sull'implementazione che qui non esiste)
- Esempio:

```
interface Verbose {
    int VERBOSE=3;
    void setVerbosity(int level);
    int getVerbosity();
}
```

19



Interfacce

- Le interfacce appaiono simili alle classi astratte ma hanno significative differenze:
 - Un interfaccia non può avere metodi con implementazione, mentre una classe astratta può fornire una implementazione parziale
 - Una classe può implementare molte interfacce ma può essere derivata da una sola superclasse
 - Le interfacce non fanno parte della gerarchia delle classi, quindi classi non discendenti l'una dall'altra possono implementare la stessa interfaccia
 - se occorre fornire ereditarietà multipla, si usano le interfacce
 - se si deve fornire una parte (comune) dell'implementazione, si usano le classi astratte
 - se tuttavia ci si trova a scrivere una classe astratta senza alcuna implementazione, di fatto è un'interfaccia

20



Interfacce - Esempio

- Definizione dell'astrazione "Collezione"
 - Cosa si intende per "Collezione"?
 - Come ci si aspetta di poter interagire con un'entità qualificata come "Collezione"?
- Indipendentemente da qualsiasi scelta o aspetto implementativo, una "Collezione" è tale perché:
 - è un contenitore
 - è possibile chiedersi se è vuota e quanti elementi contiene
 - vi si possono aggiungere e togliere elementi
 - è possibile chiedersi se un elemento è presente o no
 - ...
- Una "Collezione" è dunque una qualsiasi entità che si conformi a questo "protocollo di accesso"
- Si definiscono così astrazioni di dato in termini di comportamento osservabile, ossia di:
 - cosa ci si aspetta da esse
 - cosa si pretende che esse sappiano fare
 - rinviando a tempi successivi la realizzazione pratica di ADT (classi) che rispettino questa specifica.

21



Interfacce - Esempio

```

public interface Collection {
    public boolean add(Object x);
    public boolean contains(Object x);
    public boolean remove(Object x);
    public boolean isEmpty();
    public int size();
    ...
}

```

Interfacce - Gerarchie

- Le interfacce possono dare luogo a gerarchie, come le classi:

```
public interface List extends Collection {
    ...
}
```

- La gerarchia delle interfacce:
 - è una gerarchia separata da quella delle classi
 - è slegata dagli aspetti implementativi
 - esprime le relazioni concettuali della realtà
 - guida il progetto del modello della realtà.
- Le problematiche sono simili a quelle delle classi:
 - le costanti possono essere adombrate; è sempre possibile risalire ad una costante premettendo il nome della classe (membri statici)
 - dei metodi può essere operato l'overloading e/o l'overriding, più formale che sostanziale non esistendo alcuna implementazione₂₃

Interfacce - Gerarchie

- Come in ogni gerarchia, anche qui le interfacce derivate:
 - possono aggiungere nuove dichiarazioni di metodi
 - possono aggiungere nuove costanti
 - non possono eliminare nulla
- Significato: "Ogni lista è anche una collezione"
 - ogni lista può interagire col mondo come farebbe una collezione (magari in modo specializzato)...
 - ... ma può avere proprietà peculiari al concetto di lista, che non valgono per una "collezione" qualsiasi.

Interfacce - Gerarchie

- Ad esempio, una "Lista" ha un concetto di sequenza, di ordine fra i suoi elementi
 - esiste quindi un primo elemento, un secondo, ...
 - quando si aggiungono nuovi elementi bisogna dire dove aggiungerli (ad esempio, in coda)
 - è possibile recuperare un elemento a partire dalla sua posizione (il primo, il decimo,...)

```
public interface List extends Collection {
    public boolean add(int posizione, Object x);
    public Object get(int posizione);
    ...
}
```

25

Interfacce – Ereditarietà multipla

- Java supporta l'ereditarietà multipla fra interfacce
- una interfaccia contiene solo dichiarazioni di metodi
 - non ha implementazioni nessun problema di collisione fra metodi omonimi
 - non ha variabili nessun problema di collisione fra dati omonimi
- È un potente strumento di modellizzazione

```
public interface Worker { ... }
public interface Student { ... }
public interface WorkerStudent extends Worker, Student {
    ...
}
```

- Dopo extends può esservi un elenco di più interfacce

26

Interfacce – Implementazione

- Una interfaccia definisce una astrazione di dato in termini di comportamento osservabile, per sua natura però non implementa nulla.
- Qualcuno dovrà prima o poi implementare la astrazione definita dall'interfaccia. A questo fine, una classe può implementare (una o più) interfacce tramite la keyword `implements`; in tal caso, la classe deve implementare tutti i metodi richiesti, pena errore di compilazione.
- Una classe può anche implementare più interfacce; deve comunque fornire implementazione per tutti i metodi previsti
- se le interfacce da implementare prevedono metodi con la stessa signature, o se la classe implementa un'interfaccia estesa che ha operato l'overriding, l'implementazione fornita per tutti quei metodi che hanno (in un modo o in un altro) la stessa signature è unica; non è detto che essa tuttavia soddisfi tutti i contratti

27

Interfacce – Uso

- Il nome di una interfaccia può essere usato come identificatore di tipo per riferimenti e parametri formali di metodi e funzioni.

```

public static void main(String s[]){
    Collection c;
    ...
    List l1;
    List l2;
    ...
}

```

Interfacce – Uso

- A tali riferimenti si possono assegnare istanze di una qualunque classe che implementi l'interfaccia

```
public static void main(String s[]){
    Collection c = new Vector();
    c.add(new Point(3,4));

    ..
    List l1 = new LinkedList();
    List l2 = new LinkedList(c);
}
```

- Le classi Vector e LinkedList implementano entrambe l'interfaccia List, che a sua volta estende Collection
- Il costruttore di default di Vector e LinkedList crea una collezione di oggetti vuota

29

Interfacce – Uso

- In effetti, quando una classe implementa una interfaccia, le dichiarazioni dei metodi contengono il nome dell'interfaccia...

```
public List copy(List z){
    return new LinkedList(z);
}
```

- ... ma in realtà tali metodi ricevono e manipolano istanze di una qualche classe che implementi l'interfaccia richiesta.
- I riferimenti che introduciamo devono riflettere la vista esterna
 - Collection, List, ...
- Gli oggetti che creiamo devono necessariamente essere istanze di classi concrete
 - LinkedList, Vector, ...
- Esempi

```
Collection c = new Vector();
List l1 = new LinkedList();
List l2 = new LinkedList(c);
```

30



Interfacce – Uso

- In generale, per utilizzare un'interfaccia si può:
 - estendere una classe che la implementa
 - se occorre invece estendere un'altra classe, allora la classe data si dichiara come implementatrice dell'interfaccia, ma anziché dovere riscrivere l'implementazione per tutti i metodi, solitamente si crea un oggetto della classe che implementa già l'interfaccia, e tutti i metodi da implementare dentro la nuova classe sono ciascuno di fatto una chiamata al corrispondente (e già implementato) metodo della classe che implementa già l'interfaccia. Questa tecnica è nota come reindirizzamento (forwarding)

31



Interfacce – Uso

Esempio di Forwarding:

```
class Prova extends AltraClasse implements Interface
{
  private ClassCheImplementaGiaInterface x =
    new ClassCheImplementaGiaInterface ();
  public void metodo1Interfaccia(parametri) {
    x.metodo1Interfaccia(parametri);
  }
  public QualcheClasse metodo2Interfaccia(parametri) {
    return x.metodo2Interfaccia(parametri);
  }
}
```

32



Classi ed Interfacce innestate

- Un possibile membro di classi e di interfacce potrebbe essere un'altra classe o interfaccia; in tal caso essi sono definiti innestati
- L'innesto è ortogonale rispetto all'ereditarietà; un membro innestato è indipendente dalla classe o interfaccia entro cui si trova; viene ereditato in caso la classe/interfaccia contenitrice venga estesa
- L'innesto solitamente viene effettuato per connettere in modo semplice ed efficace oggetti correlati logicamente, caratteristica utilizzata ampiamente dal sistema AWT
- è ammesso l'innesto in profondità, ma se ne sconsiglia l'uso

33



Classi ed Interfacce innestate

- Una classe innestata può di per sé essere statica o meno; un'interfaccia innestata è invece per sua natura statica
- se una classe è innestata all'interno di un'interfaccia sarà considerata statica
- Quando una classe innestata non è statica, si definisce interna, in particolare
 - locale, se dichiarata dentro un metodo o un blocco di inizializzazione,
 - anonima, quando è priva di nome e viene allora definita solo entro il comando new
 - una classe può essere interna ad un'interfaccia, ma in tal caso non potrà essere locale o anonima (le interfacce sono vuote)

34

Classi ed Interfacce innestate

- Tassonomia degli innesti:

	Classe (statica o non)	Interfaccia
Classe statica	X	X
Classe non statica (interna, locale, anonima)	X	-
Interfaccia	X	X

- La denominazione per le classi o interfacce innestate è la notazione puntata `<nome_contenitore>.<nome_membro_innestato>`
- il membro innestato può accedere a tutti i membri del contenitore, compresi quelli privati, e viceversa, essendo di fatto l'innesto una relazione di fiducia
- un membro innestato può essere esteso da altre classi o interfacce, alla quali comunque deve risultare accessibile; la classe o interfaccia estesa non eredita comunque l'accesso privilegiato di cui il membro innestato dispone nei confronti del contenitore; un membro esteso può estendere qualsiasi classe/interfaccia, anche la contenitrice

35

Classi ed Interfacce innestate

- Quando il membro innestato è una classe statica dentro un'altra, essa viene di fatto utilizzata come componente software che vive dentro il suo contenitore
- Se ad essere innestata in una classe invece è una classe non statica, essa produrrà oggetti, che devono quindi essere associati al contenitore (se non statico)
- ogni istanza di una classe innestata non statica (interna) deve essere associata ad almeno una istanza della classe contenitrice; il viceversa non è vero, nel senso che l'istanza della classe contenitrice potrebbe anche non avere associata alcuna istanza della classe innestata o averne più di una

36



Classi ed Interfacce innestate

- L'associazione fra classe interna e classe non statica contenitrice può essere semplicemente fatto tramite un oggetto della classe innestata, ad esempio:

```
public class Contenitore {
    private int campo1;
    private Innestata x;
    public class Innestata {
        ...
    }
    public void metodo1() {
        ...
        x= new Innestata(); // o anche x=this.new Innestata();
        ...
    }
}
```

37



Classi ed Interfacce innestate

- L'estensione delle classi interne segue le stesse regole dell'estensione di qualsiasi altra classe, ma con il vincolo che ogni oggetto della sottoclasse dovrà essere collegato ad oggetti della classe contenitrice o di una sua sottoclasse, ad esempio:

```
Class Outer {          Class Inner {}          }
Class ExtendsOuter extends Outer{
    Class ExtendedInner extends Inner {}
    Inner ref=new ExtendedInner();
}
```

- Altro esempio:

```
Class Unrelated extends Outer.Inner {
    Unrelated (Outer ref) { ref.super(); }
}
```

- in questo caso, la classe Unrelated non è interna ed è completamente separata da Outer e Inner; occorre però fornire al costruttore di Inner, invocato dentro il costruttore della sottoclasse Unrelated, un riferimento ad un oggetto contenitore Outer

38



Classi ed Interfacce innestate

- La classe interna può adombrare campi e/o metodi della classe contenitrice semplicemente sovrascrivendoli
- Una classe interna può però anche nascondere membri della classe contenitrice se eredita omonimi da qualche altra classe, ad esempio:

```
class Uno {
    int x;
    class Due extends Tre {
        void increment () { x++; }
    }
}
class Tre {
    int x;
}
```

- la x su cui opera Due non è quella di Uno ma quella di Tre; occorre in tal caso evitare il problema o utilizzare riferimenti espliciti, tipo this.x o Uno.this.x

39



Classi ed Interfacce innestate

- La classe interna può essere locale quando è definita dentro un metodo o blocco di inizializzazione; in tal caso, il suo livello di visibilità è limitato al solo ambiente del metodo o blocco in cui è definita; tale classe interna non è di fatto considerabile membro della classe in cui si trova
- una classe interna può essere locale ed il suo contenitore (membro o blocco di inizializzazione) essere statico; in tal caso non si collega a nessuna istanza della classe contenitrice
- una classe interna è anonima se è senza nome e viene definita nello stesso momento in cui viene istanziata tramite il new; le classi anonime dovrebbero essere utilizzate solo se costituite da poche righe di codice, ed il loro scopo è quello di mantenere semplice il codice esistente; se ciò non accade, è bene non ricorrere a questo strumento

40