Università degli Studi di Modena e Reggio Emilia

Facoltà di Ingegneria – Reggio Emilia

CORSO DI

RETI DI CALCOLATORI
Linguaggio Java: Ereditarietà

Prof. Franco Zambonelli

Lucidi realizzati in collaborazione con
Ing. Enrico Denti - Univ. Bologna

Anno Accademico 2001-2002

AGGIORNAMENTO / MODIFICA DEI REQUISITI

Spesso si incontrano problemi che richiedono classi *simili* a classi già esistenti, ma *non identiche*.

Come fare per non rifare tutto da capo?

- ricopiare manualmente il codice della classe esistente e cambiare quel che va cambiato
  → *si ha una proliferazione di file... e poi è tutto manuale!*
- creare una nuova classe che contenga al suo interno un oggetto della classe preesistente
  → *i campi dati privati però non sono direttamente manipolabili*
  → *bisogna riscrivere anche i metodi che "rimangono uguali"*

ESEMPIO: da un `Counter` base al Counter con decremento

```
class Counter2 {
    Counter c;
    public  Counter2()      {c = new Counter(); }
    public  Counter2(int v) {c = new Counter(v);}
    public  void inc()      { c.inc(); }
    public  int  rd()       { return c.rd(); }
    public  void dec()      { c.val--; }
}
```

- il campo dati `c.val` è accessibile *perché ha visilibità package* e la classe `Counter2` è definita nello stesso package di `Counter`
- è comunque necessario riscrivere 4 metodi su 5 *solo per adattarli alla nuova classe*, ma di fatto *riscrivendoli uguali, senza aggiungere nuove funzionalità* → *spreco di tempo e di risorse*!

Occorre poter riusare le classi esistenti in modo *più flessibile*.

EREDITARIETÀ

- Si vuole riusare tutto ciò che può essere riusato
- non è utile né opportuno modificare codice già funzionante e corretto
- il cui sviluppo ha richiesto tempo (anni-uomo) e denaro

Occorre disporre nel linguaggio di un modo per *progettare alle differenze*, procedendo *incrementalmente*

- *non ripartire da zero* quando serve una nuova classe, ma ***poterla definire a partire da una già esistente***

Bisognerà dire:

- cosa la nuova classe ha *in più* rispetto alla precedente
- sia in termini di *dati*, sia in termini di *operazioni*.

ESEMPIO: da un `Counter` base al Counter con decremento

```
class Counter2 extends Counter {
    public  void dec()      { val--; }
}
```

- la classe `Counter2` **eredita tutti i metodi e i campi dati** di `Counter`, e può usarli come fossero definiti localmente (*sempre che la visibilità lo permetta*)
- *in più,* aggiunge dati e metodi *suoi specifici* (in questo esempio, nessun dato, solo un metodo, `dec`)

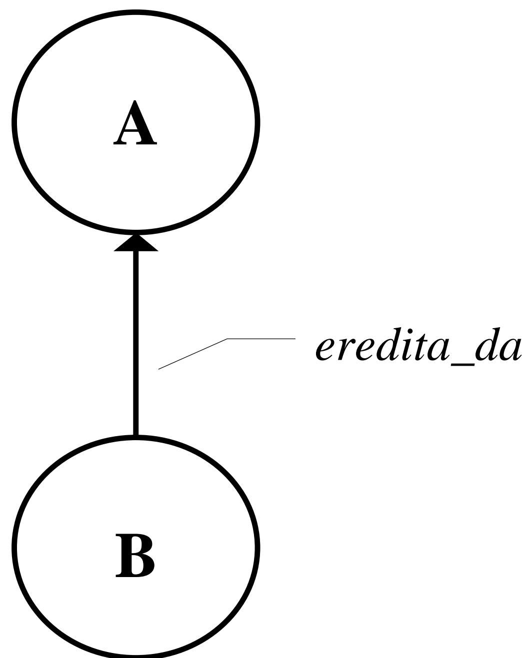
UNA NUOVA RELAZIONE FRA CLASSI

Finora, l'unica relazione introdotta era l'*istanziazione*:
- ogni oggetto è *istanza di* una certa classe
- relazione fra *un oggetto* (da un lato) e *una classe* (dall'altro)

EREDITARIETÀ = una nuova relazione fra classi

una classe B EREDITA DA una (preesistente) classe A

classe base (o ***superclasse***) è la classe ***da cui si eredita***	(A)
classe derivata (o ***sottoclasse***) è la classe ***che eredita***	(B)



LA CLASSE DERIVATA PUO':

- **aggiungere nuovi campi dati e nuovi metodi** a quelli ereditati dalla classe-base
- **ridefinire alcuni dei metodi** ereditati dalla classe-base

LA CLASSE DERIVATA NON PUO':

- **eliminare campi dati o metodi** (comportamento monotono, si accresce sempre)

VISIBILITA'

Nella classe-base:

- ciò che è *privato* è visibile *solo ai metodi della classe stessa*
- ciò che è *di package* è visibile *solo ai metodi delle classi dello stesso package*
- ciò che è *pubblico* è *visibile a tutti*

Perciò, la classe derivata:

- ***non può vedere la parte privata*** *della classe-base*, appunto perché privata;
- ***può vedere la parte di package*** *della classe-base*, purché la nuova classe sia definita nello stesso package
- ***vede, ovviamente, la parte pubblica*** *della classe-base*, in quanto visibile a tutti.

Basterà ?

MEMBRI PROTETTI

Riconsideriamo il problema precedente di definire `Counter2` (dotato di decremento) a partire da `Counter` (che ne è privo).

Per realizzare l'operazione di decremento, è necessario:

- o disporre dell'accesso diretto al campo `val`;
- o sfruttare, se esiste, un metodo (pubblico o di package) che consenta di *inizializzare il campo val* con un valore a piacere.

Nel nostro caso, **ce l'abbiamo fatta *solo perché `val` non era privato*** (aveva visibilità package) **e *perché `Counter2` è stata posta nello stesso package di `Counter`*** (quello di default),

E se `val` fosse stato privato?

E se non fosse stato possibile collocare `Counter2` nello stesso package di `Counter`?

Per sfruttare davvero l'ereditarietà occorre ***superare la rigidità*** dei tre livelli di visibilità privato / pubblico / package:

occorre una visibilità specifica per le classi derivate

`protected`

Un campo dati o un metodo `protected`:

- **è visibile <u>alle classi derivate</u> *indipendentemente dal package* in cui esse sono definite**
- **per il resto si comporta come la visibilità *package*.**

VISIBILITA' IN JAVA: RIASSUNTO

Classe di Visibilità

Accessibile da	*public*	*protected*	*package*	*private*
Stessa Classe	sì	sì	sì	sì
Classe (o Sottoclasse) nello stesso package	sì	sì	sì	no
Sottoclasse in package differente	sì	sì	no	no
Non sottoclasse, package differente	sì	no	no	no

UNA RIFLESSIONE

La qualifica `protected`:

- rende visibile un attributo *a tutte le sottoclassi* di quella classe, *presenti e future*
- perciò, costituisce un *permesso di accesso "indiscriminato"*, valido per *ogni possibile sottoclasse che possa in futuro essere definita*, senza possibilità di distinzione.

Occorre dunque valutare con attenzione quando ciò sia opportuno.

COMPATIBILITÀ FRA CLASSI

- **Un `Counter2` ha sicuramente tutti i medoti e gli attributi di di `Counter`**

- **ogni `Counter2` è anche un `Counter`**, ovvero:
  - ogni oggetto di classe `Counter2` è *implicitamente* anche di classe `Counter`
  - **ma non viceversa,** perché `Counter2` *è più ricco* di `Counter` (sa fare cose che un `Counter` non sa fare, e può avere dai campi dati inesistenti in `Counter`)

La sottoclasse è un sottotipo della superclasse

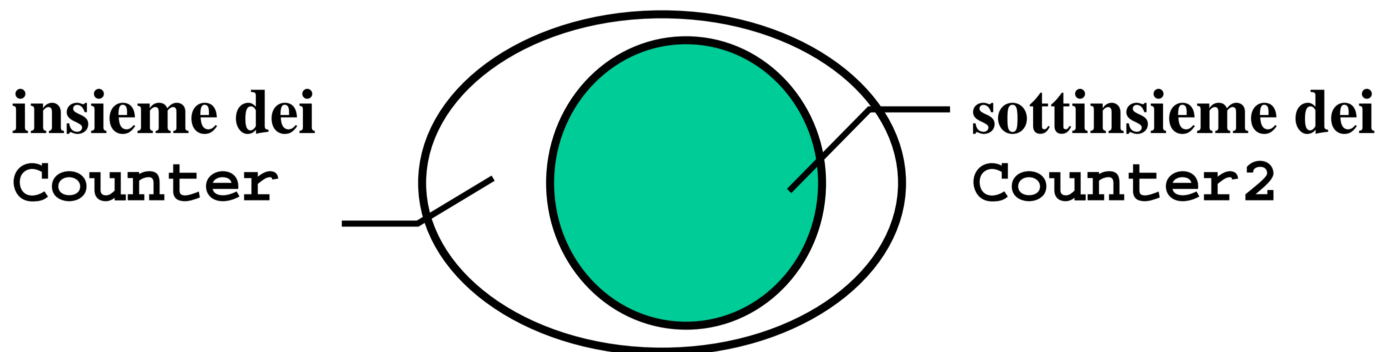
Conseguenze sul piano semantico:

- l'ereditarietà *non è soltanto un modo per "riusare codice"*
- ***induce una clasificazione*** fra le entità del "mondo"
- ***aderenza al mondo reale*** di questa "classificazione" … ?

PERCIÒ:

- **un `Counter2` può "*prendere il posto*" di un `Counter` *senza che gli altri componenti se ne accorgano***
- **ma *non il contrario***

ESEMPIO



- Una sottoclasse delimita in qualche modo un sottoinsieme della superclasse. I Counter2 sono un sottoinsieme dei Counter in quanto ne hanno tutte le caratteristiche, e hanno alcune caratteristiche particolari.
- La classe degli studenti è un sottoinsieme della classe piú generale delle persone. Perció, quando parliamo di uno persona, in generale, questa persona puó tranquillamente essere uno studente. Infatti, uno studente è anche una persona.
- Analogamente, quando usiamo un riferimento a un oggetto di una certa classe, possiamo tranquillamente sopportare il fatto che questo riferimento punti a un oggetto della sottoclasse.
- Non è vero il contrario: quando parliamo di studenti, non possiamo pensare che l'oggetto del nostro discorso diventi una persona che studente non é.

```
class Esempio1 {
    Counter c1 = new Counter(3);
    Counter2 c2 = new Counter2();
    c2.dec();      // OK: dec() esiste in Counter2
//  c1.dec();   // NO: dec() non c'è in Counter
    c1=c2;  // OK: un Counter2 è anche un Counter
//  c2=c1;      // NO: c1 è un semplice Counter
}
```

EREDITARIETÀ e COSTRUTTORI

Riconsideriamo un momento la definizione di `Counter2`:

```
class Counter2 extends Counter {
    public  void dec()      { val--; }
}
```

Abbiamo usato un oggetto senza definirne il costruttore!
Come è possibilie??

- Java *non obbliga* a definire almeno un costruttore per ogni classe, anche se ciò è *fortemente raccomandato*
- in assenza di costruttori definiti esplicitamente, Java definisce automaticamente un *costruttore di default*
- il costruttore di default *non fa nulla di specifico,* salvo **chiamare il costruttore di default della superclasse**

```
class Counter2 extends Counter {
    public  void dec()      { val--; }
    // il costruttore di default generato da Java
    // public Counter2() { super(); }
}
```

L'espressione **`super()`** invoca il costruttore di default della classe-base: se esso non esiste si ha errore.

EREDITARIETÀ e COSTRUTTORI

Con il meccanismo della ereditarietá si ereditano:

- *tutti i campi dati* (anche quelli *private* a cui la classe derivata non potrà accedere direttamente)
- *tutti i metodi* (anche quelli *private* che la classe derivata non potrà usare direttamente)

tranne i costruttori

PERCHÉ?

- Perché i costruttori non svolgono *funzioni*, ma inizializzano un'istanza della classe
  ➔ sono **specifici di quella particolare classe**
- costruire un `Counter` non è la stessa cosa che costruire un `Counter2` (un `Counter2` in generale ha più campi dati)

Ciò non significa che la classe derivata non abbia un costruttore, in quanto, come vedremo, ne viene generato uno di default.

Quindi…..

Per costruire un oggetto di una classe derivata, è *sempre necessario* chiamare un costruttore della classe-base, in quanto:

- solo il costruttore della classe base può sapere come inizializzare i campi-dati ereditati da tale classe in modo corretto
- è il solo modo per garantire l'inizializzazione di campi-dati privati (a cui la sottoclasse non può accedere direttamente)
- si evita una inutile duplicazione di codice nella sottoclasse.

Quale costruttore della classe-base viene chiamato?

- il costruttore di default definito da Java (se noi non lo definiamo) chiama il costruttore di default della classe base (se questo non esiste si ha errore)
- per i costruttori definiti dall'utente occorre *specificare esplicitamente* quale costruttore della classe base vada chiamato, mediante la notazione `super(…..)`, *che dev'essere la prima istruzione del corpo del costruttore.*

  Se questa manca, Java inserisce automaticamente una chiamata al costruttore di default della classe base (`super()`), *a meno che* non ci sia già una chiamata a un altro costruttore della stessa classe mediante la notazione `this()`.

ESEMPIO

```
class Counter2 extends Counter {
    public  void dec()      { val--; }
    // il costruttore di default generato da Java
    // public Counter2()  { super(); }
    public Counter2(int x) { super(x); }
}
```

L'espressione **`super(x)`** invoca il costruttore con un argomento della classe-base (se esso non esiste si ha errore).

super vs. this

Riassumendo:

- `this` è un riferimento all'istanza corrente
- `super` è un riferimento alla classe base

Sono espressioni lecite:

- `this.val` che indica il campo `val` della classe dell'istanza corrente
- `this.f()` che richiama il metodo `f()` della classe dell'istanza corrente
- `this(…)` che richiama un altro costruttore della stessa classe

- `super(…)` che richiama un costruttore della classe base
- `super.val` che indica il campo `val` della classe base
- `super.f()` che richiama il metodo `f()` della classe base

Le ultime due notazioni sono utili se la sottoclasse definisce un campo dati o un metodo omonimo con uno della classe base.

ESEMPIO (ricorda: `Counter` definisce un campo `val`)

```
class DualCounter extends Counter {
    double val;
    // val indica ora il campo val di DualCounter
    // mentre super.val indica il val di Counter
    public void inc() { super.inc(); val++; }
    // inc() indica il metodo inc() di DualCounter
    // super.inc() chiama l'inc() di Counter
}
```

COSTRUTTORI NON PUBBLICI

Come si è già visto, in assenza di costruttori definiti dall'utente, Java definisce sempre un costruttore di default per ogni classe: tale costruttore è *pubblico*, in modo da consentire alla classe di essere normalmente istanziata. Tale construttore o non fa niente o, se la classe è derivata, invoca il costruttore della superclasse.

In alcune circostante tuttavia si può desiderare di *impedire che una classe possa essere istanziata da chiunque*: in particolare, può essere necessario:

- impedire qualunque istanziazione
- consentire solo istanziazioni da parte di sottoclassi

Nel primo caso, l'utente deve definire un costruttore *privato*, nel secondo un costruttore *protetto*,

In questo modo, si previene la generazione del costruttore pubblico automatico da parte di Java.

CLASSI FINALI (`final`)

Come si è già visto, una variabile dichiarata **`final`** diventa una costante, ossia non è più modificabile.

Allo stesso modo, **una classe dichiarata `final` diventa una classe finale, da cui non è possibile derivare sottoclassi.**

ESEMPIO: PERSONE E STUDENTI

```
/**
* la classe Persona
@author Enrico Denti
@version 1.0, 13/4/98
*/

class Persona {
  String nome;
  int anni;
  Persona() { nome = "sconosciuto"; anni = 0; }
  Persona(String n) { nome = n; anni = 0; }
  Persona(String n, int a) { nome=n; anni=a; }
  void print(){
    System.out.print("Il mio nome e' " + nome);
    System.out.println(" e ho " + anni + " anni");
  }
}
```

```
/**
* la classe Studente che eredita da Persona
@author Enrico Denti
@version 1.0, 13/4/98
*/

class Studente extends Persona {
  int matr;
  Studente() { super(); matr = 9; }
  Studente(String n) { super(n); matr = 8; }
  Studente(String n, int a) { super(n,a); matr=7;}
  Studente(String n, int a, int m) {
        super(n,a); matr=m; }
  void print(){ super.print();
    System.out.println("Matricola = " + matr); }
}
```

PERSONE E STUDENTI (segue)

```
public class Esempio2 {
  public static void main(String args[]){
  Studente studente;
  studente = new Studente();
  studente.print();

  studente = new Studente("Anna");
  studente.print();

  studente = new Studente("Tom", 30);
  studente.print();

  studente = new Studente("Laura", 20, 1234);
  studente.print();
  }
}
```

NOTA:

- Sebbene esista un metodo `print()` sia in `Persona` che in `Studente`, viene usato quello di quest'ultimo in quanto più specifico.

COMPATIBILITA' DI TIPO

Come si è visto, se la classe B è sottoclasse dalla classe A:

- ogni oggetto di classe B è anche di classe A (non viceversa)
- e dunque la classe B è *un sottotipo* della classe A

Ciò implica una **compatibilità da sottotipo a tipo** (non viceversa), da cui segue la possibilità di usare **un oggetto della classe derivata** *al posto di* **uno della classe-base** (non viceversa).

ESEMPIO

```
class Esempio3 {
    Persona  p = new Persona("John");
    Studente s = new Studente("Tom");
    p.print();      // stampa solo il nome
    s.print();      // stampa nome e matricola
    p=s;
    p.print();      // COSA STAMPA ???
}
```

L'assegnamento `p=s`:

- **è possibile**, perché ogni `Studente` è anche una `Persona`
- **e non comporta perdite di informazione**, perché si tratta di un assegnamento *fra riferimenti* (non fra variabili!)

Questo pone un problema:

- `p` è <u>definito</u> come riferimento a una generica `Persona`
- **ma <u>fa riferimento in realtà</u> a uno `Studente`**, che è altra cosa!

Ma allora….

come verrà interpretato adesso `p`?

Come semplice `Persona` *o come* `Studente` *?*

POLIMORFISMO

Una funzione si dice ***POLIMORFA*** *se è capace di operare su oggetti di tipo diverso* ***specializzando il suo comportamento in base al tipo dell'oggetto su cui opera***

I linguaggi a oggetti hanno intrinseco il polimorfismo, in quanto possono esistere – in classi diverse – funzioni con lo stesso nome ma con effetti completamente diversi.

POLIMORFISMO ORIZZONTALE

ESEMPIO

```
public class Esempio2 {
  public static void main(String args[]){
    Studente s = new Studente("Anna");
    Point p = new Point();

    p.print(); // METODO PRINT DI Point
    s.print(); // METODO PRINT DI Studente
   }
}
```

Ovvio e naturale…..

MA ATTENZIONE: quando le classi sono legate da erediarietà possono crearsi dei problemi di polimorfismo più complessi. Infatti, si consideri la possibilità di usare un oggetto della classe derivata al posto di uno della classe base. Se entrambe le classi forniscono lo stesso metodo, quale dei due deve venire viene invocato?

POLIMORFISMO VERTICALE

```
public class Esempio2 {
  public static void main(String args[]){
    Persona p = new Studente("Anna");
    p.print(); // METODO PRINT DI PERSONA O DI
               // STUDENTE ?
  }
}
```

- se prevale *il tipo formale del riferimento* (`Persona` nell'esempio precedente) si parla di LEGAME STATICO
- se prevale *il tipo effettivo dell'istanza corrente* (`Studente` nell'esempio precedente) si parla di LEGAME DINAMICO

In Java ***prevale il tipo effettivo dell'istanza corrente***
→ LEGAME DINAMICO (LATE BINDING)

ESEMPIO

```
public class Esempio2 {
  public static void main(String args[]){
    Persona p = new Studente("Anna");
    p.print(); // STAMPA NOME E MATRICOLA !!
  }
}
```

Sebbene `p` sia un riferimento a `Persona`, poiché l'istanza corrente è uno `Studente` viene richiamato il metodo `print()` della classe `Studente`.

- In C++ si avrebbe un comportamento diverso, perché verrebbe chiamato il metodo della classe `Persona` (a meno di usare una specifica notazione)