Sapienza Università di Roma, Facoltà di Ingegneria

Corso di

Fondamenti di Informatica I

Canale 1 (A-K)

Anno Accademico 2009-2010

Corso di Laurea in Ingegneria Informatica

Docente: Camil Demetrescu

Esercitatore: Andrea Ribichini

(Dispensa basata su materiale didattico del corso di Laboratorio di Programmazione tenuto negli anni scorsi)

Ereditarietà e polimorfismo in Java

Indice

Indice della dispensa:

Derivazione fra classi

È possibile dichiarare una classe `D` come *derivata* da una classe `B`.

```
class B {                  // CLASSE BASE
  int x;
  void G() { x = x * 20; }
// ...
}

class D extends B {        // CLASSE DERIVATA
  void H() { x = x * 10; }
// ...
}
```

Princìpi fondamentali della derivazione

I quattro **princìpi fondamentali** del rapporto tra classe derivata e classe base:

1. Tutte le proprietà definite per la classe base vengono **implicitamente definite** anche nella classe derivata, cioè vengono **ereditate** da quest'ultima.

   Ad esempio, implicitamente la classe derivata `D` ha:
   - un campo dati `int x;`
   - una funzione `void G()`

2. La classe derivata può avere **ulteriori proprietà** rispetto a quelle ereditate dalla classe base.

   Ad esempio, la classe `D` ha una funzione `void H()`, in più rispetto alla classe base `B`.

Princìpi fondamentali della derivazione (cont.)

3. Ogni oggetto della classe derivata **è anche** un oggetto della classe base.

   Ciò implica che è possibile usare un oggetto della classe derivata **in ogni situazione o contesto** in cui si può usare un oggetto della classe base.

   Ad esempio, i seguenti usi di un oggetto di classe `D` sono leciti.

```
  static void stampa(B bb) {
    System.out.println(bb.x);
  }
//...
D d = new D();
d.G();     // OK: uso come ogg. di invocazione di funz. definita in B
stampa(d); // OK: uso come argomento in funz. definita per B
```

La classe `D` *è* compatibile *con la classe* `B`

Princìpi fondamentali della derivazione (cont.)

4. **Non è vero che** un oggetto della classe base è anche un oggetto della classe derivata.

   Ciò implica che **non è possibile** usare un oggetto della classe base laddove si può usare un oggetto della classe derivata.

```
B b = new B();
// b.H();
//    ^
// Method H() not found in class B.

// d = b;
//   ^
// Incompatible type for =. Explicit cast needed to convert B to D.

b = d;     // OK: D al posto di B
```

Derivazione e regole di visibilità

Una classe `D` derivata da un'altra classe `B`, anche se in un package diverso, ha una relazione particolare con quest'ultima:

- **non è un cliente qualsiasi** di `B`, in quanto possiamo usare oggetti di `D` al posto di quelli di `B`;

- **non coincide** con la classe `B`.

Per questo motivo, è possibile che `B` voglia mettere a disposizione dei campi (ad esempio i campi dati) solo alla classe `D`, e non agli altri clienti. In tal caso, questi campi devono essere dichiarati **protetti** (e non privati).

Ciò garantisce al progettista di `D` di avere accesso a tali campi (vedi tabella delle regole di visibilità), **senza tuttavia garantire tale accesso ai clienti generici** di `B`.

Costruttori di classi derivate

Al momento dell'invocazione di un costruttore della classe derivata, se il costruttore della classe derivata non contiene esplicite chiamate al costruttore della classe base (vedi dopo), viene chiamato **automaticamente** anche il costruttore senza argomenti della classe base. Ciò avviene:

- sia se la classe base ha il costruttore senza argomenti standard,
- sia se la classe base ha il costruttore senza argomenti definito esplicitamente,
- sia se la classe non ha il costruttore senza argomenti(!), in questo caso si ha un errore di compilazione.

Esempio: costruttori di classi derivate

```
class B { ... }
class D extends B { ... }

...
D d = new D(); // invoca il costruttore senza argomenti di B()
               // e quello di D()
```

Costruttori di classi derivate (cont.)

Il costruttore senza argomenti della classe base viene invocato:

- **anche se non definiamo** alcun costruttore per la classe derivata (che ha quindi quello standard senza argomenti),

- **prima** del costruttore della classe derivata (sia quest'ultimo definito esplicitamente oppure no).

Costruttori di classi derivate: esempio

```
// File ereditarieta/Esempio1.java

class B1 { protected int x_b1; }
class D1 extends B1 { protected int x_d1; } // OK: B1 ha cost. senza arg.

class B2 {
  protected int x_b2;
  public B2() { x_b2 = 10; }
}
class D2 extends B2 { protected int x_d2; } // OK: B2 ha cost. senza arg.

class B3 {
  protected int x_b3;
  public B3(int a) { x_b3 = a; }
}
// class D3 extends B3 { protected int x_d3; } // NO: B3 NON ha c. senza arg.
//       ^
// No constructor matching B3() found in class B3.
```

Costruttori di classi derivate: uso di super()

Se la classe base ha costruttori con argomenti, è probabile che si voglia **riusarli**, quando si realizzano le classi derivate.

È possibile invocare esplicitamente un costruttore qualsiasi della classe base **invocandolo**, nel corpo del costruttore della classe derivata.

Ciò viene fatto mediante il costrutto `super()`, che deve essere la *prima istruzione eseguibile* del corpo del costruttore della classe derivata.

Uso di super() nei costruttori: esempio

```
// File ereditarieta/Esempio2.java
class B {
  protected int x_b;
  public B(int a) {  // costruttore della classe base
    this.x_b = a;
  }
}
class D extends B {
  protected int x_d;
  public D(int b, int c) {
    super(b);        // RIUSO del costruttore della classe base
    this.x_d = c;
  }
}
class Esempio2 {
  public static void main(String[] args) {
    D d = new D(3,4);  }  }
```

Costruttori di classi derivate: riassunto

Comportamento di un costruttore di una classe `D` derivata da `B`:

1. **se** ha come prima istruzione `super()`, allora viene chiamato il costruttore di `B` esplicitamente invocato;

   altrimenti viene chiamato il costruttore senza argomenti di `B`;

2. viene eseguito il corpo del costruttore.

Questo vale **anche per il costruttore standard** di `D` senza argomenti (come al solito, disponibile se e solo se in `D` non vengono definiti esplicitamente costruttori).

Riuso di costruttori in una classe: `this()`

È possibile **riusare** costruttori gìa definiti anche nell'ambito di una stessa classe. Ciò è reso possibile dal costrutto `this()` che, analogamente a `super()`, deve comparire come *prima istruzione* nel costruttore della classe.

```
public class Persona {
  private String nome;
  private String residenza;

  public Persona(String n, String r) {
    nome = n;
    residenza = r;
  }

  public Persona(String n) {
    this(n, null);
  }
```

```
  public Persona() {
    this("Mario Rossi");
  }
  ...
}
```

NOTA: l'uso contemporaneo di `this()` e `super()` in uno stesso costruttore non è possibile (entrambi dovrebbero comparire come prima istruzione).

Gerarchie di classi

Una classe derivata può a sua volta fungere da classe base per una **successiva derivazione**.

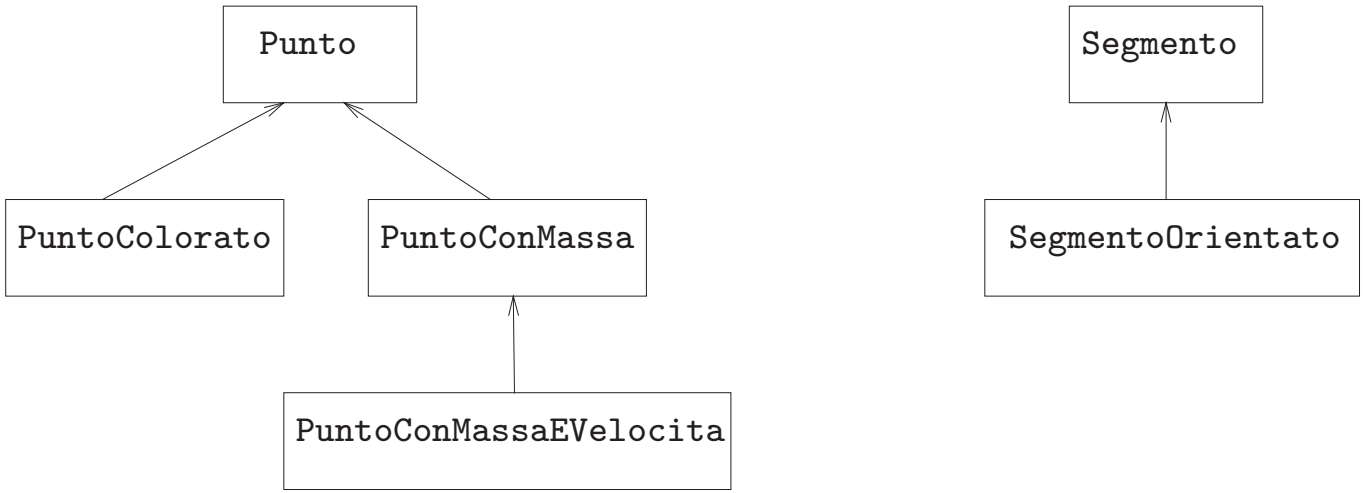
Ogni classe può avere **un numero qualsiasi** di classi derivate.

```
class B { ...
class D extends B { ...
class D2 extends B { ...
```

Una classe derivata può avere **una sola classe base**, (in Java non esiste la cosiddetta *ereditarietà multipla*).

Java supporta una sorta di ereditarietà multipla attraverso le *interfacce* – *maggiori dettagli in seguito*.

Gerarchie di classi: esempio



Significato dell'assegnazione

Abbiamo visto che, in base al principio 3 dell'ereditarietà, la seguente istruzione è lecita:

```
class B { /*...*/ }              // CLASSE BASE
class D extends B { /*...*/ } // CLASSE DERIVATA
// ...
D d = new D();
B b = d;                         // OK: D al posto di B
```

- **Non viene creato** un nuovo oggetto.

- Esiste **un solo oggetto**, di classe `D`, che viene denotato:
  - **sia** con un riferimento `d` di classe `D`,
  - **sia** con un riferimento `b` di classe `B`.

Casting

Non è possibile accedere ai campi della classe `D` attraverso `b`. Per farlo dobbiamo prima fare un **casting**.

```
// File ereditarieta/Esempio3.java
class B { }
class D extends B { int x_d; }

public class Esempio3 {
  public static void main(String[] args) {
    D d = new D();
    d.x_d = 10;
    B b = d;
    System.out.println(((D)b).x_d);  // CASTING
  }
}
```

Casting (cont.)

Regola generale: Un riferimento di tipo T può puntare solo a oggetti di tipo T o di una sua sottoclasse.

Tentare di violare questa regola mediante un cast genera un errore (semantico) a tempo di esecuzione di tipo `ClassCastException`.

```
// File ereditarieta/Esempio4.java
class B { }
class D extends B { int x_d; }

public class Esempio4 {
  public static void main(String[] args) {
    B b = new B();
    D d = (D)b; // ERRORE SEMANTICO: NON SI PUO' FARE IL CAST
    // java.lang.ClassCastException: B
    //  at Esempio4.main(Compiled Code)
    System.out.println(d.x_d);
  }
}
```

Derivazione e overloading

È possibile fare **overloading** di funzioni ereditate dalla classe base esattamente come lo si può fare per le altre funzioni.

```
public class B {
  public void f(int i) { ... }
}

public class D extends B {
  public void f(String s) { ... } // OVERLOADING DI f()
}
```

La funzione `B.f(int)` ereditata da `B` **è ancora accessibile** in `D`.

```
D d = new D();
d.f(1);        // invoca f(int) ereditata da B
d.f("prova"); // invoca f(String) definita in D
```

Overriding di funzioni

Nella classe derivata è possibile anche fare **overriding** (dall'inglese, *ridefinizione*, *sovrascrittura*) delle funzioni della classe base.

Fare overriding di una funzione `f()` della classe base `B` vuol dire definire nella classe derivata `D` una funzione con lo stesso nome, lo stesso numero e tipo di parametri della funzione `f()` definita in `B`. Si noti che **il tipo di ritorno delle due funzioni deve essere identico**.

```
public class B {
  public void f(int i) { ... }
}

public class D extends B {
  public void f(String s) { ... } // OVERLOADING DI f()
  public void f(int n) { ... }    // OVERRIDING DI f()
}
```

Overriding di funzioni: esempio

```
// File ereditarieta/Esempio5.java
class B {
  public void f(int i) { System.out.println(i*i); } }
class D extends B {
  public void f(String s) { // OVERLOADING DI f()
    System.out.println(s); }
  public void f(int n) {    // OVERRIDING DI f()
    System.out.println(n*n*n); }
}
public class Esempio5 {
  public static void main(String[] args) {
    B b = new B();
    b.f(5);          // stampa 25
    D d = new D();
    d.f("ciao");     // stampa ciao
    d.f(10);         // stampa 1000     } }
```

Overriding e riscrittura

Su oggetti di tipo `D` **non è più possibile invocare** `B.f(int)`.

È ancora possibile invocare `B.f(int)` **solo dall'interno della classe** `D` attraverso un campo predefinito `super` (analogo a `this`).

Riassunto overloading e overriding

	OVERLOADING	OVERRIDING
nome della funzione	uguale	uguale
tipo restituito	qualunque	uguale
numero e/o tipo argomenti	diverso	uguale
relazione con la funzione della classe base	coesiste con la funzione della classe base	cancella la funzione della classe base

Esercizio: overriding e compatibilità

```
// File ereditarieta/Esercizio1.java
class B {
  protected int c;
  void stampa() { System.out.println("c: " + c); }
}
class D extends B {
  protected int e;
  void stampa() {
    super.stampa();
    System.out.println("e: " + e);
  }
}
public class Esercizio1 {
  public static void main(String[] args) {
    B b  = new B();    b.stampa();
    B b2 = new D();    b2.stampa();
    D d  = new D();    d.stampa();
    D d2 = new B();    d2.stampa();
  }
}
```

Il programma contiene errori rilevabili dal compilatore?

Una volta eliminati tali errori, cosa stampa il programma?

Overriding di funzioni: late binding

Invocando `f(int)` su un oggetto di `D` viene invocata **sempre** `D.f(int)`, **indipendentemente** dal fatto che esso sia denotato attraverso un riferimento `d` di tipo `D` o un riferimento `b` di tipo `B`.

```
public class B {
  public void f(int i) { ... }
}

public class D extends B {
  public void f(int n) { ... }
}
// ...
D d = new D();
d.f(1);  // invoca D.f(int)
B b = d; // OK: classe derivata usata al posto di classe base
b.f(1);  // invoca di nuovo D.f(int)
```

Late binding (cont.)

Secondo il meccanismo del **late binding** la scelta di quale funzione invocare non viene effettuata durante la compilazione del programma, **ma durante l'esecuzione**.

```
public static void h (B b) { b.f(1); }
// ...
B bb = new B();
D d = new D();
h(d);    // INVOCA D.f(int)
h(bb);   // INVOCA B.f(int)
```

Il gestore run-time riconosce **automaticamente** il tipo dell'oggetto di invocazione:

`h(d)`: `d` denota un oggetto della classe `D` – viene invocata la funzione `f(int)` definita in `D`;

`h(bb)`: `bb` denota un oggettp della classe `B` – viene invocata la funzione `f(int)` definita in `B`.

Esercizio: cosa fa questo programma?

```
// File ereditarieta/Esercizio2.java
class B {
  protected int id;
  public B(int i) { id = i; }
  public boolean get() { return id < 0; }
}

class D extends B {
  protected char ch;
  public D(int i, char c) {
    super(i);
    ch = c;
  }
  public boolean get() { return ch != 'a'; }
}

public class Esercizio2 {
  public static void main(String[] args) {
    D d = new D(1,'b');
    B b = d;
    System.out.println(b.get());
    System.out.println(d.get());
  }
}
```

Overriding e livello d'accesso

Nel fare overriding di una funzione della classe base è possibile cambiare il livello di accesso alla funzione, **ma solo rendendolo meno restrittivo**.

```
// File ereditarieta/Esempio6.java
class B {
  protected void f(int i) { System.out.println(i*i); }
  protected void g(int i) { System.out.println(i*i*i); }
}

class D extends B {
  public void f(int n) { System.out.println(n*n*n*n); }
  //  private void g(int n) {
  //               ^
  // Methods can't be overridden to be more private.
  // Method void g(int) is protected in class B.
  //    System.out.println(n*n*n*n*n); }
}
```

Impedire l'overriding: final

Qualora si voglia **bloccare l'overriding** di una funzione la si deve dichiarare `final`.

Anche una classe può essere dichiarata `final`, impedendo di derivare classi dalla stessa e rendendo implicitamente `final` tutte le funzioni della stessa.

```
// File ereditarieta/Esempio7.java
class B {
  public final void f(int i) { System.out.println(i*i); }
}
class D extends B {
// public void f(int n) { System.out.println(n*n*n*n); }
// Final methods can't be overriden. Method void f(int) is final in class B.
}
final class BB {}
// class DD extends BB {}
// Can't subclass final classes: class BB
```

Sovrascrittura dei campi dati

Se definiamo nella classe derivata una variabile con lo stesso nome e di diverso tipo di una variabile della classe base, allora:

- la variabile della classe base **esiste ancora** nella classe derivata, ma non può essere acceduta utilizzandone semplicemente il nome;
- si dice che la variabile della classe derivata **nasconde** la variabile della classe base;
- per accedere alla variabile della classe base è necessario utilizzare **un riferimento ad oggetto della classe base**.

Sovrascrittura dei campi dati: esempio

```
// File ereditarieta/Esempio8.java
class B { int i; }
class D extends B {
  char i;
  void stampa() {
    System.out.println(i); System.out.println(super.i);
  }
}
public class Esempio8 {
  public static void main(String[] args) {
    D d = new D();
    d.i = 'f';
    ((B)d).i = 9;
    d.stampa();
  }
}
```

Classi astratte

Le classi astratte sono classi particolari, nelle quali una o più funzioni possono essere solo **dichiarate** (cioè si descrive la segnatura), ma non **definite** (cioè non si specificano le istruzioni).

Esempio:

Ha certamente senso associare alla classe `Persona` una funzione che calcola la sua aliquota fiscale, ma il vero e proprio calcolo per una istanza della classe `Persona` **dipende** dalla sottoclasse di `Persona` (ad esempio: straniero, pensionato, studente, impiegato, ecc.) a cui l'istanza appartiene.

Vogliamo poter definire la classe `Persona`, magari con un insieme di campi e funzioni normali, anche se non possiamo scrivere il codice della funzione `aliquota()`.

Classi astratte: esempio

La soluzione è definire la classe `Persona` come **classe astratta**, con la funzione `aliquota()` astratta, e definire poi le sottoclassi di `Persona` come classi non astratte, in cui definire la funzione `aliquota()` con il relativo codice:

```
abstract class Persona {
  abstract public int aliquota(); // Questa e' una DICHIARAZIONE
                                  // (senza codice)
  private int eta;
  public int getEta() { return this.eta; }
}

class Studente extends Persona {
  public int aliquota() { ... } // Questa e' una DEFINIZIONE
}

public class Professore extends Persona {
  public int aliquota() { ... } // Questa e' una DEFINIZIONE
}
```

Quando una classe va definita astratta

Una classe `A` si definirà come astratta quando:

- non ha senso pensare a oggetti che siano istanze di `A` **senza essere istanze anche di una sottoclasse (eventualmente indiretta) di** `A`;

- esiste una funzione che ha senso associare ad essa, ma il cui codice non può essere specificato a livello di `A`, mentre può essere specificato a livello delle sottoclassi di `A`; si dice che tale funzione è astratta in `A`.

Anche se spesso si dice che una classe astratta `A` non ha istanze, ciò non è propriamente corretto: la classe astratta `A` non ha istanze dirette, ma ha come istanze tutti gli oggetti che sono istanze di sottoclassi di `A` non astratte.

Si noti che la classe astratta può avere funzioni non astratte e campi dati.

Uso di classi astratte

Se `A` è una classe astratta, allora:

- **Non possiamo** creare direttamente oggetti che sono istanze di `A`. Non esistono istanze dirette di `A`: gli oggetti che sono istanze di `A` lo sono indirettamente.

- **Possiamo:**

  – definire variabili o campi di altri classi (ovvero, riferimenti) di tipo `A` (durante l'esecuzione, conterranno indirizzi di oggetti di classi non astratte che sono sottoclassi di `A`),

  – usare normalmente i riferimenti (tranne che per creare nuovi oggetti), ad esempio: definire funzioni che prendono come argomento un riferimento di tipo `A`, restituire riferimenti di tipo `A`, ecc.

Riassunto classi e classi astratte

	`class`	`abstract class`
Riferimenti	SI	SI
Oggetti	SI	SI (indirettamente)
Campi dati	SI	SI
Funzioni solo dichiarate	NO	SI
Funzioni definite	SI	SI
`extends (abstract) class`	0 o 1	0 o 1