



Programmazione III

Prof.ssa Liliana Ardissono
Dipartimento di Informatica
Università di Torino

JAVA – Ereditarietà – parte 1



Questa presentazione è distribuita sotto licenza Creative Commons CC BY ND



Ereditarietà 1

- Meccanismo per lo sviluppo incrementale di programmi
- Consente di estendere classi preesistenti aggiungendo o modificando componenti (variabili o metodi)



Ereditarietà 2

Data la classe

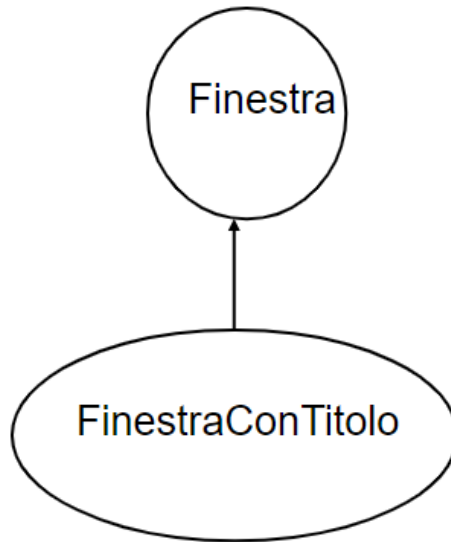
```
class Finestra {  
    Rettangolo r; ....  
    void disegnaCornice() {...}  
    void disegnaContenuto() {...}  
}
```

vogliamo estendere la finestra aggiungendo un titolo

```
class FinestraConTitolo extends Finestra {  
    String titolo;  
    void disegnaTitolo() {...}  
}
```



Sottoclassi



FinestraConTitolo è una **sottoclasse** di Finestra: gli oggetti di tipo FinestraConTitolo sono anche oggetti di tipo Finestra.

Una sottoclasse **eredita** tutte le variabili e i metodi della sopraclasse.



Ereditarietà 3

Gli oggetti della classe **Finestra** sono costituiti da

variabili

Rettangolo r;

metodi

void disegnaCornice() {...}

void disegnaContenuto() {...}

Gli oggetti della classe **FinestraConTitolo** sono costituiti da

variabili

Rettangolo r;

String titolo;

metodi

void disegnaCornice() {...}

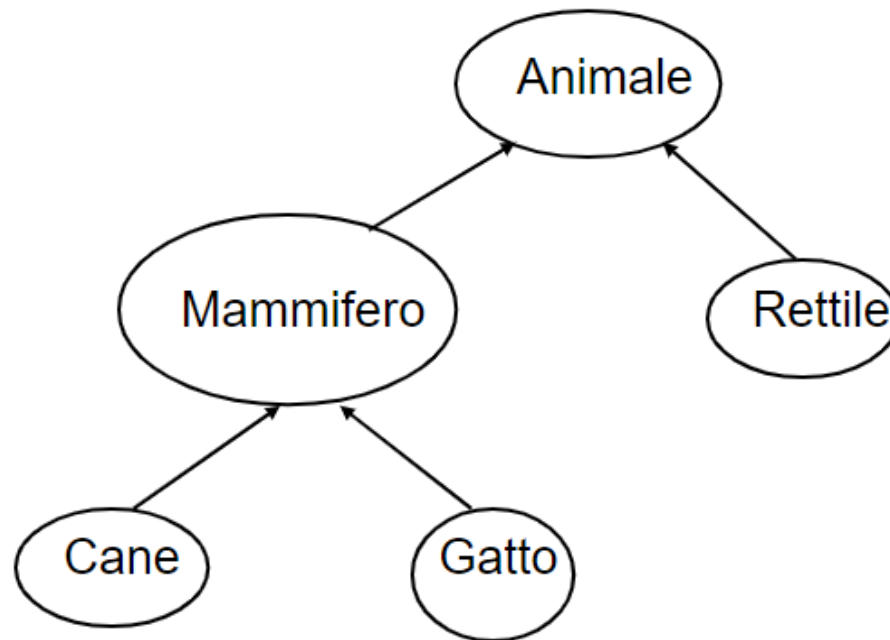
void disegnaContenuto() {...}

void disegnaTitolo() {...}



Tassonomie

Spesso l'ereditarietà è utilizzata per rappresentare tassonomie (classificazioni)



Controllo statico dei tipi 1



Java, come molti altri linguaggi, effettua un controllo dei tipi (**type checking**) **statico**.

Statico: fatto dal compilatore prima di iniziare l'esecuzione del programma. Identifica errori sintattici e mismatch di tipo tra variabili, guardando le loro dichiarazioni.

*(C'è poi il controllo di tipo **Dinamico**: fatto dall'interprete durante l'esecuzione (a runtime), che vedremo in seguito)*



Controllo statico dei tipi 2

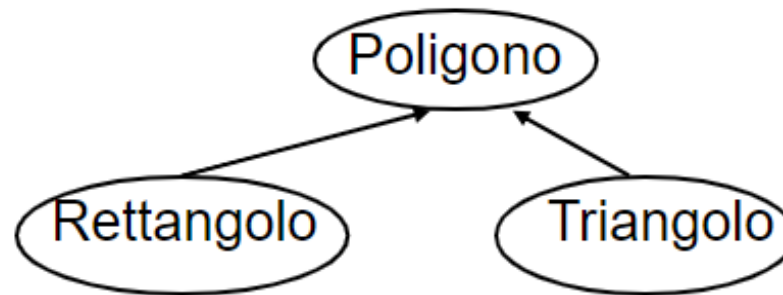
```
Finestra f;  
FinestraConTitolo ft;  
...  
ft.disegnaCornice();  
f.disegnaCornice();  
ft.disegnaTitolo();  
f.disegnaTitolo(); // errore di compilazione  
// f è una finestra e non ha il metodo disegnaTitolo()
```

Type checking statico: il compilatore controlla che per una variabile si chiami un metodo definito per la classe di quella variabile.

Sottotipi



Una sottoclasse può essere vista come l'implementazione di un **sottotipo**.



L'ereditarietà realizza una relazione **Is-a** (è un).

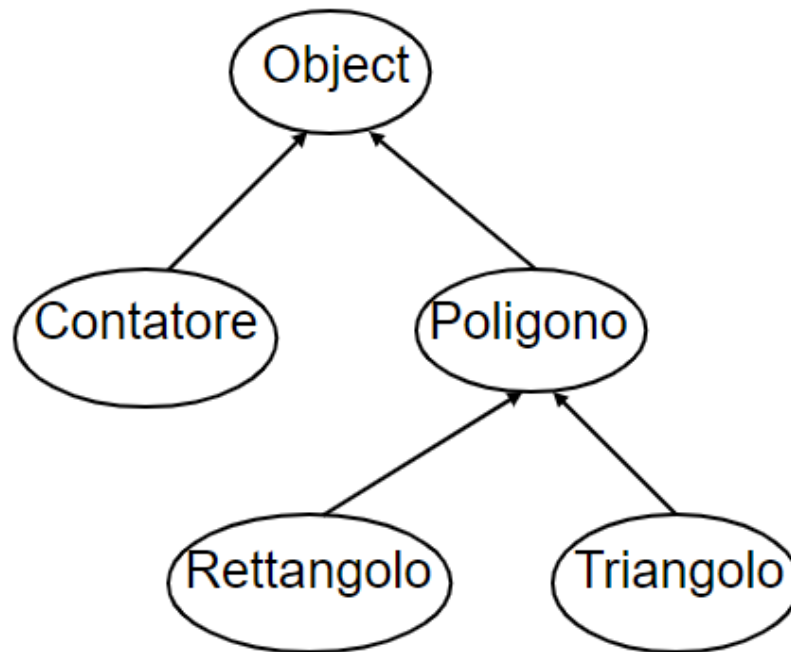
Un rettangolo **è un** poligono.

Un rettangolo ha tutte le operazioni di poligono (eventualmente ridefinite) più altre operazioni specifiche.

Ereditarietà singola



Ogni sottoclasse ha una sola sopraclasse.
Struttura ad albero.



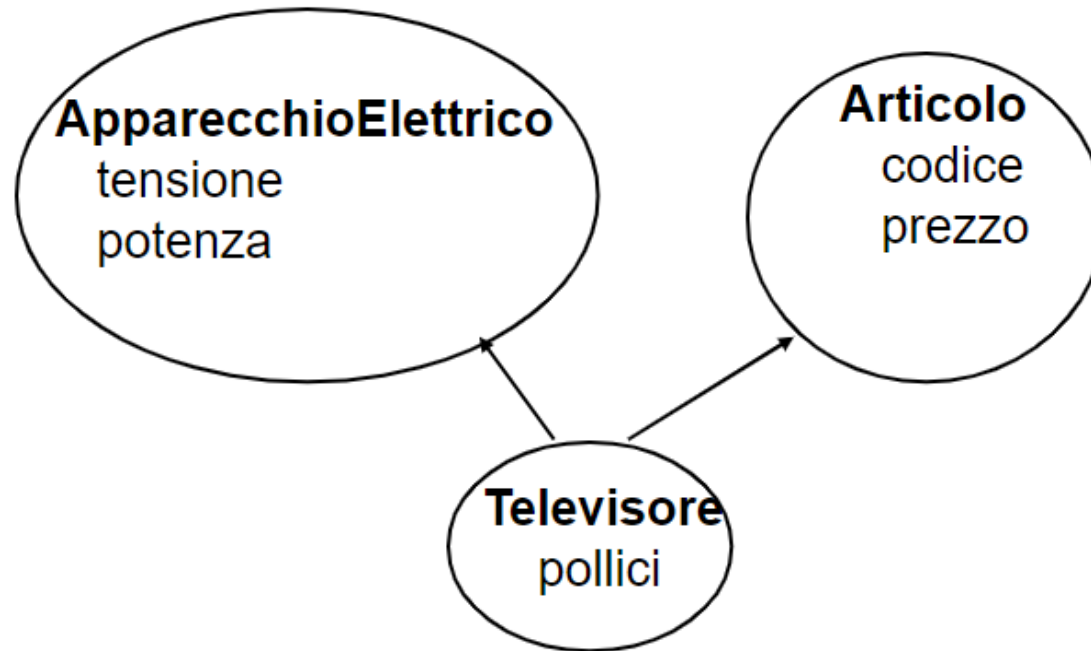
In Java la classe **Object** è la radice della gerarchia.

Qualunque oggetto è un **Object**.

I tipi primitivi non sono **Object**.



Ereditarietà multipla

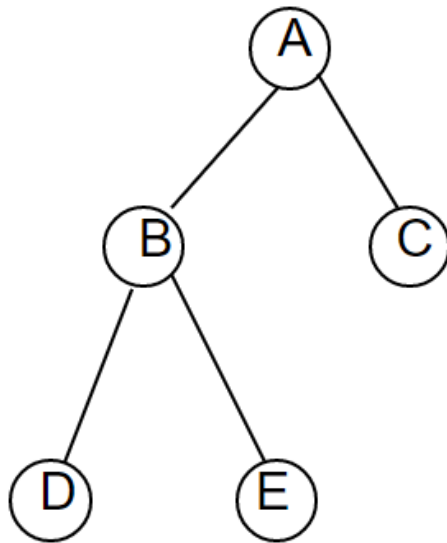


In Java l'ereditarietà multipla non è permessa.
L'ereditarietà multipla dà maggiore espressività ma crea problemi di conflitti e duplicazioni.



Polimorfismo 1

POLIMORFISMO: proprietà di un oggetto di avere più di un tipo, in accordo alla relazione di ereditarietà.



D sottoclasse di B e di A

un oggetto D è **un B** ed è **un A**

un oggetto di tipo (classe) D è
anche un oggetto di tipo (classe)
B e anche un oggetto di tipo
(classe) A

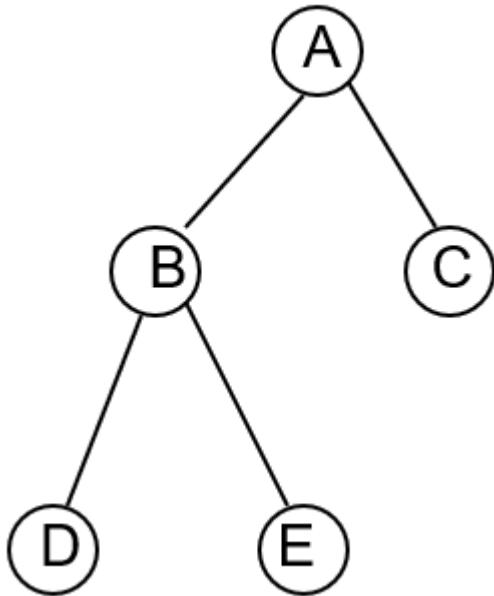


Polimorfismo 2

Questi assegnamenti sono tutti legali perché

- D extends B
- B extends A.

Quindi un oggetto di tipo D ha anche tipo B e A



```
A a; B b; D d;  
d = new D();  
b = new D();  
a = new D();  
a = b;
```



Polimorfismo 3

Sia A il tipo di x e B il tipo di $expr$. Allora, l'assegnamento

$X = expr;$

è legale per il compilatore se:

- A uguale a B, oppure
- B sottoclasse (sottotipo) di A

Analogamente se x è un parametro formale di un metodo e $expr$ è il parametro attuale (della chiamata).

Controllo statico.



Upcasting 1

```
A a; B b; D d;  
d = new D();
```

```
b = d; // d viene visto come se fosse un oggetto di tipo B
```

```
a = d; // d viene visto come se fosse un oggetto di tipo A
```

Upcasting: ci si muove da un tipo specifico ad uno più generico (da un tipo ad un suo “supertipo”).

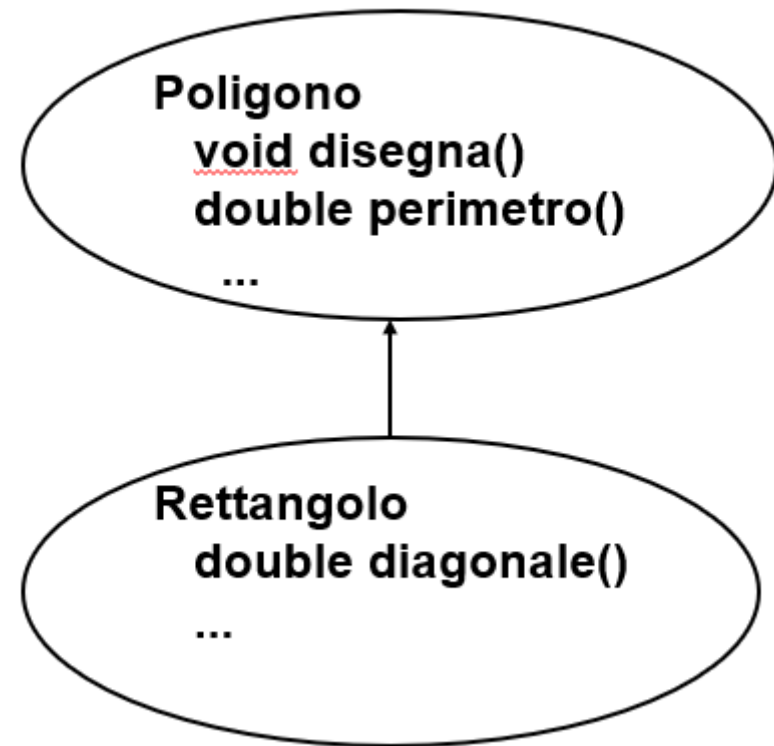
L'*upcasting* è sicuro per il type checking: dato che una sottoclasse eredita tutti i metodi delle sue sopraclassi, ogni messaggio che può essere inviato ad una sopraclasse può anche essere inviato alla sottoclasse senza il rischio di errori durante l'esecuzione.



Upcasting 2

Questo è corretto per il compilatore:

```
Poligono p;  
Rettangolo r;  
...  
p.disegna();  
r.disegna();  
p.perimetro();  
r.perimetro();
```

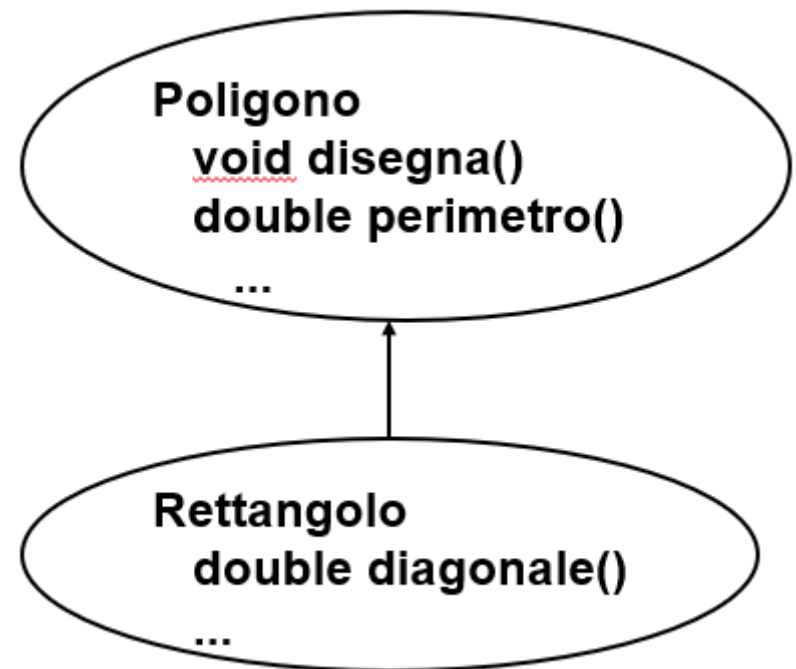


Upcasting 3



```
Poligono p;  
Rettangolo r;  
...  
r.diagonale();
```

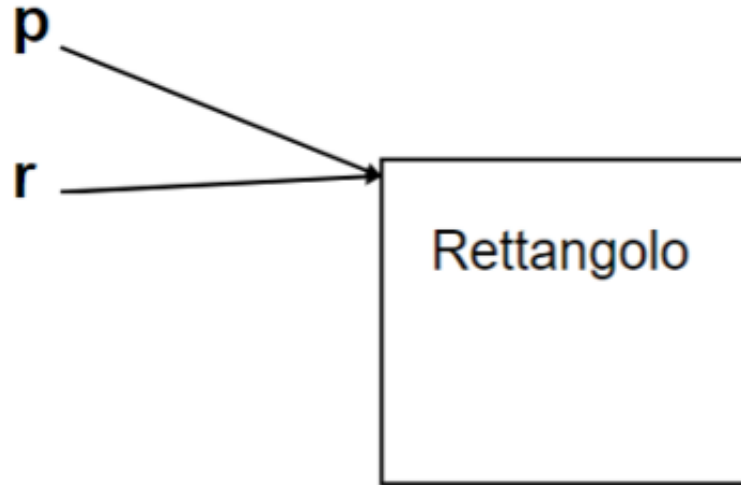
```
p.diagonale(); // errore di compilazione  
                // p è di tipo Poligono e non  
                // ha il metodo diagonale()
```





Upcasting 4

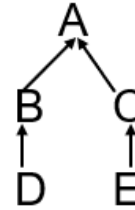
```
Poligono p;  
Rettangolo r;  
...  
p = r;  
r.diagonale();  
p.diagonale();
```



Il compilatore dà errore perché la variabile *p* ha tipo Poligono.

Tuttavia, se si facesse il controllo a runtime, **p.diagonale()** sarebbe corretto perché *p* è legata a un rettangolo, che possiede il metodo *diagonale()*.

Upcasting - esempio



```
class A { public String stampa() {return "sono un A";}}
class B extends A {public String stampa() {return "sono un B";}}
class C extends A {public String stampa() {return "sono un C";}}
class D extends B {public String stampa() {return "sono un D";}}
class E extends C {public String stampa() {return "sono un E";}}
```

```
public class UpcastingTest {
    public static void main(String[] args) {
        A a; B b;
        D d = new D(); System.out.println(d.stampa());           // stampa «sono un D»
        b = new D(); // oggetto di tipo D visto come se fosse di tipo B
        System.out.println(b.stampa());                           // stampa «sono un D»
        a = new D(); System.out.println(a.stampa()); // idem per A
                                                    // stampa «sono un D»
        a = b; // assegno a una variabile di tipo A un oggetto D (visto come A)
        System.out.println(a.stampa());                           // stampa «sono un D»
        // b = a; errore: tipi incompatibili (cerco di assegnare un A ad un B)
    }
}
```

Overriding di metodi



Una sottoclasse può **ridefinire (OVERRIDING)** un metodo della sua superclasse. Esempio:

```
class Cella {  
    int contenuto=0;  
    int get() {return contenuto;}  
    void set(int n) {contenuto=n;}  
}  
  
class CellaConMemoria extends Cella {  
    int backup=0;  
    void set(int n) {backup=contenuto;  
                    contenuto=n;}  
    void restore() {contenuto=backup;}  
}
```

Ereditarietà da Object (overriding di metodi)



```
class Complex {  
    double re,im;  
    ...  
}
```

```
Complex c = new Complex(1.5, 2.4);  
System.out.println(c); // c viene convertito in stringa con il metodo  
                        // toString() definito in Object.  
                        // Si ottiene (implementazione di Object):  
                        // Complex@.....
```

MA se si ridefinisce il metodo *toString()* Con la stampa si ottiene:

1.5 + i 2.4

```
class Complex {  
    ...  
    String toString() { return(re + " + i " + im); }  
}
```

Overriding: come estendere un metodo - I



Spesso un metodo di una sottoclasse definito per *overriding* non ridefinisce completamente il metodo con lo stesso nome della sua sopraclasse, ma lo estende soltanto. Esempio:

```
class Finestra {  
    Rettangolo r; ....  
    void disegnaCornice() {...}  
    void disegnaContenuto() {...}  
}  
class FinestraConTitolo extends Finestra {  
    String titolo;  
    void disegnaCornice() { ... disegna la cornice con il titolo ...}  
}
```

Il metodo *disegnaCornice()* di *FinestraConTitolo* estende il metodo *disegnaCornice()* di *Finestra* con il codice per disegnare il titolo.

Overriding: come estendere un metodo - II



Per non duplicare il codice, si può far riferimento ad un metodo della sopraclasse con lo stesso nome mediante la notazione **super**. Ridefiniamo il metodo `disegnaCornice()` in modo incrementale:

```
class FinestraConTitolo extends Finestra {  
    String titolo;  
    void disegnaCornice() {  
        super.disegnaCornice(); ←  
        ... codice aggiuntivo per disegnare il titolo...  
    }  
}
```

super.disegnaCornice() chiama il metodo `disegnaCornice()` della sopraclasse (che altrimenti non sarebbe visibile).



super

```
class FinestraConTitolo extends Finestra {  
    String titolo;  
    void disegnaCornice() {  
        super.disegnaCornice();  
        ... codice aggiuntivo per disegnare il titolo ...  
    }  
}
```

Cosa accadrebbe se non ci fosse **super**?



Visibilità dei membri delle classi

Ad ogni membro di una classe (variabile, metodo, ...) può essere associata uno *specificatore di accesso*:

- **private:** visibile solo dalla *classe stessa*
- *nessuna:* visibile da tutte le classi nello *stesso package*
- **protected:** visibile da tutte le classi nello *stesso package* e da tutte le *sottoclassi* (v. avanti)
- **public:** visibile da *tutti* (parte dell'interfaccia dell'oggetto)



Visibilità dei membri delle classi

Una sottoclasse non può accedere ai campi *privati* della sua sovraclassa

```
class Cella {  
    private int contenuto=0;  
    public int get() {return contenuto;}  
    public void set(int n) {contenuto=n;}  
}
```

```
class CellaConMemoria extends Cella {  
    private int backup=0;  
    public void set(int n) {backup=contenuto; // ERRORE  
                           contenuto=n;}  
    public void restore() {contenuto=backup;}  
}
```



Cella (2)

```
class Cella {  
    private int contenuto=0;  
    public int get() {return contenuto;}  
    public void set(int n) {contenuto=n;}  
}
```

```
class CellaConMemoria extends Cella {  
    private int backup=0;  
    public void set(int n) { backup = get();  
                           super.set(n); } ←  
    public void restore() { super.set(backup); }  
}
```

OK: Si accede al campo *contenuto* con i metodi *get()* e *set()*.



Super può essere usato per chiamare il costruttore della sopraclasse.

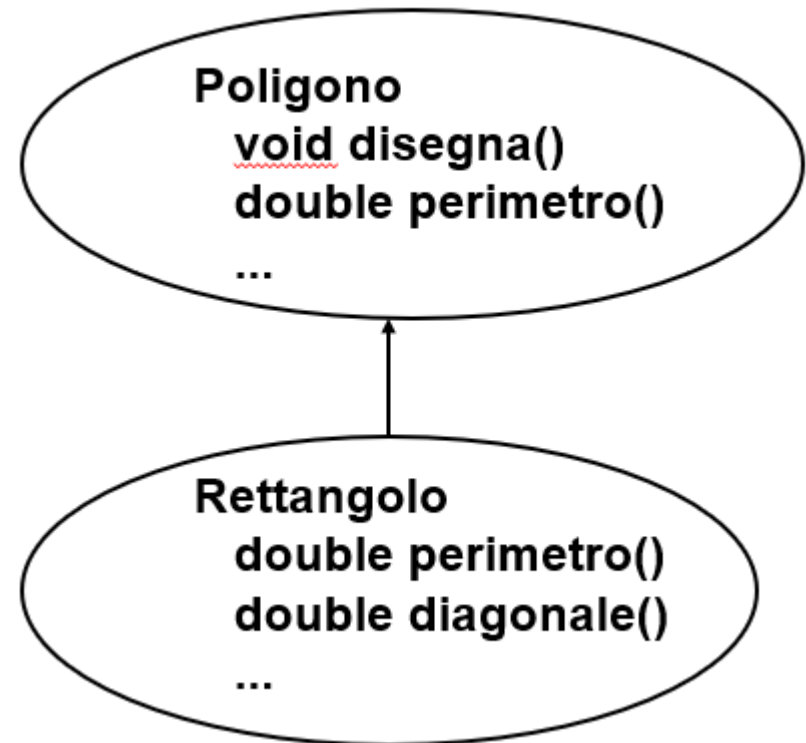
```
class Manager extends Employee {  
  
    public Manager(String n, double s, Day d) {  
        super(n,s,d);  
        secretaryName = "";  
    }  
    .....  
}
```



Upcasting 5

Questo è corretto per il compilatore, ma **quale metodo si esegue?**
Quello di **Poligono** o quello di **Rettangolo**?

```
Poligono p;  
Rettangolo r;  
...  
p = r;  
p.disegna();  
  
p.perimetro();
```



Ringraziamenti



Grazie al Prof. Emerito Alberto Martelli del Dipartimento di Informatica dell'Università di Torino per aver redatto la prima versione di queste slides.