Programmazione III

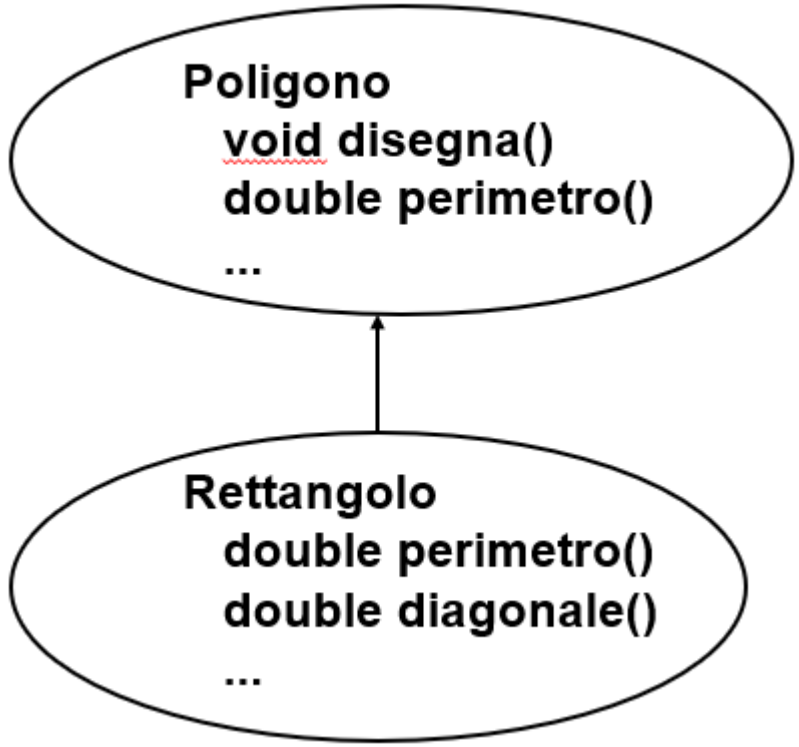
Prof.ssa Liliana Ardissono
Dipartimento di Informatica
Università di Torino

JAVA – Ereditarietà – parte 2

Upcasting 5

Questo codice 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();
```



Binding dinamico – I

p.perimetro();

Si esegue il metodo *perimetro* dell'oggetto a cui *p* fa riferimento in quel momento.

Poligono p = new Poligono();
Rettangolo r = new Rettangolo();

p.perimetro(); *//* si esegue il metodo *perimetro()* di Poligono

p = r;

p.perimetro(); *//* si esegue il metodo *perimetro()* di Rettangolo

Binding dinamico - II

In questo contesto:

Binding: legame fra il nome di un metodo in una invocazione e (codice del) metodo.

obj.m(): quale metodo **m** viene eseguito?

Nei linguaggi tradizionali le chiamate di procedura vengono risolte dal compilatore.

Nei linguaggi ad oggetti (tranne il C++) le chiamate di metodi sono risolte dinamicamente.

BINDING DINAMICO: **la forma di un oggetto determina dinamicamente quale versione di un metodo applicare.**

Binding dinamico - III


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

Nel main:

FinestraConTitolo ft;

...

ft.rinfresca(); ← **chiama il metodo *disegnaCornice()* di *FinestraConTitolo***

Esempio - I

```
class A
{       void stampaTipo() {System.out.println("A");}
        void stampa() {stampaTipo();}
}

class B extends A
{       void stampaTipo() {System.out.println("B");}
}

class EredMetodi {
        public static void main(String[] args)
        {       B b = new B();
                b.stampa(); // stampa "B"
        }
}
```

Esempio - II

Per le variabili di istanza non c'è il *binding dinamico* (esiste solo per i metodi)

```
class A
{       String tipo = "A";
        void stampa() {System.out.println(tipo);}
}       // stampa() prende il valore di “tipo” locale (in A)

class B extends A
{       String tipo = "B";
}

class EredVar {
        public static void main(String[] args)
        {       B b = new B();
                System.out.println(b.tipo);     // stampa “B”
                b.stampa();                     // stampa "A"
        }
}
```

Esempio - III

```
public class Test {
    public static void main(String[] args) {
        B b = new B();
        System.out.println(b.tipo);
        b.stampaTipo();
    }
}

class A {
    String tipo = "A";
    String getTipo() {return tipo;}
    void stampaTipo () {System.out.println(getTipo());}     // STAMPA B B
    //void stampaTipo() {System.out.println(tipo);}         // STAMPA B A
    }

class B extends A {
    String tipo = "B";
    String getTipo() {return tipo;} // getTipo() prende il valore di “tipo” locale (in B)
}
```

Esempio - IV

```
public class Test {
    public static void main(String[] args) {
        B b = new B();
        System.out.println(b.tipo);
        b.stampaTipo();
    }
}
class A {
    String tipo = "A";
    String getTipo() {return tipo;}
    //void stampaTipo () {System.out.println(getTipo());}   // STAMPA B B
    void stampaTipo() {System.out.println(tipo);}            // STAMPA B A
                    // stampaTipo() prende il valore di “tipo” locale (in A)
    }
class B extends A {
    String tipo = "B";
    //String getTipo() {return tipo;}
}
```

Classi "generiche’’

Consideriamo il caso in cui non si dispone di classi parametriche (vedremo più avanti i tipi generici).

Come definire uno Stack «generico»?

```
class Stack {
  ...
  void push(Object x) {...};
  Object pop() {...};
}
```

Non si può specificare il tipo degli elementi della pila: possono essere oggetti qualunque.

```
Stack s; Linea l; Rettangolo r;
...
s.push(l); s.push(r);
```

Downcasting 1

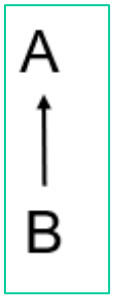
```
class Stack {
   ...
   void push(Object x) {...};
   Object pop() {...};
}
```

Supponiamo di sapere che sulla pila vengono messi solo rettangoli. Come possiamo utilizzare gli oggetti estratti dalla pila?

```
Stack s; Rettangolo r;
...
s.push(new Rettangolo());
```

r = s.pop(); ← Errore di compilazione. La pop() restituisce un *Object*, che è più generale di *Rettangolo*

Downcasting 2



Downcasting: ci si muove da un tipo più generale ad uno più specifico (da un tipo ad un sottotipo)

Se B è un sottotipo di A e se *espr* ha tipo A,

(B) *espr* ha tipo B

L'assegnamento

B x = (B) *espr* è corretto per il compilatore

(B) *espr* può dare errore a *run time*, se l'oggetto ottenuto valutando *espr* non ha tipo B.

Downcasting 3

```
class Stack {
   ...
   void push(Object x) {...};
   Object pop() {...};
}

...
Stack s; Rettangolo r;
...
s.push(new Rettangolo());
```

r = (Rettangolo) s.pop(); // Accettato dal compilatore.

Controllo a run-time. Quando si esegue questa istruzione si controlla che l'oggetto restituito da *pop()* sia veramente un Rettangolo.

Riuso del software

La programmazione ad oggetti consente di utilizzare classi già esistenti per produrre nuovo software:

Uso

Un oggetto comunica con oggetti di altre classi

Contenimento (Part-of).

Si definiscono nuove classi i cui oggetti sono composti di oggetti di classi già esistenti.

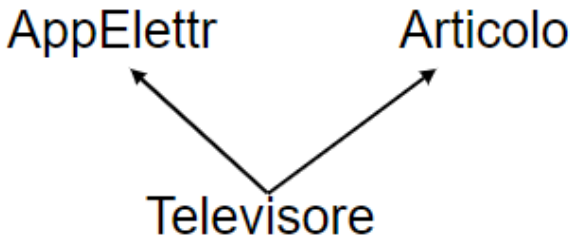
Ereditarietà (Is-a).

Favorisce lo sviluppo incrementale, estendendo classi già esistenti.

Contenimento (Part-of)

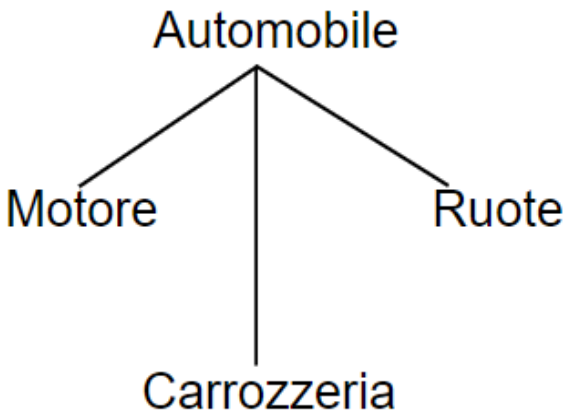
```
class Automobile {
   int lunghezza;
   Motore motore;
   Ruota[] ruote;
   ... }
class motore {
   int cilindrata;
   ... }
class Ruota {
   double pressione;
   int diametro;
   ... }

Automobile miaAuto = new Automobile();
...
miaAuto.motore.cilindrata;
miaAuto.ruote[1].pressione;
```

Ereditarietà vs contenimento



Televisore **è un** apparecchio elettrico e **è un** articolo.

La sottoclasse eredita il comportamento di altre classi.



L'automobile **ha un** motore, una carrozzeria, ...
Il Motore non eredita il comportamento dell'Automobile, ne fa parte (come componente)

Programmare con l'ereditarietà - I

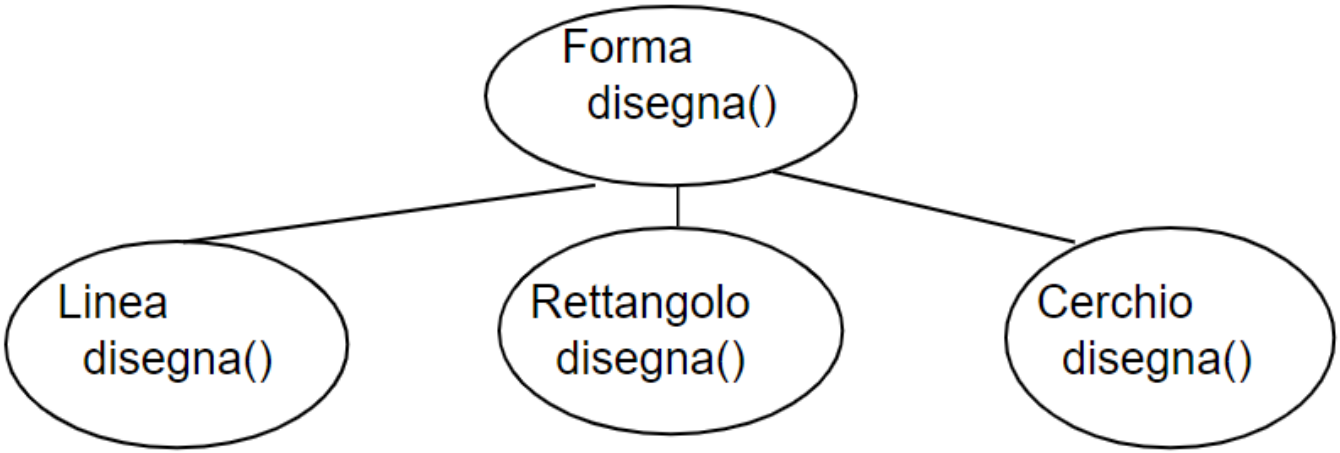
Si consideri una figura composta di diverse forme geometriche (linee, rettangoli, cerchi, …).

Nella programmazione tradizionale, una procedura per disegnare la figura dovrebbe considerare tutti i casi:

```
void disegna(figura f) {
   for ogni forma S in f
      switch(S.genere)
         case LINEA: disegnaLinea()
         case RETTANGOLO: disegnaRettangolo()
         case CERCHIO: disegnaCerchio()
```

Se si aggiunge la forma TRIANGOLO occorre modificare la procedura *disegna()*.

Programmare con l'ereditarietà – II



Con l’ereditarietà (binding dinamico) invece possiamo definire le classi che realizzano le diverse forme geometriche come sottoclassi di una sola classe **Forma**.

Programmare con l'ereditarietà – II

```
class Forma {
        void disegna() {.....}}

class Linea extends Forma {
        void disegna() {.....}}

class Rettangolo extends Forma {
        void disegna() {.....}}

class Cerchio extends Forma {
        void disegna() {.....}}
```

Ciascuna sottoclasse fa overriding del metodo disegna().

Programmare con l'ereditarietà – IV

```
class Figura {
   Forma[] s;     //s è una figura è inplementata come un array di Forme

   void disegnaFigura() {
      for (int i=0; i<s.length; i++)
         s[i].disegna();
   }
...
```

L'istruzione **s[i].disegna()** è corretta per il compilatore perché la classe **Forma** possiede il metodo **disegna()**, che è ridefinito da ogni sottoclasse.

Grazie al *binding dinamico*, quando si esegue il **for** verrà sempre eseguito il metodo **disegna()** della specifica forma geometrica.

Programmare con l'ereditarietà - V

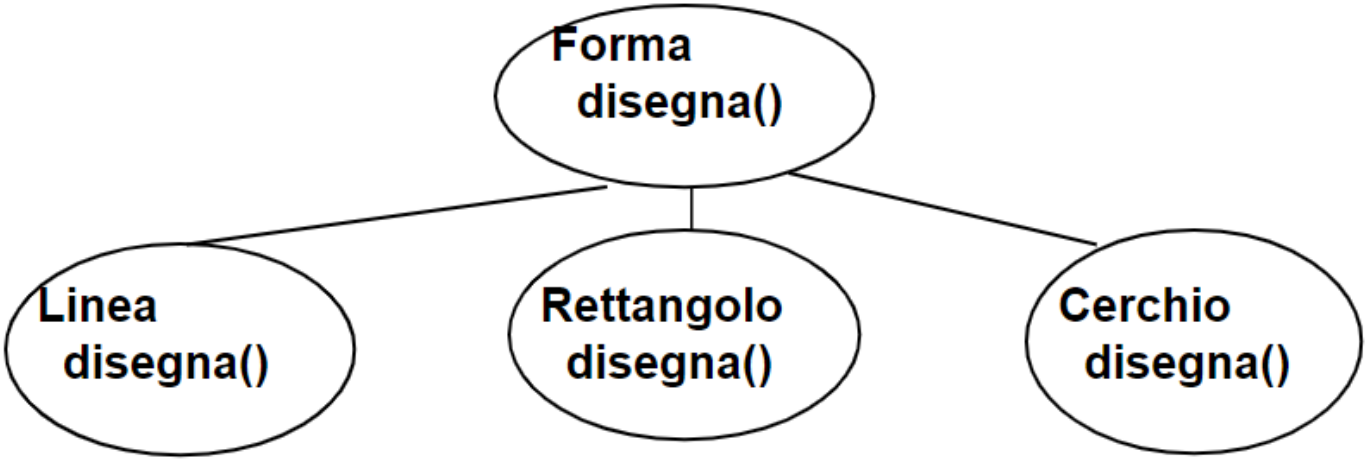
Se si aggiunge la forma *Triangolo*, è sufficiente definire una nuova sottoclasse di *Forma* con il metodo *disegna()*:

```
class Triangolo extends Forma {
   ...
   void disegna(){...}   }
```

La classe *Figura* non viene modificata.

Il binding dinamico fa sì che, se la figura contiene un triangolo, venga chiamato il metodo per disegnare un triangolo.

Classi astratte - I



In un programma non ci si aspetta di chiamare il metodo *disegna()* della classe *Forma*, né di usare oggetti di questa classe.

La classe *Forma* serve solo per avere una **interfaccia comune**, contenente il metodo *disegna()*, per le varie forme specifiche. Quindi potremmo NON implementare il metodo disegna di Forma… di qui le classi astratte.

Classi astratte - II

Le classi astratte sono classi che includono almeno un metodo non implementato: si lascia l'implementazione alle sottoclassi che verranno definite.

I metodi non implementati sono dichiarati **abstract**.

Le classi astratte possono avere il costruttore per poter inizializzare i membri privati della classe.
Tuttavia, NON possono essere istanziate perché non hanno la definizione completa che serve per creare gli oggetti
→ la new non è permessa.

Classi astratte – esempio - I

```
abstract class Forma {
   abstract void disegna();   ← Non c'è l'implementazione del metodo

}

class Linea extends Forma {
   void disegna() { ......}
}

....
Forma f = new Forma();  ← Errore di compilazione. Non si possono
                                creare oggetti di una classe astratta.

Forma f = new Linea();

f.disegna();
```

Classi astratte – esempio - II

```
abstract class Forma {
    private String nome;
    public Forma(String nome) {
        this.nome = nome;
    }
    public String getNome() {
        return nome;
    }
    abstract public void disegna();
}

class Rettangolo extends Forma {
    public Rettangolo() {
        super("Rettangolo");
    }
    public void disegna() {
        System.out.println("Disegno un rettangolo");
    }
}
```

Classi astratte – esempio - III

```
class Linea extends Forma {
    public Linea() {
        super("Linea");
    }
    public void disegna() {
        System.out.println("Disegno una linea");
    }
}
public class FormaApp {
    public static void main(String[] args) {
        Rettangolo r = new Rettangolo();
        Linea l = new Linea();
        r.disegna();  // stampa «Disegno un rettangolo»
        l.disegna();  // stampa «Disegno una linea»
    }
}
```

Interface e classi astratte

Le **classi astratte** possono essere miste, ossia possono contenere anche metodi non astratti.

Con l'ereditarietà singola di Java, una classe può essere sottoclasse solo di una classe astratta → non si possono ereditare metodi da più sopraclassi. Ma talvolta questo serve!

Le **interface** non sono soggette al vincolo della struttura gerarchica ad albero.

Una classe può implementare più di una interfaccia (qualche analogia con *ereditarietà multipla*).

Interface e Classi di libreria – esempio - I

Supponiamo di voler ordinare liste di oggetti di tipo Rettangolo.

Le librerie di Java forniscono l'interface Comparable che specifica la firma del metodo compareTo() per comparare l'oggetto su cui viene invocato con un oggetto passato per parametro.

compareTo restituisce un intero che indica quale dei due oggetti viene prima dell'altro secondo il criterio di ordinamento da applicare.

```
public interface Comparable {
   public int compareTo(Object b);
}
```

Le librerie di Java forniscono anche la classe **Arrays** che contiene un metodo **sort()** per ordinare un array di oggetti **Comparable**.

Interface e Classi di libreria – Esempio - II

Per confrontare due rettangoli, implementare l'interface:

```
class Rettangolo extends Poligono implements Comparable {
        int altezza, larghezza;
        ....
        public int compareTo(Object ob) {
                Rettangolo r = (Rettangolo)ob;
                return altezza - r.altezza;
        }
}
```

e nel main dell'applicazione:

```
Rettangolo[] a = new Rettangolo[10];
// ... Inizializzazione dell'array di rettangoli …
Arrays.sort(a);
...
```

Interfacce multiple

```
interface A {
   void metodA();
}
```

```
interface B {
   void metodB();
}
```

```
class C implements A,B {
   void metodA() {System.out.println("sono A");}
   void metodB() {System.out.println("sono B");}

   public static void main(String[] args) {
      A a = new C();          //l'oggetto è visto come un A
      B b = new C();          //l'oggetto è visto come un B
      a.metodA();             // stampa A
      b.metodB();             // stampa B
      //a.metodB();           //errore: A non offre metodB()
            ((C)a).metodoB(); // ma con downcast compilazione OK
                              // stampa B
   }}
```

Ringraziamenti

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