Ereditarietà e Polimorfismo

Emilio Di Giacomo e Walter Didimo

Obiettivo

- In questa lezione introdurremo i concetti di ereditarietà e polimorfismo
- Essi rappresentano una delle caratteristiche più importanti della programmazione orientata agli oggetti
- Vedremo i costrutti Java per realizzarli
- Mostreremo degli esempi che ce ne faranno comprendere l'utilità

Estensione di classi

- Finora abbiamo sempre definito una classe partendo da zero ed in maniera indipendente da altre classi
- L'estensione di classi ci permette di definire una classe *C2* "estendendo" il comportamento di una classe *C1*
  - *C2* viene chiamata classe estesa o classe derivata o sotto-classe
  - *C1* viene chiamata classe base o super-classe
- Sintassi:

```
<modificatore> class <nome classe estesa> extends
   <nome classe base>
```

Estensione di classi: effetti

- Se *C2* è definita estendendo *C1*, tutti i membri e le variabili di *C1* sono membri anche di *C2* (senza ridefinirli esplicitamente)
  - si dice che *C2* eredita i membri di *C1*
- Ogni istanza di *C2* è anche istanza di *C1*, cioè dove ci si aspetta un oggetto di tipo *C1* si può usare un oggetto di tipo *C2*
  - *C2* è un sotto-tipo di *C1*
  - *C1* è un super-tipo di *C2*

Estensione di classi: esempio

- Immaginiamo di voler scrivere un programma per gestire dati di natura sportiva
- Immaginiamo di definire una classe *Atleta* per rappresentare gli atleti
- La classe avrà i campi:
  - *nome*
  - *nazione*
- e i metodi:
  - *getNome()*
  - *getNazione()*
  - *toString()*

La classe Atleta

```
public class Atleta{
   private String nome; //nome dell'atleta
   private String nazione; // nazione dell'atleta

   /* crea un Atleta con nome e nazione specificati */
   public Atleta(String nome, String nazione){
        this.nome = nome;
        this.nazione = nazione;
   }

   /* restituisce il nome dell'Atleta */
   public String getNome(){
        return this.nome;
   }
   ...
}
```

La classe Atleta

```
public class Atleta{
   ...
   /* restituisce la nazione dell'Atleta */
   public String getNazione(){
       return this.nazione;
   }

   /* restituisce una descrizione testuale dell'Atleta */
   public String toString(){
       return "Nome: "+this.nome+", nazione:
   "+this.nazione;
   }
}
```

Estendiamo la classe Atleta

- Supponiamo di voler rappresentare tipologie di atleti specifici
  - ad esempio tennisti o calciatori
- Per ogni tipologia specifica ci interessano informazioni aggiuntive
  - per i tennisti la posizione nel ranking ATP
  - per i calciatori il ruolo
- Possiamo definire la classi *Tennista* e *Calciatore* estendendo la classe *Atleta*

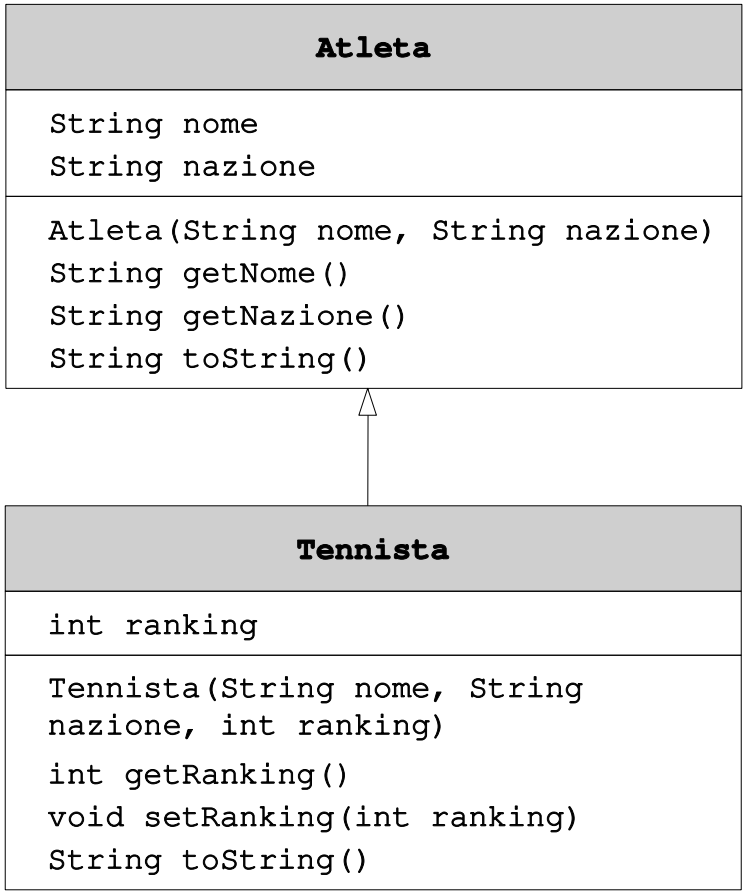
La classe Tennista

```
public class Tennista extends Atleta{
   private int ranking;

   /* costruisce un oggetto Tennista con nome, nazione e ranking
   specificati*/
   public Tennista(String nome, String nazione, int ranking){
        super(nome, nazione);
        this.ranking = ranking;
   }

   /* restituisce il ranking del Tennista*/
   public int getRanking(){
        return this.ranking;
   }
   ...
}
```

La classe Tennista

```
public class Tennista extends Atleta{
   ...
   /* modifica il ranking del Tennista*/
   public void setRanking(int ranking){
        this.ranking = ranking;
   }

   /* restituisce una descrizione testuale del Tennista*/
   public String toString(){
        return "Nome: "+this.getNome()+
               ", nazione: "+this.getNazione()+
               ", ranking ATP: "+this.ranking;
   }
}
```

Estens. di classi: rappres. grafica



Estensione di classi: costruttori

```
public Tennista(String nome, String nazione, int ranking){
        super(nome, nazione);
        this.ranking = ranking;
   }
```

- La prima istruzione del costruttore invoca il costruttore della super-classe:
  - la prima istruzione di un costruttore di una classe derivata è sempre l'invocazione di un costruttore della classe base
  - se l'invocazione non compare esplicitamente viene invocato il costruttore senza argomenti
    - se esso non esiste si ha un errore

Estensione di classi: costruttori

- L'invocazione del costruttore della super-classe serve ad inizializzare lo stato dell'oggetto relativamente alla parte ereditata:
  - nella caso di *Tennista* il costruttore della super-classe inizializza le variabili *nome* e *nazione* che la classe *Tennista* eredita da *Atleta*
- Le variabili *nome* e *nazione* sono private in *Atleta* e quindi non accessibili da *Tennista*
  - una sotto-classe eredita i membri privati della sua super-classe ma non può accederli
  - torneremo più avanti su questo punto

Il riferimento super

- La parola chiave *super* è analoga alla parola *this* e come essa ha un doppio uso
- Può essere usata per invocare un costruttore della super-classe (come abbiamo già visto)
- Può essere usata per fare riferimento all'oggetto ricevente di un metodo o costruttore
  - *this* denota l'oggetto ricevente visto come istanza della classe derivata
  - *super* denota lo stesso oggetto visto come istanza della classe base

Il riferimento super

- Ad esempio, all'interno della classe *Tennista*
  - *this.toString()* corrisponde all'invocazione, sull'oggetto ricevente, del metodo *toString()* definito nella classe *Tennista*
  - *super.toString()* corrisponde all'invocazione, sull'oggetto ricevente, del metodo *toString()* definito nella classe *Atleta*
- Pertanto le due versioni del metodo *toString()* della classe *Tennista* mostrate nella prossima slide sono equivalenti

Il riferimento super

```
public String toString(){
   return "Nome: "+this.getNome()+
        ", nazione: "+this.getNazione()+
        ", ranking ATP: "+this.ranking;
}
```

```
public String toString(){
   return super.toString()+", ranking ATP: "+this.ranking;
}
```

Overriding di metodi

- Come abbiamo già detto, una classe derivata eredita i metodi definiti nella classe base
- Ad esempio, *Tennista* eredita *getNome()* e *getNazione()*
- Il seguente codice è dunque corretto:

```
Tennista t=new Tennista("Roger Federer", "Svizzera", 1);
System.out.println(t.getNome());
System.out.println(t.getNazione());
```

Overriding di metodi

- Una classe può ridefinire un metodo ereditato se vuole che il metodo presenti un comportamento diverso da quello ereditato
  - Ad esempio, la classe *Tennista* ridefinisce il metodo *toString()*
- Si parla di overriding (o sovrascrittura) del metodo
- Una classe derivata può avere quindi:
  - metodi ereditati (e non sovrascritti)
  - metodi ereditati (e sovrascritti)
  - metodi propri

Adombramento di variabili

- Analogamente ai metodi anche le variabili ereditate possono essere ridefinite
- In questo caso si parla di adombramento (o shadowing) della variabile
  - l'adombramento è in realtà un concetto più generale
- Se ad esempio, nella classe *Tennista* definissimo un variabile *nome*
  - questa adombrerebbe la variabile *nome* di *Atleta*
  - un oggetto *Tennista* le avrebbe entrambe e si potrebbe distinguerle, all'interno di *Tennista*, usando *this* e *super*

Adombramento di variabili

- Mentre la sovrascrittura di metodi è moto utile (vedremo più avanti), l'adombramento di variabili viene usato molto limitatamente
- Esso viene permesso per consentire di modificare una classe base senza invalidare le sotto-classi

Il modificatore protected

- Abbiamo visto che una classe estesa eredita i membri privati della classe base ma non può accedere ad essi
- Ciò è coerente con la semantica del modificatore *private*:
  - se la classe base potesse accedere a tali membri, l'ereditarietà sarebbe un modo per aggirare la protezione definita dal modificatore *private*
- In alcuni casi però si vuole consentire l'accesso ai membri ereditati senza renderli però pubblici

Il modificatore protected

- Oltre ai modificatori *public*, *private* e al modificatore di default esiste il modificatore *protected*
- Un membro definito *protected* è accessibile alla classe in cui è definito, a tutte le classi dello stesso package e a tutte le sue sotto-classi definite in altri package

Modificatori: Riassunto

	Classe	Package	Sotto-classe	Ovunque
private	sì	no	no	no
default	sì	sì	no	no
protected	sì	sì	sì	no
public	sì	sì	sì	sì

Ereditarietà: ulteriori commenti

- Una sotto-classe può essere usata come classe base da un'altra classe derivata
  - ad esempio la classe *Calciatore* (che estende *Atleta*) potrebbe essere estesa dalla classe *Portiere* o *Attaccante*
  - In questo caso un'istanza di *Portiere* e *Attaccante* è un'istanza di *Calciatore* e anche di *Atleta*

Ereditarietà: ulteriori commenti

- In Java non è ammessa l'ereditarietà multipla, cioè la possibilità che una classe estenda due o più classi base
- Ad esempio, si potrebbe immaginare di definire la classe *Bicicletta* come estensione delle due classi *MezzoDiTrasporto* e *AttrezzoSportivo*
- Poiché l'ereditarietà multipla può creare situazioni ambigue, essa non viene ammessa

Classi non estendibili

- È possibile definire una classe in maniera che non sia possibile estenderla
- Per farlo bisogna utilizzare il modificatore *final* nella definizione della classe

```
public final class Inestendibile{
    ...
}
```

- Il modificatore *final* può essere applicato anche ad un metodo
  - un metodo *final* non può essere sovrascritto

Polimorfismo

Istanze e tipi

- Abbiamo detto che un'istanza di una classe derivata è anche un'istanza della classe base
  - ad esempio, un'istanza di *Tennista* è anche un'istanza di *Atleta*
- Ciò significa che in qualunque punto ci si aspetta un'istanza della classe base si può utilizzare un'istanza della classe derivata
- La seguente istruzione è quindi corretta

```
Atleta a = new Tennista("Roger Federer",
                                    "Svizzera", 1);
```

Istanze e tipi

- Data la seguente istruzione, quali metodi è possibile invocare su *a* ?

```
Atleta a = new Tennista("Roger Federer",
                                    "Svizzera", 1);
```

- I metodi che è possibile invocare su *a* sono tutti e soli i metodi definiti nella classe *Atleta*
- Quindi la seconda delle seguenti istruzioni è errata

```
System.out.println(a.getNazione()); // corretto
System.out.println(a.getRanking()); // errore!!!!
```

Istanze e tipi

- Perché il comportamento che è possibile chiedere ad un oggetto è stabilito sulla base del tipo della variabile *a* e non in base al tipo effettivo dell'oggetto da essa referenziato?
- La verifica che le invocazioni *a.getNazione()* e *a.getRanking()* siano lecite avviene in fase di compilazione
- Al momento della compilazione non è noto il tipo dell'oggetto che verrà referenziato da *a*:
  - potrebbe dipendere dall'esecuzione stessa del programma

Istanze e tipi

- In fase di compilazione quindi non è possibile tenere conto del tipo effettivo degli oggetti referenziati da *a*
- D'altra parte qualunque sia l'oggetto referenziato esso sarà di tipo *Atleta* o di una sua sotto-classe
- Quindi esso sarà in grado di eseguire i metodi definiti nella classe *Atleta*
- Capiamo meglio quanto detto con un altro esempio

Istanze e tipi

- Consideriamo il seguente metodo:

```
public static void stampa(Atleta a){
        System.out.println(a.toString());
}
```

- Si può invocare il metodo passandogli un oggetto istanza di *Atleta* o di una sua sotto-classe
- Ad ogni invocazione potrà essere passato un oggetto di tipo diverso
- In fase di compilazione non c'è modo di sapere il tipo dell'oggetto passato
- L'unica certezza è che sarà dotato dei metodi della classe *Atleta*

Istanze e tipi

- Consideriamo adesso il seguente codice:

```
Atleta a = new Tennista("Roger Federer",
                                "Svizzera", 1);
System.out.println(a.toString());
```

- A seguito dell'invocazione *a.toString()*, quale metodo verrà eseguito? Quello definito nella classe *Atleta* o quello della classe *Tennista*?

```
Nome: Roger Federer, nazione: Svizzera, ranking
ATP: 1
```

- Viene eseguito il metodo della classe *Tennista*

Binding dinamico

- Il metodo da eseguire viene stabilito, *a tempo di esecuzione*, sulla base del tipo effettivo dell'oggetto su cui il metodo viene invocato
  - nel nostro esempio il tipo *Tennista*
- Tale fenomeno va sotto il nome di binding dinamico

Binding dinamico

- Per binding si intende il processo in base al quale si stabilisce il metodo da eseguire a fronte di un'invocazione:
- Se il binding avviene a tempo di compilazione si parla di binding statico
- Se il binding avviene a tempo di esecuzione si parla di binding dinamico

Polimorfismo

- Il binding dinamico e la possibilità di assegnare ad una variabile di un certo tipo riferimenti ad oggetti di un sotto-tipo, permette di scrivere codice che presenta comportamenti diversi a seconda dei tipi effettivi degli oggetti assegnati alle variabili.
- A tale caratteristica si dà il nome di polimorfismo.

Polimorfismo: esempio

- Riconsideriamo il metodo *stampa* precedente

```
public static void stampa(Atleta a){
    System.out.println(a.toString());
}
```

- Consideriamo adesso il seguente codice:

```
Atleta a = new Atleta("Usain Bolt", "Jamaica");
Tennista t = new Tennista("Roger Federer",
                          "Svizzera", 1);
C.stampa(a);
C.stampa(t);
```

Polimorfismo: esempio

- Il codice precedente produce il seguente output:

```
Nome: Usain Bolt, nazione: Jamaica
Nome: Roger Federer, nazione: Svizzera, ranking ATP: 1
```

- Le due invocazioni del metodo *stampa* producono due output diversi grazie al binding dinamico:
  - la prima invocazione esegue il metodo *toString()* di *Atleta*
  - la seconda quello di *Tennista*

Polimorfismo

- Il polimorfismo è una delle caratteristiche più importanti della programmazione orientata agli oggetti
- Grazie al polimorfismo possiamo scrivere codice facilmente estendibile:
  - si scrive un programma facendo riferimento al comportamento definito da una super-classe (o, come vedremo in seguito, da una interface)
  - si scrivono poi varie sotto-classi che forniscono implementazioni diverse di tale comportamento

Polimorfismo

- In questo modo, le sotto-classi possono essere utilizzate per *personalizzare* e *modificare* il comportamento del programma senza bisogno di modificarne il codice:
  - basta fornire implementazioni diverse dei metodi definiti nell'interfaccia comune
- Vediamo quanto detto con un esempio

Un problema

- Supponiamo di voler realizzare un programma per gestire il magazzino di un negozio di elettrodomestici (TV, Lavatrici, ecc.)
- Vogliamo realizzare una classe che ci permetta di memorizzare tutti gli elettrodomestici nel magazzino, permettendoci di:
  - aggiungere un elettrodomestico al magazzino
  - stampare la lista di elettrodomestici presenti in magazzino

Ulteriori dettagli

- Tutti gli elettrodomestici sono caratterizzati da:
  - la marca
  - il prezzo
- le operazioni di interesse sono:
  - chiedere la marca
  - chiedere il prezzo
  - modificare il prezzo
  - chiedere una descrizione dell'articolo

Ulteriori dettagli

- Esistono degli elettrodomestici per cui interessano informazioni aggiuntive:
  - ad esempio, il numero di pollici per i televisori, la capacità di carico per le lavatrici, ecc.
- Ci interessa anche avere dei metodi per modificare e leggere tali attributi
- Inoltre nella descrizione di tali elettrodomestici dovrebbero comparire anche le informazioni aggiuntive

Proviamo a risolvere il problema

- Una prima idea molto semplice potrebbe essere quella di realizzare:
  - Una classe *Elettrodomestico* per rappresentare gli elettrodomestici generici
  - Una classe per ogni elettrodomestico specifico (*Televisore*, *Lavatrice*, ecc.)
  - La classe *Magazzino* per gestire il magazzino

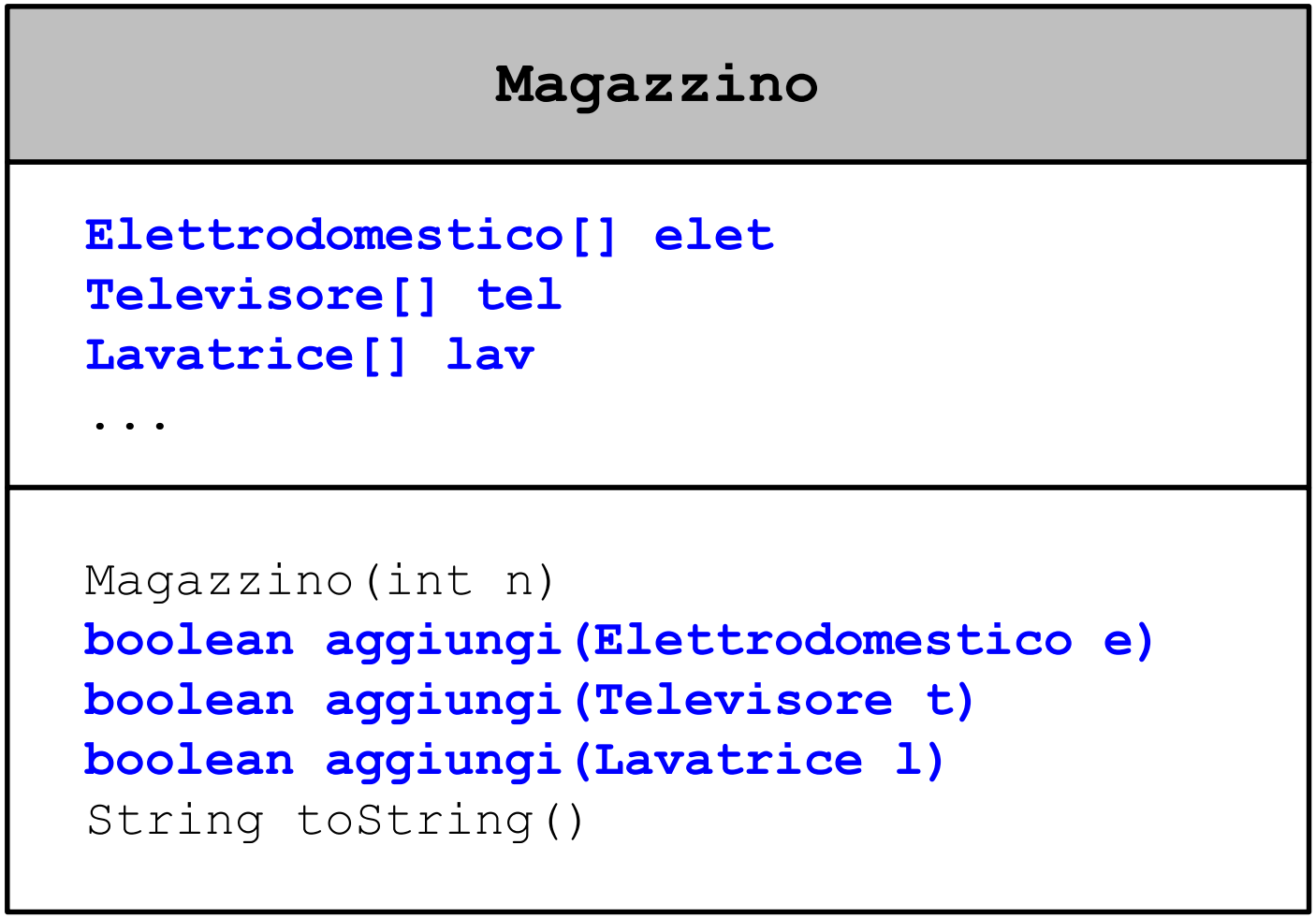
Proviamo a risolvere il problema

- Se non usiamo l'ereditarietà, le classi *Elettrodomestico*, *Televisore*, e *Lavatrice* vengono definite indipendentemente l'una dall'altra
- Per quanto riguarda la classe *Magazzino*, ogni sua istanza dovrà memorizzare un insieme di elettrodomestici, ad esempio in un array
- Di che tipo dovrà essere questo array?

Proviamo a risolvere il problema

- Poiché vogliamo memorizzare oggetti di tre tipi diversi (*Elettrodomestico*, *Televisore* e *Lavatrice*), non è possibile utilizzare un unico array
- Analogamente, il metodo per aggiungere un elettrodomestico al magazzino dovrà ricevere come parametro l'elettrodomestico da aggiungere
- Visto che l'oggetto da aggiungere potrà essere di uno tra tre tipi diversi, non possiamo avere un unico metodo per l'aggiunta di articoli al magazzino

La classe Magazzino



È una buona soluzione?

- La soluzione vista presenta alcuni svantaggi:
  - abbiamo bisogno di tre array
  - abbiamo bisogno di tre metodi *aggiungi*
- Ma il problema più grave è che se volessimo aggiungere un nuovo tipo di elettrodomestico (ad esempio il frigorifero) dovremmo modificare la classe *Magazzino*:
  - aggiungendo un nuovo array
  - aggiungendo un nuovo metodo *aggiungi*
- Tutto ciò dovrebbe essere ripetuto per ogni nuovo tipo di elettrodomestico da aggiungere

Qual è il problema?

- La difficoltà nasce perché abbiamo due esigenze contrastanti.
- Da un lato, vorremmo ignorare le differenze tra i diversi tipi di elettrodomestici e considerarli tutti come istanze del tipo *Elettrodomestico*
  - la classe *Magazzino* avrebbe un solo array e un solo metodo *aggiungi*
- D'altra parte ci interessano le specificità di alcuni tipi di elettrodomestici per i quali vogliamo memorizzare alcune informazioni specifiche e avere descrizioni più dettagliate.

Usiamo l'ereditarietà

- Possiamo superare i problemi visti sfruttando il *polimorfismo*
- Definiamo le classi *Televisore* e *Lavatrice* estendendo la classe *Elettrodomestico*
  - in effetti i televisori e le lavatrici sono casi particolari di elettrodomestici

La classe Elettrodomestico

```
public class Elettrodomestico{
   private String marca;
   private double prezzo;

   /* costruisce un Elettrodomestico con marca e
   prezzo dati */
   public Elettrodomestico(String marca, double
                                          prezzo){
       this. marca = marca;
       this. prezzo = prezzo;
   }
   ...
}
```

La classe Elettrodomestico

```
public class Elettrodomestico{
   ...
   /* restituisce la marca dell'Elettrodomestico */
   public String getMarca(){
       return marca;
   }

   /* restituisce il prezzo dell'Elettrodomestico */
   public double getPrezzo(){
       return prezzo;
   }
   ...
}
```

La classe Elettrodomestico

```
public class Elettrodomestico{
   ...
   /* modifica il prezzo dell'Elettrodomestico */
   public void setPrezzo(int newprezzo){
       this.prezzo = newprezzo;
   }

   /* restituisce una descrizione testuale
   dell'Elettrodomestico */
   public String toString(){
       return  "Elettrodomestico."+
             " Marca: "+this.marca+
             ", prezzo: euro "+this.prezzo+".";
   }
}
```

La classe Televisore

```
public class Televisore extends Elettrodomestico{
   private int pollici; //dimensione in pollici
   private String tecnologia; //LCD,Plasma,LED,ecc.

   /* crea un oggetto Televisore con marca, prezzo,
   pollici e tecnologia specificati */
   public Televisore(String marca, double prezzo, int
                              pollici, String tecnologia){
       super(marca, prezzo);
       this.pollici = pollici;
       this.tecnologia = tecnologia;
   }
   ...
}
```

La classe Televisore

```
public class Televisore extends Elettrodomestico{
   ...
   /* restiruisce la dimensione in pollici del
   Televisore*/
   public int getPollici(){
       return pollici;
   }

   /* restiruisce la tecnologia del Televisore */
   public String getTecnologia(){
       return tecnologia;
   }
   ...
}
```

La classe Televisore

```
public class Televisore extends Elettrodomestico{
   ...
   /* restituisce una descrizione testuale del
   Televisore */
   public String toString(){
       return "Televisore."+
              " Marca: "+this.getMarca()+
              ", prezzo: "+this.getPrezzo()+
              ", pollici: "+this.pollici+
              ", tecnologia: "+this.tecnologia+".";
   }
}
```

La classe Lavatrice

```
public class Lavatrice extends Elettrodomestico{
   private int capacita; //capacità di carico
   private String classe; //classe energetica

   /* crea una Lavatrice con marca, prezzo capacità e
   classe energetica */
   public Lavatrice(String marca, double prezzo, int
                              capacita, String classe){
       super(marca, prezzo);
       this.capacita = capacita;
       this.classe = classe;
   }
   ...
}
```

La classe Lavatrice

```
public class Lavatrice extends Elettrodomestico{
   ...
   /* restituisce la capacità di carico della
   Lavatrice */
   public int getCapacita(){
       return capacita;
   }

   /* restituisce la classe energetica della
   Lavatrice */
   public String getClasse(){
       return classe;
   }
   ...
}
```

La classe Lavatrice

```
public class Lavatrice extends Elettrodomestico{
   ...
   /* restituisce una descrizione testuale della
   Lavatrice */
   public String toString(){
       return "Lavatrice. "+
         "Marca: "+this.getMarca()+
         ", prezzo: "+this.getPrezzo()+
         ", capacita\': kg "+this.getCapacita()+
         ", classe energetica: "+this.getClasse()+
         ".";
   }
}
```

Commenti

- Con questa impostazione, la classe *Magazzino* avrà un solo array e un solo metodo *aggiungi(...)*
- Sia gli elementi dell'array che il parametro del metodo *aggiungi(...)* saranno di tipo *Elettrodomestico*

La classe Magazzino

```
public class Magazzino{
   private Elettrodomestico[] elettrodomestici;
   private int count;

   /* crea un Magazzino in grado di ospitare fino a
   dim elettrodomestici */
   public Magazzino(int dim){
       elettrodomestici = new Elettrodomestico[dim];
       count = 0;
   }
   ...
}
```

La classe Magazzino

```
public class Magazzino{
   ...
   /* aggiunge un Elettrodomestico al magazzino */
   public boolean aggiungi(Elettrodomestico e){
       boolean inserito = false;
       if (count<elettrodomestici.length){
             elettrodomestici[count] = e;
             count++;
             inserito = true;
       }
       return inserito;
   }
...
}
```

La classe Magazzino

```
public class Magazzino{
   ...
   /* restituisce una descrizione testuale del
   Magazzino */
   public String toString(){
       String s = "Elenco prodotti in magazzino\n";
       s += "----------------------------\n";
       for (int i = 0; i<count; i++){
             s +=elettrodomestici[i].toString();
             s +="\n\n";
       }
       return s;
   }
}
```

Commenti

- L’array può memorizzare indifferentemente oggetti di tipo *Elettrodomestico*, *Televisore* e *Lavatrice*
- Il metodo *aggiungi* può ricevere oggetti di uno qualunque dei tre tipi.

Commenti

- Il metodo *toString()* della classe *Magazzino* restituisce una descrizione di tutti i prodotti in magazzino
- Per farlo invoca su ogni oggetto nell'array, il corrispondente metodo *toString()*
  - ciò è possibile perché questo metodo è definito nella super-classe
- A tempo di esecuzione, grazie al *binding dinamico*, ogni oggetto memorizzato nell'array eseguirà il metodo *toString()* della propria classe

Commenti

- Gli oggetti di tipo *Elettrodomestico* restituiranno una descrizione sintetica in cui compariranno soltanto marca e prezzo
- Gli oggetti di tipo *Televisore* e *Lavatrice* restituiranno descrizioni più dettagliate
- È possibile aggiungere nuove tipologie di elettrodomestici senza modificare la classe *Magazzino*

Magazzino: esempio d'uso

- Il seguente brano mostra un esempio d'uso della classe *Magazzino*

```
Magazzino m = new Magazzino();
m.aggiungi(new Elettrodomestico("SuperLux",250));
m.aggiungi(new Televisore("TVSplend",400,32,LCD"));
m.aggiungi(new Lavatrice("LavaWash",300,6,"A+",3));
System.out.println(m.toString());
```

Magazzino: esempio d'uso

- L'output del codice precedente è il seguente:

```
Elenco prodotti in magazzino
--------------------
Elettrodomestico. Marca: SuperLux, prezzo: euro
   250.0.
Televisore. Marca: TVSplend, prezzo: 400.0, pollici:
   32, tecnologia: LCD.
Lavatrice. Marca: LavaWash, prezzo: 300.0,
   capacita': kg 6, classe energetica: A+.
```

La classe Object

La classe Object

- Ogni classe che viene definita in Java estende implicitamente la classe *Object*
- Tale classe definisce un super-tipo per tutte le classi Java
- La classe *Object* viene utilizzata quando si vuole scrivere del codice che deve poter operare su un oggetto di qualunque tipo

Esempio di uso della classe Object

```
public class Coppia{
   private Object el1;
   private Object el2;
   public Coppia(Object el1, Object el2){
       this.el1 = el1;
       this.el2 = el2;
   }
   public Object primoElemento(){
       return this.el1;
   }
   public Object secondoElemento(){
       return this.el2;
   }
}
```

Esempio di uso della classe Object

- La classe Coppia permette di rappresentare coppie di oggetti di qualunque tipo
- Esempio:

```
Coppia c1 = new Coppia("primo", "secondo");
Atleta a1 = new Atleta("Usain Bolt", "Giamaica");
Atleta a2 = new Atleta("Asafa Powell", "Giamaica")
Coppia c2 = new Coppia(a1, a2);
```

- L'oggetto *c1* rappresenta una coppia di stringhe
- L'oggetto *c2* rappresenta una coppia di atleti

Ancora sull'uso di Object

- Dato un oggetto di tipo *Coppia* possiamo ottenere ciascuno degli elementi della coppia tramite i metodi *primoElemento()* e *secondoElemento()*
- Gli oggetti restituiti sono però di tipo *Object*

```
String s = c1.primoElemento(); // errato !!!
Atleta a = c2.secondoElemento(); // errato !!!

Object o2 = c2.secondoElemento();
o2.getNazione(); // errato!!
```

Ancora sull'uso di Object

- Per ottenere gli elementi di una coppia come istanze della loro classe dobbiamo effettuare una conversione esplicita (cast esplicito)

```
String s = (String)c1.primoElemento();
Atleta a = (Atleta)c2.secondoElemento();
```

Ancora sull'uso di Object

- Possiamo utilizzare il cast esplicito per trasformare un riferimento di un super-tipo in un riferimento di un sotto-tipo
  - l'oggetto memorizzato nel riferimento deve essere del sotto-tipo
  - altrimenti si ha un errore di tipo *ClassCastException*

```
Atleta a1 = new Tennista("R. Federer","Svizzera",1);
Atleta a2 = new Atleta("U. Bolt", "Giamaica");
Tennista t1 = (Tennista)a1; // corretto
Tennista t2 = (Tennista)a2; // errore !!!
```

Metodi della classe Object

- La classe *Object* definisce anche alcuni metodi che vengono ereditati da ogni classe
- La classe *Object* quindi definisce un comportamento comune a tutti gli oggetti Java
- Tra i metodi della classe *Object* menzioniamo:
  - *String toString()*
  - *boolean equals(Object o)*

Il metodo toString()

- Il metodo *toString()* restituisce una descrizione testuale dell'oggetto
- Questo metodo viene invocato automaticamente tutte le volte che si esegue una concatenazione tra una stringa e un oggetto

```
Atleta a = new Atleta("Usain Bolt", "Giamaica");
System.out.println("Oggetto: "+a);
```

è equivalente a

```
Atleta a = new Atleta("Usain Bolt", "Giamaica");
System.out.println("Oggetto: "+a.toString());
```

Il metodo toString()

- La classe *Object* fornisce un'implementazione per il metodo *toString()*
  - che restituisce una stringa con il nome della classe, il simbolo @ e un codice esadecimale dell'oggetto
- Il metodo *toString()* può essere sovrascritto dalle sotto-classi di *Object* per definire un comportamento diverso
- Ogni volta che dotiamo una classe di un metodo *toString()* stiamo sovrascrivendo il corrispondente metodo della classe *Object*

Il metodo equals(...)

- Il metodo *boolean equals(Object obj)* viene invece utilizzato per stabilire l'uguaglianza tra due oggetti
- Tale metodo è necessario a quelle classi che operano su oggetti di tipo *Object* e hanno bisogno di stabilire l'uguaglianza tra di essi

Il metodo equals: esempio d'uso

- Supponiamo di voler dotare la classe *Coppia* di un metodo *contiene(Object obj)* per verificare se l'oggetto *obj* appartiene alla coppia

```
public boolean contiene(Object obj){
    boolean contiene = false;
    if (el1.equals(obj))
          contiene = true;
    if (el2.equals(obj))
          contiene = true;
    return contiene;
}
```

Il metodo equals(...)

- L'implementazione del metodo *equals* fornita dalla classe *Object* restituisce *true* solo se i due oggetti coincidono (sono cioè lo stesso oggetto)

```
Object o1 = new Object();
Object o2 = o1;
Object o3 = new Object();
System.out.println(o1.equals(o2)); // true
System.out.println(o1.equals(o3)); // false
```

- Il metodo può essere sovrascritto per definire un comportamento diverso

Interface

Interface

- Il costrutto interface è un costrutto offerto da Java per la definizione di un tipo riferimento
- L'interface ha molte analogie con il costrutto classe, ma anche molte differenze
- Come la classe, una interface *I* definisce un tipo riferimento *I*
- L'interface definisce un comportamento per il tipo *I*
  - cioè una serie di metodi la cui esecuzione può essere richiesta ad oggetti del tipo *I*
- È possibile dichiarare variabili di tipo *I* sulle quali è possibile invocare i metodi definiti dall'interface.

Interface

- A differenza di una classe però, una interface non fornisce una implementazione del tipo *I* da essa definito
- Non è quindi possibile istanziare oggetti di tipo *I*
- Se non posso creare oggetti di tipo *I* a che cosa serve l'interface *I*?
- Una interface *I* deve essere implementata da una o più classi, ciascuna delle quali fornisce una realizzazione specifica del comportamento definito dall'interface

Interface

- Una classe *C* che implementa una interface *I* definisce un sotto-tipo di *I*
  - come accade quando si estende una classe
- Un oggetto di tipo *C* è anche un oggetto di tipo *I*
  - su di esso è possibile invocare tutti i metodi definiti nell'interface *I*
- Il metodo effettivamente eseguito viene scelto a tempo di esecuzione in base al tipo effettivo dell'oggetto ricevente (*binding dinamico*)
- Otteniamo quindi un comportamento polimorfico per gli oggetti di tipo *I*

Interface

- Una interface si definisce usando la seguente sintassi.

```
⟨modificat. di accesso⟩ interface ⟨nome interface⟩{
    ⟨dichiarazione di costanti⟩
    ⟨dichiarazione metodi astratti⟩
}
```

- Il modificatore di accesso è sempre *public* (può essere omesso)

Interface

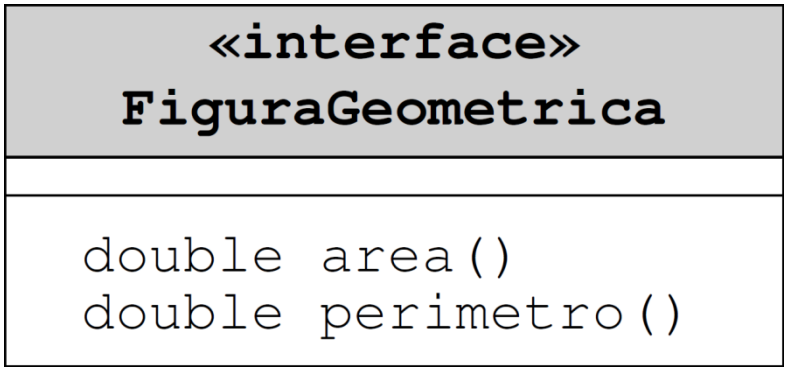
- I metodi dichiarati in una interface sono metodi astratti
  - hanno un prototipo ma non un corpo
  - definiscono un comportamento ma non ne danno una realizzazione
- Sono sempre metodi pubblici e non possono essere statici
  - si può omettere sia *public* che *abstract*

Interface

- In una interface non compaiono né costruttori né variabili
- È invece possibile definire delle costanti, cioè variabili che sono *static* e *final*
  - entrambe possono essere omesse

Interface FiguraGeometrica

```
public interface FiguraGeometrica{
   /* restituisce l'area della FiguraGeometrica*/
   public double area();

   /* restituisce il perimetro della
   FiguraGeometrica*/
   public double perimetro();
}
```

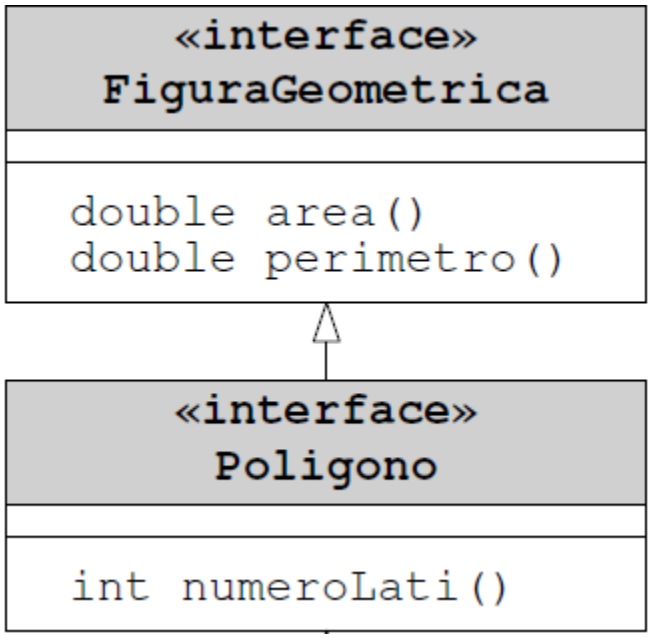


Interface ed ereditarietà

- Una interface *I2* può anche essere definita estendendo una interface *I1*
- *I2* eredita i metodi e le costanti definite in *I1*
- Ad esempio, possiamo definire una interface *Poligono* che estende *FiguraGeometrica*
- ai metodi *area()* e *perimetro()* ereditati da *FiguraGeometrica* viene aggiunto il metodo *numeroLati()*

Interface Poligono

```
public interface Poligono extends FiguraGeometrica{
   /* restituisce il numero di lati del poligono */
   public int numeroLati();
}
```



Implementazione di interface

- Una volta definita una interface, essa viene implementata da una o più classi
- Una classe che implementa una interface fornisce una realizzazione del comportamento definito dall'interface

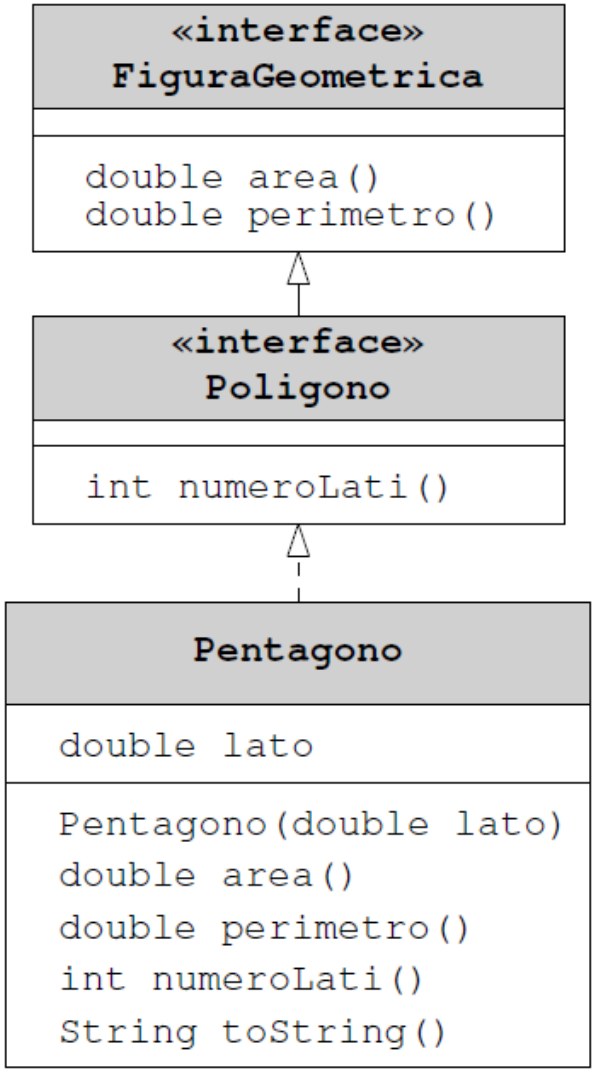
La classe Pentagono

```
public class Pentagono implements Poligono{
  private double lato; //Lunghezza del lato
  private static final numLati = 5; //Numero di lati

  /* costruisce un Pentagono regolare di lato dato*/
  public Pentagono(double lato){
    this.lato = lato;
  }
  /* restituisce l'area del Pentagono */
  public double area(){
    double a = lato/2*Math.tan(2*Math.PI/numLati);
    return a*perimetro();
  }
...
}
```

La classe Pentagono

```
public class Pentagono implements Poligono{
   ...
   /* restituisce il perimetro del Pentagono */
   public double perimetro(){
      return numLati*lato;
   }
   /* restituisce il numero di lati del Pentagono */
   public int numeroLati(){
      return numLati;
   }
   /* restituisce una descrizione del Pentagono */
   public String toString(){
      return "Pentagono di lato:"+lato;
   }
}
```

Rappresentazione grafica



Implementazione di interface

- Una classe può implementare più interface
- È possibile definire una classe estendendone un'altra ed implementando una o più interface
- I seguenti sono tutti esempi leciti di definizioni di classi:

```
public class C implements I1{...}
public class C implements I1, I2, I3{...}
public class C2 extends C1 implements I1, I2{...}
```

Implementazione di interface

- Una classe che implementa una interface *deve* implementarne tutti i metodi oppure deve essere una classe astratta
- Parleremo di classi astratte nel seguito

- Illustriamo adesso con un esempio un caso in cui possiamo beneficiare dell'uso delle interface

Contenimento di forme

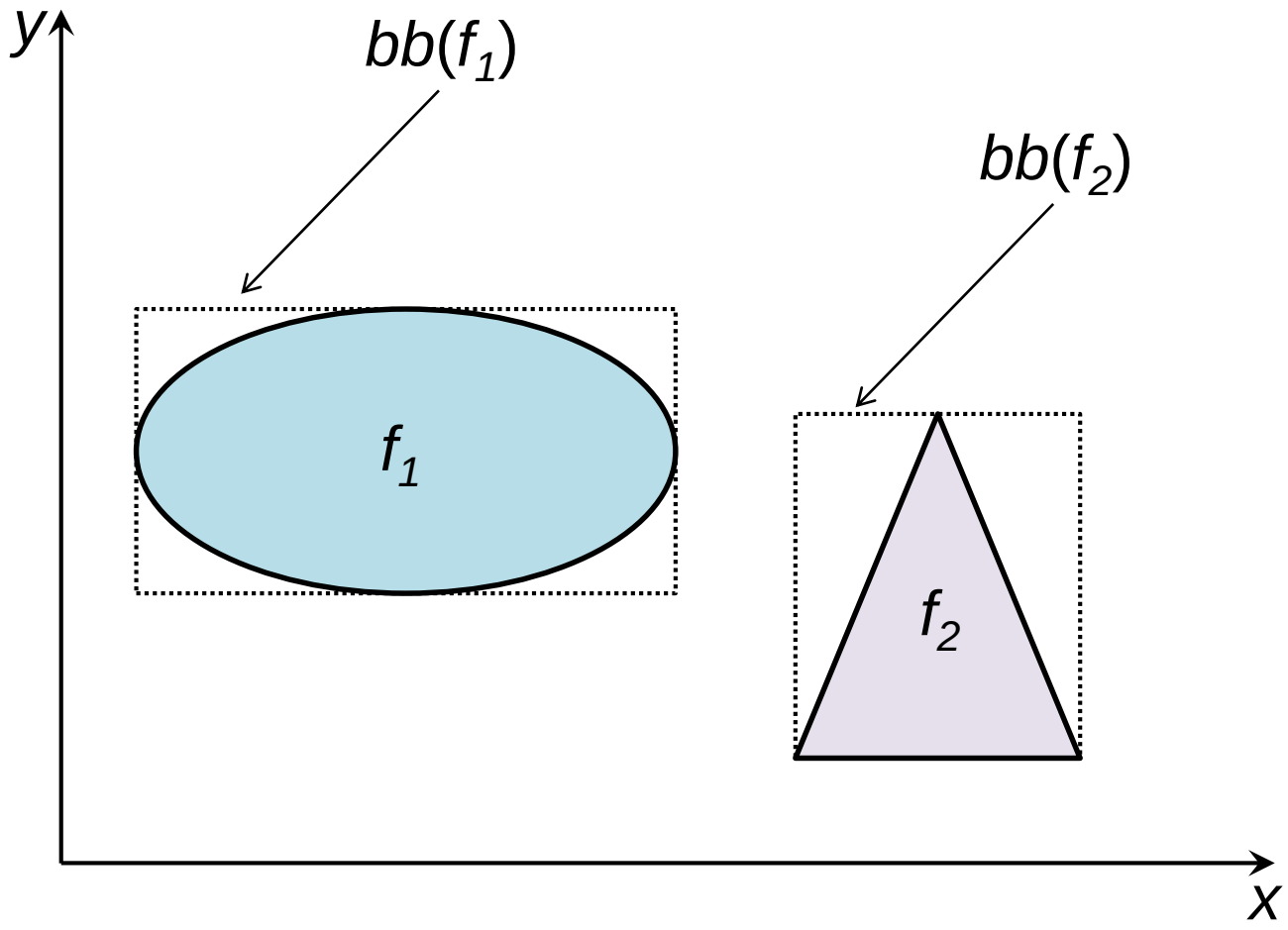
- Supponiamo di voler risolvere il seguente problema, che chiameremo il problema del contenimento di forme:
  - abbiamo un certo numero di forme geometriche (rettangoli, ellissi, poligoni, ecc.) e vogliamo stabilire se esistono coppie di forme $\langle f_1, f_2 \rangle$ tali che f_2 è contenuta in f_1

Definiamo il contenimento

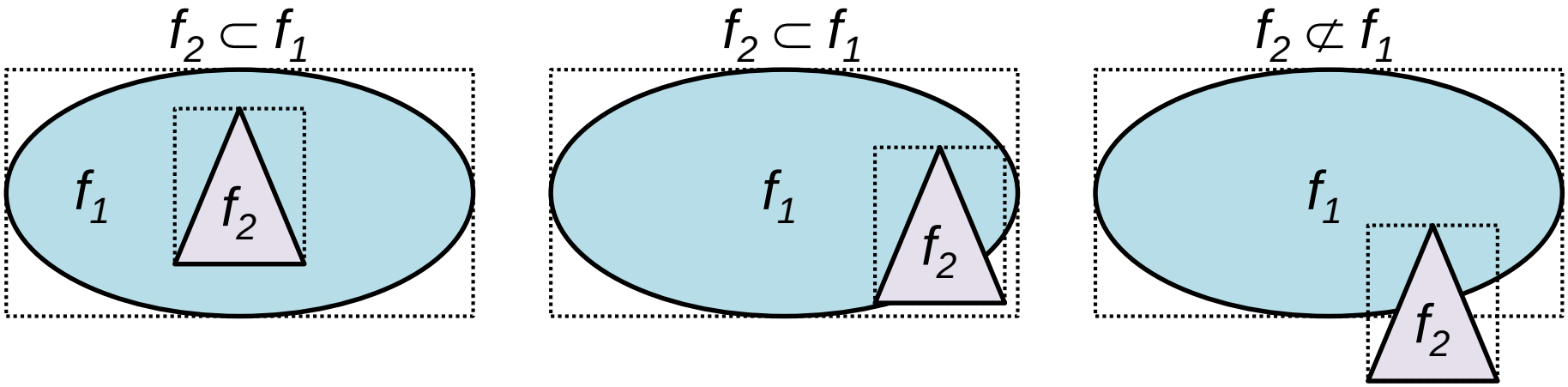
- Data una forma geometrica *f* la bounding box *bb(f)* di *f* è il più piccolo rettangolo con lati paralleli agli assi cartesiani che contiene *f*

- Per semplicità diciamo che una forma f_1 contiene una forma f_2 se la bounding box di f_1 contiene la bounding box di f_2
  - si noti che la nostra definizione di contenimento NON corrisponde al contenimento usuale

Bounding box

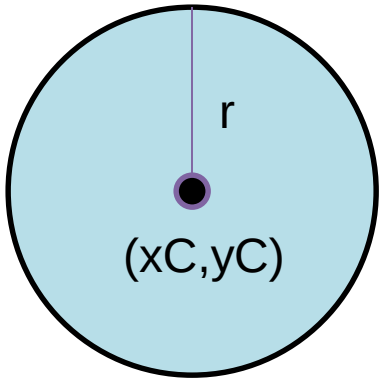


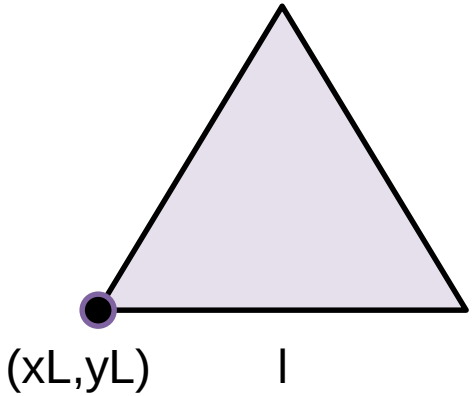
Contenimento di forme



Proviamo a risolvere il problema

- Supponiamo di avere soltanto cerchi e triangoli equilateri
- per i triangoli consideriamo solo triangoli equilateri con un lato orizzontale e con il terzo vertice al di sopra di tale lato





Soluzione 1

- Creiamo una classe *Cerchio* e una classe *TriangoloEquilatero*
- Entrambi le classi saranno dotate di un metodo *getId()* che restituisce un identificativo dell'oggetto su cui viene invocato
- Entrambi le classi avranno dei metodi per determinare la bounding box: *getXMin(), getXMax(), getYMin(), getYMax()*
- Entrambi le classi avranno due metodi per testare il contenimento: *contiene(Cerchio c), contiene (TriangoloEquilatero t)*

La classe Cerchio

Cerchio
• private double raggio • private double xC • private double yC
• Cerchio(double, double, double) • String getId() • double getXMin() • double getXMax () • double getYMin() • double getYMax() • boolean contiene(Cerchio) • boolean contiene(TriangoloEquilatero)

La classe TriangoloEquilatero

TriangoloEquilatero
• private double lato • private double xL • private double yL
• TriangoloEquilatero(double, double, double) • String getId() • double getXMin() • double getXMax () • double getYMin() • double getYMax() • boolean contiene(Cerchio) • boolean contiene(TriangoloEquilatero)

È una buona soluzione?

- Se volessi aggiungere una nuova forma (ad esempio *Rettangolo*) dovrei:
  - Scrivere una nuova classe *Rettangolo*
  - Aggiungere ad ognuna delle classi già scritte un nuovo metodo *contiene(Rettangolo r)*
- In sostanza, l'introduzione di una nuova forma comporta la modifica di tutte le classi rappresentanti le forme precedenti
  - Per passare da 100 a 101 forme oltre ad aggiungere una nuova classe che modelli gli oggetti dell'ultima forma devo anche modificare le precedenti 100 classi

Soluzione 2

- Potremmo pensare di utilizzare l'ereditarietà, creando:
  - una classe *Forma*
  - le classi delle specifiche forme (*Cerchio*, *TriangoloEquilatero*, *Rettangolo*, ecc.) come sottoclassi della classe *Forma*
- In questo modo sarebbe sufficiente un solo metodo *contiene(Forma f)* definito nella classe *Forma*

Una difficoltà

- La soluzione precedente risulta efficace per quanto riguarda la scrittura del metodo *contiene*
- Ci troviamo però di fronte ad una difficoltà: come dovrebbe essere fatta la classe *Forma*?
  - Quali sarebbero le sue variabili di istanza?
  - Come dovrebbero essere implementati i metodi *getXMin()*, *getXMax()*, *getYMin()*, *getYMax()*?

Una difficoltà

- La classe *Forma* ci permette di astrarre rispetto alle varie forme specifiche
  - questo ci permette, ad esempio, di avere un solo metodo *contiene* con parametro di tipo *Forma*
- D'altra parte però la classe *Forma* non può essere realizzata in quanto non c'è modo di rappresentare una forma generica

Di che cosa abbiamo bisogno?

- Abbiamo bisogno di definire il tipo *Forma* in maniera da poterlo usare dove ci faccia comodo:
  - ad esempio come parametro del metodo *contiene*
- D'altra parte vorremo poter definire il tipo *Forma* senza darne una realizzazione:
  - avremo diverse realizzazioni specifiche del tipo *Forma* grazie alle varie classi che realizzano forme specifiche (*Cerchio*, *TriangoloEquilatero*, *Rettangolo*, ecc.)
- È possibile fare una cosa del genere?
- Sì sfruttando il costrutto *interface*

Soluzione 3

- Possiamo risolvere il problema del contenimento di forme creando:
  - una interface *Forma*
  - le classi delle specifiche forme (*Cerchio*, *Triangolo*, *Rettangolo*, ecc.) che implementano l'interface *Forma*
- Nell'interface *Forma* definiremo i metodi:
  - *getId()*
  - *contiene(Forma f)*
  - *getXMin()*
  - *getXMax()*
  - *getYMin()*
  - *getYMax()*

L'interface Forma

```
public interface Forma{
   /* restituisce il cod. identific. della forma */
   public String getId();

   /* restituisce la coord. x minima della forma */
   public double xMin();

   /* restituisce la coord. y minima della forma */
   public double yMin();

   /* restituisce la coord. x massima della forma */
   public double xMax();
   ...
}
```

L'interface Forma

```
public interface Forma{
   ...
   /* restituisce la coord. y massima della forma */
   public double yMax();

   /* restituisce true se questa forma contiene la
   forma f */
   public boolean contiene(Forma f);
}
```

La classe Cerchio

```
public class Cerchio implements Forma{
   private String id; //codice identificativo
   private double raggio; //raggio del cerchio
   private double xC; //coord. x del centro
   private double yC; //coord. y del centro
   /* crea un cerchio con id, raggio e coord. dati */
   public Cerchio(String id, double r, double x,
                                              double y){
       this.id = id;
       this.raggio = r;
       this.xC = x;
       this.yC = y;
   }
...
}
```

La classe Cerchio

```
public class Cerchio implements Forma{
   ...
   /* restituisce il cod. identificat. del cerchio */
   public String getId(){
       return this.id;
   }

   /* restituisce la coord. x minima del cerchio */
   public double xMin(){
       return this.xC-this.raggio;
   }
   ...
}
```

La classe Cerchio

```
public class Cerchio implements Forma{
   ...
   /* restituisce la coord. y minima del cerchio */
   public double yMin(){
       return this.yC-this.raggio;
   }

   /* restituisce la coord. x massima del cerchio */
   public double xMax(){
       return this.xC+this.raggio;
   }
   ...
}
```

La classe Cerchio

```
public class Cerchio implements Forma{
   ...
   /* restituisce la coord. y massima del cerchio */
   public double yMax(){
       return this.yC+this.raggio;
   }
   /* restituisce true se questo cerchio contiene la
   forma f */
   public boolean contiene(Forma f){
       return (this.xMin()<=f.xMin()) &&
              (this.yMin()<=f.yMin()) &&
              (this.xMax()>=f.xMax()) &&
              (this.yMax()>=f.yMax());
   }
}
```

La classe TriangoloEquilatero

```
public class TriangoloEquilatero implements Forma{
   private String id; //codice identificativo
   private double lato; //lunghezza del lato
   private double xL; //coord. x vert. in basso a sx
   private double yL; //coord. y vert. in basso a sx
   /* crea un tr. equi. con id, lato e coord. dati */
   public TriangoloEquilatero(String id, double l,
                              double x, double y){
       this.id = id;
       this.lato = l;
       this.xL = x;
       this.yL = y;
   }
   ...
}
```

La classe TriangoloEquilatero

```
public class TriangoloEquilatero implements Forma{
   ...
   /* restituisce il cod. ident. del tr. equil. */
   public String getId(){
       return this.id;
   }

   /* restituisce la coord. x min. del tr. equil. */
   public double xMin(){
       return this.xL;
   }
   ...
}
```

La classe TriangoloEquilatero

```
public class TriangoloEquilatero implements Forma{
   ...
   /* restituisce la coord. y min. del tr. equil. */
   public double yMin(){
       return this.yL;
   }

   /* restituisce la coord. x max. del tr. equil. */
   public double xMax(){
       return this.xL+this.lato;
   }
   ...
}
```

La classe TriangoloEquilatero

```
public class TriangoloEquilatero implements Forma{
   ...
   /* restituisce la coord. y max. del tr. equil. */
   public double yMax(){
       return this.yL+Math.sqrt(3)*this.lato/2;
   }
   /* restituisce true se questo triangolo equilatero
   contiene la forma f */
   public boolean contiene(Forma f){
       return (this.xMin()<=f.xMin()) &&
              (this.yMin()<=f.yMin()) &&
              (this.xMax()>=f.xMax()) &&
              (this.yMax()>=f.yMax());
   }
}
```

La classe ContenimentoDiForme

```
import fond.io.*;
public class ContenimentoDiForme{
  public static void main(String[] args){
     InputWindow in=new InputWindow();
     OutputWindow out=new OutputWindow();
     int n=in.readInt("Quante forme vuoi inserire?");
     Forma[] forme=new Forma[n];
     out.writeln("Scegli 1 per inserire un cerchio,
                  2 per inserire un triangolo");
     int i=0;
     ...
```

La classe ContenimentoDiForme

```
...
while(i<n){
int opzione=in.readInt("Scegli 1 o 2");
if(opzione==1){
  double raggio,xC,yC;
  raggio=in.readDouble("Raggio?");
  xC=in.readDouble("Coord. x del centro?");
  yC=in.readDouble("Coord. y del centro?");
  forme[i]=new Cerchio("Cerchio
                              "+i,raggio,xC,yC);
  i++;
  }else if(opzione==2){
                ...
        }
}// fine while
```

La classe ContenimentoDiForme

```
   ...
   }else if(opzione==2){
     double lato,xL,yL;
     lato=in.readDouble("Lato?");
     xL=in.readDouble("Coord. x ?");
     yL=in.readDouble("Coord. y ?");
     forme[i]=new  TriangoloEquilatero("Triangolo
                                       "+i,lato,xL,yL);
     i++;
   }
}// fine while
...
```

La classe ContenimentoDiForme

```
  ...
  for(i=0;i<forme.length;i++)
     for(int j=0;j<forme.length;j++){
       if(i!=j){
         if(forme[i].contiene(forme[j]))
          out.writeln("La forma: "+forme[i].getId()+"
              contiene la forma: "+forme[j].getId());
       }
     }
  }
}
```

Classi astratte

Un’ulteriore possibilità

- Nel problema del contenimento di forme abbiamo usato una interface perché non eravamo in grado di scrivere una classe *Forma*:
  - non sapevamo come rappresentare lo stato di una forma generica
  - non potevamo realizzare i metodi *getXMin()*, *getXMax()*, *getYMin()*, *getYMax()*
- D’altra parte avremmo potuto scrivere il metodo *contiene*:
  - tale metodo è indipendente dalla specifica forma
  - infatti esso è identico in tutte le classi che implementano *Forma*

Un’ulteriore possibilità

- In altri termini, pur non potendo creare una superclasse *Forma*, ci farebbe comodo poter raccogliere in *Forma* tutto ciò che è comune a tutte le forme specifiche e indipendente dalle specifiche realizzazioni delle diverse forme
- Ciò è possibile usando una classe astratta

Classi astratte

- Una classe astratta è una “via di mezzo” tra una classe e una interface:
  - può avere variabili di istanza e metodi di cui viene fornita anche una implementazione (come le classi)
  - può definire dei metodi di cui non viene fornita una implementazione (come le interface)
- I metodi di cui non viene fornita una implementazione vengono detti metodi astratti

Classi astratte sintassi

- Una classe astratta viene definita usando la parola chiave *abstract*

```
abstract class <NomeClasse>
```

- anche i metodi astratti vengono definiti usando la parola chiave *abstract*

```
<Modificatore> abstract <TipoRestituito>
          <NomeMetodo> (<ListaParametri>)
```

Soluzione 4

- Possiamo risolvere il problema del contenimento di forme creando:
  - una classe astratta *Forma*
  - le classi delle specifiche forme (*Cerchio*, *Triangolo*, *Rettangolo*, ecc.) che estendono la classe *Forma*
- Nella classe *Forma* definiremo i metodi *getId()* e *contiene(Forma f)*
- Nella classe *Forma* definiremo inoltre i metodi astratti *getXMin()*, *getXMax()*, *getYMin()* e *getYMax()*

La classe Forma2

```
public abstract class Forma2{
   protected String id; //codice identificativo

   /* restitusice il cod. ident. della forma */
   public String getId(){
       return this.id;
   }

   /* restituisce la coord. x min. della forma */
   protected abstract double xMin();

   /* restituisce la coord. y min. della forma */
   protected abstract double yMin();
   ...
}
```

La classe Forma2

```
public abstract class Forma2{
   ...
   /* restituisce la coord. x max. della forma */
   protected abstract double xMax();

   /* restituisce la coord. y max. della forma */
   protected abstract double yMax();
   ...
}
```

La classe Forma2

```
public abstract class Forma2{
   ...
   /* restituisce true se questa forma
      contiene la forma f */
   public boolean contiene(Forma2 f){
       return (this.xMin()<=f.xMin()) &&
              (this.yMin()<=f.yMin()) &&
              (this.xMax()>=f.xMax()) &&
              (this.yMax()>=f.yMax());
   }
}
```

La classe Cerchio2

```
public class Cerchio2 extends Forma2{
   private double raggio; //raggio del cerchio
   private double xC; //coord. x del centro
   private double yC; //coord y del centro

   /* crea un cerchio con id, raggio e coord. dati */
   public Cerchio2(String id, double r, double x,
                                        double y){
       this.id = id;
       this.raggio = r;
       this.xC = x;
       this.yC = y;
   }
   ...
}
```

La classe Cerchio2

```
public class Cerchio2 extends Forma2{
   ...
   /* restituisce la coord. x min. del cerchio */
   public double xMin(){
       return this.xC-this.raggio;
   }

   /* restituisce la coord. y min. del cerchio */
   public double yMin(){
       return this.yC-this.raggio;
   }
   ...
}
```

La classe Cerchio2

```
public class Cerchio2 extends Forma2{
   ...
   /* restituisce la coord. x max. del cerchio */
   public double xMax(){
       return this.xC+this.raggio;
   }

   /* restituisce la coord. y max. del cerchio */
   public double yMax(){
       return this.yC+this.raggio;
   }
}
```

La classe TriangoloEquilatero2

```
public class TriangoloEquilatero2 extends Forma2{
   private double lato; //lunghezza del lato
   private double xL; // coord. x vert. in basso a sx
   private double yL; // coord. y vert. in basso a sx

   /* crea un tr. eq. con id, lato e coord. dati */
   public TriangoloEquilatero2(String id, double l,
                               double x, double y){
       this.id = id;
       this.lato = l;
       this.xL = x;
       this.yL = y;
   }
...
}
```

La classe TriangoloEquilatero2

```
public class TriangoloEquilatero2 extends Forma2{
   /* restituisce la coord. x min. del tr. equil. */
   public double xMin(){
       return this.xL;
   }

   /* restituisce la coord. y min. del tr. equil. */
   public double yMin(){
       return this.yL;
   }
   ...
}
```

La classe TriangoloEquilatero2

```
public class TriangoloEquilatero2 extends Forma2{
   ...
   /* restituisce la coord. x max. del tr. equil. */
   public double xMax(){
       return this.xL+this.lato;
   }

   /* restituisce la coord. y max. del tr. equil. */
   public double yMax(){
       return this.yL+Math.sqrt(3)*this.lato/2;
   }
}
```

Un piccolo riassunto

- *interface I*:
  - ha solo metodi astratti
  - non ha variabili di istanza
  - non può avere membri statici
  - può avere costanti
  - definisce un tipo *I*
  - è possibile definire variabili di tipo *I*
  - non è possibile creare istanze dell'interface *I*
  - in una variabile di tipo *I* è possibile memorizzare un oggetto istanza di qualunque classe implementi *I* (o sue sotto-classi)

Un piccolo riassunto

- *abstract class A*:
  - può avere metodi astratti
  - può avere metodi non astratti
  - può avere variabili di istanza
  - può avere membri statici
  - può avere costanti
  - definisce un tipo *A*
  - è possibile definire variabili di tipo *A*
  - non è possibile creare istanze della classe *A*
  - in una variabile di tipo *A* è possibile memorizzare un oggetto istanza di qualunque classe estenda *A* (o sue sotto-classi)

Un piccolo riassunto

- *class C*:
  - ha solo metodi non astratti
  - può avere variabili di istanza
  - può avere membri statici
  - può avere costanti
  - definisce un tipo *C*
  - è possibile definire variabili di tipo *C*
  - è possibile creare istanze della classe *C*
  - in una variabile di tipo *C* è possibile memorizzare un oggetto istanza di *C* o di qualunque sua sotto-classe