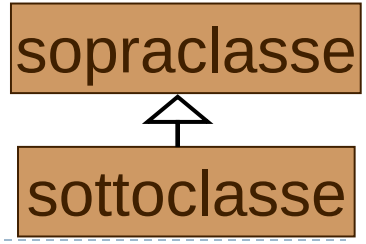
Ereditarietà

Ereditarietà



- È possibile, tramite il meccanismo dell'**ereditarietà**, costruire nuove classi (**sottoclassi o derivate**) a partire da classi già esistenti (**sopraclasse o base**). Migliora il riutilizzo del software.
- La sottoclasse **eredita** tutti i metodi e gli attributi della sopraclasse anche privati e può fare riferimento nel codice direttamente a tutti quelli non privati.
- Le classi vengono così organizzate in una **gerarchia**.
- La sottoclasse specializza la sopraclasse per:
  - **Estensione**: vengono aggiunti nuovi attributi o metodi
  - **Ridefinizione**: ridefinisce attributi (**hiding**) o metodi della sopraclasse (**overriding**)

```
public class SottoClasse extends SuperClasse {
<nuove_variabili_istanza>    <nuovi_metodi>  }
```


40

Ereditarietà

- La classe base definisce **attributi e interfaccia comuni (ereditati)**
- Le classi derivate possono **invocare i metodi delle classi base**, purché definiti **pubblici o protetti**, ma non ereditano i costruttori
- Le classi derivate **non possono eliminare** dati o metodi delle sopraclassi
- Seguono il **Principio di Sostituibilità di Liskov LSP**
  - *Oggetti di una classe derivata (subclass) sono sostituibili ad oggetti della classe più generale (base class or super-class) ovvero dovunque si usa un oggetto della classe madre deve essere possibile sostituirlo con un oggetto di una qualunque delle classi figlie*

41

Ereditarietà
Costruttore di una sottoclasse

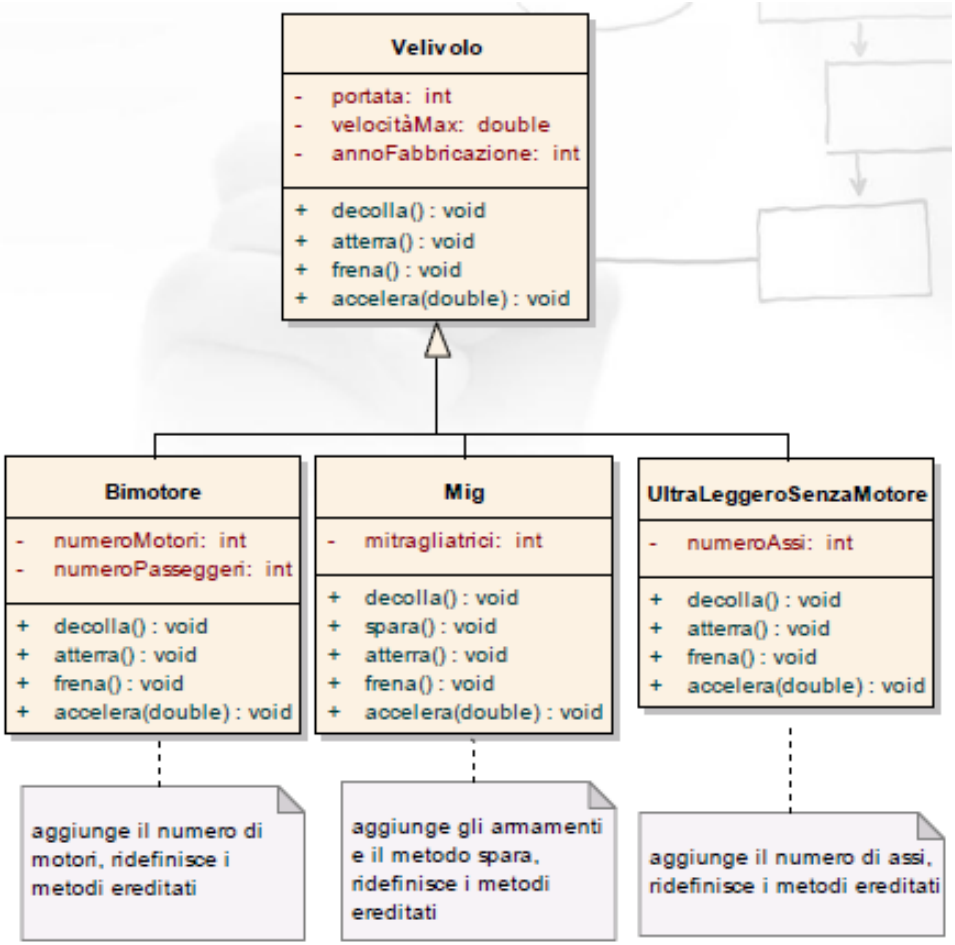
- I costruttori non vengono ereditati
- Ogni istanza della classe derivata comprende in sé, indirettamente, un oggetto della classe base, quindi quando istanzio un oggetto di una classe derivata, viene automaticamente richiamato il costruttore della classe base e poi quello della classe derivata. È possibile richiamare esplicitamente il costruttore della superclasse tramite l'istruzione `super(listaParametriAttuali)` posta come prima istruzione del costruttore
- Il costruttore è passibile di overloading
- ATTENZIONE: se non c'è un costruttore nella sopraclasse senza parametri, devo definire nella classe derivata un costruttore che richiami in modo esplicito **super()** con i parametri richiesti (il compilatore dà un errore)

42

Ereditarietà UML

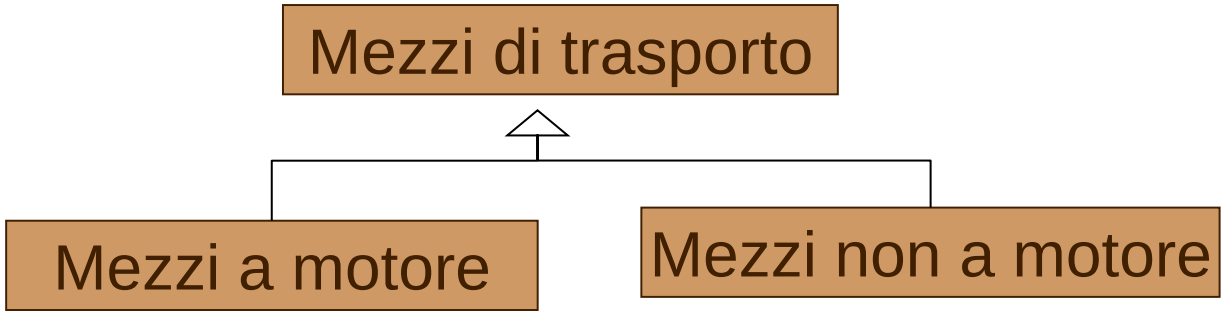
In UML la relazione che esprime l'ereditarietà è la **generalizzazione**

Si usa una freccia continua che indica una relazione **IS-A**



Ereditarietà Singola

- **Singola**: se una sottoclasse deriva da una sola sopraclasse

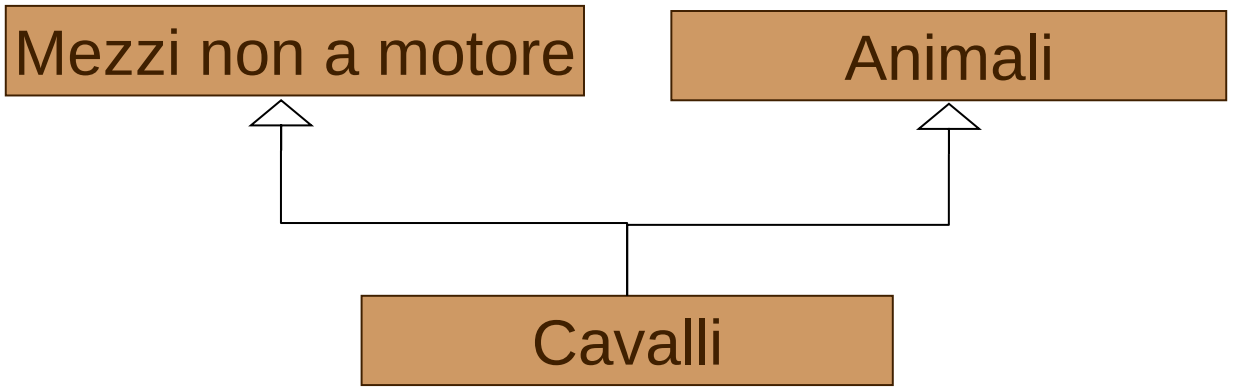


- Java supporta solo questo tipo di ereditarietà, e ogni classe, se non specificata esplicitamente come sottoclasse di un'altra classe, viene vista comunque come sottoclasse della classe **Object**, che costituisce la radice comune dell'albero di ereditarietà delle classi

44

Ereditarietà
Multipla

- **Multipla**: se una sottoclasse deriva da più sopraclassi (non tutti i linguaggi di programmazione la prevedono)



45

Ereditarietà
Polimorfismo

- Si ha **polimorfismo** se un metodo è in grado di adattare il suo comportamento allo specifico oggetto su cui deve operare. Ha 2 aspetti:
  - **Prima forma**: lo stesso metodo di una classe base è applicato ai diversi oggetti delle classi derivate
  - **Seconda forma**: lo stesso metodo genera l'esecuzione di codice diverso.

46

Ereditarietà
Polimorfismo

- **Seconda forma**: lo stesso metodo genera l'esecuzione di codice diverso.
- Si può distinguere:
  - **Statico (a livello di compilazione) overloading:** quando lo stesso metodo, ha tipo o numero di parametri diversi, (**firma** diversa) e quindi codice diverso (non basta cambiare il tipo restituito) (all'interno della stessa classe o in classi derivate per estensione)
  - **Dinamico (a livello di esecuzione) overriding:** quando lo stesso metodo ha la stessa firma, ma codice diverso a seconda della classe, implica l'ereditarietà (sottoclasse ridefinisce metodo della sopraclasse)

47

Ereditarietà
overriding e overloading

- Per fare l'**overriding** di un metodo, nella classe base questo deve essere visibile (public o protected) e overridable (né finale, né statico perché si ha un hiding)

```
class Base
        [{Protected|Public}] NomeMetodo([par])
class Derivata extends Base
        [{Protected|Public}] NomeMetodo([par])
```

- Per fare l'**overloading** di un metodo (sono tutti overloadable per default) (ATTENZIONE non basta cambiare il tipo restituito)

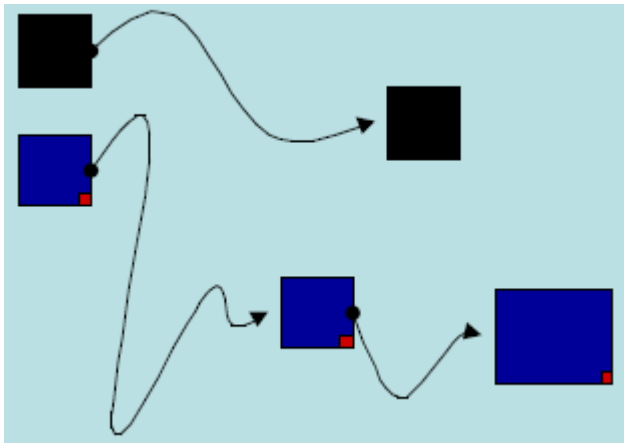
```
        modifAccesso NomeMetodo([par])
        modifAccesso NomeMetodo([parDiversi])
```

Si possono avere 2 metodi **statici** con lo stesso nome, ma restano distinti e quello della sottoclasse non sovrascrive (overriding) quello della superclasse, ma lo nasconde (hiding). Entrambi restano raggiungibili attraverso il nome della classe (o con super). Se si cerca di ridefinirlo non statico si segnala errore

Ereditarietà
Esempio

Generare quadrati da sottoporre a due tipi di operazioni:

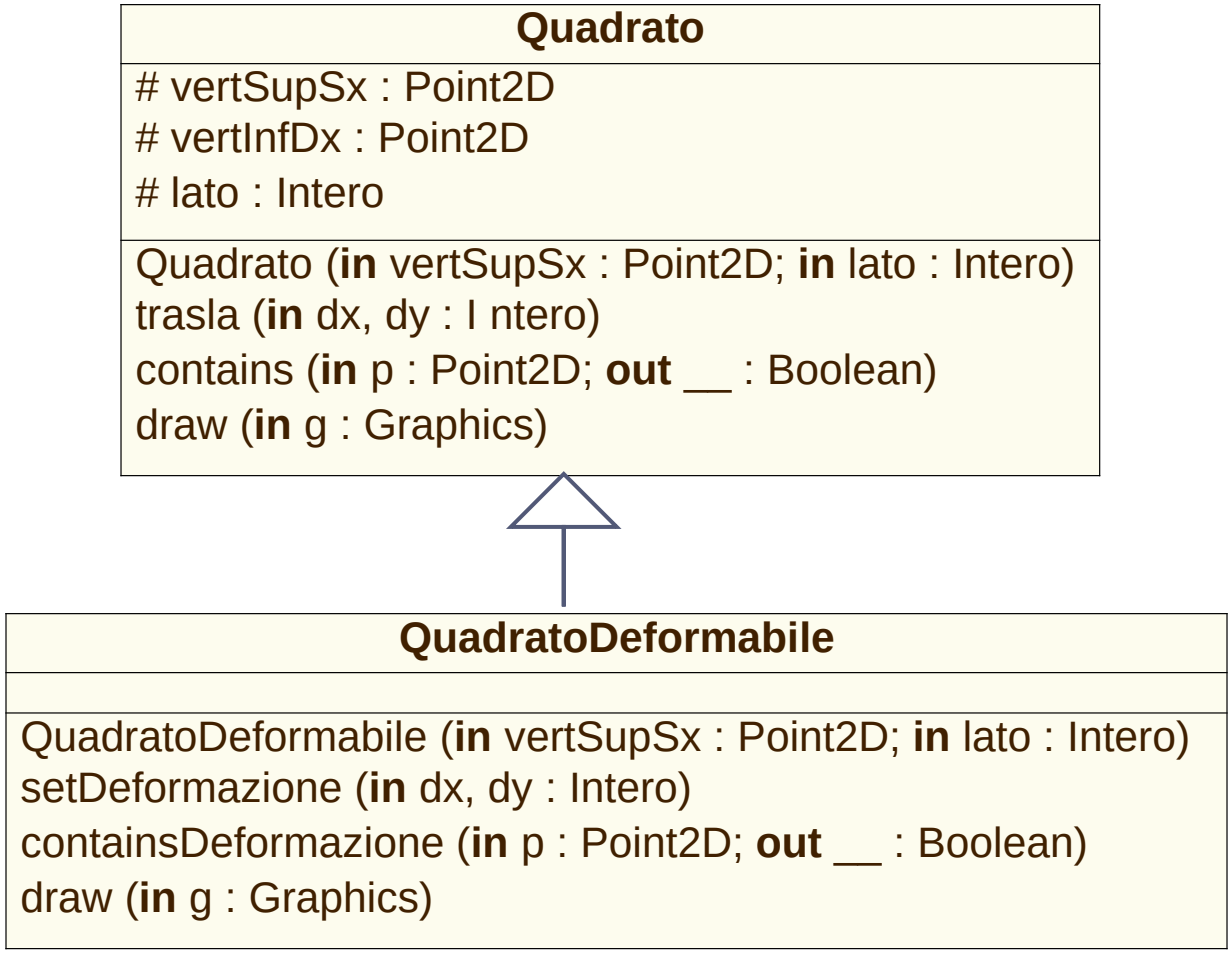
- traslazione
- traslazione e deformazione



- **Operazioni disponibili:**
  - istanzazione di oggetti di tipo Quadrato
  - verifica di punto interno (ovvero esterno) a un quadrato
  - tracciatura di un quadrato
  - istanzazione di oggetti di tipo QuadratoDeformabile
  - registrazione della deformazione di un quadrato deformabile traslazione di un quadrato
  - verifica di punto interno (ovvero esterno) alla regione di deformazione

49

Ereditarietà
Esempio



50

Ereditarietà
Esempio

Istanze di Quadrato

- *vertSupSx*
- *vertInfDx*
- *lato*
- *trasla*()
- *contains*()
- *draw*()

Istanze di QuadratoDeformabile

- *vertSupSx* ereditato
- *vertInfDx* ereditato
- *lato* ereditato
- *trasla*() ereditato
- *contains*() ereditato
- *draw*() ereditato e sovrascritto
- *setDeformazione*() aggiunto
- *containsDeformazione*() aggiunto

Un oggetto che è istanza di QuadratoDeformabile è anche un oggetto di tipo Quadrato (possiede attributi e metodi ereditati)

51

Ereditarietà
NOTA BENE per overriding

- Un metodo ridefinente può cambiare il tipo restituito solo se il tipo di ritorno è a sua volta un sottotipo del tipo restituito dal metodo della superclasse. Se il tipo è primitivo, deve restare lo stesso (compilatore dà errore). Esempio in cui Rettangolo è una sottoclasse di Shape:

```
class Madre { …
            Shape metodo1() { … } }
class Figlia extends Madre { …
            Rettangolo metodo1() { …
      } }
```

- La clausola throws può differire, purché le eccezioni intercettate siano le stesse o sottotipi di quelle intercettate nel metodo della superclasse

52

Ereditarietà
NOTA BENE per overriding

- L'overriding è possibile solo se il metodo è accessibile, visibile. Altrimenti se nella sottoclasse viene creato un metodo con la stessa signature di quello non accessibile della superclasse, i due metodi saranno completamente scorrelati
- I metodi che operano l'overriding possono avere i propri modificatori di accesso, che possono tuttavia solo ampliare (rilassare) la protezione. Ad esempio un metodo protected nella superclasse può essere ridefinito protected o public, ma non private
- Il metodo ridefinente può essere reso final, se ritenuto opportuno e può essere reso abstract anche se non lo è quello della superclasse

53

Ereditarietà
NOTA BENE per overriding

- Attributi e metodi statici NON possono essere sovrascritti, ma solo adombrati (hiding); questo comunque non ha alcuna rilevanza perché si utilizza comunque il nome della classe di appartenenza per accedervi
- Un attributo non viene ridefinito ma adombrato (hiding): il nuovo campo rende inaccessibile quello definito nella superclasse (si deve usare super)

```
class Madre {int x;   public x() {…}
public class Figlia extends Madre {
  int x;
public Figlia(int k){
  if k==super.x this.x=k;
  super.x();}}
```


54

Ereditarietà
Livelli di protezione

Analizziamo i vari livelli di protezione nel contesto dell'ereditarietà

- **private** impedisce a chiunque di accedere al dato, anche a una classe derivata. Se deve usare per dati "veramente privati", nella maggioranza dei casi è troppo restrittivo
- **public** non implementa l'information hiding
- **package** è quello di default, ma il concetto di package non c'entra niente con l'ereditarietà
- **protected** è come package per chiunque non sia una classe derivata, ma consente libero accesso a una classe derivata, presente o futura, indipendentemente dal package in cui essa è definita tramite super. È quello da preferire, applicabile ad attributi e metodi

55

Ereditarietà
Up-casting, down-casting

- In una gerarchia di classi derivate è possibile assegnare a un riferimento (var) un oggetto istanza di una qualsiasi sottoclasse, ma non viceversa (**principio di sostituibilità**) perché non è possibile la corretta invocazione dei metodi: il riferimento vede i metodi e gli attributi della classe di dichiarazione a cui deve corrispondere del codice. Quindi ad un riferimento di tipo Object può essere assegnato qualsiasi istanza di oggetto perché sopraclasse di tutti
- I riferimenti agli oggetti si dicono **polimorfi**, ovvero possono riferirsi ad oggetti di tipo diverso all'interno della gerarchia
- Un oggetto non può essere assegnato ad un riferimento di una sottoclasse (errore di compilazione)

56

Ereditarietà
Up-casting, down-casting

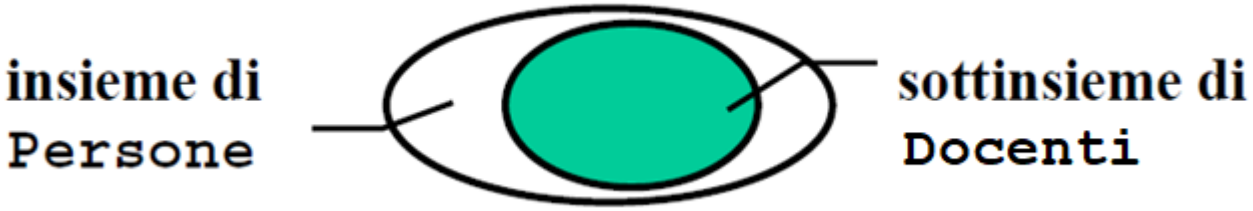
- Si possono fare operazioni di casting per trasformare il riferimento ad un oggetto in un riferimento a:
  - una sopraclasse **up-casting** (sempre possibile e sottinteso)
  - una sottoclasse **down-casting** (possibile solo se il riferimento è in realtà ad un oggetto della stessa classe. Altrimenti verrà sollevata un'eccezione di tipo *ClassCastException* in esecuzione)

```
Persona p1,p2;
Docente doc1,doc2;//Docente extends Persona
doc1= new Docente("Rossi Ugo","Mate");
p2= new Persona("Verdi Lia");
//errore doc2= new Persona("tizio");
p1=doc1;     //up-casting sottinteso (Persona)doc1
doc2=(Docente)p2;     //down-casting     non     lecito
   ClassCastException
doc2=(Docente)p1;  //down-casting lecito perché dip1 in
   realtà si riferisce ad un Docente
```


57

Ereditarietà
Up-casting, down-casting

- Poiché Docente eredita da persona, possiamo usare uno Docente ovunque sia richiesta una Persona.



- Ciò equivale infatti a dire che il tipo Docente è compatibile col tipo Persona, esattamente come float è compatibile con double.
- Quindi, se s è uno Studente e p una Persona, l'istruzione:

p = s

è LECITA perchè *non comporta perdita di informazione*.

Ereditarietà
instanceof

- Facendo il casting è possibile in fase di compilazione riferirsi ai metodi e agli attributi della nuova classe, ma posso sorgere errori in fase di esecuzione. Bisognerebbe verificare la classe di appartenenza
- **`instanceof`** è un operatore che consente di determinare a run-time il tipo di un oggetto. Restituisce true o false a seconda che l'oggetto sia o no un'istanza della classe di confronto o di una delle sue superclassi. L'operatore si applica anche alle interfacce.

```
if (p1 instanceof Docente)
    ((Docente)p1).setMateria("info");
```


59

Ereditarietà
Overloading e overriding

- L'**overloading** consente di definire in una stessa classe più metodi aventi lo stesso nome, ma che differiscano nella ***firma***, cioè nella sequenza dei tipi dei parametri formali. È il **compilatore** che può determinare quale dei metodi verrà invocato, in base al numero e al tipo dei parametri attuali
- L'**overriding** consente di ridefinire un metodo in una sottoclasse: il metodo originale e quello che lo ridefinisce hanno necessariamente la stessa firma. A tempo di esecuzione verrà invocato il metodo corrispondente alla classe dell'istanza a cui si fa riferimento. Solo l'**interprete** può quindi determinare quale deve essere eseguito

60

Ereditarietà
Binding

- Il meccanismo che determina quale metodo deve essere invocato in base alla classe di appartenenza dell'oggetto si chiama **binding (legame).**
- Si distingue in:
  - **binding statico** o **early binding**: avviene a tempo di compilazione (standard in C, default in C++ e VB, per i metodi statici o final in Java)
  - **binding dinamico** o **late binding**: avviene a tempo di esecuzione (non presente in C, possibile in C++ e VB (virtual), default in Java)

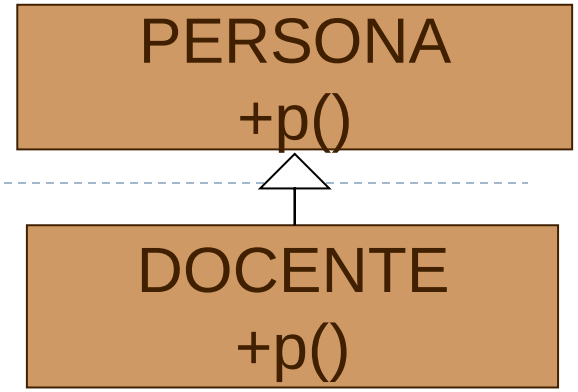
61

Ereditarietà
Binding

- **binding statico** o **early binding**: le chiamate ai metodi sono collegate alla versione del metodo a tempo di compilazione. Per sapere il codice associato il compilatore percorrerà a ritroso la gerarchia delle classi fino a trovare la sua implementazione. Tale collegamento non può essere indefinito (il compilatore dà errore) e non può cambiare in fase di esecuzione
- **binding dinamico** o **late binding**: le chiamate ai metodi sono collegate alla versione del metodo determinata a tempo di esecuzione, basandosi sul tipo dinamico dell'oggetto referenziato in quel momento. Si demanda la scelta del metodo da invocare all'interprete. Il compilatore si preoccuperà solo di assicurare che la chiamata sia collegabile a "qualche" metodo di quella classe o di una sua derivata, senza interessarsi di "quale" sarà.

62

Ereditarietà
Binding Statico

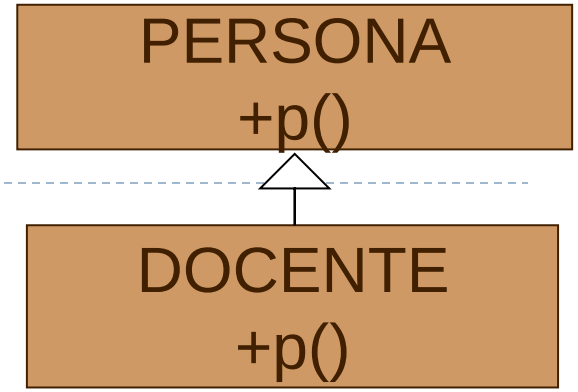



```
Persona p1;
Docente doc1;
p1= new Persona("Verdi Lia");
doc1= new Docente("Rossi Ugo","Mate");
p1.p()   //della classe Persona
doc1.p() //della classe Docente
```

Si sa quale usare in fase di compilazione.

In Java però, anche in questo caso, il binding è fatto a tempo di esecuzione. Solo se il metodo è non overridable (static o final), per cui non è possibile un overriding, Java assegna il codice in fase di compilazione

Ereditarietà
Binding dinamico




```
int cond = Input.readInt();
Persona tmp;
if (cond > 0)
  tmp = new Persona("Mario Rossi");
else
  tmp = new Docente("Mario Rossi", "mate");
tmp.p();
```

L'oggetto `tmp` si saprà solo in fase di esecuzione, il metodo da lanciare non è conosciuto in fase di compilazione.
Il binding dinamico è quello usato di default in Java

64

Ereditarietà
Binding statico vs dinamico

STATICO

- **PRO** il programma eseguibile finale ha un numero di istruzioni macchina minore e quindi è più efficiente
- **CONTRO** se modifico un metodo devo ricompilare tutto il progetto

DINAMICO

- **PRO** riusabilità del codice eseguibile (si ricompila solo il modulo modificato)
- **CONTRO** meno efficienza in fase di esecuzione perché, per eseguire un metodo, devono anche eseguirsi le istruzioni per riconoscere l'oggetto a cui appartiene

65

Ereditarietà
esempio

```
public class Animale {
public void verso () {} //metodo vuoto
}

public class Scoiattolo extends Animale {
public void verso() {
   System.out.println("SQUIT SQUIT"); }
}

public class Cane extends Animale {
public void verso() {
   System.out.println("BAU BAU");}
public void ringhiare() {
   System.out.println("GRRRR"); }
}
```


66

Ereditarietà
esempio

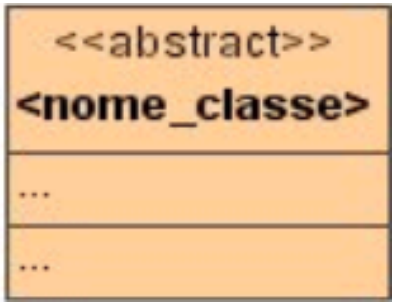
```
public class Zoo {
 public static void main (String[] argv) {
  Animale[] gabbie= new Animale[2];
  gabbie[0] = new Scoiattolo();
  gabbie[1] = new Cane();
  for (int k=0; k< gabbie.length; k++){
    gabbie[k].verso();
    if (gabbie[k] instanceof Cane)
       ((Cane)gabbie[k]).ringhiare(); //downcasting
  }
 }
}
```

output

```
SQUIT SQUIT
BAU BAU
GRRRR
```


67

Classi astratte



- Una **classe astratta o virtuale** definisce
  - un tipo di dato
  - un insieme di operazioni sul tipo di dato
    - alcune operazioni sono implementate: metodi concreti
    - alcune operazioni possono non essere implementate: metodi astratti o virtuali (abstract)
- **Una classe astratta non può essere istanziata**, ma è possibile derivarne sottoclassi.
- Viene usata come classe base le cui sottoclassi vanno a specificare i comportamenti o le proprietà.
- In genere possiede almeno un metodo astratto non implementato. Una classe che ha un metodo abstract deve essere definita abstract.

68

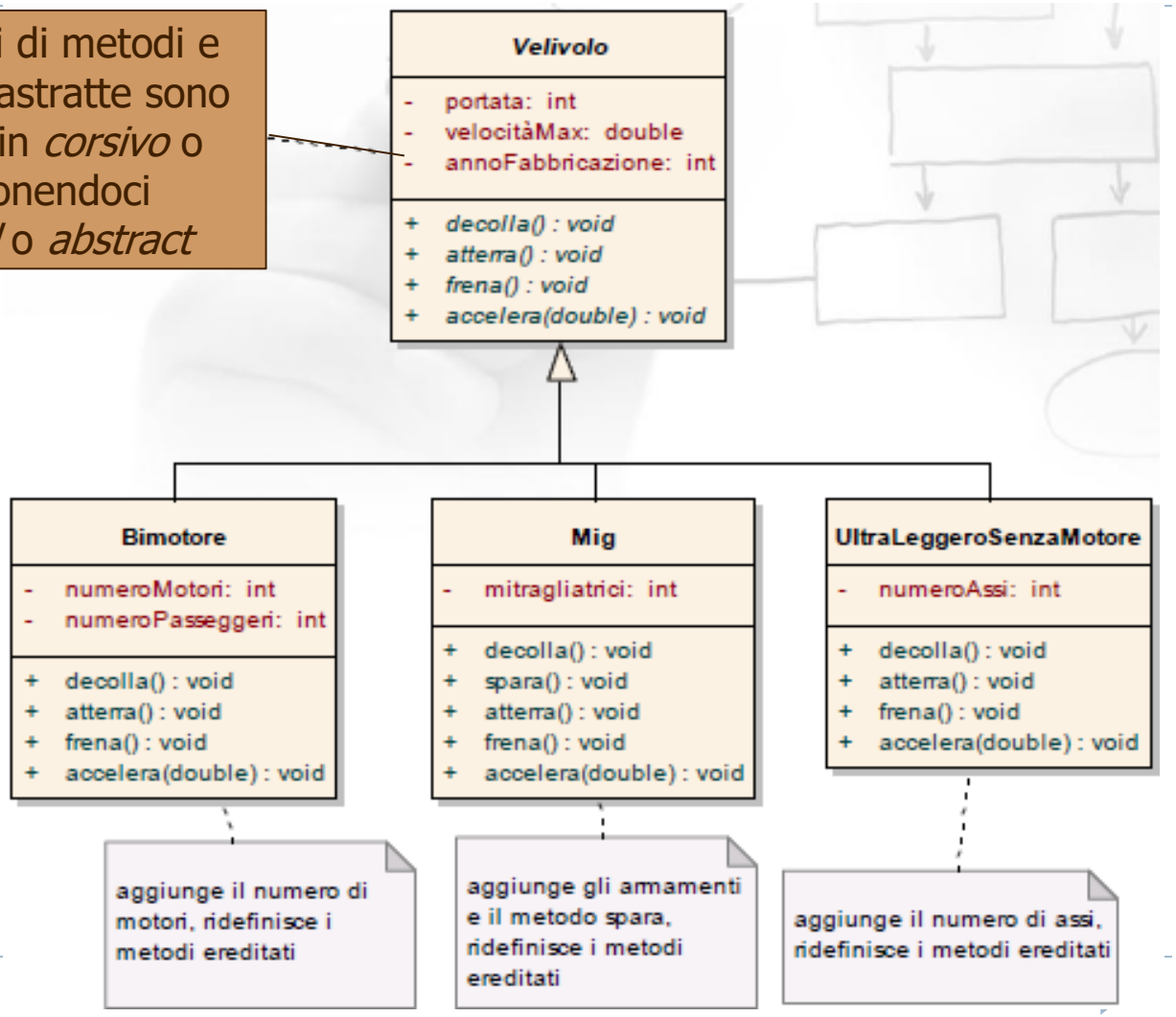
Classi astratte

- Può avere anche dei costruttori. Sarà il compilatore a segnalare un errore se si cerca di istanziarne un oggetto.
- Le sottoclassi della classe astratta implementeranno i metodi virtuali. Quando tramite la gerarchia delle classi, tutti i metodi virtuali saranno implementati, la classe finale può non essere più astratta e quindi si potranno istanziare gli oggetti di questa sottoclasse
- Una classe astratta può contenere chiamate di metodi astratti prescindendo dalla loro implementazione, una classe concreta non può usare metodi astratti

69

Classi astratte UML

I nomi di metodi e classi astratte sono scritti in *corsivo* o anteponendoci *virtual* o *abstract*

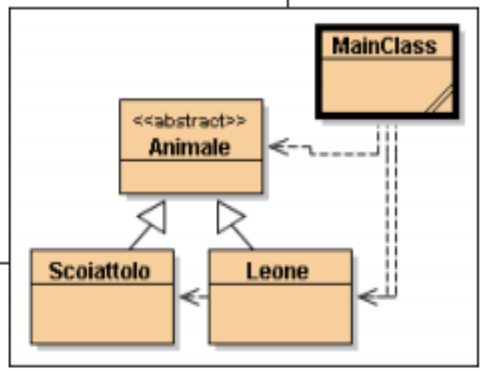


70

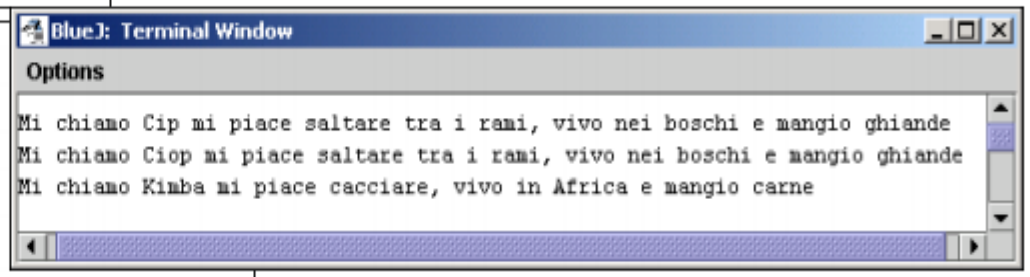
Classi astratte

```
1  public abstract class Animale
2  {
3  public String nome;
4  Animale(String nome)
5  {
6  this.nome=nome;
7  }
8  public abstract String attivita();
9  public abstract String vive();
10 public abstract String mangia();
11 public void presentati()
12 {
13 System.out.println("Mi chiamo "+nome+
14                    " mi piace "+attivita()+", "+
15                     "vivo "+vive()+
16                     " e mangio "+mangia());
17 }
18 }
```

```
1  public class MainClass
2  {
3  public static void main(String arg[])
4  {
5  Animale[] a=new Animale[3];
6  a[0]=new Scoiattolo("Cip");
7  a[1]=new Scoiattolo("Ciop");
8  a[2]=new Leone("Kimba");
9
10 for(int i=0;i<3;i++)
11 {
12 a[i].presentati();
13 }
14 }
15 }
```




```
1  public class Scoiattolo extends Animale
2  {
3  public Scoiattolo(String s)
4  {
5      super(s);
6  }
7  public String attivita(){ return "saltare tra i rami";}
8  public String vive(){ return "nei boschi";}
9  public String mangia(){ return "ghiande";}
10 }
```



Interfacce

- Ogni oggetto possiede implicitamente un'interfaccia: l'insieme degli attributi e dei metodi pubblici, con relative firme della classe di appartenenza
- È possibile definire esplicitamente un'**interface**, ovvero un'evoluzione più restrittiva del concetto di classe astratta, contenente solo le firme di metodi pubblici astratti e la dichiarazione di attributi pubblici, final e statici (i modificatori se non dichiarati sono sottintesi). Non possono contenere codice

```
[public] interface Nome{…}//senza public è package
```


72

Interfacce

- Contengono solo
  - metodi `public abstract`
  - attributi `public static final`
- Una classe le può implementare (realizzare) definendo tutti i "corpi" dei metodi

  `[public] class C` **`implements`** `int [,int2]`
- Una interfaccia non può essere istanziata, ma definisce un protocollo del comportamento che deve essere fornito. Una qualsiasi classe che implementa una data interfaccia è quindi obbligata a fornire l'implementazione di tutti i metodi elencati nell'interfaccia

73

Interfacce

- Si dice che un'interfaccia è un contratto tra chi la implementa e chi la usa
- La classe che implementa un'interfaccia, essendo obbligata ad implementarne tutti i metodi garantisce la fornitura di un servizio
- Chi usa un interfaccia ha la garanzia che il contratto di servizio è effettivamente realizzato: non può accadere che un metodo non possa essere chiamato.

74

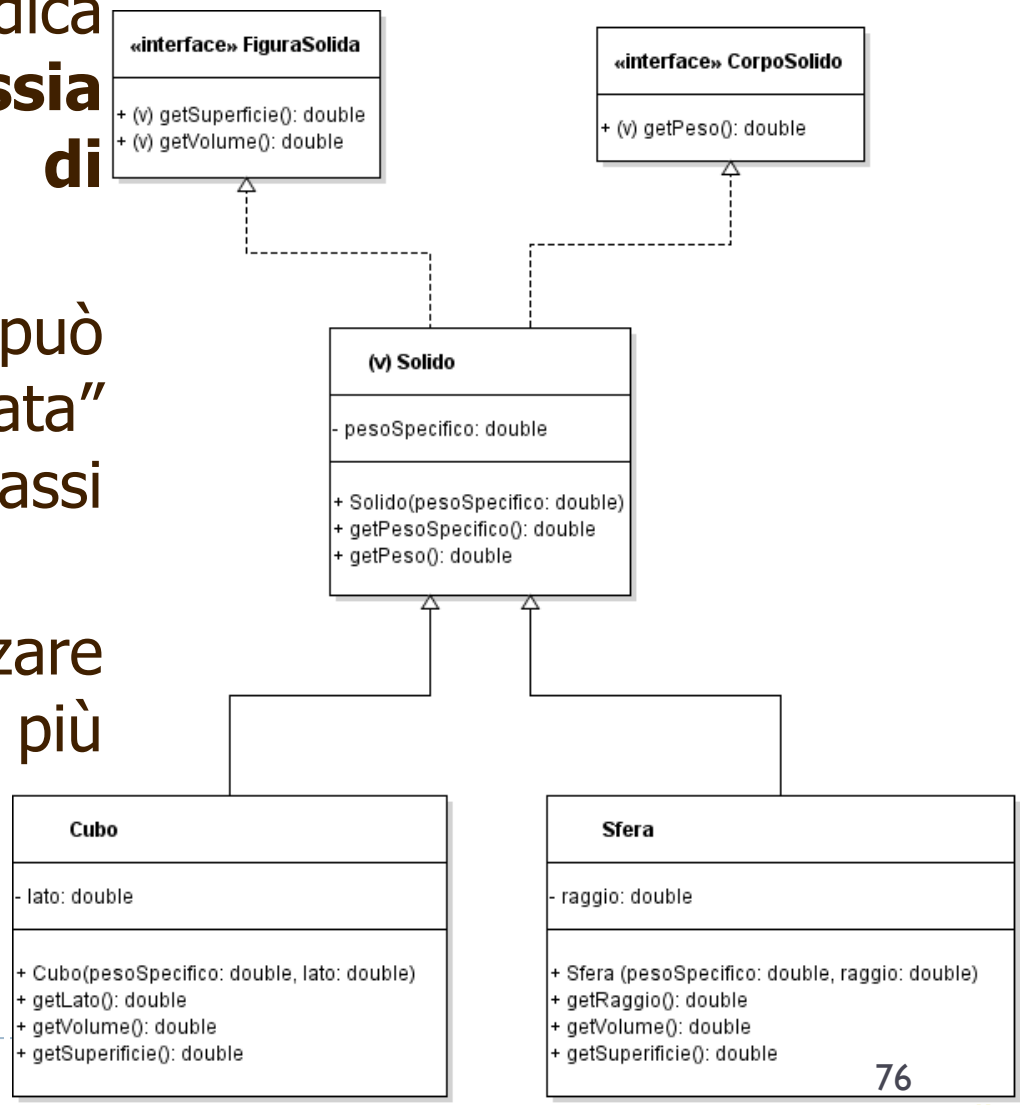
Interfacce

- Una **interfaccia** (**interface)** ha una struttura simile a una classe, ma può contenere SOLO **metodi d'istanza astratti** e **costanti statiche** (è sottinteso che lo siano, quindi non può contenere *costruttori*, *variabili statiche*, *variabili di istanza* e *metodi statici*).
- Si può dichiarare che una classe **implementa** (**implements**) una data interfaccia: in questo caso deve **realizzare** tutti i suoi metodi astratti. Se non li implementa tutti, la classe dovrà essere astratta. La realizzazione di un metodo deve rispettare la specifica firma del corrispondente metodo astratto.
- Un'interfaccia può estendere per ereditarietà altre interfacce (si crea una gerarchia di interfacce)

75

Interfacce UML

- La freccia tratteggiata indica la "**realizzazione", ossia l'implementazione di un'interfaccia**
- Una classe interfaccia può essere "realizzata" (implementata) da più classi concrete
- Una classe può realizzare (implementare) più interfacce



Interface
raggruppamento di costanti

- Poichè qualsiasi campo in un'interfaccia è automaticamente static e final, l'interfaccia è stata anche usata per creare gruppi di valori costanti (come con enum in C o in C++) (in Java c'è Enum o Enumerated Type)

Esempio

```
public interface Months {
int JANUARY = 1, FEBRUARY = 2, MARCH =
  3, APRIL = 4, MAY = 5, JUNE = 6, JULY
  = 7, AUGUST = 8, SEPTEMBER = 9,
  OCTOBER = 10, NOVEMBER = 11, DECEMBER
  = 12; }
```


77

Interfacce e classi astratte

Si noti che:

- Una classe (astratta o concreta) può implementare più interfacce (simula ereditarietà multipla), ma estendere solo una classe (astratta o concreta)
- Una classe (astratta o concreta) può essere usata per "fattorizzare" codice comune alle sue sottoclassi, una interfaccia non può contenere codice
- Una classe astratta può contenere chiamate di metodi astratti prescindendo dalla loro implementazione, una classe concreta non può usare metodi astratti, una interfaccia non può contenere codice

78

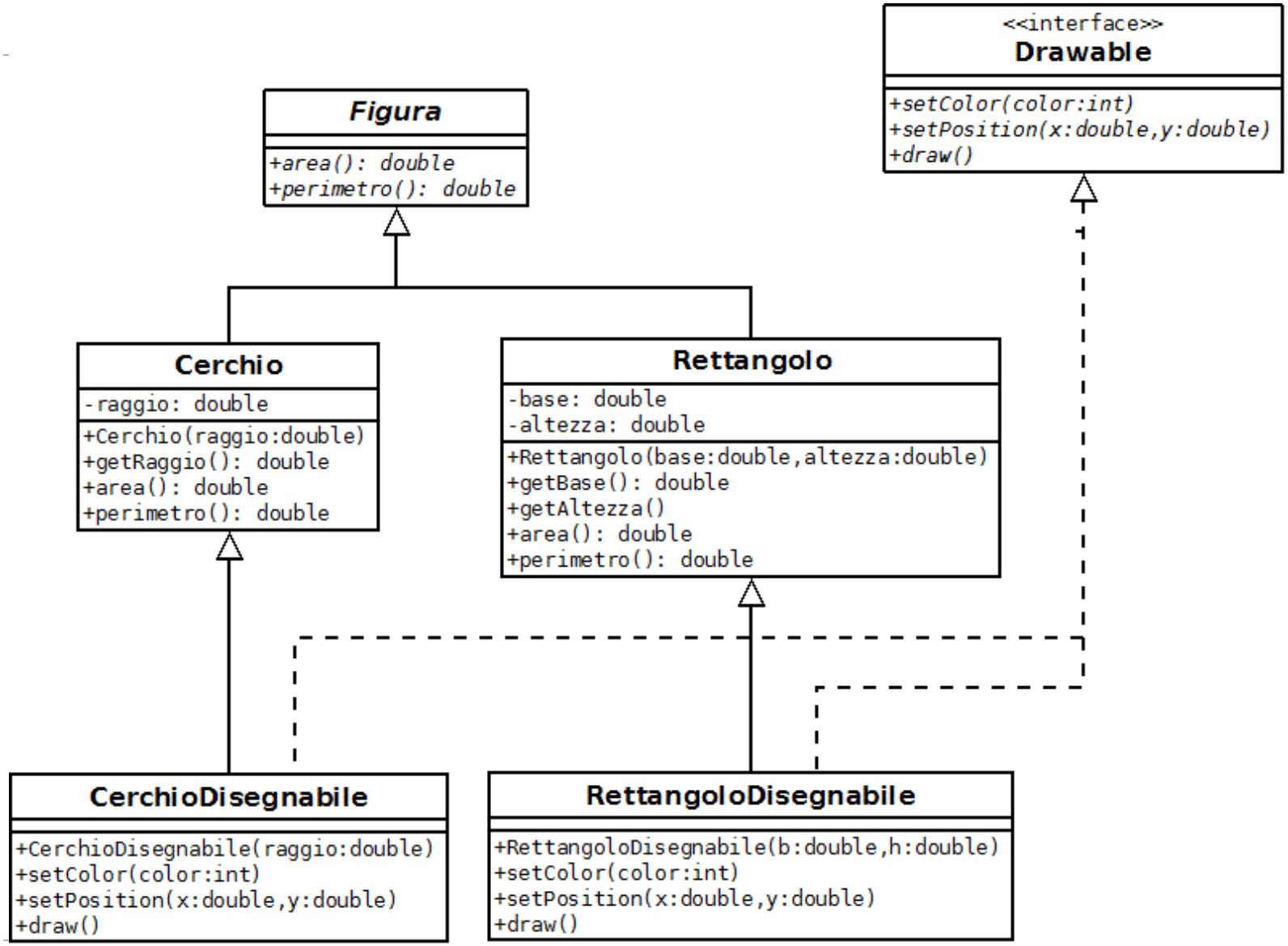
Interfacce come tipi

- Analogamente alle classi, ogni interfaccia definisce un tipo di dati in Java
- Un oggetto che implementa una data interfaccia ha come tipo anche il tipo dell'interfaccia, un oggetto può implementare molte interfacce di conseguenza può avere molti tipi
- Possiamo dichiarare una variabile indicando come tipo un'interfaccia. Ad una variabile di tipo interfaccia possiamo assegnare solo istanze di classi che implementano l'interfaccia.
- Su di una variabile di tipo interfaccia possiamo invocare solo metodi dichiarati nell'interfaccia (o nelle sue "super-interfacce").

79

Interfacce come tipi

- Es.



Interfacce come tipi

```
public class UsaDrawable {
   public static void main(String args[]) {
  Drawable[] drawables = new Drawable[3];
  drawables[0]= new CerchioDisegnabile(2.5);
  drawables[1]= new RettangoloDisegnabile(1.2, 3.0);
  for (int i = 0; i < drawables.length; i++) {
    drawables[i].setPosition(i*10.0, i* 20.0);
    drawables[i].draw();
  }
…}
```

- Grazie all'uso dell'interfaccia abbiamo potuto costruire un array che contiene indifferentemente istanze di classi diverse che implementano l'interfaccia Drawable e disegnarle tutte insieme con un solo ciclo for

81

Interfacce come tipi

- Le interfacce permettono una forma di compatibilità e di sostituibilità tra classi indipendente dalla catena di ereditarietà
- Per esempio è possibile definire una classe che implementa Drawable, ma non discende da Figura: un testo che può essere disegnato in una data posizione.

```
public class DrawableText implements Drawable {
   protected int c; protected double x, y;
   protected String s;
   public DrawableText (String s) { this.s = s; }
   public void setColor(int c) { this.c = c; }
   public void setPosition(double x, double y) {
     this.x = x; this.y = y; }
   public void draw() {…}
}
```


82

Quando utilizzare l'interfaccia e quando la classe astratta?

- Si usa una **classe astratta** per condividere codice fra più classi, se più classi hanno in comune metodi e campi o se si vogliono dichiarare metodi comuni che non siano necessariamente campi static e final.
- Si decide di utilizzare una **interfaccia** se ci si trova nella situazione in cui alcune classi (assolutamente non legate fra di loro) si trovano a condividere i metodi di una interfaccia, se si vuole specificare il comportamento di un certo tipo di dato (ma non implementarne il comportamento) o se si vuole avere la possibilità di sfruttare la "multiple inheritance".

83

L'interfaccia di ogni classe è definita al passo 2 (Individua attributi e metodi) e al passo 4 (individua il livello di protezione)

Approccio per la stesura di programmi OOP

1. Esaminare la realtà, ed individuare gli elementi essenziali della stessa (protagonisti)
2. Differenziare i ruoli: gli elementi importanti diventeranno classi, quelli meno diventeranno attributi; le azioni che gli oggetti possono fare o subire diventano invece metodi. Inizialmente, si devono solo individuare classi, attributi e metodi
3. Se qualche classe dovrà possedere attributi e/o metodi già posseduti da altre, sfruttare il meccanismo di ereditarietà
4. Stabilire il livello di protezione degli attributi e gli eventuali metodi per la gestione degli stessi (metodi probabilmente necessari in caso di attributi privati); occorre anche stabilire il livello di protezione dei metodi.
5. Stesura dei costruttori delle classi, per decidere qual è lo stato iniziale degli oggetti
6. Implementazione dei metodi

84