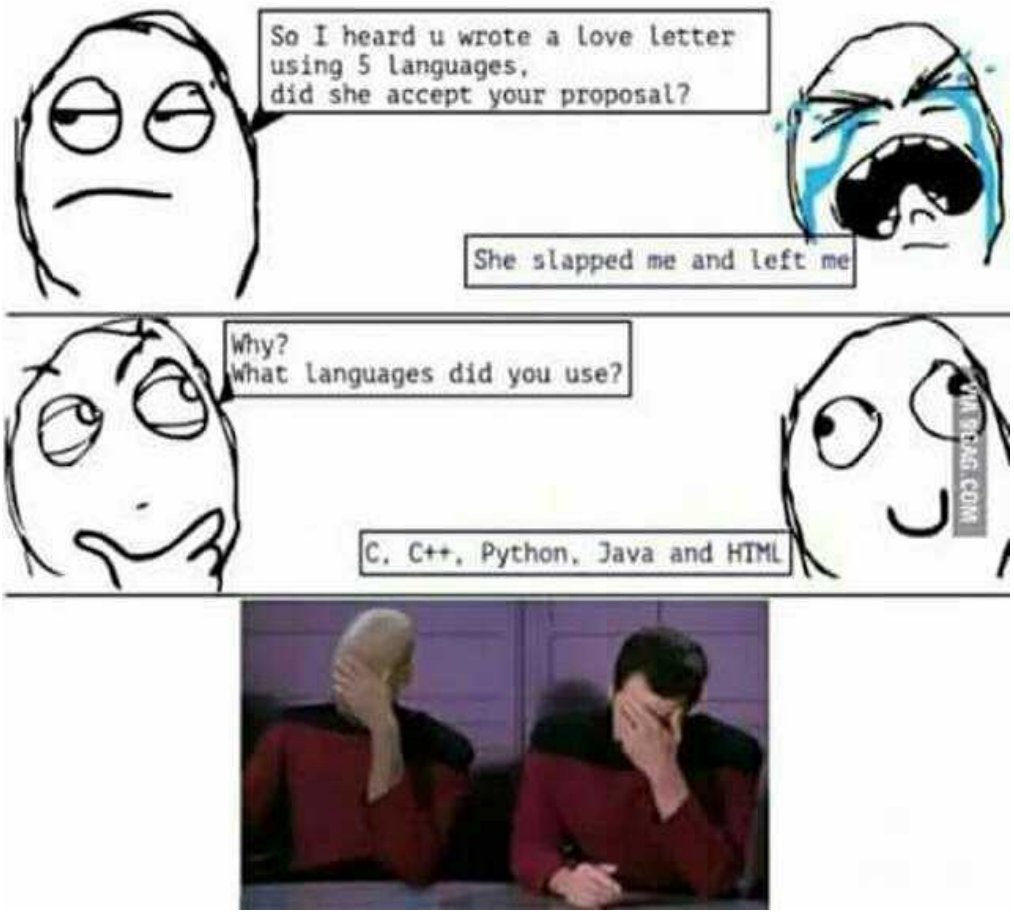
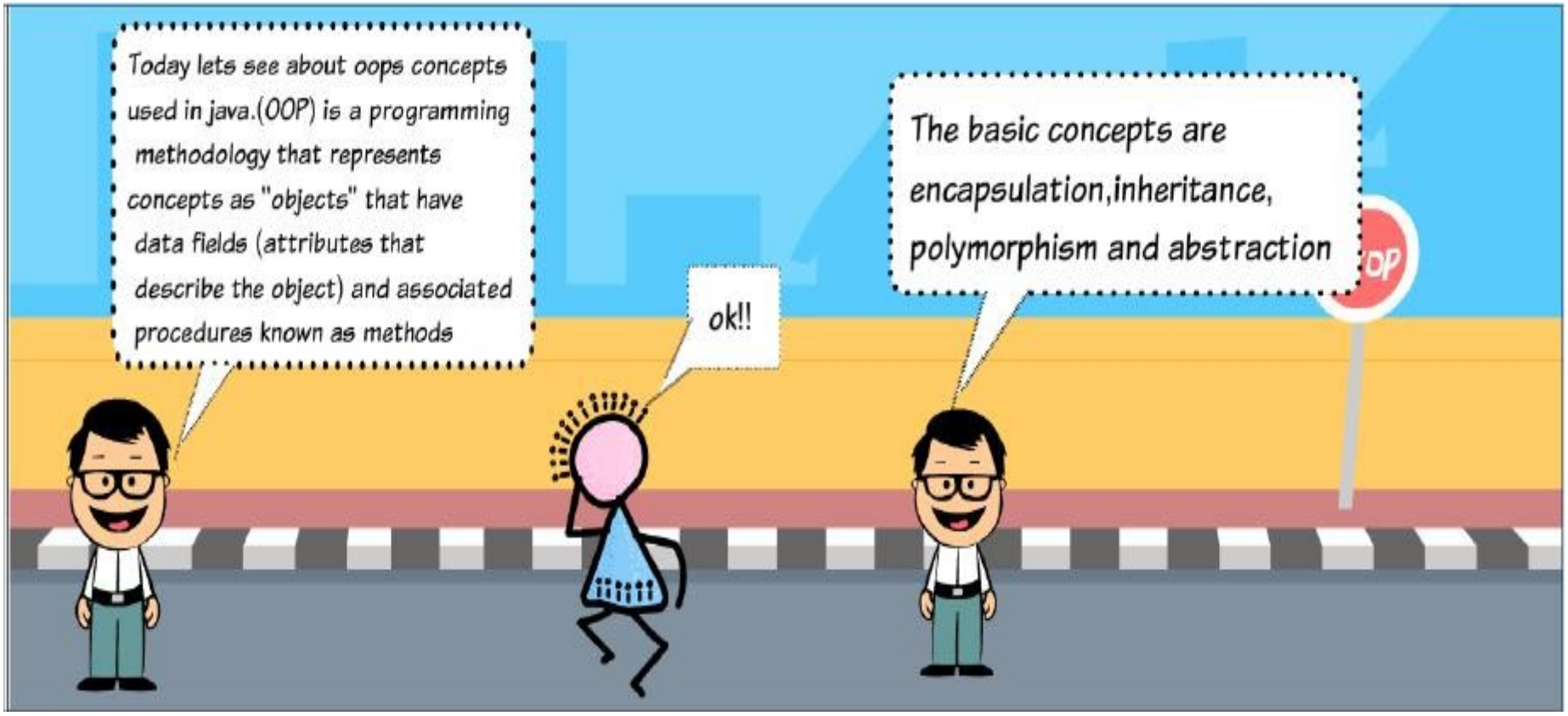
OOP:i tre pilastri (IEP)

Prof.Francesco Viglietti
www.in4matika.altervista.org



Concetti base



Information hiding

Nelle classi viste precedentemente, abbiamo utilizzato delle variabili d'istanza che avevano il livello di visibilità **public** con il quale il codice esterno alla classe poteva accedere direttamente ai loro valori.
In OOP è consigliabile garantire una maggiore "ristrettezza" riguardo l'accesso ai campi istanza della classe, cioè "mascherare" la natura e la struttura dei campi inclusi nella classe. L'utilizzatore della classe non deve interessarsi di come sono stati creati e/o manipolati gli attributi dell'oggetto che dovrà gestire all'interno di una applicazione, ma li dovrà solamente utilizzare.
I progettisti di una classe, nascondono, rendendoli privati, tutti (o quasi) le variabili d'istanza di una classe.

Classe rettangolo

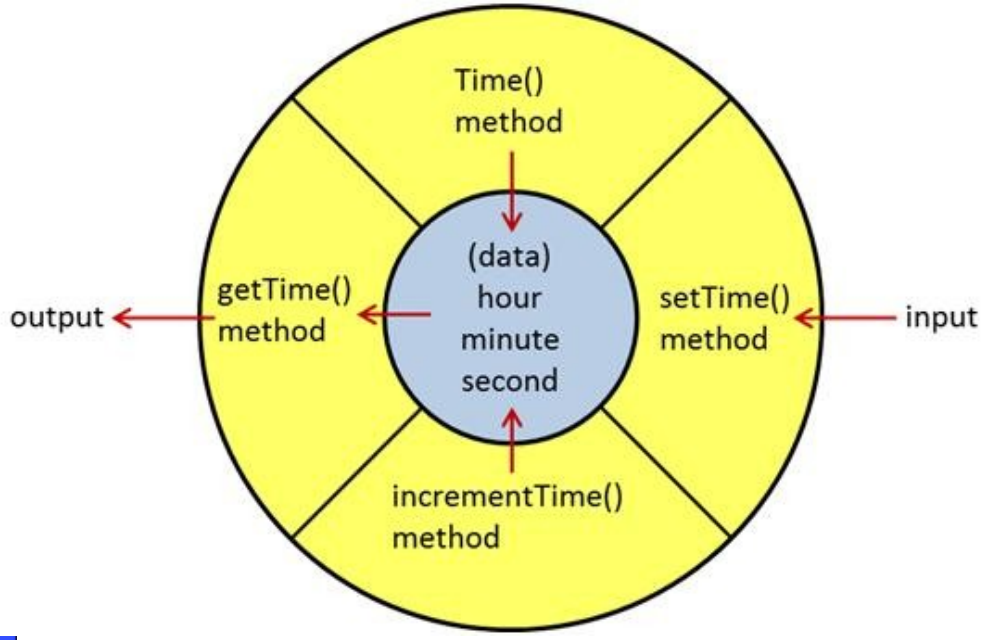
```
//classe che calcola area e perimetro di un generico rettangolo
public class Rettangolo{
//attributi
 public int larghezza
 public int altezza
//costruttore
public Rettangolo(){} //costruttore di default
public Rettangolo(int larghezza,int altezza){
  this.larghezza=larghezza;
  this.altezza=altezza;
 }
//metodi
public  int area(){
  return (larghezza*altezza);
 }
public  int perimetro(){
  return 2*(larghezza+altezza);
 }
}
```

A questo punto… la domanda nasce spontanea: "Se gli attributi vengono dichiarati privati, come si possono utilizzare all'interno di una applicazione se non possiamo accedervi?"

incapsulamento

Incapsulamento

Il termine **incapsulamento** indica la proprietà degli oggetti di incorporare al loro interno sia gli attributi che i metodi, cioè le caratteristiche e i comportamenti dell'oggetto.



Get e Set

Grazie ai metodi ***getter*** e ***setter*** siamo in grado di accedere ai campi istanza sia in lettura che in scrittura; questa è la sintassi:

```
public tiporestituito getnomeattributo(){
        return nomeattributo;
 }
public void setnomeattributo(tiporestituito parametro){
        nomeattributo=parametro;
 }
```

I metodi sono public, quindi accessibili dal codice esterno alla classe.
Le parole chiave get e set consentono l'effettivo accesso al campo istanza ed in particolare:

Get → saremo in grado di leggere il contenuto del campo. Essa ci restituisce il valore del campo.

Set → saremo in grado di assegnare un preciso valore al campo tramite il parametro

Classe rettangolo

```
//classe che calcola area e perimetro di un generico rettangolo
public class Rettangolo{
//attributi
 private int larghezza
 private int altezza
public Rettangolo(){}
public Rettangolo(int larghezza,int altezza){
  this.larghezza=larghezza;
  this.altezza=altezza;
 }
//getter e setter
public int getAltezza(){    return altezza; }
public int getLarghezza(){    return Larghezza; }
public void setAltezza( int altezza){    this.altezza=altezza; }
public void setLarghezza(int larghezza){    this.larghezza=larghezza; }
//metodi
public  int perimetro(){
  return 2*(larghezza+altezza);
 }
 public  int area(){
  return (larghezza*altezza);
 }
}
```

Interfaccia e implementazione

il termine interfaccia riferito alle classi:

L'**interfaccia** di una classe consiste nell'intestazione dei metodi pubblici, delle sue costanti pubbliche, insieme ai commenti che indicano al programmatore come usare i metodi e le costanti.

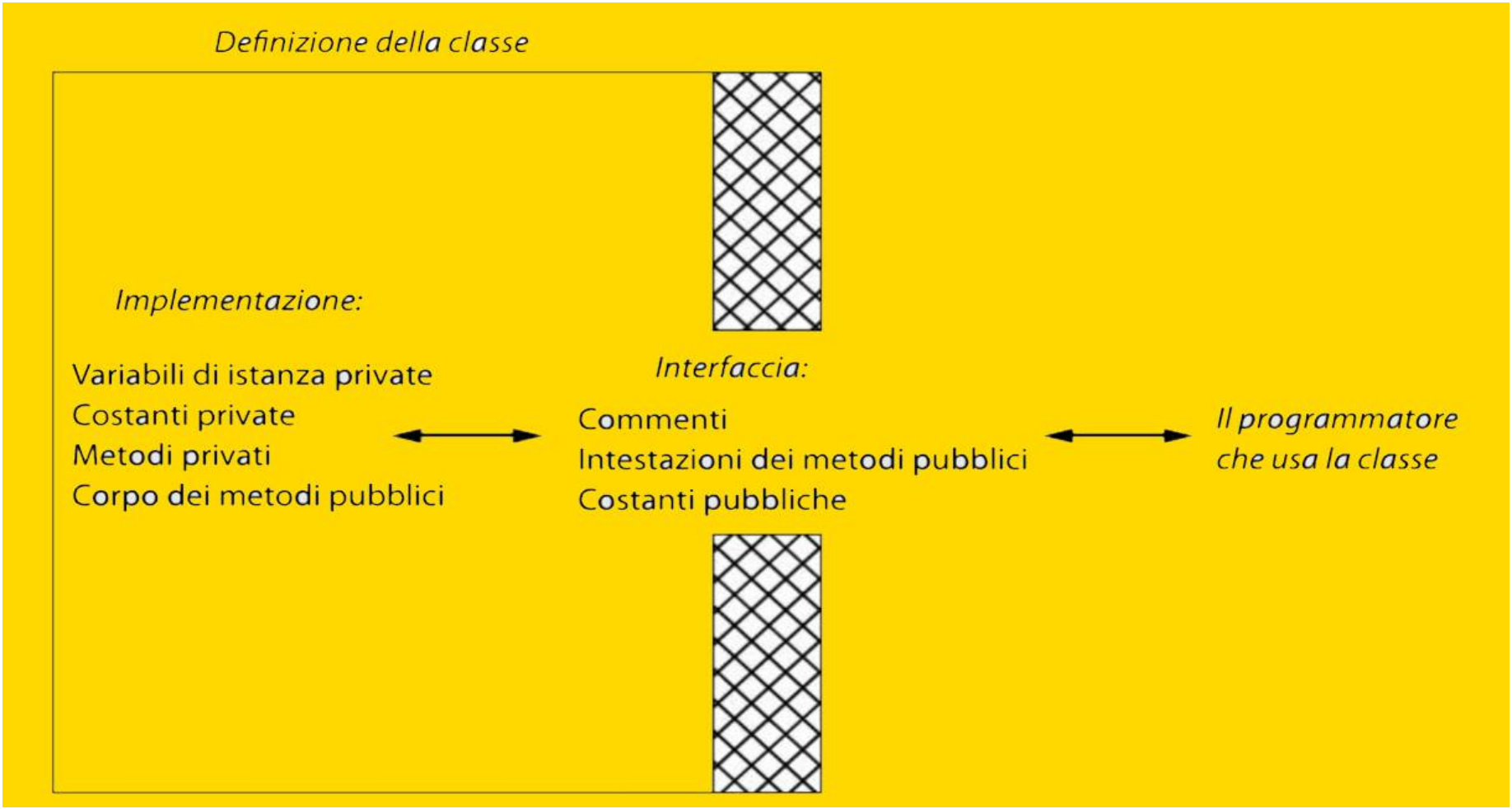
per esempio, l'interfaccia della classe *Rettangolo*:

```
public Rettangolo();
public int getAltezza();
public int getLarghezza();
public void setAltezza(int altezza);
public void setLarghezza(int larghezza);
public int area();
```

Quindi l'incapsulamento divide una classe in due parti: l'interfaccia e l'implementazione.

L'**implementazione** di una classe consiste in tutti gli elementi privati, principalmente le variabili d'istanza private e le definizioni dei metodi pubblici e privati

Definizione classe

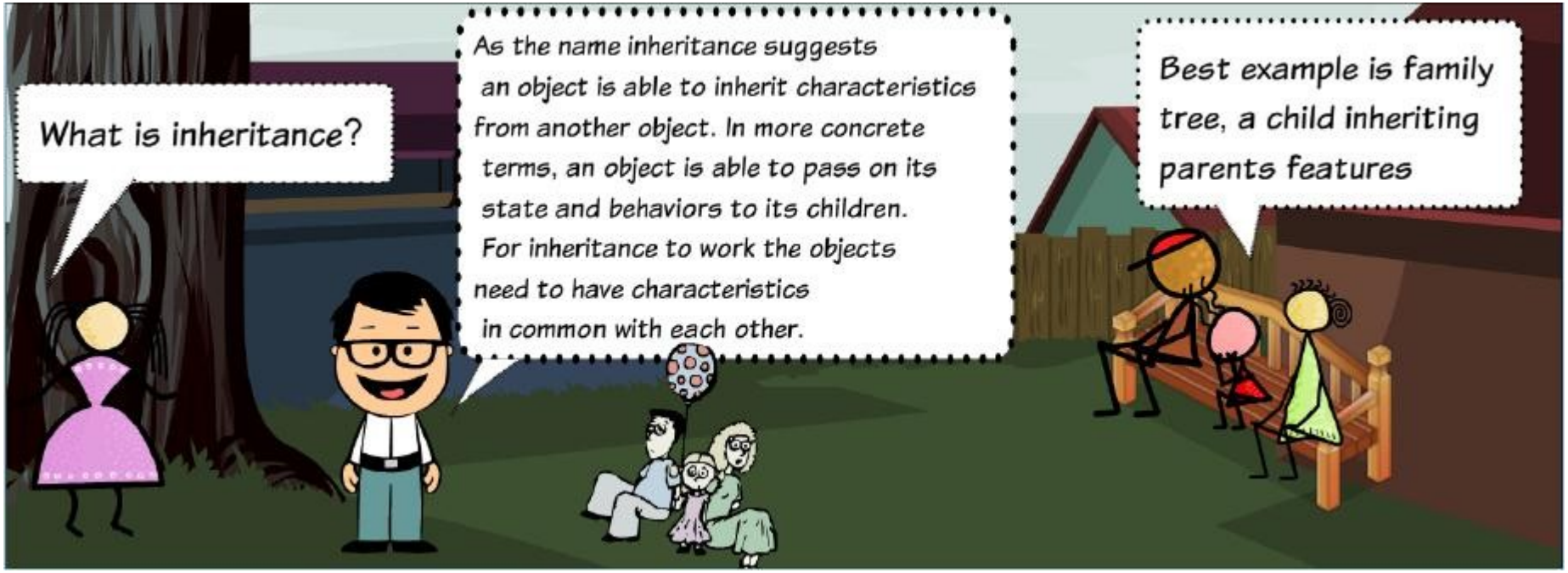


Regole per incapsulamento

- Si predisponga un commento prima della definizione della classe che descriva al programmatore cosa rappresenta la classe senza descrivere come lo fa. Se, per esempio, la classe rappresenta una somma di denaro, nel commento devono apparire termini quali Euro e centesimi e non il modo in cui questi sono rappresentati nella classe.
- Si dichiarino tutte le variabili di istanza della classe come private.
- Si forniscano metodi *get* pubblici per recuperare i dati in un oggetto. Si forniscano, inoltre, metodi pubblici per qualsiasi altra necessità di base di cui un programmatore potrebbe aver bisogno per gestire i dati della classe. Questi metodi potrebbero includere, per esempio, i metodi *set*.
- Si predisponga un commento prima di ogni intestazione di metodo pubblico che specifichi chiaramente come usare il metodo.
- Si rendano privati i metodi ausiliari.
- Si scrivano commenti all'interno della classe per descrivere i dettagli implementativi.

/** … */ per le interfacce, // per l'implementazione

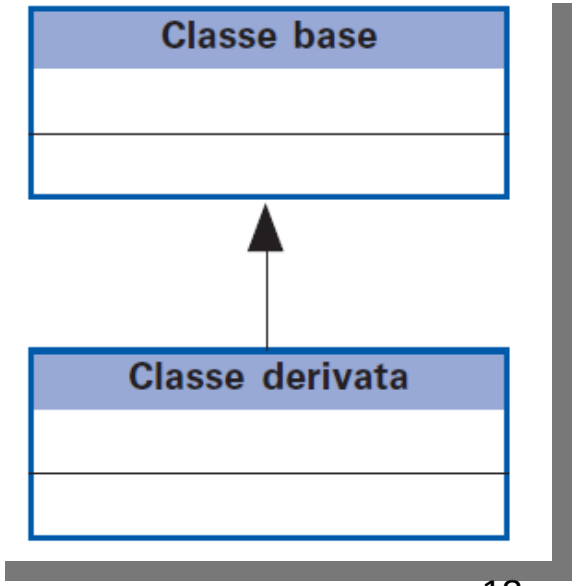
Ereditarietà



Ereditarietà

L'**ereditarietà** rappresenta la possibilità di creare nuove classi a partire da una classe già esistente (**classe base**). La nuova classe eredita tutti gli attributi e i metodi della classe base e può essere arricchita con nuovi attributi e nuovi metodi. La classe così ottenuta si chiama **classe derivata**.

Se si dispone già di una classe e si può adattarla al problema, perché riscriverla?

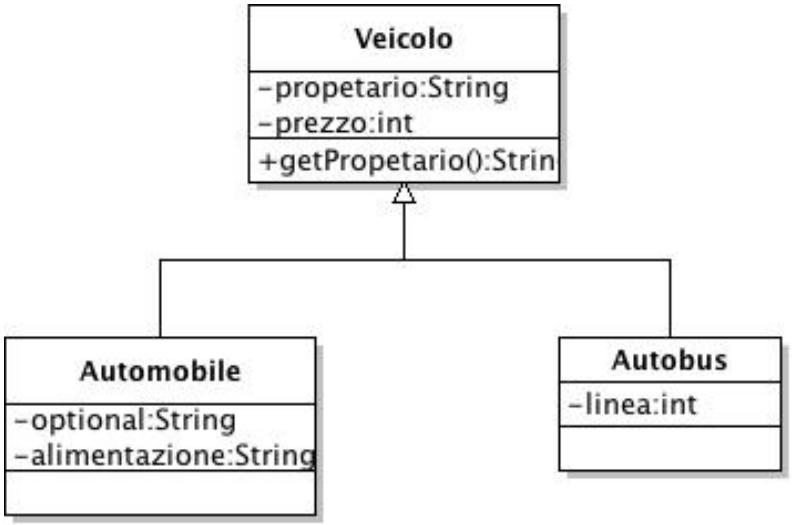


Sottoclasse e superclasse

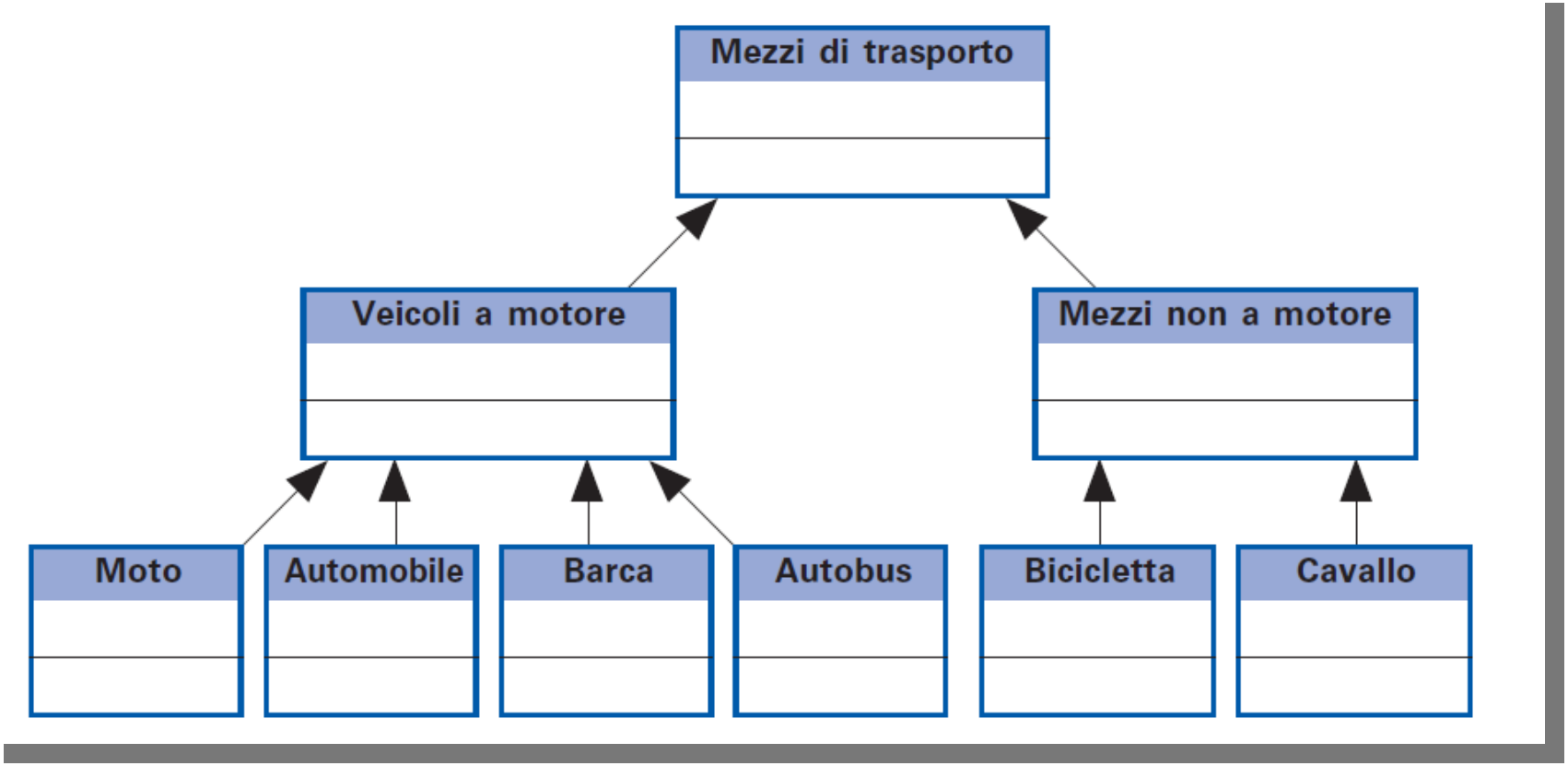
La classe che è stata derivata da un'altra usando l'ereditarietà prende il nome di **sottoclasse**.

La classe generatrice di una sottoclasse si chiama **superclasse** o *sopraclasse*.

Attenzione: gli attributi o i metodi privati non vengono ereditati



Grafo di gerarchia



Es. la classe Veicoli a motore eredita attributi e metodi dalla classe Mezzi di trasporto. La classe Barca eredita attributi e metodi dalla classe Veicoli a motore e quindi anche quelli di Mezzi di trasporto…..

Classe derivata

La nuova classe si differenzia dalla superclasse aggiungendo o modificando i metodi:

- per **estensione:** la sottoclasse aggiunge nuovi attributi e metodi che si aggiungono a quelli sella superclasse.
- per **ridefinizione:** la sottoclasse ridefinisce i metodi ereditati (**overriding** del metodo). Perciò si riscrive il codice del metodo utilizzando lo stesso nome.

  Esistono due tipi di ereditarietà: (java ha solo la singola)

- Singola → una sottoclasse deriva da un'unica superclasse.
- Multipla → una sottoclasse deriva da due o più superclassi, fondendo attributi e metodi delle superclassi.

Dichiarazione di una sottoclasse

```
class NomeSottoClasse extends NomeSopraClasse{
  . . .
}
```

La sottoclasse eredita tutti gli attributi e i metodi ad eccezione di quelli definiti private.

class *A{*
public int *numA;*
}
class *B* ***extends*** *A{*
public int *numB;*
}
…. un oggetto creato come istanza di B
B obj_b = ***new*** *B();*
Può accedere sia all'attributo numB che a quello numA.
La parola chiave ***super*** *è un oggetto speciale che fa riferimento alla superclasse della classe in cui viene utilizzata.*

L'oggetto super

Ereditare i costruttori

Se voglio solo ereditare il costruttore: definisco il costruttore con ***super(****parametri****)*** *come corpo:*

```
class NewPoint extends Point {
    NewPoint() {  super();   }
    NewPoint(int x, int y) {   super(x, y);   }
    NewPoint(Rectangle r) {
        this.x=r.x+r.width/2;
        this.y=r.y+r.height/2;
    }
}
```

```
class Point {
 int x,y;
 public Point(){}
 public Point(int x,int y) {
      this.x=x;
      this.y=y;
 }
}
```

Di solito, conviene ridefinire il costruttore.

La parola chiave super

Il costruttore della superclasse

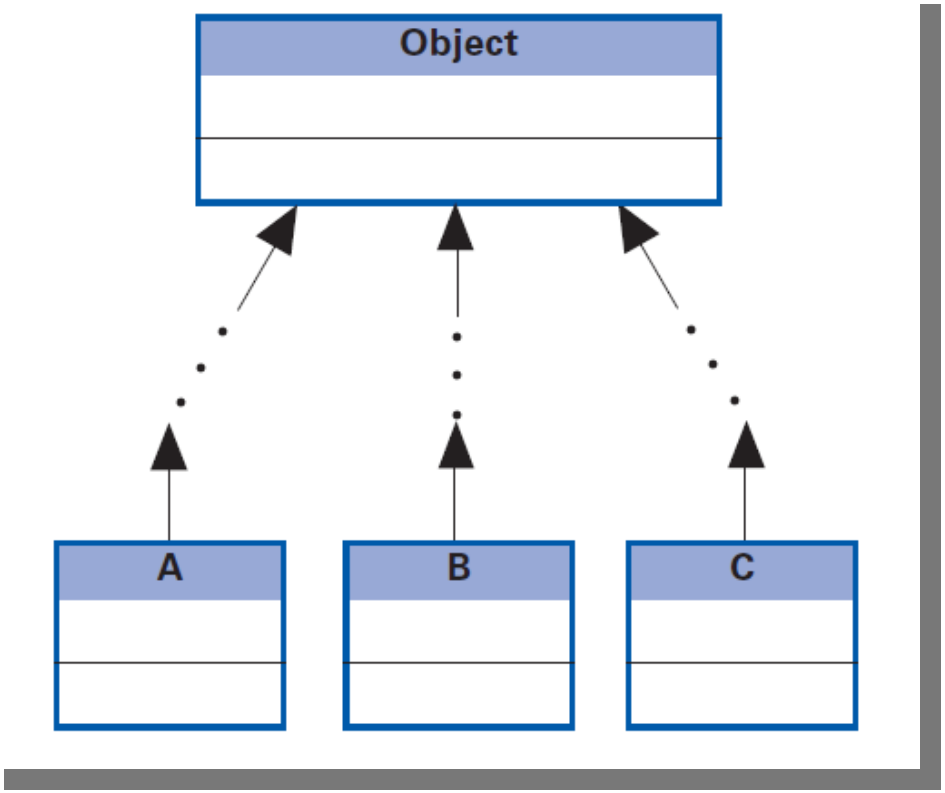
super(parametri) invoca un costruttore della classe che è stata estesa
Quando si invoca il costruttore della superclasse, questa deve essere la prima istruzione del metodo.

```
class Point3D extends Point {
  int z;
 Point3D() {  }
 Point3D(int x, int y, int z) {
    super(x, y);
    this.z=z;
  }
}
```

Quando un metodo non inizia con super(argomenti), si assume automaticamente super(), ossia il costruttore senza argomenti.

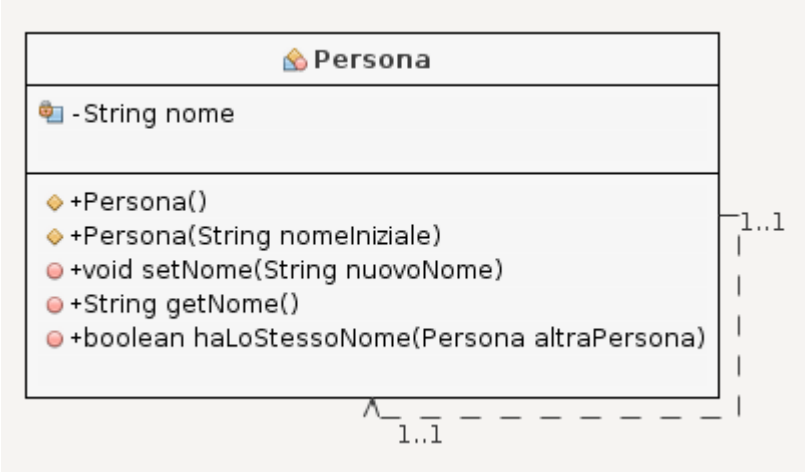
Classe Object

Classe da cui vengono derivate tutte le altre



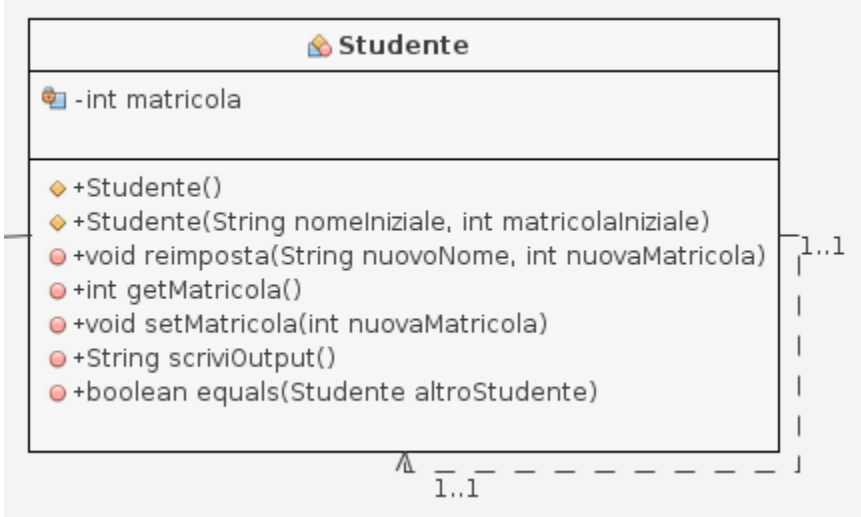
Tutte le nuove classi saranno sottoclassi di Object.
Se una classe viene dichiarata usando la parola chiave ***final,*** essa non potrà generare sottoclassi.
Final viene usata anche nei metodi che quindi non potranno essere usati nelle sottoclassi, come se fossero private. (la differenza è che possono essere visti da altre classi).

Esempio



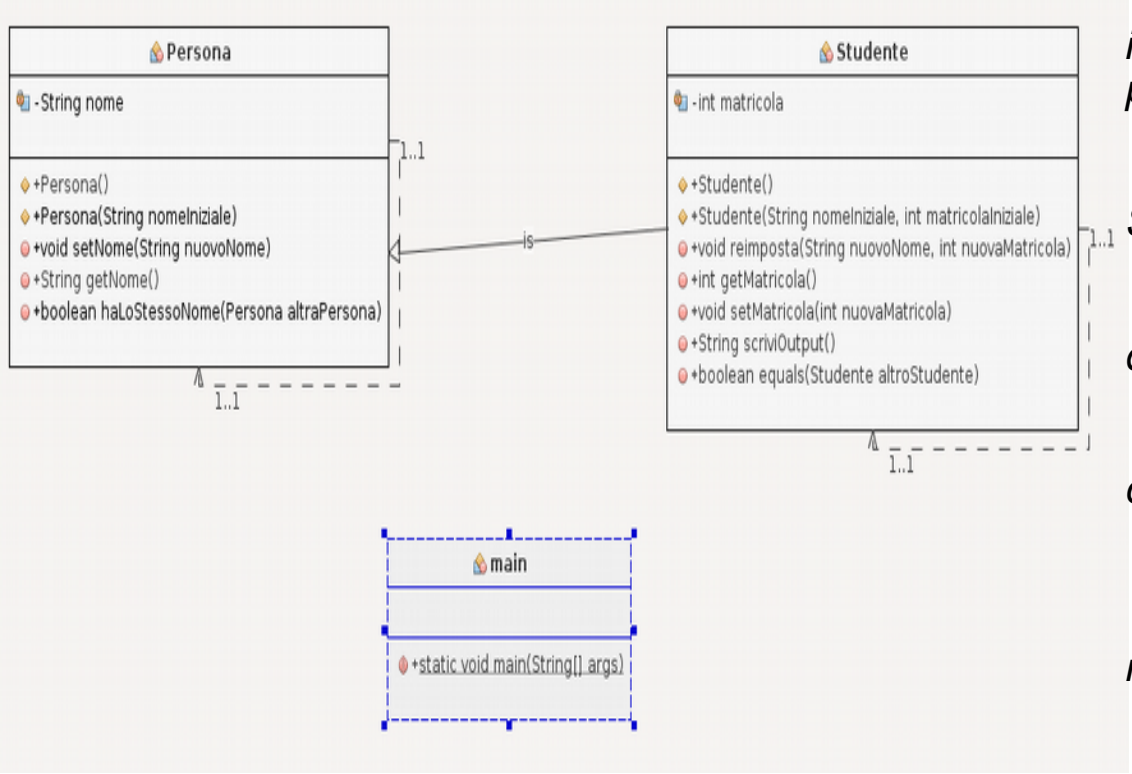

```
public class Persona {
//attributi
private String nome;

//costruttori
public Persona() { }

public Persona(String nomeIniziale) { nome =
nomeIniziale; }

public void setNome(String nuovoNome) { nome =
nuovoNome; }

public String getNome() { return nome; }

public boolean haLoStessoNome(Persona altraPersona){
return this.nome.equalsIgnoreCase(altraPersona.nome); }
}
```

Esempio




```
public class Studente extends Persona {
//attributi
private int matricola;
//costruttori
public Studente() { super(); matricola = 0;//Ancora
nessuna matricola }
public Studente(String nomeIniziale, int
matricolaIniziale) { super(nomeIniziale); matricola =
matricolaIniziale; }
//metodi
public void reimposta(String nuovoNome, int
nuovaMatricola) { setNome(nuovoNome); matricola =
nuovaMatricola; }
public int getMatricola() { return matricola; }
public void setMatricola(int nuovaMatricola) { matricola
= nuovaMatricola; }
public String scriviOutput() {
String tmp=getNome()+"Matricola: ";
return tmp+matricola; }
public boolean equals(Studente altroStudente) { return
this.haLoStessoNome(altroStudente) && (this.matricola
==  altroStudente.matricola); }
}
```

Esempio



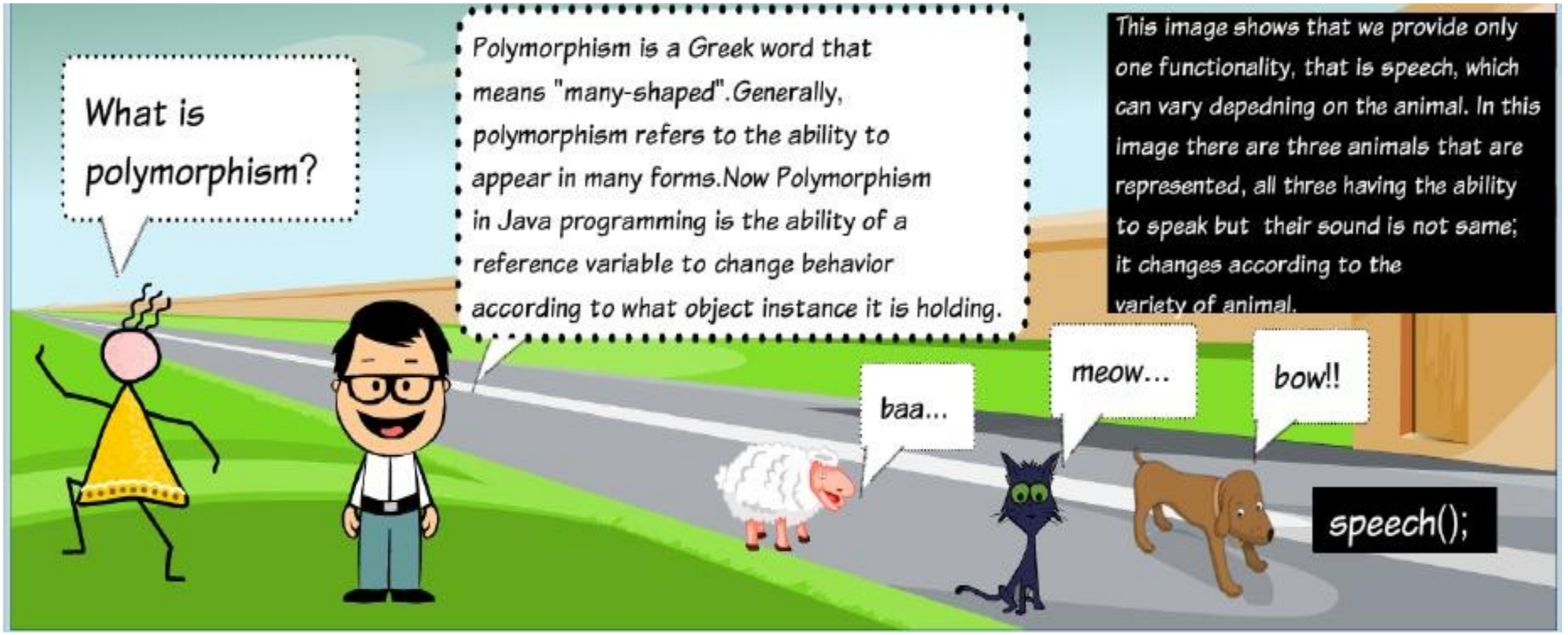

```
import java.util.*;
public class main {
    public static void main(String []args){
        Scanner in=new
Scanner(System.in);
        Studente s=new Studente();
        System.out.print("Inserisci il nome
dello studente: ");
        String txt=in.next();
        System.out.print("Inserisci il
cognome dello studente: ");
        txt=txt+" "+in.next();
        s.setNome(txt);
        System.out.print("Inserisci la
matricola dello studente: ");
        int matricola=in.nextInt();
        s.setMatricola(matricola);
        System.out.println("Stampa
riepilogo:\n"+s.scriviOutput());
            }
}
```

esercizio

Definire la nuova classe NonLaureato, che viene ereditata dalla classe Studente. Quindi l'oggetto della classe NonLaureato possiede tutto ciò che è pubblico della classe Studente, ed essendo Studente derivata dalla classe Persona a sua volta possiede tutto ciò che è pubblico di Persona. La classe NonLaureato avrà come attributo ***annoDiCorso***, si dovrà ridefinire il metodo ***reimposta*** (serve per modificare l'oggetto)***,*** e si dovrà ridefinire il metodo ***equals*** (serve per poter verificare se due istanze della classe sono uguali)

esercizio7

Polimorfismo



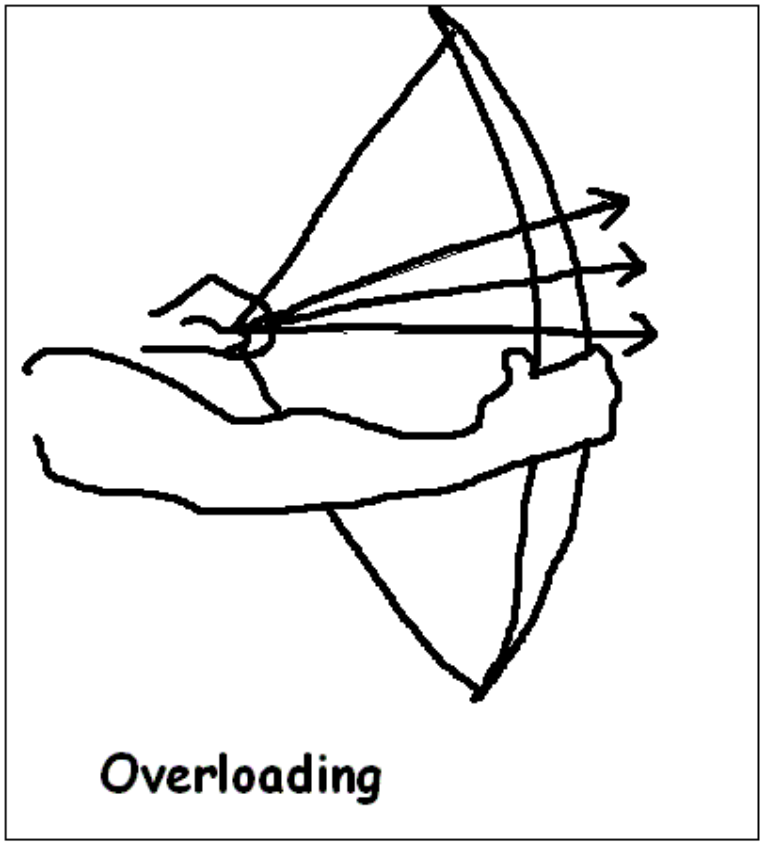
Polimorfismo

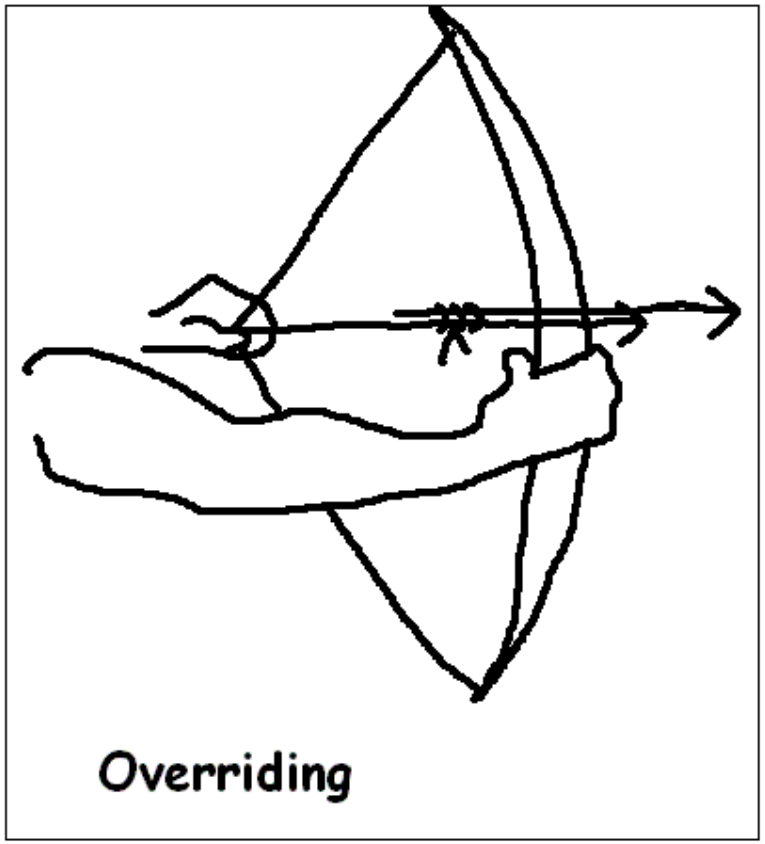
Il **polimorfismo** indica la possibilità per i metodi di assumere forme, cioè implementazioni, diverse all'interno della gerarchia delle classi.

Esistono due tipi di polimorfismo:

- l'**overriding**, o *sovrapposizione* dei metodi,
- l'**overloading**, o *sovraccarico* dei metodi.

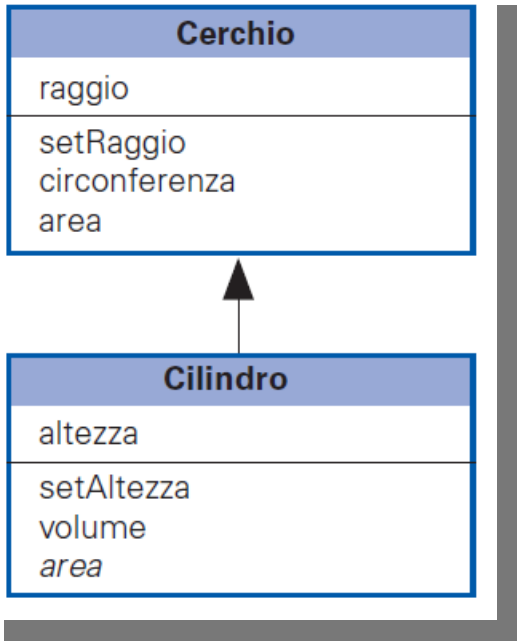
Polimorfismo





Overriding

- **Overriding:** ridefinisce, nella classe derivata, un metodo ereditato con lo scopo di modificarne il comportamento. Il nuovo metodo deve avere lo stesso nome e gli stessi parametri del metodo che viene sovrascritto.




```
public double area()
{
        double supBase, supLater;
        supBase=super.area()*2;
        supLater=circonferenza()*altezza;
        return supBase + supLater;
}
```

Overloading

- L’**overloading** di un metodo è la possibilità di utilizzare lo stesso nome per compiere operazioni diverse. Solitamente si applica ai metodi della stessa classe che si presentano con lo stesso nome, ma con un numero o un tipo diverso di parametri.