Programmazione I

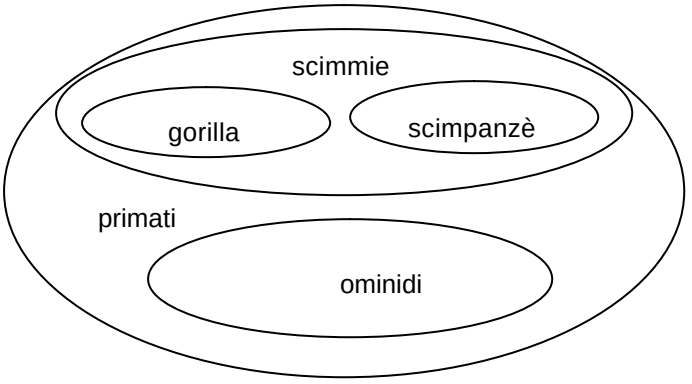
A.A. 2002-03

Programmazione Orientata agli Oggetti:

EREDITARIETA' (Lezione XXVIII)

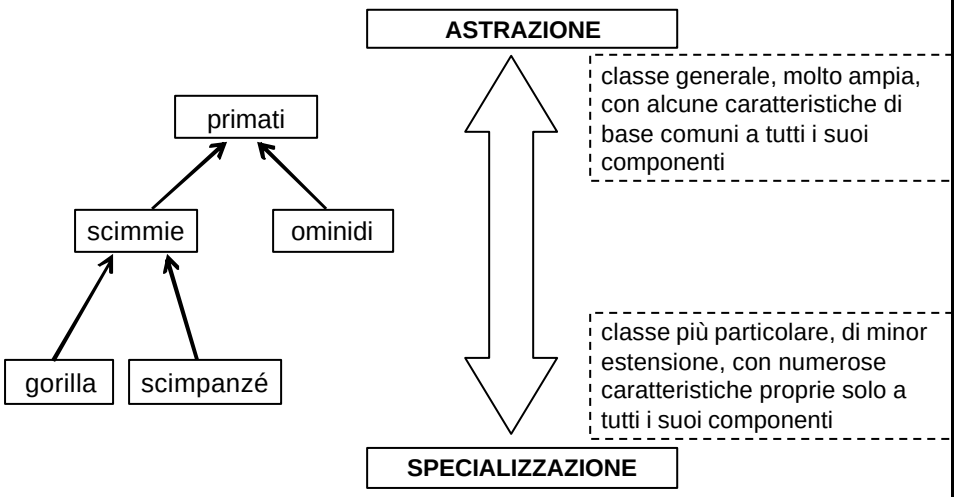
Prof. Giovanni Gallo
Dr. Gianluca Cincotti
Dipartimento di Matematica e Informatica
Università di Catania
e-mail : {gallo, cincotti}@dmi.unict.it

partiamo da lontano

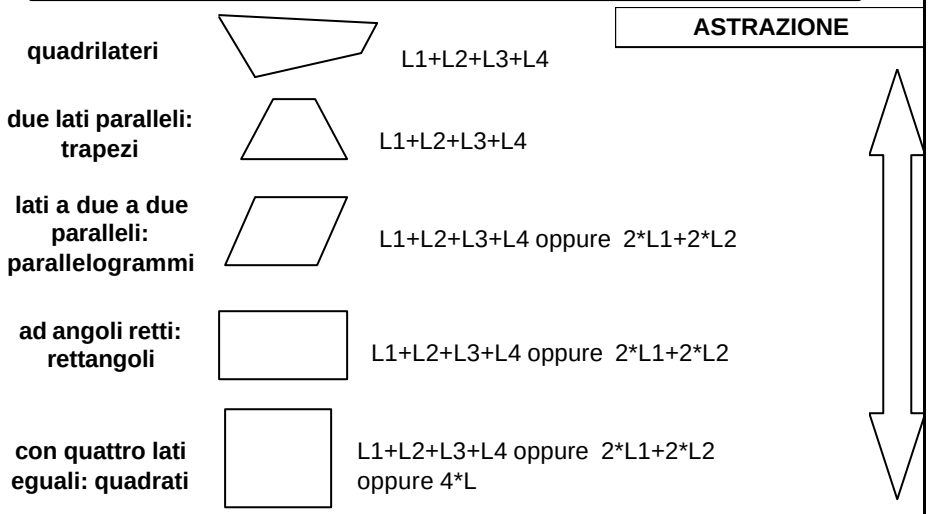


Classificare è una delle attività intellettuali più comuni ed importanti

Dall'astratto (generale) verso il concreto (particolare)



La eredità per i quadrilateri e il calcolo del perimetro



Non è raro vedere la stessa cosa negli oggetti della programmazione

Se scriviamo un metodo per tutti i quadrilateri perché non riutilizzarlo quando dobbiamo lavorare con i quadrati?

Oppure, valutare se sostituirlo (over-riding) con uno più efficiente? (come per la formula del perimetro)

E se un oggetto è una specializzazione di una classe di oggetti più generici potrà eventualmente avere "comportamenti" e proprietà aggiuntive
(per esempio la possibilità di ruotare di 90 gradi sovrapponendosi esattamente a se stesso come il quadrato ma non come il rettangolo)

Nella Programmazione OO: eredità o estensione

```
class quadrilatero
{ …
  p=L1+L2+L3+L4
}
```

```
class trapezio extends quadrilatero
{ …
  p=L1+L2+L3+L4
}
```

```
class parallelogramma extends trapezio
{ …
  p=2*L1+2* L2     //OVERRIDING
}
```

```
class rettangolo extends parallelogramma
{ …
  p=2*L1+2* L2
}
```

```
class quadrato extends rettangolo
{ …
  p=4*L     //OVERRIDING
}
```

catena dell'eredità

quadrilatero* è SUPERCLASSE di *trapezio, parallelogramma, rettangolo, quadrato.

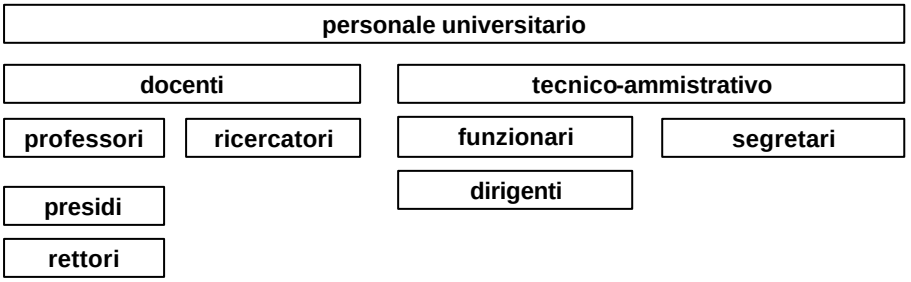
***quadrilatero* non è SOTTOCLASSE degli altri**

quadrato* è SOTTOCLASSE di *rettangolo, parallelogramma, trapezio, quadrilatero

***quadrato* non è SUPERCLASSE degli altri**

GERARCHIA d'eredità

L'esempio precedente mostrava una CATENA. In realtà si ha generalmente una situazione più ramificata: un albero. Prendiamo una gerarchia di "tipi" che descriva il personale universitario.



Nonostante il linguaggio possa essere fuorviante essere in una sottoclasse non vuol dire essere meno "potenti".
ANZI: le sottoclassi hanno più funzionalità e informazioni delle superclassi.

La parola chiave "extends"

Partiamo dall'analisi dell'esempio1.java

```
class triangolo
{…}
```

Nuova parola chiave!

```
class isoscele extends triangolo
{…}
```

```
class equilatero extends isoscele
{…}
```

La parola chiave "super"

Sempre dall'esempio1.java:

- La costruzione di **`isoscele`** si "appoggia" alla costruzione di **`triangolo`**.
- La costruzione di **`equilatero`** si "appoggia" alla costruzione di **`isoscele`**

La parola chiave "**`super`**" viene usata per indicare i metodi della classe "progenitrice" e in particolare super(…) indica il metodo costruttore di tale classe.

ATTENZIONE! super(…) deve essere il PRIMO comando del metodo costruttore della classe derivata**.**

Ancora: Potrei "forzare" un isoscele ad usare il metodo perimetro di triangolo usando la chiamata super.perimetro() anziché perimetro().

Sovra-scrivere i metodi

Tutti i discendenti ereditano **tutti** gli attributi e i metodi dei genitori.

Non sarebbe quindi stato necessario ridefinire **`perimetro()`** per le classi derivate così come non è utile ripetere il codice per **`getInfo().`**

Però:

- Il perimetro del triangolo isoscele può essere calcolato in maniera differente che per i triangoli più generali.
- Lo stesso accade per i triangoli equilateri.

E' quindi comodo sovra-scrivere (OVERRIDING) i metodi che possono essere migliorati in casi speciali.

Per farlo è sufficiente RIPETERE LA DICHIARAZIONE del metodo di un progenitore e farla seguire da un nuovo body.

Metodi non modificabili: il qualificatore final

E' possibile qualificare una classe, o qualcuno dei suoi metodi o qualcuna delle sue variabili con l'aggettivo: **final**

- Una classe final non può essere "estesa";
- Un metodo final non può essere "sovrascritto";
- Una variabile final non può cambiare valore;

Attenzione: dichiarare un metodo o una variabile "static" implicitamente le qualifica come "final".

Public, private o …protected

Si osservi l'esempio2.java. La sottoclasse **`quadrato`** è derivata dalla superclasse **`rettangolo`**. Quale accesso dare alle variabili di istanza di rettangolo?

public:
sarebbero visibili da tutti gli oggetti, violazione di Encapsulation Principle!

private:
se quadrato volesse utilizzarle non le vedrebbe…

Soluzione proposta dai creatori di JAVA:

protected:
visibile a tutte le classi derivate, alle altre classe dello stesso "package" (per esempio a quelle definite nello stesso file) e non visibile alle altri classi.

Compatibilità dei tipi: arance+pesche+biscotti = non si può!

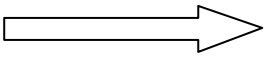
JAVA è un linguaggio *TIPATO*: non accetta che una variabile di un tipo contenga informazioni, o faccia riferimento, a oggetti o dati di altro tipo.

Questo è vero sia per i dati predefiniti che per i dati "costruiti", come gli oggetti.

REGOLA GENERALE: Il riferimento ad un tipo di oggetto non può essere ad un oggetto di classe diversa.

Ma oggetti in una catena di eredità sono di "tipo complentamente diverso"? NO!

C'è dunque una eccezione alla regola…



Compatibilità dei tipi: arance+limoni+clementine =agrumi

I triangoli isosceli sono particolari triangoli.

I triangoli equilateri sono particolari triangoli isosceli e particolari triangoli.

Non è vero il viceversa: un generico triangolo non è necessariamente nè isoscele nè equilatero.

In JAVA è possibile che un riferimento relativo alla classe di un progenitore punti ad un discendente. Non è possibile il viceversa!

CORRETTO

```
triangolo A =
       new isoscele(10,30);
isoscele B =
       new equilatero(20);
triangolo C =
       new equilatero(30);
```

ERRATO

```
isoscele A =
       new triangolo(10,30,22);
equilatero B =
       new isoscele(20,12);
equilatero C =
       new triangolo(30,24,27);
```

POLIMORFISMO

JAVA non ammette "impostori"!

Se si dichiara che la variabile A sarà un triangolo e poi si assegna ad A un triangolo isoscele si sta mantenendo la parola eventualmente garantendo più proprietà di quelle richieste. Questo comportamento "*flessibile*" di JAVA (e degli altri linguaggi OO) è detto tecnicamente "**polimorfismo**": gli oggetti possono assumere diverse "forme" dentro la propria gerarchia

Se invece si dichiara una variabile B come isoscele e poi si assegna un triangolo che non ha due dei suoi lati di eguale misura: è una bugia!

E non passa inosservata all'occhio vigile del compilatore!!

Oggetti della medesima gerarchia in un solo array

Partiamo dall'esempio2.java. In esso è definita una classe **`"rettangolo"`** e da essa viene derivata una classe **`"quadrato".`**

Il main prevede la creazione di un array A di oggetti di tipo **`rettangolo`** con una piccola variante: gli oggetti vengono generati a caso e nel 50% dei casi si generano oggetti di tipo **`quadrato`** anziché **`rettangolo.`**

Attenzione: al momento della creazione del codice non si può sapere quali elementi dell'array saranno **`quadrato`** e quali **`rettangolo.`**
Il compilatore non può quindi far nulla. La situazione dovrà essere gestita dall'interprete JVM.

Nessun problema però: un oggetto **`quadrato`** è un particolare tipo di **`rettangolo`** e secondo le regole di casting viste prima la JVM usa il riferimento ad un rettangolo per riferirsi ad un quadrato.

Il viceversa avrebbe ovviamente prodotto un errore rilevato già dal compilatore!

CASTING esplicito di oggetti

```
CORRETTO:
quadrato A = new quadrato(20);
rettangolo B = (rettangolo)A;
rettangolo C= A; // il cast esplicito è facoltativo
```

Un oggetto di una sottoclasse viene trasformato da un "cast" in un oggetto di una super-classe, non c'è pericolo di perdere informazioni!

```
ERRATO:
rettangolo A = new rettangolo(20,23);
quadrato B = A; // produce errore di compilazione
quadrato C =(quadrato)A
// nessun errore di compilazione ma…
// possibile errore a run time!
```

Un oggetto di una superclasse viene trasformato da un "cast" in un oggetto di una sotto-classe: possibile perdita di informazione!

```
INSENSATO:
rettangolo A = new rettangolo(20,23);
isoscele B = (isoscele)A;
```

Un oggetto di classe viene trasformato da un "cast" in un oggetto di una classe completamente scorrelata: non ha senso!

Di che classe è questo oggetto?
Il metodo getClass()

E' un metodo predefinito di tutti gli oggetti di JAVA.
Un esempio di sua chiamata è:

```
rettangolo A = new rettangolo(12,13);
quadrato B = new quadrato(34);
rettangolo C= B;
System.out.println( " classe di A = "+A.getClass());
System.out.println( " classe di B = "+B.getClass());
System.out.println( " classe di C = "+C.getClass());
```

Esso produce:

```
classe di A = class rettangolo
classe di B = class quadrato
classe di C = class quadrato
```

Di che classe è questo oggetto?
*l'operatore "**istanceof**"*

E' un operatore predefinito di JAVA.
Ha la seguente forma:

NOME_OGGETTO istanceof NOME_CLASSE

Esso restituisce il booleano **true** se l'oggetto appartiene alla classe.

Esso restituisce **false** se l'oggetto non appartiene alla classe

JAVA ha altri elementi che gli permettono di scrivere classi che agiscono sulle classi… ma non possiamo occuparcene in questo corso.

vedi ancora esempio2

In una gerarchia: più metodi con la stessa firma

Proseguiamo nell'esame dell'esempio2.

Adesso, in un unico ciclo for vogliamo calcolare l'area dei quadrilateri conservati nell'array A.

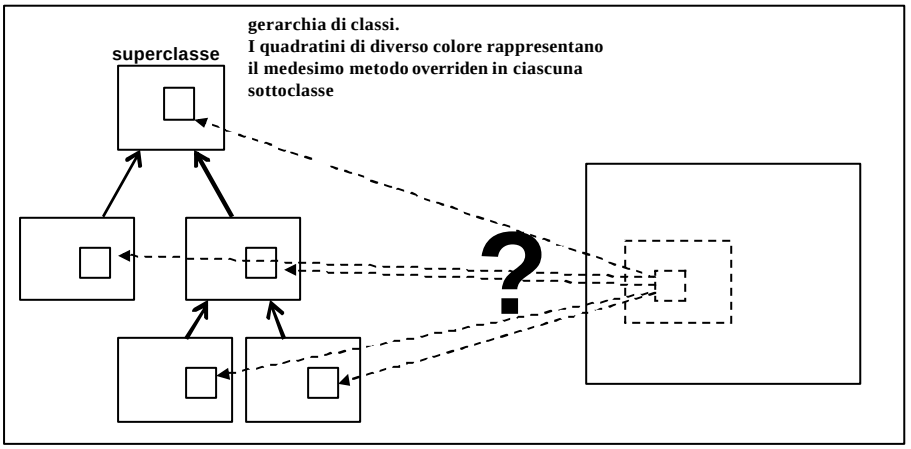
Il comando sarà dunque:

```
A[i].area();
```

Ma quale metodo area viene invocato?

La JVM opera con la tecnica del "BINDING DINAMICO": il codice di un metodo viene "legato" agli oggetti ***nel momento della esecuzione*** (dinamicamente) dalla JVM.

Ciò consente di chiamare per i rettangoli il metodo che computa **`base*altezza`** e per i quadrati il metodo che calcola il **`lato*lato`**.



Possiamo scrivere una classe che usi oggetti della superclasse.

Essa funzionerà anche usando oggetti delle sottoclassi.

NON E' NECESSARIO SAPERE AL MOMENTO DELLA COMPILAZIONE A QUALE CLASSE APPARTENGANO GLI OGGETTI INVOCATI.

Essi vengono determinati *dinamicamente* al momento della esecuzione: si ha il "binding dinamico".

La classe COSMICA: ***Object***

JAVA prevede una classe dalla quale TUTTE le altre classi vengono derivate:
Object

Ogni altra classe deriva, entro una qualche gerarchia, da **Object.**

Se una classe viene definita senza essere esplicitamente derivata da altre, essa è implicitamente da considerarsi una estensione della classe **Object.**

Object è una vera classe e possiede alcuni metodi importanti che sono ereditati da TUTTI gli oggetti. Essi sono numerosi (vedere la documentazione).

A noi interessano:

•il metodo di controllo e debugging "toString";

•il metodo di confronto "equals".

Il metodo toString()

Il metodo **toString()** della classe Object restituisce una String;

Esso è invocato implicitamente tutte le volte che si fa un cast di un oggetto in una String. Per esempio:

```
pippo A = new pippo();
System.out.println(pippo);
```

In questo caso l'oggetto A, della classe pippo, viene implicitamente "cast" a un oggetto di tipo String ed è il risultato di tael operazione che viene stampato.

Il metodo toString() originale è però molto scarno di informazioni;
se invocato per un qualunque oggetto restituisce il nome della classe cui appartiene l'oggetto seguito dalla locazione di memoria in cui l'oggetto è memorizzato. Non è molto utile così!

Overriding toString()

Il meccanismo è sempre lo stesso dell'overriding:

dare una nuova definizione del metodo e richiedere che se invocato vengano stampate informazioni complete sull'oggetto.

Vedi esempio3.java

toString() è un metodo prezioso per correggere e "tracciare" il comportamento del proprio software! Usatelo!!

Il metodo equals(Object unOggetto)

Nella sua forma originale (così come definito dentro la classe Object) esso consente di confrontare due oggetti e restituisce il booleano true se l'oggetto che chiama il metodo e l'oggetto passato come parametro sono IDENTICI, cioè se essi indicano con esattezza la stessa locazione di memoria.

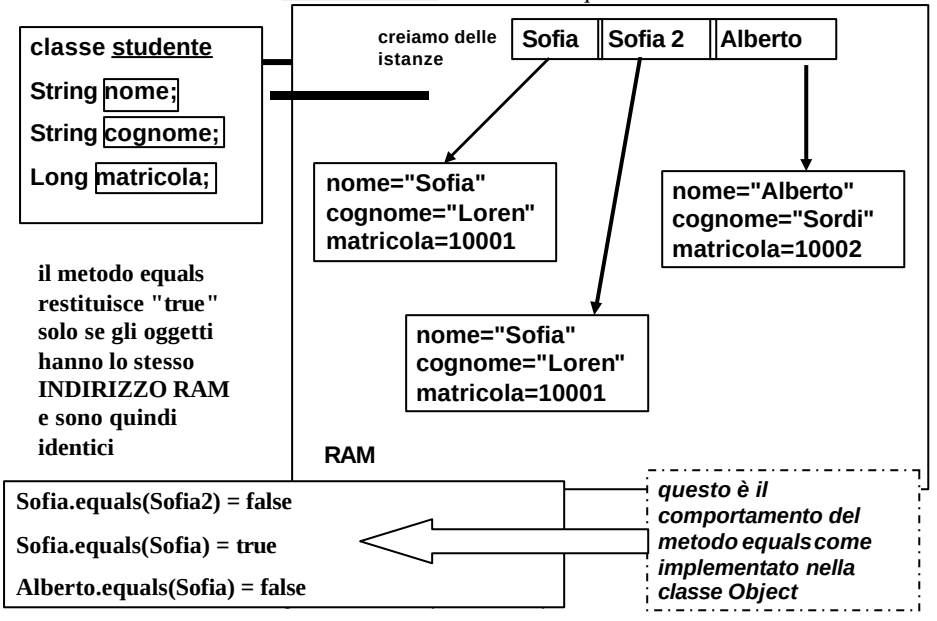
Questo però il più delle volte non è particolarmente utile.

Per esempio le stringhe a="pippo" e b="pippo"

sono LOGICAMENTE eguali anche se corrispondendo a due variabili differenti sono conservate in due locazioni di memoria diverse.

Se vogliamo che l'eguaglianza che il metodo equals controlla sia quella LOGICA, dobbiamo farne un overriding.

Consideriamo la classe "studente" con tre variabili di istanza.
La classe "studente" non ridefinisce il metodo "equals"



Overriding ***equals*** *(nel caso della classe "studente")*

```
public boolean equals(Object unOggetto)
{ // prima vediamo se gli oggetti sono identici,
  //cioè se essi occupano la stessa locazione di memoria
if (this = = unOggetto) return true;
  // l'altro possibile caso è che unOggetto non sia stato costruito
  // e quindi sia nullo, e quindi diverso dall'oggetto "studente"
  // che invoca il test equals(…)
if (unOggetto = = null) return false;
  // perché due oggetti siano eguali debbono essere nella stessa
  // gerarchia di eredità. Se così non è essi sono certamente diversi.
  // Confrontiamo dunque le loro classi, usando getClass()
if (this.getClass()!=unOggetto.getClass() return false;
  // poiché sappiamo che unOggetto è nella gerarchia di "studenti"
  // possiamo fare un cast e confrontare i relativi campi
       studente s=(studente)unOggetto;
        return ((this.nome==s.nome)&&
                 (this.cognome==s.cognome)&&
                 (this.matricola==s.matricola));
}
```


Fine