Programmazione Orientata agli Oggetti

Clausole import
Ereditarietà tra classi
Dynamic Binding
Polimorfismo

Libero Nigro

Forme di import

- import java.util.Iterator;
- import java.util.*;

- import poo.util.Mat;
- import poo.util.*;

- Importazione static:
- import static java.lang.Math;

  dopo questo, i metodi di Math sono utilizzabili nel ns programma senza il prefisso di classe: sqrt(12.34); anziché Math.sqrt(12.34);

Alcuni package della libreria di Java:

java.lang	importato di default
java.util	
java.io	
java.net	
java.math	
….	

Importazione qualificata e risoluzione conflitti

- Specie quando si fanno **import path-package.*,** ma non solo, sono possibili *clash di nomi*, nel senso che nel ns codice Java possono risultare presenti due o più classi con lo stesso nome, esportate da package diversi
- La risoluzione di questi conflitti si può ottenere spesso con l'import della classe specifica che si intende usare
- Meglio è la **importazione qualificata**, ossia l'importazione realizzata in modo esplicito nel contesto d'uso della classe e/o del metodo desiderati:

- System.out.println(java.util.Arrays.toString(v));
- java.lang.Comparable c=new Data();
- java.util.LinkedList l=new java.util.LinkedList();

Ereditarietà tra classi

- Una proprietà importante delle classi di Java è la possibilità di progettare una nuova classe per *estensione* di una classe esistente, dunque per *differenza*. Tutto ciò permette di concentrarsi solo sulle novità introdotte dalla nuova classe e di ereditare quelle che si mantengono inalterate, favorendo in tal modo la *produttività* del programmatore.
- Di seguito si considera una classe Disco programmata come una particolarizzazione della classe Punto. È intuitive che un disco possa essere visto come un caso particolare di Punto, in cui oltre al centro, c'è il raggio e proprietà conseguenti.
- La classe Disco "incorpora" tutto quanto proviene da Punto (variabili di istanza e metodi). In più, introduce nuove variabili, metodi etc.

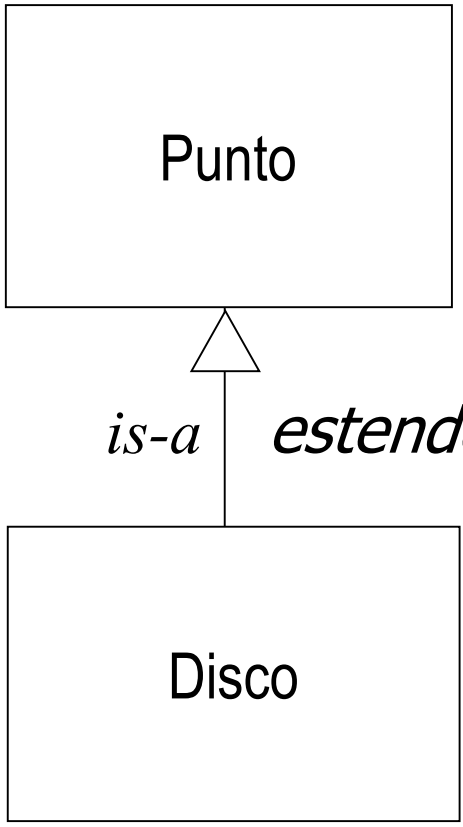
```
package poo.geometria;

public class Disco extends Punto{
    private double raggio;
    public Disco( double raggio ){//centro nell'origine
            super(); //evoca il costruttore di default di Punto. Si può anche omettere
            this.raggio=raggio;
    }
    public Disco( double x, double y, double raggio ){
            super(x,y);
            this.raggio=raggio;
    }
    public Disco( Punto p, double raggio ){ super(p.getX(),p.getY());
    this.raggio=raggio; }
    public Disco( Disco d ){ super(d.getX(),d.getY()); this.raggio=d.raggio; }
    public double getRaggio( ){ return raggio; }
    public double perimetro(){ return 2*Math.PI*raggio; }
    public double area(){ return raggio*raggio*Math.PI; }
    public String toString(){ return "Disco di raggio "+raggio+" e centro
    "+super.toString(); }
}//Disco
```

Si nota l'uso del pronome super per evocare un costruttore della super classe o per invocare il metodo toString di Punto.

In un main si può avere:

```
Punto p=new Punto(3,5);
…
Disco d=new Disco( 2, 7, 4 ); //<2,7> sono le coordinate del centro

d.sposta( 5, 6 ); //un'operazione di Punto invocata sul disco

double dist=p.distanza(d); //calcola la distanza tra il punto p ed il centro del disco

double xc=d.getX(); //metodo ereditato da Punto
double r=d.getRaggio(); //metodo proprio di Disco

System.out.println(d); //scrive i dati del disco d, invocando il toString
```

Relazione di Ereditarietà in UML



generale

Super classe (o classe base)

particolare

Sotto classe (o classe derivata)

Un disco **è** un punto, ma non viceversa

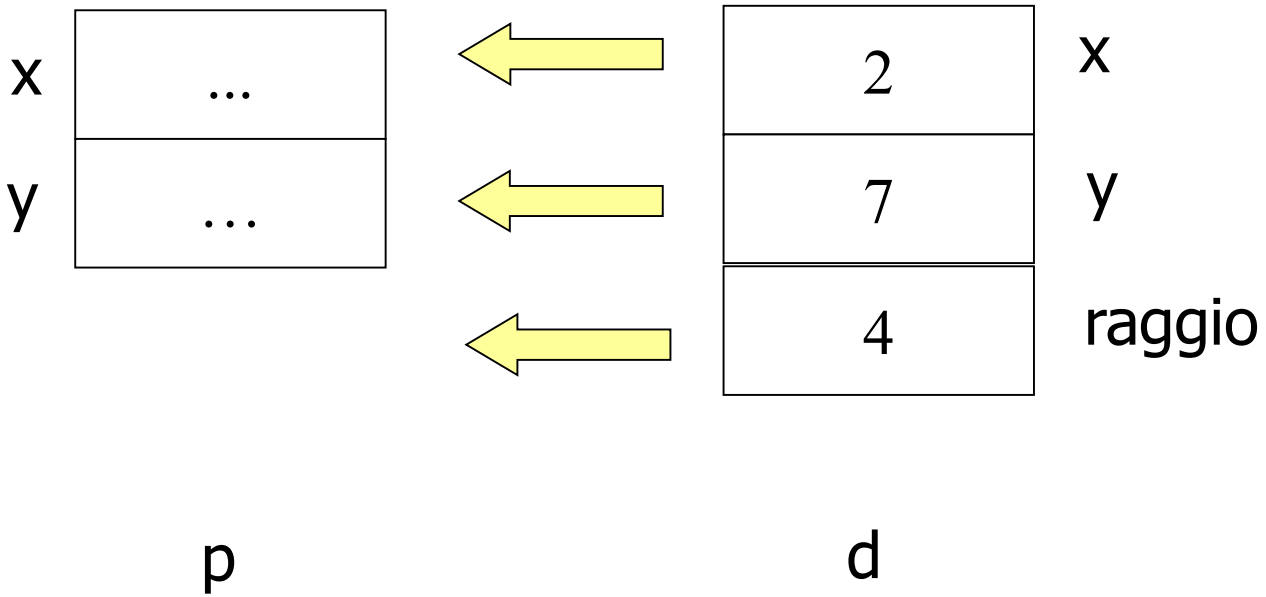
Sostituibilità dei tipi (principio di Liskov)

- La relazione di ereditarietà è ben definita se un oggetto della classe derivata si può utilizzare in tutti i contesti in cui è atteso un oggetto della classe base.
- **Esempio**: il metodo distanza di Punto, riceve un Punto come parametro. Gli si può passare un Disco in quanto un Disco ***is-a*** Punto.
- La parantela esistente tra classe base e classe derivata consente quanto segue:

```
Punto p=new Punto(3,5);
…
Disco d=new Disco(2,7,4);
…
p=d; //assegnazione dal particolare al generale OK
d=p; //NO – un punto non è un disco, non possiede il raggio etc.
```

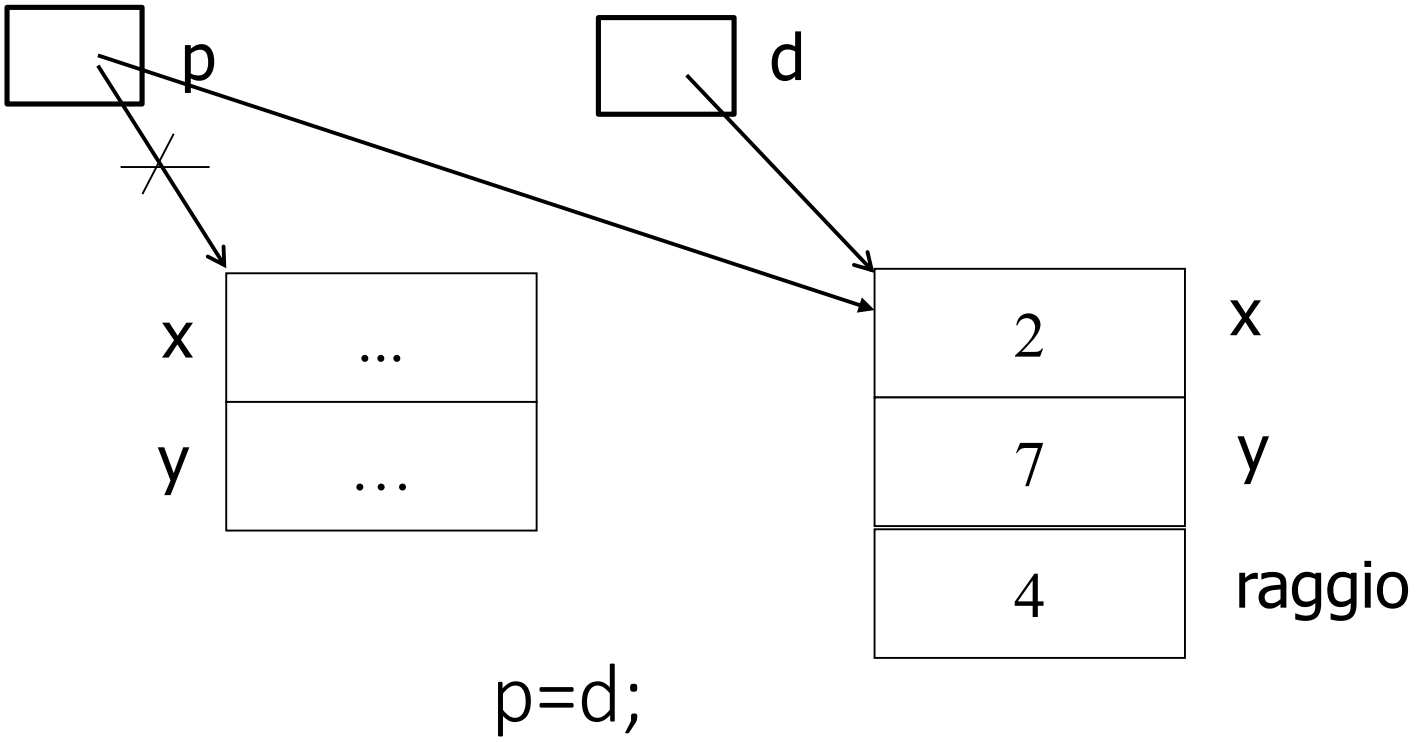
Assegnazione come proiezione geometrica

p=d;



assegnazione OK anche se il raggio non esiste in p

Ciò che succede in Java



p ora punta ad un oggetto Disco che possiede x,y, e raggio. Ma gli "occhiali" di p non consentono di vedere che c'è il raggio

- Dopo l'assegnazione **p=d;** ogni uso di *getX(), toString() etc* si riferisce alla sotto classe Disco.

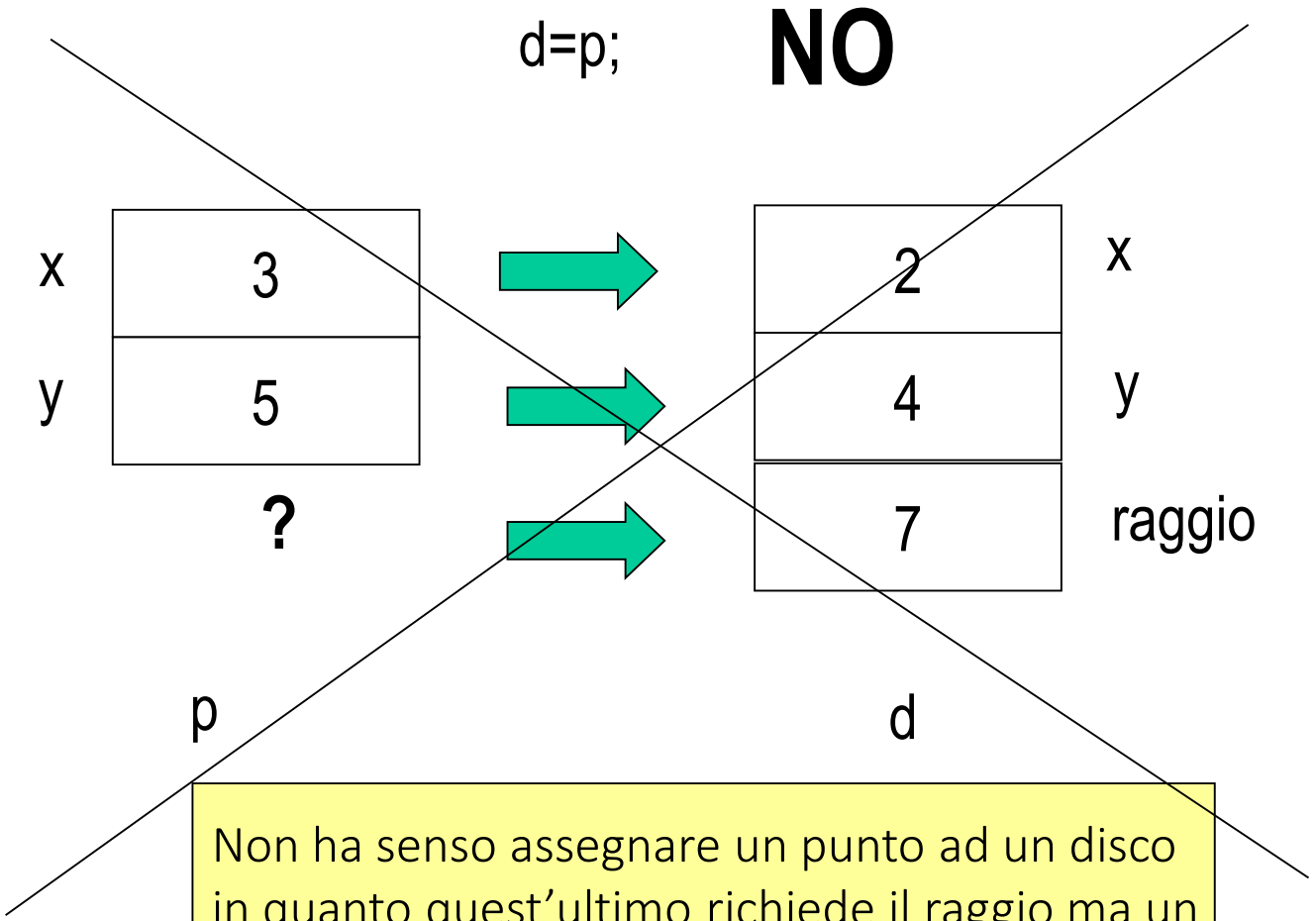
- Si dice che p ha *tipo statico* (legato cioè alla dichiarazione) Punto e *tipo dinamico* (legato cioè all'assegnazione) Disco

- Il **tipo statico** dice *cosa* si può fare su p (quali metodi sono utilizzabili)

- Il **tipo dinamico** dice *quale particolare metodo* va in esecuzione, se uno della super classe o uno della sotto classe.

- **Esempio**: Il toString() è presente sia in Punto che in Disco. Dopo l'assegnazione p=d, ogni uso di p.toString() si riferisce al metodo toString della classe Disco (tipo dinamico di p).

Dal generale al particolare ?



Non ha senso assegnare un punto ad un disco in quanto quest'ultimo richiede il raggio ma un punto ne è sprovvisto

- Tuttavia, se p ha tipo dinamico Disco, si può di fatto cambiare *punto di vista* su p in modo da vederlo come Disco e quindi accedere a tutte le funzionalità di Disco

```
if( p instanceof Disco ){
        d=(Disco)p; //casting
        r=d.getRaggio();
        …
}
```

o direttamente:

```
r=((Disco)p).getRaggio();
```

- Su una variabile p di classe Punto (tipo statico, dovuto alla dichiarazione) possono essere richieste sempre e **solo** le funzionalità della classe cui appartiene

- Ma se p ha tipo dinamico Disco, invocando un metodo ridefinito in Disco (es. toString()), di fatto si esegue la versione di Disco del metodo

- Se p ha tipo dinamico Disco (cosa controllabile con l'operatore **instanceof**) è possibile cambiare il punto di vista su p (casting) in modo da vederlo effettivamente come un oggetto di tipo Disco. Dopo il cambiamento del punto di vista si possono richiedere le funzionalità estese della sottoclasse

Dynamic binding e polimorfismo

- Il **dynamic binding** (collegamento dinamico) si riferisce alla proprietà che invocando un metodo su un oggetto come p, dinamicamente possa essere eseguita la versione di un metodo definito in Punto o quella definita in Disco, in dipendenza del tipo dinamico posseduto da p

- Il termine **polimorfismo** significa "più forme" ed esprime la proprietà che un oggetto possa appartenere a più tipi

- assegnando p=d, l'oggetto p acquisisce un altro tipo (aumenta il polimorfismo). Il polimorfismo di p si può verificare come segue

  if(p instanceof Punto) è TRUE
  if(p instanceof Disco) è TRUE

- A ben riflettere, dynamic binding e polimorfismo sono le due facce di una stessa medaglia

- Proprio perchè sussiste il polimorfismo, si ha l'effetto del dynamic binding.

Ereditarietà e ridefinizione di metodi

- Disco ridefinisce il metod toString già presente nella super classe Punto

- Occorre prestare attenzione che per essere una vera **ridefinizione**, bisogna *rispettare* la sua **intestazione** (*signature*). Se cambia qualcosa nell'intestazione (nome del metodo e tipi dei parametri), allora si tratta di overloading anzichè una ridefinizione

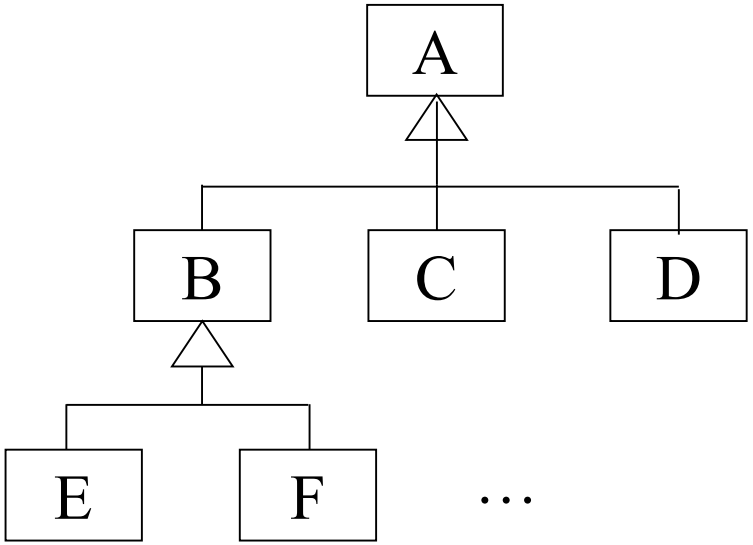
- Perchè funzioni correttamente il dynamic binding/polimorfismo, è necessario osservare l'esatta intestazione dei metodi che ad es.

  ```
  @Override
  public String toString(){…}
  ```

- L'annotazione @Override permette al compilatore di controllare ed eventualmente segnalare problemi, durante una ridefinizione.

Ereditarietà singola

- In Java ogni classe può essere erede di una sola classe (**ereditarietà singola**). Tutto ciò permette la costruzione di gerarchie di classi secondo una struttura ad albero, in cui ogni classe appartiene ad un solo percorso sino alla radice



L'esistenza di una gerarchia di classi, accresce le possibilità di polimorfismo. Ad es. oggetti di classe E hanno come tipi possibili: E, B ed A. Ciò è dovuto alla relazione di ereditarietà. Ad un oggetto di classe A è possibile assegnare un oggetto di una qualsiasi sottoclasse B, C, D, E, F … dunque il tipo dinamico di un oggetto di classe A può essere una qualsiasi delle classi elencate

Ereditarietà vs Composizione

- Occorre sempre riflettere bene se una relazione di ereditarietà sia opportuna o se piuttosto non rappresenti una “forzatura”, alla luce del principio di *sostituibilità dei tipi* (di Liskov)

- Ad es. una Linea (segmento) è caratterizzata da due punti. Progettando la classe Linea come erede della classe Punto, contando sul fatto che un punto proviene dalla superclasse, un altro lo si può aggiungere, si commette un errore grossolano. Infatti, una Linea *non* è un Punto piuttosto è *composta* da due punti. Dunque anzichè ricorrere alla estensione, è opportuno programmare Linea come abbozzato di seguito:

```
class Linea{
        Punto p1, p2; //composizione mediante variabili di istanza
        …
}
```

L’antenato cosmico: Object

- In Java, ogni classe eredita direttamente o indirettamente da **Object** (*radice* di tutte le gerarchie di classi)
- Quando una classe non specifica la clausola extends, in realtà ammette implicitamente la clausola:

  extends Object
- La comune discendenza da Object si manifesta in diverse questioni, es.
  - stile di programmazione
  - polimorfismo
- I seguenti metodi ammettono già un’implementazione in Object che necessariamente è generica. Essi vanno di norma ridefiniti per avere un significato “tagliato su misura” delle nuove classi:
  - **String toString()** – ritorna lo stato di this sotto forma di stringa
  - **boolean equals(Object x)** – ritorna true se this ed x sono uguali. Object definisce l’uguaglianza in modo “superficiale”: due oggetti sono uguali se sono in aliasing, ossia condividono lo stesso riferimento
  - **int hashCode()** – ritorna un hash code (numero intero unico) per this

Un metodo equals per la classe Data

Si mostra una ridefinizione del metodo equals() che basa l'uguaglianza degli oggetti conti bancari sul *contenuto* degli stessi (nozione "profonda" di uguaglianza):

equals deve accordarsi con compareTo quando è presente il confronto

```
@Override
 public boolean equals( Object o ){
      if( !(o instanceof Data) ) return false;
      if( o==this ) return true;
      Data d=(Data)o;
      return this.A==d.A && this.M==d.M && this.G==d.G;
  }//equals
```

Se l'oggetto o passato a equals non avesse tipo dinamico Data, si ritorna false

Se o e this sono in aliasing (coincidenza del riferimento), essi denotano lo stesso oggetto Data e dunque si ritorna true. Altrimenti, si castizza o a Data e si confrontano i tre campi G, M e A

Strutture dati generiche ed eterogenee

- In virtù delle proprietà della relazione di ereditarietà, risulta ad es. che dichiarando

  Object []v=new Object[10];

  si possono memorizzare in v oggetti di QUALUNQUE classe concreta. L'array è **generico** ed **eterogeneo**. L'utente può comunque "scoprire" a runtime il tipo di un elemento con instanceof:

  if(v[i] instanceof String) …

Riassunto modificatori

- Gli attributi di una classe (campi o metodi) possono avere un modificatore tra
  - **public** se sono esportati a tutti i possibili client
  - **private** se rimangono ad uso esclusivo della classe
  - **protected** se sono esportati solo alle classi eredi
  - (*nulla*) se devono essere accessibili all'interno dello stesso package (familiarità o amicizia tra classi).
- Attenzione: gli attributi protected sono accessibili anche nell'ambito del package di appartenenza

- Una classe può essere **public** se esportata per l'uso in altri file
- Non avere il modificatore **public**, se l'uso della classe è ristretto al package (eventualmente anonimo) di appartenenza
- Una classe può essere **final** se non può essere più estesa da classi eredi. Similmente, un metodo **final** non può essere più ridefinito nelle sottoclassi

Un’altro esempio di gerarchia

- Si considera una classe **Contatore** che fornisce l’astrazione di un contatore, ossia una variabile intera che può essere incrementata/decrementata
- La classe dispone di tre costruttori: (1) quello di default che inizializza a zero il contatore; (2) quello normale che imposta il valore iniziale del contatore con il valore di un parametro; (3) quello di copia che imposta il contatore dal valore di un altro contatore

```
package poo.contatori;
public class Contatore{
    protected int valore;
    public Contatore(){ //costruttore di default
        valore=0;
    }
    public Contatore( int valore ){
        this.valore=val;
    }
    public Contatore( Contatore c ){
        this.valore=c.valore;
    }
```

```
    public int getValore(){ return valore; }
    public void incr(){ valore++; }
    public void decr(){ valore--; }
    public String toString(){ return “Contatore ”+valore; }
    public boolean equals( Object o ){
        if( !(o instanceof Contatore ) ) return false;
        if( o==this ) return true;
        Contatore c=(Contatore)o;
        return this.valore==c.valore;
    }//equals
}//Contatore
```

Una classe erede: ContatoreModulare

- Un oggetto **Contatore** può assumere qualsiasi valore intero (positivo/negativo/zero) e può anche traboccare (overflow/underflow)
- Un **ContatoreModulare** si fonda sul concetto di **modulo**, per cui attinto il modulo ritorna da zero e viceversa
- Ad esempio, un contatore decimale (modulo 10), assume tutti i valori tra 0 e 9
- La classe che segue implementa questa nuova specie di contatore che perfeziona il precedente

```
package poo.contatori;
public class ContatoreModulare extends Contatore{
    protected int modulo;
    public ContatoreModulare(){
        super(); //invoca il costruttore di default della super classe
        modulo=10;
    }
    public ContatoreModulare( int modulo ){
        super();
        if( modulo<=1 ) throw new IllegalArgumentException();
    }
    public ContatoreModulare( int val ,int modulo){
        super( val );
        if( val<0 || val>=modulo || modulo<=1)
                throw new IllegalArgumentException();
        this.modulo=modulo;
    }
    public ContatoreModulare( ContatoreModulare cm ){
        super( cm.valore );
        this.modulo=cm.modulo;
    }
```

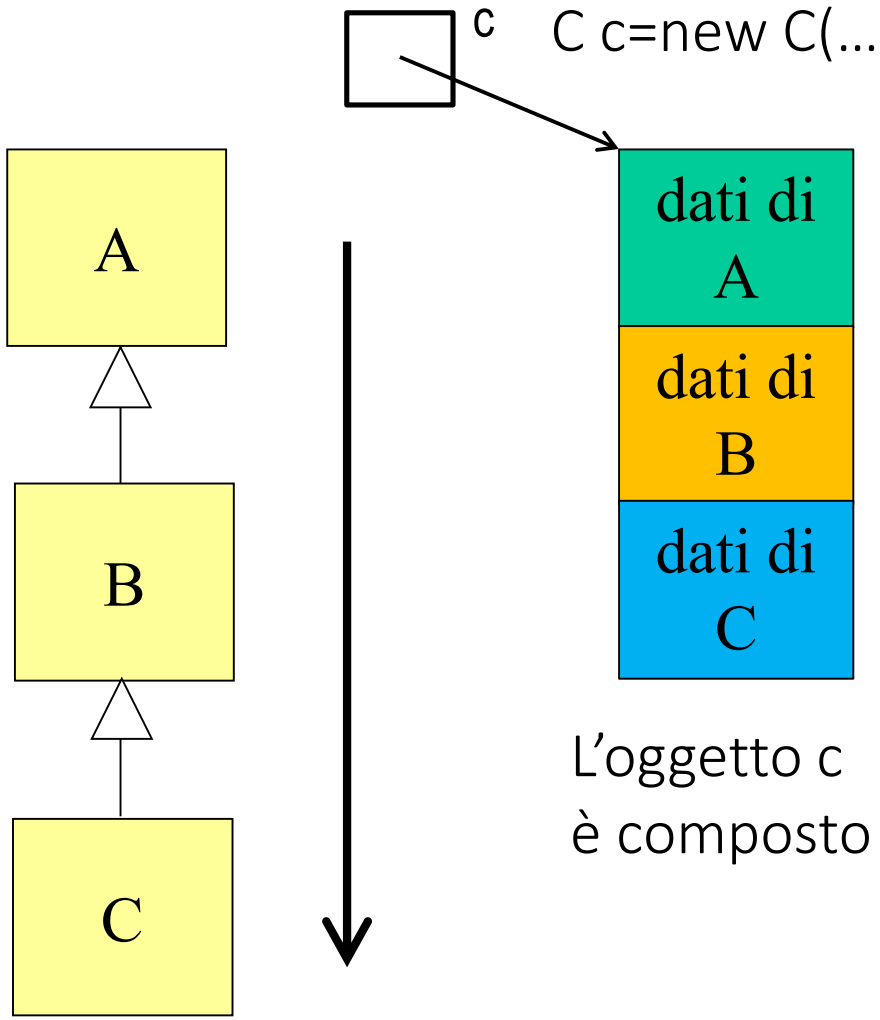
```
    public int getModulo(){ return modulo; }
    @Override
    public void incr(){ valore=(valore+1)%modulo; } //ridefinizione
    public void decr(){ valore=(valore-1+modulo)%modulo; } //rid.
    public String toString(){ //ridefinizione o overriding
        return "Contatore modulo "+modulo+" "+super.toString();
    }//toString
    public boolean equals( Object o ){ //ridefinizione
        if( !(o instanceof ContatoreModulare ) ) return false;
        if( o==this ) return true;
        ContatoreModulare c=(ContatoreModulare)o;
        return c.getValore()==valore && c.modulo==modulo;
    }//equals
}//ContatoreModulare
```

Un main di test

```
public class TContatori{
    public static void main( String []args ){
        Contatore c=new Contatore();
        ContatoreModulare cm=new ContatoreModulare( 8 );
        System.out.println("10 incrementi da "+cm.getValore());
        for( int i=0; i<10; i++ ){
            cm.incr(); System.out.println( cm );
        }
        System.out.println("10 decrementi da "+cm.getValore());
        for( int i=0; i<10; i++ ){
            cm1.decr(); System.out.println( cm );
        }
        c=cm;
        System.out.println(c1); //quale toString parte e perchè?
        for( int i=0; i<5; ++I ) c.incr();
        System.out.println(c); //cosa verrà scritto e perchè?
    }//main
}//TContatori
```

Ordine d'invocazione dei costruttori

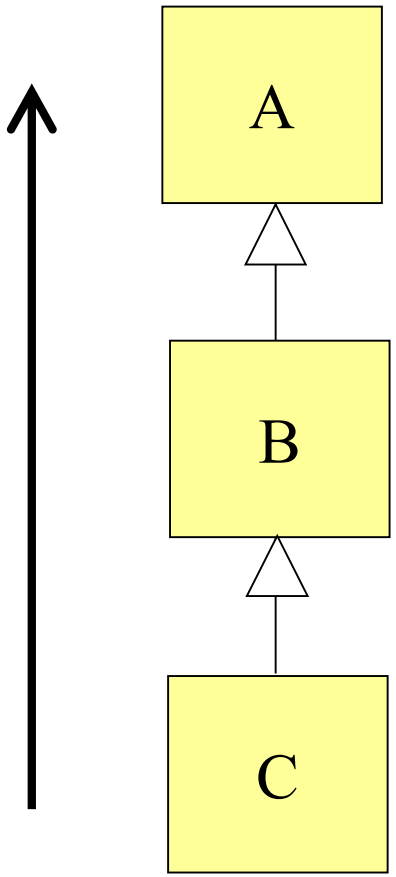
Un oggetto di classe C ingloba *anche* i campi dati che derivano da B *e* da A
L'inizializzazione dei dati dell'oggetto composito ha inizio dalla superclasse A e giù giù sino a C ("dall'alto verso il basso")
Tutto ciò spiega, tra l'altro, che chiamate tipo **super**(…) o **this**(…) che invocano esplicitamente un costruttore devono essere la *prima* azione di un costruttore



Anatomia creazione di un oggetto

1. **new**: Prima si cerca spazio nell'heap per creare l'oggetto (tenendo conto della sua dimensione) e si ritorna il suo indirizzo
2. **tabula rasa**: tutti i campi dell'oggetto sono inizializzati ai loro valori di default
3. **inizializzazione effettiva**: si eseguono i costruttori dalla super classe più in alto, in sequenza, sino al costruttore della sotto classe
4. ***possibilità di eccezioni***: in un costruttore di un "piano alto", si invoca un metodo ridefinito in una sotto classe, i cui dati sono ancora affetti dalla "tabula rasa"…

Ordine d'invocazione di finalize()



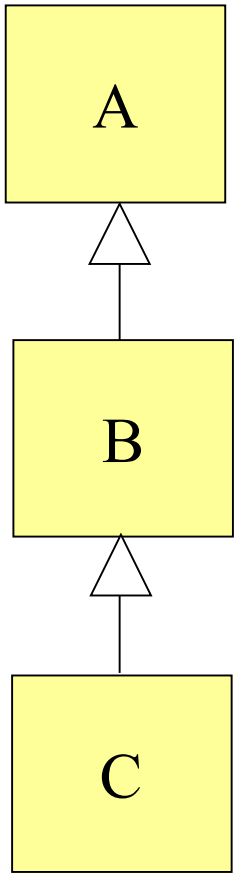
finalize() realizza operazioni di pulizia (es. chiusura di un file, di una connessione di rete, etc.) giusto *prima* che il garbage collector recuperi la memoria dell'oggetto.
Il processo di finalizzazione, ossia l'invocazione dei metodi finalize(), procede dal "basso verso l'alto"

Il metodo getClass()

- In Java anche le classi sono oggetti. La classe di una classe è un oggetto di classe **Class**
- Il metodo **getClass()** di Object ritorna dinamicamente l'oggetto Class associato ad un'istanza. In tale oggetto sono contenute tutte le informazioni relative alla classe dell'istanza. Su di esse si basa l'*introspezione* (reflection), la capacità cioè di poter inferire a runtime quanti e quali costruttori sono disponibili, i campi presenti ed i loro tipi, i metodi con annessi parametri e tipi di ritorno etc.
- Di seguito ci si limita ad osservare che il metodo getClass() può essere utilizzato, in qualche caso, come alternativa di **instanceof**

Il metodo getClass()

Data la gerarchia di classi di figura e le istruzioni




```
A a=new A();
B b=new B();
C c=new C();
…
a=c;
```

il tipo dinamico di **a** può essere scoperto anche attraverso un comando come segue:

if(a.getClass()==C.**class**) … **a** riferisce un oggetto di C

oppure:

if(a.getClass().getName().equals(“C”)) …

Ancora sull'oggetto class

- L'oggetto class di un'istanza è ovviamento unico. Pertanto se ad **a** si assegna **c**, allora **a** viene legato all'oggetto class **C.class** che dentro di sé ingloba (per via dell'ereditarietà) gli attributi di B e di A. Tutto ciò lascia intendere che gli usi di instanceof e di getClass in generale non coincidono. Dopo **a=c;** si ha:

  a instanceof C = true

  a instanceof B = true

  a instanceof A = true

  a.getClass==C.class = true

  a.getClass==B.class = false

  a.getClass==A.class = false