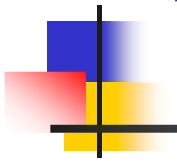
Linguaggi

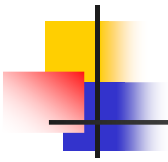
Corso M-Z - Laurea in Ingegneria Informatica
A.A. 2008-2009

Alessandro Longheu
http://www.diit.unict.it/users/alongheu
alessandro.longheu@diit.unict.it



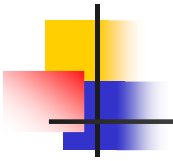
- lezione 07 -

Ereditarietà e Polimorfismo in Java



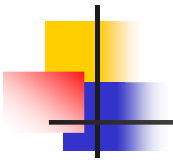
Ereditarietà

- Durante lo sviluppo di codice frequentemente i programmatori sviluppano codice *molto simile* a codice gia esistente
- Questo, spesso, viene fatto manipolando il codice esistente con operazioni di "cut" e "paste"
- Si vuole riusare tutto ciò che può essere riusato (componenti, codice, astrazioni)
- Non è utile né opportuno modificare codice già funzionante e corretto il cui sviluppo ha richiesto tempo (anni-uomo) ed è costato (molto) denaro
- Occorre quindi un modo per catturare le similitudini e formalizzarle, ed un linguaggio che consenta di progettare codice in modo incrementale.



Ereditarietà

- L'ereditarieta' consente di riutilizzare in modo vantaggioso una classe gia' definita che e' simile a quella che vogliamo definire
- Consente di utilizzare il polimorfismo
- La nuova classe e' chiamata "sottoclasse"
- Attraverso l'estensione della classe pre-esistente (chiamata "superclasse"), noi possiamo:
  - aggiungere nuovi dati (attributi) a quelli presenti nella superclasse
  - aggiungere nuovi metodi a quelli presenti nella superclasse, eventualmente con overloading (caratteristica ortogonale all'ereditarietà)
  - ridefinire alcuni metodi della superclasse secondo le nuove esigenze (Overriding dei metodi)

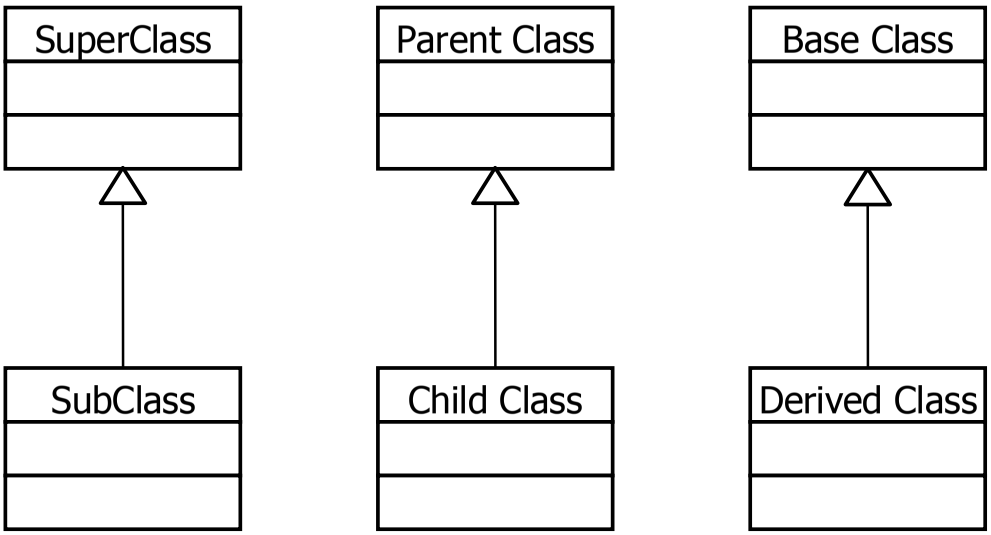


Ereditarietà

- Una relazione tra classi:
  - si dice che la nuova classe B (CLASSE DERIVATA o SOTTOCLASSE) eredita dalla pre-esistente classe A (CLASSE BASE o SUPERCLASSE)
- La nuova classe che ESTENDE un classe già esistente
  - può aggiungere nuovi dati o metodi
  - può accedere ai dati ereditati purché il livello di protezione lo consenta
  - non può eliminare dati o metodi perché il principio di base dei linguaggi OO è che dovunque si usa un oggetto della classe madre deve essere possibile sostituirlo con un oggetto di una qualunque delle classi figlie
- La classe derivata condivide quindi la struttura e il comportamento della classe base

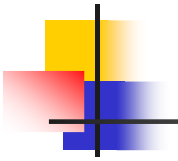
Ereditarietà

- Diverse sono le terminologie utilizzate:



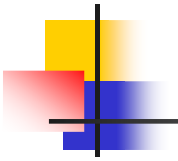
Ereditarietà – Esempio 1

```
public class Point {
    int x;
    int y;                              // attributi
    public Point(int x, int y) {        // Costruttore 2
          setX(x);
          setY(y);
    }
    public Point() {                    // costruttore 1
          x=y=0;                        // sostituibile con this(0,0);
    }
    public void setX(int x) {    this.x = x;      }
    public int getX() {  return x;        }
    public void setY(int y) {    this.y = y;      }
    public int getY() {  return y;        }
} // class Point
```



Ereditarietà – Esempio 1

```
class NewPoint extends Point {

}

class Prova {
   public static void main(String args[]) {
        NewPoint pc = new NewPoint();
        pc.setX(42); pc.x=42;
   NewPoint p2=new NewPoint(0,7);
   p2.x
        System.out.println(pc.getX()+","+pc.getY());
   }
}
```

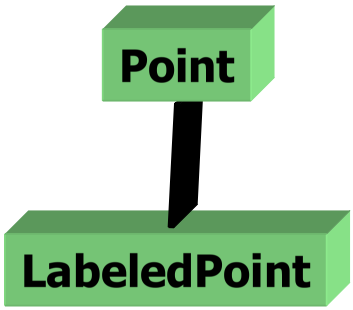



Ereditarietà – Esempio 1

```
public class LabeledPoint extends Point {
   String name;
   public LabeledPoint(int x, int y, String name) {
super (x,y);
        setName(name);
   }
   public void setName(String name) {
        this.name = name;
   }
   public String getName() {
        return name;
   }
}
```

Ereditarietà – Esempio 1

- Noi abbiamo creato una nuova classe con la riscrittura di circa il 50% del codice



parent/superclass

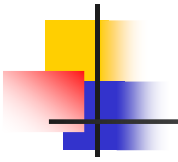
child/subclass

- Cio' e' molto utile

Ereditarietà – Approccio OO

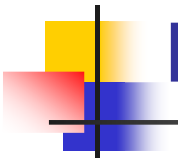
l'approccio per la stesura di codice OO può seguire i passi:

1. **esaminare la realtà**, ed individuare gli elementi essenziali (protagonisti) della stessa
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. occorre poi **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. il passo successivo è la **stesura dei costruttori** delle classi, per decidere qual è lo stato iniziale degli oggetti
6. individuati attributi e metodi (passo 2) insieme al livello di protezione (passo 4), l'interfaccia di ogni classe è definita; è quindi possibile passare **all'implementazione dei metodi**



Ereditarietà

- Cosa si eredita?
  - tutti i dati della classe base
    - anche quelli privati, a cui comunque la classe derivata non potrà accedere direttamente
  - tutti i metodi
    - anche quelli che la classe derivata non potrà usare direttamente
  - tranne i costruttori, perché sono specifici di quella particolare classe.

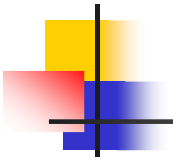


Ereditarietà

Esempio (anomalo) di una classe madre che contiene un oggetto di una classe figlia:

```
public class Figlia extends MadreCheContieneFiglia {   double z;
        public Figlia(double ext) { z=ext;}
        public String toString() { return "valore di z..."+z; }
        }
public class MadreCheContieneFiglia {        int x; Figlia o;
        public MadreCheContieneFiglia() { x=5; o=new Figlia(3.5); }
        public String toString() { return "oggetto : x="+x+" e figlia="+o;}
        public static void main(String args[]) {
                MadreCheContieneFiglia M=new MadreCheContieneFiglia();
                System.out.println ("Ecco un MadreCheContieneFiglia..."+M);
                }       }
```

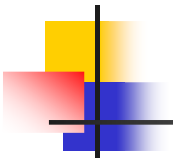
- La compilazione funziona correttamente, ma l'esecuzione provoca:
- Exception in thread "main" java.lang.StackOverflowError
- at Figlia.<init>(Figlia.java:3)
- at MadreCheContieneFiglia.<init>(MadreCheContieneFiglia.java:4)
- at Figlia.<init>(Figlia.java:3)
- at MadreCheContieneFiglia.<init>(MadreCheContieneFiglia.java:4)
- ...



Ereditarietà - Overriding

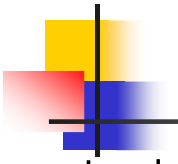
- In caso di overriding, la signature del metodo deve essere la stessa del metodo della superclasse; se differiscono, si tratta di overloading e non di overriding
- un metodo che costituisce l'overriding di un altro, può cambiare il tipo restituito solo se il tipo di ritorno è a sua volta un sottotipo del tipo restituito dal metodo della superclasse, esempio:
  - class madre { …shape metodo1() { … } }
  - class figlia extends madre { … rettangolo metodo1() { … } }
  - dove rettangolo è a sua volta una sottoclasse di shape

  se il tipo è primitivo, deve restare lo stesso, altrimenti si ha un errore nel compilatore



Ereditarietà - Overriding

- 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 protected o public ma non private nel corrispondente omonimo della sottoclasse
- i modificatori synchronized, strictfp e native sono invece liberi ed indipendenti dal metodo della superclasse perché riguardano le implementazioni dei due metodi, le quali sono indipendenti dal legame di overriding
- il metodo ridefinente può essere reso final, se ritenuto opportuno; può essere reso abstract anche se non lo è quello della superclasse

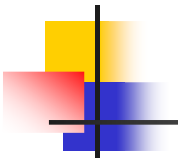


Ereditarietà - Overriding

- La clausula throws può differire, purchè le eccezioni intercettate siano le stesse o sottotipi di quelle intercettate nel metodo della superclasse
- Di un metodo, l'overriding è possibile solo se è accessibile; nel caso non lo sia, e nella sottoclasse venga creato un metodo con la stessa signature di quello non accessibile della superclasse, i due metodi saranno completamente scorrelati
- Un campo non viene ridefinito ma adombrato, nel senso che il nuovo campo rende inaccessibile quello definito nella superclasse, a meno che non si utilizzi *super*

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

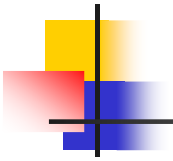
- attributi e metodi statici NON possono essere sovrascritti, ma solo adombrati; questo comunque non ha alcuna rilevanza perché si utilizza comunque il nome della classe di appartenenza per accedervi



Ereditarietà - Overriding

Esempio di overriding:

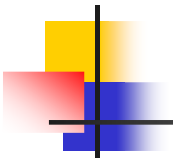
```
class Esempio1 {
        public int metodo1(int x){ return (++x);}
        private int metodo2 (int y) { return (y+3);}
   }
public class Esempio2 extends Esempio1{
   // ESEMPI DI OVERRIDING
        public int metodo1(int z) { return (--z);}
   // IL METODO SEGUENTE DA ERRORE IN COMPILAZIONE
   //(TENTATIVO DI CAMBIARE IL TIPO DI RITORNO)
        public float metodo1(int z) { return (--z);}
        public int metodo2 (int k) { return (k-4); }
```



Ereditarietà - Overriding

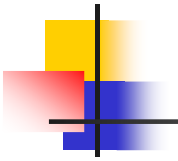
Esempio di overriding:

```
…

    public static void main(String args[]){
          Esempio2 obj=new Esempio2();
// SE NON SI CREA L'OGGETTO NON FUNZIONEREBBE,
//DIREBBE CHE SI TENTA UN ACCESSO STATICO A MEMBRI NON STATICI
    System.out.println("metodo1 locale (override)..."+ obj.metodo1(50));
    //System.out.println("lo stesso con THIS ..."+this.metodo1(50));
    System.out.println("metodo2 superclasse..."+ obj.metodo2(50));


        }
    }
```

Ereditarietà – Accesso membri

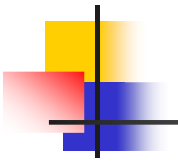
- Nel caso dell'invocazione di un metodo, la classe a cui appartiene l'oggetto anteposto al nome del metodo determina quale metodo viene usato
- Nel caso dell'utilizzo di un campo, il discriminante è il tipo a cui appartiene il riferimento anteposto al nome del campo stesso
- Questa differenza spesso suggerisce di evitare l'adombramento, e piuttosto di privatizzare i campi della superclasse, rendendoli accessibili tramite metodi



Ereditarietà – Accesso membri

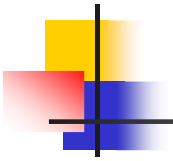
- Esempio:

```
Class Uno {
  public String s="uno";
  public void m() {  System.out.println("uno: "+s);  }}
Class Due extends Uno {
  public String s="due";
  public void m() {  System.out.println("due: "+s);  } }
Class Prova {
  public static void main (String args[]) {
  Due d=new Due();
  Uno u=d; }}
1 u.m();                                        // STAMPA due
2 d.m();                                        // STAMPA due
3 System.out.println("uno: "+u.s);     // STAMPA uno
4 System.out.println("due: "+d.s);     // STAMPA due
5 System.out.println("uno: "+((Due)u).s);// CASTING UTILE: STAMPA due
6 ((Uno)u).m();       // CASTING INUTILE: STAMPA due
```

Ereditarietà - Protezione

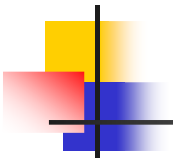
- Problema: il livello di protezione private impedisce a chiunque di accedere al dato, anche a una classe derivata
  - va bene per dati "veramente privati"
  - ma è troppo restrittivo nella maggioranza dei casi
- Per sfruttare appieno l'ereditarietà occorre rilassare un po' il livello di protezione
  - senza dover tornare per questo a public
  - senza dover scegliere per forza la protezione package di default: il concetto di package non c'entra niente con l'ereditarietà
- Si utilizza lo specificatore **protected**, applicabile ad attributi e metodi:
  - è come package (il default) per chiunque non sia una classe derivata
  - ma consente libero accesso a una classe derivata, indipendentemente dal package in cui essa è definita.



Ereditarietà – Esempio Protezione

- Dal contatore (solo in avanti):

```
public class Counter {
   private int val;
   public Counter() { val = 1; }
   public Counter(int v) { val = v; }
   public void reset() { val = 0; }
   public void inc() { val++; }
   public int getValue() { return val;}
}
```

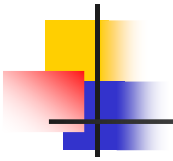
Ereditarietà – Esempio Protezione

- al contatore avanti/indietro (con decremento)

```
public class Counter2 extends Counter {
   public void dec() { val--; }
}
```

Questa nuova classe:

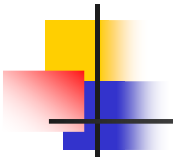
- eredita da Counter il campo val (un int)
- eredita da Counter tutti i metodi
- aggiunge a Counter il metodo dec()
- Ma val era privato, quindi il codice è errato: nessuno può accedere a dati e metodi privati di qualcun altro!
- Una soluzione quindi, nell'ipotesi che il codice della classe madre sia accessibile, è quella di "rilassare" la protezione da privata a protected



Ereditarietà – Esempio Protezione

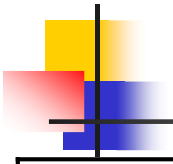
Passando da privato a protetto:

```
public class Counter {
   protected int val;
   public Counter() { val = 1; }
   public Counter(int v) { val = v; }
   public void reset() { val = 0; }
   public void inc() { val++; }
   public int getValue() { return val;}
}
public class Counter2 extends Counter {
   public void dec() { val--; }
}
```

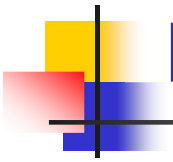
Ereditarietà – Protezione

- La qualifica protected:
  - rende accessibile un campo a tutte le sottoclassi, presenti e future
  - costituisce perciò un permesso di accesso "indiscriminato", valido per ogni possibile sottoclasse che possa in futuro essere definita, senza possibilità di distinzione.
  - I membri protected sono citati nella documentazione prodotta da Javadoc (a differenza dei membri qualificati privati o con visibilità package).



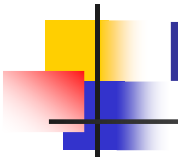
Ereditarietà – Protezione

Accessibile da:	*public*	*protected*	*package*	*private*
Classe di definizione	si	si	si	si
Classi nello stesso package	si	si	si	No
Subclass in differenti package	si	si	No	No
Non-subclass in differenti package	si	No	No	No



Ereditarietà – Protezione

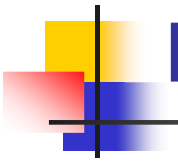
- Rilassare la protezione di un attributo nella classe madre è possibile solo se si ha l'accesso al sorgente
- Se questo non è possibile, e l'attributo è privato, non c'e' modo di usufruirne nella classe figlia in modi diversi da quelli permessi dalla madre
- Ad esempio, se nella classe figlia tentiamo di adombrare l'attributo privato con uno omonimo definito nella figlia, nella speranza che prevalendo l'adombramento per il nuovo attributo possano adattarsi i metodi ereditati dalla madre, il risultato non è permesso, nel senso che l'attributo locale e quello (privato) ereditato, restano separati e i metodi ereditati operano solo su quello ereditato, rendendo necessaria la scrittura di metodi analoghi per intervenire sull'attributo adombrante (locale), operazione che di fatto rende inutile l'avere ereditato da un'altra classe (era meglio "accontentarsi" dell'attributo privato ereditato)



Ereditarietà – Protezione

- Esempio

```
import java.io.*;
class Counter2 extends Counter {
private int val;
    public Counter2() {
                //super();
    }
    public Counter2 (int v) { super(v); }
    public void change() { val++; }
    // #1
// public int getValue() { return val; }
    // #2
    public int getLocalValue() { return val; }
}
...
```

Ereditarietà – Protezione

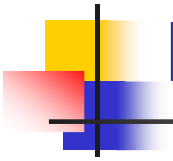
```
public class ProvaCounter2 {
    public static void main (String args[]){
    //  I COSTRUTTORI NON SONO EREDITATI, INFATTI L'ISTRUZIONE
    SEGUENTE E' KO
    // SE NON CREO IO UN COSTRUTTORE IN COUNTER2
    Counter2 c=new Counter2(3);
    // LE DUE STAMPE CHE SEGUONO OPERANO SULLA VAL PRIVATA
    DELLA MADRE
    // A MENO CHE NON DECOMMENTI LA RIGA #1 (VEDI SOPRA)
    System.out.println("counter 2 ..." +c.getValue());
    c.change();
    System.out.println("counter 2 ..." +c.getValue());
```

Ereditarietà – Protezione

```
/*      NEL PRIMO CASO...
D:>java ProvaCounter2
        counter 2 ...3
        counter 2 ...3
        DECOMMENTANDO #1
D:|>java ProvaCounter2
        counter 2 ...0
        counter 2 ...1      */

        // PROVIAMO A USARE I METODI EREDITATI
        c.reset();
        System.out.println("counter 2 ..." +c.getValue());
/*      DECOMMENTANDO #1, SI VEDE CHE reset() NON HA EFFETTO
D:>java ProvaCounter2
        counter 2 ...0
        counter 2 ...1
        counter 2 ...1      */
```

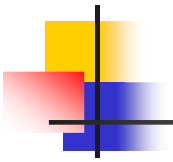

Ereditarietà – Protezione

```
c.inc();
System.out.println("counter 2 ..." +c.getValue());
/*    DECOMMENTANDO #1, SI VEDE CHE reset() NON HA EFFETTO
D:|>java ProvaCounter2
counter 2 ...0
counter 2 ...1
counter 2 ...1
counter 2 ...1
INVECE COMMENTANDO #1...
D:|>java ProvaCounter2
counter 2 ...3
counter 2 ...3
counter 2 ...0
counter 2 ...1          */
```



Ereditarietà – Protezione

```
c.inc();
    System.out.println("Local counter 2 ..." +c.getLocalValue());
    System.out.println("Mother counter 2 ..." +c.getValue());
    /*
    D:|>java ProvaCounter2
    counter 2 ...3
    counter 2 ...3
    counter 2 ...0
    counter 2 ...1
    Local counter 2 ...1
    Mother counter 2 ...2
    */
    }
    }
```

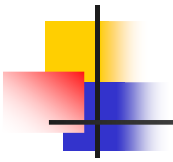



Ereditarietà – Scope

- Ricerca scope attributi:
  - Java prima esamina il metodo corrente, controllando variabili locali e parametri formali
  - In caso negativo, Java esamina la classe corrente
  - successivamente Java esamina la superclasse, continuando eventualmente nella gerarchia delle classi fino a quando non ci sono piu' superclassi da esaminare (generando a questo punto un errore di compilazione).

Ereditarietà – Scope

- Ricerca scope metodi:
  - Java esamina la classe corrente, cercando se esiste un metodo con lo stesso nome ed un numero e tipo compatibile di argomenti; in merito a quest'ultimo punto, Java prima prova a cercare una corrispondenza diretta (senza trasformazioni), quindi prova ad effettuare conversioni boxing (prova a convertire i parametri di tipo primitivo in oggetti delle corrispondenti classi wrapper), ed eventualmente prova ad applicare il numero variabile di argomenti
  - in caso di più metodi possibili presenti nella classe corrente, Java cerca il più specifico (ad esempio quello che accetta String come parametro in vece di quello che accetta Object);
  - In caso negativo, Java esamina la superclasse;
  - Java continua nella gerarchie delle classi fino a quando non ci sono piu' superclassi da esaminare (in tal caso genera l'errore in compilazione)

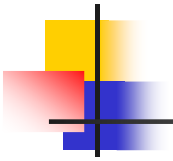


Ereditarietà – this e super

- Java consente di superare le regole di ambiente per attributi e metodi utilizzando:
- la keyword super per specificare metodi e attributi della superclasse, ad esempio super.metodo(par…) o super.attributo
- la keyword this per specificare metodi e attributi dell'oggetto corrente

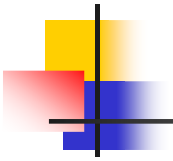
```
super(xxx)     // chiama il costruttore della superclasse
super.xxx      // accede agli attributi della superclasse
super.xxx( )   // chiama i metodi della superclasse
this(xxx)      // chiama il costruttore della classe corrente
this.xxx       // accede agli attributi della classe corrente
this.xxx( )    // chiama i metodi della classe corrente
```

non si puo' invocare un costruttore di una classe dalle nipoti con super.super<something>



Ereditarietà – this e super

```
public class Student {
    public String name = "";
    public String gtNumber = "";
    public int transcriptCount  = 0;
   public void identifySelf( ) {
      System.out.println("My name is " + name);
   }  // identifySelf
   public void requestTranscript( ) {
      sendTranscript( );
      transcriptCount++;
   } // requestTranscript
   public void sendTranscript( ) {
      System.out.println("Sending transcript");
   } // sendTranscript
} // Student
```

Ereditarietà – this e super

- Per il momento non ci occupiamo di "private" e "public"
- students ha un name, gtNumber, e un contatore di richieste. Esso può identificare se stesso e fare una richiesta.
- Esempio di utilizzo:

```
Student eddie = new Student();
eddie.setName("Eddie");
eddie.requestTranscript();
```

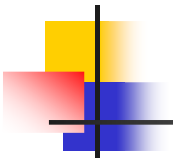
- Output:

"Sending transcript"

Ereditarietà – this e super

Ora si definisce una classe figlia di Student:

```
public class GradStudent extends Student  {
   int cost;
   public GradStudent() {
      this.cost = 2; // oppure this.setCost(2);
   } // constructor
   public int getCost(){ return cost; }
   public void setCost(int cost) { this.cost = cost; }
   public void sendTranscript( )  {
      this.identifySelf();
      System.out.println("I am a graduate student.");
      System.out.println("Transcript Cost: " + getCost());
      setCost(getCost()+1);
   }  // sendTranscript
} // GradStudent
```

Ereditarietà – this e super

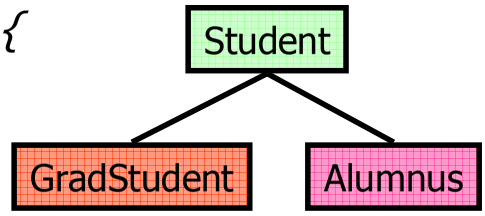
- Riscriviamo il metodo sendTranscript (overriding)
- Esempio di utilizzo:

```
GradStudent ray = new GradStudent();
ray.setName("Raymond");
ray.requestTranscript();
```

Output:

"My name is Raymond"
"I am a graduate student."
"Transcript Cost: $2"

Ereditarietà – this e super




```
public class Alumnus extends Student  {
   int year;
   public Alumnus(int year) {
      setYear(year);
   } // constructor
   public void setYear(year){ this.year = year; }

   public void sendTranscript( )  {
      this.identifySelf();
      System.out.println("I am an alumnus.");
      System.out.println("Sending transcript");
   }  // sendTranscript

} // Alumnus
```


Ereditarietà – this e super

- Riscriviamo il metodo sendTranscript (overriding)
- Esempio di utilizzo:

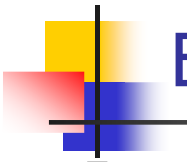
```
Alumnus rick = new Alumnus(2005);
rick.setName("Rickie");
rick.requestTranscript();
```

Output:

"My name is Rickie"
"I am an alumnus."
"Sending transcript"

Ereditarietà – this e super

```
public class Alumnus extends Student  {
   int year;
   public Alumnus(int year) {
      setYear(year);
   } // constructor
   public int setYear(year){ this.year = year; }
   public void sendTranscript( )  {
      this.identifySelf();
      System.out.println("I am an alumnus.");
          // ora deve fare System.out.println("Sending transcript");
          // il codice quindi è lo stesso del
          // corrispondente metodo della superclasse, quindi …
      super.sendTranscript();
          // senza super, si provocherebbe la ricorsione
   }  // sendTranscript
} // Alumnus
```

Ereditarietà – this e super

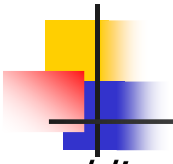
- Esempio di utilizzo:

```
Alumnus rick = new Alumnus(2006);
rick.setName("Rickie");
rick.requestTranscript();
```

Output:

"My name is Rickie"
"I am an alumnus."
"Sending transcript"

La chiamata di requestTranscript porta nella superclasse, ma l'esecuzione di requestTranscript al suo interno invoca sendTranscript, che senza this o super pre-posto porta all'esecuzione del sendTranscript della classe Alumnus



Ereditarietà – this e super

```
public class Student {        public String name; ...}
public class GradStudent extends Student {
   String name; // adombramento del campo della superclasse
   public GradStudent(String name) {this.name = name;
}
void twoNames() {
        System.out.println("Student: " + name);
        System.out.println("Grad Student: " + super.name);
   }
public static void main (String [] args){
     GradStudent fred = new GradStudent("Fred");
     fred.twoNames();
  }
}
```

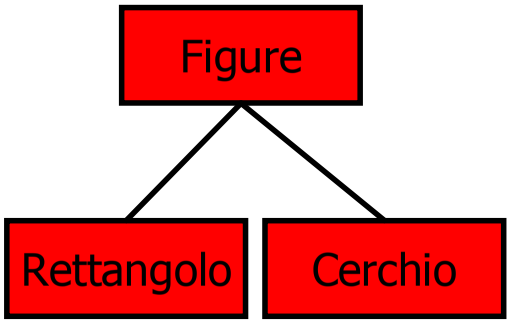
Output:

- **Student: fred**
- **Grad Student: null**

Ereditarietà – this e super

```
public class Nonno {
   public void method1() {...}
}
public class Padre extends Nonno {
   public void method1() {...}
   public void method2() {...}
}
public class Figlio extends Padre {
   public void method1() {...}
   public void method3() {
        method1();
        method2();
        super.method1(); // chiama il Padre
   // non è possibile chiamare il Nonno con super.super.method1
  }
}
```

Ereditarietà – this e super

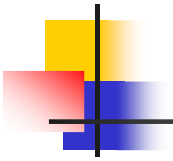



```
public class Figure {
   public String name;

   public String getName ()  {
        return (this.name);
   }  // getName

   public int area () {
        return (0);
    // questo metodo poteva essere
    // astratto
   }  // area

}
```

Ogni classe derivata implementera' il metodo area.

Ereditarietà – this e super

```
public class Rettangolo extends Figure {
 private int base, altezza;
Rettangolo() { this(0, 0); } // costruttore
Rettangolo(int base, int altezza)
        {this(base, altezza, "rettangolo"); }
Rettangolo(int base, int altezza, String name) {
   // chiamata implicita a super()
this(base,altezza);
    this.base = base;
    this.altezza = altezza;
    this.name = name;
 } // costruttore
public int area() { return (base * altezza);  } // area
...
```



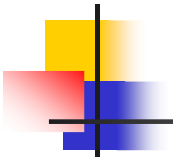
Ereditarietà – this e super

```
...
public String getName () {
  if (base == altezza) return "quadrato: " + super.getName();
  else   return super.getName());
 }  // getName
public String toString () {
    String answer;
    answer = new String("Il rettangolo chiamato " +
      getName() + " con altezza " + altezza +
          " e base " + base);
    return (answer);
 } // toString

} // Rettangolo
```


Ereditarietà – Costruttori

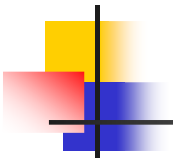
Regole dei costruttori di sottoclassi:

- La prima istruzione del costruttore di una sottoclasse può essere:
  - una chiamata esplicita ad un costruttore della superclasse
  - una chiamata esplicita ad un (altro) costruttore della classe corrente
  - una generica istruzione; in tal caso, Java implicitamente aggiungerà super() prima dell'esecuzione della prima istruzione del costruttore
- Quando si crea un oggetto tramite un new <costruttore>(…):
  - 1) viene allocata la memoria necessaria
  - 2) le variabili sono inizializzate ai valori di default (0, null…)
  - 3) viene invocato un costruttore della superclasse
  - 4) vengono inizializzati i campi mediante inizializzatori (nella dichiarazione) e/o tramite blocchi di inizializzazione
  - 5) vengono eseguite le istruzioni del costruttore
  - i passi 3,4,5 sono applicati ricorsivamente



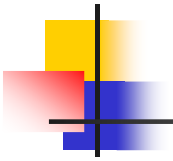
Ereditarietà – Costruttori

- Una classe derivata non può prescindere dalla classe base, perché ogni istanza della classe derivata comprende in sé, indirettamente, un oggetto della classe base.
- Quindi, ogni costruttore della classe derivata deve invocare un costruttore della classe base affinché esso costruisca la "parte di oggetto" relativa alla classe base stessa:
  - "ognuno deve costruire ciò che gli compete"



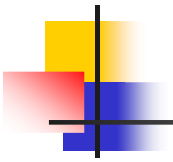
Ereditarietà – Costruttori

- Perché bisogna che ogni costruttore della classe derivata invochi un costruttore della classe base?
  - solo il costruttore della classe base può sapere come inizializzare i dati ereditati in modo corretto
  - solo il costruttore della classe base può garantire l'inizializzazione dei dati privati, a cui la classe derivata non potrebbe accedere direttamente
  - è inutile duplicare nella sottoclasse tutto il codice necessario per inizializzare i dati ereditati, che è già stato scritto.



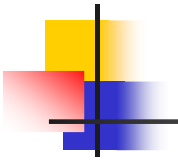
Ereditarietà – Costruttori

- In assenza di altri costruttori, il sistema genera automaticamente un costruttore di default, senza parametri
- Se però è definito anche solo un costruttore, il sistema assume che noi sappiamo il fatto nostro, e non genera più il costruttore di default automatico; questo potrebbe provocare errori se nelle sottoclassi sono presenti costruttori che cercano di invocare super(); in tal caso, è bene che esista anche il costruttore di default senza parametri. In sostanza, se è stato scritto un costruttore, è bene anche scrivere quello ()



Ereditarietà – Costruttori

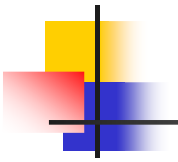
- Di norma, i costruttori sono public
  - in particolare, è sempre pubblico il costruttore di default generato automaticamente da Java
  - Almeno un costruttore pubblico deve sempre esistere, a meno che si voglia impedire espressamente di creare oggetti di tale classe agli utenti "non autorizzati"
  - caso tipico: una classe che fornisce solo costruttori protetti è pensata per fungere da classe base per altre classi più specifiche
  - non si vuole che ne vengano create istanze.



Ereditarietà – Costruttori

- Esempio di costruttore privato ereditato ma non accessibile

```
public class SenzaPar {
    int interna;
    public SenzaPar(int x) { interna=x;}
    private SenzaPar() {}
    }
public class UsaSenzaPar extends SenzaPar {
      public UsaSenzaPar() {          System.out.println("prova");      }
      public static void main (String args[]) {
                UsaSenzaPar obj=new UsaSenzaPar();
       }
      /*
D:|>javac SenzaPar.java
D:|>javac UsaSenzaPar.java
UsaSenzaPar.java:2: SenzaPar() has private access in SenzaPar
            public UsaSenzaPar() {
   1 error    */}
```

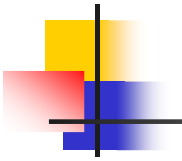



Ereditarietà – Classi final

- Una classe finale (final) è una classe di cui si vuole impedire a priori che possano essere definite, un domani, delle sottoclassi
- Esempio:

```
public final class TheLastCounter
extends Counter {
            ...
            }
```

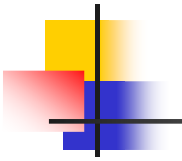
- Un'alternativa che lascia maggiore flessibilità potrebbe essere quella di rendere final tutti i metodi della classe, in modo da preservarne le funzionalità senza pregiudicare l'ereditabilità dalla classe; in tal caso, è bene che tali metodi operino su variabili finali o private, in modo da impedire cambiamenti illeciti indiretti



Ereditarietà – Classi final

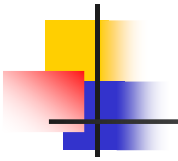
- Qualora fosse necessario creare una classe simile ad una classe finale, e non fosse quindi possibile sfruttare l'ereditarietà, una possibile soluzione potrebbe essere la seguente:

```
public final class Last {
    float f;
    ...
    }
public class LastLast {
    Last L=new Last(...);
    String campo1;
    int x;
    public void LastLast(float fext, String campo1ext, int xext) {
      L.f=fext;
      campo1=campo1ext;
      x=xext;
      }       }
```

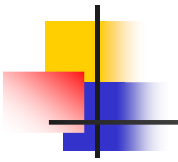
Ereditarietà – Classi final

- Sostanzialmente, la classe LastLast vorrebbe estendere la Last, aggiungendo i propri campi campo1 e x, ma non può perché il progettista della Last l'ha resa finale. Allora chi scrive la LastLast fa sì che essa includa un oggetto di tipo Last (L), al fine di utilizzarne i campi ed i metodi.
- L'inclusione di un oggetto di tipo Last non rende La classe LastLast un'estensione di Last, perche' la relazione fra le due classi non è di ereditarietà (indicata anche con "IsA", nel senso che classefiglia "IsA" classeMadre), ma piuttosto di contenimento (spesso indicata con "HasA", nel senso che LastLast "HasA" Last)



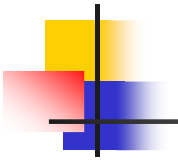
Ereditarietà – Classi final

- La classe LastLast al suo interno deve fare un ulteriore passaggio per accedere agli attributi di L (ad esempio f), utilizzando la notazione puntata L.f, laddove se f fosse stato un attributo ereditato, il semplice nome f sarebbe stato sufficiente ed avrebbe posto f sullo stesso piano di x e campo1; purtuttavia, se al mondo esterno questa cosa viene mascherata (come l'esempio del costruttore qui sopra mostra), si può dare l'illusione che gli oggetti di tipo LastLast siano "quasi" di tipo Last
- Questo modo di procedere è sintatticamente corretto, però in genere è meglio usare l'ereditarietà tutte le volte che questo è possibile, anche per fruire dei vantaggi che porta con sé, ad esempio il polimorfismo (secondo il codice scritto qui sopra un oggetto di classe LastLast NON potrebbe prendere il posto di uno di classe Last, richiedendo di riscrivere il relativo codice) e più in generale non consente il riuso del software



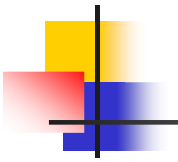
Ereditarietà – Classe Object

- Tutte le classi estendono implicitamente la classe Object
- Non tutti i linguaggi OO prevedono che una classe debba necessariamente ereditare da qualcun'altra
- La classe Object prevede dei metodi standard utilizzabili in tutte le altre, anche se spesso vanno riscritti:
  - *public boolean equals (Object obj)*, che come implementazione standard controlla se this==obj; il metodo va riscritto se la verifica dell'uguaglianza deve essere basata su criteri differenti (non su "==")
  - *public String toString()*, che nella versione base restituisce una stringa contenente il nome della classe dell'oggetto, la @, ed il codice hash esadecimale dell'istanza su cui è invocato, ovvero torna *getClass().getName() + '@' +Integer.toHexString(hashCode())*
  - *protected Object clone()*, che effettua la clonazione dell'oggetto



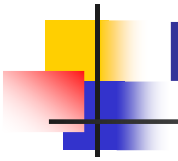
Ereditarietà – Clonazione

- La clonazione di un oggetto ne restituisce un altro, indipendente dal primo, il cui stato iniziale è una copia dello stato corrente dell'oggetto su cui il metodo è stato invocato
- una classe può prevedere quattro opzioni nei confronti della clonabilità:
  - supportarla;
  - non supportarla;
  - la supporta condizionatamente: la classe può clonare se stessa, ma non si richiede che tutte le sottoclassi abbiano la stessa capacità
  - non la supporta direttamente, ma lo permette nelle sottoclassi



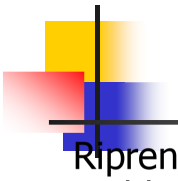
Ereditarietà – Clonazione

- la clonazione di default non sempre è corretta, ad esempio se un oggetto contiene un array, il clone farebbe riferimento allo stesso array, il che probabilmente non è quello che si vuole
- l'implementazione di default di clone è detta clonazione superficiale, che semplicemente copia campo per campo;
- la clonazione profonda, da implementare manualmente, opera invece ricorsivamente anche su eventuali array e/o oggetti contenuti nell'oggetto da clonare



Estensione di classi

- L'estensione della classe tramite ereditarietà è anche detta relazione "IsA" perché se la classe B estende la A, si dice che B "IsA" A
- Esiste anche la relazione "HasA", in cui invece un oggetto contiene riferimenti ad altri oggetti (contenimento); le due relazioni spesso non sono di facile distinzione
- Una classe dovrebbe sempre essere dotata di due interfacce, una pubblica rivolta ai programmatori che intendono *usare* la classe (HasA), ed una protetta rivolta ai programmatori che intendono *estendere* la classe (IsA)



Casting

Riprendendo l'esempio dei contatori:

public class Counter { … }

public class Counter2 extends Counter {public void dec() { val--; } }

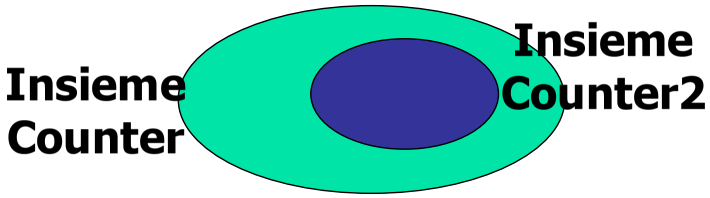
- Ogni oggetto di classe Counter2 è anche implicitamente di classe Counter
- Ma non viceversa; un Counter è meno ricco di un Counter2
- Counter2 può quindi essere usato al posto di un Counter se necessario
- Ogni Counter2 è anche un Counter

Esempio:

```
public class Esempio6 {
public static void main(String args[]) {
        Counter c1 = new Counter(10);
        Counter2 c2 = new Counter2(20);
        c2.dec();                   // OK: c2 è un Counter2
        // c1.dec();                // NO: c1 è solo un Counter
        c1=c2;          // OK: c2 è anche un Counter
        // c2=c1;                   // NO: c1 è solo un Counter
        } }
```

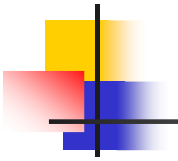
Casting

- Dunque, la classe Counter2 definisce un sottotipo della classe Counter
  - Gli oggetti di classe Counter sono compatibili con gli oggetti di classe Counter2 (perché la classe Counter2 è inclusa nella classe Counter) ma non viceversa
  - Ovunque si possa usare un Counter, si può usare un Counter2 (ma non viceversa)



Casting

- c1=c2 è possibile, comporta un casting implicito di tipo widening (perché c1 è più ampia di c2). La conversione widening è anche detta upcast.
- c2=c1 invece non è possibile a meno di non ricorrere ad un casting esplicito, quindi c2=(Counter2)c1, in cui c1 dovrebbe essere ristretto (conversione narrowing) per diventare come c2. La conversione narrowing è anche detta downcast, e potrebbe determinare errori in fase di compilazione
- in caso di dubbio è sempre possibile utilizzare l'operatore *instanceOf* per conoscere il tipo di un dato oggetto.



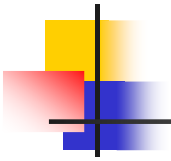
Casting

- Dire che ogni Counter2 è anche un Counter significa dire che l'insieme dei Counter2 è un sottoinsieme dell'insieme dei Counter:
  - Se questo è vero nella realtà, la classificazione è aderente alla realtà del mondo; Se invece è falso, questa classificazione nega la realtà del mondo, e può produrre assurdità e inconsistenze
  - Esempi:
  - Studente che deriva da Persona → OK (ogni Studente è anche una Persona)
  - Reale che deriva da Intero → NO (non è vero che ogni Reale sia anche un Intero)

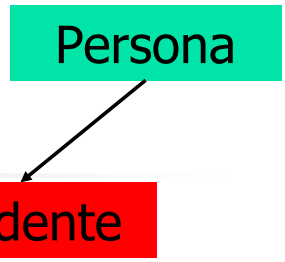
Polimorfismo

Viene anzitutto creata una classe persona ed una studente:

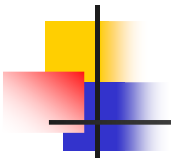
```
public class Persona {
   protected String nome;
   protected int anni;
   public Persona()
         {nome = "SCONOSCIUTO"; anni = 0; }
   public Persona(String n) {nome = n; anni = 0; }
   public Persona(String n, int a)  {nome=n; anni=a; }
   public void print() {
      System.out.print("Mi chiamo " + nome);
      System.out.println(" e ho " +anni+ "anni");
       }
}
```



Polimorfismo




```
public class Studente extends Persona {
    protected int matr;
    public Studente() {super(); matr = 9999; }
    public Studente(String n) {
          super(n); matr = 8888; }
    public Studente(String n, int a) {
          super(n,a); matr=7777; }
    public Studente(String n, int a, int m) {
          super(n,a); matr=m; }
    public void print() {
          super.print();
          System.out.println("Matricola = " + matr);
          }
}
```

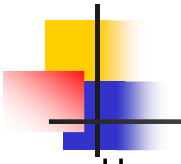



Polimorfismo

```
public class EsempioDiCitta {
   public static void main(String args[]){
   Persona p = new Persona("John");
   Studente s = new Studente("Tom");
   p.print(); // stampa nome ed età
   s.print(); // stampa nome, età, matricola
   p=s; // OK, Studente estende Persona, casting implicito widening
   p.print(); // COSA STAMPA ???
    }
}
```

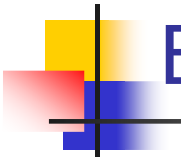
p è un riferimento a Persona ma gli è stato assegnato uno Studente

- Se prevale la natura del riferimento, stamperà solo nome ed età
- Se prevale invece la natura dell'oggetto puntato, stamperà nome, età e matricola
- È un problema di POLIMORFISMO



Polimorfismo

- Un metodo si dice polimorfo quando è in grado di adattare il suo comportamento allo specifico oggetto su cui deve operare.
- In Java, la possibilità di usare riferimenti a una classe, ad esempio Persona, per puntare a oggetti di classi più specifiche (ad esempio, Studente), introduce in astratto la possibilità di avere polimorfismo:
  - se prevale il tipo del riferimento, non ci sarà mai polimorfismo e in tal caso, p.print() stamperà solo nome ed età, perché verrà invocato il metodo print() della classe Persona
  - se invece prevale il tipo dell'oggetto, ci potrà essere polimorfismo e in tal caso, p.print() stamperà nome, età e matricola, perché verrà invocato il metodo print() della classe Studente
- Java supporta il Polimorfismo e quindi prevale il tipo dell'oggetto
- l'accesso ai metodi, come visto nella relativa sezione, è quindi regolato dal polimorfismo, mentre l'accesso agli attributi è regolato dall'adombramento; è possibile comunque intervenire in qualche misura utilizzando il casting esplicito, ossia visto che prevale il tipo dell'oggetto, si può cercare di forzarlo a proprio piacimento



Polimorfismo
Binding statico e dinamico

- Binding statico
  - le chiamate ai metodi sono collegate alla versione del metodo prestabilita a tempo di compilazione, basandosi sul tipo statico del riferimento. E' efficiente, ma non flessibile
  - standard in C, default in C++, assente in Java
- Binding dinamico
  - le chiamate ai metodi sono collegate alla versione del metodo determinata a run-time, basandosi sul tipo dinamico dell'oggetto referenziato in quel momento. Un po' meno efficiente, ma molto flessibile
  - non presente in C, possibile a richiesta in C++ (virtual), default in Java

Polimorfismo
Binding statico e dinamico

- Ogni istanza contiene un riferimento alla propria classe e include una tabella che mette in corrispondenza i nomi dei metodi da essa definiti con il codice compilato relativo a ogni metodo
- Chiamare un metodo comporta quindi:
  - accedere alla tabella opportuna in base alla classe dell'istanza
  - in base alla signature del metodo invocato, accedere alla entry della tabella corrispondente e ricavare il riferimento al codice del metodo
  - invocare il corpo del metodo così identificato.

Polimorfismo
Binding statico e dinamico

```
class Animali {
   public void verso () { System.out.println("Io sono un Animale.");
    }
} // end classe Animali
class Pesce extends Animali {
   public void verso() {System.out.println("Glug glug gurgle
    gurgle");}
} // end classe Pesce
class Uccello extends Animali {
   public void verso() { System.out.println("Tweet tweet flap
    flap"); }
} // end classe uccello
class Cane extends Animali {
   public void verso() { System.out.println("Sniff sniff woof woof");
    }
   public void ringhiare() { System.out.println("Arf Arf");  }
} // end classe Cane
```

Polimorfismo
Binding statico e dinamico

```
public class Zoo  {
   public static void main (String[ ] argv) {
      Animali[ ] AnimaliArray = new Animali[3];
      int index;
      AnimaliArray[0] = new Uccello( );
      AnimaliArray[1] = new Cane( );
      AnimaliArray[2] = new Pesce( );
    for (index = 0; index < AnimaliArray.length; index++)
    { AnimaliArray[index].verso( );}
   } // end del main
} // end classe prova
```

- La classe Animali ha *verso()* cosi ogni membro della classe puo' fare un verso
- Output
  Tweet tweet flap flap
  Sniff sniff woof woof
  Glug glug gurgle gurgle

Polimorfismo
Binding statico e dinamico

- Polimorfismo significa "prendere molte forme" … un riferimento a una data classe puo' prendere la forma di ognuna delle sue sottoclassi.
- Polimorfismo e Legame Dinamico (Dynamic Binding) insieme asssicurano il corretto funzionamento del metodo verso() dell'esempio precedente.
- Un oggetto di una sottoclasse puo sostituire un oggetto della superclasse: "Un uccello e' un Animale"
- Il contrario non e' vero: non si puo sostituire un elemento di una sottoclasse con uno della superclass "Un Animale non e' un uccello".
- Abbiamo una singola interfaccia per un comportamento multiplo:
  - Solo una interfaccia per la chiamata del metodo.
  - Comportamento multiplo basato sulla sottoclasse

Polimorfismo
Binding statico e dinamico

```
public class Esempio2 {
   public static void main(String[ ] argv)  {
      Animali AnimaliArray [ ] = new Animali[3];
      Cane c;
      int i;
      AnimaliArray[0] = new Uccello( );
      AnimaliArray[1] = new Cane( );
      AnimaliArray[2] = new Pesce( );
      for (i = 0; i < AnimaliArray.length; i++){
        AnimaliArray[i].verso();
        if (AnimaliArray[i] instanceof Cane){
          c = (Cane) AnimaliArray[i];
          c.ringhiare( );
        }
      }
   } //  main
} //  Esempio2
```


Polimorfismo
Binding statico e dinamico

- Il Casting è usato qui per dare ad un oggetto di una sottoclasse la forma della sottoclasse appropriata per consentire la chiamata del metodo; infatti:

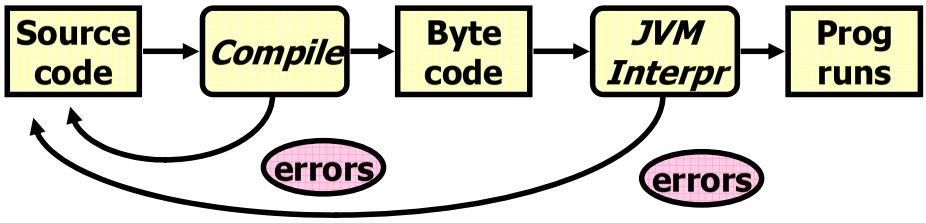
```
if (AnimaliArray[i] instanceof Cane) {
   AnimaliArray[i].ringhiare();
}
```

- produce un errore perchè un oggetto della classe Animali non ha il metodo ringhiare(). Cosi, noi prima eseguiamo il cast dell'oggetto

```
if (AnimaliArray[i] instanceof Cane) {
   c = (Cane) AnimaliArray[i]
   c.ringhiare( );
}
```

- ma se Java puo determinare cos'è (o non è) una dato oggetto attraverso l'uso di instanceof, perchè e' necessario il cast? Perche Java non puo' fare questo per noi?

Polimorfismo
Binding statico e dinamico



- Errori a Compile-time:
- Quelli che sono rilevabili senza che il programma sia un esecuzione.

```
int index
string strName="x"
index = strName;
```

- Istruzione certamente illegale

- Errori a Run-time:
- Quelli che sono riconoscibili solo durante l'esecuzione con I valori reali.

```
AnimaliArray[i] = UnAnimale
```

- L'istruzione è corretta ma può non essere corretta per alcuni valori di indice

Polimorfismo
Binding statico e dinamico

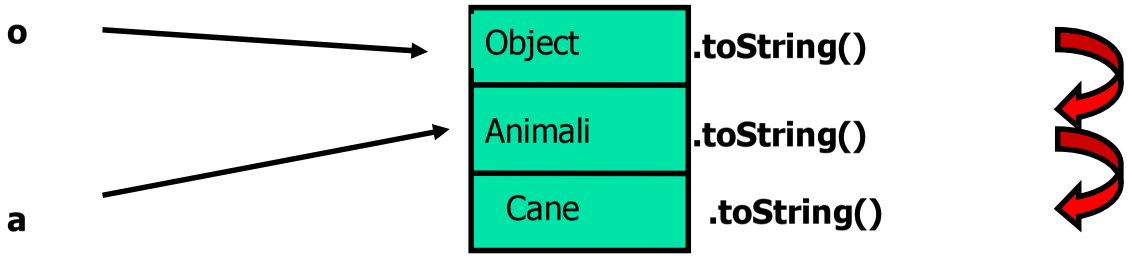
```
if (AnimaliArray[i] instanceof Cane){
    AnimaliArray[i].ringhiare();
}
```

- La prima riga è corretta, la seconda no a meno che l'array non sia dicharato di elementi Cane
- Il compilatore non puo' vedere il legame fra le istruzioni, il runtime system potrebbe ma per garantire le performance non sono effettuate queste verifiche durante la compilazione, lasciando al programmatore l'incombenza del controllo necessario per non provocare l'errore:

```
if (AnimaliArray[i] instanceof Cane) {
    c = (Cane) AnimaliArray[i];
    c.ringhiare( ); }
```

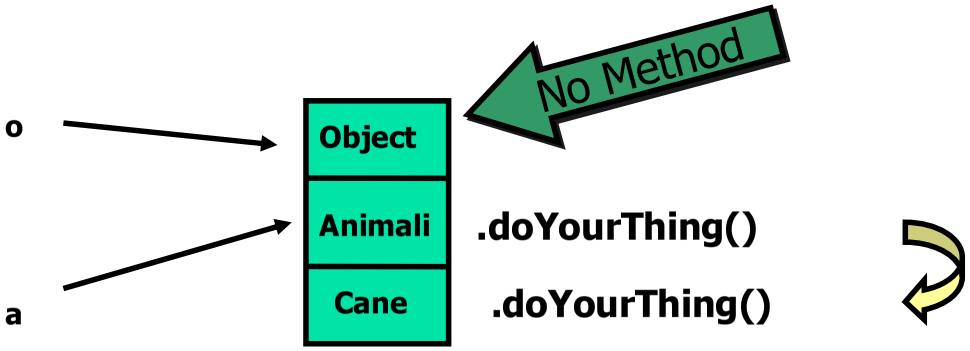
- casting, polimorfismo e binding dinamico sono quindi legati

Polimorfismo
Binding statico e dinamico



- Quando chiamiamo un metodo su un riferimento, il metodo deve esistere (o essere ereditato) nel tipo.
- Comunque, la specifica implementazione e determinata a run-time. Cioe' viene utilizzato il 'dynamic binding'.

Polimorfismo
Binding statico e dinamico



- Il dynamic binding non fa miracoli. Il tipo deve avere il metodo disponibile (nella classe corrente o nella sua superclasse) altrimenti da un errore di compilazione:
- *o.doYourThing()* non esiste