Prof. A. Longheu

Ereditarietà nei linguaggi OO

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
- 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)

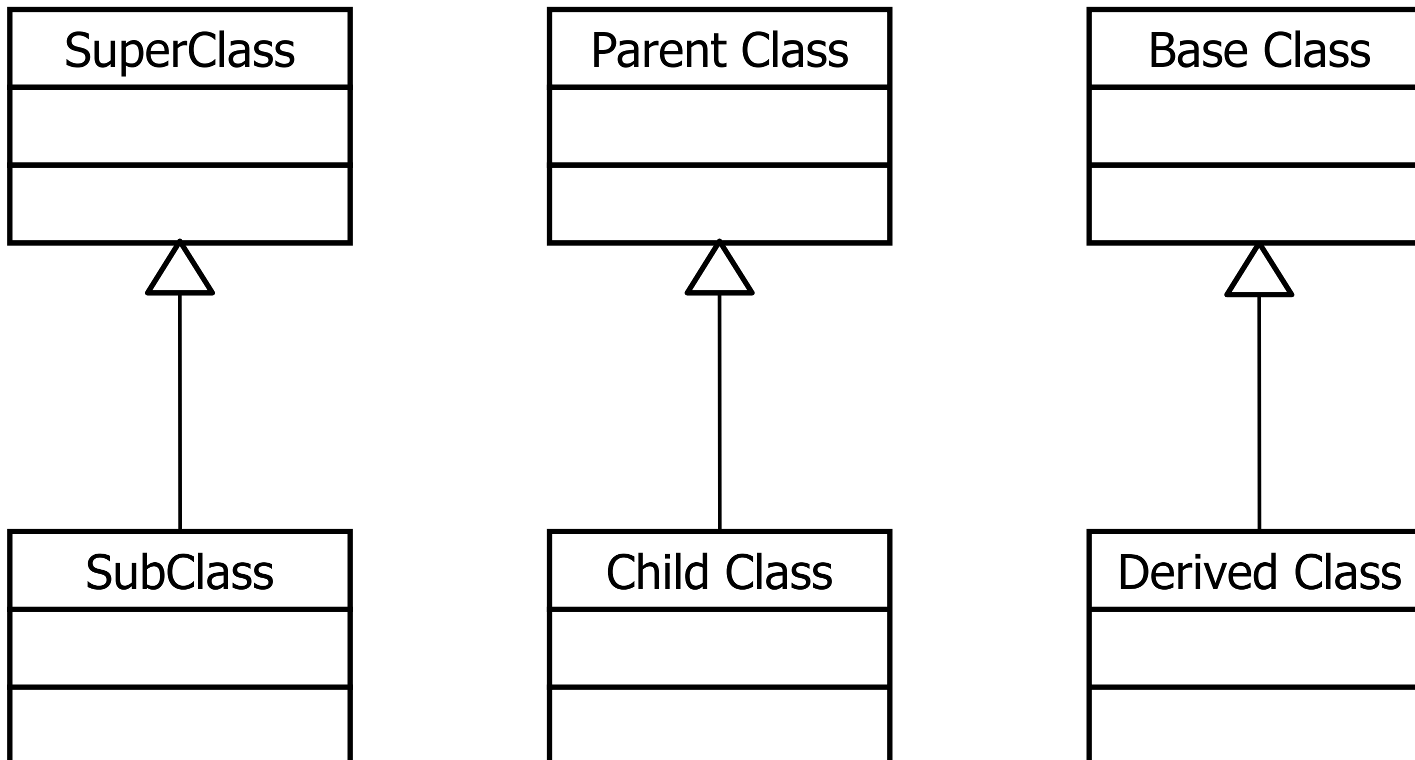
Tutto questo non ha ripercussioni nella superclasse

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

- 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

```
public class Point {
    private int x;
    private 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

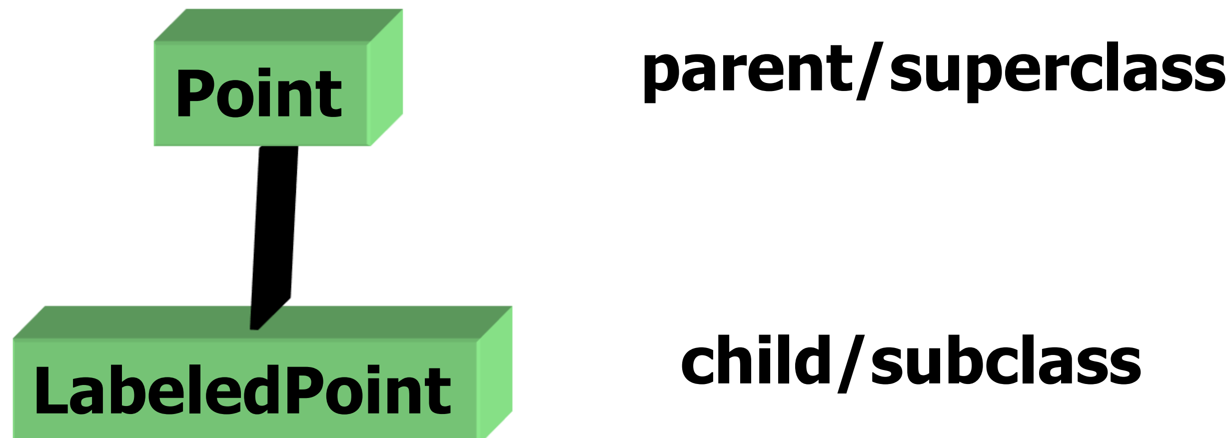
```
class NewPoint extends Point {

}

class Prova {
   public static void main(String args[]) {
         NewPoint pc = new NewPoint();
         pc.setX(42);
         pc.x=42;     //ERRORE
         NewPoint p2=new NewPoint(0,7); //ERRORE
         p2.setX(3);
         System.out.println(pc.getX()+","+pc.getY());
   }
}
```

Ereditarietà – Esempio

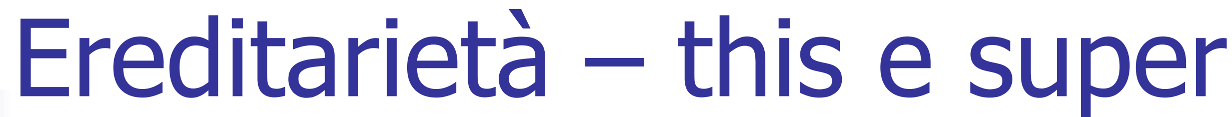
```
public class LabeledPoint extends Point {
    String name;
    public LabeledPoint(int x, int y, String name) {
        super (x,y);
        setName(name);
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
}
```

Ereditarietà – Esempio

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



- Cio' e' molto utile

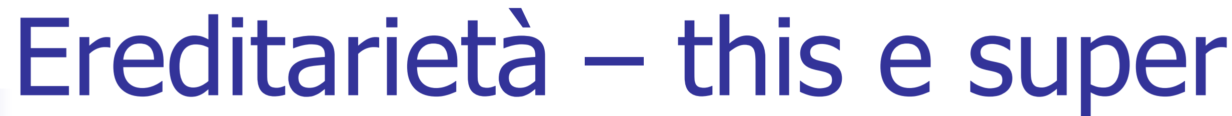


Ereditarietà – this e super

- 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

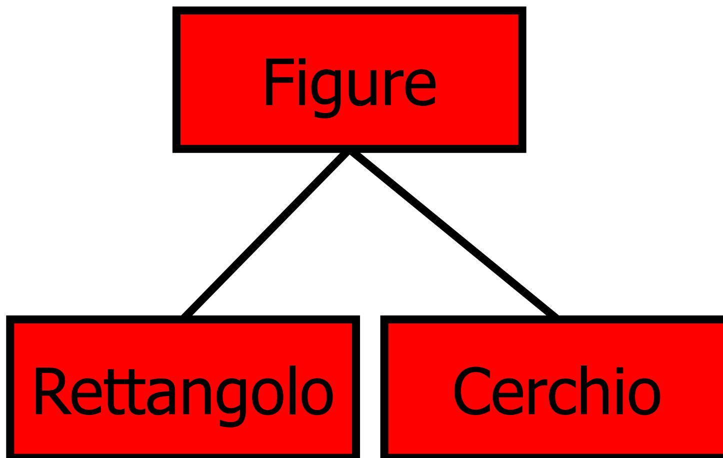
non si puo' invocare un costruttore di una classe dalle nipoti con super.super<something>



Ereditarietà – this e super

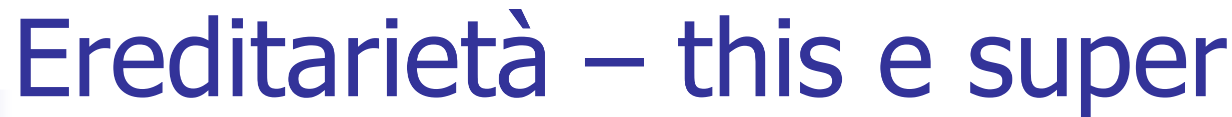
```
public class Nonno {
    public void method1() {...}
}
public class Padre extends Nonno {
    public void method1() {...}
    public void method2() {...}
}
public class Figlio extends Padre {
    public void method1() {...}
    public void method3() {
            method1();
            method2();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

Ereditarietà – this e super



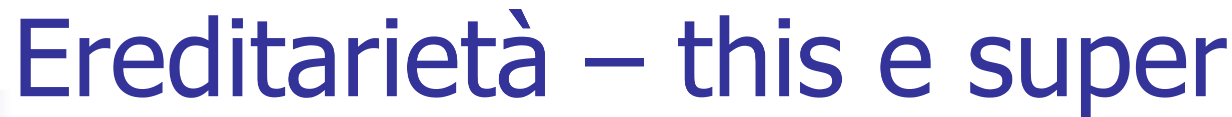

```
public class Figure {
   public String name;

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

   public int area () {
       return (0);
      }  // area

}
```

Ogni classe derivata implementera' il metodo area.



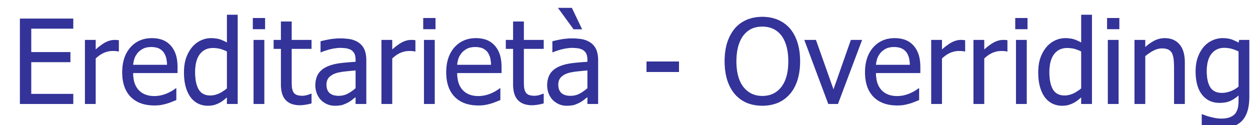
Ereditarietà – this e super

```
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) {
    this.base = base;
    this.altezza = altezza;
    this.name = name;
 } // costruttore
...
```



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
```

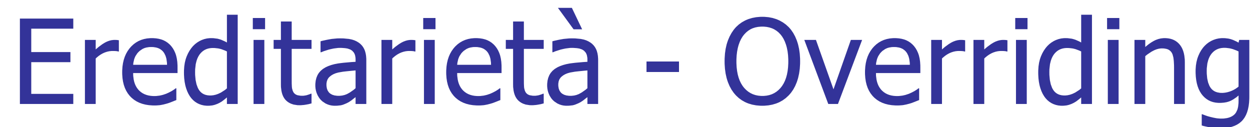


Ereditarietà - Overriding

- L'overriding è il riscrivere il codice di un metodo ereditato, solitamente fatto perché deve eseguire azioni differenti, ad esempio:

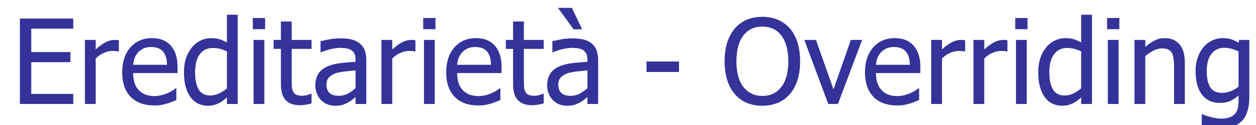
```
public class Cliente {
        int soldi;
        void paga( int costo ) { soldi=soldi-costo;}      }
public class ClienteConTesseraPunti extends Cliente {
        int punti;
        void paga (int costo) { soldi=soldi-costo; punti++;}   }
```

- In caso di overriding, la signature del metodo (nell'esempio *paga (int costo)*) deve essere la stessa del metodo della superclasse; se differiscono, si tratta di overloading e non di overriding



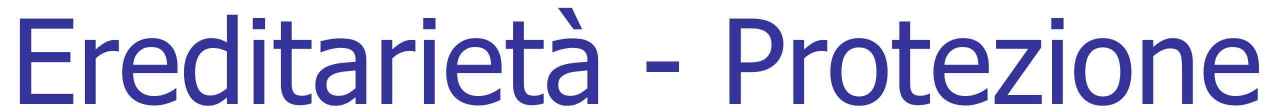
Ereditarietà - Overriding

- i metodi che operano l'overriding possono avere i propri modificatori di accesso, che possono tuttavia solo ampliare (rilassare) la protezione rispetto a quella assegnata al metodo originale nella madre
- ad esempio un metodo protected nella superclasse può essere protected o public ma non private nel corrispondente omonimo della sottoclasse, altrimenti verrebbe violato il principio che dovunque si usa un oggetto della classe madre deve essere possibile sostituirlo con un oggetto di una qualunque delle classi figlie (quella figlia avrebbe il metodo privato e quindi inutilizzabile)



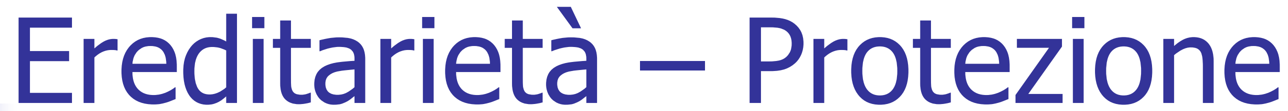
- Di un metodo, l'overriding è possibile solo se è accessibile; nel caso sia privato, e nella sottoclasse venga creato un metodo con la stessa signature di quello non accessibile della superclasse, i due metodi saranno considerati completamente scorrelati
- Di un attributo non si puo' fare l'overriding; se nella classe figlia definisco un campo identico ad uno ereditato, il nuovo campo rende inaccessibile quello definito nella superclasse (si dice che lo **adombra**) , a meno che non si utilizzi *super*

```
class madre {
    int x;
public figlia extends madre {
    int x;
    public esempio() { this.x=9; super.x=12}
    }}
```



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
- Si utilizza lo specificatore **protected**, applicabile ad attributi e metodi:
  - consente libero accesso a tutte le classi derivate da una, e solo a quelle;
  - Le classi non imparentate non possono accedere



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 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 privato, rendendo necessaria la scrittura di metodi omonimi per intervenire sull'attributo nuovo della figlia, operazione che di fatto rende inutile l'avere ereditato da un'altra classe (era meglio "accontentarsi" dell'attributo privato ereditato)

Ereditarietà relazione madre-figlia

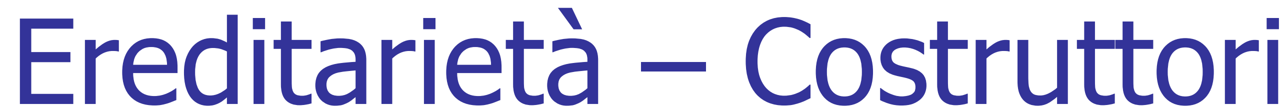
1. Un qualunque cambiamento effettuato nella superclasse (madre), modifica automaticamente le classi derivate, comprese le loro istanze (una sorta di "aggiornamento automatico")
2. Una classe madre tuttavia non è cosciente se ha o meno classi figlie
3. Una classe può avere figli, nipoti, pronipoti ecc. (gerarchia di ereditarietà)
4. Una classe può accedere alle informazioni della madre, perché appunto le eredita, pero' non può vedere niente al di sopra della madre (nonno, bisnonno… sono inaccesibili)
5. Una classe derivata non può avere meno caratteristiche della sua superclasse, in quanto il concetto guida dell'ereditarietà è che dovunque si usi un oggetto della madre, deve essere possibile sostituirlo con un oggetto di una qualunque delle classi figlie.
6. Due classi figlie della stessa madre (sorelle) non sono coscienti l'una dell'esistenza dell'altra

Ereditarietà relazione madre-figlia

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)
- ...

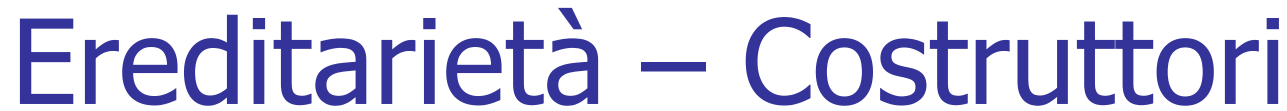


Ereditarietà – Costruttori

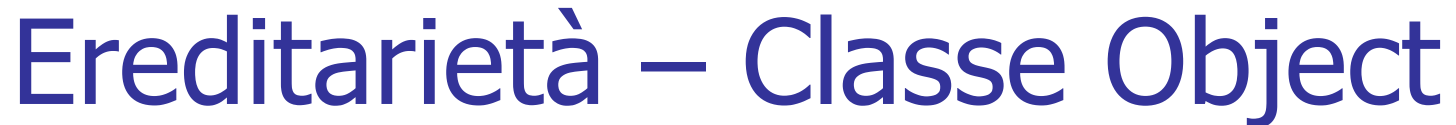
- Una classe derivata non può prescindere dalla classe base, perché ogni istanza della classe derivata comprende in sé, indirettamente, un oggetto della classe base.
- Quindi, ogni costruttore della classe derivata deve invocare un costruttore della classe base affinché esso costruisca la “parte di oggetto”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

```
public class LabeledPoint extends Point {
    String name;
    public LabeledPoint(int x, int y, String name) {
        super (x,y);    // LAVORO DI COMPETENZA DELLA MADRE
        setName(name);       // LAVORO SPECIFICO DELLA FIGLIA
    }
```

Ereditarietà – Costruttori

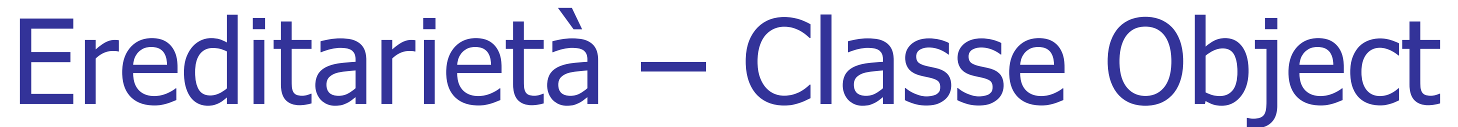


- Perché bisogna che ogni costruttore della classe derivata invochi un costruttore della classe base?
  - solo il costruttore della classe base può sapere come inizializzare i dati ereditati in modo corretto
  - solo il costruttore della classe base può garantire l'inizializzazione dei dati privati, 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.



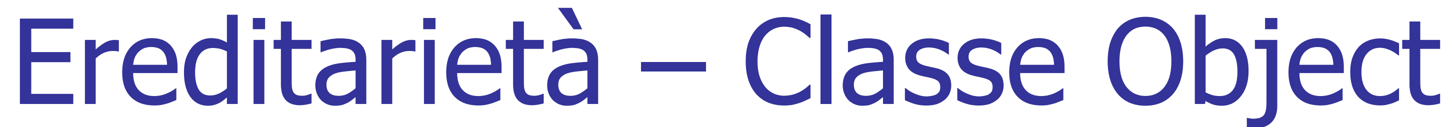
Ereditarietà – Classe Object

- 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 una scritta poco utile
  - *protected Object clone()*, che effettua la clonazione dell'oggetto



- Tutti gli oggetti hanno i metodi toString e equals, il che uniforma il modo di lavorare con tutte le classi
- quelli ereditati da object non sono pero' utili, meglio fare l'overriding:

```
Public class Cliente {
String nome;
String cognome;
int eta;
}

public class prova {
        public static void main (String args[]) {
        Cliente c=new Cliente();
        c.nome="Oronzo";
        System.out.println(c);          // STAMPA NON UTILE}}
```

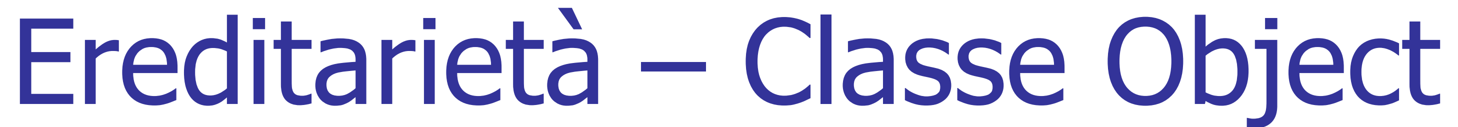


Ereditarietà – Classe Object

- Aggiungiamo il metodo toString:

Public class Cliente {
String nome;
String cognome;
int eta;
public String toString() {
return "il cliente si chiama "+nome+" "+cognome+" ed ha "+eta+" anni"; // FACCIO STAMPARE QUELLO CHE MI SERVE
}}

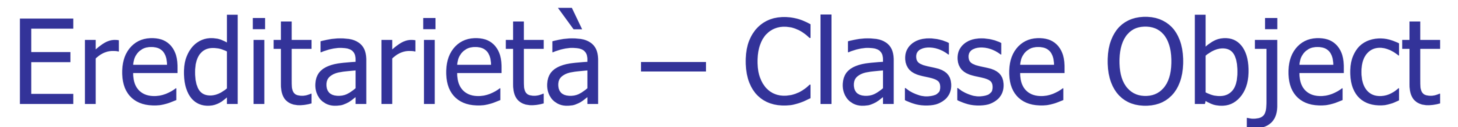
public class prova {
public static void main (String args[]) {
Cliente c=new Cliente();
c.nome="Oronzo";
System.out.println(c); // CHIAMA IL toString}}



Ereditarietà – Classe Object

- Anche il metodo equals serve, infatti il confronto fra due oggetti con == controlla solo se sono lo stesso oggetto (identica zona di memoria) e NON se hanno gli stessi valori nei campi:

```
Public class Cliente { … }

public class prova {
        public static void main (String args[]) {
        Cliente c1=new Cliente();
        Cliente c2=new Cliente();
        c1.nome="Oronzo";
        c2.nome="Oronzo";
        If (c1==c2) … // GLI RISULTANO DIVERSI, ANCHE SE I
            CAMPI SONO TUTTI UGUALI}}
```



Ereditarietà – Classe Object

- Overriding del metodo equals (che nella classe Object non fa nulla, come il toString()…):

```
Public class Cliente { …
        public boolean equals(Cliente c) {
        if  (this.nome==c.nome)  &&  (this.cognome==c.cognome)
&& (this.eta==c.eta) return true ;
        else return false; }}

public class prova {
        public static void main (String args[]) {
        Cliente c1=new Cliente();
        Cliente c2=new Cliente();
        …
        If (c1.equals(c2)) …
        // ORA FUNZIONA PER COME VOGLIAMO}}
```

Casting

Se abbiamo

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

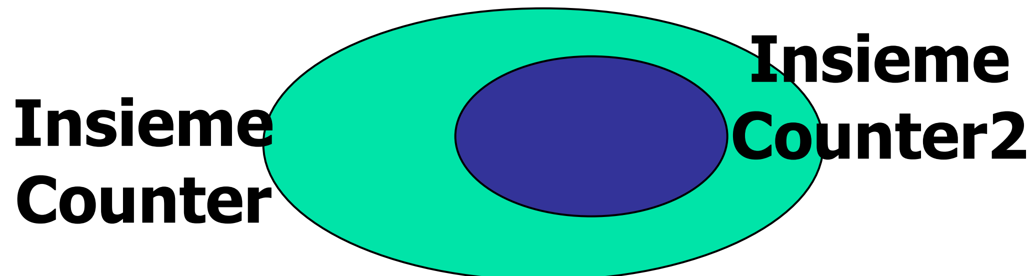
- Ogni oggetto di classe Counter2 è anche implicitamente di classe Counter, ma non viceversa; un Counter è meno ricco di un Counter2
- Counter2 può quindi essere usato al posto di un Counter se necessario
- Ogni Counter2 è anche un Counter

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

- 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

Letteralmente, la parola polimorfismo indica la possibilità per uno stesso oggetto di assumere più forme.
Utilizzando un esempio del mondo reale, si pensi al diverso comportamento che assumono un uomo, una scimmia e un canguro quando eseguono l'azione del camminare. L'uomo camminerà in modo eretto, la scimmia in maniera decisamente più goffa e curva mentre il canguro interpreterà tale azione saltellando.
Riferendoci ad un sistema software ad oggetti, il polimorfismo indicherà l'attitudine di un oggetto a mostrare più implementazioni per una singola funzionalità.

Polimorfismo

Lo scenario in cui si manifesta il polimorfismo è quello dell'ereditarietà e dell'overriding, ossia esistono metodi omonimi fra madre e figlia che (ragionevolmente) operano in maniera diversa pur avendo stesso numero e tipo di parametri, stesso nome e stesso tipo di ritorno, ossia stessa interfaccia (come "camminare" è il gesto comune uomo, scimmia, canguro).

Supponiamo quindi di avere:

```
public class Cliente {
        int soldi;
        void paga( int costo ) { soldi=soldi-costo;}        }
public class ClienteConTesseraPunti extends Cliente {
        int punti;
        void paga (int costo) { soldi=soldi-costo; punti++;}    }
```

Polimorfismo

Se ora abbiamo un codice che lavora con Cliente:

```
Public class Agenzia {
        …
        public void fai_pagare (Cliente c, soldi s) {
                c.paga(s)       }
        …
        public static void main(String args[]) {
        Agenzia a=new Agenzia();
        Cliente c1=new Cliente();
        ClienteConTesseraPunti c2=new ClienteConTesseraPunti();
        a.fai_pagare(c1,20);
        a.fai_pagare(c2,30);     // FUNZIONA
}
```

Polimorfismo

L'istruzione *a.fai_pagare(c1,20);*

È corretta perché passo un Cliente (c1), e il metodo *fai_pagare* è predisposto per accettare gli oggetti di tipo Cliente

L'istruzione *a.fai_pagare(c2,30);* non dovrebbe funzionare, ma:

1) dovunque si usa un oggetto della classe madre deve essere possibile sostituirlo con un oggetto di una qualunque delle classi figlie, quindi anche qui è possibile

2) Il polimorfismo del linguaggio permette al metodo *fai_pagare* di adattarsi a c1, invocando il *paga* di Cliente, e a c2, invocando il metodo *paga* di *ClienteConTesseraPunti*, anche se in *fai_pagare* si chiama sempre e solo *c.paga(s)* (si cerca il metodo appropriato)

3) Chiaramente questo è possibile perché ho due metodi *paga* identici nell'interfaccia, quindi l'overriding è essenziale

Polimorfismo

Se il polimorfismo non fosse esistito, il codice doveva tenere conto di tutti i casi possibili:

```
public class Cliente {
        void paga( int costo ) { soldi=soldi-costo;}       }
public class ClienteConTesseraPunti extends Cliente {
        void paga1 (int costo) { soldi=soldi-costo; punti++;}  }
Public class Agenzia {
        …
        public void fai_pagare (Cliente c, soldi s) {
          if c.instanceOf(Cliente)                         c.paga(s);
          elseif c.instanceOf(ClienteConTesseraPunti) c.paga1(s);
          elseif ALTRI CASI…
        …}
```

Polimorfismo

Quindi, l’ereditarietà ed l’overriding permettono di riciclare il codice della classe Agenzia, anche se nel futuro saranno create altre sottoclassi di Cliente (il riuso del codice si mantiene nel tempo)

Ad esempio, la classe Cliente potrebbe essere stata creata nel 1979, Agenzia nel 1980, ClienteConTesseraPunti nel 2010 e magari una altra figlia di Cliente nel 2050; la classe Agenzia resta riutilizzabile e funzionante anche dopo 70 anni!!!

Chiaramente, la classe Agenzia non ha percezione dell’esistenza delle nuove classi, quindi le accetta solo fintantochè si comportano come la madre; se si pretende di usare le caratteristiche specifiche delle figlie (ad esempio, “stampare i punti”), si deve necessariamente cambiare la classe Agenzia, eventualmente riscrivendola se il suo codice sorgente non è più accessibile

Polimorfismo

Esempio con diverse classi figlie:

```
class Animali {
   public void verso () { System.out.println(“sono un Animale"); }
}
class Pesce extends Animali {
   public void verso() {System.out.println("Glu glu");}
}
class Uccello extends Animali {
   public void verso() { System.out.println("Tweet tweet "); }
}
class Cane extends Animali {
   public void verso() { System.out.println("woof woof"); }
}
```

Polimorfismo

```
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
  woof woof
  Glu glu