

---

# Ereditarietà in Java

di Roberta Molinari



# Ereditarietà

---

- Java supporta solo l'**ereditarietà singola**: ogni classe al massimo ha una superclasse.
- La sottoclasse **eredita** tutti i membri "non privati" della sopraclasse e può usarli direttamente nel suo codice. Attenzione: gli attributi private sono ereditati (inizializzati,...), ma non sono visibili
- È possibile **estendere** la classe super aggiungendo nuovi membri alla sottoclasse
- È possibile **ridefinire** alcuni membri della sopraclasse. Gli attributi saranno nascosti *hiding* (si deve usare super.), mentre per i metodi si può fare l'**overriding** o l'**overloading**

▶ **overriding**: stessa firma, comportamento diverso nei metodi delle sottoclasse  
**overloading**: stesso nome, ma tipo o numero di parametri diverso

# Ereditarietà

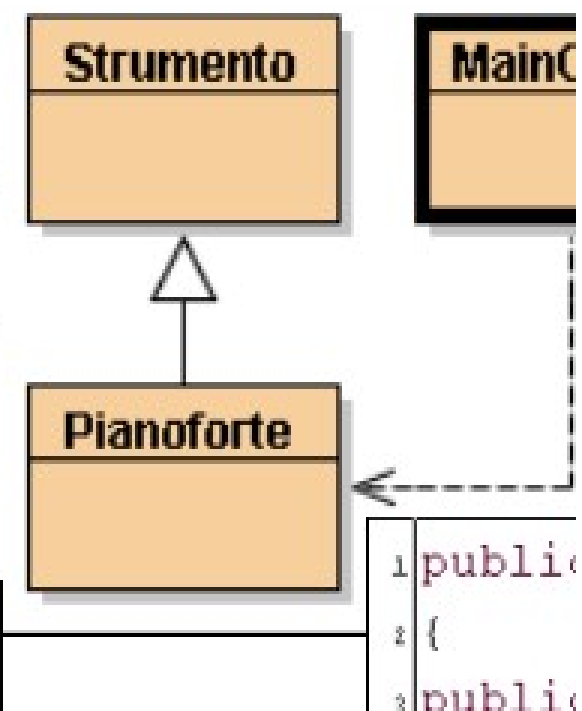
## extends

È possibile creare una gerarchia di classi, tramite l'istruzione

```
class NomeClasse extends nomeSuperClasse{}
```

```
1 class Strumento {
2 void play() {
3     System.out.println("play di strumento");
4 }
5 }
```

```
1 public class Pianoforte extends Strumento {
2 void play() {
```



```
1 public
2 {
3 public
```

# Ereditarietà

---

## final

- I **metodi final** non possono essere ridefiniti dalle sottoclassi: per tali metodi è vietato l'overriding (posso ridefinirlo con altri parametri). È bene che tali metodi operino su variabili finali o private, in modo da impedire cambiamenti illeciti indiretti

```
final tipoRitornato nomeMetodo([parametri])  
{body}
```

- Le **classi final** non possono essere ereditate e non possono avere sottoclassi

```
final class NomeClasse [extends SuperClasse]  
{ CorpoClasse }
```

Si possono avere 2 metodi **statici** con lo stesso nome, ma restano distinti e quello della sottoclasse non sovrascrive (overriding) quello della superclasse, ma lo nasconde (hiding). Entrambi restano raggiungibili attraverso il nome della classe (o con super). Se si cerca di ridefinirlo non statico si segnala errore

## overriding e overloading

- Per fare l'**overriding** di un metodo, nella classe base questo deve essere visibile (public o protected) e overridable (né finale, né statico perché si ha un hiding)

```
class Base
```

```
    [{Protected|Public}] NomeMetodo([par])
```

```
class Derivata extends Base
```

```
    [{Protected|Public}] NomeMetodo([par])
```

- Per fare l'**overloading** di un metodo (sono tutti overloadable per default) (ATTENZIONE non basta cambiare il tipo restituito)

```
    modifAccesso NomeMetodo([par])
```

```
    modifAccesso NomeMetodo([parDiversi])
```

# Il riferimento super

---

Sono implicitamente presenti in ogni classe

- **super**: fa riferimento alla sopraclasse dell'oggetto, si usa per riferirsi a variabili, metodi della superclasse (anche quelli sovrascritti dalla sottoclasse)

```
super.nomeMetodo(listaParamAttuali);  
super.nomeAttributo;
```

- **super()**: fa riferimento al costruttore della sopraclasse dell'oggetto (è invocato in automatico quando si crea un oggetto di una sottoclasse, è la prima istruzione eseguita nei costruttori delle sottoclassi). Può solo essere invocato in costruttori e deve essere la loro prima istruzione

```
NomeClasseDerivata(par, attr) {  
    super(par); //chiama costruttore della superclasse  
    super.attr =attr; //toglie ambiguità  
}
```

# Ereditarietà

## Costruttore di una sottoclasse

- I costruttori non vengono ereditati
- Ogni istanza della classe derivata comprende in sé, indirettamente, un oggetto della classe base, quindi quando istanzio un oggetto di una classe derivata, viene automaticamente richiamato il costruttore della classe base e poi quello della classe derivata. È possibile richiamare esplicitamente il costruttore della superclasse tramite l'istruzione `super(listaParametriAttuali)` posta come prima istruzione del costruttore
- Il costruttore è passibile di overloading

ATTENZIONE: se non c'è un costruttore nella sopraclasse senza parametri, devo definire nella classe derivata un costruttore che richiami in modo esplicito `super()` con i parametri richiesti (il compilatore dà un errore)

# Ereditarietà

## Costruttore di una sottoclasse

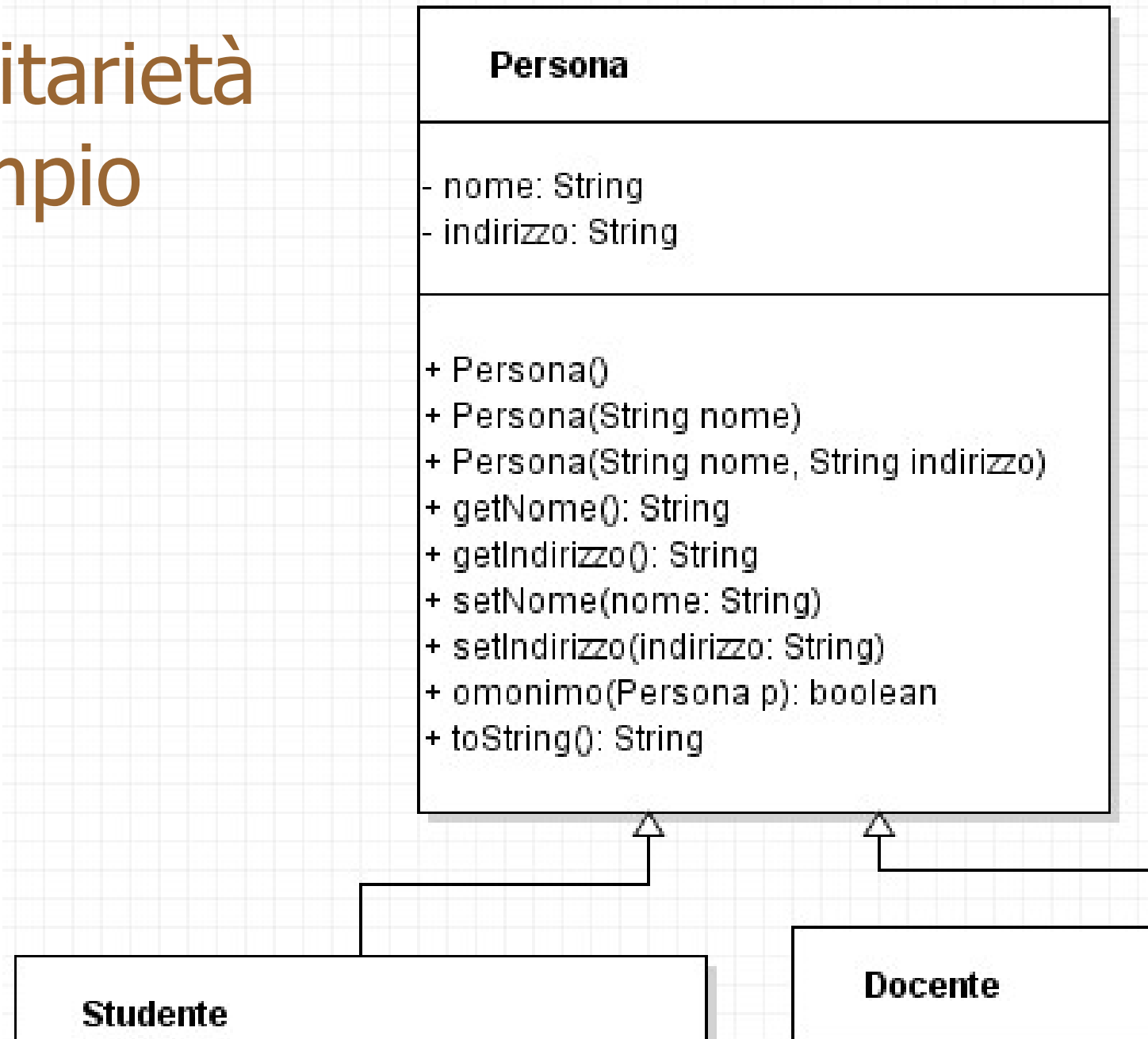
- La prima istruzione del costruttore di una sottoclasse può essere:
  - una chiamata esplicita ad un costruttore della superclasse
  - una chiamata esplicita ad un (altro) costruttore della classe corrente
  - una generica istruzione; in tal caso, Java implicitamente aggiungerà `super()` prima dell'esecuzione della prima istruzione del costruttore
- Quando si crea un oggetto tramite una new:
  1. viene allocata la memoria necessaria
  2. Gli attributi sono inizializzati 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

## Esempio

Java



# Ereditarietà

---

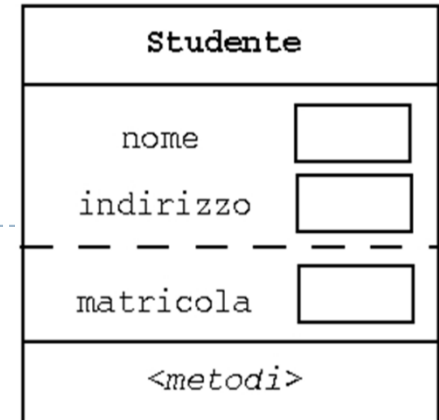
## Esempio

```
public class Persona {
    private String nome;
    private String indirizzo;
    public Persona() {this("anonimo","ignoto");}
    public Persona(String nome) {this(nome,"ignoto");}
    public Persona(String nome, String indirizzo) {
        this.nome = nome;
        this.indirizzo = indirizzo;}
    public void setNome(String nome) {this.nome = nome;}
    public void setIndirizzo(String i){this.indirizzo = i;}
    public String getNome() {return nome;}
    public String getIndirizzo(){return indirizzo;}
    public String toString() {
        return("Nome:" + nome + " Indirizzo: " + indirizzo);}
    public boolean omonimo(Persona p) {
        return this.nome.equalsIgnoreCase(p.nome);}
}
```

# Ereditarietà

---

## Esempio



```
public class Studente extends Persona {  
    // Studente eredita variabili e metodi da Persona  
    private int matricola;    // Nuova variabile istanza  
    static int nextMatricola = 1;    //Nuova var statica  
  
    public Studente(String nome, String indirizzo) {  
        super(nome, indirizzo);  
        this.matricola = nextMatricola++;  
    }  
    // Nuovo metodo  
    public int getMatricola() { return matricola;}  
    // Metodo sovrascritto  
    public String toString() {  
        return(super.toString()+ " Matricola: " +  
            matricola);  
    }  
}
```

# Ereditarietà

## Esempio

```
public class Docente extends Persona {  
    private float stipendio;  
    private String materie;  
    public Docente(String nome, String indirizzo, float  
        stipendio) {  
        super(nome, indirizzo);  
        this.stipendio = stipendio;  
        this.materie = "";  
    }  
    public float getStipendio(){return stipendio; }  
    public void aumentaStipendio(int aumento) {  
        this.stipendio += aumento; }  
    public String getMaterie() {return materie;}  
    public void aggiungiMateria(String m) {  
        materie += m+ "\n"; }  
    public String toString() {  
        return (super.toString() + " Stipendio: " +  
            stipendio + " Materie: " + materie);}  
}
```

# Ereditarietà

## La classe Object

- Tutte le classi discendono dalla classe **Object**
- Se una classe non eredita esplicitamente da una superclasse allora eredita implicitamente dalla classe Object
- La classe Object fornisce i seguenti metodi:
  - `boolean equals(Object)` : verifica se l'oggetto su cui è invocato è uguale (equivalente) a quello passato per argomento (restituisce true se due riferimenti sono alias)
  - `String toString()` : restituisce una rappresentazione testuale dell'oggetto in forma di stringa, è molto utile ad esempio per le stampe.

# Metodi della classe Object

## .equals()

Se si confrontano due oggetti con `==` sarà true se il valore delle variabili istanza sono `=` ovvero contengono lo stesso riferimento, puntano alla stessa area di memoria.

Dentro la classe Object è definito il metodo **equals()** che per default **è equivalente a `==`**:

```
public boolean equals (Object obj)
    { return this == obj; }
```

Se si vuole che due oggetti di una classe siano "uguali" perché hanno gli stessi valori negli attributi, si deve ridefinire il metodo `equals()` e definire il criterio per l'uguaglianza

Per ridefinire il metodo `equals()` in Eclipse cliccare con il tasto destro sul codice, quindi *Source* → *Generate hashCode() and equals()* e selezionare su quali attributi creare i metodi. Con IntelliJ *Generate... equals() and hashCode()*

# Metodi della classe Object

## .toString()

Restituisce la rappresentazione dell'oggetto come stringa.

Per i wrapper restituisce il valore dell'oggetto

```
Integer x=new Integer(5);  
System.out.print(x.toString()); //5
```

per gli oggetti restituisce il nome della classe seguita dal riferimento relativo all'oggetto

```
NomeClasse@indirizzoInMemoria
```

Per creare la propria rappresentazione dell'oggetto si deve ridefinire il metodo `toString()`

Per ridefinire il metodo `toString()` in Eclipse cliccare con il tasto destro sul codice, quindi *Source* → *Generate toString()* e selezionare quali attributi e/o metodi si vuole restituire il valore. IN IntelliJ *Generate... toString()*

# Classi astratte

---

- ▶ Una **classe astratta o virtuale** definisce
  - un tipo di dato
  - un insieme di operazioni sul tipo di dato
    - alcune operazioni sono implementate: metodi
    - alcune operazioni possono non essere implementate: metodi astratti o virtuali (abstract)
- ▶ Viene usata come classe base le cui sottoclassi vanno a specificare i comportamenti o le proprietà.



# Le classi astratte

---

- Una classe astratta non può essere istanziata, ma è possibile derivarne sottoclassi. Viene usata come classe base le cui sottoclassi vanno a specificare i comportamenti o le proprietà.

```
visibilità abstract class nomeClasse {  
    //definizione attributi  
    //definizione costruttori e metodi  
    //definizione metodi astratti  
}
```

- In genere possiede almeno un metodo definito **abstract** ovvero senza il corpo

```
public abstract metodo(par); //senza {}
```

Una classe che ha un metodo **abstract** deve essere definita **abstract**.

# Le classi astratte

---

- Può avere dei metodi costruttori, ma non può essere istanziata, si devono ridefinire nelle sottoclassi usando il `super()`. Può riferirsi a istanze di sottoclassi

```
Generica g;  
g=new SottoGenerica();
```

- Se una classe deriva da una classe astratta deve implementare tutti i metodi derivati per divenire concreta e poter istanziare oggetti, altrimenti rimane anch'essa astratta

# Le classi astratte

```

1 public abstract class Animale
2 {
3     public String nome;
4     Animale(String nome)
5     {
6         this.nome=nome;
7     }
8     public abstract String attivita();
9     public abstract String vive();
10    public abstract String mangia();
11    public void presentati()
12    {
13        System.out.println("Mi chiamo "+nome+
14                            " mi piace "+attivita()+", "+
15                            "vivo "+vive()+
16                            " e mangio "+mangia());
17    }
18 }

```

```

1 public class Main
2 {
3     public static void main(String[] args)
4     {
5         Animale[] a=new Animale[3];
6         a[0]=new Scoiatta();
7         a[1]=new Scoiatta();
8         a[2]=new Leone();
9
10        for(int i=0;i<a.length;i++)
11        {
12            a[i].presentati();
13        }
14    }
15 }

```

```

1 public class Scoiatta extends Animale

```

BlueJ: Terminal Window

# Le classi interface

---

- ▶ Le classi **interface** sono un'evoluzione del concetto di classe astratta. Contengono solo le firme di metodi pubblici e la dichiarazione di attributi pubblici, final e statici. Non possono contenere codice

```
public interface nome{...}
```

- ▶ Contengono solo

- metodi `public abstract`
- attributi `public static final`

- ▶ Una classe le può implementare (realizzare) definendo tutti i "corpi" dei metodi

```
[public] class C implements int [,int2]
```

# Le classi interface

---

```
[public] interface Int //senza public è package
    [extends Int1, Int2, ...] {
    tipol var1= val1;          ...
    tipoRes1 metodo1 ( parametri );    ...
}
```

- Gli attributi sono sempre `static final` (sottinteso)
- I metodi sempre `public` e `abstract` (sottinteso), saranno `public` anche quelli delle sottoclassi
- Una classe può implementare una o più interfacce, ma estendere una sola sovraclassa. In questo modo si supera il limite della ereditarietà singola

```
public
    class    nomeClasse    [extends    nomeSuperClasse]
                        implements    Int1, Int2, ...{    ...
    }
```