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 sovraclasse. Gli attributi saranno nascosti *hiding* (si deve usare super.), mentre per i metodi si può fare **l'overriding** o l'**overloading**

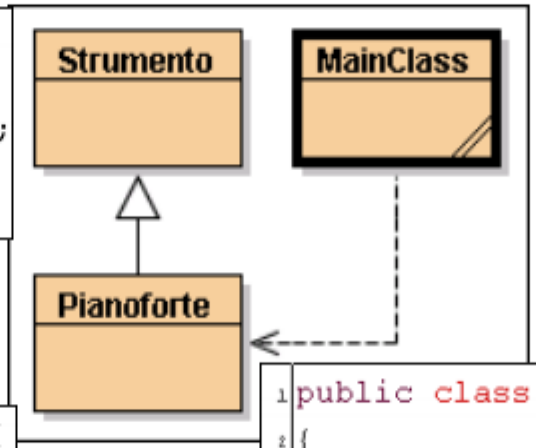
overriding: stessa firma, comportamento diverso nei metodi delle sottoclasse
overloading: stesso nome, ma tipo o numero di parametri diverso

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() {
3 System.out.println("play di pianoforte");
4 }
5 }
```

```
1 public class MainClass
2 {
3 public static void main(String arg[])
4 {
5     Pianoforte p=new Pianoforte();
6     p.play();
7 }
8 }
```


141

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 }
```


142

Ereditarietà
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])
```

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

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
}
```


144

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)

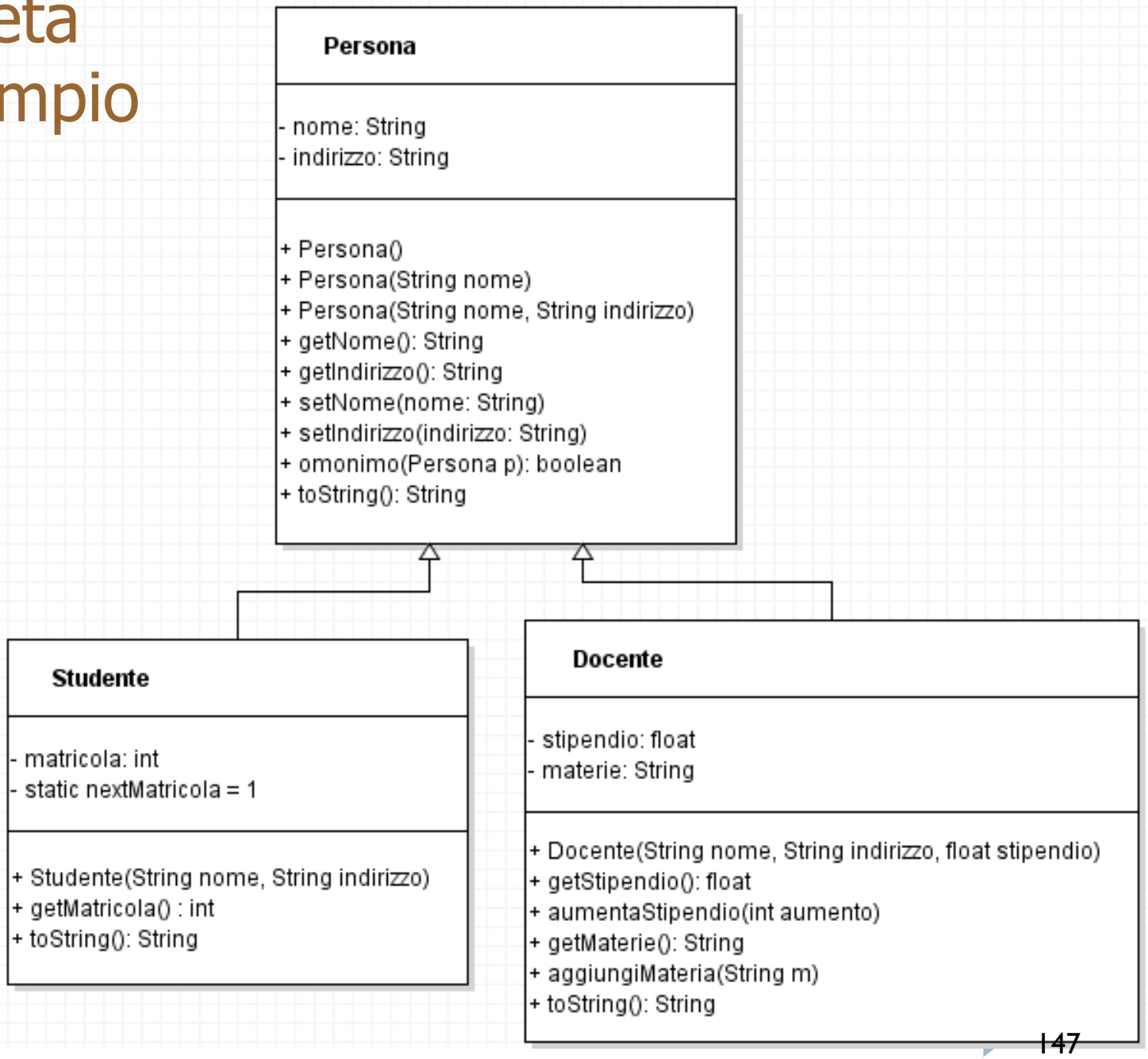
145

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 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

146

Ereditarietà
Esempio



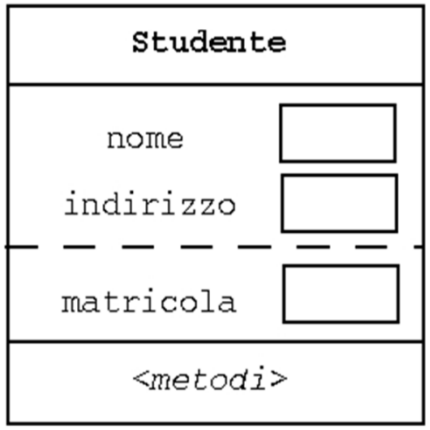
147

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);}
}
```


148

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);
  }
}
```


149

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);}
   }
```


150

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 la rappresentazione dell'oggetto come stringa ovvero il nome della classe e l'indirizzo dell'oggetto.

151

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: il nome della classe e l'indirizzo dell'oggetto.

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()*

Le 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.

155

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();
```

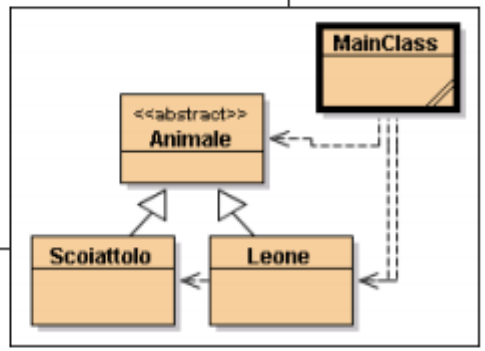
- Se una classe deriva da una classe astratta deve implementare tutti i metodi derivati per divenire concreta e poter istanziare oggetti, altrimenti rimane anch'essa astratta

Le classi astratte

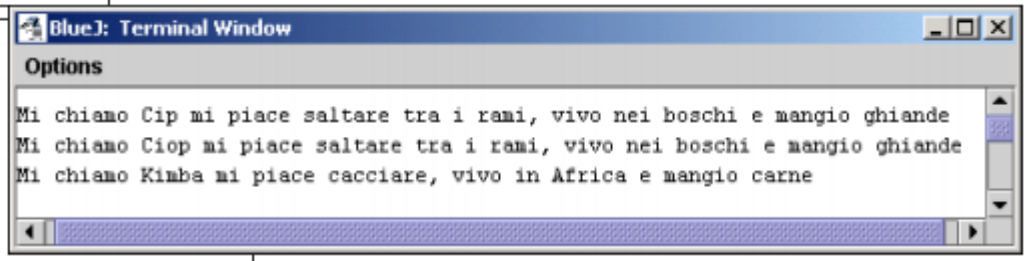
```
public abstract class Animale
{
public String nome;
Animale(String nome)
{
this.nome=nome;
}
public abstract String attivita();
public abstract String vive();
public abstract String mangia();
public void presentati()
{
System.out.println("Mi chiamo "+nome+
                   " mi piace "+attivita()+", "+
                    "vivo "+vive()+
                    " e mangio "+mangia());
}
}
```

```
public class MainClass
{
public static void main(String arg[])
{
Animale[] a=new Animale[3];
a[0]=new Scoiattolo("Cip");
a[1]=new Scoiattolo("Ciop");
a[2]=new Leone("Kimba");

for(int i=0;i<3;i++)
{
a[i].presentati();
}
}
}
```




```
public class Scoiattolo extends Animale
{
public Scoiattolo(String s)
{
    super(s);
}
public String attivita(){ return "saltare tra i rami";}
public String vive(){ return "nei boschi";}
public String mangia(){ return "ghiande";}
}
```



157

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 Interf [,Interf2]
```


158

Le classi interface

```
[public] interface Inter //senza public è
  package
        [extends Inter1, Inter2, ...] {
   tipo1 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 sopraclasse. In questo modo si supera il limite della ereditarietà singola

```
public
  class nomeClasse [extends nomeSuperClasse]
    implements Inter1, Inter2, ...{...}
```


159

Le classi interface
interfacce funzionali

Le interfacce come *ActionListener*, la cui definizione è presente sotto, vengono chiamate in Java 8, **interfacce funzionali (functional interface)** e sono caratterizzate dalla presenza di un solo metodo. Oltre alle classi EventListener, interfacce come *Runnable*, *Comparator* o *FileFilter* sono da considerarsi in modo simile. Le interfacce funzionali sono sfruttate per l'utilizzo con le espressioni lambda.

```
package java.awt.event;
import java.util.EventListener;

        public interface ActionListener extends
  EventListener {
      public void actionPerformed(ActionEvent e);
    }
```


160

Funzioni anonime
Espressioni Lambda

- In matematica e informatica in generale, **un'espressione lambda è una funzione.**
- In Java, un'espressione lambda fornisce un modo per creare una funzione anonima, introducendo di fatto un nuovo tipo: *funzione anonima,* che può essere passato come argomento o restituito in uscita nei metodi.
- È una sorta di scorciatoia che consente di scrivere una funzione nello stesso posto dove ti serve.

```
(Lista degli argomenti) -> Espressione
```

oppure

```
(Lista degli argomenti)->{ istruzioni; }
```

Nota:

- è possibile omettere il tipo dei parametri
- è possibile omettere le parentesi se c'è solo un parametro.

161

Funzioni anonime
Espressioni Lambda

```
// prende in input due interi e restituisce la somma
(int x, int y) -> x + y
// prende in input una stringa e restituisce la sua
  lunghezza
s -> s.length()
// espressione senza argomenti che restituisce 50
() -> 50
// prende in input una stringa e non restituisce nulla
(String s) -> { System.out.println("Benvenuto " +s);}
```

- Le istruzioni di *break* e *continue* non si possono usare all'interno del blocco anche se sono permessi all'interno di cicli.
- Se il corpo produce un risultato, ogni possibile ramo del flusso del codice deve restituire qualcosa o lanciare un'eccezione.

162

Funzioni anonime
Espressioni Lambda

```
public class RunnableTest {
 public static void main(String[] args) {
  System.out.println("=== RunnableTest ===");

  // Anonymous Runnable
  Runnable r1 = new Runnable(){
      @Override
      public void run(){System.out.println("Hello old!");
      }
  };

  // Lambda Runnable
  Runnable r2 = () -> System.out.println("Hello Lambda!");

   r1.run();
   r2.run();
 }
}
```


163

Funzioni anonime
Espressioni Lambda

lambda expression passata come argomento di un metodo

```
// Anonymous ActionListener
JButton testButton = new JButton("Test Button");
testButton.addActionListener(new ActionListener(){
   @Override public void actionPerformed(ActionEvent ae){
      System.out.println("Click Detected by Anon Class");}
   });

// Lambda ActionListener
testButton.addActionListener (e ->
  System.out.println("Click Detected by Lambda Listner"));

// Swing stuff
JFrame frame = new JFrame("Listener Test");
. . .
```


164