



# Java

Capitolo 5 – Ereditarietà

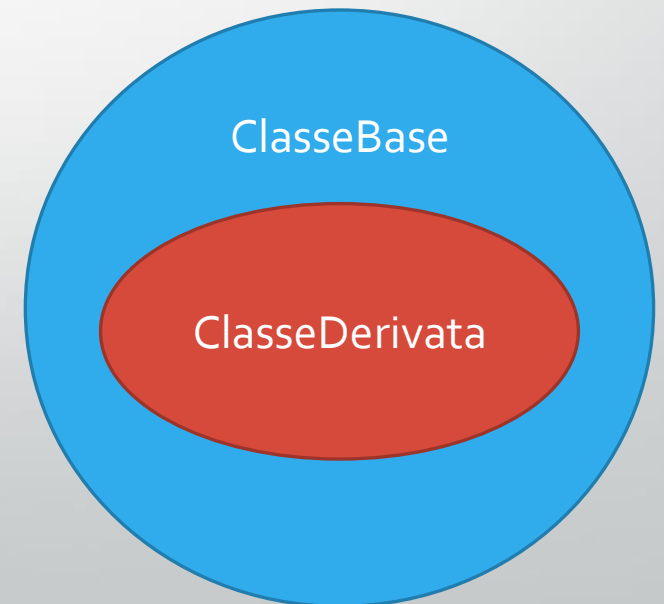
Prof. Ivan Gentile

# Riferimenti

- <https://profs.sci.univr.it/~macedonio/web/Teaching/progBioinfo2014/20-Ereditarieta.pdf>

# Ereditarietà: cos'è?

- Meccanismo che permette di creare nuove classi sulla base di quelle esistenti
  - Si dice anche **derivazione**
- In particolare, permette di
  - **riusare** il codice (metodi e campi);
  - **aggiungere** nuovi metodi e nuovi campi;
  - **ridefinire** metodi e campi esistenti (**overriding**)



# Sintassi

- **Classe A:** *classe base, classe padre o superclasse*
- **Classe B:** *classe derivata, classe figlio, o sottoclasse*

```
class A {  
    // ...  
}  
class B extends A {  
    // Differenze rispetto ad A  
}
```

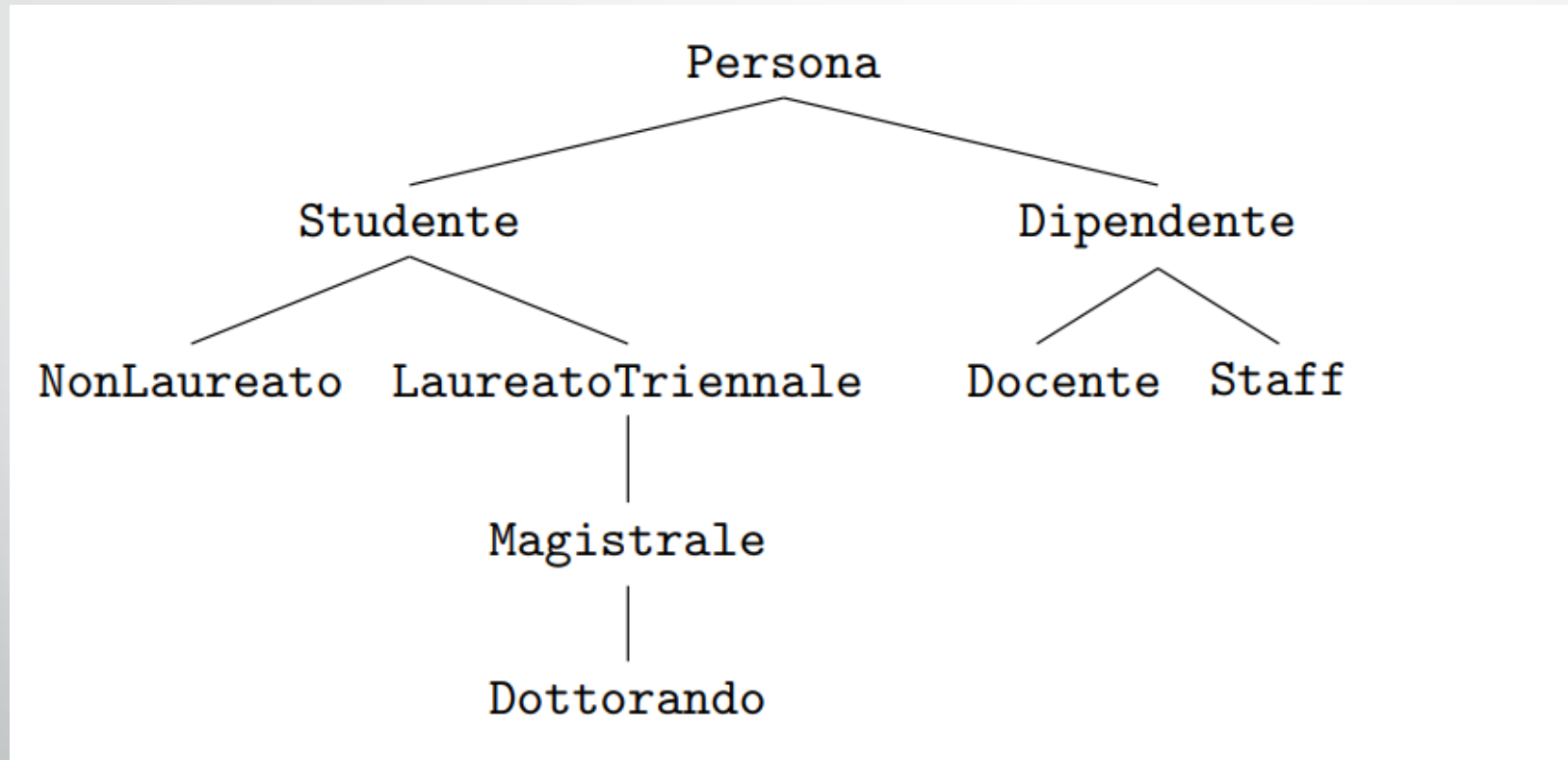
# La classe Object: introduzione

- Tutte le classi di Java derivano implicitamente dalla classe **Object**
- Quindi anche se non lo scriviamo è come se scrivessimo `extends Object`
- Se una classe B deriva da A allora non deriva (direttamente) da Object
  - Ma alla fine della risalita della gerarchia ci sarà una classe che deriva da Object
- La classe Object la vedremo in seguito

# Cosa si eredita?

- La sottoclasse
  - eredita tutti i membri della superclasse
    - Ma *non può accedere* ai membri privati della superclasse
    - Solo a quelli `public` o `protected`
  - può aggiungere nuovi membri
  - può ridefinire i metodi (**override**) della superclasse
    - A volte anche per gli attributi si parla di *override*, ma è più corretto per essi parlare di **hiding** (**nascondere, adombrare**) come vedremo

# Esempio



# Classe Persona

```
1 public class Persona {  
2     private String nome;  
3  
4     public Persona() {  
5         nome = "Ancora nessun nome";  
6     }  
7  
8     public Persona(String nomeIniziale) {  
9         nome = nomeIniziale;  
10    }  
11  
12    public void setName(String nuovoNome) {  
13        nome = nuovoNome;  
14    }
```

```
15    public String getName () {  
16        return nome;  
17    }  
18  
19    public String toString() {  
20        return "persona: " + nome;  
21    }  
22  
23    public boolean haLoStessoNome(Persona other) {  
24        if (other != null)  
25            return this.nome.equalsIgnoreCase(other.nome);  
26        return false;  
27    }  
28 }
```



# Classe Studente



```
1 public class Studente extends Persona {
2
3     private int matricola;
4
5     public Studente() {
6         super(); ←
7         matricola = 0;
8     }
9
10    public Studente(String nome, int matricola) {
11        super(nome); ←
12        this.matricola = matricola;
13    }
14
15    public void setMatricola(int nuovaMatricola) {
16        matricola = nuovaMatricola;
17    }
18
19    public void reimposta(String nuovoNome, int nuovaMatricola) {
20        setNome(nuovoNome);
21        setMatricola(nuovaMatricola);
22    }
23 }
```

```
23 public int getMatricola() {
24     return matricola;
25 }
26
27 public String toString() { ←
28     return "studente: " + getNome() + " (" + matricola + ")";
29 }
30
31 public boolean equals(Studente other) {
32     if (other == null)
33         return false;
34     return haLoStessoNome(other) && (matricola == other.matricola);
35 }
36 }
```

# Uso della classe Studente

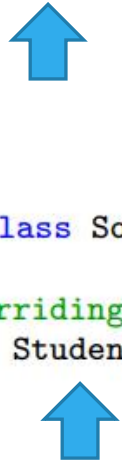
```
1 public class Main {  
2  
3     public static void main(String[] args) {  
4         Studente s = new Studente();  
5         s.setNome("Alan Turing"); // metodo ereditato dalla classe Persona  
6         s.setMatricola(1234); // metodo definito dalla classe Studente  
7         System.out.println(s.toString());  
8     }  
9  
10 }
```

## Output

studente: Alan Turing (1234)

# Focus sull'overriding

```
public class SuperClasse {  
    ...  
    public Persona getIndividuo(String nome) {  
        ...  
    }  
    ...  
}  
  
public class SottoClasse extends SuperClasse {  
    ...  
    // overriding  
    public Studente getIndividuo(String nome) {  
        ...  
    }  
    ...  
}
```



L'overriding del metodo `getIndividuo` cambia il tipo di ritorno. Tale ridefinizione non introduce una restrizione: uno `Studente` è una `Persona` con alcune proprietà aggiuntive.

# Overloading vs Overriding

## Overriding

La nuova definizione di un metodo nella sottoclasse ha lo **stesso nome**, lo **stesso tipo di ritorno** (a meno di covarianza) e gli **stessi parametri** in termini di tipo, ordine e numero.

## Overloading

Il metodo nella sottoclasse ha lo **stesso nome** e lo **stesso tipo di ritorno**, ma un **numero diverso di parametri** o anche un **solo parametro di tipo differente** dal metodo della superclasse.

# Esempio di Overloading

```
public class Persona{  
    ...  
    public String getNome() {  
        return nome;  
    }  
    ...  
  
public class Studente extends Persona {  
    ...  
    public String getNome(String titolo) {  
        return titolo + getNome();  
    }  
    ...  
}
```

La classe Studente ha ora **due** metodi `getNome`:


- Eredita il metodo `String getNome()` da `Persona`
- Definisce il metodo `String getNome(String titolo)`

# Estendere Sottoclassi: la Classe NonLaureato

```
1 public class NonLaureato extends Studente {
2     private int annoDiCorso; // 1 per il primo anno, 2 per il secondo anno
3                               // 3 per il terzo anno, 4 fuori corso
4
5     public NonLaureato() {
6         super();
7         annoDiCorso = 1;
8     }
9
10    public NonLaureato(String nome, int matricola, int annoDiCorso) {
11        super(nome, matricola);
12        setAnnoDiCorso(annoDiCorso);
13    }
14
15    // overloading
16    public void reimposta (String nuovoNome, int nuovaMatricola, int
17                           nuovoAnnoDiCorso) {
18        reimposta(nuovoNome, nuovaMatricola); // metodo reimposta() di Studente
19        setAnnoDiCorso(nuovoAnnoDiCorso);
20    }
```

# Estendere Sottoclassi: la Classe NonLaureato

```
20 public int getAnnoDiCorso() {
21     return annoDiCorso;
22 }
23
24 public void setAnnoDiCorso(int annoDiCorso) {
25     if ((1 <= annoDiCorso) && (annoDiCorso <= 4))
26         this.annoDiCorso = annoDiCorso;
27 }
28
29 public String toString() {
30     return getNome() + " (" + getMatricola() + ") " + " anno: " + annoDiCorso;
31 }
32
33 // Overriding? No, overloading!
34 // la classe Studente definisce il metodo boolean equals (Studente other)
35 public boolean equals(NonLaureato other) {
36     return super.equals(other) && (this.annoDiCorso == other.annoDiCorso);
37     // osservazione:
38     // con la chiamata super.equals(other) copriamo anche il caso other == null)
39 }
40 }
```



Come vedremo sarebbe  
meglio fare **Overriding**

# Esempio

```
1 public class Main {  
2  
3     public static boolean confrontaMatricole(Studente s1, Studente s2) {  
4         if (s1 != null && s2 != null)  
5             return (s1.getMatricola() == s2.getMatricola());  
6         return false;  
7     }  
8  
9     public static void main(String[] args) {  
10        Studente a = new Studente("Alan Turing", 1234);  
11        NonLaureato b = new NonLaureato("James Gosling", 1234, 4);  
12        System.out.println(confrontaMatricole(a, b));  
13        System.out.println(a.haLoStessoNome(b));  
14        System.out.println(b.haLoStessoNome(a));  
15    }  
16 }
```

Output:

```
true  
false  
false
```



# Il modificatore **protected**

- Spesso si dice che un membro **protected** è accessibile solo alle classi derivate
- In realtà il modificatore **protected** è più complicato

# Bloccare l'ereditarietà: `final`

- Si può decidere che da una certa classe non se ne possano ereditare altre
- Usare il modificatore **`final`** nel nome della classe
- `String` è un esempio di classe `final`

# Upacasting

- Si può cambiare il **riferimento** facendolo puntare a uno della classe padre o figlia.
- Se lo facciamo puntare alla classe figlio si parla di **upcasting**
  - Bisogna leggerlo da destra: converto un figlio in un genitore
- `Persona p = new Persona(...);`
- `Studiante s = new Studiante(...);`
- `p = (base) s; // Upcasting: (base) non necessario (implicito)`
- Tanto un figlio ha tutto del genitore

# Downcasting

- Se lo facciamo puntare alla classe di un padre si parla di **dowscasting**
  - Bisogna leggerlo da destra: converto un genitore genitore
- Normalmente non si può fare
- `Persona p = new Persona(...);`
- `Studiante s = new Studiante(...);`
- `s = p; // Downcasting (implicito) ERRORE`
  - Perché un padre non può fare tutto quello che può fare il figlio
- Bisogna fare il casting esplicitamente
  - `s = (Studiante) p; // Downcasting (esplicito)`
- E poi bisogna stare attenti a quali metodi e attributi usiamo

# Upcasting: serve al late binding

```
1 package animali;  
2  
3 public class Animale {  
4  
5     public void verso() {  
6         System.out.println("Sono un Animale (generico)!");  
7     }  
8  
9 }
```



```
1 package animali;  
2  
3 public class Pesce extends Animale {  
4  
5     public void verso() {  
6         System.out.println("Tweet tweet flap flap!");  
7     }  
8  
9 }
```

```
1 package animali;  
2  
3 public class Uccello extends Animale {  
4     public void verso() {  
5         System.out.println("Tweet tweet flap flap");  
6     }  
7  
8 }
```

```
1 package animali;  
2  
3 public class Cane extends Animale {  
4     public void verso() {  
5         System.out.println("Bau Bau Bau!");  
6     }  
7 }
```

# Upcasting: serve al late binding

```
1 package zoo;
2 import animali.*;
3
4 public class Zoo {
5     public static void main (String[ ] argv) {
6         Animale[ ] animali = new Animale[3];
7
8         animali[0] = new Uccello();
9         animali[1] = new Cane();
10        animali[2] = new Pesce();
11
12        for(Animale animale: animali)
13            animale.verso( );
14    }
15 }
```



Tweet tweet flap flap  
Bau Bau Bau!  
Tweet tweet flap flap!

Con il binding statico scriverebbe  
sempre *"Sono un Animale (generico)!"*

# Classe Object (1/2)

La classe `Object` è antenata di ogni classe:  
ogni oggetto di tipo classe è un `Object`.

Se una classe non ne estende esplicitamente un'altra, essa estende automaticamente `Object`. Infatti le seguenti dichiarazioni sono equivalenti:

```
public class C {  
    ...  
}
```

```
public class C extends Object {  
    ..  
}
```

È possibile scrivere metodi che hanno parametri di tipo `Object`. In questo modo sarà possibile passare come argomenti oggetti di qualsiasi tipo classe.

# I metodi di Object

- Che possono essere sovrascritti
  - clone
  - equals/hashCode
  - finalize
  - toString
- Che non possono essere sovrascritti
  - getClass
  - notify
  - notifyAll
  - wait



## Due Metodi della Classe Object

`String toString()` "Returns a string consisting of the name of the class of which the object is an instance, the at-sign character @, and the unsigned hexadecimal representation of the hash code of the object."

`boolean equals(Object other)` "Implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values *x* and *y*, this method returns true if and only if *x* and *y* refer to the same object (*x* == *y* has the value true)."

Questi due metodi non possono funzionare correttamente per tutte le classi, perché troppo generici. È necessario ridefinirli con nuove definizioni più appropriate.

# Overloading del metodo equals

La classe `Studente` **eredita** dalla classe `Object` il metodo

```
boolean equals(Object other)
```

e **definisce** il metodo

```
boolean equals(Studente other)
```

**Questo è overloading non overriding!**

I due metodi hanno **differenti tipi di parametro**, quindi la classe `Studente` ha entrambi i metodi.

# Il modo migliore per fare un metodo equals: **overriding**

equals su una classe  
di nome "Classe"

```
1 public boolean equals(Object obj) {  
2     if (this == obj || obj == null)  
3         return false;  
4     if (this == obj)  
5         return true;  
6     if (getClass() != obj.getClass())  
7         return false;  
8     Classe c = (Classe) obj;  
9     if (this.primitivo1 == c.getPrimitivo1() &&  
10        this.primitivo1 == c.getPrimitivo1() &&  
11        ... // etc. per tutti i tipi primitivi  
12        this.oggetto1.equals(c.getOggetto1()) &&  
13        this.oggetto2.equals(c.getOggetto2())  
14        ... // etc. per tutti i tipi oggetto  
15    )  
16        return true;  
17    return false;  
18 }
```

Nota: E' un  
deep equals

# Clonare un oggetto: metodo clone di Object

- Object:
  - `protected Object clone() throws CloneNotSupportedException {...}`
- Effettua una shallow copy: va ridefinito
- L'eccezione `CloneNotSupportedException` viene lanciata se l'oggetto non è istanza della tag interface **Cloneable**

# Clonare un oggetto: Deep Copy

```
import java.awt.*;

class Segmento {
    Point i, f;

    ...

    public Object clone() {
        try {
            Segmento s;
            s=(Segmento) super.clone();

            s.i=(Point) this.i.clone();
            s.f=(Point) this.f.clone();

            return s;
        }
        catch(CloneNotSupportedException e) {
            return null;
        }
    }
}
```

# Clonare un oggetto Shallow Copy

Classe  
Base

```
1 public class ClasseBase implements Clonable {
2     // Attributi
3     // Metodi
4     public Object clone() {
5         try {
6             return super.clone(); // oppure return (Classe) super.clone();
7         }
8         catch (CloneNotSupportedException e) {
9             return null; // oppure throw e;
10        }
11    }
12
13 }
```

Classe  
Derivata

```
15 public class ClasseDerivata {
16     // Attributi
17     // Metodi
18     public Object clone() {
19         return super.clone(); // oppure return (ClasseDerivata) super.clone();
20     }
21
22 }
```

# Classe Astratta

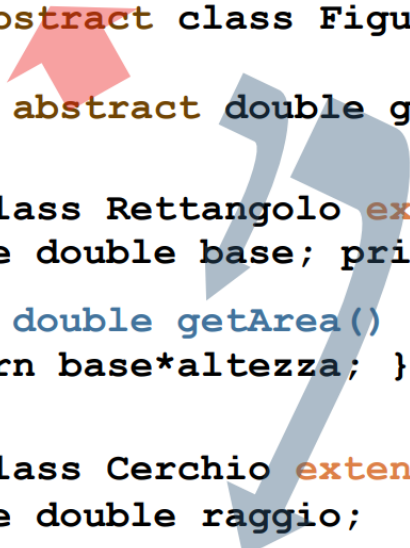
- E' una classe di cui non possono essere istanziati oggetti
- Serve solo per fattorizzare attributi o metodi comuni a più classi concrete che saranno derivate da essa
- **`abstract public class NomeClasse{...}`**
- La dichiarazione di un metodo astratto rende la classe astratta
- **`abstract public tipoRestituito nomeMetodo(...);`**
- Nelle classi derivate vanno implementati tutti i metodi astratti
- Altrimenti sono astratte anch'esse

# Classe Astratta

```
public abstract class Figura
{ ...
    public abstract double getArea();
}

public class Rettangolo extends Figura
{ private double base; private double altezza;
  ...
  public double getArea()
  { return base*altezza; }
}

public class Cerchio extends Figura
{ private double raggio;
  ...
  public double getArea()
  { return raggio*raggio*Math.PI; }
}
```





# Interfaccia

- E' uno schema che può essere utile a determinate classi che vogliono rispettarle
- Gli attributi sono solo costanti o static
- Ogni metodo è automaticamente pubblico e astratto

```
public interface AnimaleTerrestre {  
    public deambula();  
}
```

# Interfacce

- Una classe può implentare più interfacce
- Deve definire tutti i metodi delle interfacce che implementa

```
public interface Primate extends Mammifero {  
    public void manipola();  
}
```

```
public interface SuperPippo  
    extends Mammifero, SuperEroe {  
    ...  
}
```

```
public class MammiferoTerrestre extends Mammifero  
    implements AnimaleTerrestre {  
    public void deambule() {DEFINIZIONE!}  
}
```