Java: Ereditarietà

Damiano Macedonio

Dipartimento di Informatica, Università degli Studi di Verona

Corso di Programmazione per Bioformatica

Ereditarietà in Sintesi

- È uno dei concetti chiave della programmazione orientata agli oggetti.
- Rafforza il riutilizzo del software.
- Permette di definire una classe in una forma molto generale che può essere utilizzata come base di partenza per definire classi derivate che
  - sono specializzazioni della classe generale;
  - ereditano metodi e le variabili di istanza della classe generale;
  - possono definire nuovi variabili di istanza e nuovi metodi.

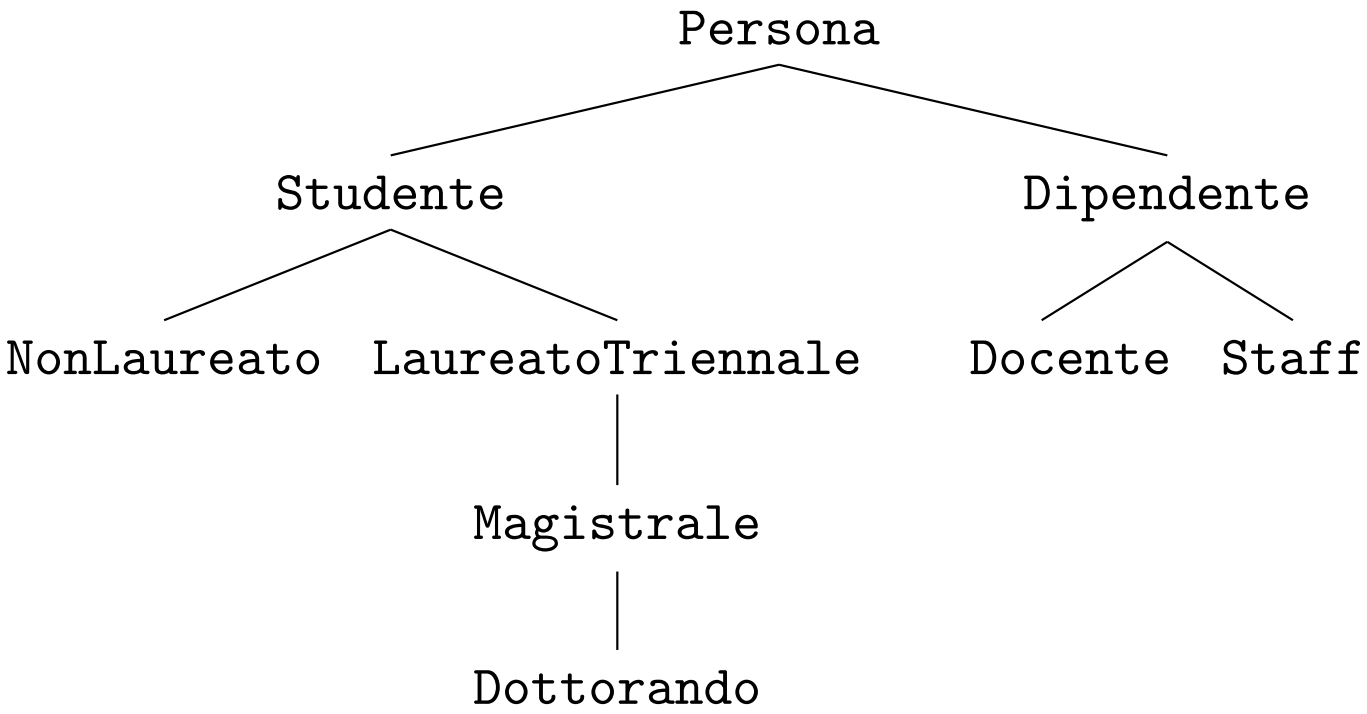
La classe Persona

```
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 setNome(String nuovoNome) {
13     nome = nuovoNome;
14   }
```

La classe Persona

```
15    public String getNome () {
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  }
```

Superclassi / Sottoclassi
Classi Base / Classi Derivate
Antenati / Discendenti



La 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   }
```

La classe Studente

```
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  }
```

La Keyword `extends`

```
public class Studente extends Persona
```

- La classe `Studente` è una sottoclasse di `Persona`;
- La classe `Persona` è una superclasse di `Studente`;
- La sottoclasse eredita
  - tutte le variabili di istanza della superclasse
  - tutte le variabili statiche della superclasse
  - tutti i metodi pubblici della superclasse
- La classe `Studente` non deve ridefinire le variabili di istanza e i metodi pubblici ereditati.
- La classe `Studente` deve definire solo le variabili di istanza e i metodi aggiuntivi.

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)
```

Overriding: Ridefinire i Metodi Ereditati

Se una sottoclasse definisce un metodo che ha lo stesso nome, gli stessi parametri (in termini di tipo, ordine e numero) e anche lo stesso tipo di ritorno di un metodo della superclasse, il metodo della sottoclasse ridefinisce il metodo della superclasse.

Nel codice precedente

```
System.out.println(s.toString())
```

viene usata la definizione di `toString()` della classe `Studente`, non la definizione della classe `Persona`, poiché `s` è un oggetto della classe `Studente`.

Overriding e Tipo di Ritorno

- Quando si ridefinisce un metodo, si può cambiare a piacere il corpo della sua definizione, ma non si può modificare la sua intestazione.
- Esiste un solo caso in cui è possibile modificare il tipo di ritorno.

Tipo di Ritorno Covariante

Se il tipo di ritorno è una classe, il metodo ridefinito può restituire una qualsiasi delle sue sottoclassi.

Esempio

```
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.

Overriding e Modificatori di Accesso

Un metodo dichiarato come private nella superclasse può essere ridefinito come public nella sottoclasse.

```
public class SuperClasse {
  ...
  private void doSomething() {
    ...
```

```
public class SottoClasse extends SuperClasse {
  ...
  public void doSomething() {
    ...
```

In generale...

Il modificatore di accesso può essere ridefinito in un qualsiasi modo che renda più permissivo l'accesso.

Overriding vs. Overloading

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: 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)`

Campi `private`

Accessibilità

Le varibili di istanza che sono dichiarate `private` in una superclasse non sono accessibili per nome dai metodi delle sottoclassi.

Esempio

```
public class Persona {
  private String nome;
  ...

public class Studente extends Persona {
  ...
```

Il campo `nome` è ereditato dalla classe `Studente`, ma è accessibile soltanto attraverso i metodi definiti nella classe `Persona`.

Esempio

Uso Scorretto

```
1  public class Studente extends Persona {
2
3    private int matricola;
4    ...
5    public String toString() {
6      return "studente: " + nome + " (" + matricola + ")";
7      //ERRORE: il campo nome è private in Persona
8    }
```

Uso Corretto

```
1  public class Studente extends Persona {
2
3    private int matricola;
4    ...
5    public String toString() {
6      return "studente: " + getNome() + " (" + matricola + ")";
7      //CORRETTO: accediamo con un metodo public di Persona
8    }
```

Metodi `private`

Accessibilità

I metodi che sono dichiarati `private` in una superclasse non sono invocabili dai metodi delle sottoclassi. Possono essere invocati solo dai metodi della superclasse stessa.

Una sottoclasse può invocare un metodo pubblico `m1` della superclasse che a sua volta chiama un metodo privato `m2`, a patto che `m1` e `m2` siano stati definiti nella stessa superclasse.

Costruttori nelle Sottoclassi

Una classe derivata ha i suoi costruttori e non eredita nessun costruttore dalle superclassi che estende.

```
1  public class Studente extends Persona {
2
3    private int matricola;
4    ...
5    public Studente(String nome, int matricola) {
6      super(nome); // invoca il costruttore Persona(nome) della superclasse
7      this.matricola = matricola;
8    }
9    ...
```

La parola chiave `super`

- Viene invocata come un nome di un metodo per invocare un costruttore della superclasse.
- Deve essere sempre la prima azione specificata nella definizione di un costruttore.
- Se non viene inclusa un'invocazione esplicita al costruttore della superclasse, Java includerà automaticamente una invocazione al costruttore di default della superclasse.
- Se la superclasse non ha definito un costruttore di default e nel costruttore della sottoclasse si omette l'invocazione a uno dei costruttori della superclasse, si ottiene un errore di compilazione.

Sono equivalenti:

```
public Studente() {
  super();
  matricola = 0;
}
```

```
public Studente() {

  matricola = 0;
}
```

`this` VS. `super`

`this` riferisce alla classe che si sta definendo.
`super` riferisce alla superclasse che si sta estendendo.

- Quando utilizzato all'interno di un costruttore, l'invocazione a `this` deve essere la prima azione.
- Quando utilizzato all'interno di un costruttore, l'invocazione a `super` deve essere la prima azione.

Quindi...

La definizione di un costruttore non può contenere sia un'invocazione a `this` che una a `super`.

Come si possono fare entrambe le invocazioni?

Si utilizza `this` per invocare un costruttore che ha `super` come prima azione!

Invocare un Metodo Ridefinito

Nella definizione di un metodo di una sottoclasse si può invocare un metodo della superclasse che è stato ridefinito facendolo precedere da `super.`

```
public class Studente extends Persona {
  ...
  public String chiamaSuper() {
   return super.toString(); // chiama il metodo della superclasse
  }
  ...
```

```
public class Main {

public static void main(String[] args) {
    Studente s = new Studente();
    s.setNome("Alan Turing");
    s.setMatricola(1234);
    System.out.println(s.toString()); // output: "studente: Alan Turing (1234)"
    System.out.println(s.chiamaSuper()); // output: "persona: Alan Turing"
  }
}
```

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   // overloanding
16   public void reimposta (String nuovoNome, int nuovaMatricola, int
         nuovoAnnoDiCorso) {
17     reimposta(nuovoNome, nuovaMatricola); // metodo reimposta() di Studente
18     setAnnoDiCorso(nuovoAnnoDiCorso);
19   }
```

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  }
```

Relazione *is-a*

Ogni `Studente` è una `Persona`.

Ogni `NonLaureato` è uno `Studente`.

Ogni `NonLaureato` è una `Persona`.

Un oggetto può comportarsi come se fosse di più tipi in virtù dell'ereditarietà.

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
```

Compatibilità di assegnamento

```
Persona p1 = new Persona(); // legale
Persona p2 = new Studente(); // legale
Persona p3 = new NonLaureato(); // legale

Studente s1 = new Persona(); // Type mismatch! ILLEGALE
Studente s2 = new Studente(); // legale
Studente s3 = new NonLaureato(); // legale

NonLaureato n1 = new Persona(); // Type mismatch! ILLEGALE
NonLaureato n2 = new Studente(); // Type mismatch! ILLEGALE
NonLaureato n3 = new NonLaureato(); // legale
```

La Classe `Object`

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.

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.

Overriding del metodo `equals`

Operatore `instanceof`

Espressione: *`oggetto`* `instanceof` *`NomeClasse`*

Valore: `true` se *`oggetto`* non è `null` ed è di tipo *`NomeClasse`*; `false` altrimenti.

Overriding

```
1 public boolean equals(Object other) {
2   if (other instanceof Studente) {
3     Studente altroStudente = (Studente) other; // cast di other a Studente
4     return haLoStessoNome(altroStudente) && (matricola == altroStudente.matricola);
5   }
6 return false;
7 }
```