Programmazione Orientata agli Oggetti

Ereditarietà:

dettagli e approfondimenti

Contenuti

- Ereditarietà: dettagli e approfondimenti
  - Membri protetti
- La gerarchia di classi Java
  - **`Object`**
- Ereditarietà multipla
- Metodi e classi **`final`**

Accesso ai membri

- I membri **`private`** della classe base non sono accessibili dall'esterno nemmeno da una classe estesa
- In Java esiste un altro modificatore di accesso, che consente l'accesso ai campi alle sole sottoclassi
- È il modificatore di accesso: **`protected`**
- Un membro di una superclasse con accesso protetto è visibile a tutte le sottoclassi

Accesso ai membri: `protected`

- Più precisamente, è possibile accedere ad un membro **`protected`**:
  - da tutte le classi estese (anche se in package diversi)
  - da tutte le classi dello stesso package
- I membri protetti sono una violazione (seppure controllata) dell'information hiding
  - Vanno pertanto usati con molta accortezza
  - Se possibile vanno evitati

Overriding e information hiding

- Solo i metodi pubblici e protetti della classe base possono essere ridefiniti
- Se proviamo a ridefinire un metodo **`private`** della classe base quello che otteniamo è un nuovo metodo

Metodi e classi `final`

- Per evitare che un metodo possa essere ridefinito si usa il modificatore **`final`**
  - Esempio

```
public class boo {
 public final int foo(){…}
}
```

    In questo modo il metodo **`foo()`** non può essere ridefinito nelle classi che estendono la classe **`boo.`**
  - Verrebbe sollevato un errore a tempo di compilazione

```
public class goo extends boo {
 public int foo(){} // ERRORE
}
```

Metodi e classi `final`

- E' possibile rendere **`final`** una intera classe
  - Esempio

```
public final class NonMiPoteteEstendere {
…
}
```

    La classe **`NonMiPoteteEstendere`** non può essere estesa
  - Anche in questo caso verrebbe sollevato un errore a tempo di compilazione

```
public class CiProvo
        extends NonMiPoteteEstendere { //ERRORE
…}
```

La gerarchia delle classi Java: `Object`

- In Java tutte le classi estendono automaticamente la classe **`Object`**
- E' una classe predefinita, che viene automaticamente estesa da ogni nuova classe (direttamente o indirettamente)
- La classe **`Object`** ha un insieme di metodi, che sono ereditati e possono essere ridefiniti da ogni nuova classe
- Tra questi metodi ce ne sono alcuni noti!

`Object`: alcuni metodi

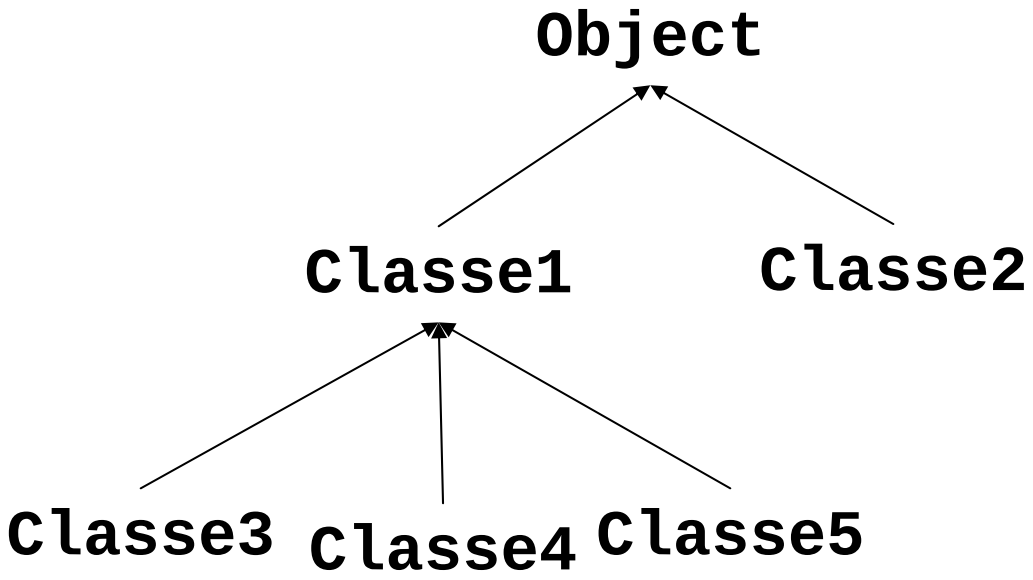
- Vedere documentazione
  - **`boolean equals(Object o)`**
  - **`int hashCode()`**
  - **`String toString()`**

Gerarchie di classi

- In Java ogni classe estende sempre una ed una sola classe
- Tranne **`Object`**, che è la radice predefinita delle classi
- Non ci può essere ereditarietà multipla
- Ma una classe può essere estesa da molte classi

La gerarchia delle classi in Java

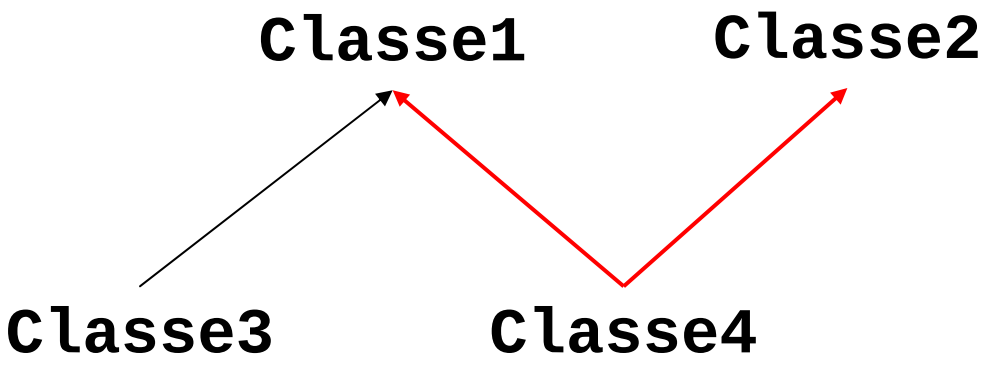
- Un'unica radice: **`Object`**
- Ogni classe* ha una e una sola superclasse
- Ogni classe* può avere zero o più sottoclassi



* con l'eccezione di **`Object`**

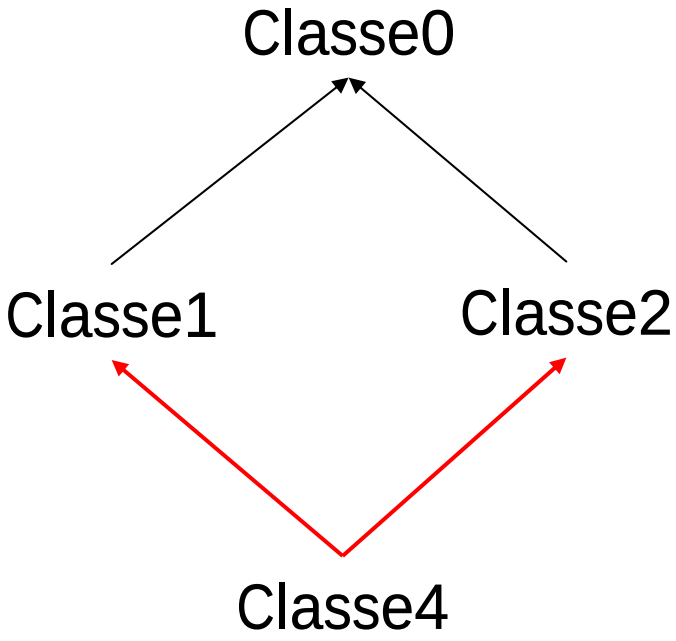
L'ereditarietà multipla non è ammessa

- **`Classe4`** erediterebbe sia da **`Classe1`** che da **`Classe2`**

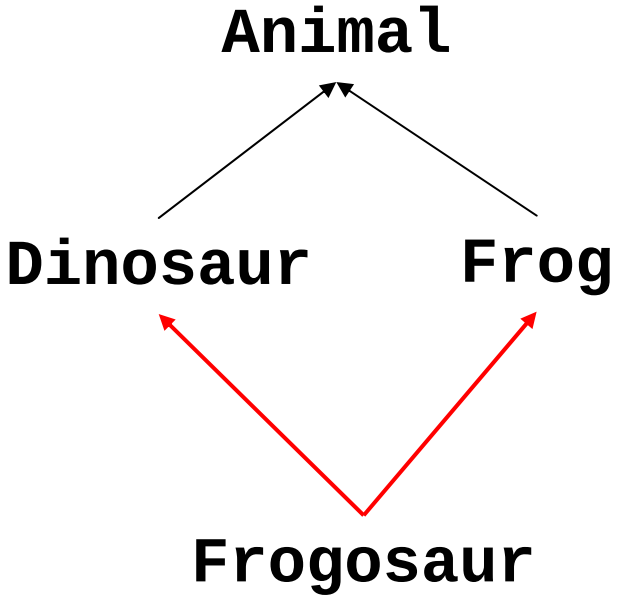


- In Java questo **non** è possibile
- Creerebbe problemi: quali ?

Problemi con l'ereditarietà multipla: *l'ereditarietà a diamante*



Un Esempio: il *"frogosauro"*



La gerarchia del frogosauro

```
class Animal {
  void talk() {
      System.out.println("…");
}

class Frog extends Animal {
    void talk() {

          System.out.println("Ribit, ribit.");
  }
}

class Dinosaur extends Animal {
      void talk() {
          System.out.println("I'm a dinosaur: I'm
  OK! ");
      }
}
```

Il frogosauro

```
// (non compila.)
   class Frogosaur extends Frog, Dinosaur {
   }
```

Cosa dovrebbe fare la seguente chiamata a **`talk()`**?

```
Animal animal = new Frogosaur();
animal.talk();
```

Problemi con l'Ereditarietà multipla

- Se un membro (metodo o campo) è definito in entrambe le classi base, da quale delle due la classe estesa "eredita"?
- E se le due classi base a loro volta estendono una superclasse comune ?
- Il problema è legato alle implementazioni (che vengono ereditate)
- Per questo motivo:
  - una classe può implementare tante interfacce
  - ma può estendere una sola classe

Java Interface ed ereditarietà

- Abbiamo visto che anche una interface può essere definita per estensione da un'altra interface
- Il problema dell'ereditarietà multipla non sussiste con le interface
- Quindi una interface può estendere anche più di una interface
- … e una classe può implementare più di una interface