Programmazione ad Oggetti

Ereditarietà

A.A. 2022/2023

Docente: Prof. Salvatore D'Angelo
Email: salvatore.dangelo@unicampania.it

Università degli Studi della Campania *Luigi Vanvitelli*
Dipartimento di Ingegneria

Riuso del codice

Composizione

- Qualcosa di già visto
- Gli attributi della nostra classe sono oggetti di classi già esistenti (della VM o create da noi)

Ereditarietà

- È una dei meccanismi fondamentali della programmazione ad oggetti

Composizione

Classi componenti

```
class Engine {
   public void start() {}
   public void rev() {}
   public void stop() {}
}
class Wheel {
    public void
inflate(int psi) {}
}
class Window {
   public void rollup()
{}
   public void rolldown()
{}
}
```

```
class Door {
  public Window window =
   new Window();
  public void open() {}
  public void close() {}
}
```

Classe principale

```
public class Car {
    public Engine engine = new Engine();
    public Wheel[] wheel = new Wheel[4];
    public Door left , right;  // 2-door
    public Car() {
        left = new Door(); right = new Door();
        for(int i = 0; i < 4; i++)
            wheel[i] = new Wheel();
                }

public static void main(String[] args) {
        Car car = new Car();
        car.left.window.rollup();
        car.wheel[0].inflate(72);
    }
}
```

Inizializzazione dei componenti

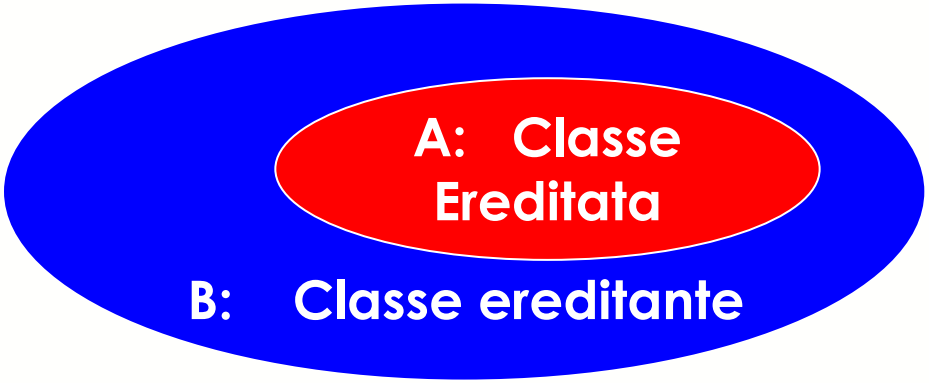
Occorre fare attenzione ad inizializzare gli oggetti componenti di una classe:

- Nella dichiarazione
- Nel costruttore della classe
- Appena prima di essere usati

Ereditarietà

Ereditarietà

La relazione di ereditarietà equivale alla relazione di inclusione tra gli insiemi.
Dire che una classe B eredita un'altra A equivale a dire che B ha sicuramente tutti gli attributi ed i metodi di A.
Ereditando A possiamo estendere la sua definizione completando B



Indicazioni d'uso !

Si usa la **composizione** quando una classe deve fornire le funzionalità realizzate in altre classi già esistenti

Si usa l'**ereditarietà** quando la nuova classe deve presentare un estensione dell'interfaccia della vecchia classe

Esempio

Classe veicolo:
- Pilota
- Ruote

- **Classe Auto**
  - Eredita veicolo
  - Volante
  - Motore
  - Targa

- **Classe Bici**
  - Eredita veicolo
  - Manubrio
  - Pedali

- **Classe Autotreno**
  - Eredita auto
  - Rimorchio

Ereditarietà

Esiste però anche un altro motivo, di ordine pratico, per cui conviene usare l'ereditarietà, oltre quello di descrivere un sistema secondo un modello gerarchico; questo secondo motivo è legato esclusivamente al concetto di **riuso del software**

In alcuni casi si ha a disposizione una classe che non corrisponde esattamente alle proprie esigenze. Anziché scartare del tutto il codice esistente e riscriverlo, si può seguire con l'ereditarietà un approccio diverso, costruendo una nuova classe che eredita il comportamento di quella esistente, salvo che per i cambiamenti che si ritiene necessario apportare

Tali cambiamenti possono riguardare sia l'aggiunta di nuove funzionalità che la modifica di quelle esistenti

Ereditarietà

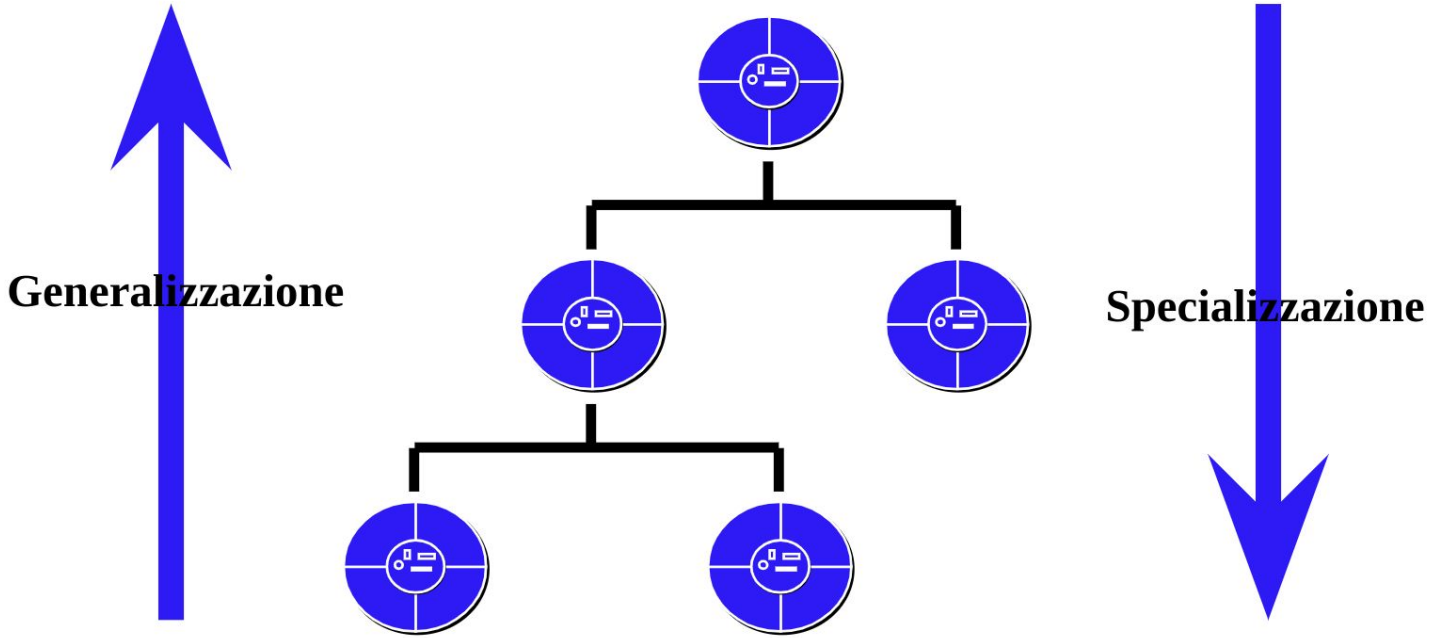
In definitiva, l'ereditarietà offre il vantaggio di ridurre i tempi di sviluppo, in quanto minimizza la quantità di codice da scrivere quando occorre:

- definire un nuovo tipo d'utente che è un sottotipo di un tipo già disponibile, oppure
- adattare una classe esistente alle proprie esigenze

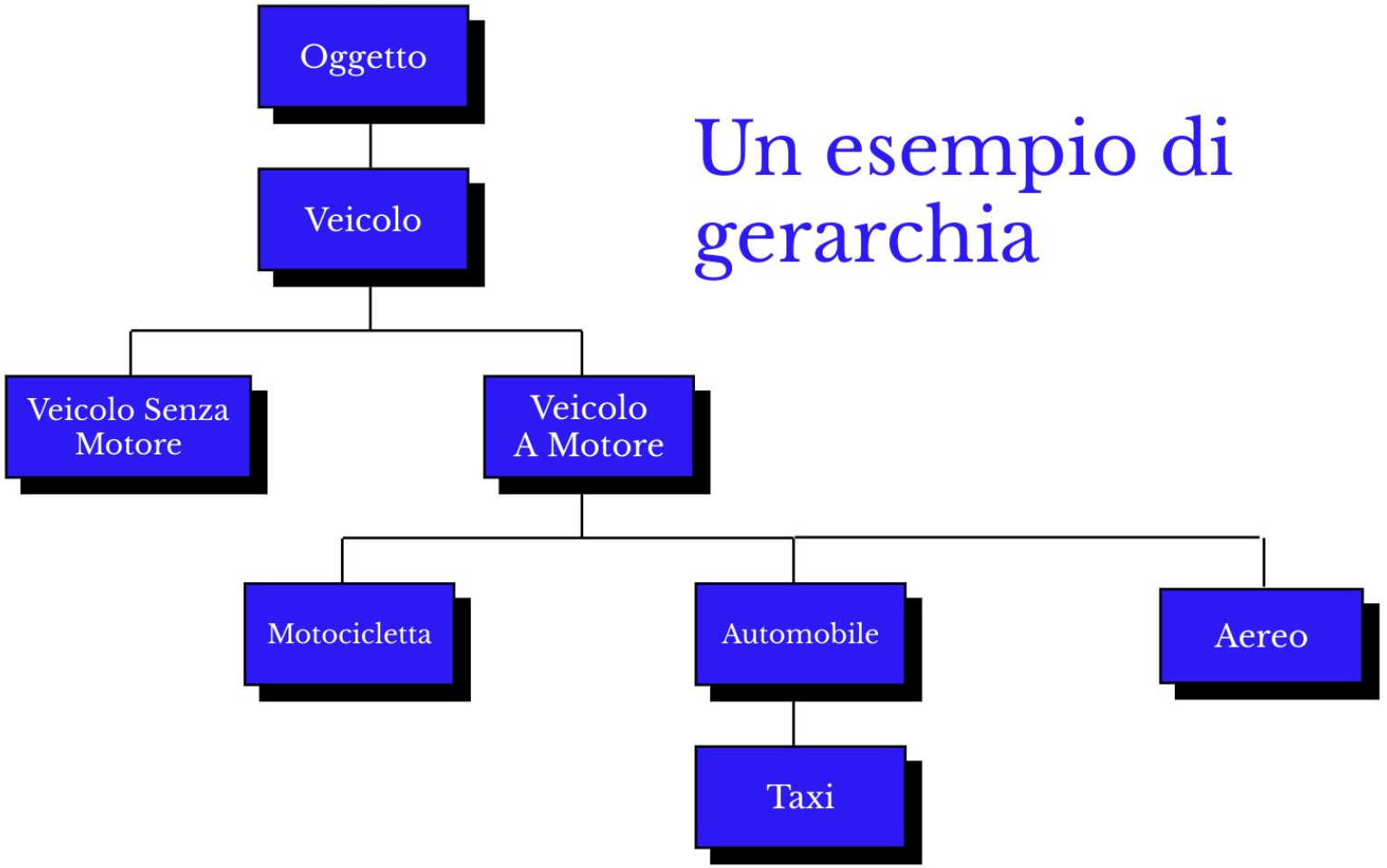
Non è necessario conoscere in dettaglio il funzionamento del codice da riutilizzare, ma è sufficiente modificare (mediante aggiunta o specializzazione) la parte di interesse

Ereditarietà

Generalizzazione: dal particolare al generale
Specializzazione o particolarizzazione: dal particolare al generale



Ereditarietà



Un esempio di gerarchia

Ereditarietà

L’ ereditarietà è una tecnica che permette di descrivere una nuova classe, nei termini di un'altra già esistente

Essa consente inoltre di compiere le modifiche o estensioni necessarie.

La sottoclasse eredita tutti i metodi (operazioni) della genitrice, e può essere estesa con metodi ed attributi locali che ne completano la descrizione (**specializzazione** e **generalizzazione**)

Tale meccanismo consente di derivare una sottoclasse da una classe data per aggiunta, per occultamento o per ridefinizione di uno o più membri rispetto alla classe di partenza (che diventa una **superclasse** della nuova classe)

Specializzazione della classe derivata

La classe ereditata è detta **classe base**

La classe che eredita è detta **classe derivata**

La classe che eredità può:

- Utilizzare i metodi della classe ereditata
- Modificare i metodi della classe ereditata
- Definire nuovi metodi propri

Esempio

```
class Base {
  public void stampa (int i) {
   System.out.println(“Stampa”+i);}
}

public class Derivata extends Base {
  public void leggi(int i) {
   System.ou.println(“Leggi”+i);}

public stati void main(String args[]){
       Derivata f=new Derivata();
       f.leggi(3);
       f.stampa(5);
   }
}
```

Esempio: classe base

```
class Struccante {
  private String s = new String(“Detersivo");
  public String toString() { return s; }

  public void append(String a) { s += a; }

  public void dilute() { append(" dilute()"); }
  public void apply() { append(" apply()"); }
  public void pulisci() { append(" pulisci()"); }

  public static void main(String[] args) {
      Struccante x = new Struccante();
      x.dilute(); x.apply(); x.pulisci();
      System.out.println(x.toString());
  }
}
```

Esempio: classe derivata

```
public class Detergente extends Struccante {
     //aggiunta di un nuovo metodo
     public void spuma() {
          append(" spuma()");
          }

     // Occultare all'esterno
     private void dilute() {}

     // Specializzazione: overriding
     public void pulisci() {
          append(" Detergente.pulisci()");
          }

     public void pulisci-old() {
          // Chiamata del metodo della classe base
          super.pulisci(); //non si tratta di una ricorsione!!
          System.out.println("Puluto");}

    public static void main(String[] args) {
          Detergente x = new Detergente();
          x.apply(); x.spuma();
          x.dilute();  // Solo allinterno della classe detergente
          x.pulisci();
          x.pulisci-old();
          System.out.println(x.toString());
}}
```

Meccanismi dell'ereditarietà

extends indica quale classe si vuole ereditare

- JAVA permette di ereditare una sola classe
- C++ supporta l'ereditarietà multipla

Overriding *(dare precedenza)*

- Si ridefinisce il metodo della classe base specializzandone o sostituendone il comportamento (Esempio pulisci)

Overloading

- In Java l'overload di un metodo della classe base, effettuato nella classe derivata, non causa oscuramento

Shadowing

- Oscurare un metodo o un attributo della classe madre dichiarandolo privato

Sintassi JAVA

super

- Punta alla classe base

protected

- Nuovo specificatore di accesso. Indica che l'accesso a quell'attributo è consentito a tutte le classi del package ed a quelle figlie.

Overloading: Esempio

```
class BaseNonno {
  public void stampa ()
      { System.out.println("stampa()");}
}

class BasePadre extends BaseNonno {
  public void stampa (int i)
      { System.out.println("stampa(int)");}
}

public class Derivata extends BasePadre {
  public void stampa (float i)
      {   System.out.println("stampa(float)");}

    public static void main(String args[]) {
        Derivata f = new Derivata();
        f.stampa();
        f.stampa((float)3.5);
        f.stampa(5);
    }
}
```

Ereditarietà e Costruttori

L'ereditarietà non fornisce alla classe derivata solo una interfaccia uguale a quello della classe base.

Quando si crea una classe derivata essa contiene tutte le componenti della classe base.

Quindi anche tutte queste componenti devono essere inizializzate correttamente.

L'unico modo per garantire ciò correttamente è chiamare il costruttore della classe base *(il quale è in grado di effettuare le inizializzazioni correttamente)*.

Java inserisce automaticamente la chiamata al costruttore della classe base in quello della classe derivata

Esempio

```
class Art {
    Art() {System.out.println("Art constructor");}
}

class Drawing extends Art {
    Drawing() {System.out.println("Drawing
constructor");}
}

public class Cartoon extends Drawing {
    public Cartoon()
          {System.out.println("Cartoon constructor");}

    public static void main(String[] args)
          {Cartoon x = new Cartoon();}
}
```

Esempio:output

```
Art constructor
Drawing constructor
Cartoon constructor
```

Costruttori con argomenti

Una classe base può presentare diversi costruttori che si distinguono per numero, tipo o ordine dei parametri

In tal caso è il programmatore che deve scegliere quale costrutto della classe base chiamare

Si utilizza la parola chiave **super** per richiamare il costruttore desiderato

Esempio

```
class Game {
      Game(int i) { System.out.println("Game constructor“+i);}
      Game(flot j) { System.out.println("Game constructor“+j);}
}

class BoardGame extends Game {
  BoardGame(int i) {
     super(i);
     System.out.println("BoardGame constructor“+i);
}}

public class Chess extends BoardGame {
   Chess() {
      super(11);
      System.out.println("Chess constructor“);
    }

    public static void main(String[] args) {Chess x = new Chess();}
}
```

Sviluppo incrementale

Occorre notare che:

- È possibile sviluppare un sistema di oggetti in modo incrementale
- Ogni nuova classe completa quella ereditata aggiungendo delle caratteristiche e senza modificare il codice della classe base
- Se la classe ereditata funzionava correttamente siamo sicuri che un errore non può che trovarsi nel nuovo codice
- Siamo sicuri che un errore non pregiudica il funzionamento della classe ereditata

Considerazoni

Ereditarietà: Costruttori e distruttori

I costruttori sono metodi speciali

Costruzioni di classi derivate:

- Viene chiamato ricorsivamente il costruttore della classe base
- Si comincia la costruzione dalla radice della gerarchia
- Si istanziano gli oggetti creati nella parte dichiarativa
- Si eseguono i costruttori

La distruzione avviene al contrario

- Occorre distruggere prima gli elementi della classe derivata

Esempio

```
class Meal { Meal() { System.out.println("Meal()"); } }
class Bread { Bread() { System.out.println("Bread()"); } }
class Cheese { Cheese() { System.out.println("Cheese()"); } }
class Lettuce { Lettuce() { System.out.println("Lettuce()"); } }

class Lunch extends Meal { Lunch() { System.out.println("Lunch()"); }
}

class PortableLunch extends Lunch {
    PortableLunch() { System.out.println("PortableLunch()");} }

public class Sandwich extends PortableLunch {
  private Bread b = new Bread();
  private Cheese c = new Cheese();
  private Lettuce l = new Lettuce();
  public Sandwich() { System.out.println("Sandwich()"); }

  public static void main(String[] args) {new Sandwich();}
}
```

Esempio:output

```
"Meal()"
"Lunch()"
"PortableLunch()"
"Bread()“
"Cheese()"
"Lettuce()"
"Sandwich()"
```

La classe java.lang.Object

In Java:

- Gerarchia di ereditarietà semplice
- Ogni classe ha una sola super-classe

Se non viene definita esplicitamente una super-classe, il compilatore usa la classe predefinita **Object**

- Object non ha super-classe!

Metodi di Object

Object definisce un certo numero di **metodi pubblici**

- Qualunque oggetto di qualsiasi classe li eredita
- La loro implementazione base è spesso minimale
- La tecnica del polimorfismo permette di ridefinirli

public boolean **equals**(Object o)

- Restituisce “vero” se l’oggetto confrontato è identico (ha lo stesso riferimento) a quello su cui viene invocato il metodo
- Per funzionare correttamente, ogni sottoclasse deve fornire la propria implementazione polimorfica

Metodi di Object

public String **toString**()

- Restituisce una rappresentazione stampabile dell'oggetto
- L'implementazione base fornita indica il nome della classe seguita dal riferimento relativo all'oggetto (java.lang.Object@10878cd)

public int **hashCode**()

- Restituisce un valore intero legato al contenuto dell'oggetto
- Se i dati nell'oggetto cambiano, deve restituire un valore differente
- Oggetti "uguali" **devono** restituire lo stesso valore, oggetti diversi **possono** restituire valori diversi
- Utilizzato per realizzare tabelle hash

Riepilogo visibilità

I seguenti specificatori indicano quando l'attributo/metodo a cui si riferiscono è utilizzabile e da chi:

Accesso privato	si può accedere solo dall'ambito della stessa classe
Accesso di default	… dello stesso package
Accesso protetto	... delle sottoclassi e dello stesso package
Accesso pubblico	Completamente disponibile per qualsiasi altra classe che voglia farne uso

Esercizi

- Si definisca una classe Dipendente i cui oggetti rappresentano le schede dei dipendenti di un'azienda. Si derivi questa classe dalla classe Persona nel Listato 10.1. Un dipendente eredita il nome dalla classe Persona. In aggiunta, un dipendente possiede una retribuzione annuale rappresentata come un valore di tipo double, una data di assunzione che fornisce l'anno di assunzione come un valore di tipo int e un numero identificativo che è un valore di tipo String. Si definiscano gli appropriati costruttori, i metodi get e set e un metodo equals. Si scriva un programma per verificare la definizione della classe.
- Si definisca una classe Dottore i cui oggetti rappresentano le schede dei dottori di una clinica. Si derivi questa classe dalla classe Persona fornita nel Listato 10.1. Un dottore ha un nome, definito nella classe Persona, una specializzazione descritta tramite una stringa (per esempio Pediatra, Ostetrico, Medico generale e così via) e una parcella per le visite in ufficio (si usi il tipo double). Si definiscano gli appropriati costruttori, i metodi get e un metodo equals. Si scriva un programma di prova per verificare tutti i metodi.

Esercizi

- Si crei una classe base Veicolo che possiede il nome della casa automobilistica produttrice (di tipo String), il numero di cilindri del motore (di tipo int) e il proprietario (di tipo Persona fornita nel Listato 10.1). Si crei, quindi, una classe chiamata Camion che è derivata dalla classe Veicolo e possiede delle caratteristiche addizionali: la capacità di carico in tonnellate (di tipo double dal momento che può contenere cifre decimali) e la capacità di carico del rimorchio (di tipo double). Si dotino le classi di costruttori opportuni, di tutti i metodi get e del metodo equals. Si scriva un programma driver per verificare il funzionamento dei metodi definiti.
- Si crei una nuova classe Cane che sia derivata dalla classe Animale fornita nel Listato 9.1 del Capitolo 9. La nuova classe avrà gli attributi aggiuntivi razza (tipo String) e comandoDiRichiamo (tipo boolean), il quale sarà vero se l'animale ha il suo comando di richiamo e falso altrimenti. Si dotino le classi di opportuni costruttori e di tutti i metodi get. Si scriva un programma driver per verificare tutti i metodi, poi si scriva un programma che legga le informazioni per cinque animali di tipo Cane e visualizzi il nome e la razza di tutti gli animali che siano oltre i due anni di età e non abbiano assegnati i loro comandi di richiamo.