Fondamenti di Java

Il linguaggio cult per la programmazione a oggetti in quattro lezioni

La nascita, la metodologia, lo sviluppo e le caratteristiche uniche che hanno fatto di Java il linguaggio più usato in rete

Principi fondamentali di OOP

- La OOP si basa su tre principi fondamentali:
  - L'incapsulamento
  - L'ereditarietà
  - Il polimorfismo
- L'incapsulamento consiste nel racchiudere tutto il codice all'interno delle classi,inserendo in questi moduli i dati e il codice che opera su di essi

L'ereditarietà

- Immaginiamo di avere già definito una classe Persona e di voler ora definire una classe Studente.
- Uno studente ha tutte le caratteristiche e i comportamenti di una persona e in più ha altre caratteristiche e altri comportamenti specifici della sua condizione.
- Ad esempio, oltre a nome ed età, lo studente ha le gli attributi facoltà e numero esami sostenuti e, oltre a presentarsi è in grado di sostenere esami universitari.
- Sarebbe utile non dover riscrivere la classe Studente a partire dal nulla, ma basandosi sui contenuti della classe Persona ed aggiungendo a questi gli elementi necessari a identificare una persona di tipo studente

Definizione di Ereditarietà

- Nella programmazione *object oriented* si creano moduli software indipendenti proprio per poterli utilizzare in diverse occasioni componendoli in modo differente.
- Spesso dunque si ha la necessità di sviluppare delle classi che siano la specializzazione o il potenziamento di classi già esistenti.
- Questo principio, noto come ereditarietà, è uno dei postulati fondamentali della programmazione a oggetti e consiste nel definire nuove classi a partire da quelle esistenti operando per differenza.
- Non è necessario ridefinire nella nuova classe anche variabili e metodi d'istanza già definiti: la nuova classe può *ereditarli* da una classe preesistente.
- La nuova classe, definita classe *figlia* o *sottoclasse*, eredita tutti gli attributi e i metodi non privati della classe *madre* o *superclasse*.

La parola chiave extends

- La parola chiave che permette la creazione di una sottoclasse in Java è *extends*, che si inserisce nella dichiarazione della sottoclasse ed è seguito dal nome della classe madre da estendere.
- Per dichiarare una classe Studente derivata dalla classe Persona si scriverà quindi:

  Class Studente extends Persona
- Avendo dichiarato Studente come estensione di Persona, non sarà necessario ridichiarare in essa gli attributi e i metodi comuni

Classe Persona

```
import javax.swing.JOptionPane;

public class Persona {
    String nome;
    int n;

    Persona(String nome, int n){
       this.nome=nome;
       this.n=n;}

    void presenta(){
      System.out.println("Buongiorno, mi chiamo "+nome+" e ho "+n+" anni");}

    void presentaVideo(){
      JOptionPane.showMessageDialog(null, "Buongiorno, mi chiamo "+ nome +
     "e ho "+n+" anni");}}
```

La classe Studente

```
import javax.swing.JOptionPane;

public class Studente extends Persona{

    String laurea;
    int numeroEsami;

    Studente(String nome, int n, String laurea, int numeroEsami){
          super(nome,n);
          this.laurea=laurea;
          this.numeroEsami=numeroEsami;}

    void presentaVideo(){
        JOptionPane.showMessageDialog(null,"Salve, mi chiamo "+nome+" e frequento "+laurea);}

    void esame(String nomeEsame){
        int voto=Integer.parseInt(JOptionPane.showInputDialog("Inserisci voto"));
        if(voto>17){
            JOptionPane.showMessageDialog(null,"Esame di "+ nomeEsame+" superato con voto: "+voto);
            numeroEsami++;}
       else
          JOptionPane.showMessageDialog(null,"esame non superato");}}
```

Classe ProvaPersona

```
public class ProvaPersona {

    public static void main(String[] args) {
       Persona p=new Persona("Marco Bianchi",30);
       p.presenta();

     Studente s=new Studente("Mario
    Rossi",19,"ingegneria",4);
       s.presenta();
       s.presentaVideo();
       s.esame("analisi 1");
    }
}
```

Ereditarietà e polimorfismo

- Nell'esempio precedente nella classe Studente sono dichiarati solo gli attributi e i metodi non compresi nella classe Persona.
- Gli attributi nome e n e il metodo presenta() possono essere utilizzati dagli oggetti di classe Studente perché ereditati da Persona
- Il metodo presentaVideo() è stato ereditato, ma anche ridefinito nella classe Studente, quindi quando viene invocato si presenta con le caratteristiche proprie della sottoclasse
- Nel costruttore Studente() è possibile fare riferimento al costruttore della classe Persona usando la parola chiave super,che consente di accedere ad attributi e metodi non privati della superclasse, si devono pertanto assegnare solo gli attributi specifici.

Polimorfismo, overriding e overloading

- Si definisce il polimorfismo come la possibilità di scegliere, all'interno di una classe generale di azioni, un'azione specifica in base alle condizioni di evoluzione del programma.
- Aspetti del polimorfismo: *overloading* e *overridding*
- *overloading:* possibilità di definire in una classe, più metodi con lo stesso nome, purché distinguibili per il numero o il tipo dei parametri (*firma* del metodo)
- *overriding:* possibilità per le classi figlie di ridefinire uno o più metodi ereditati dalla classe madre, conservandone il nome, il tipo e la firma, ma fornendo un'implementazione diversa.

Nozioni importanti sull'ereditarietà

- Da una classe madre si possono derivare più classi figlie con caratteristiche diverse
- da una sottoclasse è generalmente possibile derivare un'altra sottoclasse.
- Se si vuole impedire che da una classe si possano derivare delle sottoclassi, questa deve essere dichiarata con la clausola *final*.
- la clausola *final* posta davanti a un metodo impedisce che il metodo in questione possa essere ridefinito in una sottoclasse

Information hiding

Attributi e metodi possono essere definiti:

- *public*: se ogni altre classe può accedervi e possono essere ereditati da eventuali sottoclassi
- *protected*: se l'accesso è consentito alle classe stessa in cui sono definiti e a tutte eventuali le sottoclassi da questa derivate
- *private*: se l'accesso è ristretto alla classe o al file in cui la classe è dichiarata

Un esempio di ereditarietà

- Esaminiamo una classe Utente che contiene le informazioni relative agli utenti di un sito:

```
import javax.swing.JOptionPane;
public class Utente {
    public String id;
    public String psw;
    Utente(String a, String b){
        id=a;
        psw=b;}

    public void messaggio(){
        JOptionPane.showMessageDialog(null,"Benvenuto
        "+id);}
}
```

Un esempio di ereditarietà

- La classe Socio è derivata dalla classe Utente e descrive gli iscritti che accedono al forum tramite nickname

```
import javax.swing.JOptionPane;
public class Socio extends Utente{
    String nick;
    String commento;
    Socio(String a, String b, String n){
        super(a,b);
        nick=n;
        commento="";}
    public String invia(){
        commento=JOptionPane.showInputDialog("invia commento");
        return commento;}
}
```

Note all'esempio precedente

- La classe Socio eredita dalla classe Utente gli attribuiti *id* e *psw* E il metodo *messaggio()*, che non viene ridefinito.
- Quando si invoca il metodo *messaggio()* su un oggetto di tipo Socio viene eseguito il codice descritto nella superclasse.
- Nel costruttore della sottoclasse è possibile ricorrere al costruttore della classe madre per istanziare gli attributi comuni. Per riferirsi al costruttore della classe madre si utilizza la parola chiave *super*.
- La parola *super* si userà ogni volta in cui è necessario invocare metodi e attributi della classe madre nelle sottoclassi

Un esempio di overloading

```
import javax.swing.JOptionPane;

public class Prodotto {
	String desc;
	double prezzo;
	Prodotto(String d, double p){
		desc=d;
		prezzo=p;
	}
	public void visualizza(){
		JOptionPane.showMessageDialog(null, desc+" "+ prezzo);
	}
	public void aggiorna(String d){
		desc=d;
	}
	public void aggiorna(double p){
		prezzo=prezzo+p;
	}
	public void aggiorna(String d, double p){
		desc=d;
		prezzo=prezzo+p;
	}
}
```

Un esempio di overriding

- Consideriamo un'altra classe derivata da Utente, la classe Cliente. Descrive gli utenti del sito che fanno acquisti on-line:

```
public class Cliente extends Utente{
        public String mail;
        public double saldo;
        Cliente(String a, String b, String e, double f){
                super(a,b);
                mail=e;
                saldo=f;}
        public void messaggio(){
                JOptionPane.showMessageDialog(null, "Benvenuto "+id+" il tuo saldo  è "+
                saldo);}
        public void acquista(Prodotto ac){
                if(saldo>=ac.prezzo){
                                saldo=saldo-ac.prezzo;
                                JOptionPane.showMessageDialog(null, "Acquisto avvenuto, il tuo
                                saldo è "+ saldo);}
                else
                                JOptionPane.showMessageDialog(null, "Credito insufficiente");}}
```

Note all'esempio precedente

- Nella classe Cliente non è necessario dichiarare gli attributi *id* e *psw* che sono stati ereditati dalla classe madre, ma solo i nuovi attributi e i nuovi metodi che sono specifici della classe Cliente.

- La classe Cliente implementa l'*overriding:* eredita dalla classe Utente il metodo *messaggio()*, ma lo ridefinisce per comunicare il saldo attuale del cliente

- La classe Cliente definisce un metodo *acquista(),* che necessita in input di un parametro di tipo Prodotto che indica il prodotto da acquistare.

Ereditarietà multipla

- L'ereditarietà multipla consiste nel derivare una classe figlia estendendo due o più classi genitore.
- Tale sottoclasse eredita tutti i metodi e tutti gli attributi non privati delle superclassi.
- Tuttavia, questa seconda soluzione pone numerosi problemi implementativi
- Se le classi genitore possiedano uno stesso metodo e lo implementano in modi diversi e se il metodo in questione non viene ridefinito nella sottoclasse in fase di esecuzione non è possibile determinare quale implementazione deve essere eseguita.
- Per scelta, Java permette la sola ereditarietà singola.

Interfacce in Java

- L'ereditarietà multipla viene recuperata attraverso l'introduzione delle interfacce.
- Interfacce: classi astratte costituite di soli metodi, in cui i metodi stessi vengono dichiarati, ma non implementati.
- Per utilizzare il metodo di un'interfaccia è indispensabile estendere l'interfaccia stessa, fornendo l’implementazione del metodo nella sottoclasse
- Una classe può implementare più interfacce