Metodologie di Programmazione

Lezione 13: Ereditarietà (parte 1)

Lezione 13: Sommario

UNITELMA SAPIENZA
SAPIENZA UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

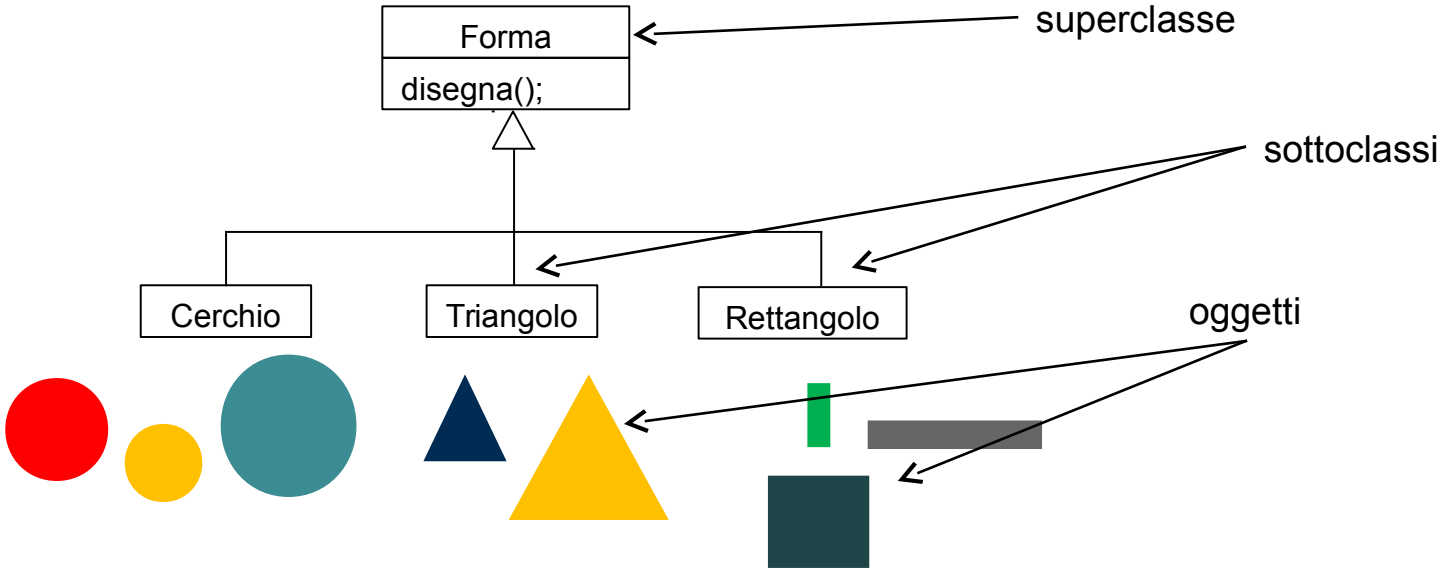
- Introduzione all'ereditarietà
- Classi e metodi astratti
- Le parole chiave this e super nei costruttori
- Overloading e overriding

Parliamo di ereditarietà

- Un concetto cardine della programmazione orientata agli oggetti
- Una forma di riuso del software in cui una classe è creata:
  - "assorbendo" i membri di una classe esistente
  - aggiungendo nuove caratteristiche o migliorando quelle esistenti
- Programmazione mattone su mattone
  - Detto anche: non si butta via niente
- Aumenta le probabilità che il sistema sia implementato e manutenuto in maniera efficiente

Molti tipi di “forma”

- Si può progettare una classe Forma che rappresenta una forma generica e poi specializzarla estendendo la classe



Le forme Triangolo e Cerchio estendono la classe Forma


```
public class Forma
{
    private Color colore;

    /**
     * Questo metodo non dovrebbe avere implementazione vuota!
     */
    public void disegna() { }
    public Color getColore() { return colore; }
}
```

Estende la classe Forma

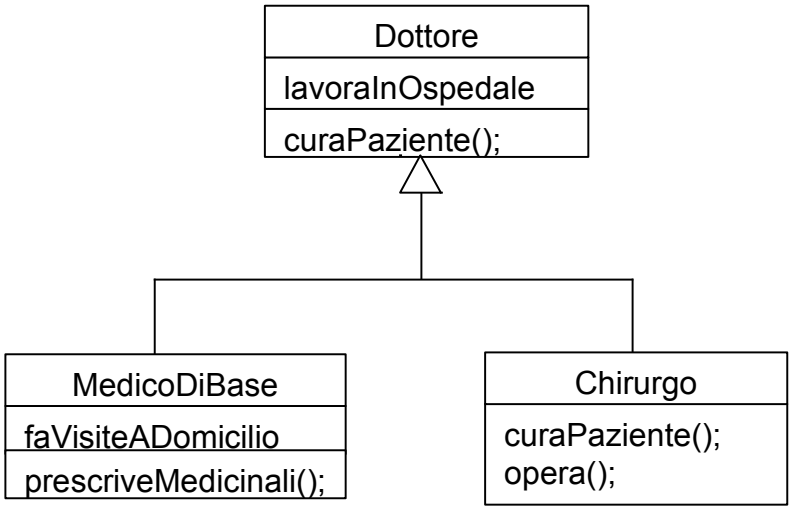
```
public class Triangolo extends Forma
{
    private double base;
    private double altezza;

    public Triangolo(double base, double altezza)
    {
        this.base = base;
        this.altezza = altezza;
    }

    public double getBase()
    {
        return base;
    }

    public double getAltezza()
    {
        return altezza;
    }
}
```

```
public class Cerchio extends Forma
{
    /**
     * Raggio del cerchio
     */
    private double raggio;

    public Cerchio(int raggio)
    {
        this.raggio = raggio;
    }

    public double getRaggio() { return raggio; }
    public double getCirconferenza() { return 2*Math.PI*raggio; }
}
```

Ereditarietà: che cosa si eredita?

- Una sottoclasse estende la superclasse
- La sottoclasse eredita i membri della superclasse
  - campi e metodi d'istanza secondo il livello di accesso specificato
- Inoltre la sottoclasse può:
  - aggiungere nuovi metodi e campi
  - ridefinire i metodi che eredita dalla superclasse (tipicamente **NON** i campi)

Esempio: il dottore, il medico di base e il chirurgo



Esempio: Paperino & company

Classi astratte

- Una classe astratta (definita mediante la parola chiave abstract) non può essere istanziata
- Quindi **NON** possono esistere oggetti per quella classe

```
/**
 * Classe astratta: non e' possibile istanziarla
 */
public abstract class PersonaggioDisney
{



}
```

- Tipicamente verrà estesa da altre classi, che invece potranno essere istanziate

```
public class Paperoga extends PersonaggioDisney
{



}
```

Metodi astratti

- Anche i metodi possono essere definiti astratti
  - Esclusivamente all'interno di una classe dichiarata astratta
- **NON** forniscono l'implementazione per quel metodo

```
/**
 * Classe astratta: non e' possibile istanziarla
 */
public abstract class PersonaggioDisney
{
    /**
     * Metodo astratto senza implementazione
     */
    abstract public void faPasticci();
}
```

- Impongono alle sottoclassi non astratte di implement

```
public class Paperoga extends PersonaggioDisney
{
    public void faPasticci()
    {
        System.out.println("bla bla bla bla bla bla bla");
    }
}
```

Un esempio: Impossible Mission

- Abbiamo tanti "oggetti" (in senso lato)
  - Piattaforme
  - Computer
  - Oggetti in cui cercare indizi
- Alcuni sono "personaggi"
  - Il giocatore
  - I robot
  - La bomba
- Che cosa hanno in comune tutti?
- E che cosa li distingue?

Modellare Impossible Mission: la classe “astratta” Entità

- Abbiamo bisogno di una classe molto generale (quindi astratta) che rappresenti oggetti mobili e immobili nel gioco:

```
package it.uniroma1.impmiss;

abstract public class Entita
{
    protected int x;
    protected int y;

    public Entita(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public int getX() { return x; }
    public int getY() { return y; }
}
```

Campi a **visibilità protetta**

- La visibilità protetta (protected) rende visibile il campo (o il metodo) a tutte le sottoclassi
- Ma anche a tutte le classi del package (!)

Modellare Impossible Mission: gli oggetti

- Modelliamo gli oggetti immobili in astratto:

```
package it.uniroma1.impmiss;

abstract public class Oggetto extends Entita
{
    private TesseraPuzzle tessera;

    public Oggetto(int x, int y)
    {
        this(x, y, null);
    }

    public Oggetto(int x, int y, TesseraPuzzle tessera)
    {
        super(x, y);
        this.tessera = tessera;
    }

    public TesseraPuzzle search() { return tessera; }
}
```

Doppio costruttore (overloading)

Riuso del codice chiamando l'altro costruttore mediante la parola chiave this

Richiama il costruttore della superclasse (**OBBLIGATORIO** perché ha almeno un parametro) con la parola chiave super

Metodo aggiuntivo

- contenere una tessera del puzzle del

```
package it.uniroma1.impmiss;

public class TesseraPuzzle
{
    // da implementare
}
```

Parole chiave this e super

- La parola chiave this usata come nome di metodo **obbligatoriamente** nella **prima riga del costruttore** permette di richiamare un altro costruttore della stessa classe
- La parola chiave super usata come nome di metodo **obbligatoriamente** nella prima riga del costruttore permette di richiamare un costruttore della superclasse
- Ogni sottoclasse deve esplicitamente definire un costruttore se la superclasse **NON** fornisce un costruttore senza argomenti
  - Questo avviene se si definisce un costruttore **con almeno un argomento** ma non si definisce il costruttore senza argomenti
- **Nota bene:** anche le classi astratte possono avere costruttori! **Perché?**

Costruzione incrementale dell'oggetto

- Ogni classe sa come "costruire" e inizializzare lo stato dell'oggetto
- E' **da evitare** che una sottoclasse inizializzi **direttamente** (e non tramite il costruttore della superclasse) i campi della superclasse
- Mattone su mattone:
  - Prima si inizializza lo stato specificato nella superclasse
  - Poi si inizializza la parte aggiuntiva specificata nella sottoclasse

Un esempio di dinamica delle chiamate a costruttori

- Definiamo tre classi Z -> Y -> X:

```java
public class X
{
    public X(int k)
    {
        System.out.println("X(int k)");
    }

    public X()
    {
        System.out.println("X()");
    }
}
```

```java
public class Y extends X
{
    public Y(int k)
    {
        System.out.println("Y(int k)");
    }
}
```

```java
public class Z extends Y
{
    public Z(int k)
    {
        super(k);
        System.out.println("Z(int k)");
    }

    public Z()
    {
        this(0);
        System.out.println("Z()");
    }

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

- La sequenza delle chiamate è la seguente:

```
X()
Y(int k)
Z(int k)
Z()
```

Costruttori della superclasse (X) e della sottoclasse (Y)


```
public class X
{
}

public class Y extends X
{
}
```

Costruttori di default in X e Y

```
public class X
{
      public X()
      {

      }
}

public class Y extends X
{
}
```

Costruttore esplicito con zero argomenti per X e di default per Y (quello di Y chiama automaticamente quello di X)

```
public class X
{
}

public class Y extends X
{
      public Y(int k)
      {

      }
}
```

Costruttore di default in X e con un argomento in Y (chiama in automatico il costruttore di default di X)

Costruttori della superclasse (X) e della sottoclasse (Y)


```
public class X
{
        public X(int k)
        {

        }
}

public class Y extends X
{
}
```

Costruttore con un parametro per X e di default per Y

```
public class X
{
        public X()
        {

        }

        public X(int k)
        {

        }
}

public class Y extends X
{
}
```

Costruttori con zero e un parametro per X e di default per Y (quello di Y chiama automaticamente quello di X)

Costruttori della superclasse (X) e della sottoclasse (Y)

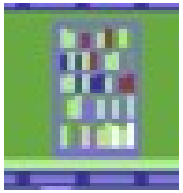

```
public class X
{
    private int x;
    private int y;

    public X(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

public class Y extends X
{
    public final static int X = 10, Y = 20;

    public Y()
    {
        super(X, Y);
    }
}
```

Costruttori con più parametri per X e con zero parametri per Y (quello di Y chiama quello di X con i parametri giusti)

Modellare Impossible Mission: un oggetto e il computer

- Modelliamo un possibile oggetto immobile:

```
package it.uniroma1.impmiss;

public class Libreria extends Oggetto
{
    public Libreria(int x, int y, TesseraPuzzle tessera)
    {
        super(x, y, tessera);
    }

    public Libreria(int x, int y)
    {
        super(x, y);
    }
}
```



-

```
package it.uniroma1.impmiss;

public class Computer extends Entita
{
    public Computer(int x, int y)
    {
        super(x, y);
    }

    public void login() { /* da implementare */ }
    public void logout() { /* da implementare */ }
}
```

Metodi aggiuntivi

Modellare Impossible Mission: entità mobili

- Modelliamo un generico personaggio:

```
package it.uniroma1.impmiss;

abstract public class Personaggio extends Entita
{
    public enum Direzione
    {
        DESTRA,
        SINISTRA,
        ALTO,
        BASSO;
    }

    private String nome;
    private int velocita;

    public Personaggio(int x, int y, String nome, int velocita)
    {
        super(x, y);
        this.nome = nome;
        this.velocita = velocita;
    }

    public String getNome() { return nome; }
    public int getVelocita() { return velocita; }

    public void muoviti(Direzione d)
    {
        switch(d)
        {
            case DESTRA: x += velocita; break;
            case SINISTRA: x -= velocita; break;
            // in futuro: emetti eccezione!
            default: System.out.println("Direzione non ammessa"); break;
        }
    }
}
```

Enumerazione delle direzioni

Campi aggiuntivi

Si possono fornire **implementazioni** nella classe astratta!

Metodi aggiuntivi

Modellare Impossible Mission: il giocatore

- Modelliamo il giocatore (ovvero la spia):

```
package it.uniroma1.impmiss;

public class Spia extends Personaggio
{
    public Spia(int x, int y, String nome, int velocita)
    {
        super(x, y, nome, velocita);
    }

    public void salta()
    {
        // ...
    }
}
```

Modellare Impossible Mission: nemico e robot

- Modelliamo un generico nemico:

```
package it.uniroma1.impmiss;

abstract public class Nemico extends Personaggio
{
    public Nemico(int x, int y, String nome, int velocita)
    {
        super(x, y, nome, velocita);
    }

    abstract public void attacca();
}
```

Metodo astratto!

- E i robot:

```
package it.uniroma1.impmiss;

public class Robot extends Nemico
{
    public Robot(int x, int y, String nome, int velocita)
    {
        super(x, y, nome, velocita);
    }

    public void incenerisci() { /* fulmine elettrico */ }

    public void attacca() { /* da implementare */ }
}
```

Nella sottoclasse siamo **obbligati** a definire il metodo astratto

Modellare Impossible Mission: la bomba

- Modelliamo la bomba:

```
package it.uniroma1.impmiss;

public class Bombone extends Nemico
{
    public Bombone(int x, int y, String nome, int velocita)
    {
        super(x, y, nome, velocita);
    }

    public void attacca()
    {
        // da implementare
    }

    public void muoviti(Direzione d)
    {
        switch(d)
        {
            case DESTRA: case SINISTRA: super.muoviti(d); break;
            case ALTO: y -= getVelocita(); break;
            case BASSO: y += getVelocita(); break;
        }
    }

    public void muoviti(Spia p)
    {
        muoviti(p.x > this.x ? Direzione.DESTRA : Direzione.SINISTRA);
        muoviti(p.y > this.y ? Direzione.ALTO : Direzione.BASSO);
    }
}
```

Overriding del metodo

Overloading del metodo

Esempi di riuso del codice

Differenza tra overriding e overloading

- L'overriding consiste nel ridefinire (reimplementare) un metodo con la stessa **intestazione** ("segnatura") presente in una superclasse
- L'overloading consiste nel creare un metodo con lo stesso nome, ma una **intestazione** diversa (diverso numero e/o tipo di parametri)

Mantenere il contratto!

- Nell'overriding gli argomenti devono essere gli stessi
- I tipi di ritorno devono essere compatibili (lo stesso tipo o una sottoclasse)
- Non si può ridurre la visibilità (es. da public a private)

- Nell'overloading i tipi di ritorno possono essere diversi
  - **MA:** non si può cambiare **SOLO** il tipo di ritorno
- Si può variare la visibilità in qualsiasi direzione