

PROGRAMMAZIONE A OGGETTI: I CONCETTI DI BASE

Ancora una volta: Classe != Oggetto.

Rammento ancora: una **classe** è l'insieme di specifiche necessarie a realizzare quella che è un'implementazione concreta della medesima, cioè un **oggetto**.

Una classe è la ricetta per fare un dolce.

Un oggetto è il tiramisù.

Se avete fame, non vi mangiate la ricetta bensì una o più fette di un tiramisù reale.

Il tiramisù che avete appena mangiato occupa un suo spazio nel mondo - fuor di metafora, un oggetto è un'**istanza di una classe**, ed occupa una porzione di memoria fisica del computer ove state lavorando.

Sappiamo già che le classi sono costituite da due grandi categorie di elementi: i **campi**, cioè le loro proprietà ed i **metodi**, cioè le azioni che possiamo operare su di esse.

Sfruttare materiale già scritto: ereditarietà

Supponiamo di dover scrivere un programma per creare dolci: continuando con l'analogia *classe* = *ricetta* e semplificando molto, ogni ricetta è costituita da una lista di ingredienti [*campi*] e da una serie di operazioni [*metodi*].

Vorrei poter eseguire questo codice, indicando le quantità di zucchero, farina e uova:

```
Dolce d = new Dolce(250, 250, 3);  
d.prepara();  
System.out.println("Lascia stare i fornelli e torna al PC.");
```

Una base sensata può essere la seguente - non scrivo i *get()* ed i *set()* per comodità di lettura, ed ometto storie tipo lievito ecc., non me ne vogliate.

```

class Dolce{

    // espresse in grammi per gli ingredienti non numerabili;
    protected int zucchero;
    protected int farina;
    protected int uova;

    // costruttore vuoto
    public Dolce(){}

    // costruttore con le quantità
    public Dolce(int zucchero, int farina, int uova){
        this.setZucchero(zucchero);
        this.setFarina(farina);
        this.setUova(uova);
    }

    // il metodo prepara() esegue operazioni in ordine
    public void prepara(){
        this.mescola();
        this.cuoci();
    }

    // ogni azione fa cose per una certa durata
    protected void mescola(int minuti){

        // qui aggiungiamo gli ingredienti e mscoliamo
    }

    protected void cuoci(int temperatura, int minuti){

        // la metafora inizia a farsi strettta ^^
    }

}

```

Così non possiamo fare granché però, visto che ogni dolce ha ingredienti molto diversi ed operazioni sufficientemente eterogenee da compiere. Lavoro sprecato? Beh no.

Ad esempio, i campi ed i metodi indicati sono comuni a molti dolci, quindi sarebbe tedioso doverli riscrivere in ogni classe - perché non sfruttare quanto già fatto? Presto detto:

```
class Ciambellone extends Dolce{

    // aggiungiamo un nuovo ingrediente
    protected int cacao;

    // costruttore vuoto
    public Ciambellone(){}

    // costruttore con le quantità
    public Ciambellone(int zucchero, int farina, int uova, int
cacao){

        // richiamo il costruttore della superclasse
        super(zucchero, farina, uova);
        this.setCacao(cacao);
    }

    // il metodo prepara() ora ha un'operazione in più
    public void prepara(){

        // eseguo le operazioni definite nella classe Dolce
        super.prepara();
        this.decora();
    }

    // ridefinisco il metodo mescola()
    protected void mescola(){

        // qui facciamo altre cose, tipo aggiungere il cacao
    }

    protected void decora(){

        // abbelliamo il nostro ciambellone con una guarnizione
    }
}
```

Notare le seguenti:

- campi e metodi che un tempo erano privati sono stati definiti **protected** - in tale modo sono visibili anche dalle sottoclassi: Ciambellone “vede” i campi farina, zucchero e uova;
- la keyword **super** accede ai metodi della superclasse. Senza dover riscrivere per intero il metodo **prepara()**, mi limito a richiamare quanto fatto nella superclasse e poi ad aggiungere eventualmente altro.
- la scrittura **super([parametri])** invoca un *costruttore* della superclasse: se si vuole sfruttare tale funzionalità, occorre chiamare tale metodo come prima istruzione del costruttore corrente, pena il farsi redarguire severamente da Java.

Bene, abbiamo *esteso* la classe Dolce con un nuovo campo ed nuovo metodo ed abbiamo ridefinito qualcuno di quelli già presenti.

Passando alla teoria più stretta, è bene ricordare che Java non supporta l'*ereditarietà multipla*, cioè non permette di fare:

```
class Ciambellone extends Dolce, Toro {...}
```

dove per toro non si intende il fiero bovino ovviamente, ma il solido di rotazione. Si può invece implementare ereditarietà a qualsiasi livello di profondità, cioè:

```
class Ciambellone extends Dolce {...}  
class CiambellaSpeciale extends Ciambellone{...}  
...
```

Lecitissimo. In dettaglio *ogni* classe di Java, sia quelle esistenti sia quelle che definiremo noi di volta in volta, estendono implicitamente la superclasse cosmica, cioè **Object**.

Object ha dei metodi di comune utilità quali il ben noto **toString()**, per maggiori dettagli consultare la documentazione a riguardo.

L'ereditarietà vi consente quindi di sfruttare materiale già prodotto, di stabilire vincoli utili tra entità legate, e di risparmiarvi tanto lavoro quando dovrete aggiornare metodi comuni, visto che dovrete farlo una sola volta per ogni legame di ereditarietà. Nonché di preparare buoni dolci come avete visto.

Concetti più generali: le classi astratte.

Riflettiamo un attimo sull'esempio appena considerato: avete mai mangiato un dolce generico? Avrete gustato gelati, torroni, biscotti, sachertorte, millefoglie, meringhe, profiterol - ognuno con la propria identità specifica! Idem avrete visto cani, gatti, cavalli, linci, delfini, balene, elefanti, ornitorinchi, scilipoti, melissasatta - ma sfido chiunque ad aver visto un mammifero che non appartenesse a nessuna specie.

Bene, tali concetti perfettamente leciti ma non istanziabili concretamente in Java sono definiti come **classi astratte**: servono a definire proprietà comuni nella catena ereditaria, rendendo al contempo più robusto e flessibile il processo di modellazione.

Una classe astratta non può essere istanziata, pena l'incorrere in un errore di compilazione. Essa può essere dotata di **metodi astratti**, i quali sono *privi di corpo* e fungono da monito per chiunque voglia estendere la classe cui appartengono: per effettuare con successo tale compito è necessario ridefinire tutti i metodi astratti contenuti nella classe stessa. Un esempio ci sarà d'aiuto - consideriamo la classe **Animale**:

```
abstract class Animale{  
    ...  
    abstract public void sposta();  
}
```

Poniamo per semplicità che tutti gli animali si spostino in qualche modo. Bene, le classi figlie dovranno per forza ridefinire il metodo `sposta()` [notare che esso è privo di *corpo*, cioè privo di graffe!]:

```
class Cane extends Animale{  
    ...  
    public void sposta(){  
        // ridefinizione del metodo  
    }  
}
```

Ragionevole: le classi astratte ci permettono di definire entità generali senza correre il rischio che qualcuno possa inavvertitamente istanziarne qualcuna, preservando così la coerenza del ragionamento prima esposto.

Andare oltre: le interfacce.

L'ereditarietà è sicuramente uno strumento killer in ambito descrittivo: se B estende A, si può facilmente leggere che “B è un A” [ma non viceversa!]. Ma se vogliamo azione, il verbo essere non è di certo il regista più adatto: per rendere le nostre classi degne di Luc Besson, abbiamo bisogno di un ulteriore strumento.

Immaginiamo di voler sfidare in una gara di velocità i partecipanti più disparati, esseri viventi ed artefatti umani compresi: non possiamo di certo creare una classe **Corridore** e pretendere che tutti la estendano, sarebbe poco sensato.

La cosa migliore è quella di lasciare ogni cosa al proprio posto, lasciando pure che **Ghepardo** estenda **Felino** e che **Lamborghini** estenda **Automobile**; basta dotare tutti i concorrenti di una proprietà *trasversale*, contrapposta alla *verticalità* del vincolo ereditario.

Definiamo quindi la seguente interfaccia:

```
public interface Velocista{  
  
    public void corri();  
  
}
```

Essa è formalmente una classe astratta dotata di una collezione di metodi privi di corpo. Tutte le classi che vogliono *implementare* una data interfaccia, devono ridefinire *tutti* i metodi in essa contenuti. Per rendere una qualsiasi classe un **Velocista**, è necessario quindi ridefinire il metodo **corri()**, in maniera coerente con la firma del medesimo [in questo caso il metodo non vuole parametri in ingresso, e non restituisce nulla in quanto void, ma non è assolutamente la regola].

Ora occorre aggiornare tutti i partecipanti di conseguenza:

```
class Ghepardo extends Felino implements Velocista{  
  
    public int posizione = 0;  
  
    ...  
  
    public void corri(){  
  
        this.posizione += 5;  
  
    }  
  
}
```

```

class Lamborghini extends Automobile implements Velocista{
    public int posizione = 0;

    public void corri(){
        this.posizione += 15;
    }
}

```

La gara potrà essere gestita da una classe di questo tipo:

```

class Gara{

    // preparo il vettore dei partecipanti
    private Velocista[] partecipanti;

    // definisco la distanza dell'arrivo in metri
    private int arrivo = 1000;

    public Gara(){

        partecipanti = new Velocista[2];
        partecipanti[0] = new Ghepardo();
        partecipanti[1] = new Lamborghini();

        this.partenza();
    }

    private void partenza(){

        while(true){

            for(int i = 0; i < partecipanti.length; i++){

                partecipanti[i].corri();
                if(partecipanti[i].posizione >= arrivo){

                    //qui gestisco il termine corsa
                }
            }
        }
    }
}

```

...

Abbiamo visto come implementare proprietà trasversali per famiglie di classi.

Importante: non c'è limite al numero di interfacce che una singola classe può implementare!

```
class Rana implements Saltatore, Nuotatore{  
  
    ...  
  
    public void salta(){  
  
    }  
  
    public void nuota(){  
  
    }  
  
}
```

dopo aver opportunamente definito le interfacce necessarie, chiaro.

Chinese boxes: le classi interne.

Pensiamo per un attimo di aver compiuto una rampante scalata sociale, e di esser quindi opulenti borghesi padroni di una villa dotata di un immenso giardino all'italiana: vorremo di certo avere un giardiniere personale deputato al mantenimento della nostra area verde. Potremo di certo chiamarne uno esterno di volta in volta, ma ci accorgeremo presto che sarà più conveniente cercare una persona fidata e dar lui accesso alle nostre risorse:

```
class Villa{  
  
    private Giardino nostroGiardino;  
  
    ...  
  
    class Giardiniere{  
  
        public Giardiniere(){}  
  
        public void mantieni(){  
  
            this.pota( nostroGiardino.getSiepi() );  
  
        }  
  
    }  
  
}
```

La classe **Giardiniere** è dichiarata all'interno della classe **Villa**: essa può quindi accedere ai campi ed ai metodi privati di essa, senza che le venga passato riferimento alcuno. Fuor di metafora, la comodità risiede nella possibilità di dichiarare classi interne per gestire interfacce grafiche ad esempio, in modo da poter operare tramite esse sulla classe esterna senza inondare il codice di passaggi per riferimento.

In sede di compilazione è utile portare la seguente osservazione: normalmente ogni classe di Java viene compilata in un file chiamato **NomeClasse.class**, mentre nel caso di classi interne vedremo dei file del tipo **ClasseEsterna\$ClasseInterna.class**, a sottolineare il vincolo di inclusione tra le due entità.

Per completezza, ricordo che le classi interne possono essere definite anche all'interno di un metodo: in tal caso siamo al cospetto di *classi interne locali*, le quali hanno il privilegio di accedere alle variabili dichiarate all'interno del metodo stesso, ma non vivono oltre il medesimo per ovvie ragioni [quali?].

One step beyond: classi annidate anonime.

La questione delle *anonymous inner classes* risulta sufficientemente criptica al programmatore novizio nonché abbastanza limitante per quello esperto: non per questo ci si deve privare di tale conoscenza, perché risulta particolarmente comoda in contesti di progettazione rapida [leggete: in sede di test].

Il concetto di classe interna è stato appena definito, rammentiamo quello più semplice di *classe anonima*: per essa si intende un'istanza di una data classe che sia stata inizializzata senza dichiarazione di un nome esplicito di variabile, ad esempio:

```
public static void main(String[] args){  
  
    Villa v = new Villa(  
        new Giardino()  
    );  
  
}
```

Poniamo che **Villa** [pur passando un momento poco lieto nel Barça] abbia un costruttore avente come parametro un **Giardino**: se non vi è necessità di creare esplicitamente un'istanza di tale classe allo stesso livello della dichiarazione della **Villa v**, allora possiamo costruire il **Giardino** *direttamente* nella chiamata al costruttore della classe **Villa**. Come vedete tale istanza di **Giardino** non ha nome!

Bene, una classe annidata anonima è una classe interna ad un'altra, che sia stata creata senza avere nome esplicito; per fissare questo concetto bisogna fare un esempio concreto alla classe **ActionListener**, che vedremo essere di utilità fondamentale nell'immediato futuro.

Riassumendo: la classe Cronometro.

Per fissare quanto visto negli ultimi paragrafi, costruiamo assieme una classe che scandisca il tempo tramite l'ausilio della classe **Timer** contenuta nel pacchetto **javax.swing**.

Il **Timer** è un oggetto semplice: vuole solo numero di millisecondi ed un oggetto che implementi l'interfaccia **ActionListener** cui "parlare" ad intervalli pari al numero specificato; per parlare si intende invocare il metodo **actionPerformed()** dell'oggetto passato come riferimento. Tale metodo è l'unico dell'interfaccia considerata, ergo l'implementazione consiste nella sua sola ridefinizione.

```
// importo le classi necessarie
import java.awt.event.*;
import javax.swing.Timer;

// la classe implementa l'interfaccia ActionListener
class Cronometro implements ActionListener{

    private int secondi = 0;
    private Timer t;

    public Cronometro(){

        // il Timer vuole un ActionListener - this!
        t = new Timer(this, 1000);
        t.start();

    }

    // ActionListener ha un solo metodo, lo ridefiniamo così
    public void actionPerformed(ActionEvent e){

        this.secondi ++;

        System.out.println( this.secondi );

    }

}
```

Ci sono altri modi per gestire il problema, elencati per completezza:

- costruire una *classe esterna* che implementi **ActionListener**, istanziarla e passarla come riferimento al **Timer**;
- costruire una *classe interna* che implementi **ActionListener**, istanziarla e passarla come riferimento al **Timer**;
- costruire una *classe annidata anonima* che implementi **ActionListener** direttamente all'interno del costruttore del **Timer**.

Nella speranza che questo materiale vi risulti utile, vi auguro buon lavoro.