CORSO JAVA AESYS
Parte Prima

2022

www.aesystech.it

Programmazione orientata agli oggetti

- I paradigmi della programmazione ad oggetti
- Astrazione
- Riuso
- UML: Diagramma delle classi
- Incapsulamento
- Incapsulamento funzionale
- Il reference this
- Modificatori d’accesso

I paradigmi della programmazione ad oggetti

La programmazione ad oggetti ci permetterà di creare programmi più semplici da evolvere e manutenere, che riusano parti di codice ben definite, e che facilitano l'interazione tra il programmatore e il codice. I vantaggi non si limitano solo a queste caratteristiche e scopriremo nuovi benefici nel prosieguo del corso.

Ciò che caratterizza un linguaggio orientato agli oggetti è il supporto che esso offre ai cosiddetti paradigmi della programmazione ad oggetti:

- ❑ Incapsulamento
- ❑ Ereditarietà
- ❑ Polimorfismo

In effetti si dovrebbero considerare paradigmi della programmazione ad oggetti anche l'astrazione e il riuso. Questi ultimi due sono però considerati secondari rispetto agli altri, non perché meno potenti e utili, ma perché non sono specifici della programmazione orientata agli oggetti. Infatti l'astrazione e il riuso sono concetti che appartengono anche alla programmazione funzionale.

Astrazione

L’astrazione potrebbe definirsi come “l’arte di sapersi concentrare solo sui dettagli veramente essenziali nella descrizione di un’entità”. L’astrazione è un concetto chiarissimo a tutti noi, dal momento che lo utilizziamo in ogni istante della nostra vita. Per esempio, mentre state leggendo questo manuale, vi state concentrando sull’apprenderne correttamente i contenuti, senza badare troppo alla forma, ai colori, allo stile e tutti particolari fisici e teorici che compongono la pagina che state visualizzando (o almeno lo speriamo!). Per formalizzare un discorso altrimenti troppo “astratto”, potremmo parlare di almeno tre livelli di astrazione per quanto riguarda la sua implementazione nella programmazione ad oggetti:

- ❑ Astrazione funzionale
- ❑ Astrazione dei dati
- ❑ Astrazione del sistema

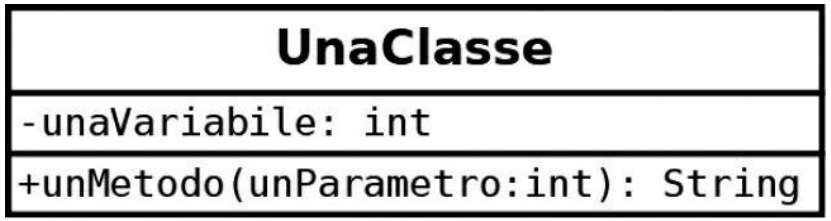
Riuso

Con il termine riuso ci riferiamo principalmente al concetto di riuso che abbiamo nel mondo reale, che riportato nel mondo Java viene tradotto nel riuso di elementi di programmazione come classi e metodi, ma anche di concetti e processi che solitamente utilizziamo quando sviluppiamo un programma. Il riuso è da considerarsi una conseguenza dell'astrazione e degli altri paradigmi della programmazione ad oggetti (incapsulamento, ereditarietà e polimorfismo). Questi infatti favoriscono il riuso intrinsecamente.
Possiamo anche in questo caso parlare di tre livelli di astrazione per quanto riguarda la sua implementazione nella programmazione ad oggetti:

- ❑ Riuso funzionale
- ❑ Riuso dei dati
- ❑ Riuso del sistema

UML: Diagramma delle classi

La notazione di classe in UML è molto intuitiva, un rettangolo diviso in tre sezioni: nella prima viene scritto il nome della classe, nella seconda la lista degli attributi, nella terza l'elenco dei metodi.



UML è un linguaggio di modellazione indipendente dal linguaggio di programmazione che vuole rappresentare. Questo significa che non dobbiamo aspettarci attributi e metodi descritti con la sintassi Java. Come è possibile vedere dalla figura gli attributi vengono definiti con la seguente sintassi UML:

simbolo_di_visibilità nomeAttributo : tipoAttributo

Notare che per i metodi in UML non viene specificato il corpo. Il diagramma delle classi infatti permette la definizione dell'interfaccia pubblica degli oggetti che vogliamo creare, nascondendo l'implementazione interna. Il perché convenga separare l'interfaccia pubblica dall'implementazione interna, è argomento della prossima slide

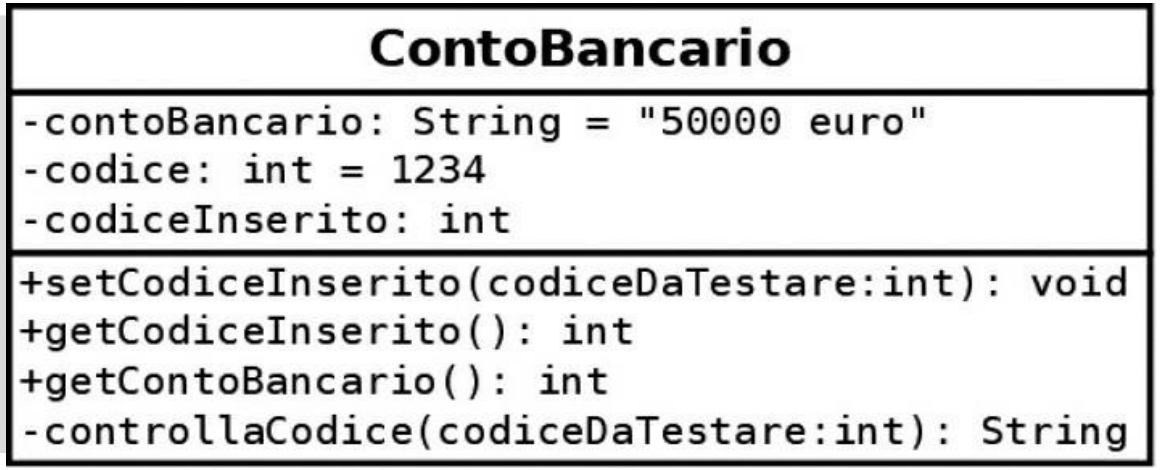
Incapsulamento

L’incapsulamento è la chiave della programmazione orientata agli oggetti. Tramite esso, una classe riesce ad acquisire caratteristiche di robustezza, indipendenza e riusabilità. Inoltre la sua manutenzione risulterà più semplice al programmatore. Una qualsiasi classe è essenzialmente costituita da dati e metodi. La filosofia dell’incapsulamento è semplice. Essa si basa sull’accesso controllato ai dati mediante metodi che possono prevenirne l’usura e la non correttezza. A livello di implementazione, ciò si traduce semplicemente nel dichiarare privati gli attributi di una classe e quindi inaccessibili fuori dalla classe stessa. Per tale scopo introdurremo un nuovo modificatore: private. L’accesso ai dati privati potrà essere fornito da un’interfaccia pubblica costituita da metodi dichiarati public e quindi accessibili da altre classi. In questo modo tali metodi potrebbero ad esempio permettere di realizzare controlli prima di confermare l’accesso ai dati privati. Se l’incapsulamento è gestito in maniera intelligente le nostre classi potranno essere utilizzate nel modo migliore e più a lungo, giacché le modifiche e le revisioni potranno riguardare solamente parti di codice non visibili all’esterno

Incapsulamento funzionale

Utilizzando la keyword private, anche come modificatore di metodi, ottenendo così un incapsulamento funzionale. Un metodo privato infatti, potrà essere invocato solo da un metodo definito nella stessa classe, che potrebbe a sua volta essere dichiarato pubblico. Per esempio la classe ContoBancario definita precedentemente, potrebbe evolversi nel seguente modo:

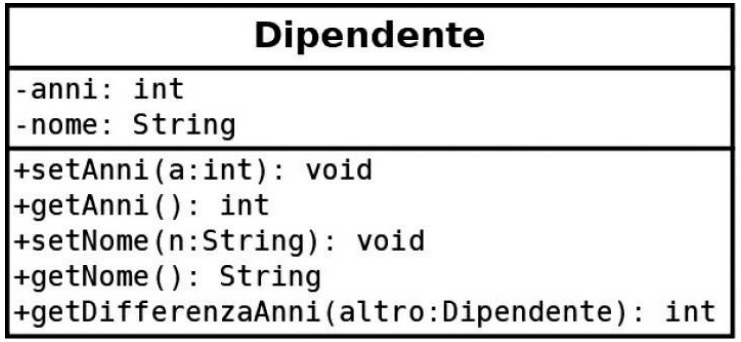
```
public class ContoBancario {
   //...    public String getContoBancario(int codiceDaTestare) {       return
controllaCodice(codiceDaTestare);
   }

   private String controllaCodice(int codiceDaTestare) {       if (codiceInserito ==
codiceDaTestare) {          return contoBancario;
      }       else {          return "codice errato!!!";
      }
   }
}
```



Ciò favorirebbe il riuso di codice in quanto, introducendo nuovi metodi (come probabilmente accadrà in un progetto che viene manutenuto), questi potrebbero riusare (riuso funzionale) il metodo controllaCodice().

Incapsulamento e riuso 1/2

Solitamente si pensa che un membro di una classe dichiarato private diventi “inaccessibile da altre classi”. Questa frase è ragionevole per quanto riguarda l’ambito della compilazione, dove la dichiarazione delle classi è il problema da superare. Ma se ci spostiamo nell’ambito della Java Virtual Machine dove, come abbiamo detto i protagonisti assoluti non sono le classi ma gli oggetti, dobbiamo rivalutare l’affermazione precedente. L’incapsulamento infatti permetterà a due oggetti istanziati dalla stessa classe di accedere in “modo pubblico” ai rispettivi membri privati. Consideriamo la classe Dipendente rappresentata in UML



L’implementazione in Java potrebbe essere la seguente: (vedi slide seguente)

Incapsulamento e riuso 2/2

Nel metodo getDifferenzaAnni() notiamo che è possibile accedere direttamente alla variabile privata anni dell'oggetto altro, senza dover utilizzare il metodo getAnni().
Il codice precedente è quindi valido per la compilazione, ma il seguente metodo:
public int getDifferenzaAnni(Dipendente altro) {
return (getAnni() – altro.getAnni()); }

favorirebbe sicuramente di più il riuso di codice, e quindi è da considerarsi preferibile. Infatti, il metodo getAnni() si potrebbe evolvere introducendo controlli (magari per la sostituzione della variabile anni con la variabile dataDiNascita), che conviene richiamare piuttosto che riscrivere

```
public class Dipendente {  private String nome;  private int anni;
//intendiamo età in anni
  //...  public String getNome() {
return nome;
  }
  public void setNome(String n) {  nome = n;
  }
  public int getAnni() {

    return anni;
  }
  public void setAnni(int n) {  anni = n;
  }
  public int getDifferenzaAnni(Dipendente altro) {  return (anni – altro.anni);
  }
}
```

Il reference this

Il metodo getDifferenzaAnni() dell'esempio precedente potrebbe aver provocato qualche dubbio. Sino ad ora avevamo dato per scontato che l'accedere ad una variabile d'istanza all'interno della classe dove è definita, fosse un "processo naturale" che non aveva bisogno di reference. Ad esempio, all'interno del metodo getGiorno() nella classe Data accedevamo direttamente alla variabile giorno senza utilizzare reference. Alla luce dell'ultimo esempio e considerando che potrebbero essere istanziati tanti oggetti dalla classe Data, ci potremmo chiedere: se giorno è una variabile d'istanza, a quale istanza appartiene? La risposta a questa domanda è: dipende dall'oggetto corrente, ovvero dall'oggetto su cui è chiamato il metodo getGiorno(). Per esempio, nella fase d'esecuzione di una certa applicazione, potrebbero essere istanziati due particolari oggetti, che supponiamo si chiamino mioCompleanno e tuoCompleanno. Entrambi questi oggetti hanno una propria variabile giorno. Ad un certo punto, all'interno del programma potrebbe presentarsi la seguente istruzione:

```
System.out.println(mioCompleanno.getGiorno());
```

Sarà stampato a video il valore della variabile giorno dell'oggetto mioCompleanno. Quindi il metodo getGiorno() che è definito dal seguente codice:

```
public int getGiorno() {       return giorno;     }
```

deve usare un reference in qualche modo per la variabile giorno, perché nel momento in cui questo metodo viene chiamato sull'oggetto mioCompleanno, esso restituisce la variabile giorno dell'oggetto mioCompleanno, e non quello dell'oggetto tuoCompleanno. Ovvero è come se il metodo fosse definito così:

```
public int getGiorno() {       return mioCompleanno.giorno; }
```

Gestione dei package

Il significato di public è chiaro quando applicato a variabili e metodi. Ma come mai sino ad ora abbiamo definito tutte le classi public? Per capirlo dobbiamo introdurre meglio il concetto di package. Abbiamo visto come la libreria standard di Java sia organizzata in package. Grazie a questo concetto, il programmatore ha quindi la possibilità di organizzare anche le proprie classi. È molto semplice infatti dichiarare una classe appartenente ad un package. La parola chiave package permette di specificare, prima della dichiarazione della classe, il package di appartenenza. Ecco un frammento di codice che ci mostra la sintassi da utilizzare:

```
package programmi.gestioneClienti;
public class AssistenzaClienti {     . . . . .
```

In questo caso la classe AssistenzaClienti appartiene al package gestioneClienti, che a sua volta appartiene al package programmi. Dichiarare la classe AssistenzaClienti pubblica, permetterà a tutte le classi che vengono dichiarate anche esternamente al package programmi.gestioneClienti di utilizzarla. Se non avessimo anteposto la parola chiave public alla classe AssistenzaClienti, solo le classi appartenenti allo stesso package avrebbero potuto usare la classe
AssistenzaClienti.
I package fisicamente sono semplici cartelle (directory). Ciò significa che, dopo aver dichiarato la classe appartenente a questo package, dovremo inserire la classe compilata all'interno di una cartella chiamata gestioneClienti, situata a sua volta all'interno di una cartella chiamata programmi.

Modificatori d’accesso

Sino ad ora abbiamo visto che esistono private e public, due parole chiavi che rappresentano degli aggettivi con cui andiamo a definire delle caratteristiche per i membri della nostra classe. Questi aggettivi vengono detti modificatori e ne abbiamo già incontrati altri come final o static. Un modificatore è una parola chiave capace di cambiare il significato di un componente di un’applicazione Java.

- ❑ Modificatore public
- ❑ Modificatore protected
- ❑ Modificatore private
- ❑ Il modificatore static

Inoltre abbiamo

- ❑ Metodi statici
- ❑ Variabili statiche (di classe)