Programmazione I

A.A. 2002-03

Programmazione Orientata agli Oggetti (1):

Principi generali

(Lezione XXV)

Prof. Giovanni Gallo
Dr. Gianluca Cincotti
Dipartimento di Matematica e Informatica
Università di Catania
e-mail : {gallo, cincotti}@dmi.unict.it

Prima lezione sugli oggetti: agenda

La programmazione ad oggetti è molto semplice se si comprendono pienamente alcuni principi fondamentali e alcuni concetti.

In questa lezione presentiamo:

a) Le ragioni e i principi che hanno richiesto una evoluzione dalla programmazione tradizionale;

b) Un esempio concreto implementato in maniera tradizionale e con gli oggetti e ne prenderemo pretesto per introdurre le idee principali sugli oggetti.

ATTENZIONE: in questa fase non è importante la sintassi e l'esempio (piuttosto banale) serve solo a scopo illustrativo.

Questa prima lezione sull'argomento non vuole presentare la sintassi e i dettagli con cui JAVA implementa gli "oggetti".

Vogliamo invece CAPIRE LA FILOSOFIA DEGLI OGGETTI: Concentriamoci su questo!!!

Il paradigma della PROGRAMMAZIONE STRUTTURATA

Tutti gli esempi ed esercizi svolti sono stati articolati come un unico programma (contenuto dentro una "classe") contenente:

a1) **Dati** visibili da tutti i metodi della classe;

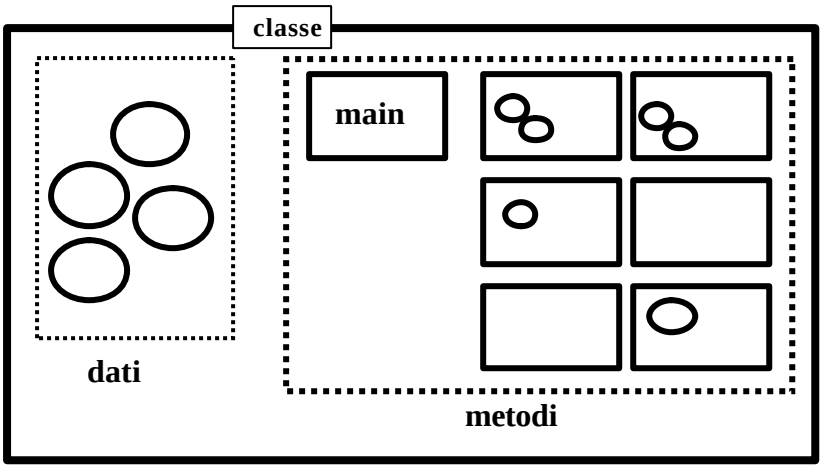
a2) **Dati** visibili solo dentro ciascun metodo;

b) **Metodi** o funzioni, "pilotati" dal metodo main.

I metodi ripartiscono in "moduli" in maniera razionale le operazioni da effettuare: si parla di "programmazione strutturata".

Poiché l'accento in questo contesto è sulle azioni svolte dai metodi si parla anche di "programmazione procedurale".

Generico programma strutturato: una rappresentazione grafica



La programmazione strutturata è stata lo "standard" degli anni 70/80

- **Passo avanti**:
  - maggiore astrazione mediante l'uso di "moduli" e "funzioni" che scompongono in passi più semplici procedure complesse.
- **Limiti:**
  - difficile realizzare, mantenere, far evolvere e riutilizzare programmi davvero complessi (maggiori di 1000/2000 linee di codice);

Ragioni della insufficienza: Alcuni principi importanti non sono "forzati" dalla programmazione strutturata/procedurale e sono lasciati solo alla "disciplina" e buona volontà del programmatore:

- Information Hiding
- Encapsulation;
- Semplicità.

Principio della "Information hiding"

Ciascuna "unità" di un programma deve "rendere note" alle altre unità solo le informazioni indispensabili.

Si ha un errore se informazioni non indispensabili sono visibili all'esterno della unità.

E' quindi importante che una unità di programma renda chiaramente visibile alle altre unità "**COSA FA**" e nasconda il "**COME**" (quali tecniche? quali variabili? quali algoritmi?).

Esempio di "Information Hiding" all'opera

I metodi di JAVA:

Tutti i metodi "*conoscono*" solo i parametri di ingresso richiesti dagli altri metodi e il tipo di dato restituito da ciascuno di essi.

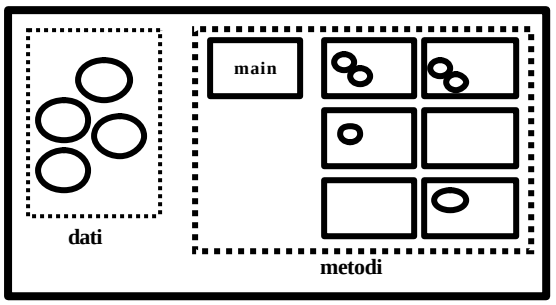
Per tutti gli altri aspetti essi sono delle "scatole nere". Per esempio le variabili dichiarate dentro un metodo non sono accessibili agli altri metodi.

esempio1

La"Information Hiding" non è garantita se si lavora solo con una classe

Violazione 1: tutti i metodi vedono i dati "globali" sia che debbano elaborarli sia che non debbano usarli.

Violazione 2: tutti i metodi vedono tutti gli altri metodi sia che debbano chiamarsi a vicenda, sia che essi compiano funzioni del tutto indipendenti.



Queste violazioni al principio IH sono inevitabili ma non hanno serie conseguenze se i metodi e i dati sono relativi alla descrizione di un unico semplice problema.

Ne segue la necessità per problemi complessi di suddividere il programma in più di una classe:

Un programma JAVA nasce dalla "cooperazione" di diverse classi tra loro.

Principio di "Encapsulation"

Una **unità di programma** (una **classe**) racchiude, in una sorta di guscio protettivo:

a) Tutti i dati utili alla descrizione di una entità che si vuole modellare;

b) Tutti i metodi necessari per leggere, scrivere, elaborare ed alterare i dati di cui al punto a).

Non è consentito a nessuna altra unità di programma di accedere in nessun modo ai dati se non utilizzando esplicitamente alcuni dei metodi previsti al punto b) e che sono visibili al di fuori della "capsula".

Questo principio è dettato dalla necessità di "proteggere" i dati dall'accesso casuale, non documentato e non espresso in maniera esplicita nel codice.

(Problema molto frequente nella programmazione procedurale/strutturata classica, specialmente in progetti complessi).

ENCAPSULATION in pratica

Le variabili che descrivono un oggetto possono essere elaborate (inizializzate, lette, alterate etc.) SOLO da metodi propri dell'oggetto che descrivono.

Non dovrebbe essere mai possibile accedere alle variabili di un oggetto direttamente da un oggetto ad esso esterno

(se ciò accade nei vostri programmi si tratta di un CATTIVO ESEMPIO di OO Programming che viene valutato come tale agli esami… e nella pratica professionale!).

Principio di "Semplicità"

- Gli oggetti che implementiamo nei nostri programmi JAVA debbono essere "semplici" corrispondenti a blocchi di informazioni omogenee o logicamente strettamente correlate in una unità.
- Nel programmare ad oggetti si deve SIMULARE ciò che osserviamo nella realtà dove la complessità proviene dall'interazione tra oggetti semplici.

- Un programma JAVA con un unico complicato oggetto **NON è** un programma OO: *gli studenti che si ostineranno a produrli… non si sorprendano delle insufficienze agli esami…*

Il programma mono-classe è l'eccezione non la regola!

Information Hiding, Encapsulation e Simplicity suggeriscono in modo naturale di organizzare i programmi in diverse unità dette classi che contengono dati omogenei e metodi per elaborarli.

Le varie classi collaborano tra loro chiamando vicendevolmente i metodi o i dati di ciascuna classe che sono stati resi visibili all'esterno di essa da chi ha progettato la classe.

Prima di entrare nella sintassi e nei "punti sottili" analizziamo un esempio "giocattolo" che ci sarà di guida.

Simulare un semplice gioco con i dadi

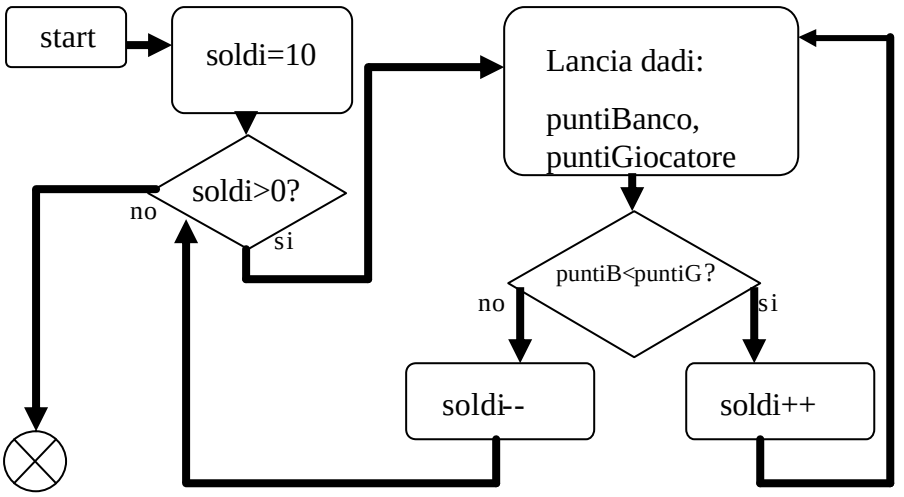
Il banco (rappresentato dal computer) e un giocatore (l'utente) lanciano a turno un dado.

Se il punteggio del giocatore è maggiore di quello del banco il giocatore vince 1.00 €. Altrimenti lo perde.

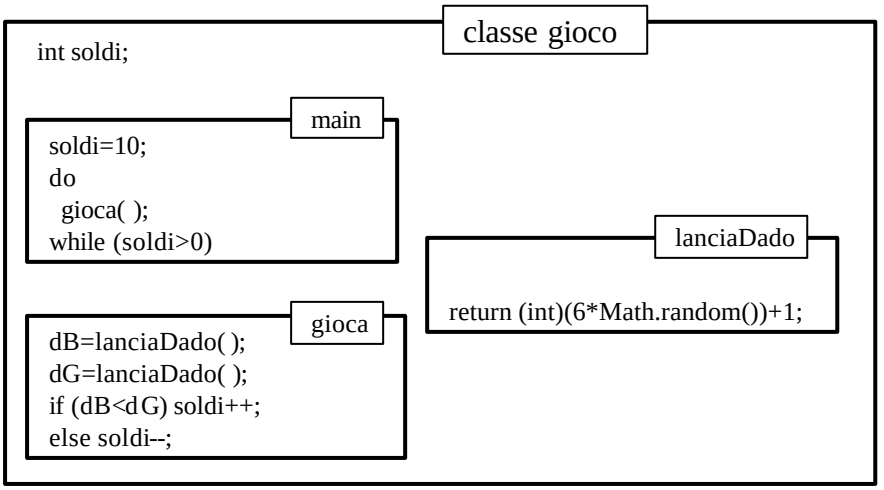
Il giocatore inizia con 10.00 € e il gioco finisce quando il giocatore ha esaurito il suo budget (se è molto fortunato non finirà mai!)

Svilupperemo lo schema di una implementazione procedurale (mono-classe) del gioco e lo confronteremo con uno schema di implementazione ad oggetti.

Gioco coi dati: diagramma di flusso



Gioco coi dati: schema procedurale



Esaminiamo lo schema procedurale

L'azione complessa del gioco è stata scomposta in azioni più semplici o elementari e i dettagli di ogni azione nascosti nei metodi (applicazione di **Information Hiding**)

I nomi dei metodi? Tutti verbi! Abbiamo focalizzato l'attenzione sulle **azioni** da compiere. L'analisi si è concentrata sulle procedure che compongono il programma.

Per consentire ai metodi di aggiornare "soldi" abbiamo definito tale variabile in modo che fosse visibili da tutti i metodi della classe (violazione di Information Hiding!).

Il metodo gioca( ) accede alla variabile soldi e la altera, ma ciò non è esplicitamente visibile nella sua intestazione! (violazione di Encapsulation!)

Passiamo allo schema "ad oggetti"

Il programma si comporrà di oggetti costruiti a partire dalle specifiche di classi di oggetto: dado, giocatore e gioco.

In particolare ci saranno due istanze di oggetti di tipo dado, una istanza di tipo giocatore e una istanza di tipo gioco.

Nel seguito useremo i seguenti aggettivi:

Privato = *visibile solo all'interno della classe secondo le regole di visibilità già note.*

Pubblico = *visibile anche a tutte le classi esterne.*

Gioco coi dadi, classi degli oggetti necessari

Parte privata Black Box	***Parte pubblica - visibile alle altre classi***
dado	**dado( )** *metodo "pubblico" costruttore per creare un nuovo dado*
	int lancio( ) *metodo "pubblico" che restituisce un intero tra 1 e 6.*
giocatore	**giocatore(int n)** *metodo "pubblico" costruttore per creare un nuovo giocatore con n euro di credito*
int soldi; *variabile "privata"*	**int getSoldi( )** *metodo "pubblico" che dice il numero di euro in cassa.*
	int incrementaSoldi( ) *metodo "pubblico" che aumenta di uno il numero di euro disponibili.*
	int decrementaSoldi( ) *metodo "pubblico" che diminuisce di uno il numero di euro disponibili.*

Gioco coi dadi, schema ad oggetti:il regista.

Parte privata

gioco

giocatore Pippo=new giocatore(10);
variabile privata che rappresenta il giocatore.

dado dadoBanco=new dado( );
variabile privata per il dado del Banco.

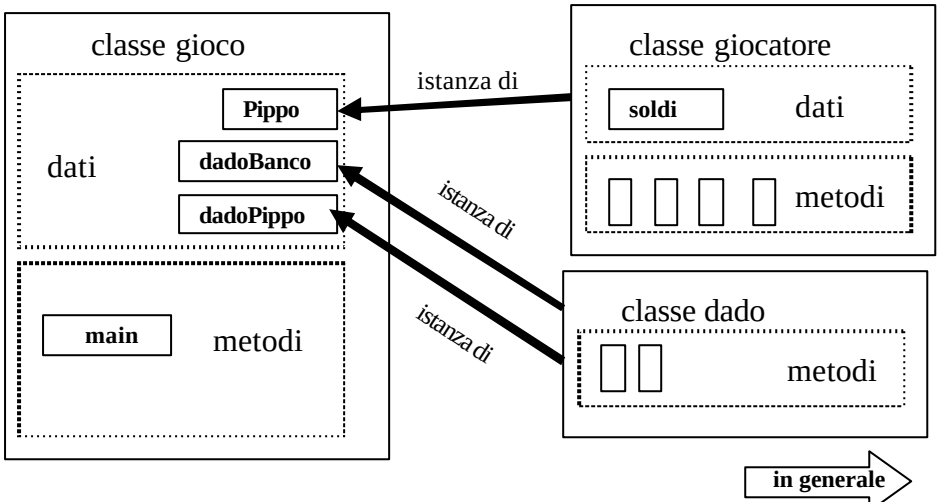
dado dadoPippo=new dado( );
variabile privata per il dado di Pippo.

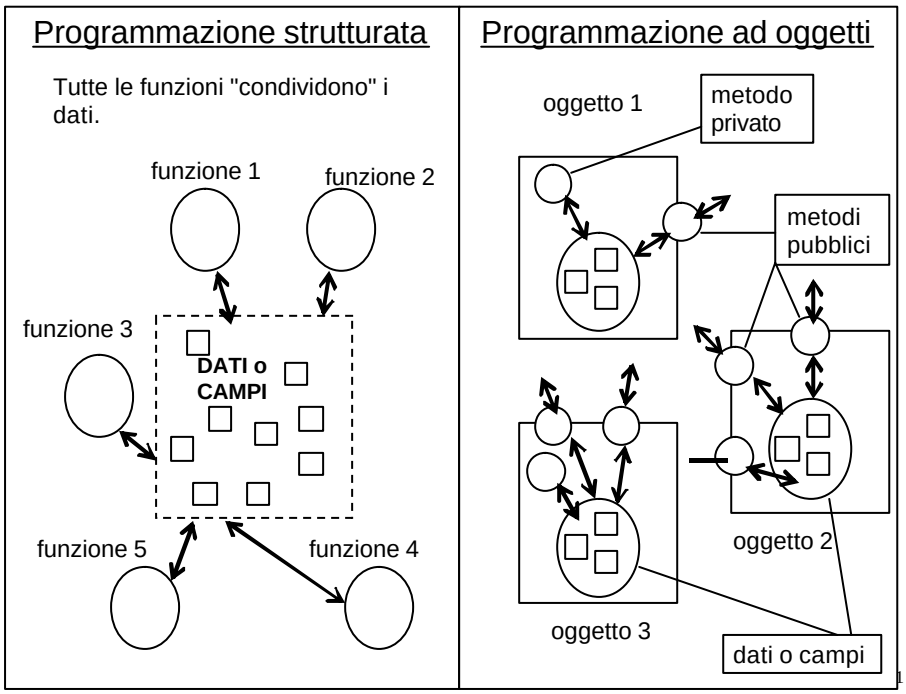
Parte pubblica

main (…) *metodo "pubblico" invocato dalla JVM appena le si dà in input la classe gioco. Ecco il suo body essenziale:*

```
do
 {if (dadoBanco.lancio()<dadoPippo.lancio())
        Pippo.incrementaSoldi();
  else Pippo.decrementaSoldi();
}
while (Pippo.getSoldi()>0);
```


L'ambiente previsto dallo schema ad oggetti





Esaminiamo lo schema ad oggetti dell'esempio

L'azione complessa del gioco è stata scomposta in azioni più semplici o elementari e i dettagli di ogni azione nascosti nei metodi dei vari oggetti che intervengono (applicazione di **Information Hiding**).

L'analisi è stata svolta individuando "chi" compie le varie azioni.

I nomi dei metodi? Tutti verbi!

I nomi degli oggetti? Tutti sostantivi!

"soldi" è una variabile privata: viene aggiornata esplicitamente solo da metodi della classe giocatore (applicazione di **Encapsulation**)

Commenti "pratici" allo schema ad oggetti

Non abbiamo curato l'input-output (fatelo per esercizio o vedete il file "gioco.java");

Abbiamo creato due dadi, uno del Banco e uno di Pippo (questo rende il codice più comprensibile, ma non era strettamente necessario, esattamente come avverrebbe nella realtà!).

In pratica?

Le idee che abbiamo abbozzato sono interessanti ma… come si traducono in codice?

Guarderemo alla sintassi e ai dettagli nel seguito.

In questa fase è però importante "allenarsi" nel nuovo modo di pensare.

Potete guardare il codice completo nel file gioco.java.

Dal "progetto" di una classe all'"istanza"

Le classi dado e giocatore sono descrizioni generiche di "oggetti".

Per potere utilizzare questi oggetti dobbiamo "crearli" o "costruirli".

Con il comando "new dado()" costruiamo un oggetto concreto, che obbedisce alle specifiche che il programmatore ha definito nella classe dado.

Con il comando "new giocatore(10)" costruiamo un oggetto concreto, che obbedisce alle specifiche che il programmatore ha definito nella classe giocatore ed inoltre poniamo a 10 il numero di euro disponibili.

Gli oggetti concreti dadoPippo e dadoBanco sono istanze della classe dado.

L'oggetto concreto Pippo è una istanza della classe giocatore.

Il metodo costruttore (1)

Per generare una istanza a partire dal "progetto" di una classe si usa la sintassi:

```
<nomeClasse> nomeVariabile =
                  new <nomeClasse>(<parametri>)
```

Esempi:

```
dado mioDado = new dado();
```

```
giocatore Pluto = new giocatore(1000);
```

Compito del metodo costruttore è riservare un'area di memoria sufficiente a contenere i dati che descrivono l'istanza dell'oggetto, inizializzare tali dati e restituire a chi lo ha invocato l'indirizzo RAM di dove si trovi quest'area.

Il metodo costruttore (2)

Il metodo costruttore è un metodo indispensabile per ogni classe.

Se non viene definito dal programmatore JAVA ne adopera uno di default nel quale per le variabili di istanza viene riservato dello spazio ma esse non vengono inizializzate.

Il metodo costruttore a differenza dei metodi già visti non è né "void" né restituisce un preciso tipo. La sua intestazione è sempre del tipo:

```
public nomeClasse (eventuali parametri)
{…}
```


Il metodo costruttore (3)

Dopo l'esecuzione del comando:

```
dado mioDado = new dado( );
```

la variabile mioDado contiene solo il riferimento riferimento RAM dove l'oggetto mioDado è conservato.

```
System.out.print(mioDado)
```
produrrà dunque una stampa del tipo:

```
dado@23f45
```

Questo comportamento è identico a quello già osservato per array e String che sono esempi di oggetti JAVA!

esempio2

Overload del metodo costruttore

Gli oggetti reali spesso sono descritti in differenti modi.

Esempio. Una retta nel piano può essere descritta mediante:

- Le coordinate di due suoi punti: $(x_1, y_1),(x_2, y_2)$;
- Le coordinate di un suo punto e il coefficiente angolare: (x1,y1),m;
- I coefficienti della sua equazione cartesiana: Ax+By+C=0.

I costruttori di JAVA come gli altri metodi ammettono l'overload, con la consueta regola di non produrre due metodi con la medesima firma ("signature").

esempio3

Chi costruisce la classe "gioco"?

La classe gioco è costruita automaticamente dalla JVM quando viene passato il suo bytecode all'interprete JAVA.

JVM costruisce la classe e subito dopo attiva il programma main.

Vedremo in seguito con maggiore dettaglio cosa avviene nella RAM quando si costruisce un oggetto.

Esercizi: carta e penna!

Quali oggetti? Quali metodi per ciascun oggetto?

- **Problema:** simulare le operazioni di deposito e prelievo dal conto corrente di un cliente.
- **Problema:** simulare un gioco con un mazzo di carte, vince chi pesca la carta più alta in valore e seme.
- **Problema:** trovare le coordinate del punto P di intersezione tra la retta del piano che passa per i punti A e B e la retta che passa per i punti C e D. (attenzione… alle rette parallele!).
- **Problema:** il gioco della tombola.

Fine