5. STRUTTURA DI UN PROGRAMMA JAVA E FONDAMENTI DEL LINGUAGGIO (par 3 libro)

Un progetto java è costituito da un insieme di classi. Ogni classe si ottiene compilando un file che ha come estensione **.java** e come nome il nome della classe. Almeno la classe principale (che noi chiameremo generalmente MainClass) deve avere un metodo main() da cui avrà inizio l'esecuzione del programma. Una volta compilati (in bytecode), i file delle classi hanno estensione **.class**.

Fare: cercare, in un progetto, il path dove si trovano i file .java, i . class e i package.

I package

I **package**: sono contenitori di classi (ossia di file .java) "legate" fra loro dal punto di vista logico. Un package definisce uno **spazio dei nomi per le classi** (…alla fine, poi, il package non è altro che una cartella). Creare uno spazio dei nomi (detto namespace) è come "dare un cognome" alla classe, è come dire "a quale famiglia appartiene", e questo si fa con i package. Quando più programmatori collaborano ad un progetto, possono disinteressarsi della sovrapposizione dei nomi delle classi se i package di cui tali classi fanno parte sono diversi.

Ad esempio, ipotizziamo di realizzare un progetto che sia una "simile" ad Autocad. Nel progetto si potrà creare un package chiamato "strutture" in cui verrà creata una classe chiamata "Finestra" che serve per modeillazzare degli oggetti che sono degli elementi architettonici "Finestra di un edificio". Inoltre si potrebbe creare un package "interfaccia" contente un'altra classe chiamata "Finestra" che rappresenta una finestra sullo schermo del pc. **(OSSERVAZIONE: per convenzione i package iniziano con la minuscola e le classi con la maiuscola).**

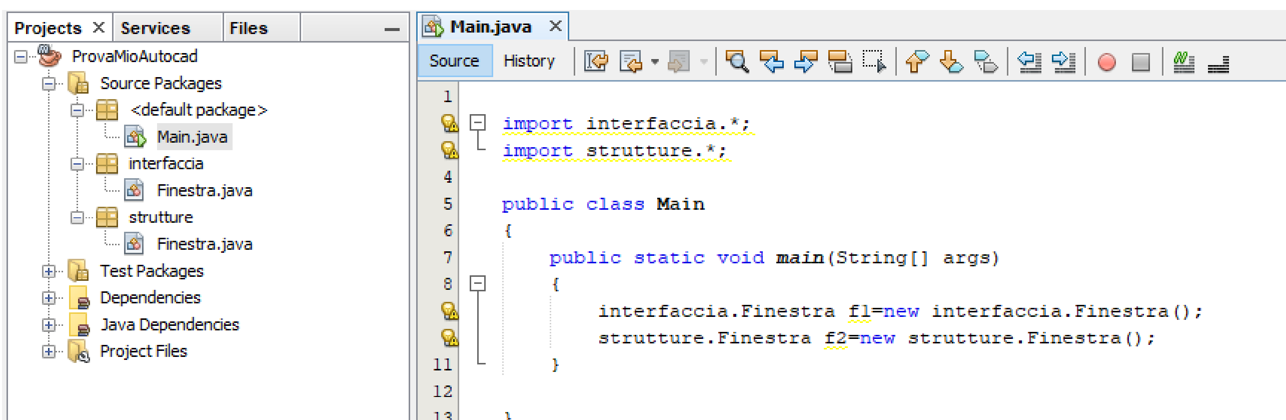
A questo punto nel programma vi sono due classi Finestra che hanno due significati diversi e che, giustamente, fanno parte di due package diversi. Ipotizziamo di voler istanziare, all'interno di una Main class, sia oggetti "finestra" della prima classe (la finestra dell'edificio) che oggetti "finestra" della seconda classe (la finestra del pc). Dopo aver importato i rispettivi package, le due classi Finestra potranno essere distinte l'una dall'altra utilizzando la "dot" notation ed indicando il nome del package prima del nome della classe:

strutture.Finestra

interfaccia.Finestra

Un po' come se il package fosse "Il cognome" di una classe che permette di distinguerla da altre classi con lo stesso nome.

Esempio:



Quando si scrive il codice Java per realizzare una classe, prima della definizione della classe stessa (public class..) va riportata l'indicazione del package di appartenenza, ad esempio:

package strutture;

oppure

package interfaccia;

I package non sono altro che cartelle, e anche per essi è possibile definire una struttura gerarchica. (nel nostro esempio tipo "Autocad", si potrebbe pensare che i package *"strutture", "materiali"*, siano posti all'interno di un package "*architettura"*)

Se non è specificato alcun package per una classe, essa viene posta in un package chiamato "*default package"*.

Per utilizzare, all'interno di una classe, altre classi appartenenti a packages diversi, è necessario "importare" tali altri packages. Per importare le classi contenute in un package la sintassi utilizza la dot notation:

import package_piu_esterno.package_piu_interno.nome_classe

Ad esempio, abbiamo visto che per scrivere sulla console con l'istruzione System.out.println() è necessario importare la classe System, che fa parte del package "lang", che a sua volta fa parte del package "java" che è il più esterno (quello che contiene tutti i package del linguaggio). Quindi per importare la classe System la sintassi è:

import java.lang.System;

Se si utilizza l' * anziché il nome della classe, vengono importate tutte le classi di un package

import java.lang.*;

La classe
Come già visto la classe è una rappresentazione astratta della descrizione di oggetti. La struttura di una classe in linguaggio java, ossia la sequenza di elementi da indicare nel file .java, è la seguente:

- Commento che spiega cosa fa la classe (per commentare una riga si usa //, per più righe si usa /*…..*/)
- dichiarazione del package a cui appartiene la classe
- importazione di eventuali altri package (per utilizzare nella classe altre classi statiche o oggetti)

  per importare i package si usa la parola chiave “**import”** seguita da tutta la gerarchia del package. Con * si indicano tutte le classi di un package:

  import java.io.* (package “io” incluso nel package java, che è il più esterno)
  import java.lang.*

  In Eclipse e in NetBeans i package più utilizzati sono già importati di default senza doverlo specificare. Noi importeremo package solo quando necessario.
- definizione della classe

public class nomeClasse
{
- Attributi pubblici (quasi mai)
- Attributi privati
- Attributi protetti
- Altri Attributi (se non si dichiara niente sono di livello package, vedremo)

- Metodi Costruttori (sono più di uno)

- Altri Metodi

}

Membri di una classe

Con il termine “membri di una classe” si indicano gli attributi e metodi della classe.

Attributi: costituiscono la componente informativa di una classe. Assumeranno dei valori all’interno degli oggetti della classe nel momento in cui tali oggetti verranno istanziati.

Oltre agli attributi è possibile definire **variabili locali** all’interno dei metodi (contatori ecc.)

La **visibilità** (**scope**) funziona come in C/C++, se una variabile globale e una variabile locale hanno lo stesso identificativo (nome) la variabile locale “copre” quella globale con lo stesso nome.

Le **costanti** sono definite premettendo al tipo di dato il token “**final”**

final float PI_GRECO = (float)3.14; (scriviamo sempre in maiuscolo le costanti)

Metodi: definiscano il “comportamento” degli oggetti, sono definiti in maniera analoga alle funzioni del C/C++ (c’è una signature con tipo di dato restituito e parametri) ma **non è possibile specificare il passaggio di parametri per indirizzo,** i parametri dei metodi sono sempre passati **per valore.**

Allora un metodo non potrà mai modificare il valore di una variabile passata per parametro?
La risposta è: non potrà farlo se il parametro è una variabile, ma potrà farlo se il parametro è un oggetto.

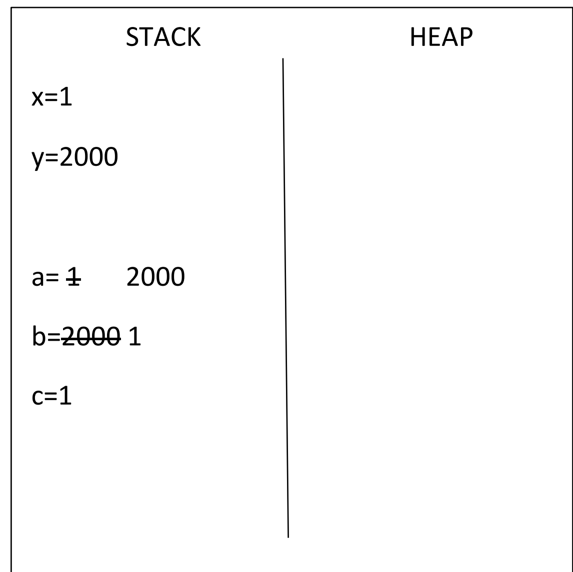
Vediamo il seguente esempio:

si vuole realizzare un metodo che scambia il valore di due numeri interi (in c/c++ si poteva fare questo grazie al passaggio per indirizzo).

Il seguente metodo “scambia” **non consente** di scambiare fra loro i valori dei due variabili intere perché vengono scambiate solamente le “copie”

```
public static void scambia (int a , int b)
  {
     int c;
     c=a;
     a=b;
     b=c;
  }

public static void main(String[] args)
  {
    int x=1;
    int y=2000;
    scambia(x,y);
    System.out.println("x="+x);
    System.out.println("y="+y);
  }
```



Se però si costruisce una classe chiamata, per esempio “numero” che contiene come attributo un numero intero, allora si potrà realizzare un metodo “scambia” che consente di scambiare fra loro il valore di due oggetti istanza di tale classe passati come parametro:

Classe numero:

```
public class Numero
{
   private int x;

public Numero(int x)
   {
      this.x=x;
   }
   public void setNumero(int x)
   {
      this.x=x;
   }
   public int getNumero()
   {
      return x;
   }
}
```

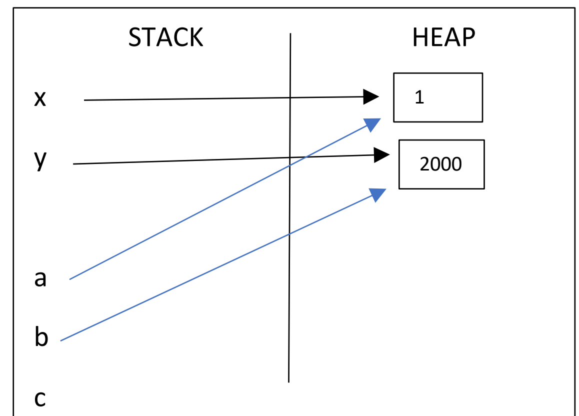
Numero
-int x
+ Numero(in x:int) **+ getNumero():int** **+ setNumero(in x:int):void**

Classe main:

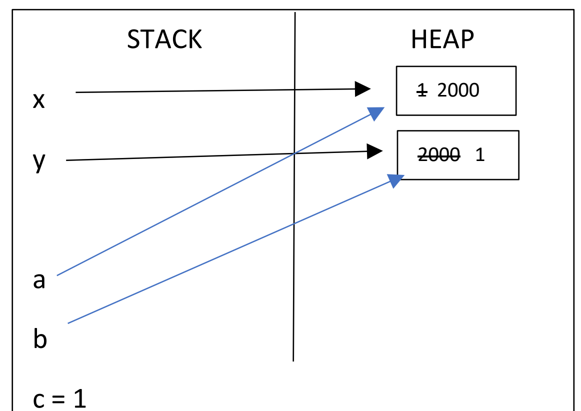
```
public static void main(String[] args)
  {
     Numero x=new Numero(1);
     Numero y=new Numero(2000);
     scambia(x,y);
     System.out.println("Fuori\n");
     System.out.println("x="+x.getNumero());
     System.out.println("y="+y.getNumero());
  }

  public static void scambia(Numero a, Numero b)

{
     int c;
     c=a.getNumero();
     a.setNumero(b.getNumero());
     b.setNumero(c);
  }
```



Il contenuto di x e y viene scambiato!



Accessibilità

Il concetto di accessibilità si applica sia agli attributi che ai metodi. Come già detto gli oggetti comunicano fra loro attraverso lo scambio di messaggi, ossia un oggetto manda un messaggio ad un altro oggetto quando ne invoca un metodo. Un oggetto A può dunque invocare un metodo di un oggetto B (teoricamente potrebbe anche accedere agli attributi di B, ma questo lo deve evitare il programmatore). Il **livello di accessibilità** di un attributo o metodo, specifica **quali** altri metodi (della stessa classe o di altre classi) vi possono accedere, ed è riassunto nella seguente tabella:

Usati molto raramente

per ora non li usiamo

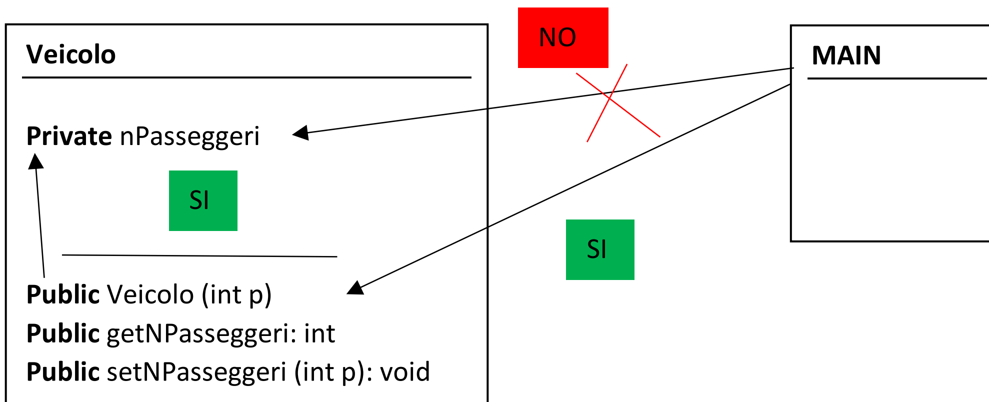
TABELLA **1**

Accessibilità	Metodi della classe	Metodi di una classe derivata	Metodi di una classe dello stesso *package*	Metodi di tutte le classi
`private`	SÌ	NO	NO	NO
`protected`	SÌ	SÌ	SÌ	NO
`public`	SÌ	SÌ	SÌ	SÌ
`default/package`	SÌ	NO	SÌ	NO

Tranne nel caso in cui faccia parte dello stesso package

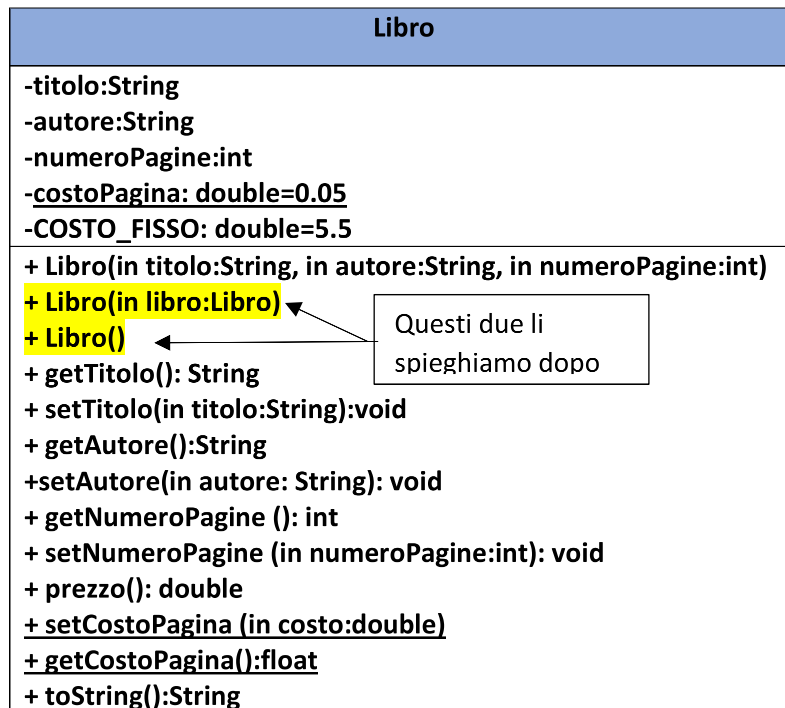
Private: un metodo o attributo private è accessibile solo da metodi della stessa classe (la classe è protetta da tutte le altre classi)

Public: un metodo o attributo public è accessibile da tutte le classi (nessuna protezione)



Membri statici

Consideriamo la seguente classe Libro:



Premettendo il modificatore **static** ad un attributo o a un metodo nella definizione di una classe, si rende tale membro della classe statico, ossia l'elemento (metodo o attributo) farà riferimento alla classe e non più ad ogni singolo oggetto istanziato. Le conseguenze sono che:

- Una modifica di un attributo statico di un oggetto si ripercuote su tutti gli oggetti istanziati di quella classe (esempio libro p.40, *l1.setCostoPagina(0.1)*)
- Un metodo pubblico statico può essere invocato direttamente con la dot notation applicata alla classe anziché all'oggetto (esempio p.40 L*ibro.setCostoPagina(0.1));*
- I metodi statici non possono (non è consentito) accedere agli attributi **non statici** di una classe poiché questi attributi hanno valori diversi per ogni istanza, e l'utilizzo di un metodo statico andrebbe ad uniformare tali valori.
  (esempio non posso creare un metodo statico setNumeroPagine, poiché questo metodo andrebbe ad impostare lo stesso numero di pagine in tutti i libri istanziati).

Nell'esempio si osservi l'utilizzo della parola chiave **this** per fare riferimento all'oggetto corrente in caso di ambiguità di nome fra attributi di una classe e parametri di un metodo. Tale parola chiave non è necessaria se non c'è ambiguità fra il nome dell'attributo e nome di un parametro o variabile.

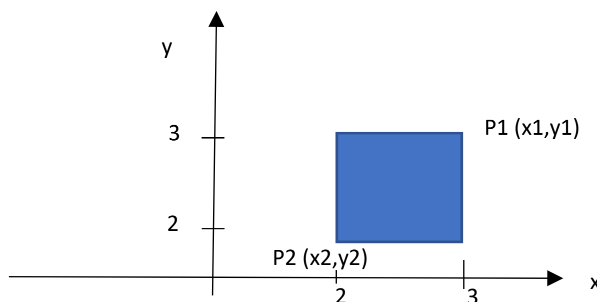
ESERCIZIO: PENSARE E REALIZZARE UNA CLASSE CHE POTREBBE AVERE ALMENO UN ATTRIBUTO STATICO E REALIZZARLA CON ALMENO I GETTER E I SETTER.

Costruttori

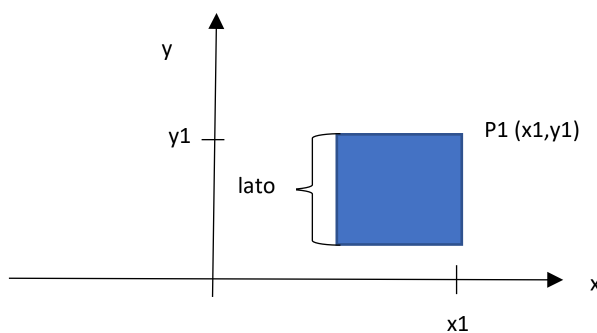
Abbiamo già parlato del costruttore. Il costruttore è un metodo che viene invocato per istanziare un oggetto di una classe. Il **nome** del costruttore è lo stesso della classe, non è necessario specificare il tipo restituito, la sua accessibilità è sempre **public** visto che deve essere accessibile da un'altra classe, il costruttore viene sempre preceduto dall'istruzione **new.** Scopo del costruttore è quello di assegnare i valori agli attributi dell'oggetto creato (ossia svolgere quell'operazione chiamata **inizializzazione**). Possono esserci, e solitamente ci sono, più di un costruttore. Perché?

Perché può essere utile costruire un oggetto in diversi modi, ossia partendo da parametri diversi. Facciamo un esempio: un oggetto "quadrato" sul piano cartesiano è univocamente individuato quando sono note le coordinate di due punti, ad esempio il punto P1 in alto a destra e il punto p2 in basso a sinistra. Gli attributi che consentono di definire un quadrato sono quindi le coordinate dei due punti: x1,y1 (coordinate di p1) e x2,y2 (coordinate di p2). Al costruttore andrebbero quindi passati come parametri le coordinate dei due punti:

```
public Quadrato (int x1,int y1, int x2,int y2)
{
        ......
}
```



Un oggetto "quadrato" potrebbe però essere istanziato anche a partire da altre informazioni: ad esempio il punto in alto a dx (p1) ed il valore del lato (l):



Si potrebbe quindi realizzare un altro costruttore che accetta come parametri, le coordinate del punto p1 e il valore del lato. In tal caso la *signature* del metodo costruttore sarebbe:

```
public Quadrato (int x1, int y1, int lato)
        {
                ….
        }
```

In presenza di più costruttori il compilatore selezionerà quello corretto in funzione del tipo di parametri utilizzati nell'invocazione. La presenza di più metodi con lo stesso nome, ad esempio più metodi costruttori di una classe, viene indicato con il termine di *overloading*. Il motivo di questo nome è che, nella situazione descritta, il nome del metodo viene "sovraccaricato" di significati.

Esistono due costruttori particolari che è buona pratica inserire sempre quando si crea una classe:

- Il **costruttore di default**: non ha parametri ed assegna dei valori di default agli attributi (es. p.42 libro)
- Il **costruttore copia**: ha come parametro un oggetto istanza della stessa classe e consente di costruire un clone dell'oggetto ossia un oggetto identico a quello passato per parametro (ma che ha una propria "vita" nello heap) (es. p.42 libro, ad esempio se ho più copie dello stesso libro).

Javabeans: classe riutilizzabile in diversi contesti che richiede, per convenzione, che ci sia il costruttore di default e che per gli attributi siano definiti i metodi getter e setter.

9. CONVENZIONI DI CODIFICA DEL LINGUAGGIO JAVA (par 5 del libro)

Vediamo le **convenzioni di scrittura del codice JAVA** (molte le conosciamo già).
Come al solito gli identificatori devono avere un nome significativo per aumentare la leggibilità del codice. Usare nomi "simpatici" può sembrare divertente e non crea problemi finché si fanno programmi brevi e semplici, ma poi complica le cose. Allo stesso modo, non facilitare la lettura del codice può sembrare una cosa furba (il mio codice lo capisco solo io….) ma invece è una cosa poco utile e, nei casi reali si ritorce sempre contro colui che lo fa.
Le convenzioni non sono obbligatorie ma consentono una migliore lettura del codice quindi la possibilità di condividerlo e una maggiore facilità nella revisione. Sarebbe pertanto opportuno seguire le convenzioni che già abbiamo visto e che ricordo:

- Case sensitive **(obbligo, non convenzione)**
- **indentare il codice**
- **i package**: hanno iniziale minuscola
- **le classi**: Iniziano con lettera maiuscola, indicano delle cose, sono al singolare (*Televisore*, *Dado*)
- **gli attributi e i metodi** hanno la prima lettera minuscola e, se formati da più parole, le successive prime lettere maiuscole (contaPersone, aggiungiStudente…, questa modalità di indicazione dei nomi è detta"**low camel case**")

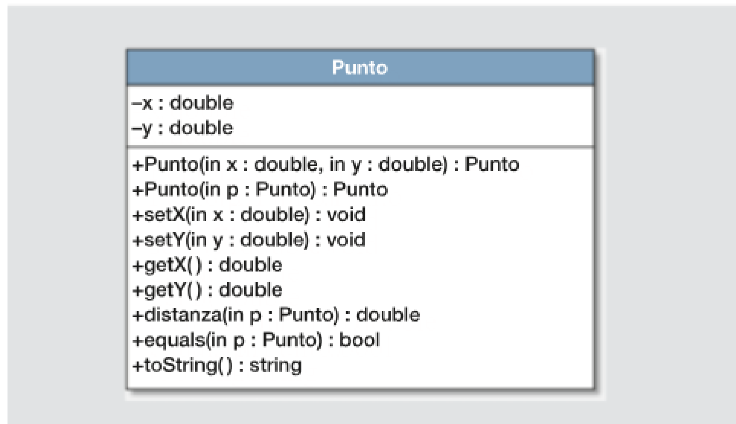
  - Usare identificatori che spigano "cosa sono". Ad esempio per le classi: *FiguraGeometrica*, *Automobile*. Ad esempio per un metodo che fornisce una descrizione: *getDescrizione*.
  - Usare identificatori non abbreviati (*valoreMedio* e non *valMed*) ma nemmeno troppo lunghi (valoreMedio e non *valoreMedioCalcolatoConLaFormulaStandardCheMiPiaceTanto*)
  - Per gli attributi costanti si usa la parola chiave **final**, identificatori con lettere maiuscole e parole separate da _ (*final Int VALORE_MAX=10, final double PI_GRECO=3.14*).
  - Usare identificatori omogenei in tutto il codice. Se ho una classe *Prodotto*, e una classe "Ordini" che contiene istanze della classe "*Prodotto*" il metodo che aggiunge prodotti all'ordine si dovrà chiamare chiamerà *aggiungiProdotto* e non, ad esempio, *aggiungiArticolo*.
  - Le convenzioni per i **javabeans** indicano che per ogni attributo pubblico vi siano i metodi ***getNomeAttributo*** e ***setNomeAttributo*** che consentano di impostare o acquisire i valori degli attributi.
  - Se il metodo restituisce un valore boolean le convenzioni indicano di farne preceder il nome da is (*public boolean isAcceso()*) oppure da has (*public boolean hasDipendenti()*).

- Il valore di ritorno di un metodo va messo in una variabile che poi viene restituita con *return*.

10. LA STRUTTURA DI BASE DI UNA CLASSE E IL METODO MAIN (par 4 del libro)

Un programma java è costituito da una classe principale, solitamente si chiama App.java in NetBeans, che contiene un metodo main nel quale vengono istanziati oggetti delle altre classi. Dopo la compilazione vengono generate le classi in bytecode (estensione.class) eseguibili dalla JVM.

Realizziamo la classe Punto dal diagramma UML delle classi di p. 43 del libro.



Si può trovare il codice della classe Punto a p. A 44 del libro

Nella MainClass istanziamo 3 punti (P1(1,1), P2(2,2), P3(1,1)) chiedendo l'input all'utente

- mostriamo i 3 punti inseriti invocando *toString()* per ciascun punto
- Calcoliamo la distanza, a due a due, fra i 3 punti invocando *p1.distanza(p2), p2.distanza(p3), p3.distanza(p1).*
  Risultato atteso:

  distanza p1-p2: 1.414213
  distanza p2-p3: 1.414213
  distanza p3-p1: 0

- Per testare il metodo equals verifichiamo se p1 e p3 sono coincidenti invocando *p1.equals(p3);*

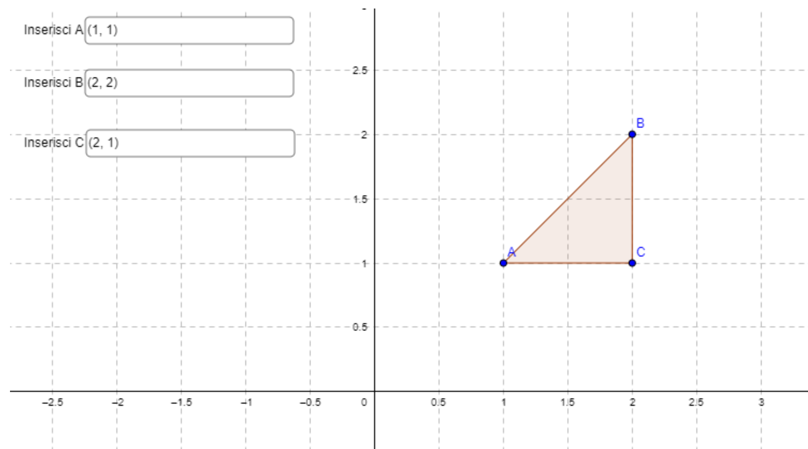
OSSERVAZIONE IMPORTANTE:

- Il metodo **toString**() è già presente di default per ogni oggetto che viene creato perché è ereditato dalla classe predefinita **Object**, che è la classe da cui derivano tutte le altre classi. Il comportamento del metodo toString va però sempre **ridefinito** in base agli attributi specifici della classe. Altrimenti, di **default, il metodo toString() invocato su un oggetto restituisce semplicemente il nome della classe di cui tale oggetto è istanza, e l'indirizzo in memoria dell'oggetto (ossia il contenuto del reference).**
  Un'operazione come questa, di riscrittura di un metodo ereditato, è chiamata "**override".**

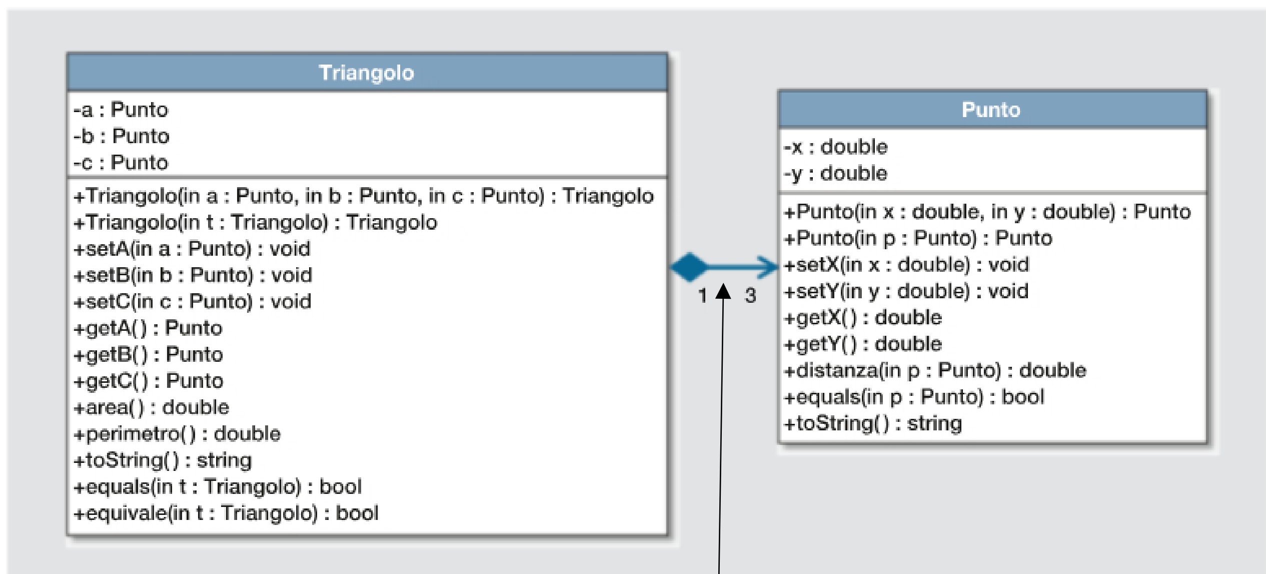
- **Il metodo equals()** è già presente di default, anch'esso ereditato dalla classe Object. Anche esso va ridefinito (**override**) perché nella sua versione di default coincide con l'operatore "==" restituendo true solamente se due reference puntano allo stesso oggetto, mentre generalmente, in una applicazione, interessa verificare che due oggetti diversi abbiano gli stessi valori per gli attributi.

Classe triangolo

Ora costruiamo la classe triangolo che istanzia oggetti di tipo Triangolo a partire da tre oggetti di tipo Punto:

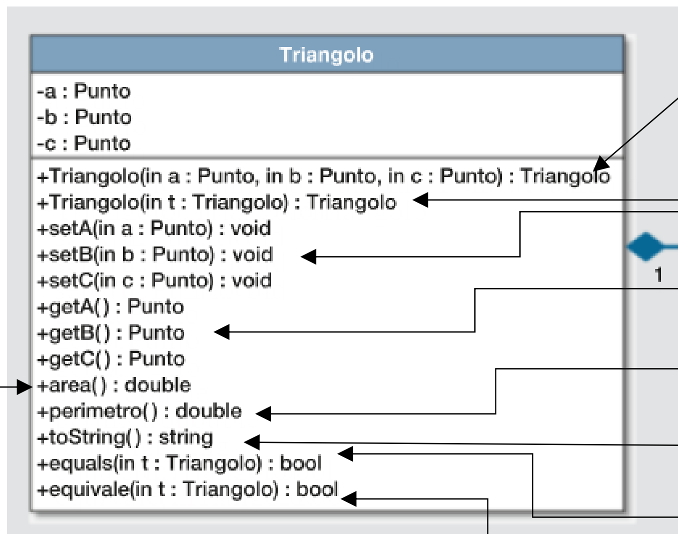


Il diagramma delle classi che rappresenta la relazione che c'è fra le due classi Triangolo e Punto è il seguente:



Questo simbolo rappresenta una associazione di tipo "composizione", indica che un triangolo sarà "composto" da 3 punti (come si può vedere dagli attributi di Triangolo)

Analizziamo più in dettaglio gli attributi della classe Triangolo



costruttore

costruttore di copia

metodi setter: assegnano un valore di tipo Punto agli attributi

metodi getter: restituiscono il valore degli attributi

calcola il perimetro

visualizza i tre punti che costituiscono il triangolo

verifica se due triangoli (uno passato per parametro e l'altro è quello su cui viene invocato il metodo) sono uguali, ossia costituiti dagli stessi tre punti

verifica se due triangoli (uno passato per parametro e l'altro è quello su cui viene invocato il metodo) sono equivalenti, ossia hanno la stessa area

calcola l'area con la formula di Erone.

La formula di Erone consente di calcolare l'area di un triangolo conoscendo la lunghezza dei lati (lato1,lato2,lato3).

Chiamando s il semiperimetro, la formula è:

A=sqrt (s (s-lato1)(s-lato2)(s-lato3))

Bella dimostrazione della formula, per chi non ci crede:
https://www.youtube.com/watch?v=_FjUyePK6I0

Si può trovare il codice della classe Triangolo a p. A 46 del libro

ATTENZIONE: poiché nel calcolo dell'area potrebbero esserci errori di arrotondamento nella memorizzazione in binario dei numeri con la virgola, nel codice viene utilizzata una costante ERR_MAX=0.00000001 per impostare uguale a 0 il risultato del calcolo dell'area eventualmente inferiore a tale valore.

Ora nella Main class andiamo ad istanziare un triangolo per fare un'**osservazione importantissima**:

- Istanziamo i 3 punti: p1 (1,1) p2(2,2) p3(2,1)
- Istanziamo il triangolo t1 costituito dai punti p1,p2,p3

Triangolo t1=new Triangolo (p1,p2,p3);

Osservazione importantissima:
Nel costruttore della classe Triangolo, per istanziare un triangolo vengono invocati i metodi setter che consentono di inizializzare gli attributi di tipo Punto a,b,c. All'interno di questi metodi setter viene utilizzato il costruttore di copia della classe Punto:

```
public void setA(Punto a)
{
        this.a=new Punto(a);
}
```

Assegna all'attributo "a" **una copia** del punto "a" passato come parametro. **Non lo stesso** punto "a" passato come parametro, ma una sua copia, che è un altro oggetto.

Questo è **importantissimo**, perché consente, quando si sistanzia un oggetto Triangolo t1, di garantire **l'indipendenza** fra l'oggetto Triangolo t1 e i punti (p1,p2,p3) utilizzati per "costruirlo". Grazie a questa indipendenza, se nella Main class, dopo aver istanziato il triangolo si modifica uno qualsiasi dei tre punti, il triangolo non viene modificato (provare a farlo nel main). Ciò è corretto, perché la OOP chiede che gli oggetti siano sempre, il più possibile fra loro **INDIPENDENTI**.

Se invece i metodi setter non utilizzassero il costruttore di copia della classe "Punto" per inizializzare gli attributi del triangolo, si creerebbe **enorme dipendenza** fra un triangolo e i punti da cui è costituito. Infatti utilizzando metodi setter scritti nel seguente modo, una eventuale modifica (nella Main class) del punto "p1" va a modificare il triangolo:

```
public void setA(Punto a)
{
        this.a=a
}
```

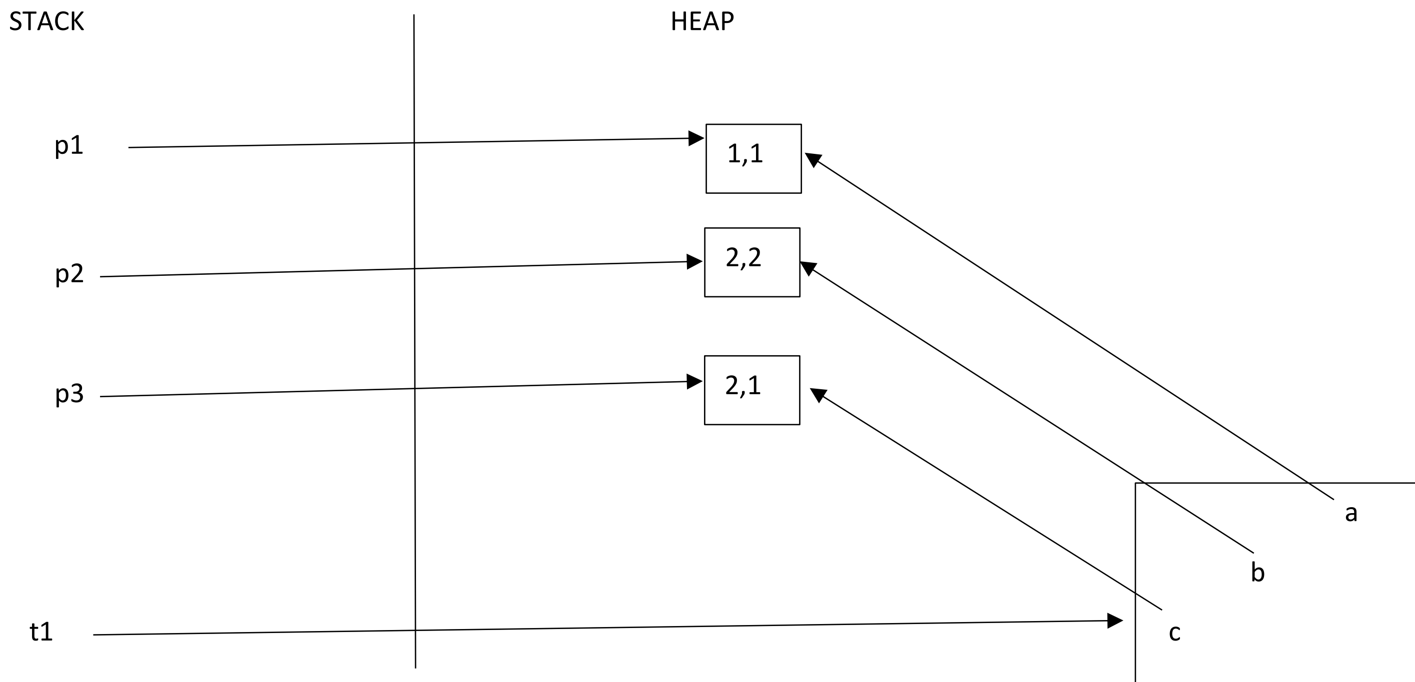
Assegna all'attributo "a" **non una copia** del punto "a" passato come parametro ma esattamente lo stesso oggetto puntato dal parametro "a", ossi il punto "p1". Quindi se il Punto "p1" si modifica, anche il triangolo si modifica. Non deve succedere. Questa è cattiva programmazione.

Provare a modificare i metodi setter e a modificare (nella main class) il punto "p1", verificare che il triangolo si modifica. Non deve succedere. Questa è cattiva programmazione.

Perché non deve succedere?
Perché NON E' CORRETTO che per modificare un oggetto Triangolo si agisca su un altro oggetto (un oggetto "Punto"). Per modificare un triangolo si deve agire solamente sul triangolo stesso, ossia invocare i metodi

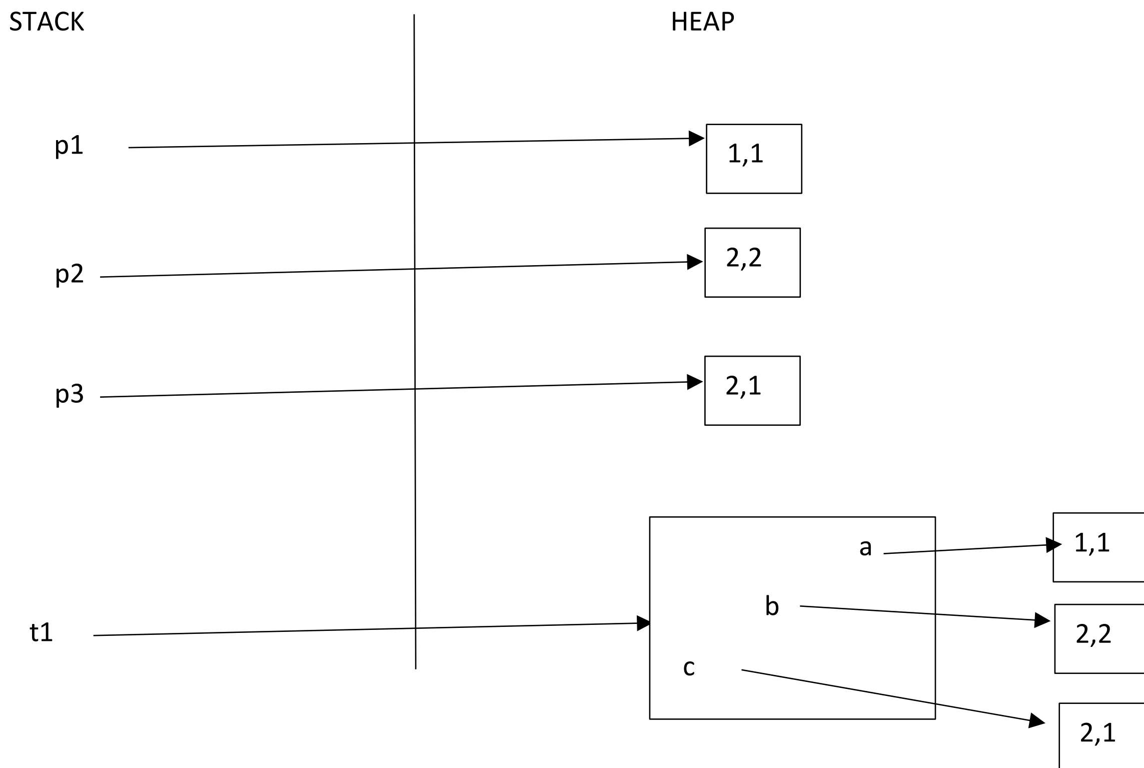
setter del triangolo! La classe triangolo deve essere costruita in modo da mantenere l'indipendenza dagli oggetti utilizzati per "costruirlo".
Per capire meglio cosa succede osserviamo lo stack e lo heap nei due casi:

Caso "**sbagliato**" (senza costruttore di copia nel setter).



Gli attributi del triangolo t1 "puntano agli stessi oggetti "p1","p2","p3" usati per istanziarlo.
Se cambia il valore contenuto nel Punto "p1", il triangolo si modifica: **enorme dipendenza.** Non la vogliamo!

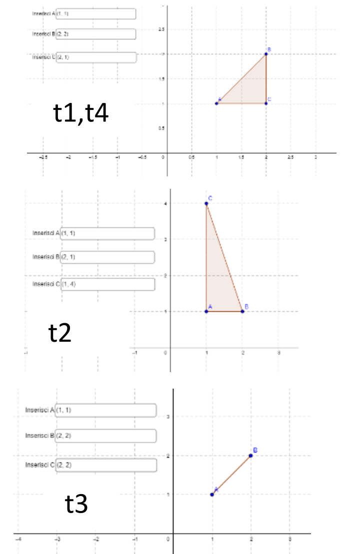
Caso “**corretto**” (con costruttore di copia nel setter).



Gli attributi del triangolo puntano a tre NUOVI oggetti Punto.
Se cambia il valore contenuto nel punto “p1” , il triangolo t1 non si modifica: **totale indipendenza fra gli oggetti.** Questa è buona programmazione!

Questo non vuol dire che “non possiamo modificare” il triangolo. Possiamo modificarlo, certo, ma agendo sui metodi del triangolo (sui metodi setter) e NON sui metodi dei punti usati per costruirlo.

Ora testiamo i vari metodi della classe Triangolo nella Main class:



- Istanziamo 5 punti:
  - p1(1,1)
  - p2(2,2)
  - p3(2,1)
  - p4(1,4)
  - p5(3,3)
- Istanziamo Quattro triangoli:
  - t1(p1,p2,p3)
  - t2(p1,p3,p4)
  - t3(p1,p2,p2) //non è un triangolo
  - t4 copia di t1
- Invochiamo per ogni triangolo i metodi *toString(), perimetro()* e *area()*
- verifichiamo se t1 e t4 sono uguali con il metodo *equals()*
- verifichiamo se t1 e t4 sono equivalenti con il metodo *equivale()*
- verifichiamo se t1 e t2 sono uguali con il metodo *equals()*
- verifichiamo se t1 e t2 sono equivalenti con il metodo *equivale()*

Il risultato è il seguente:

```
Triangolo T1: A(1.0;1.0) B(2.0;2.0) C(2.0;1.0)
Perimetro triangolo T1: 3.414213562373095
Area triangolo T1: 0.4999999999999998
Triangolo T2: A(1.0;1.0) B(2.0;1.0) C(1.0;4.0)
Perimetro triangolo T2: 7.16227766016838
Area triangolo T2: 1.5000000000000007
Triangolo T3: A(1.0;1.0) B(2.0;2.0) C(2.0;2.0) NON È UN
TRIANGOLO!
Perimetro triangolo T3: 2.8284271247461903
Area triangolo T3: 0.0
Triangolo T4: A(1.0;1.0) B(2.0;2.0) C(2.0;1.0)
Perimetro triangolo T4: 3.414213562373095
Area triangolo T4: 0.4999999999999998
I triangoli T1 e T4 sono uguali
I triangoli T1 e T4 sono equivalenti
I triangoli T1 e T2 non sono uguali
I triangoli T1 e T2 non sono equivalenti
```