Corso di fondamenti di informatica – terza parte

Ingegneria industriale - polo di Frosinone

- Programmazione orientata agli oggetti……………………………….pag. 2
- Gli oggetti e le classi: attributi e metodi……………………………pag. 3
- Diagramma delle classi e rappresentazione grafica ………………pag. 6
- UML: rappresentazione grafica di classi e istanze…………………pag. 9
- L'incapsulamento……………………………………………………….pag. 10
- La dichiarazione e l'utilizzo di una classe……………………………pag. 11
- I livelli di visibilità……………………………………………………….pag. 14
- La dichiarazione degli attributi e dei metodi…………………………pag. 18
- La creazione degli oggetti ed il metodo costruttore………………..pag. 22
- La parola chiave this……………………………………………………pag. 26
- Gli attributi e i metodi static…………………………………………..pag. 27
- Un esempio di applicazione con più classi……………………………pag 29
- Il mascheramento dell'informazione nelle classi…………………..pag. 33
- Gli array di oggetti……………………………………………………….pag. 42
- L'ereditarietà……………………………………………………………….pag. 45
- Il polimorfismo…………………………………………………………….pag. 57
- Le stringhe………………………………………………………………….pag. 60
- Le strutture di dati dinamiche…………………………………………pag. 65
- Esercizi………………………………………………………………………pag. 80

Programmazione orientamento agli oggetti

La **programmazione strutturata** è una metodologia che pone l'attenzione principalmente **sull'organizzazione dell'algoritmo** risolutivo. L'interesse è rivolto a quello che il programma deve fare e il problema viene sviluppato individuando le procedure e le sottoprocedure, cioè i passi dell'elaborazione che portano alla soluzione. Il programma viene costruito come un insieme di funzioni che vengono richiamate nell'ordine corretto a partire dalla funzione principale chiamata **main.**

Con il passare del tempo è aumentata la richiesta di sviluppare programmi sempre più complessi e in grado di gestire grandi quantità di dati. Si è quindi sviluppato un nuovo stile di scrittura del codice che ne consente un reimpiego sempre maggiore. Questo modo di programmare si chiama programmazione orientata agli oggetti. La programmazione orientata agli oggetti non presuppone l'eliminazione delle tecniche precedenti, ma piuttosto le completa, aggiungendo loro nuova funzionalità.

Gli oggetti e le classi: attributi e metodi

La programmazione orientata agli oggetti, denominata **OOP(Object – Oriented – Programming)**, prende il nome dall'elemento su cui si basa: l'oggetto. Al centro dell'attenzione c'è il sistema che si vuole analizzare: l'analisi del sistema si occupa di individuare le entità che fanno parte del sistema stesso e le interazioni tra queste entità.

Un programma realizzato con un orientamento ad oggetti si sviluppa quindi attraverso le **interazioni** tra gli oggetti durante l'esecuzione del programma, gli oggetti possono cambiare il loro stato e possono richiedere l'esecuzione di operazioni associate ad altri oggetti.

Gli oggetti rappresentano le **entità** del problema o della realtà che si vuole automatizzare. Un oggetto è in grado di memorizzare le informazioni che riguardano il suo stato; è anche possibile associare a un oggetto un insieme di operazioni che esso può compiere.

In particolare, un oggetto può essere definito elencando sia le sue caratteristiche (**attributi**), sia il modo con cui interagisce con l'ambiente esterno, cioè i suoi comportamenti (**metodi)**.

Gli oggetti e le classi

Gli **attributi** rappresentano gli elementi che caratterizzano l'oggetto, utili per descrivere le sue proprietà e definire il suo stato.

I **metodi** rappresentano le funzionalità che l'oggetto mette a disposizione.

Esempio:

Un'automobile può essere analizzata come un oggetto in termini di caratteristiche e comportamenti.

Alcune sue **caratteristiche** (attributi) sono: la velocità, il colore, il numero di porte, il livello del carburante e la posizione della marcia. Come si vede, queste caratteristiche possono descrivere le proprietà fisiche dell'oggetto, come il colore e il numero di porte, oppure possono indicare lo stato dell'oggetto in un determinato momento, come la velocità che può variare se si accelera.

Tra i **comportamenti (**metodi) dell'oggetto automobile possiamo avere: accelera, frena, gira destra, gira a sinistra, cambia marcia, riforniscíti, vai indietro, vai avanti, ecc.

Gli oggetti e le classi

Le funzionalità espresse dai comportamenti possono concretizzarsi con le azioni oppure con il cambiamento dello stato dell'oggetto. I comportamenti accelera e frena agiscono modificando la velocità dell'automobile. Il comportamento cambia marcia e riforniscití influenzano rispettivamente l'attributo marcia e livello carburante.

Si noti che i comportamenti (metodi) sono indicati tramite i **verbi**, mentre le caratteristiche (attributi) sono indicati tramite **aggettivi**.

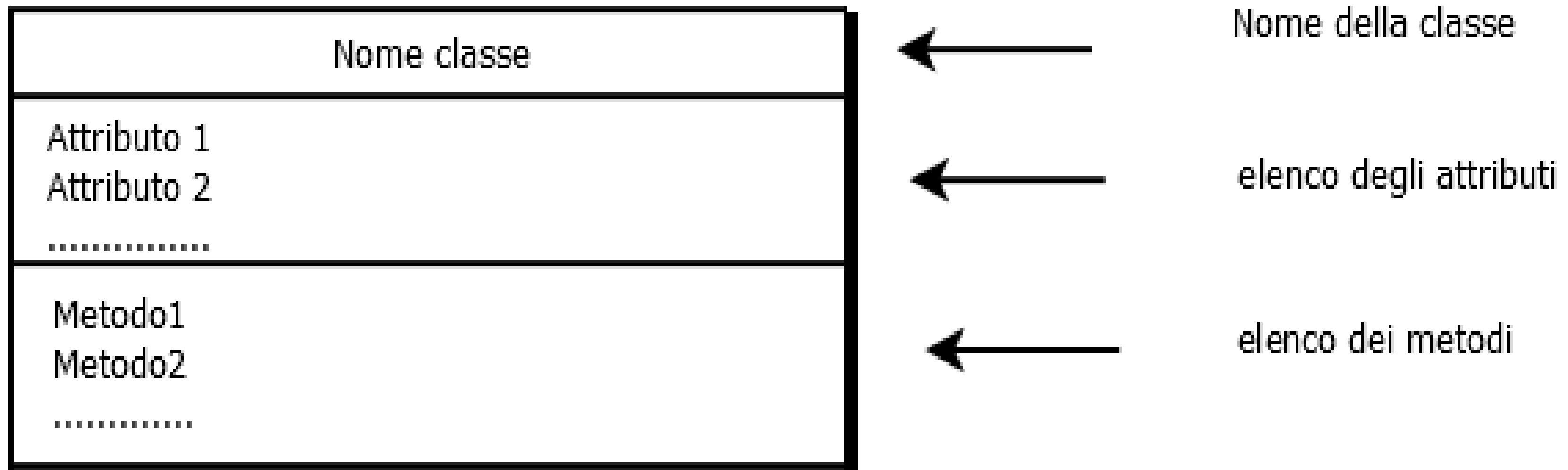
Le classi e le sue istanza

La struttura di un oggetto è completamente descritta quando vengono elencate sia le caratteristiche sia i comportamenti dell'oggetto. Nei linguaggi di programmazione orientati agli oggetti, la struttura di un oggetto viene descritta con le classi.

La **classe** è la descrizione astratta degli oggetti attraverso gli attributi e i metodi.

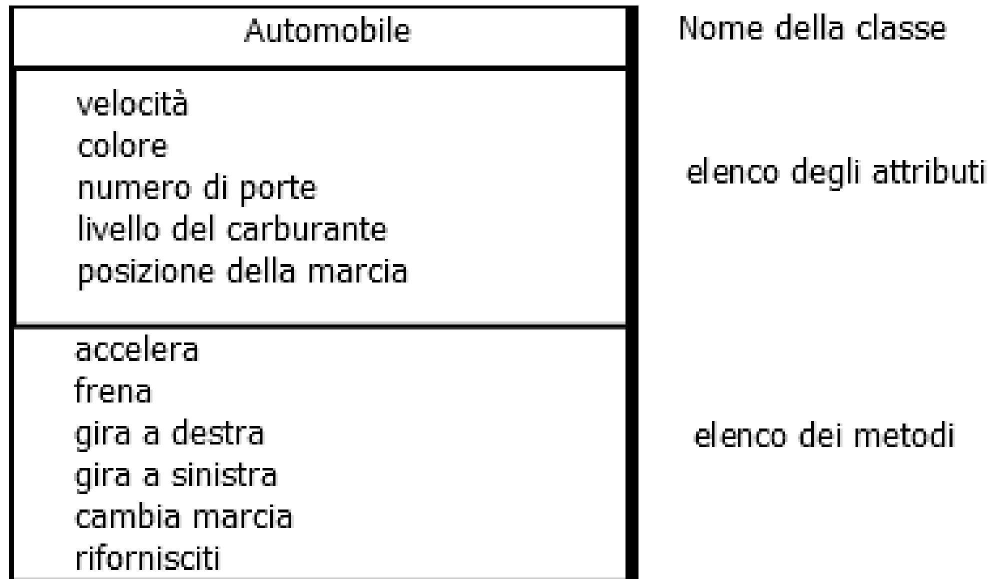
Digramma delle classi e rappresentazione grafica

Una classe viene rappresentata con uno schema grafico detto **diagramma delle classi,** che ne evidenzia il nome, gli attributi e i metodi. Il diagramma è costituito da un rettangolo diviso in tre zone tramite linee: in alto si indica il nome della classe che viene definita, nella zona centrale l'elenco degli attributi e in basso l'elenco dei metodi.



Gli oggetti e le classi

Con riferimento all'esempio precedente, la classe Automobile è rappresentata con il seguente diagramma della classe:



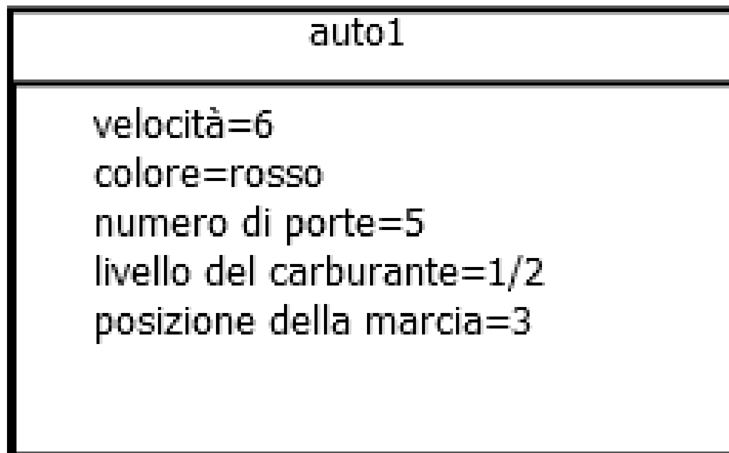
La classe può essere immaginata come uno stampo dal quale vengono creati più esemplari (oggetti), tutti con gli stessi attributi e gli stessi metodi.

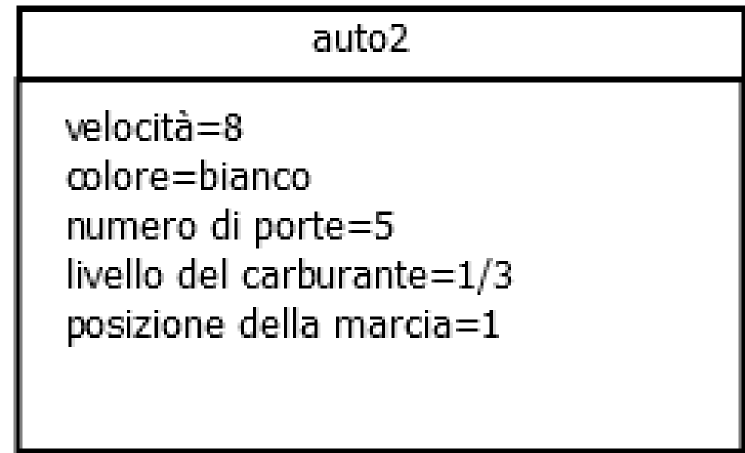
Nei programmi orientati agli oggetti, un oggetto può esistere solo se esiste la relativa classe che ne descrive le caratteristiche e le funzionalità.

Gli oggetti e le classi

Per utilizzare un oggetto occorre crearlo come esemplare della classe, cioè come **istanza** di una classe.

Creando due istanza auto1 e auto2 della classe Automobile, si ottengono due oggetti.





Per descrivere gli oggetti dell'esempio sopra è stata usata la notazione grafica, chiamata diagramma degli oggetti, che utilizza un rettangolo contenente nella parte alta il nome dell'oggetto e nella parte centrale l'elenco delle caratteristiche dell'oggetto con i rispettivi valori. Si noti che i metodi non vengono riportati nel diagramma, in quanto sono comuni a tutti gli oggetti della stessa classe.

UML: rappresentazione grafica di classi e istanze

I simboli utilizzati negli schemi grafici precedenti provengono dagli standard del linguaggio UML (Unified Modeling Language).
L'UML è una metodologia di analisi e progettazione a oggetti divenuta oramai uno standard di fatto nell'ambiente degli sviluppatori object-oriented: rappresenta un progetto software attraverso un formalismo grafico che fa uso di una serie di diagrammi, ciascuno dei quali può essere utilizzato per descrivere un particolare aspetto del sistema software che si vuole sviluppare.
Un'interessante caratteristica di UML è l'indipendenza dal linguaggio a oggetti utilizzabile per lo sviluppo dell'applicazione software.

L’incapsulamento

Tramite l’uso delle classi, tutto ciò che si riferisce a un certo oggetto è racchiuso e contenuto all’interno della classe stessa. Questo raggruppamento conferisce alla classe il significato di unità di programmazione, riutilizzabile in altri programmi. Questi aspetti descrivono il concetto di incapsulamento, che è uno dei concetti base della programmazione ad oggetti.

Il termine incapsulamento indica la proprietà degli oggetti di incorporare al loro interno sia gli attributi che i metodi, cioè le caratteristiche e i comportamenti degli oggetti.

In pratica si crea una capsula, una barriera concettuale, che isola l’oggetto dalle cose esterne. Si dice che gli attributi e i metodi sono incapsulati nell’oggetto.

Questa impostazione di raccogliere tutto quello che riguarda una singola entità all’internodi un oggetto è uno dei vantaggi offerti dalla programmazione orientata agli oggetti.

La dichiarazione e l’utilizzo di una classe

La struttura base della dichiarazione di una classe in Java è la seguente:

```
Class NomeClasse
{
   //attributi
  //metodi
}
```

L’utilizzo della classe all’interno del programma avviene attraverso la creazione delle sue istanze, cioè gli oggetti. La dichiarazione di un oggetto segue la normale sintassi della dichiarazione delle variabili:

```
NomeClasse nomeOggetto;
```

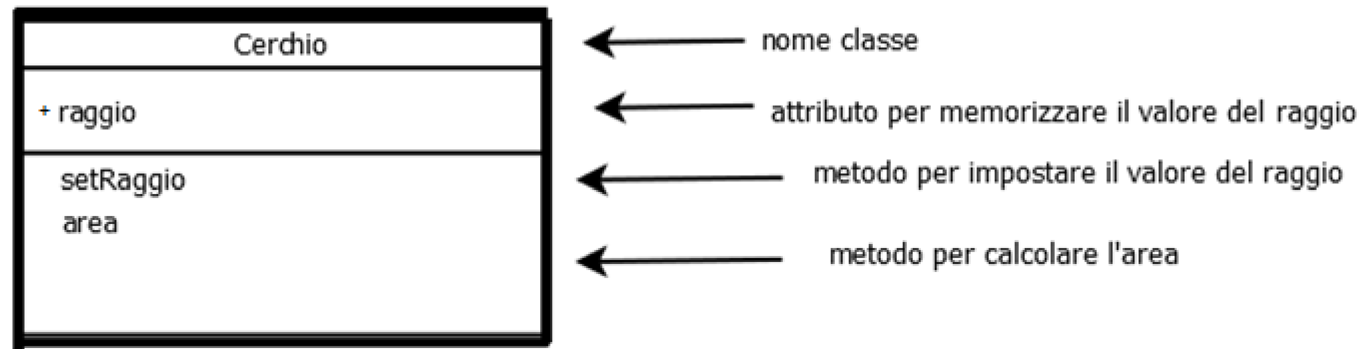
La creazione dell’istanza nomeOggetto si concretizza con l’uso della parola chiave new, seguita dal nome della classe e dalle parentesi tonde:

```
nomeOggetto = new NomeClasse();
```

Per convenzione, i nomi degli oggetti si indicano con la lettera iniziale minuscola.

Descrivere la classe cerchio

Descrivere la classe per il programma che calcola l’area di un cerchio conoscendone il raggio. L’operatore + indicata il livello di visibilità public.




```
Class Cerchio
{
  private double raggio;  //attributo
  public void setRaggio(double r)  //metodo
  {
     raggio=r;
  }
   public double area()  //metodo
  {
     return (raggio *raggio * Math.PI);
  }}
```

Descrivere la classe cerchio

Il codice precedente non può ancora essere utilizzato, in quanto manca il metodo principale main. Di seguito riportiamo il progetto completo.

```
package cerchio;
import java.util.Scanner;
public class Cerchio
{
  private double raggio;  //attributo
  public void setRaggio(double r)  //metodo
  {
     raggio=r;
  }
   public double area()  //metodo
  {
     return (raggio *raggio * Math.PI);
  }
  public static void main(String[] args)
  {
      Cerchio tavolo; //dichiarazione dell'oggetto
      tavolo=new Cerchio(); //crezione dell'istanza dell'oggetto
      tavolo.setRaggio(0.75); // i metodi della classe Cerchio sono richiamati con nomeOggetto.nomemetodo
      System.out.println("Area del tavolo="+tavolo.area());
   }}
```

I livelli di visibilità degli attributi

I livelli di visibilità di un attributo stabiliscono se l'attributo è accessibile da altre classi, cioè se dall'esterno della classe che lo contiene si può leggere o modifica il suo valore. I livelli di visibilità sono:

- **public**
- **private**
- **protected**

Con la visibilità **public** l'attributo è accessibile da qualsiasi altra classe.
Al contrario, l'uso di **private** permette di nascondere l'attributo all'interno dell'oggetto, in modo che non possa essere visto da nessun'altra classe: solo la classe che lo contiene può decidere di modificarne il valore.
Un attributo dichiarato come **protected** è visto all'esterno solo dalle classi che appartengono alla stessa libreria oppure dalle sottoclassi della classe in cui è stato dichiarato l'attributi.
Se non viene specificato alcun livello di visibilità, l'attributo è visibile solo dalle classi che appartengono alla stessa libreria (protected).

I livelli di visibilità degli attributi

In aggiunta ai livelli di visibilità, in fase di dichiarazione si possono specificare altre due caratteristiche per gli attributi:

Static

Final

La parola **static** indica un attributo legato alla classe, nel senso che, se vengono creati più oggetti di quella classe, esiste solo una copia dell’attributo. Una variabile statica, essendo condivisa da tutte le istanze della classe, assumerà lo stesso valore per ogni oggetto di una classe.

Un attributo è invece dichiarato final se lo si vuole rendere una costante, in modo che possa assumere un unico valore. Dopo aver ricevuto un valore iniziale, non può essere sottoposto a operazioni di assegnazione di valori diversi. Solitamente gli attributi final vengono anche dichiarati pubblici, perché non esiste il rischio che possano venire modificati dall’esterno in modo indesiderato.

Un esempio di attributo static

```
package classediesempio;
public class Classediesempio
{
   public static int a = 0;
   public static void main (String args[])
   {
      System.out.println("a = "+Classediesempio.a);
      Classediesempio ogg1 = new Classediesempio();
      Classediesempio ogg2 = new Classediesempio();
      ogg1.a = 10;
      System.out.println("ogg1.a = " + ogg1.a+ " ogg2.a = " + ogg2.a);
      ogg2.a =30;
      System.out.println("ogg1.a = " + ogg1.a+" ogg2.a ="+ ogg2.a);
   }
}
```

L’output generato sarà il seguente:

```
a = 0
ogg1.a = 10 ogg2.a = 10
ogg1.a = 30 ogg2.a =30
```

I livelli di visibilità dei metodi

I livelli di visibilità di un metodo, come per gli attributi, specificano se il metodo può essere visto e richiamato da altri oggetti. I livelli di visibilità che possono essere specificati sono:

- **public**
- **private**
- **protected**

Questi tre diversi livelli applicabili ai metodi hanno lo stesso significato già visto per i livelli riferiti agli attributi.

- Un metodo dichiarato con **public** può essere richiamato da qualunque altra classe;
- Un metodo **private** non può essere richiamato esternamente alla classe in cui è dichiarato;
- Un metodo dichiarato **protected** è visibile solo dalle classi che appartengono alla stessa libreria oppure dalle sottoclassi della classe in cui è dichiarato.

Solitamente i metodi vengono dichiarati **public**. Il livello private viene scelto quando un metodo è usato solo all'interno della classe.

La dichiarazione degli attributi

Gli attributi possono essere immaginati come fossero le variabili, con la differenza che vengono dichiarati nel blocco di una classe anziché all'interno del metodo main. Gli attributi vengono anche chiamati **variabili di istanza** o **variabili membro.**

La dichiarazione di un attributo è simile a quella di una variabile, con l'aggiunta dell'indicazione del livello di visibilità:

Esempio: **private int x,y,z;**

Private, come abbiamo già anticipato, permette di nascondere l'attributo all'interno della classe, in modo che non possa essere visto da nessun'altra classe: solo la classe che lo contiene può decidere di modificarne il valore.

Quindi i livelli di visibilità di un attributo stabiliscono se l'attributo è accessibile da altre classi, cioè se dall'esterno della classe che lo contiene si può leggere o modificare il suo valore.

La dichiarazione dei metodi

I metodi rappresentano la parte dinamica di una classe. Servono per implementare le operazioni che una classe può eseguire: i metodi possono modificare gli attributi, cioè lo stato interno di un oggetto, oppure calcolare i valori che verranno restituiti al richiedente.

Un metodo è caratterizzato da cinque diversi elementi che devono essere dichiarati:

- un nome;
- un blocco di istruzioni;
- un tipo di valore di ritorno;
- un elenco di parametri;
- un livello di visibilità.

La dichiarazione dei metodi deve essere fatta all'interno di un blocco di una classe e assume la segue te struttura generale:

```
LivelloDiVisibilità tipoRestituito nomeMetodo (parametri)
{
 //variabili
 // istruzioni
}
```

La dichiarazione dei metodi

Gi elementi obbligatori sono il **tipoRestituito** e il nome **nomeMetodo**, mentre il livello di visibilità e i parametri sono facoltativi e possono anche non essere presenti.

Esempio: il metodo seguente esegue la somma di due numeri interi passati come parametri, vediamone il frammento di codice.

```
public int addizione(int a, int b)
{
  int sum;   /*   variabile locale, viene creata quando viene eseguito il metodo e viene
                  distrutta quando termina l'esecuzione del metodo*/
  sum=a+b;
   return sum;
}
```

Valore di ritorno: nell'esempio sopra il valore di ritorno è int, indica quale sarà il valore restituito al termine dell'esecuzione del metodo. Un metodo può anche non restituire nessun valore di ritorno, in tal caso si una la parola chiave void. Per ritornare un valore il metodo deve terminare con l'istruzione return, che indica qual è il valore restituito.

L'elenco dei parametri: *in java i parametri formati da variabili semplici possono essere passati solo per valore*. Ogni volta che viene richiamato un metodo, i parametri assumono il ruolo di variabili locali. ***Le eventuali modifiche apportate ai parametri all'interno del metodo non vengono applicate ai parametri originari.***

La dichiarazione dei metodi

L'elenco dei parametri:

Se invece passiamo come parametro un array, allora viene creata una copia del riferimento e non dell'array. *L'array quindi non viene duplicato, ma viene creta una variabile locale che contiene la copia del riferimento allo stesso array*. ***Ne segue che le modifiche all'interno di un metodo, fatte usando il parametro, modificheranno anche l'array originale.***

Nell'esempio seguente il metodo raddoppia riceve come parametro un array di numeri interi e li raddoppia modificando direttamente l'array originale.

```
public void raddoppia(int valori[])
{
   for (int i=0;i<valori.length;i++)
   {
      valori[i]*=2;               //corrisponde all'istruzione    valori[i]=valori[i]*2
   }
}
```

La proprietà length restituisce la lunghezza del vettore di oggetti di nome valori.

Il metodo sopra è dichiarato **public** e quindi può essere richiamato da qualunque altra classe. Se invece dichiariamo un metodo private allora esso non può essere richiamato esternamente alla classe. Diversamente se dichiarato **protected**, il metodo è visibile solo alle classi che appartengono alla stessa libreria o alle sue sottoclassi.

La creazione degli oggetti ed il metodo costruttore

Come già anticipato, le classi non possono essere sfruttate direttamente, ma è necessario **creare un oggetto** , cioè un'istanza particolare di quella classe. Un'**istanza di classe** è in pratica una variabile (oggetto) che viene dichiarata indicando come tipo il nome della classe. Facendo un parallelismo tra le variabili e gli oggetti, si può dire che alle variabili è associato un tipo, mentre agli oggetti è associata una classe. La creazione di un oggetto può essere eseguita negli stessi punti dove vengono dichiarate le variabili e si compone dei seguenti passi:
- dichiarazione dell'oggetto;
- allocazione dell'oggetto.

La **dichiarazion**e dell'oggetto deve indicare il nome della classe seguito dal nome che si vuole assegnare all'oggetto, nel modo seguente: Cerchio tavolo;

L'**allocazione** dell'oggetto riserva lo spazio per memorizzare gli attributi dell'oggetto e viene codificato con l'operatore **new** seguito dal nome della classe.

Il metodo costruttore

Con l'allocazione dell'oggetto viene attivato in modo implicito un particolare metodo predefinito, detto **costruttore** della classe, che consente di creare un oggetto. Se una classe viene costruita senza specificare il costruttore, come fatto nelle pagine precedenti per la classe Cerchio, a essa viene associato un costruttore vuoto.

La creazione di un oggetto può essere raggruppata in una sola riga che riunisce la dichiarazione e l'allocazione, oltre al metodo costruttore.

Cerchio tavolo = new Cerchio();

Classe Cerchio Oggetto tavolo metodo costruttore Cerchio()

I metodi costruttori

Durante l'implementazione di una classe, oltre a definire attributi e metodi, si deve dunque considerare anche la creazione di uno o più metodi costruttore: essi vengono eseguiti automaticamente ogni volta che viene creato un nuovo oggetto e sono solitamente usati per le *operazioni di inizializzazione dell'oggetto*.

Nella definizione della classe Cerchio si può aggiungere il metodo costruttore Cerchio per l'assegnazione del valore iniziale per il raggio del cerchio.

```
package cerchio;
import java.util.Scanner;
public class Cerchio
{
  private double raggio;  //attributo
  public Cerchio(double  r)   //  costruttore
  {
       raggio=r;
  }
  public void setRaggio(double r)  //metodo
  {
     raggio=r;
  }
   public double area()  //metodo
  {
     return (raggio *raggio * Math.PI);
  }
  public static void main(String[] args)
  {
    Cerchio     tavolo=new     Cerchio(0.41);     /*crezione
          dell'oggetto          con          assegnazione
          parametro al costruttore*/
      System.out.println("Area del tavolo="+tavolo. area());
  }
}
```

L'uguaglianza tra oggetti

Il concetto di uguaglianza tra oggetti è diverso rispetto a quello di uguaglianza tra variabili:

•due **variabili** sono uguali se contengono lo stesso valore;

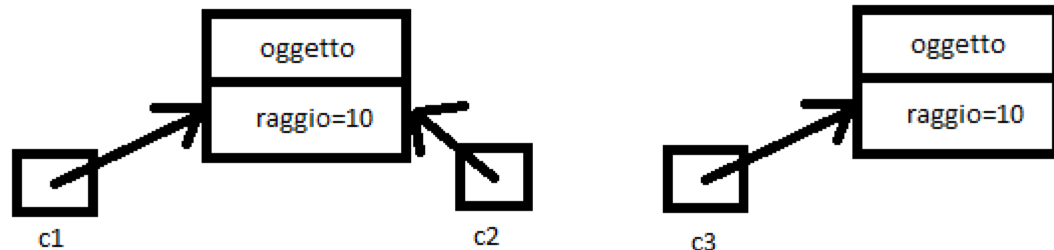
Esempio: int v1=10; int v2=10;

Le variabili v2 e v2 sono uguali: se si confrontano con l'operatore==viene restituito true.

•due **oggetti** sono uguali se contengono il riferimento alla stessa area di memoria.

```
Cerchio c1=new Cerchio(10);
Cerchio c2=new Cerchio(10);
Cerchio c3=c1;
```

Gli oggetti c1 e c2 sono diversi: se si confrontano con l'operatore==viene restituito false. Invece gli oggetti c1 e c3 sono uguali.



In sostanza il confronto tra gli oggetti, usando l'operatore ==, verifica **l'uguaglianza dei riferimenti** piuttosto che l'uguaglianza degli attributi dell'oggetto.

All'interno dei programmi, si possono creare più riferimenti allo stesso oggetto tramite l'operazione di assegnamento. Nell'esempio precedente l'oggetto c1 è stato assegnato all'oggetto c3. La **copia dei riferimenti** si verifica anche quando un oggetto viene passato come parametro a un metodo. Esempio: public void stampa(Cerchio c4) { …… }

Se viene passato il cerchio c1 con l'istruzione stampa(c1), sia c4 che c1 puntano alla stessa area di memoria.

L'utilizzo degli oggetti

Dopo aver creato l'oggetto, esso può essere utilizzato in due modi:

- accedendo e manipolando il valore dei suoi attributi;
- Invocando ed eseguendo i suoi metodi.

Si può accedere agli attributi di un oggetto usando l'operatore punto;

nomeoggetto.attributo

Con nomeoggetto si intende un'istanza di classe, che sia stata precedentemente dichiarata e allocata. L'operatore punto viene inserito tra l'oggetto e l'attributo. Quest'ultimo si riferisce a un attributo visibile e quindi dichiarato con il livello di visibilità **public** o **protected.** Gli attributi **private** non sono accessibili con questa modalità.

L'**invocazione di un metodo** viene eseguita usando l'operatore punto nel seguente modo:

nomeOggetto-metodo(parametri)

Nella programmazione orientata agli oggetti si usa il termine scambio di messaggi per indicare l'interazione tra gli oggetti realizzata tramite l'invocazione dei metodi.

Quando un oggetto vuole comunicare con un altro per eseguire un'azione, manda a esso un messaggio. Il nomeOggetto che precede il punto corrisponde al destinatario del messaggio, mentre il metodo deve essere uno dei metodi pubblici dichiarata nella classe del destinatario. Se il metodo richiede dei parametri, essi devono essere indicati alla fine tra parentesi tonde; se non ci sono parametri, le parentesi tonde vanno comunque indicate.

L'invocazione di un metodo, se restituisce un valore di ritorno, può essere posizionata alla destra di un'istruzione di assegnamento e in tutte le posizioni in cui vengono utilizzate le espressioni.   Esempio: superficie=tavolo.area();

La parola chiave this

In ogni classe è implicitamente presente un oggetto speciale identificato dalla parola chiave **this**. Questa parola chiave costituisce un riferimento all'oggetto medesimo, permettendo all'oggetto di riconoscere se stesso. Inoltre l'uso di this permette di risolvere le eventuali omonimie presenti tra i nomi degli attributi e i nomi dei parametri.

Il costruttore della classe Cerchio può essere scritto nel seguente modo:

```
//costruttore
public Cerchio(double raggio)
{
  this.raggio.raggio;
}
```

Il nome utilizzato nell'esempio per il parametro del costruttore, raggio, è identico al nome dell'attributo. L'uso dello stesso nome consente di non inventare nomi di variabili diversi, rendendo così più semplice la lettura del codice. Il parametro raggio, all'interno del costruttore, diventa una variabile locale che maschera l'attributo dichiarato con lo stesso nome. Per poter accedere all'attributo si devono usare **l'operatore punto** e la parola chiave **this**.

Gli attributi e i metodi static

Gli attributi e i metodi dichiarati con la parola **static** possiedono caratteristiche particolari che li differenziano dagli altri dichiarati non static. **Un attributo static** viene anche chiamato **variabile di classe**, perché è un attributo legato alla classe: se vengono creati più oggetti per quella classe, esiste solo una copia dell'attributo. Normalmente, per gli attributi non statici viene creata una copia per ogni oggetto istanziato. Invece un attributo statico è quindi condiviso da tutte le istanze della classe: tutte possono leggere e modificare l'attributo comune .

Un esempio di attributo statico è l'attributo PI, contenente il valore di pigreco e dichiarato nella classe Math, che fa parte delle librerie di Java:

public static final double PI; → attributo costante final e accessibile da chiunque (public)

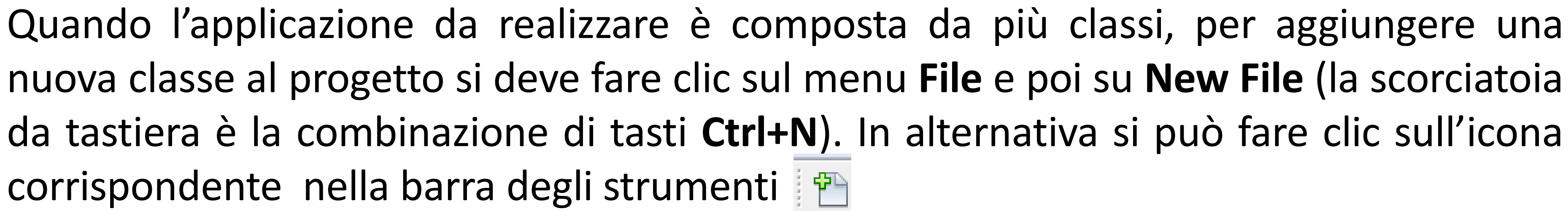
Un attributo stati è accessibile in un modo diverso dagli altri attributi. Normalmente gli attributi sono accessibili usando l'operatore punto preceduto dal **nome dell'oggetto** a cui ci si riferisce. *Gli attributi statici tuttavia possono essere indicati anche in un altro modo, usando l'operatore punto preceduto dal **nome della classe**. Perché gli attributi statici sono unici; per questo motivo si chiamano anche **variabili di classe.***

Ad esempio si può usare l'istruzione **Math.PI** , in quanto ci si riferisce all'attributo statico PI. Non è stato necessario creare un oggetto di classe Math per usare l'attributo.

I metodi statici sono anche chiamati **metodi classe** perché vengono invocati usando il nome della classe e non dell'oggetto. I metodi statici possono essere utilizzati senza creare in anticipo un'istanza della classe. I metodi statici possono operare solo su attributi statici e richiamare altri metodi statici. Un esempio è il metodo random, che è un metodo presente nella classe Math e dichiarato nel seguente modo: public static double random()

E' un metodo dichiarato pubblico e statico e restituisce un valore di tipo double. Per richiamare questo metodo statico basta far precedere all'operatore punto il nome della classe**: Math.random().**

L'applicazione con più classi in ambiente Netbeans

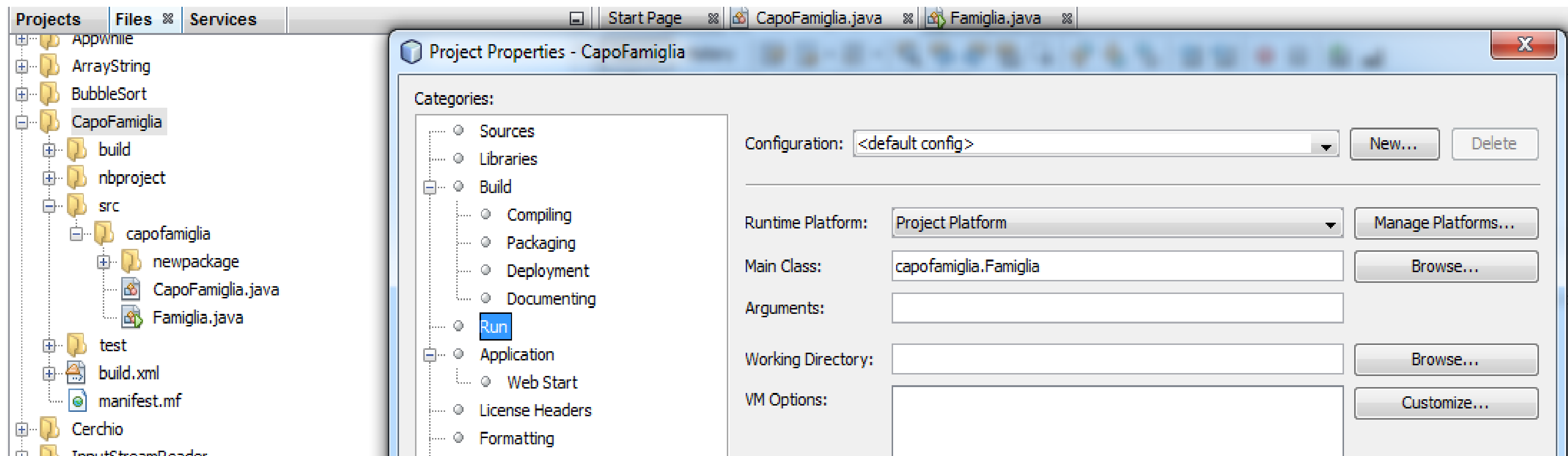
Quando l'applicazione da realizzare è composta da più classi, per aggiungere una nuova classe al progetto si deve fare clic sul menu **File** e poi su **New File** (la scorciatoia da tastiera è la combinazione di tasti **Ctrl+N**). In alternativa si può fare clic sull'icona corrispondente nella barra degli strumenti 

Viene aperta una finestra in cui si deve scegliere il tipo di file **Java Class**: fare clic su *Next* poi inserire il nome della classe nella casella *Class Name* e infine fare clic su *Finish*.

In presenza di più classi in un progetto java standard, occorre impostare la classe main, cioè la classe da cui far partire l'esecuzione.

Per fissare la classe main, occorre fare clic con il tasto destro del mouse sul nome del progetto nella finestra **Projects** e scegliere **Properties.**

Nella finestra che si apre, clic su **Run** e inserire il nome della classe nella classe **Main Class.**



Un esempio di applicazione con più classi

Esempio –
Si vuole creare un progetto in java contenente una classe **CapoFamiglia** formata da tre variabili membro **(nome, cognome e datadinascita**), un costruttore **CapoFamiglia(String sNome, String sDataN)** ed una funzione membro **Stampa().**
Si chiede di implementare la classe ***CapoFanmiglia*** in modo da consentire, tramite il metodo costruttore ***CapoFamiglia(parametri)*** sopra indicato, il passaggio di parametri sia del nome che della data di nascita. La classe ***CapoFamiglia*** dovrà contenere anche un metodo ***Stampa()*** che permette di stampare il nome, il cognome e la data di nascita. Creare successivamente una classe di nome ***Famiglia*** contenente il metodo main. La classe ***Famiglia*** dovrà istanziare quattro oggetti **padre, madre, figlio1 e figlio2.** Si precisa che la variabile membro *cognome*, all’interno della classe CapoFamiglia, è di tipo static: questo permette all’istruzione **CapoFamiglia.Cognome =“Mello”**, presente nella classe Famiglia, di inserire un unico cognome valido per il padre e figli.
Si riporta il codice sorgente nelle prossime slide.
Prima di provare il programma, così come indicato nella slide precedente, si ricorda di indicare il nome del file contenente il metodo main: nel nostro caso il file da indicare è **Famiglia.java**, stringa da inserire nella voce *Main Class* della categoria ***Run*** **.**

Implementazione della classe CapoFamiglia.java

```
// File CapoFamiglia.java
package capofamiglia;
public class CapoFamiglia
{
    private String Nome;
    public static String Cognome;  //dichiarato di tipo static
    private String Datanascita;
    // metodo costruttore CapoFamiglia
    CapoFamiglia(String sNome, String sDataN)
    {
        Nome = sNome;
        Datanascita = sDataN;
    }
   // metodo Stampa
    public void Stampa()
    {
         System.out.println(Nome + " " +Cognome + " è nato/a il " + Datanascita);
    }
}
```

Implementazione della classe Famiglia.java

```
package capofamiglia;
public class Famiglia {
  public static void main(String[] args)  {
    CapoFamiglia padre = new
        CapoFamiglia("Roberto", "26/01/1966");
    CapoFamiglia madre = new
        CapoFamiglia("Enza", "25/01/1967");
    CapoFamiglia figlia1 = new
        CapoFamiglia("Francesca", "13/06/1996");
    CapoFamiglia figlia2 = new
        CapoFamiglia("Giulia", "22/06/1997");
    CapoFamiglia.Cognome = "Mello";
    padre.Stampa();
    figlia1.Stampa();
    figlia2.Stampa();
    CapoFamiglia.Cognome = "Velardi";
    madre.Stampa();
  }
}
```

La chiusura del progetto e la cartella build e dist

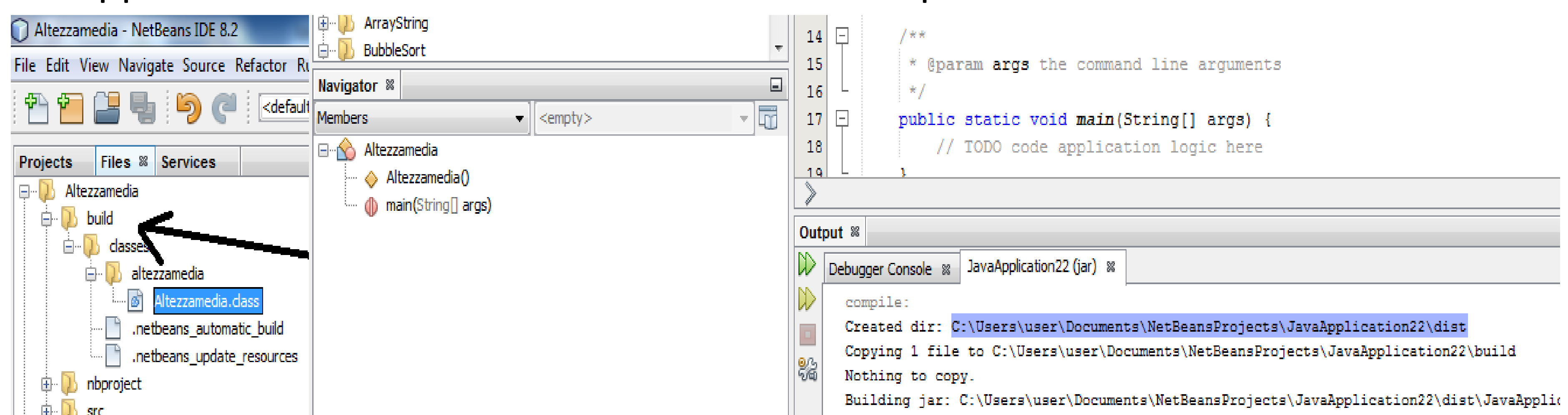
Per chiudere il progetto aperto nell'ambiente di sviluppo NetBeans, dopo aver salvato il lavoro, fare clic su **File** e poi su **Close Project**.

Le cartelle build e dist

Le operazioni di compilazione e di collaudo dell'esercizio creano , all'interno della cartella del progetto java, due sottocartelle **build** e **dist**, con i file necessari per eseguire l'applicazione direttamente dal sistema operativo:

- la cartella **build** contiene i file di lavoro utilizzati per creare il programma eseguibile (nella cartella si troverà il file con estensione .class
- La cartella **dist** è l'insieme dei file da distribuire agli utenti finali.

Nella cartella **dist** viene automaticamente creato il file **JAR (Java Archive)** avente lo stesso nome del progetto e l'estensione .jar, che consente di eseguire l'applicazione dalla linea comandi del sistema operativo.



Il mascheramento dell'informazione nelle classi –L'information hiding

Come già visto nei paragrafi precedenti, il livello di visibilità degli attributi e dei metodi permette di identificare ciò che rientra nella sezione pubblica (public) e ciò che rientra nella sezione privata (livello private).

L'information hiding

Il termine information hiding indica il mascheramento delle modalità di implementazione di un oggetto, rendendone disponibili all'esterno solo le funzionalità.

L'applicazione pratica dell'information hiding, secondo cui devono essere tenuti nascosti dettagli implementativi della classe, consiste nel dichiarate gli attributi usando il livello di visibilità **private**. In questo modo le strutture di un oggetto risultano nascoste e non sono direttamente accessibili per la lettura e la modifica.

Una classe che possiede attributi dichiarati come private può comunque permettere la loro lettura e modifica definendo opportuni metodi. In particolare la modifica può essere gestita tramite i metodi con livello di visibilità *public,* all'interno dei quali si possono inserire gli opportuni controlli per validare le modifiche richieste.

Solitamente i due metodi per leggere e modificare il valore degli attributi vengono indicati con il termine inglese **get** e **set**.

Vediamo nella prossima slide un esempio.

Il mascheramento dell’informazione nelle classi –L’information hiding

Come già visto nei paragrafi precedenti, il livello di visibilità degli attributi e dei metodi permette di identificare ciò che rientra nella sezione pubblica (public) e ciò che rientra nella sezione privata (livello private).

L’information hiding

Il termine information hiding indica il mascheramento delle modalità di implementazione di un oggetto, rendendone disponibili all’esterno solo le funzionalità.

L’applicazione pratica dell’information hiding, secondo cui devono essere tenuti nascosti dettagli implementativi della classe, consiste nel dichiarate gli attributi usando il livello di visibilità **private**. In questo modo le strutture di un oggetto risultano nascoste e non sono direttamente accessibili per la lettura e la modifica.

Una classe che possiede attributi dichiarati come private può comunque permettere la loro lettura e modifica definendo opportuni metodi. In particolare la modifica può essere gestita tramite i metodi con livello di visibilità *public,* all’interno dei quali si possono inserire gli opportuni controlli per validare le modifiche richieste.

Solitamente i due metodi per leggere e modificare il valore degli attributi vengono indicati con il termine inglese **get** e **set**.

Vediamo nella prossima slide un esempio.

```
class Automobile
{
   private final int POSIZIONE_MAX=6;
   private int posMarcia;
   public Automobile()
   {
      posMarcia=1;
   }
   public int getMarcia()
   {
      return posMarcia;
   }
   public void setMArcia(int m)
   {
      if ((m>=1) && (m<=POSIZIONE_MAX))
      {
         posMarcia=m;
      }
   }
}
```

Il mascheramento dell'informazione nelle classi –L'information hiding

Il metodo getMarchia dichiara come valore di ritorno un intero e ha il compito di restituire il valore dell'attributo posMarchia. Poiché l'attributo è private e non può essere visibile al di fuori della classe, l'uso del metodo pubblico consente la lettura del valore dell'attributo anche all'esterno.

Il metodo setMarcia riceve come parametro un valore intero e lo assegna all'attributo posMarcia. Il suo compito è quello di modificare il valore dell'attributo solo se il nuovo valore rispecchia certi criteri di validità. In questo modo non viene modificato direttamente il valore dell'attributo, che viene invece cambiato indirettamente attraverso l'esecuzione di un metodo.

L'interfaccia della classe

Nella programmazione ad oggetti si utilizza spesso anche il termine interfaccia riferito alle classi.

L'interfaccia di una classe indica l'elenco dei metodi pubblici, cioè l'insieme delle funzionalità utilizzabili dalle istanze della classe.

L'interfaccia della classe Automobile è rappresentata dai tre metodi seguenti:

public Automobile(); public int getMarcia(); public void setMarcia(int m)

I programmi che vogliono usare la classe Automobile devono conoscere la sua interfacci; in particolare hanno bisogno di sapere il nome dei metodi pubblici, il tipo dei valori di ritorno, il numero e il tipo dei parametri.

La realizzazione di programmi

Un programma object-oriented scritto in Java è solitamente composto da più classi. Anche se è possibile fare diversamente, conviene salvare su disco ogni classe in un file separato. In questo modo, a ogni classe viene associato un file con estensione **.java** che, una volta compilato, genera il corrispondente file **.class**.

Tra tutte le classi di un programma, una deve contenere il metodo **main**, che è il primo metodo attivato dall'interprete durante l'esecuzione del programma. Se un programma è composto da più classi, l'interprete Java deve essere richiamato indicando il nome della classe con il metodo main.

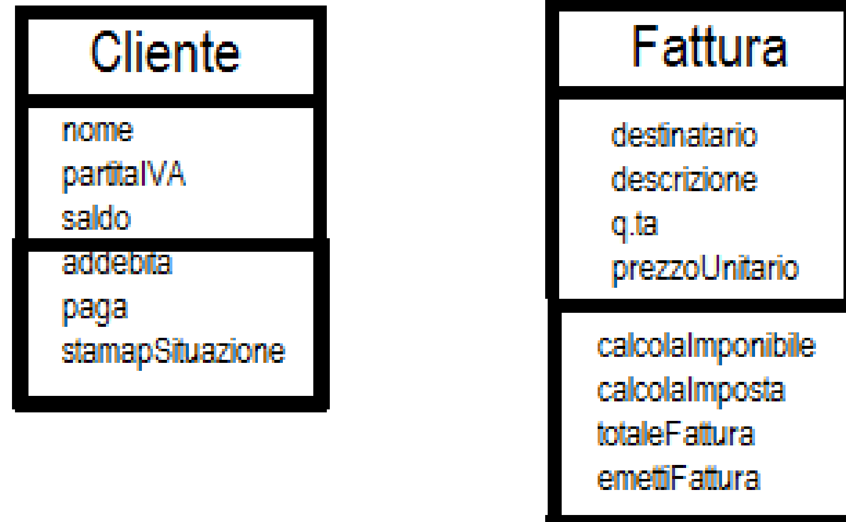
Creiamo un programma per la **gestione della fatturazione**, addebitando le fatture ai clienti e accreditando i pagamenti ricevuti.

Il problema richiede la descrizione di un sistema di fatturazione che, per semplicità, verrà limitato al caso di emissione della fattura a un cliente per un solo prodotto. Le entità rilevanti per questa descrizione sono il Cliente e la Fattura. I clienti sono descritti con i loro dati anagrafici e il saldo dei loro pagamenti. Il saldo aumenta se al cliente viene addebitata una fattura, al contrario diminuisce se il cliente esegue un pagamento. Il totale della fattura viene calcolato sommando l'imponibile (prezzo unitario*quantità) e l'imposta (imponibile * aliquota IVA).

I diagrammi delle classi Cliente e Fattura sono riportati nella prossima slide .

La realizzazione di programmi

I diagrammi delle classi Cliente e Fattura sono i seguenti:



L'implementazione delle classi Cliente e Fattura e un programma di esempio, composto da una classe contenete il metodo main, sono riportati di seguito.

Realizzazione file progFattura.java

```
package progfattura;
import java.io.*;
public class ProgFattura {
    public static void main(String[] args) {
        //creazione del cliente
        Cliente computerSpa=new Cliente("Computer SPA","0112121222");
        //crea due fatture per lo stesso cliemte
        Fattura ordine1=new Fattura(computerSpa);
        Fattura ordine2=new Fattura(computerSpa);
   // registra i dati delle fattue e le addebita
   ordine1.descrizione="mouse";
   ordine1.qta=28;
   ordine1.prezzoUnitario=17.5;
   ordine1.emettiFattura();
   ordine2.descrizione="monitor";
   ordine2.qta=38;
   ordine2.prezzoUnitario=107.55;
   ordine2.emettiFattura();
   //incassa il pagamento di un bonifico
   double bonifico =1000.0;
   System.out.println("Pagamento="+bonifico);
   computerSpa.paga(bonifico);
   computerSpa.stampaSituazione();
    }}
```

Realizzazione del file Cliente.java

```
package progfattura;
public class Cliente {
   private String nome;
   private String partitaIva;
   private double saldo;
   public Cliente(String nome, String partitaIva)    {
      this.nome=nome;
      this.partitaIva=partitaIva;
      saldo=0;    }
   public void addebita(double importo)
   {
      saldo+=importo;
   }
   public void paga(double importo)
   {
       saldo-=importo;
   }
   public void stampaSituazione()
   {
      System.out.println("----------------------");
      System.out.println("Clietne   ="+nome);
      System.out.println("Partita IVA  ="+partitaIva);
      System.out.println("Saldo     ="+saldo);
      System.out.println("----------------------");
      } }
```

Realizzazione file Fattura.java

```
package progfattura;
public class Fattura {
   private final int percIva=22; //costante
   private Cliente destinatario; //attributi
   public String descrizione;
   public int qta;
   public double prezzoUnitario;
   public Fattura(Cliente dest) {
      destinatario=dest;    }
   private double calcolaImponibile()    {
      return (qta*prezzoUnitario);    }
   private double calcolaImposta()       {
      double imp=calcolaImponibile();
      return (imp*percIva)/100;    }
   public double totaleFattura() // calcola il totale fattura    {
      double totale;
      totale=calcolaImponibile()+calcolaImposta();
      return totale;    }
   public void emettiFattura()    {
      double totale=totaleFattura();
      System.out.println("Totale fattura ="+totale);
      //addebita la fattura al cliente
      destinatario.addebita(totale);
      destinatario.stampaSituazione();    }}
```

Gli array di oggetti

La dichiarazione e l'allocazione di un array di oggetti funzionano nello stesso modo con cui vergono dichiarati e allocati gli array basati sui tipi predefiniti.
Con riferimento alla classe Cerchio utilizzata negli esempi precedenti, la dichiarazione di un array di 10 cerchi è la seguente:

Cerchio collezione[]=new Cerchio[10];

Dopo la dichiarazione dell'array si devono allocare i singoli oggetti. Per ogni oggetto della collezione si deve predisporre lo spazio in memoria.

collezione[0]=new Cerchio(0.58);
collezione[1]=new Cerchio(4,71);
collezione[i]=new Cerchio (3.58);

L'uso degli array è utile nelle situazioni in cui si devono creare molti oggetti della stessa classe e li si vuole gestire in maniera efficace. Hanno però una limitazione : la dimensione deve essere prestabilita e non è più possibile modificarla, a meno di creare un nuovo array. In ogni caso il linguaggio java mette a disposizione, tra le sue librerie, una classe che gestisce i vettori di dimensione variabile (le strutture dinamiche) che vedremo più avanti; questa classe rende più flessibile la manipolazione delle collezioni di oggetti. Il prossimo programma implementa una classe Harddisk con l'utilizzo di un array di oggetti. Ad ogni modello vengono assegnati dei valori e alla fine verrà calcolato, tramite un punteggio, il peggiore ed il migliore modello con la stampa del punteggio.

Implementazione di array di oggetti – File Proghd.java

```
package proghd;
public class Proghd {
  public static void main(String[] args) {
      final int MAX_HD=5;  //costante
      int totalePunteggio=0; //variabili
      int punteggioMedio=0;
      Harddisk peggiore,migliore;  //oggetti
      Harddisk hd[]=new Harddisk[MAX_HD];
//array di oggetti
      for(int i=0;i<hd.length;i++)
      {
         hd[i]=new Harddisk(); //crea gli hard
disk
         hd[i].leggiDati(i);  //legge i dati
      }
      peggiore=hd[0];
      migliore=hd[0];
//calcola il migliore-peggiore e il punteggio
medio
for(int i=0;i<hd.length;i++)
{   totalePunteggio+=hd[i].punteggio();
 if (hd[i].punteggio()<peggiore.punteggio())
   {        peggiore=hd[i];         }
else
if
(hd[i].punteggio()>migliore.punteggio(
))
        {            migliore=hd[i];        }

      }

punteggioMedio=totalePunteggio/MA
X_HD;
      System.out.println("\nPunteggio
medio ="+punteggioMedio);
      System.out.println("\n***
HardDidk migliore ***=");
      migliore.stampaDati();
      System.out.println("\n***
HardDisk peggiore ***");
      peggiore.stampaDati();
         }}
```

Implementazione di array di oggetti – File Harddisk.java

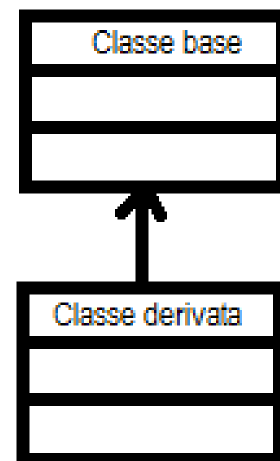
```
package proghd;
import java.io.*;
public class Harddisk {
   //attributi
   private String marca;
   private int rpm;
   private double accesso;
   private double capacita;
   //costanti per il calcolo del punteggio
   private final int PUNTI_RPM=1;
   private final int PUNTI_ACCESSO=-200;
   private final int PUNTI_CAPACITA=500;
   //legge gli attributi dell'hard disk
   public void leggiDati(int i)
   {
      InputStreamReader input=new InputStreamReader(System.in);
      BufferedReader tastiera=new BufferedReader(input);
      System.out.println("\nHARD DISK "+i);
      System.out.println("inserire marca: ");
      try
      {
        marca=tastiera.readLine();
      }
      catch(IOException e){}
      System.out.print("Inserire la velocità (rpm):");
      try
      {
         String numeroLetto =tastiera.readLine();
         rpm=Integer.valueOf(numeroLetto).intValue();
      }
catch (Exception e)
      {
         System.out.println("\nVelocità non corretta!");
         System.exit(1); //viene invocato il metodo statico exir della classe
System; provoca
      }
            //l'interruzione dell'esecuzione del programma. Il parametro passato
      System.out.print("Inserisci il tempo di accesso in ms:"); /*indica il codice di stato
che viene restituito al sistema operativo.*/
      try                /*per convenzione un valore diverso da zero indica che
l'esecuzione*/
      {     /*è terminata con un errore.*/
        String numeroLetto=tastiera.readLine();
        accesso=Double.valueOf(numeroLetto).doubleValue();
      }
      catch (Exception e)
      {
        System.out.println("\nTempo di accesso non corretto!");
        System.exit(1);
      }
      System.out.print("Inserisci la capacità in GB:");
      try
      {     /*è terminata con un errore.*/
        String numeroLetto=tastiera.readLine();
        capacita=Double.valueOf(numeroLetto).doubleValue();
      }
      catch (Exception e)
      {
        System.out.println("\ncapacità non corretta!");
        System.exit(1);
      }
   }
   //formatta i dati e li stamap su standard output
   public void stampaDati()
   {
        System.out.println("Marca            ="+marca);
        System.out.println("Velocità         ="+rpm+ " rpm");
        System.out.println("Tenpo di accesso   ="+accesso + " ms");
        System.out.println("Capacità         ="+capacita+ " GB");
        System.out.println("Punteggio         ="+punteggio());
   }
     //restituisce il punteggio assegnato a questo HD
   public int punteggio()
   {
        int punt=0;
        punt+= (int) (rpm *PUNTI_RPM);
        punt+= (int) (accesso*PUNTI_ACCESSO);
        punt+= (int) (capacita*PUNTI_CAPACITA);
        return punt;   } }
```

L’ereditarietà

Sviluppando un sistema ad oggetti significa trovare e definire le classi necessarie per risolvere il problema, dalle quali verranno creati gli oggetti. Non sempre occorre partire dal nulla nel costruire una classe, soprattutto se si dispone già di una classe che è simile a quella chi si vuole costruire. In questo caso si può pensare di prendere la classe esistente e di estenderla per adattarla alle nuove necessità**.**

L’ereditarietà è un concetto fondamentale della programmazione ad oggetti e rappresenta la possibilità di creare nuove classi a partire da una classe già esistent**e (classe base).** La nuova classe eredita tutti gli attributi e i metodi dalla classe base e può essere arricchita con nuovi attributi e nuovi metodi. La classe così ottenuta si chiama **classe derivata.**

La derivazione di una classe si rappresenta con i diagrammi di classe, UML, attraverso i due rettangoli delle classi uniti da una frecci a che parte dalla classe derivata e si dirige verso la classe base. Il concetto di ereditarietà è usato anche in biologia: è la capacità che hanno gli organismi di trasmettere i loro caratteri ai discendenti. Nella programmazione ad oggetti, gli organismi rappresentano le classi e i caratteri che vengono trasmessi sono gli attributi e i metodi.



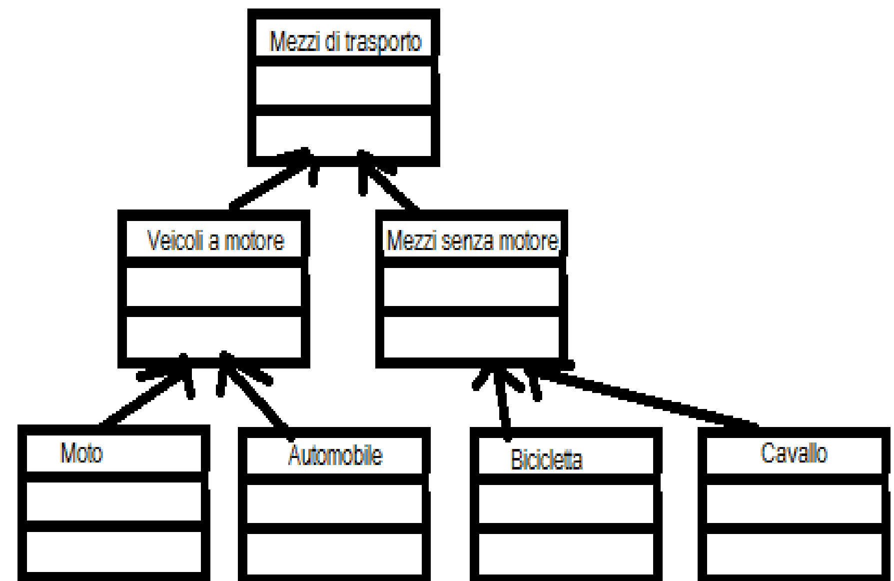
L’ereditarietà

La gerarchia.

La classe che è stata derivata da un’altra usando l’ereditarietà prende il nome di **sottoclasse.** La classe generatrice di una sottoclasse si chiama **superclasse** o sopraclasse. Queste relazioni tra le classi individuano una **gerarchia di classi** che nasce da un processo di specializzazione. Infatti le classi che si trovano in cima alla gerarchia sono le più generiche e man mano che si scende si trovano classi più specializzate.

La gerarchia delle classi si può descrivere graficamente usando il grafo di gerarchia: esso è costituito da un grafo orientato in cui i nodi sono rappresentati dalle classi, o meglio dai diagrammi delle classi, e le frecce individuano la presenza di una relazione di ereditarietà. La direzione della freccia è dalla sottoclasse verso la sopraclasse.

Riportiamo un esempio di grafo di gerarchia, dove la classe *Automobile* è sottoclasse della classe *Veicoli a motore*, che a sua volta è sottoclasse di *Mezzi di trasporto.* In alto ci sono le classi più generiche, che diventano più specializzate man mano che si scende lungo la gerarchia.



L’ereditarietà

La gerarchia.

LA sottoclasse eredita dalla sopraclasse tutti gli attributi ed i metodi, evitando di ripetere la descrizione degli elementi comuni nelle sottoclassi. L’ereditarietà consente quindi di condividere ciò che è simile tra le classi, con la possibilità di inserire le differenze. La nuova classe si differenzia dalla sopraclasse in due modi:

- **per estensione,** quando la sottoclasse aggiunge nuovi attributi e metodi che si sommano a quelli ereditati;
- **per ridefinizione,** quando la sottoclasse ridefinisce i metodi ereditati. In pratica viene data un’implementazione diversa di un metodo: si crea un nuovo metodo che ha lo stesso nome del metodo ereditato da una sopraclasse, ma che ha funzionalità diverse. Quando a un oggetto della sottoclasse arriva un messaggio che richiama un metodo ridefinito, viene eseguito il nuovo codice e non quello che era stato ereditato.

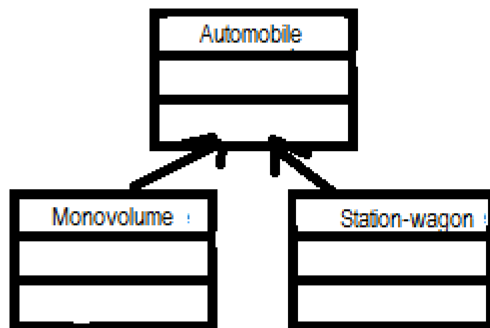
Una sottoclasse ridefinisce un metodo perché vuole associare un comportamento diverso oppure perché vuole utilizzare un algoritmo più efficiente rispetto a quello della sopraclasse. Quando una sottoclasse ridefinisce un metodo si dice che lo sovrascrive **(overriding del metodo)** utilizzando lo stesso nome usato dalla sopraclasse. In sostanza, l’ereditarietà serve a definire nuove classi, derivate da altre, che ereditano gli attributi e i metodi, con la possibilità di ridefinirli o di aggiungerne di nuovi.

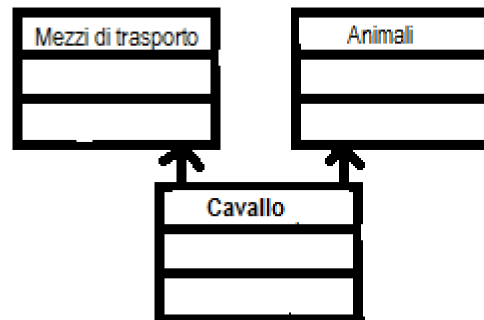
L'ereditarietà singola e multipla

Esistono due tipi di ereditarietà: singola e multipla.

L'**ereditarietà singola** si verifica quando una sottoclasse deriva da un'unica sopraclasse. Usando i diagrammi delle classi, l'ereditarietà singola è indicata con un'unica freccia che esce dalla sottoclasse ed è diretta verso la sopraclasse. La sottoclasse ha una sola classe generatrice, ma questo non vieta alla sopraclasse di essere ereditata da più classi.

L'**ereditarietà multipla** , invece, comporta che una classe derivi da due o più sopraclassi. In questo modo si possono fondere insieme gli attributi e i metodi provenienti da classi diverse per creare una nuova. Usando il diagramma delle classi riportato a destra, l'ereditarietà multipla è indicata con due o più frecce che escono dalla sottoclasse e sono dirette verso le sopraclassi.





Java prevede solo l'ereditarietà singola. L'ereditarietà è uno strumento che permette il riutilizzo del software creato.

La dichiarazione e l'utilizzo di una sottoclasse

In java, così come già anticipato, è presente solo **l'ereditarietà singola**, cioè ogni classe può avere la massimo una sopraclasse.
Una sottoclasse si dichiara aggiungendo nell'intestazione il nome della sopraclasse do la parola chiave **extends:**

```
Class NomeSottoClasse extends NomesopraCLasse
{
  ….
  ….
}
```

Dopo la parola chiave ***extends*** può apparire solo il nome di una classe. Per il resto la sottoclasse mantiene la struttura di una classe normale. La sottoclasse eredita gli attributi ed i metodi definiti nella sopraclasse a esclusione di quelli che sono stati dichiarati ***privati.*** Questo significa che gli oggetti creati come istanza di una sottoclasse possono accedere anche agli attributi e ai metodi che vengono ereditati dalla sopraclasse.

La dichiarazione e l'utilizzo di una sottoclasse

Si supponga di aver definito le seguenti due classi:

```
class A
{
  public int numA;
}
 class B extends A
{
  public int numB;
}
```

Un oggetto creato come istanza della classe B nel seguente modo_

```
B oggB=new B();
```

Può accedere sia all'attributo numB che all'attributo numA, ereditato dalla sopraclasse A, con le seguenti istruzioni:

```
oggB.numB=10;
oggB.numA=5;
```

Quanto detto vale anche per i metodi ereditati. I membri di una classe dichiarati come *private* non sono ereditati dalle sottoclassi. Solo gli attributi e i metodi che sono dichiarati *protected* o *public vengono ereditati. Anche i costruttori e gli static non vengono ereditati. Per accedere e modificare gli attributi provati nelle sottoclassi si possono creare i metodi pubblici get e set nella sopraclasse, che sono ereditati dalla sottoclasse.*

Il volume del cilindro

Descrivere la classe per il programma che calcola il volume di un cilindro utilizzando la classe Cerchio già esistente. Con riferimento alla classe Cerchio già sviluppata, si vuole creare una classe per il cilindro come estensione della classe Cerchio. Il cilindro eredita gli attributi e i metodi del cerchio e in aggiunta definisce gli elementi propri. Questi ultimi includono l'attributo altezza e i metodi *setAltezza e volume.*

Il costruttore della classe Cilindro riceve come parametri il raggio del cerchio di base del cilindro e l'altezza e viene definito nel seguente modo:

```
public Cilindro(double raggio, double altezza)
{
  super(raggio);
 this.altezza=altezza;
}
```

La parola chiave **super** è un oggetto speciale che fa riferimento alla sopraclasse della classe in cui viene utilizzata; in questo caso si riferisce alla classe Cerchio. In particolare, l'uso di super all'interno del costruttore serve per richiamare il costruttore della sopraclasse Cerchio e, tra parentesi, viene indicato il parametro raggio.

Successivamente è riportato il programma completo in linguaggio Java, che mostra la codifica della classe derivata e dei suoi metodi, oltre ad alcuni esempi di utilizzo del nuovo oggetto.

Il volume del cilindro

Il costruttore della classe Cilindro riceve come parametri il raggio del cerchio di base del cilindro e l'altezza e viene definito nel seguente modo:

```
public Cilindro(double raggio, double altezza)
{   super(raggio);
    this.altezza=altezza;
}
```

La parola chiave **super** è un oggetto speciale che fa riferimento alla sopraclasse della classe in cui viene utilizzata; in questo caso si riferisce alla classe Cerchio. In particolare, l'uso di super all'interno del costruttore serve per richiamare il costruttore della sopraclasse Cerchio e, tra parentesi, viene indicato il parametro raggio.

Successivamente è riportato il programma completo in linguaggio Java, che mostra la codifica della classe derivata e dei suoi metodi, oltre ad alcuni esempi di utilizzo del nuovo oggetto.

Prima di passare al progetto, è doveroso descriviamo il concetto di **Modularità**.

Per ***Modularità*** intendiamo il partizionamento del nostro problema in tante parti, pertanto la logica è quella di scomporre il nostro progetto in varie classi. Una classe, così concepita, è vista come una componente di un sistema più vasto. Per indicare al sistema che esiste un raggruppamento di più classi sotto un medesimo dominio si usa il **packages**. Un **package** può essere definito come uno strumento utile per raggruppare classi legate fra di loro. **L'istruzione package** deve essere inserita come prima istruzione all'inizio del file, inoltre all'interno di ogni file possiamo inserire un'unica dichiarazione del package. Un package va pensato come un percorso **(path)** di una struttura ad albero di directory in cui sono definite le classi e in cui verranno creati i files .class compilati.

File ProgCilindro.java

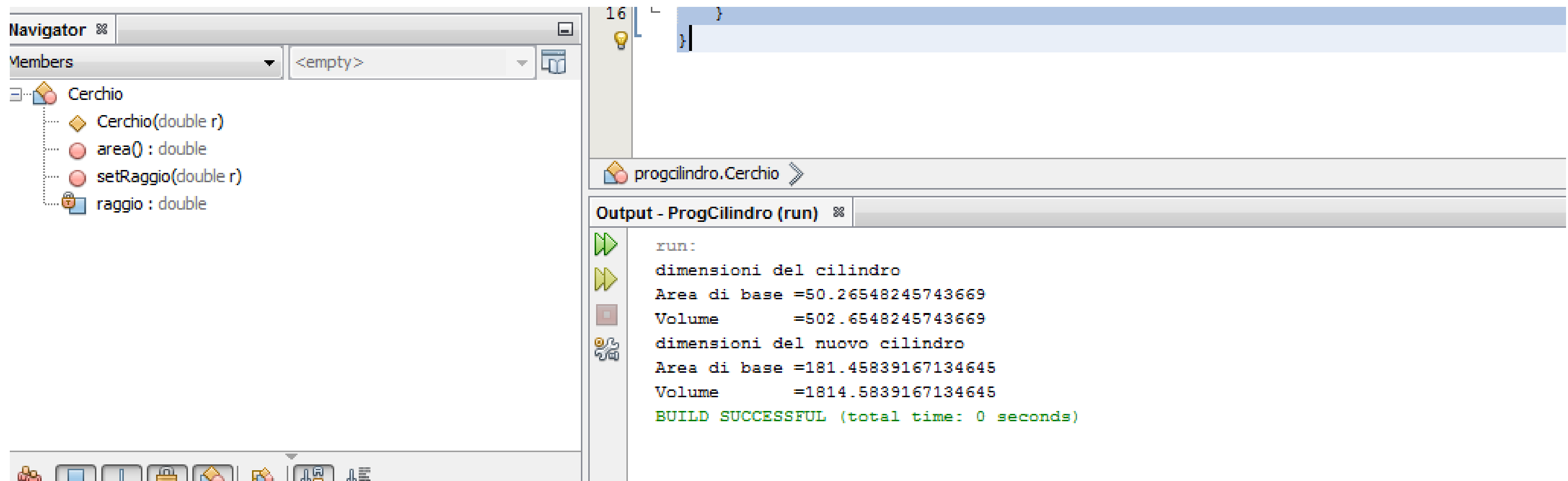
```
package progcilindro;
public class ProgCilindro {
    public static void main(String[] args) {
       //istanza dell'oggeto
       Cilindro cil=new Cilindro(4.0,10.0);
      //Cilindor(4.0,10.0) richiama il costruttore della classe Cilindro
       System.out.println("dimensioni del cilindro");
       System.out.println("Area di base ="+cil.area());
       System.out.println("Volume      ="+cil.volume());
       //modifica il cilindro usando il metodo ereditato
       cil.setRaggio(7.6);
       cil.setAltezza(23.5);
       System.out.println("dimensioni del nuovo cilindro");
       System.out.println("Area di base ="+cil.area());
       System.out.println("Volume      ="+cil.volume());
   }
}
```

File Cilindro.java

```
package progcilindro;
public class Cilindro extends Cerchio{
   private double altezza;
   public Cilindro(double raggio, double altezza)
   {
      super(raggio);
      this.altezza=altezza;
   }
   public void setAltezza(double alterzza)
   {
      this.altezza=altezza;
   }
   public double volume()
   {
      //usa il metodo ereditato
      //il metodo volume usa il metodo area
      //ereditato dalla classe Cerchio
      double vol=area()*altezza;
      return vol;
   }
}
```

File Cerchio.java

```
package progcilindro;
public class Cerchio
{
  private double raggio;  //attributo
  public Cerchio(double  r)   //  costruttore
  {
       raggio=r;
  }
  public void setRaggio(double r)  //metodo
  {
     raggio=r;
  }
   public double area()  //metodo
  {
     return (raggio *raggio * Math.PI);
  }
}
```

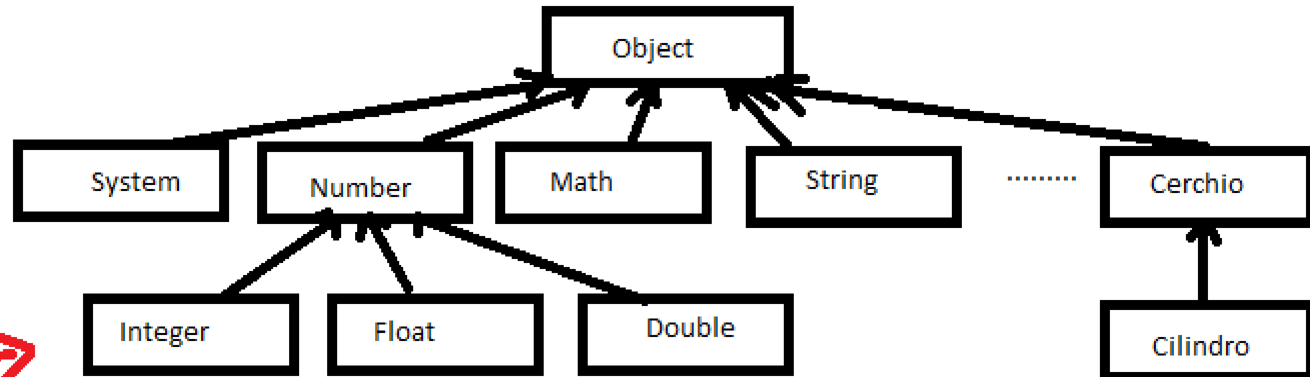


La gerarchia delle classi

In java esiste una classe, chiamata classe **Object,** da cui vengono derivate tutte le altre classi e che è la sopraclasse di ogni altra classe.

Volendo generalizzare, il grafo di gerarchia delle classi Java si potrebbe rappresentare con la seguente figura, in cui la radice è la sopraclasse Object.

Le classi Integer, Float e Double non vanno confuse con i tipi di dato predefiniti int, float e double. Rappresentano invece delle vere classi con propri attibuti e propri metodi.



Le ultime classi della gerarchia

Se una classe viene dichiarata usando la parola **final**, da questa non possono essere generate sottoclassi. L'uso di ***final*** serve per terminare la gerarchia di ereditarietà.

final class Ultima{....} la classe Ultima non può più avere sottoclassi

Se si inserisce dopo la parola chiave **extends** il nome di questa classe, viene segnalato un errore:

class Nuova extends Ultima{...} la dichiarazione genera un errore

Anche i **metodi** possono essere dichiarati usando la parola ***final*** nell'intestazione e prima del nome del metodo: **public final void aggiorna(){....}** Un **metodo** così dichiarato non può essere sovrascritto all'interno delle sottoclassi, cioè non possono esistere dei metodi nelle sottoclassi che hanno lo stesso nome. Lo stesso vale per i metodi dichiarati con **private**.

Il polimorfismo

Il termine **polimorfismo**, in generale, indica la capacità di assumere più forme. Nella programmazione orientata agli oggetti è strettamente legato all'ereditarietà e alla ridefinizione dei metodi: considerando una relazione di ereditarietà, le sottoclassi hanno la possibilità di ridefinire i metodi ereditati (mantenendo lo stesso nome) oppure lasciarli inalterati perché già soddisfacenti.

Il **polimorfismo** indica la possibilità per i metodi di assumere forme (cioè implementazioni) diverse all'interno della gerarchia delle classi.

Nei linguaggi ad oggetti sono presenti due tipi di polimorfismo:

-l'**overriding**, o sovrapposizione dei metodi;

-L'**overloading**, o sovraccarico dei metodi.

L'overriding di un metodo consiste nel ridefinire, nella classe derivata, un metodo ereditato, con lo scopo di modificarne il comportamento. Il nuovo metodo deve avere lo stesso nome e gli stessi parametri del metodo che viene sovrascritto.

Si consideri la classe Cilindro definita precedentemente (pag. 54), tra i metodi che essa eredita dalla classe Cerchio (pag. 55) c'è il metodo ***area***, che calcola l'area del cerchio alla base del cilindro. Questo metodo può essere sovrascritto per calcolare l'area della superficie totale del cilindro, formata dall'area dei due cerchi di base più l'area della superficie laterale.

Vediamo nella pagina successiva il codice del metodo sovrascitto.

Il polimorfismo

```
public double area()
{
  double supBase,supLaterale;
  supBase=super.area()*2;     /*calcolo della superficie di base: richiama il metodo area della classe Cerchio. Viene moltiplicato per 2 perché il cilindro ha due superfici circolari. */
  supLaterale=circonferenza()*altezza;   /*calcolo della superficie laterale:richiama il metodo circonferenza ereditato dalla classe Cerchio */
   return supBase+supLaterale;  }
```

Dopo questa dichiarazione il metodo area, ereditato dalla classe Cerchio, viene sovrascritto con il nuovo metodo. Tutte le volte che un oggetto di classe Cilindro invoca l'esecuzione del metodo area, si attiva l'esecuzione del nuovo metodo.

Il metodo che è stato sovrascritto può essere richiamato facendo riferimento all'oggetto speciale **super**, perché quest'ultimo contiene il riferimento alla sopraclasse. Il metodo area dell'esempio può essere richiamato sia dalle istanze della classe Cerchio che dalle istanze della classe Cilindro. A seconda dell'istanza che richiede l'esecuzione, viene attivato un metodo piuttosto che l'altro.

L'overloading

L'overloading di un metodo è la possibilità di utilizzare lo stesso nome per compiere operazioni diverse. Solitamente si applica ai metodi della stessa classe che si presentano con lo stesso nome, ma con un numero o un tipo diverso di parametri.

All'interno della stessa classe, usando l'overloading, si possono dichiarare più metodi che hanno lo stesso nome ma possiedono parametri diversi. Il compilatore, comparando il numero e i tipi dei parametri, individua il metodo corretto da invocare.

Il polimorfismo

L'overloading

Un esempio di metodo che utlizza l'overloading è **println.**

Questo metodo consente di stampare a video i diversi tipi di parametri che vengono passati, tra cui stringhe, numeri interi e reali. Esistono quindi diversi metodi println con lo stesso nome, che vengono richiamati allo stesso modo ma con parametri diversi.

```
System.out.println("stringa di testo");
System.out.println(23);
```

L'overloading fornisce un modo per far assumere allo stesso metodo dei comportamenti classificabili come polimorfismo: a seconda dei parametri che vengono passati, viene eseguito un comportamento diverso.

L'overloading è anche usato per offrire più di un costruttore a una classe.

```
Class Numero
{  private int num;
public Numero()
{
  num=0;
}
public Numero(int num)
{
  this.num=num; } }
```

Quindi un oggetto di classe Numero può essere creato in due modi

```
Numero n1=new Numero();
Numero n2=new Numero(100);
```

Le stringhe

Le stringhe non sono un tipo di dato predefinito in Java: sono definite usando una classe chiamata **String** che è inclusa nel package.java.lang. Questa classe serve per creare gli oggetti che rappresentano array di caratteri. Dopo aver fissato la dimensione della stringa, essa non può più essere modificata a meno di creare un nuovo oggetto.

La classe String contiene vari metodi che consentono di manipolare le stringhe: è possibile contare il numero di caratteri, leggere i singoli caratteri, creare sottostringhe, convertire tutta la stringa in lettere minuscole o maiuscole.

Per usare le stringhe in Java si deve creare un oggetto di classe String.

String nome= new String("Elena");

Esiste un altro modo per allocare le stringhe: invece di usare l'operatore new, si indica tra virgolette la stringa che si vuole assegnare all'oggetto.

String nome="Laura";

Un ulteriore modo per generare le stringhe è tramite l'utilizzo dei metodi presenti nella classe String, che restituiscono un oggetto di classe String.

Una nuova stringa composta dai primi tre caratteri del nome si ottiene con:

String abbreviazione=s.substring(0,3);

L'operatore di concatenazione tra stringhe (+) crea automaticamente un oggetto di classe String. Esso consente di concatenare anche numeri effettuando automaticamente la conversione in stringa.

Int a =10;

String s="il numero"+a+" e' pari. ";

La classe String non possiede un attributo accessibile.

Le stringhe

Vediamo quali sono i metodi principali della classe String:

-Il metodo **length** restituisce un numero che corrisponde alla lunghezza della stringa.

-Il metodo **charAt** restituisce il carattere che si trova nella posizione indicata dal parametro. Il primo carattere si trova a partire dalla posizione 0, mentre l'ultimo è in posizione s.length()-1

-I metodi **toLowerCase** e **toUpperCase** convertono tutti i caratteri rispettivamente in minuscole e in maiuscole.

-Il metodo **substring** restituisce una sottostringa della stringa a cui viene applicato. Il primo parametro indica la posizione iniziale della sottostringa, mentre il secondo indica la posizione finale.

-Il metodo **equals** controlla se la stringa che riceve come parametro è uguale a quella a cui viene applicato. Restituisce un valore booleano: true se le stringhe sono uguali false se diverse.

Se si applica il significato di questi metodi al seguente oggetto:

String s="Internet è la Rete delle reti";

Si ottiene che:

-s.length() restituisce il valore 29;

-s.charAt(2) restituisce il valore 't';

-s.toUpperCase() crea la nuova stringa "INTERNET E' LA RETE DELLE RETI";

-DICHIARANDO String s1=s.substring(15,19), si ottiene l'oggetto s1 contenente la stringa "Rete";

-s.equals("Internet") restituisce false.

Esercizio sulle stringhe

1) Le vocali e gli spazi di una frase.

Analizzare una frase inserita da tastiera calcolando la frequenza delle vocali e degli spazi bianchi.

Il problema chiede di contare il numero di vocali e di spazi a partire da una stringa di testo inserita da tastiera. Le informazioni sul testo, le vocali, gli spazio e la logica di calcolo devono essere incapsulate nella classe **GestoreFrasi**. L'attributo **testo** deve contenere la stringa da analizzare, per il conteggio utilizzare gli attributi **vocali** e **spazio**, il metodo **analizza** deve ricevere come input il testo da analizzare e calcolare il numero di vocali e spazio. Prevedere inoltre due metodi getFreqVocali e getFreqSpazi per restituire la frequenza di vocali e degli spazi. Infine si chiede di prevedere un metodo privato **arrotonda** che viene utilizzato dai due metodi pubblici **getFreqVocali** e **getFreqSpazi** per restituire un valore arrotondato. Si vedano i file nelle prossime due slide: GestoreFrasi.java e ProgAnalisi.java.

Il metodo analizza contiene un ciclo che itera su tutti i caratteri della stringa testo. Il singolo carattere viene memorizzato in una variabile di tipo char con la seguente istruzione:

```
carattere=testo.charAt(i);
```

Successivamente viene utilizzata la struttura switch per verificare se il carattere è una vocale o uno spazio e per incrementare il corrispondente attributo.

Nel metodo mai la stringa da analizzare viene letta e memorizzata nell'oggetto frase, che viene creato come istanza della classe String e inizializzato con una stringa vuota tramite la seguente istruzione:

```
String frase="";
```

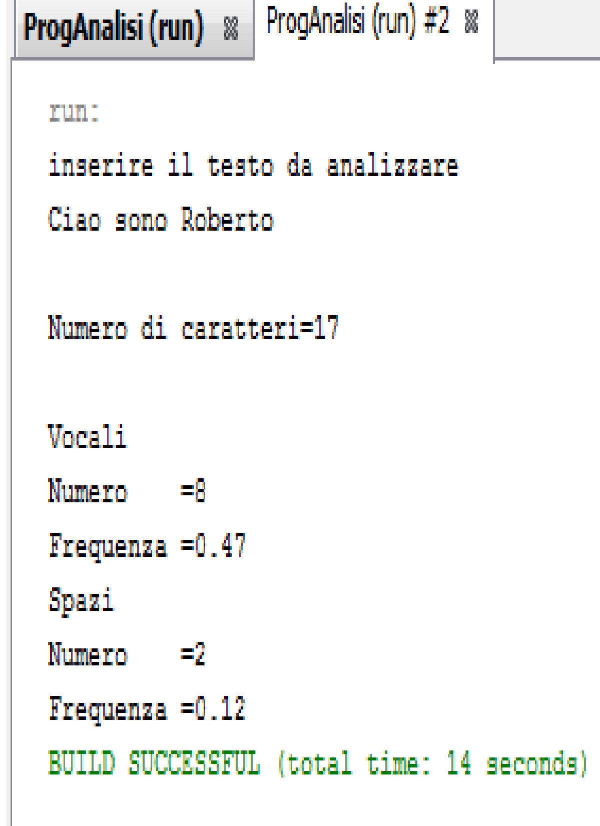
File GestoreFrasi.java

```
package proganalisi;
public class GestoreFrasi {
 private String testo;
 public int vocali;
 public int spazi;
 //costruttore: inizializza gli attributi
 public GestoreFrasi()
 {
    testo="";
    vocali=0;
    spazi=0;
 }
 //calcola il numero di vocali e di spazi
 public void analizza(String t)
 {
    char carattere;
    this.testo=t;
    testo.toLowerCase();
    for(int i=0;i<testo.length();i++)
    {
       carattere=testo.charAt(i);
       switch (carattere)
       {
          case 'a':
          case 'e':
          case 'i':
          case 'o':
          case 'u':vocali++;
             break;
          case ' ':spazi++;
             break;   }  }  }
```

```
//calcola la frequenza delle vocali
 public double getFreqVocali()
 {
    double freqVocali;
    freqVocali=(double) vocali/testo.length():
    return arrotonda(freqVocali);
 }
 //calcola la frequenza degli spazi
 public double getFreqSpazi()
 {
    double freqSpazi;
    freqSpazi=(double) spazi/testo.length();
    return arrotonda(freqSpazi);
 }
 //arrotonda a due cifre decimali
 private double arrotonda(double valore)
 {
    return (double)
Math.round(valore*100)/100;
 }
}
```

File ProgAnalisi.java

```
package proganalisi;
import java.io.*;
public class ProgAnalisi {
  public static void main(String[] args) {
    InputStreamReader input=new InputStreamReader(System.in);
    BufferedReader tastiera =new BufferedReader(input);
    String frase="";
    GestoreFrasi gestore = new GestoreFrasi();
    System.out.println("inserire il testo da analizzare");
    try{
      frase=tastiera.readLine();
    }
    catch (IOException e){}
    {
     gestore.analizza(frase);
     System.out.println("\nNumero di caratteri="+frase.length());
     System.out.println("\nVocali");
     System.out.println("Numero    ="+gestore.vocali);
     System.out.println("Frequenza ="+gestore.getFreqVocali());
     System.out.println("\Spazi");
     System.out.println("Numero    ="+gestore.spazi);
     System.out.println("Frequenza ="+gestore.getFreqSpazi());
     } } }
```



Le strutture di dati dinamiche

Le strutture di dati dinamiche sono un modo per organizzare i dati di cui a priori non è nota la dimensione.

Il problema nel gestire questi dati sorge perché non esiste un numero prefissato indicante la loro quantità e durante l'esecuzione questo numero può aumentare e diminuire continuamente. Per poter trattare questo tipo di dati servono quindi strutture apposite, che vengono definite strutture di dati dinamiche.

Le strutture di dati dinamiche hanno in comune la possibilità di adattarsi al variare delle dimensioni dei dati da trattare e si differenziano tra di loro per il tipo di operazioni che vengono eseguiti sui dati. Le operazioni tipiche di queste strutture si dividono in:

-**operazioni di modifica**: inserimento ed eliminazione di un elemento nella struttura;

-**operazioni di interrogazione**: ricerca di un certo elemento nella struttura, prelevamento di un elemento con particolari caratteristiche (per esempio il primo elemento inserito), conteggio del numero di elementi inseriti nella struttura.

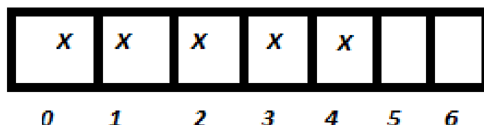
Gli array dinamici

Le strutture di dati dinamiche possono essere pensate in termini di oggetti, in quanto sono composte da dati e da un insieme di operazioni che agiscono sui dati. Per **implementare** le strutture di dati dinamiche non possono essere usate strutture statiche come gli array, in quanto la loro dimensione è predeterminata e limita il numero massimo di elementi che possono essere trattati. Si potrebbe pensare di fissare una dimensione grande per l'array in modo da comprendere il caso peggiore, ma in questo modo si ottiene un inutile spreco di memoria. La realizzazione di strutture dinamiche invece comporta un'interazione continua con il gestore della memoria per allocare nuove porzioni di memoria quando la struttura di dati cresce e per deallocare quando decresce.

Le strutture di dati dinamiche

Per risolvere i problemi legati all'allocazione e deallocazione della memoria si usano i vettori di oggetti senza una dimensione prefissata: può aumentare o diminuire in base alle necessità.
Java rende disponibile una struttura dati di questo tipo attraverso la classe **Vector**, inclusa nel package **java.util** che deve essere importato nel programma.
Questa classe implementa una array di oggetti dinamico nel senso che è basata su un array le cui dimensioni possono aumentare quando vengono aggiunti nuovi dati e diminuendo quando i dati vengono eliminati. L'operazione di adattamento è eseguita automaticamente : viene allocato nuovo spazio quando serve e deallocato quando non è più utilizzato. La classe **Vector** mette a disposizione i metodi per aggiungere gli elementi in fondo all'array e per prelevare o eliminare gli elementi in una certa posizione. Questi metodi sono quelli utilizzati per implementare le operazioni tipiche delle strutture di dati dinamiche.
Gli elementi di un Vector sono accessibili attraverso un indice, come avviene per gli array. Il primo elemento del vettore ha indice 0. Ogni vettore ha una sua capacità che è data dal numero di oggetti che può memorizzare. Quando il vettore si riempie e c'è bisogno di ulteriore spazio, la capacità del vettore viene incrementata automaticamente.
Per dichiarare un vettore si possono usare tre diversi **costruttori.**

Vector v=new Vector(); // crea un vettore senza specificare la capacità
Vector v=new Vector(5); // crea un vettore specificando la capacità tramite un parametro
Vector v=new Vector(5,2); /* crea un vettore senza specificare la capacità iniziale con il primo parametro e il valore di incremento con il secondo parametro*/



Quando si deve inserire il sesto oggetto, la dimensione del vettore viene incrementata di 2 come specificato dall'ultimo costruttore

Le dinamiche del vettore si manifestano anche in senso contratorio: quando gli elementi vengono eliminati, le dimensioni del vettore vengono ridotte di conseguenza.

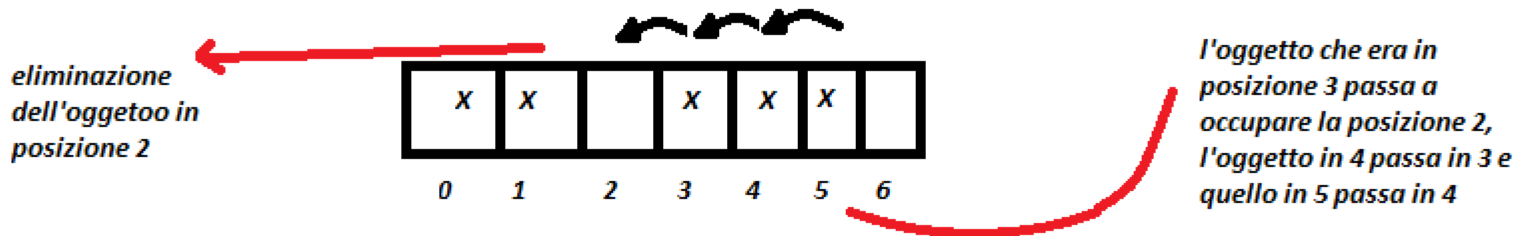
Gli array dinamici

I metodi della classe Vector

I metodi principali della classe Vector che consentono di manipolare un vettore sono i seguenti:

-**addElement(Object obj)** riceve come parametro un oggetto di qualunque classe e lo aggiunge in fondo al vettore, all'interno del vettore gli oggetti subiscono un casting verso la classe Object.

-**removeElementAt(int** index) toglie dal vettore l'elemento nella posizione indicata dall'indice; la sua posizione non resta vuota, perché tutti gli elementi con un indice più grande vengono fatti scalare per occupare la posizione e ottimizzare la memoria.



Se il parametro fa riferimento a un indice non corretto, cioè un indice negativo o maggiore della dimensione del vettore, viene generata un'eccezione.

La classe Vector, oltre a quelli già elencati, dispone anche di altri metodi:

-**size()** restituisce un numero intero che indica il numero di oggetti effettivamente mermorizzati nel vettore; è un numero minore o uguale alla capacità del vettore;

-**elementAt(int** index) restituisce un oggetto di classe Object che si trova nella posizione indicata dall'indice; per convertirlo in un oggetto della classe originaria va effettuato il casting. Il parametro deve essere un indice consentito, in caso contrario viene generata un'eccezione.

I vertici di un poligono

Utilizzando un array dinamico per memorizzare un insieme di coordinate cartesiane che rappresentano i vertici di un poligono, di cui non si consoce la cardinalità.
L'array dinamico viene implementato usando l'oggetto poligono di classe Vector definito come attributi privato nella classe Poligono nel seguente modo. I file appartenenti al progetto sono tre (Poligono.java, Punto.java e TestPoligono.java) . Di seguito il file **Poligono.java**

```
package testpoligono;
import java.util.*;
public class Poligono {
   private Vector poligono;
   public Poligono()
   {
      poligono=new Vector(1,1);  //vettore con capacità unitaria e valore di incremento
    }                            //unitario
   public void aggiungi(Punto p)
   {
      poligono.addElement(p);
   }
   public void stampa()
   {
      Punto p;                            //la variabile temporanea p memorizza
      System.out.println("Elenco punti"); // l'oggetto recuperato dal vettore
       for(int i=0;i<poligono.size();i++)  // dimensione del vettore ottenuta con
      {                                    // il metodo size
         p=(Punto) poligono.elementAt(i); // l'oggetto restituito dal metodo elementAt
         System.out.println(p);           //è di classe Object, quindi si deve applicate
      }                                   //il casting verso la classe corretta
   }
   public void togli(int i)
   {
      if ((i>=0) && (i<poligono.size()))    //verifica che l'indice sia nell'intervallo corretto
      {
         poligono.removeElementAt(i);  //posizione
      }      } }                       // dell'oggetto da eliminare
```

L'aggiunta di un oggetto al vettore **poligono** utilizza il metodo **addElement** a cui viene l'oggetto da aggiungere, come mostrato nel metodo pubblico **aggiungi** della classe **Poligono**. Quasi tutte le elaborazioni applicate ai singoli elementi del vettore hanno bisogno di scorrete il vettore dal primo all'ultimo elemento. Per eseguite questa operazione si utilizza un ciclo che parte dall'elemento in posizione o e scorre tutti gli elementi. Il metodo pubblico stampa della classe **Poligono** scorre gli elementi del vettore e ne stampa il contenuto. Un oggetto si elimina dal vettore poligono usando il metodo **removeElementAt** e indicando, con il parametro, la posizione dell'oggetto da eliminare, come mostrato nel metodo pubblico **togli** della classe **Poligono**.

File Punto.java

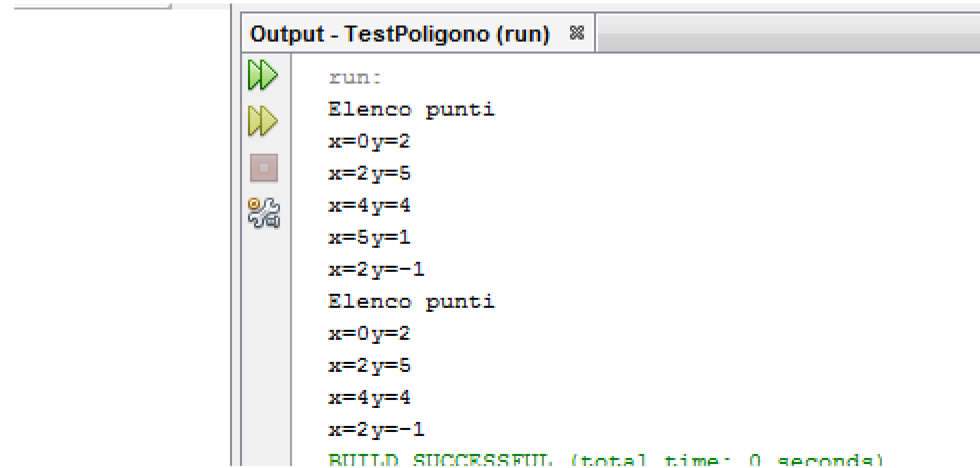
Il vettore poligono è composto da oggetti generati come istanze della classe Punto, composto dagli attributi privati x e y e definita nel seguente modo:

```
package testpoligono;
public class Punto {
private int x,y;
public Punto(int x, int y)
{
   this.x=x;
   this.y=y;
}
public String toString()
{
   return "x="+x+"y="+y;
}
}
```

Implementazione TestPoligono.java

Il seguente programma verifica il comportamento delle precedenti classi Poligono e Punto creando un poligono, aggiungendo cinque vertici e, successivamente, eliminando un vertice.

```
package testpoligono;
public class TestPoligono {
  public static void main(String[] args) {
   Punto p1=new Punto(0,2);
   Punto p2=new Punto(2,5);
   Punto p3=new Punto(4,4);
   Punto p4=new Punto(5,1);
   Punto p5=new Punto(2,-1);
   Poligono poli=new Poligono();
   poli.aggiungi(p1);
   poli.aggiungi(p2);
   poli.aggiungi(p3);
   poli.aggiungi(p4);
   poli.aggiungi(p5);
   poli.stampa();
   poli.togli(3);
   poli.stampa();
    }
 }
```



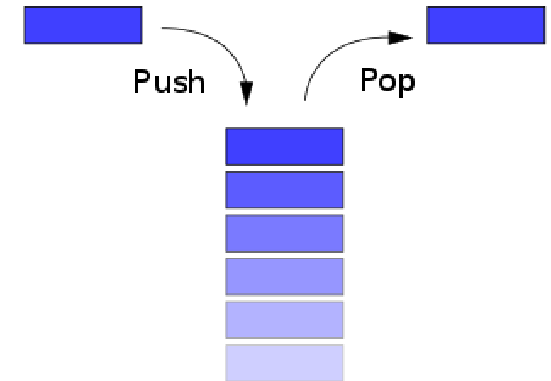
La pila

La pila è una struttura di dati dinamica gestita usando la modalità LIFO (Last In First Out): l'inserimento e l'estrazione dei dati avvengono da un'unica estremità.

Gli elementi vengo aggiunti da un'estremità (**push**) e sono posizionati uno sopra l'altro, formando idealmente una pila di oggetti. Il prelevamento di un elemento (**pop**) avviene a partire da quello che si trova in cima alla pila ed è stato inserito per ultimo.

La pila viene schematizzata con il seguente diagramma di classe.

Struttura Pila



Le operazioni di **modifica** sono:

- push: inserire un elemento in cima alla pila;
- pop: preleva un elemento eliminandola dalla cima della pila;

Le operazioni di **interrogazioni** sono:

- top:restituisce l'elemento in cima alla pila senza eliminarlo;
- vuota:segnala, con valore booleano, se la pila non contiene elementi;
- size: restituisce il numero di elementi presenti nella pila

La pila

Modello generico di pila

Implementare la pila come struttura di dati generica, adatta a trattare qualunque tipo di oggetto. Questa classe potrà essere riutilizzata tutte le volte che serve un struttura di dati dinamica con le caratteristiche della pila. La classe Pila contiene un attributo privato in cui verranno memorizzati i dati. Questo attributo a un oggetto di classe **Vector** dichiarato nel seguente modo:

```
private Vector  elementi;
```

Il numero degli elementi nella pila viene controllato con il metodo **size** ed il valore viene memorizzato in una variabile intera chiamata size;

```
int  size=elementi.size();
```

L'operazione **pop** legge l'elemento che si trova in posizione **size-1**, lo cancella e lo restituisce al chiamante.

Nota bene: in questa implementazione della classe **Pila** i singoli elementi sono gestiti usando la classe **Object.** Questo significa che qualunque oggetto può essere aggiunto alla pila, in quanto la classe Object è la sopraclasse di ogni altra classe. Al momento di eseguire l'operazione **push**, viene automaticamente applicato il ***casting*** verso la classe **Object**. Quando viene prelevato un elemento, la pila lo restituisce come oggetto di classe Object: sarà compito di chi riceve l'oggetto convertirlo tramite un ulteriore ***casting*** verso la classe originaria.

Una pila realizzata in questo modo è una struttura di dati generica che può essere riutilizzata per trattare qualunque tipo di oggetto.

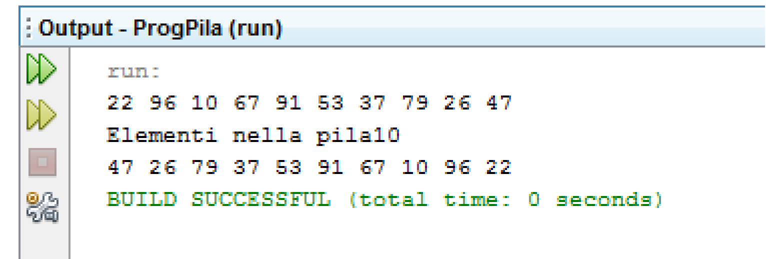
Vediamo nelle prossime pagine l'implementazione della classe **Pila.java** utilizzata per popolare una pila di dieci numeri casuali tramite il file **ProgPila.java.** La pila viene svuotata mostrando l'ordine invertito degli elementi, dall'ultimo al primo.

File Pila.java

```
package progpila;
import java.util.*;
public class Pila {
    private Vector elementi;
    public Pila()
    {
        elementi=new Vector();
    }
    public void push(Object obj)
    {
        elementi.addElement(obj);
    }
    public Object pop()
    {
        Object obj=null;
        int size=elementi.size();
        if (size>0)
        {
            obj=elementi.elementAt(size-1);
            elementi.removeElementAt(size-1):
        }
        return obj;    }
public Object top()
    {
        Object obj=null;
        int size=elementi.size();
        if (size>0)
        {
            obj=elementi.elentAt(size-1);
        }
        return obj;
    }
    public boolean vuota()
    {
        if (elementi.size()>0)
        {
            return false;
        }
        else
        {
            return true;
        }
    }
    public int size()
    {       return elementi.size();   } }
```

File class ProgPila.java

```
package progpila;
public class ProgPila {
  public static void main(String[] args) {
    // crea una pila vuota
    Pila pila=new Pila();
    int num;
    Integer numObj;
    //aggiunge elementi alla pila
    for(int i=0;i<10;i++)
    {
      num=(int) (Math.random()*100);
      numObj=new Integer(num);
      System.out.print(numObj+" ");
      pila.push(numObj);
    }
    System.out.println("\nElementi nella pila"+pila.size());
    //toglie gli elementi e li visualizza
    while (!pila.vuota())
    {
      numObj=(Integer) pila.pop();
      System.out.print(numObj+" ");
    }
    System.out.println();
  }
}
```



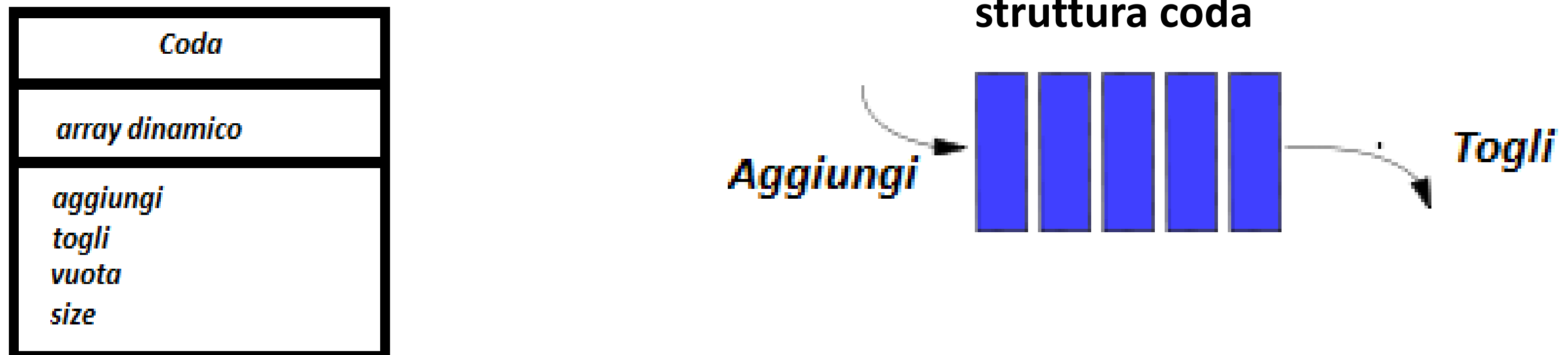
La coda

La **coda** è una struttura dinamica di dati gestita usando la modalità FIFO (First In First Out): l’inserimento avviene da un’estremità e l’estrazione dall’altra.

Gli elementi vengono aggiunti da un’estremità e sono posizionati uno di seguito all’altro, formando idealmente una coda di oggetti. Il prelevamento di un elemento avviene a partire da quello che è stato inserito per primo e si trova all’inizio della coda.

Questa struttura di dati viene usata quando si vuole tener traccia delle informazioni per poi recuperarle nello stesso ordine nel quale sono state inserite .

La coda viene schematizzata con il seguente diagramma di classe.



Le operazioni di **modifica** sono:

- aggiungi: inserire un elemento alla fine della coda;
- togli: preleva un elemento all’inizio della coda.

Le operazioni di **interrogazione** sono:

- vuota: segnala, con un valore booleano, se la coda non contiene elementi;
- size: restituisce il numero di elementi presenti nella coda.

La coda

Modello generico di coda

Implementare la coda come struttura di dati generica, adatta a trattare qualunque tipo di oggetto. Questa classe potrà essere riutilizzata tutte le volte che serve una struttura di dati dinamica con le caratteristiche della coda.

Come la pila, anche la coda contiene un attributo privato di classe Vector in cui verranno memorizzati i dati. I singoli elementi sono gestiti usando la classe Object.

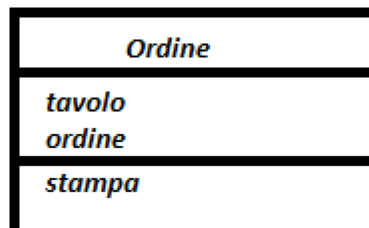
Il codice completo della classe Coda è riportato nella prossima pagina.

Per fare un esempio pratico, creiamo un programma che simuli il funzionamento di un bar. Può essere inclusa la classe Coda in un programma Java che utilizza le code.

Un bar può essere immaginato in modo semplificato come un servizio in cui i clienti generano delle ordinazioni e i gestori soddisfano queste ordinazioni.

Il nostro progetto può contenere quindi tre file**: Ordine.java, ProgBar.java e Coda.java**

Ogni ordinazione è rappresentata dal numero del tavolo e dal contenuto dell'ordine e può essere descritta con il seguente diagramma di classe:



La coda – gestione di un Bar - file progbar.java

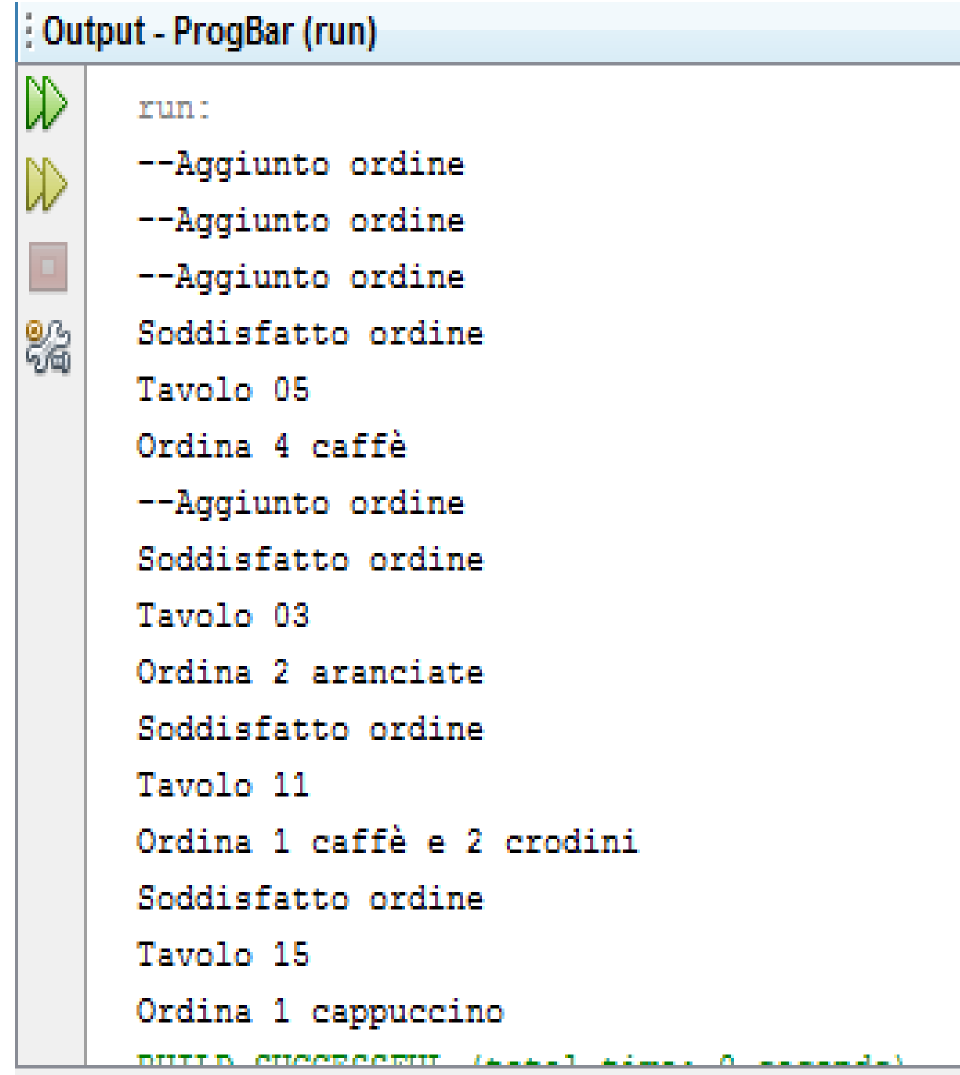
```
package progbar;
 public class ProgBar {
   public static void main(String[] args) {
    Coda codaOrdinazioni=new Coda();
    Ordine ord;
    ord=new Ordine("05","4 caffè");
    codaOrdinazioni.aggiungi(ord);
    System.out.println("--Aggiunto ordine");
    ord=new Ordine("03","2 aranciate");
    codaOrdinazioni.aggiungi(ord);
    System.out.println("--Aggiunto ordine");
    ord=new Ordine("11","1 caffè e 2 crodini");
    codaOrdinazioni.aggiungi(ord);
    System.out.println("--Aggiunto ordine");
    ord=(Ordine) codaOrdinazioni.togli();
    System.out.println("Soddisfatto ordine");
    ord.stampa();
    ord=new Ordine("15","1 cappuccino");
    codaOrdinazioni.aggiungi(ord);
    System.out.println("--Aggiunto ordine");
    ord=(Ordine) codaOrdinazioni.togli();
    System.out.println("Soddisfatto ordine");
    ord.stampa();
    ord=(Ordine) codaOrdinazioni.togli();
    System.out.println("Soddisfatto ordine");
    ord.stampa();
    ord=(Ordine) codaOrdinazioni.togli();
    System.out.println("Soddisfatto ordine");
    ord.stampa();
} }
```

La coda – gestione di un Bar – file Coda.java

```
package progbar;
import java.util.*;
public class Coda {
   private Vector elementi;
   public Coda()
   {
      elementi=new Vector();
   }
   public void aggiungi(Object obj)
   {
      elementi.addElement(obj);
   }
   public Object togli()
   {
      Object obj=null;
      int size =elementi.size();
      if (size>0)
            {
               obj=elementi.elementAt(0);  //gi
lementi vengono tolti dalla coda sempre
               elementi.removeElementAt(0);
//dalla posizione 0
            }
            return obj;
   }
   public boolean vuota()
   {
      if (elementi.size()>0)
      {
         return false;
      }
      else
      {
         return true;
      }
   }
   public int size()
   {
      return elementi.size();
   }
}
```

La coda – gestione di un Bar – file Ordine.java

```
package progbar;
public class Ordine {
private String tavolo;
private String ordine;
public Ordine(String tav, String ord)
{
   tavolo=tav;
   ordine=ord;
}
public void stampa()
{
   System.out.println("Tavolo "+tavolo);
   System.out.println("Ordina "+ordine);
}
}
```



Esercizi

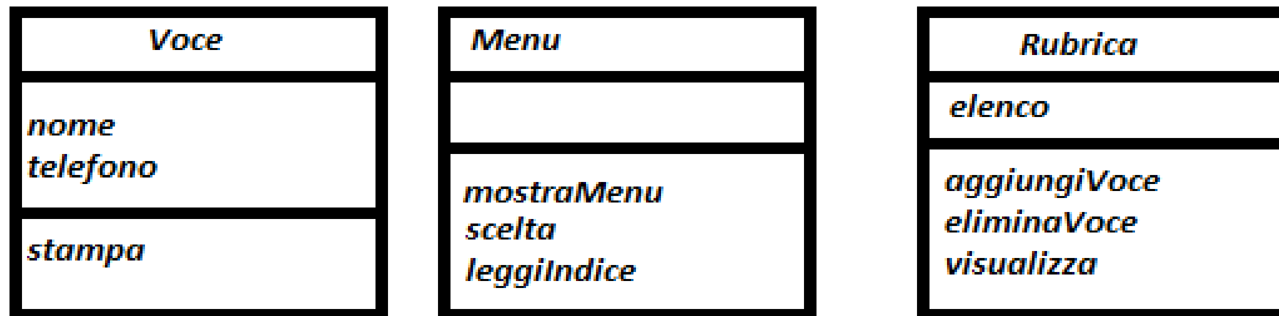
1) La rubrica telefonica in un array.

Costruire un programma per gestire una rubrica telefonica.

La rubrica telefonica che si vuole gestire contiene un numero variabile di elementi; ogni voce della rubrica memorizza un nome e un numero di telefono.

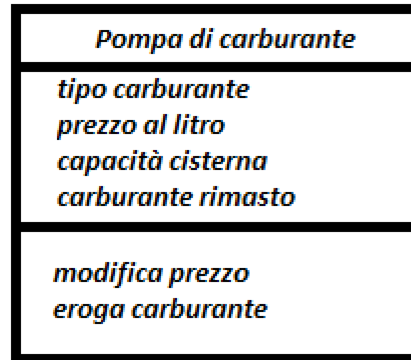
Il programma si presenta agli utenti con un menu per la gestione della rubrica, composto dalle seguenti opzioni: aggiunta di una voce, eliminazione di una voce, visualizzazione della rubrica.

La classe Voce racchiude le informazioni di un singolo nominativo, mentre la classe Rubrica contiene l'elenco dei nominativi in un vettore, dichiarato come attributo privato. Le operazioni di modifica della rubrica sono incapsulate nei metodi della classe Rubrica, mentre la classe Menu viene utilizzata per controllare le scelte dell'utente. Queste tre classi sono descritte nei seguenti diagrammi di classe.



Esercizi

2) Data la seguente classe, crea tre istanze (oggetti) rappresentandole con il diagramma degli oggetti. Implementare la classe e gli oggetti nel linguaggio Java.



3) Usando il digramma delle classi, descrivi un forno a microonde. Indica tutti i possibili attributi (per esempio marca) e metodi (per esempio chiudiSportello). Descrivi poi graficamente il diagramma degli oggetti forno1 e forno2 attribuendo opportuni valori agli attributi. Implementa la classe e gli oggetti nel linguaggio Java.

4) Utilizza il diagramma delle classi, descrivi un dispositivo capace di trattare segnali elettrici (alimentatore, amplificatore, modulatore, campionatore, ecc.). Rappresenta graficamente, con il diagramma degli oggetti, due istanze della classe attribuendo opportuni valori agli attributi. Implementa la classe e gli oggetti nel linguaggio Java.

5) Dati dieci nomi di persona, conta e visualizza solo quelli che iniziano con una vocale.

Esercizi

6) Crea la classe per rappresentare una retta avente equazione $y=ax+b$ con il metodo per controllare se un generico punto appartiene alla retta. Deriva la classe per rappresentare la parabola $y=ax^2+bx+c$, modificando il metodo per controllare se un punto appartiene alla parabola.
7) Scriver un programma che conti le lettere e i numeri contenuti in una stringa.
8) Dati dieci nomi di persona, conta e visualizza solo quelli che iniziano con una vocale.
9) Scrivi un programma che consenta di manovrare un'automobile. Le due operazioni possibili sono accelerare (A) e frenare(F). Il programma resta in attesa che l'utente inserisca un comando. Se inserisce il carattere A, l'automobile aumenta di velocità di 5 km/h. Se inserisce il carattere F, l'automobile rallenta la velocità di 10 km/h. Dopo l'inserimento di ogni comando si deve mostrare la velocità attuale dell'automobile. Se si supera i 90 km/h deve essere segnalato un avvertimento "vai troppo forte. Rallenta". La velocità non può essere inferiore a zero. Inizialmente l'automobile sta viaggiando a 50 km/h.
10) Inserisci un vettore di dieci libri, specificando per ognuno il titolo e il numero di pagine. Alla fine dell'inserimento stampa il titolo di tutti i libri che hanno meno di 100 pagine.