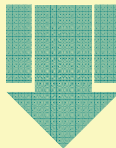
La Programmazione Orientata agli Oggetti (O.O.P.)

Roberta Gerboni

PROGRAMMAZIONE STRUTTURATA	PROGRAMMAZIONE OOP
Nella risoluzione dei problemi si applicano i principi della **programmazione strutturata** con una metodologia di analisi Top-Down scomponendo un problema complesso in sottoproblemi e procedendo per affinamenti successivi. In questo caso l'attenzione è rivolta più alle operazioni da compiere: MODELLO ORIENTATO AL PROCESSO, pur dando importanza all'analisi per identificare le strutture dati astratte più idonee e le possibili interazioni tra i dati.	Dal 1980 si afferma la tecnica della **Object Oriented Programming** che focalizza l'attenzione sui dati da manipolare piuttosto che sulle procedure che consentono di manipolarli e impone che siano questi ultimi alla base della scomposizione in moduli del software: MODELLO ORIENTATO AI DATI. Si pensa ad un sistema costituito da un insieme di **entità**, **oggetti**, che **interagiscono** tra loro. Si parla di **tipo di dato astratto** (ADT). Un **dato** è caratterizzato da due aspetti fondamentali: un **insieme di valori** e un **insieme di operazioni** che possono essere applicate a esso .

Programmazione
strutturata

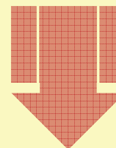
Problema complesso



Scomposizione in
Procedure

Programmazione
Orientata agli Oggetti

Sistema complesso



Scomposizione in
entità Interagenti
(oggetti)

Programmazione OOP

Programmare *ad oggetti* non velocizza l'esecuzione dei programmi...

Programmare *ad oggetti* non ottimizza l'uso della memoria...

Programmare *ad oggetti*
facilita la progettazione
e il mantenimento di sistemi software molto complessi

Principali vantaggi:

- supporto naturale alla modellazione software degli oggetti del mondo reale o del modello astratto da riprodurre
- più facile gestione e manutenzione di progetti di grandi dimensioni
- modularità e riuso di codice
- permette di definire oggetti software in grado di interagire gli uni con gli altri attraverso lo scambio di messaggi
- Riduce la dipendenza del codice di alto livello dalla rappresentazione dei dati in quanto l'accesso ai dati è mediato da un'interfaccia.

Oggetti e classi di Oggetti

L'elemento fondamentale della OOP è la **classe**.

La **classe** è un tipo aggregato che presenta forti analogie con le **strutture**.

Il ***concetto di struttura*** nasce dall'esigenza di manipolare insiemi eterogenei d'informazioni fra loro logicamente collegate. Ciò consente di creare delle nuove variabili che meglio modellano i dati riguardanti il problema che l'utente sta trattando

La **classe** è la **descrizione astratta** di un tipo di dato (ADT) e descrive una **famiglia di oggetti** con *caratteristiche* e *comportamenti* simili.

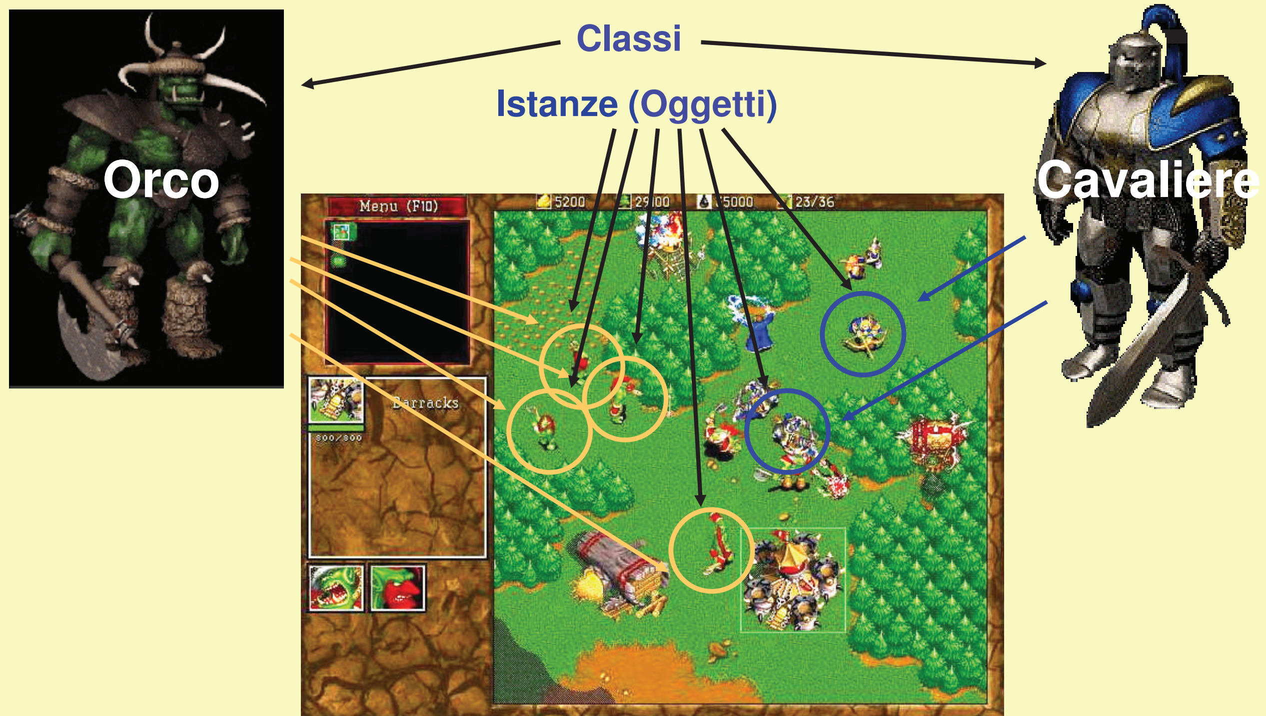
Un **oggetto** è una **istanza della classe**: quando si istanzia una variabile definendola di una certa classe, si crea un oggetto di quella classe rappresentato dal nome della variabile istanziata.

La differenza tra classe e oggetto è la stessa differenza che c'è tra tipo di dato e dato.

Oggetti e classi di Oggetti

Simulazione: un Videogioco

Oggetti e classi di Oggetti



Oggetti e classi di Oggetti

Ogni **oggetto** è definito da:

- **Attributi** che rappresentano le sue **caratteristiche** o **proprietà** fisiche *utili a definire il suo stato e sono **campi (variabili** o **costanti)***
- **Metodi** che rappresentano i **comportamenti ammissibili** o le **azioni**, le **proprietà dinamiche**, cioè le funzionalità dell'oggetto stesso e chi usa l'oggetto può attivarli.
  I metodi vengono realizzati con le **funzioni** contenenti le istruzioni che implementano le azioni dell'oggetto e possono avere parametri e fornire valori di ritorno.

Lo **stato di un oggetto**, è l'insieme dei valori delle sue proprietà in un determinato istante di tempo (**valori assunte dalle variabili**). Se cambia anche un solo valore di una proprietà di un oggetto, il suo stato varierà di conseguenza.

Nel corso dell'elaborazione, un oggetto è un'entità soggetta ad una **creazione**, ad un **suo utilizzo** e, infine, alla sua **distruzione**.

Oggetti e classi di Oggetti

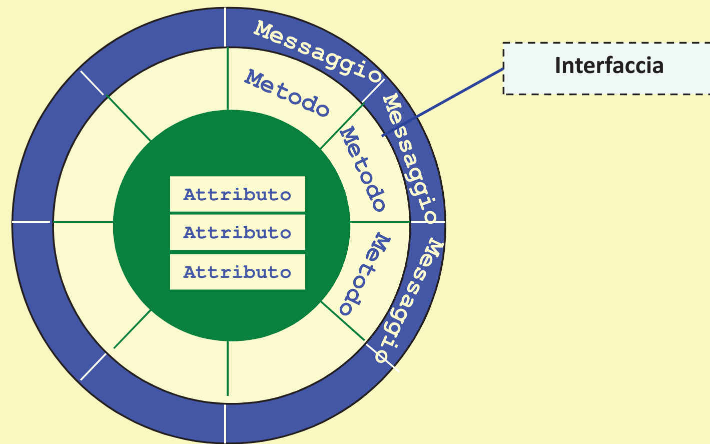
Membri di una classe

Oggetto : Attributi Metodi

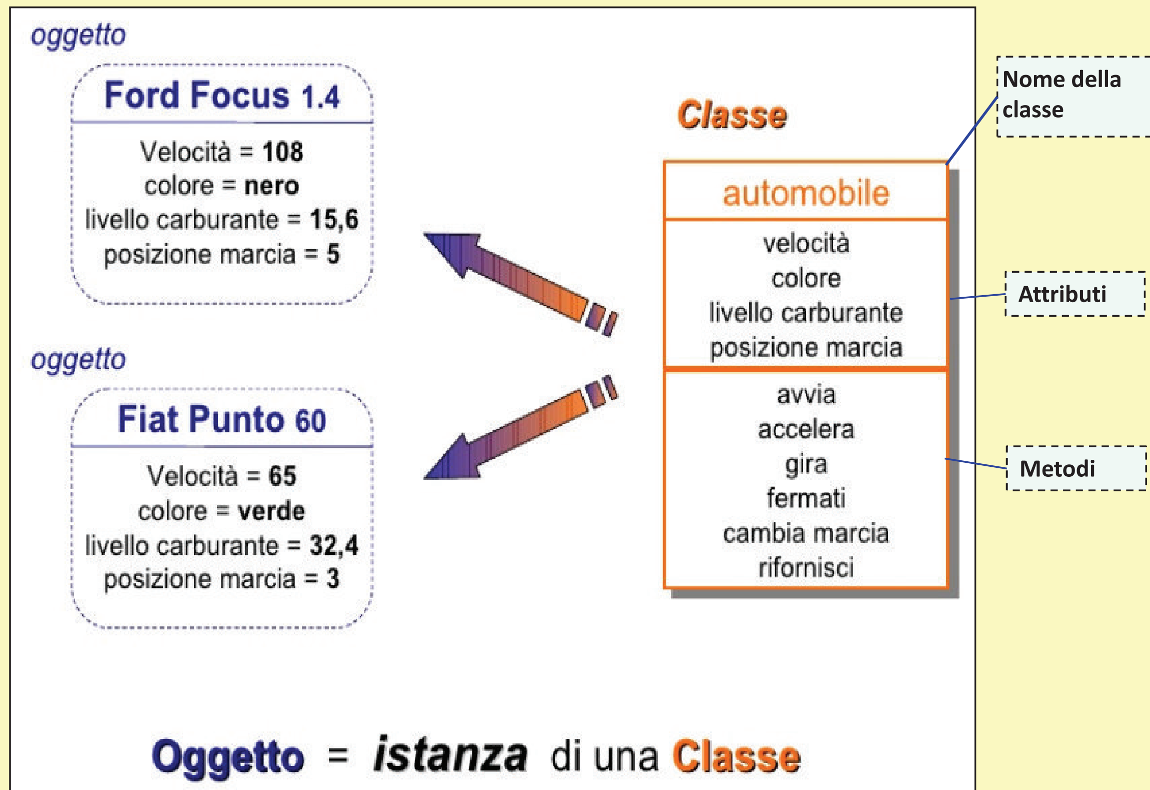
Dati membro — funzioni membro

Rappresentazione di una classe

I **metodi** di una classe costituiscono l'**interfaccia** della classe (cioè i messaggi che l'oggetto può interpretare) e sono l'unico meccanismo tramite il quale è possibile interagire con un oggetto della classe.



Esempio



Nome della classe

Attributi

Metodi

Esempio

- Tutti i soldati devono capire il messaggio *attacca*. Il messaggio ha conseguenze diverse a seconda del tipo di soldato:
  - un arciere scaglia una freccia
  - un fante colpisce di spada
  - un cavaliere lancia una lancia
- Il giocatore/computer deve gestire una lista di soldati e poter chiedere ad ogni soldato di *attaccare* indipendentemente dal tipo.


```
// esempio di codice rigido!

void attacca(Soldato tipo)
{
  if  ( tipo == arciere )
          scagliaLaFreccia();
  else if ( tipo == fante )
                 colpisciDiSpada();
        else if ( tipo == cavaliere )
                          lanciaLaLancia();
}
```

Che succede se devo aggiungere nuovi tipi di soldato?

```
// esempio di codice flessibile
Spadaccino s;
// . . .
s.attacca();
// . . .
```

Se devo aggiungere nuovi tipi di soldato il codice non cambia!

Programmazione OOP

Principali proprietà

1. Incapsulamento e information hiding

Uno dei grossi vantaggi è l'**incapsulamento**, cioè la proprietà degli oggetti di *mantenere al loro interno* sia gli ***attributi*** (le variabili) che i ***metodi*** (le funzioni), che descrivono rispettivamente lo stato e le azioni eseguibili sull'oggetto.

Si ha quindi come una capsula che **isola** l'oggetto dall'ambiente esterno proteggendo l'oggetto stesso. Legato a questo concetto c'è anche quello di **information hiding** (**mascheramento delle informazioni** rese quindi **invisibili** dall'esterno).

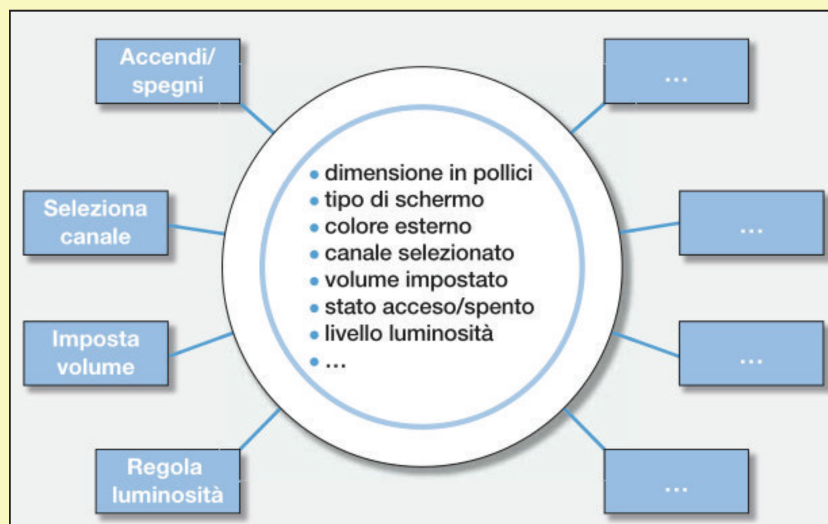
Infatti l'incapsulamento:

- **nasconde l'implementazione interna**: dall'esterno della classe **è noto cosa fa** un metodo pubblico, cioè se ne conosce l'interfaccia (funzionalità svolta, parametri in ingresso e tipo restituito), **ma non è noto come lo fa**.
- **consente un accesso protetto** e «ragionato» dall'esterno: ad esempio è possibile proteggere da scrittura un attributo definendolo *private* e implementando un metodo pubblico che ne restituisce il valore.

Esempio

La struttura privata interna di un oggetto che modella un **televisore** è incapsulata e protetta dall'interfaccia che espone all'esterno le sole *operazioni pubbliche* che permettono di interagire con l'oggetto stesso.

Classe Televisore



Incapsulamento

La proprietà dell'oggetto di ***incorporare*** al suo interno ***attributi*** e ***metodi*** viene detta *incapsulamento*

L'oggetto è quindi un **contenitore** sia di strutture dati e sia di procedure che li utilizzano

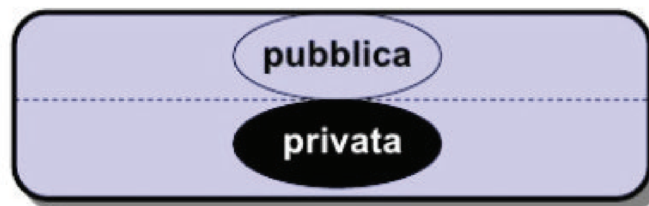
Viene visto come una ***scatola nera*** (*o blackbox*) permettendo così il ***mascheramento dell'informazione*** (***information hiding***)

Sezione ***Pubblica***

Sezione ***Privata***

Informatica – La Programmazione Orientata agli Oggetti

Struttura degli oggetti



Sezione ***Pubblica*** — ***attributi*** *e* ***metodi*** che si vogliono rendere **visibili** all'esterno *(e quindi utilizzabili dagli altri oggetti)*

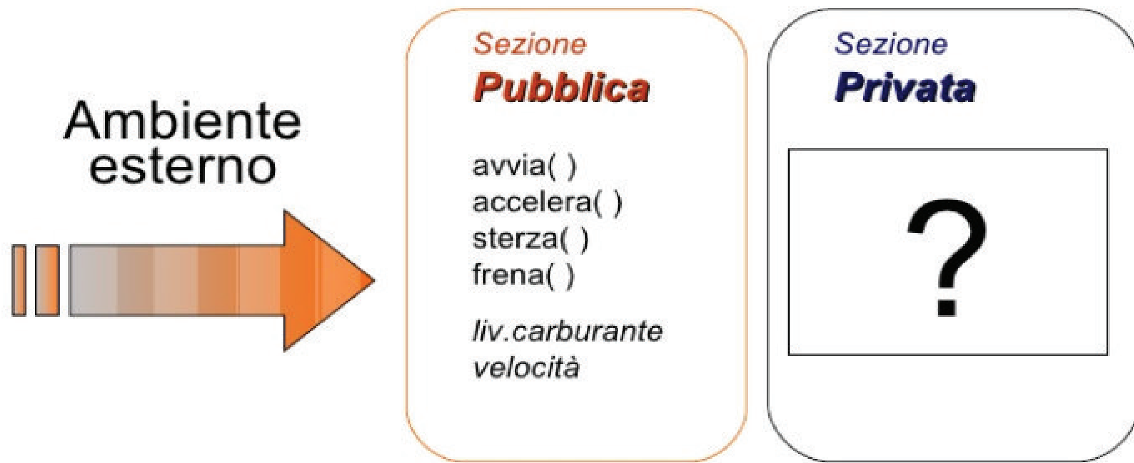
Sezione ***Privata*** — ***attributi*** *e* ***metodi*** che ***non sono accessibili*** ad altri oggetti *(e quindi si rendono invisibili all'esterno)*

L'interfaccia verso l'esterno

Un oggetto può essere utilizzato ***inviando ad esso dei messaggi***

L'insieme dei messaggi rappresenta ***l'interfaccia*** di quell'oggetto

L'interfaccia ***non consente di vedere*** come sono implementati i metodi, ma ne permette il loro utilizzo e l'accesso agli attributi pubblici



Programmazione OOP

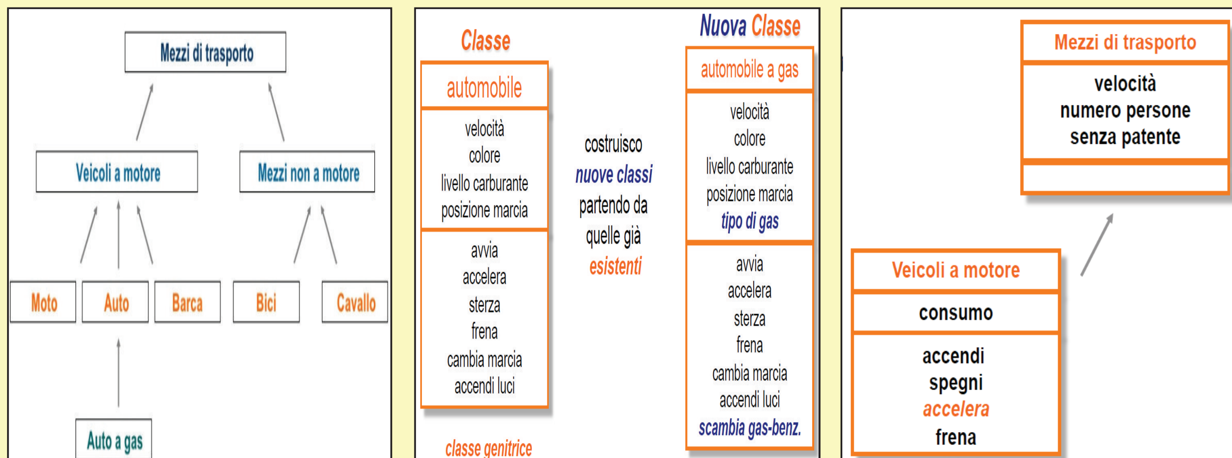
Principali proprietà

2. Ereditarietà

Uno strumento molto efficace nella programmazione ad oggetti è l'**ereditarietà**, che consente di:

definire una ***nuova classe*** che mantiene le proprietà di una classe già esistente ,

ma ***aggiunge nuovi attributi e metodi***.



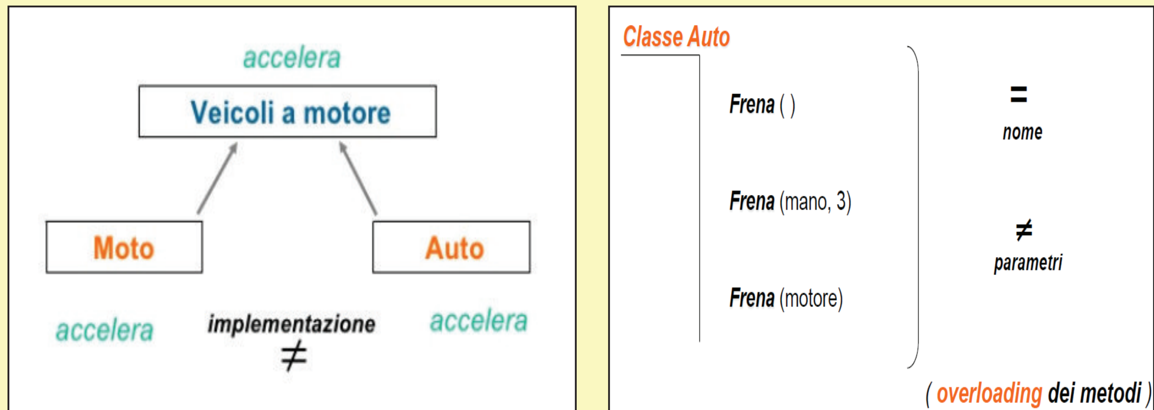
Programmazione OOP

Principali proprietà

3. Polimorfismo

Il **polimorfismo** è la possibilità di richiamare un unico metodo che avrà un comportamento diverso in base al tipo di oggetto su cui viene applicato.

Questo è reso possibile grazie all'**ereditarietà** e all'**overloading**: possiamo definire un metodo con lo stesso nome su due classi. Richiamando il nome del metodo otterremo risultati diversi in base al tipo di oggetto su cui è stato richiamato.



Classi e oggetti, attributi e metodi nei diagrammi UML

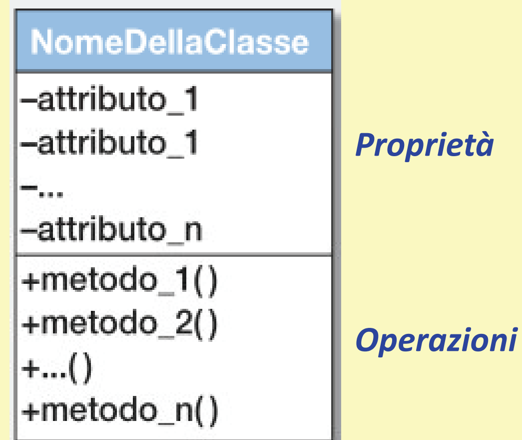
Tra i vari formalismi grafici che UML mette a disposizione per descrivere i diversi aspetti di un sistema software a molteplici livelli di dettaglio, uno in particolare fornisce una notazione grafica per formalizzare le classi e le relazioni che intercorrono tra di esse: il **diagramma delle classi** (class diagram):

È inoltre possibile classificare le componenti:

	simbolo
private	–
pubbliche	+
protette	#

premettendo ai singoli nomi rispettivamente i simboli:

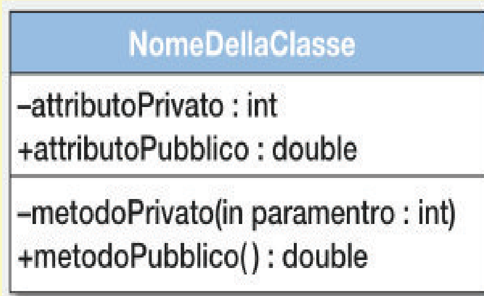
«–» «+» «#»



Proprietà

Operazioni

UML: diagramma delle classi



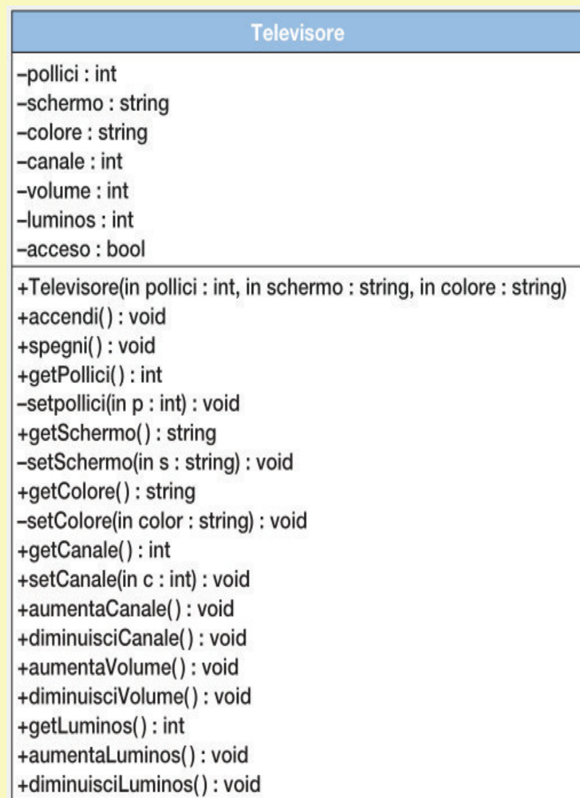
Per ogni metodo, oltre al suo livello di visibilità (pubblica o privata) è necessario definire:

- i **nomi** e i **tipi degli eventuali parametri** e il **loro ruolo** (in/out/in-out);
- il **tipo dell'eventuale valore restituito** (che viene scritto dopo le parentesi e il simbolo ‘ **:** ’).

Esempio

Classe Televisore

- Tutti gli attributi sono privati e tutti i metodi sono pubblici;
- per accedere agli attributi privati sono stati definiti metodi convenzionalmente noti come **getter/setter**:
  - Un metodo il cui nome inizia con il prefisso **get**, seguito dal nome di un attributo della classe, ***restituisce il valore dell'attributo***.
  - Un metodo il cui nome inizia con il prefisso **set**, seguito dal nome di un attributo, ha lo scopo di ***impostare un nuovo valore*** la cui congruità viene controllata dal metodo stesso.



Esempio

Classe Televisore

- Non per tutti gli attributi sono stati definiti dei metodi setter: in particolare alcuni attributi di un televisore non sono modificabili nel corso della sua esistenza e possono essere impostati solo dal **metodo costruttore** **che assume il nome della stessa classe**.

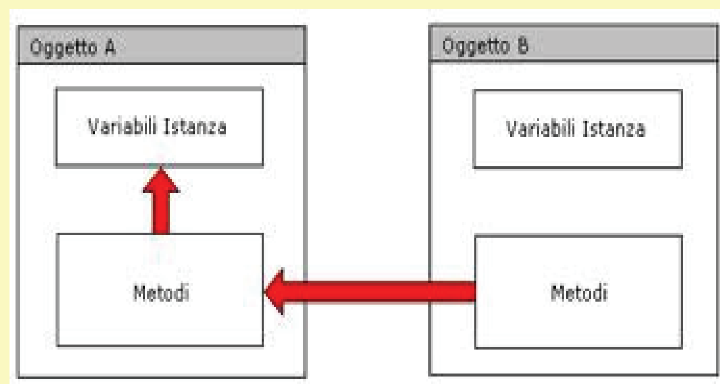
- I metodi che consentono di modificare gli attributi aumentandone o diminuendone il valore devono essere implementati in modo che non possano impostare valori inferiori al minimo valore predefinito, o superiori al massimo valore predefinito per ogni attributo.

Televisore
–pollici : int –schermo : string –colore : string –canale : int –volume : int –luminos : int –acceso : bool
+Televisore(in pollici : int, in schermo : string, in colore : string) +accendi() : void +spegni() : void +getPollici() : int –setpollici(in p : int) : void +getSchermo() : string –setSchermo(in s : string) : void +getColore() : string –setColore(in color : string) : void +getCanale() : int +setCanale(in c : int) : void +aumentaCanale() : void +diminuisciCanale() : void +aumentaVolume() : void +diminuisciVolume() : void +getLuminos() : int +aumentaLuminos() : void +diminuisciLuminos() : void

Programmazione OOP

Per costruire un programma orientato agli oggetti occorre:

- Identificare gli **oggetti** che caratterizzano il modello del problema
- Definire le **classi** , indicando gli **attributi** e i **metodi**
- Stabilire come gli oggetti **interagiscono** fra loro attraverso il meccanismo dello **scambio di messaggi**.



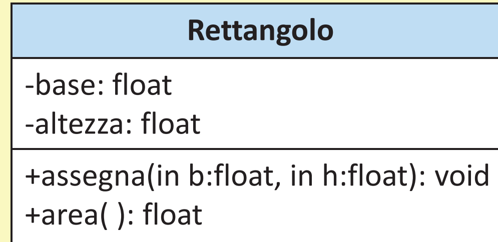
Definizione di una classe in C++

```
class NomeClasse {
    // Attributi
    tipo1  attributo1 ;
    tipo2  attributo2 ;
    ……
    // Metodi
    tipo1  funzione1;
    tipo2  funzione2;
    …..
};
```

Dati membro

Funzioni membro

Diagramma UML



Esempio

```
class  Rettangolo {
    // Attributi
    float  base, altezza ;
public:
    // Metodi
    void assegna(float  b, float h) {
       base = b;
       altezza = h;
    }
    float  area( ) {
       return base*altezza;
    }
};
```

ATTENZIONE:

La clausola **private** è di default.

Per gli attributi e i metodi pubblici si deve specificare **public**.

Dichiarazione di un oggetto di una classe in C++
(creazione di un'istanza)

```
NomeClasse   nomeoggetto;
```

Una volta dichiarati i membri di una classe, è possibile accedere ad essi utilizzando il carattere "."

```
int  main ()
{ Rettangolo   quadro;
  quadro.assegna(0.80 , 0.60);
  cout<<"Il quadro occupa  una superficie
            di "<<quadro.area()<<" mq";
  return 0;
}
```

Nella programmazione OOP la **creazione di un oggetto** ha come conseguenza due azioni consequenziali:

- **allocare** un'area di memoria per la memorizzazione dell'oggetto stesso;
- **inizializzare i valori degli attributi** che costituiscono la componente informativa dell'oggetto.

La seconda azione viene espletata dal **costruttore** della classe, ossia da uno speciale metodo, normalmente denominato con lo **stesso nome della classe**.

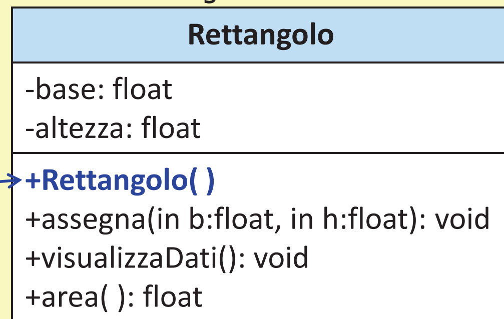
Esempio

```
class Rettangolo {
      // Attributi
      float base, altezza ;
public:
      // Costruttore
      Rettangolo () {
          base = 0.0;
          altezza = 0.0;
      }

      // Metodi
      void assegna(float b, float h) {
         base = b;
         altezza = h;
      }
      void visualizzaDati() {
        cout<<"Base = "<<base<<endl;
        cout<<"Altezza="<<altezza<<endl;
      }
      float area( ) {
         return base*altezza;
      }
};
```

Diagramma UML

costruttore →



Il **costruttore**:

- Ha lo **stesso nome della classe**
- viene **richiamato automaticamente in modo implicito** alla creazione di un'istanza (anche se non è stato dichiarato)
- **non prevede la restituzione di alcun valore** e, di conseguenza, non deve essere specificato alcun tipo per questo metodo.
- Il programmatore può definire all'interno della classe una propria personalizzazione del costruttore inserendo un metodo con lo stesso nome della classe

Esempio

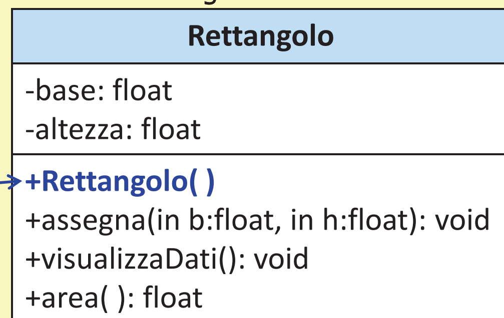
```
class Rettangolo {
      // Attributi
      float base, altezza ;
public:
      // Costruttore
      Rettangolo () {
          base = 0.0;
          altezza = 0.0;
      }
      // Metodi
      void assegna(float b, float h) {
         base = b;
         altezza = h;
      }

      void visualizzaDati() {
        cout<<"Base = "<<base<<endl;
        cout<<"Altezza="<<altezza<<endl;
      }

      float area( ) {
         return base*altezza;
      }
};
```

Diagramma UML

costruttore →




```
int main ( )
{ Rettangolo  quadro;
  quadro.visualizzaDati( );
  quadro.assegna(0.80 , 0.60);
  cout<<"Area="<<quadro.area();
}
```

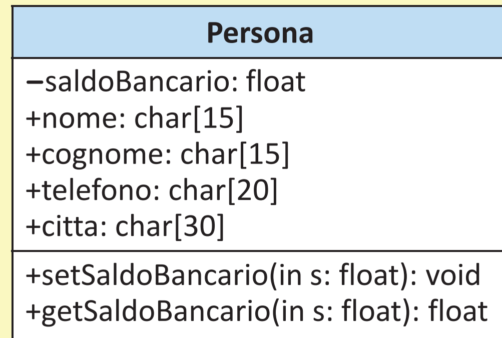
Esercizio

Completare con il metodo ....
perimetro...

Esempio

```
class Persona{
    // Attributi
    float saldoBancario;
public:
    // Attributi
    char nome[15], cognome[15];
    char telefono[20] ;
    char citta[30] ;
    // Metodi
    void setSaldoBancario(float s) {
      if (s<0)
         cout << "Saldo non accettato\n";
      else
         saldoBancario = s;
    }
    float getSaldoBancario( ){
      return saldoBancario;
    }
};
```

Diagramma UML




```
int main ( )
{ Persona Io;
  float saldo;
  cout<<"Inserisci nome: ";
  cin>>Io.nome;
  cout<<"Inserisci cognome: ";
  cin>>Io.cognome;
  …
  cout<<"Inserisci saldo bancario: ";
  cin>>saldo;
  Io.setSaldoBancario(saldo);
……
}
```

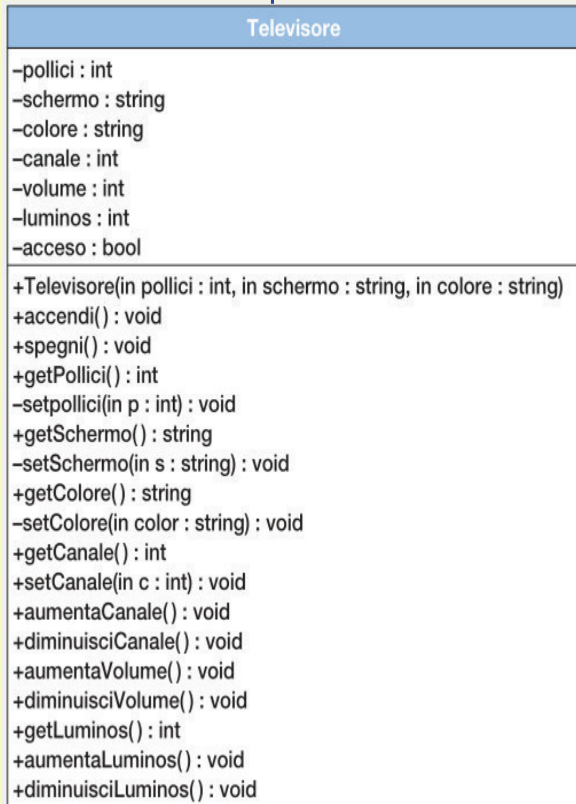
Esercizio

Completare con altri metodi ……..

deposita…

preleva …

Esempio




```
#include <iostream>
using namespace std;

class Televisore {
    // attributi caratteristiche apparecchio
     int pollici;
     string schermo;
     string colore;
    // attributi stato apparecchio
     int canale;
     int volume;
     int luminos;
     bool acceso;

 public:
   // costruttore
     Televisore(int pollici, string schermo, string colore) {
         setPollici(pollici);
         setSchermo(schermo);
         setColore(colore);
         canale = 1;
         volume = 10;
         luminos = 30;
         acceso = false;
    }
```

```
// getter/setter
 private:
   void setPollici(int p) {pollici = p;}
   void setColore(string c) {colore = c;}
   void setSchermo(string s) {schermo = s;}

 public:
   int getPollici() {return pollici;}
   string getColore() {return colore;}
   string getSchermo(){return schermo;}
   int getCanale(){return canale;}
   int getVolume(){return volume;}
   int getLuminos(){return luminos;}
   void setCanale(int c) {if (c>0 && c<99) canale = c;}
   bool isAcceso() {return acceso;}

// operazioni
 public:
   void accendi() {acceso = true;}
   void spegni(){acceso = false;}
   void aumentaCanale()
     {
      if (canale<99) canale++;
     }
   void diminuisciCanale()
     {
      if (canale>0) canale--;
     }
```

```
void aumentaVolume()
    {
     if (volume<50) volume++;
    }
  void diminuisciVolume()
    {
     if (volume>0) volume--;
    }
  void aumentaLuminos()
    {
     if (luminos<80) luminos ++;
    }
  void diminuisceLuminos()
    {
     if (luminos>0) luminos--;
    }
};
```

```
int main () {
 Televisore tcucina= Televisore(32,"LED","nero");
 tcucina.accendi();
 tcucina.canaleSuccessivo();
 tcucina.aumentaVolume();
 cout<<tcucina.getColore()<<endl;
 cout<<tcucina.getCanale()<<endl;
 cout<< tcucina.getVolume()<<endl;
}
```


Esercizi

Per ciascuno dei seguenti problemi disegnare il diagramma delle classi (UML), rispettando rigorosamente la sintassi, e implementare il corrispondente programma in C++ utilizzando la tecnica di programmazione ad oggetti:

1) Realizzare un contatore modulo N che si possa incrementare e decrementare di una unità ripetutamente
2) Si vuole rappresentare un punto a coordinate reali nel piano cartesiano. Di un certo punto si vuole sapere in quale quadrante si trova. Il punto può essere traslato di un deltax e deltay. Prevedere un costruttore senza parametri per istanziare il punto origine degli assi cartesiani, un costruttore personalizzato che crea un punto sull’asse delle ascisse e un costruttore personalizzato per istanziare un punto qualsiasi con coordinate assegnate da tastiera.
3) Implementare la classe cerchio caratterizzato dalle coordinate del centro nel piano cartesiano e dalla misura del raggio. (Il costruttore di default istanzia un cerchio con centro nell’origine degli assi cartesiani).
4) Simulare il comportamento di un ascensore che serva un edificio di tre piani. In particolare il programma deve gestire i comandi impartiti dalla pulsantiera interna. La pulsantiera prevede di selezionare il numero del piano a cui andare. L’ascensore quando è fermo ad un piano ha le porte aperte.