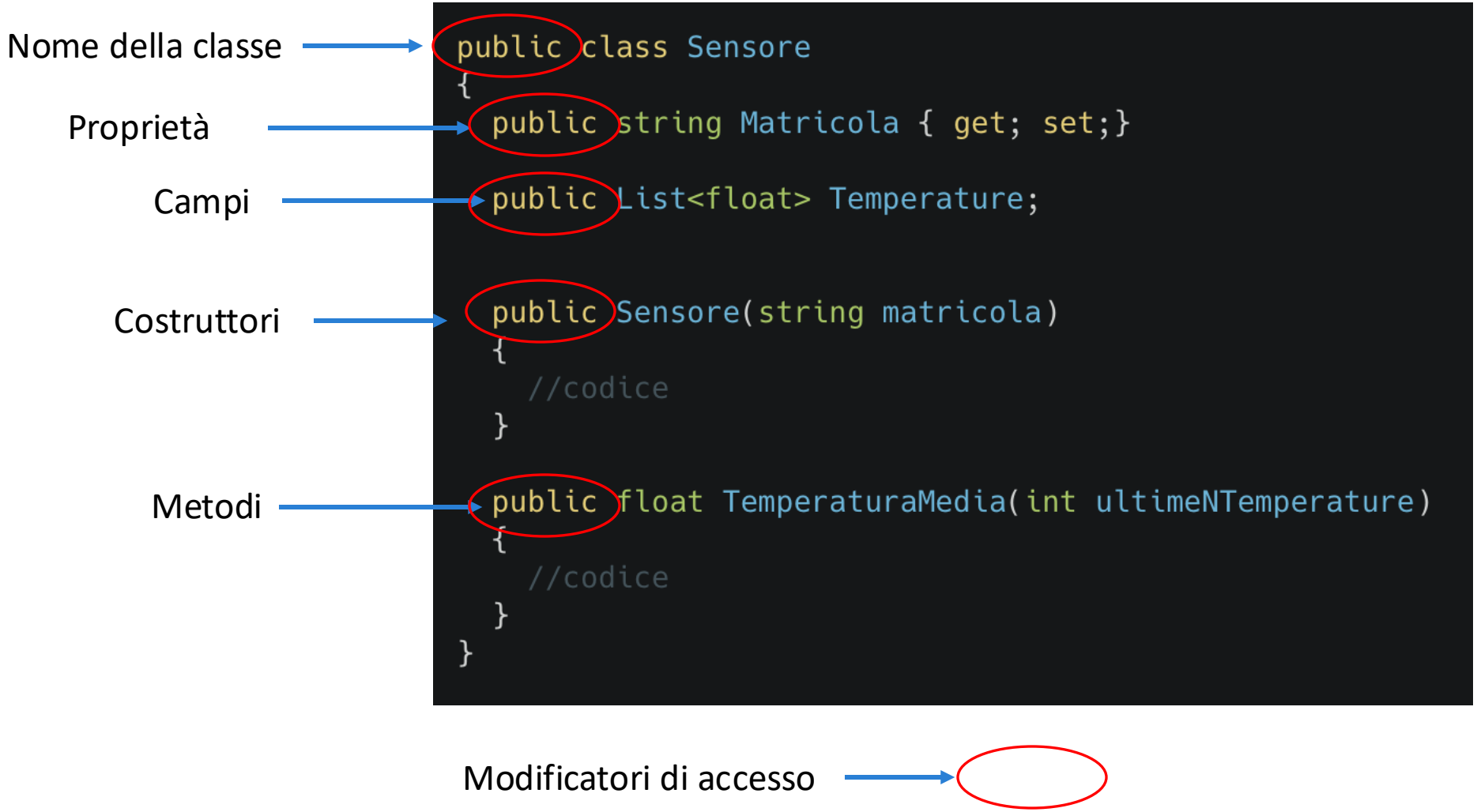
Object Oriented Programming (OOP)

Anatomia di una classe



Classi - campi

I campi sono la forma più semplice con cui una classe può contenere dati.

Si definiscono all'interno della classe, utilizzando le stesse modalità con le quali si definisce una qualsiasi variabile.

La definizione del campo deve essere fatta precedere dall'access modifier che ne determina le modalità di utilizzo (lo vedremo in seguito)

```
public class Studente
{
  //Campi
  public string Matricola;
  public string Cognome;
  public string Nome;
  public DateTime DataNascita;
  public List<byte> Voti;
}
```

Classi - campi

I campi possono essere inizializzati direttamente nella definizione o successivamente tramite costruttore.

```
public class Studente
{
  //Campi
  public string Matricola;
  public string Cognome;
  public string Nome;
  public DateTime DataNascita;
  public int EsamiPrevisti=10;
  public List<byte> Voti;
}
```

Classi - costruttori

Sono metodi speciali che vengono eseguiti esclusivamente nel momento in cui un oggetto viene creato tramite la parola chiave `new`.

Sono utilizzati per inizializzare l'oggetto, eventualmente configurandolo con informazioni parametriche.

E' possibile avere 0 o più costruttori, che si devono differenziare esclusivamente per il numero/tipo di parametri utilizzati (overloading)

Il costruttore ha lo stesso nome della classe seguito dall'elenco dei parametri separati da virgola. Per ciascun parametro deve essere specificato il tipo.

La definizione del costruttore deve essere fatta precedere dall'access modifier che ne determina le modalità di utilizzo (lo vedremo in seguito).

Classi - costruttori

```
public class Studente
{
  //Costruttori
  public Studente() //costruttore vuoto
  {
    //Codice del costruttore vuoto
  }

  public Studente(string matricola)
  {
    //Codice del costruttore
  }

  public Studente(string matricola, string Cognome, string Nome)
  {
    //Codice del costruttore
  }
}
```

Il costruttore che verrà utilizzato dipende dalla modalità di creazione dell'oggetto:

```
Studente s1 = new Studente();  //utilizzo del costruttore vuoto

Studente s2 = new Studente("M001") //utilizzo del costruttore con 1 parametro
```

Classi - costruttori

All'interno del costruttore le variabili passate come parametro possono essere utilizzare per inizializzare campi e proprietà definite nella stessa classe.

```
public class Studente
{
  //Campi
  public string Matricola;

  //Costruttori
  public Studente(string matricola)
  {
    //Codice del costruttore
    Matricola = matricola;
  }
}
```

Classi - costruttori

Nel caso di omonimia tra il nome del campo e il nome del parametro, si può utilizzare l'oggetto generico **this** che rappresenta la classe su cui si sta operando

```
public class Studente
{
  //Campi
  public string Matricola;

  //Costruttori
  public Studente(string Matricola)
  {
    //Codice del costruttore
    this.Matricola = Matricola;
  }
}
```

Classi - costruttori

L'utilizzo dei costruttori non è obbligatorio. E' possibile creare classi che non ne implementano.

Tuttavia, in questi casi, verrà comunque creato un costruttore di default, senza parametri, che non fa nulla, da utilizzarsi comunque per la creazione di oggetti

```
public class ClasseSenzaCostruttore
{
  //Campi
  public string CampoDellaClasse;

}


ClasseSenzaCostruttore IstanzaDellaClasse = new ClasseSenzaCostruttore();
```

Classi – concatenare i costruttori

Per evitare di duplicare codice all'interno dei costruttori è possibile creare costruttori che utilizzano il codice di altri costruttori.

Il caso d'uso più frequente è quello di lasciare al costruttore vuoto gran parte dell'inizializzazione, gestendo negli altri costruttori solamente la parte che dipende dai parametri

```
public class Studente
{
  //Campi
  public DateTime CreazioneOggetto;
  public string Matricola;

  //Costruttori
  public Studente()
  {
    this.CreazioneOggetto = DateTime.Now;
  }

  public Studente(string Matricola)
  {
    this.CreazioneOggetto = DateTime.Now;
    this.Matricola = Matricola;
  }
}
```

```
public class Studente
{
  //Campi
  public DateTime CreazioneOggetto;
  public string Matricola;

  //Costruttori
  public Studente()
  {
    this.CreazioneOggetto = DateTime.Now;
  }

  public Studente(string Matricola) : this()
  {
    this.Matricola = Matricola;
  }
}
```

Classi – Object initializer

Esiste una modalità alternativa all'utilizzo dei costruttori che consiste nello specificare i valori da assegnare a campi e proprietà:

```
Studente s = new Studente { Matricola = "M001"}
```

Non è obbligatorio specificare i valori per tutte i campi / proprietà.

Prima dell'assegnazione dei valori viene comunque eseguito il costruttore di default, quello senza parametri.

Classi – Access Modifiers

E' possibile gestire la "visibilità" e, quindi, l'utilizzabilità di classi, campi, proprietà e metodi.

E' cioè possibile specificare in quale contesti la classe, il campo, la proprietà e il metodo può essere utilizzata.

L'access modifier va inserito prima della definizione della classe, del campo, della proprietà o del metodo. Sono possibili 5 valori:

- public
- private
- protected
- internal (default)
- protected internal

Classi – Access Modifiers – public e private

L'access modifier **public** rende la classe, il campo, la proprietà e il metodo visibili ed utilizzabili in ogni punto del programma.

L'access modifier **private** rende il campo, la proprietà e il metodo visibili ed utilizzabili solamente all'interno della classe in cui sono stati definiti. Private non può pertanto essere utilizzato su una classe.

L'utilizzo combinato di public e private permettono di realizzare classi che espongono un comportamento esterno (identificato con public) mentre nascondono l'implementazione interna (identificata con private).

PRIMO PRINCIPIO DELLA OOP → INCAPSULAMENTO

Classi – Access Modifiers – public e private

```
public class Studente
{
  //Campi
  private DateTime creazioneOggetto
  public string Matricola;

  //Costruttori
  public Studente()
  {
    this.creazioneOggetto = DateTime.Now;
  }
}

Studente s = new Studente();
s.Matricola = "M001"; //OK
Console.WriteLine(s.creazioneOggetto); //Errore
```

Classi – Proprietà

L'utilizzo di campi public non è adatto a coprire tutte le necessità che una classe può avere:

- Non è possibile limitare il valore assunto da un campo numerico (per esempio limitare il voto da 0 a 30)
- Più in generale non è possibile effettuare verifiche sui dati inseriti. Il campo accetta tutti i valori previsti dalla sua tipologia
- Non è possibile realizzare campi in sola lettura

Per coprire questi casi si utilizzano le proprietà, composte da:

- 1 campo privato destinato a contenere il dato
- 1 proprietà pubblica, associata al campo privato, che può eseguire una porzione di codice tutte le volte in cui c'è un accesso, sia in lettura che in scrittura.

Classi – Proprietà

```
public class Voto
{
   private int _punteggio;

   public int Punteggio
   {
       get
       {
         return this._punteggio;
       }
       set
       {
          this._punteggio = value;
       }
   }
}
```

I due blocchi di codice sono identificati dalle parole chiave get e set.

Il blocco get (getter) è eseguito quando la proprietà viene letta.

Il blocco set (setter) è eseguito quando la proprietà viene scritta. In questo blocco è disponibile la variabile value contenente il valore che si vuole scrivere.

Sia in get che in set è possibile accedere a tutti i campi della classe.

All'interno di get e set è possibile implementare tutte le logiche di controllo desiderate. E' anche possibile ometterne uno dei due ottenendo proprietà in sola lettura (se omettiamo il set) o in sola scrittura (se omettiamo il get).

Classi – Proprietà

```
public class Voto
{
    private int _punteggio;

    public int Punteggio
    {
        get
        {
          return this._punteggio;
        }
        set
        {
            if( value > 30 )
                this._punteggio = 30;
            else
                this._punteggio = value;
        }
    }
}
```

Classi – Proprietà

Nel caso di proprietà "standard", con il getter che restituisce il campo privato e il setter che lo valorizza, è possibile utilizzare una sintassi più compatta.

Non è necessario definire la variabile privata e non è necessario definire il codice di getter e setter, ma solamente le parole chiave get e set.

```
public class Voto
{
   private int _punteggio;

   public int Punteggio
   {
       get
       {
         return this._punteggio;
       }
       set
       {
          this._punteggio = value;
       }
   }
}
```

equivalente

```
public class Voto
{
   public int Punteggio { get; set; }
}
```

Classi - Metodi

Sono i costrutti con cui una classe implementa funzionalità. Un metodo esegue una elaborazione e contiene pertanto delle istruzioni.

Un metodo ha inoltre le seguenti caratteristiche:

1. Può accedere ai campi, alle proprietà e ai metodi definiti nella stessa classe.
2. Può avere dei **parametri** da utilizzare all'interno del codice eseguibile.
3. Può restituire un valore di qualsiasi tipo all'istruzione che ha eseguito il metodo (valore di ritorno)

Il nome di un metodo, unitamente all'elenco dei parametri e al valore di ritorno è detta firma, o **signature**, del metodo. Non possono esistere due metodi con stessa firma. Possono invece esistere due metodi con lo stesso nome a patto che abbiamo parametri differenti

Classi - Metodi

Un metodo si definisce specificando:

- Access modifier
- Tipo del valore di ritorno
- Nome del metodo
- Elenco dei parametri tra parentesi tonde e separati da virgola
  - Ogni parametro è specificato con tipo e nome
- Codice da eseguire racchiuso tra { }

```
public class Studente
{

  //metodo
  public float MediaVoti()
  {
    //implementazione del metodo senza parametri
  }

  public float MediaVoti(bool mediaPesata)
  {
    //implementazione del metodo con 1 parametro boolen
  }

}
```

I parametri diventano variabili utilizzabili all'interno del metodo. Hanno pertanto il metodo come scope.

Classi – Metodi – Valori di ritorno

All'interno di un metodo è possibile utilizzare tutti le istruzioni C#, anche utilizzando librerie e oggetti di sistema.

E' possibile accedere ai campi, alle proprietà presenti nell'oggetto ed eseguire gli altri metodi.

Il valore di ritorno viene restituito tramite la parola chiave **`return`**, seguita dal valore da restituire.

Return interrompe immediatamente l'esecuzione del metodo, generando il valore di ritorno e restituendo il controllo al chiamante. Il chiamante rimante pertanto in attesa del completamento del metodo.

Qualora non ci sia la necessità di restituire un valore di ritorno, si può utilizzare la parola chiave `void`, nella signature del metodo, al posto del tipo di ritorno. Return continua ad essere utilizzato per terminare l'esecuzione del metodo.

Classi – Metodi – Valori di ritorno

```
Studente s = new Studente();

//...
//Codice che aggiunge voti
//...

float media = s.MediaVoti();
Console.WriteLine(media);
```

```
public class Studente
{
  //campo privato
  List<int> _voti = new List<int>();

  //metodo
  public float MediaVoti()
  {
    float media = 0;
    foreach( int voto in _voti ) media += voto;
    return (float) media / _voti.Count;
  }

}
```

Classi – Passaggio dei parametri

Il passaggio dei parametri ad un metodo avviene con le modalità previste per il tipo di dato interessato:

- Value Type: viene passata una copia del valore
- Reference Type: viene passata una copia del riferimento all'area di memoria heap contenente il valore.

La modifica, all'interno di un metodo, di una variabile reference type può ripercuotersi anche all'esterno del metodo.

La stessa cosa accade per il valore di ritorno:

- Value Type: viene restituita una copia del valore
- Reference Type: viene restituita una copia del riferimento all'area di memoria heap contenente il valore.

Classi – Passaggio dei parametri

```
public class Test
{
    public bool Incrementa(int valore)
    {
        valore = valore + 1;
        return true;
    }
    public bool Incrementa(DateTime valore)
    {
        valore = valore.AddDays(1);
        return true;
    }
    public bool Incrementa(string valore)
    {
        valore = valore + "!";
        return true;
    }

    public bool Incrementa(int[] valori)
    {
        for (int i = 0; i < valori.Length; i++)
          valori[i] = valori[i] + 1;
        return true;
    }

    public bool Incrementa(Studente valore)
    {
         valore.Matricola = valore.Matricola + "!";
         return true;
    }
}
```

Cosa viene stampato su console in questi casi?

```
Test t = new Test();

int valore1 = 10;
t.Incrementa(valore1);
Console.WriteLine(valore1);

DateTime valore2 = DateTime.Today;
t.Incrementa(valore2);
Console.WriteLine(valore2);

String valore3 = "Hello";
t.Incrementa(valore3);
Console.WriteLine(valore3);

int[] valori4 = new int[3] { 10, 20, 30 };
t.Incrementa(valori4);
foreach(int valore in valori4) Console.WriteLine(valore);

Studente valore5 = new Studente("M001");
t.Incrementa(valore5);
Console.WriteLine(valore5.Matricola);
```

Classi statiche

Metodi, campi e proprietà possono essere contrassegnati con la parola chiave `static` per poterli utilizzare anche senza prima creare un oggetto.

```
public class Voto
{
  public int Punteggio { get; set; }

  public static int PunteggioMax { get; } = 30;
}


Console.WriteLine($"Il punteggio massimo è {Voto.PunteggioMax}");
```

Una classe può essere forzata a contenere solamente elementi static contrassegnando anche la classe con la stessa parola chiave.

Classi - Ereditarietà

L'ereditarietà (inheritance) è una relazione tra due classi che permette ad una classe di ereditare il comportamento da un'altra classe, per poi estenderlo.

La relazione tra due classi è di tipo is a: se la classe B eredita dalla classe A possiamo dire che B is a A. Esempio:

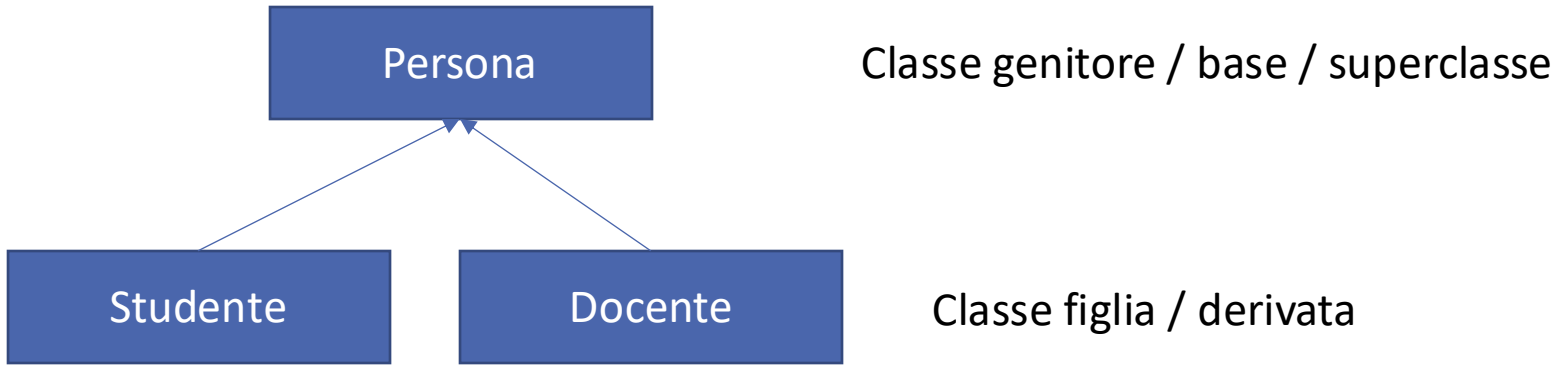
- Classe Persona
- Classe Studente che eredita da Persona → Studente is a Persona

Tramite l'ereditarietà abbiamo benefici in termini di:

- Riuso del codice: implemento del codice in una classe e lo riuso nelle classi che ereditano
- Polimorfismo: lo vedremo in seguito

Classi - Ereditarietà

L'ereditarietà tra classi può essere rappresentata graficamente:



NB: Una classe può ereditare al massimo da una superclasse. Inoltre, in .NET, tutte le classi ereditano da una unica classe base `Object`.

Classi - Ereditarietà

Dal punto di vista sintattico l'ereditarietà avviene specificando il nome della classe genitore da cui si vuole ereditare:

```
public class Persona
{
  public string Cognome { get; set; }

  public string Nome { get; set; }

  public string CodiceFiscale { get; set; }

  public DateTime DataNascita { get; set; }

  public int Età()
  {
    return (DateTime.Today - DataNascita).Days / 365;
  }
}
```

```
public class Studente : Persona
{
  List<Voto> Voti { get; set; }
}
```

```
Studente s = new Studente();
s.Cognome = "Rossi";
```

Classi – Ereditarietà e access modifiers

Una classe derivata NON può accedere alla parte private della classe base.

Metodi, campi e proprietà che utilizzano l'access modifier **protected** possono essere utilizzati all'interno della classe che li definisce (come private) ma anche all'interno delle classi derivate.

ATTENZIONE: Limitare l'utilizzo di protected solamente ai casi in cui è indispensabile. In tutti gli altri casi evitare di utilizzarlo, in quanto "espone" l'implementazione interna della superclasse.

Classi – Ereditarietà e costruttori

I costruttori della classe base e della classe derivata sono indipendenti l'uno dall'altro e NON si ereditano.

La classe base definisce i propri costruttori e le classi derivate definiscono i propri.

In ogni caso i costruttori della classe base vengono eseguiti prima di quelli della classe derivata.

Non potendo accedere alla parte privata, la classe derivata costruisce la classe base sfruttando i suoi costruttori. Se non diversamente specificato si utilizza il costruttore vuoto

```
public class Persona
{
  public string CodiceFiscale { get; set; }

  public Persona(string CodiceFiscale)
  {
    this.CodiceFiscale = CodiceFiscale;
  }
}
```

```
public class Studente : Persona
{
  List<Voto> Voti { get; set; }

  public Studente(string CodiceFiscale, List<Voto> Voti) : base(CodiceFiscale)
  {
    this.Voti = Voti;
  }
}
```

Classi – Override dei metodi

Ci sono casi in cui una classe derivata necessità di modificare il comportamento di un metodo della classe base.

Il metodo mantiene la stessa firma (altrimenti sarebbe semplicemente un nuovo overload dello stesso metodo) ma implementa un comportamento differente.

Non è possibile effettuare l'override di tutti i metodi ma solamente di quelli che lo prevedono, tramite la parola chiave `virtual`. Il metodo della classe derivata deve inoltre utilizzare la parola chiave `override` e può utilizzare i metodi della superclasse tramite la parola chiave `base`.

```
public class Persona
{
  public virtual string GetInfo()
  {
    //Dfault implementation
  }
}
```

```
public class Studente : Persona
{
  public override string GetInfo()
  {
    //Override implementation
  }
}
```

Classi – Polimorfismo

Il polimorfismo, che significa "multiforme" è il terzo pilastro della OOP, insieme all'incapsulamento e l'ereditarietà.

Grazie al polimorfismo, sfruttando la relazione di ereditarietà is a, oggetti di una classe derivata possono essere trattati come oggetti della classe base (o di una classe antenata)

```
public class Persona
{}

public class Studente : Persona
{}

Persona p = new Studente();

List<Persona> l = new List<Persona>();
l.Add(new Persona());
l.Add(new Studente());
```

Classi – Polimorfismo

```
public class Persona
{
  public virtual void ChiSono()
  {
     Console.WriteLine("Persona");
  }
}

public class Studente : Persona
{
  public virtual void ChiSono()
  {
     Console.WriteLine("Studente");
  }

  public float MediaVoti()
  {
    //...
  }
}

Persona p = new Studente();

p.MediaVoti(); //ERRORE!
p.ChiSono(); //Cosa stampa??
```

Quando tratto un oggetto di un classe derivata come fosse di una classe base, grazie al polimorfismo, posso utilizzare solamente metodi e proprietà definiti nella classe base.

In caso di override viene però eseguito il codice della classe derivata.

Classi Abstract e Sealed

Talvolta è necessario controllare le possibilità di una classe di essere estesa. Possiamo avere due casi:

- Classi che devono necessariamente essere estese e non possono essere istanziate direttamente. → classi abstract
- Classi che non possono essere estese → classi sealed

Entrambi i comportamenti vengono specificati, rispettivamente, con le parole chiave abstract e sealed inserite nella definizione della classe, dopo l'access modifier

```
public abstract class Libro
{

}
```

```
public sealed class Studente
{

}
```

Classi Abstract

Nelle classi abstract è possibile contrassegnare contrassegnare come abstract anche singoli metodi.

In tal caso si rende obbligatorio l'override nelle classi derivate e non è necessario fornirne l'implementazione.

```
public abstract class Libro
{
  public abstract string GetInfo();
}
```

Classi – Interfacce

La creazione di una classe astratta pura, composta solo da metodi abstract, può essere utilizzata come una sorta di "contratto". Specifica infatti solamente la firma che le classi derivate devono rispettare (ed estendere) senza però determinare alcun comportamento.

Per rappresentare meglio questo comportamento esiste il concetto di interfaccia.

L'interfaccia permette di specificare metodi, campi e proprietà che una classe deve implementare se decide di aderire all'interfaccia. Ci sono due grandi vantaggi rispetto alle classi abstract:

- Classi che non sono in relazione di ereditarietà diretta possono implementare la stessa interfaccia (e quindi possono essere considerate omogenee rispetto all'interfaccia)
- Una classe può aderire a più interfacce contemporaneamente

Classi – Interfacce

Le interfacce si definiscono utilizzando la parola chiave interface. E' consuetudine farne iniziare il nome con una I maiuscola in modo da meglio identificarle.

Una classe può aderire ad una interfaccia specificandola in maniera analoga alla classe base da cui ereditare (eventualmente separando con virgole)

```
public interface IGetInfo
{
  public override string GetInfo();
}

public class LibroCartaceo : Libro, IGetInfo
{
  public override string GetInfo();
}

public class Studente : Persona, IGetInfo
{
  public override string GetInfo();
}
```

```
List<IGetInfo> list = new List<IGetInfo>() {
    new LibroCartaceo(...),
    new Studente(...)
}

foreach(IGetInfo i in list)
{
  Console.WriteLine(i.GetInfo());
}
```

Struct

Struct

Le struct sono strutture dati apparentemente simili alle classi, in quanto:

- Incapsulano campi, proprietà e metodi
- Prevedono access modifiers
- Hanno costruttori
- Possono avere metodi statici
- Possono implementare interfacce

Struct

```
public struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; set; }
    public double Y { get; set; }

    public string Print()
    {
      return $"({X}, {Y})";
    }
}


Coords point = new Coords(10, 10);
```

Struct

In realtà presentano alcune differenze "sostanziali":

- Per definire una struttura si usa la parola chiave `struct`, anziché `class`
- Le strutture sono value types, mentre le classi sono reference types
  - Allocazione nello stack anziché nell'heap
- Per istanziare una struttura non è necessaria la parola chiave new
- Le strutture non prevedono l'ereditarietà
  - Non possono essere utilizzati i modificatori protected e protected internal
  - Le strutture non possono essere abstract o sealed
- Le strutture possono avere solamente costruttori con parametri
- Le strutture non possono avere distruttori

Struct

```
public struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; set; }
    public double Y { get; set; }

    public string Print()
    {
      return $"({X}, {Y})";
    }
}


Coords point1 = new Coords(10, 10);
Coords point2 = point1;
point2.X = 20;

point1.Print();
point2.Print();
```

```
public struct Coords
{
    public double X;
    public double Y;

    public string Print()
    {
      return $"({X}, {Y})";
    }
}


Coords point;

point.X = 20;
point.Y = 20;
```

Class vs Struct

L'utilizzo delle struct è più snello rispetto alle classi ed ha prestazioni migliori grazie all'allocazione nello stack.

E' pertanto preferibile utilizzare strutture per modellare situazioni semplici che non richiedono le caratteristiche esclusive delle classi (ereditarietà in primis).