

A.S. 2024/25

CLASSE 4, ITI INDIRIZZO INFORMATICA
PROF. FRANCO SICCA

Manuale di C# e Programmazione ad Oggetti con Windows Forms

MANUALE SVILUPPATO CON IL SUPPORTO DI CHATGPT 4.0

Introduzione

Manuale di C# e Programmazione ad Oggetti con Windows Forms	1
Introduzione	1
Capitolo 1: Introduzione a C#	3
Capitolo 2: Fondamenti di C#	4
1. if	4
2. switch	5
3. for	7
4. while	8
5. do-while	8
Osservazioni	9
6. STAMPA LA TABELLINA IN C++ E IN C#. GESTIONE INPUT/OUTPUT IN C#	10
7. USO DI FOREACH SU UN ARRAY E SU UNA LISTA	14
7.1 GESTIONE DEGLI ARRAY	14
7.2 USO DI FOREACH SU UN ARRAY	16
7.3 USO DI FOREACH SU UNA LISTA	18
7.5 RIDIMENSIONARE UN ARRAY	20
8. USO DI TRY CATCH PER LA GESTIONE DEGLI ERRORI	21
Capitolo 3: Definizione di Classi	25
Capitolo 4: Concetti Base della Programmazione ad Oggetti	31
Capitolo 5: Incapsulamento	41
Capitolo 6: Ereditarietà e Polimorfismo	45
6.1 LA CLASSE CONVERT	46
6.2 LA CLASSE STRINGBUILDER	48
6.3 LE CLASSI FILESTREAM, STREAMREADER E STREAMWRITER	52

6.4 OVERRIDE DI UN METODO	57
VIRTUAL VS ABSTRACT	59
1. virtual	59
Esempio con virtual	59
2. abstract	60
Esempio con abstract	60
Riepilogo delle differenze	61
Capitolo 7: Introduzione a Windows Forms	61
Capitolo 8: Controlli di Base	62
Capitolo 9: Eventi e Gestione degli Eventi	63
Capitolo 10: Layout e Design	64
Capitolo 11: Esempi Pratici	65
Capitolo 12: Gestione di Dati con Liste e Array	66
Capitolo 13: Persistenza dei Dati	67
Capitolo 14: Applicazioni Multi-form	69
Capitolo 15: APPROFONDIMENTI	70
LINK ESTERNI	70
LE VARIABILI	70
Ereditarietà	70
IL POLIMORFISMO	70
Esempio di Polimorfismo Dinamico con Override dei Metodi	71
LE CLASSI ASTRATTE	72
16. ESERCIZI DI TIPO CONSOLE	74
ESERCIZIO STAMPA LA TABELLINA	74
18. ESERCIZI CON WINDOWS FORM	77
18.1 ESERCIZIO GENERATORE PASSWORD	77
18.2 ESERCIZIO GESTIONE PICTURE BOX	82
18.3 ESERCIZIO GESTIONE TIMER	82
18.4 ESERCIZIO PROGRAMMAZIONE ASINCRONA : BLACKJACK	83
18.5 ESERCIZIO DI GESTIONE DELLE IMMAGINI : TRIS	94
18.6 ESERCIZIO PER APRIRE E GESTIRE 2 FORM	97
18.7 ESERCIZIO PER QUESTIONARIO	98
18.8 ESERCIZIO PER VEICOLO	102
18.9 ESERCIZIO PER CONVERTITORE DA BINARIO A DECIMALE	109

Capitolo 1: Introduzione a C#

Questo manuale fornisce una base per comprendere C#, la programmazione ad oggetti, e lo sviluppo di applicazioni desktop con Windows Forms. Ogni capitolo include esempi di codice e attività pratiche per rafforzare l'apprendimento. È rivolto a studenti che conoscano le basi della programmazione (le variabili, i costrutti condizionali, le iterazioni, e che abbiano sviluppato un semplice programma in un linguaggio qualsiasi, meglio se in C++ o java)

Storia di C# C# è un linguaggio di programmazione sviluppato da Microsoft come parte della piattaforma .NET. È stato progettato per essere semplice, moderno, e orientato agli oggetti.

Caratteristiche del linguaggio C# combina la potenza di linguaggi come C++ con la semplicità di linguaggi come Visual Basic. È strettamente tipizzato e supporta l'OOP, il che facilita la gestione di progetti complessi.

Ambiente di sviluppo (Visual Studio) Visual Studio è l'IDE principale per sviluppare applicazioni C#. Offre strumenti per il debugging, il testing e il deployment delle applicazioni.

OOP: OBJECT ORIENTED PROGRAMMING

OOP, o programmazione orientata agli oggetti, è un paradigma di programmazione che utilizza "oggetti" e le loro interazioni per progettare applicazioni e programmi. Questo paradigma si basa su diversi concetti chiave:

1. **Oggetti:** Entità che combinano stato (attributi o proprietà) e comportamento (metodi o funzioni). Gli oggetti rappresentano istanze di classi.
2. **Classi:** Definizioni generiche o modelli per creare oggetti. Una classe definisce un tipo di oggetto specifico, includendo attributi e metodi che l'oggetto possiederà.
3. **Incapsulamento:** La pratica di raggruppare dati (attributi) e metodi che operano su quei dati all'interno di una classe, nascondendo i dettagli dell'implementazione e consentendo l'accesso solo attraverso metodi pubblici (getter e setter).
4. **Ereditarietà:** Un meccanismo per creare nuove classi (sottoclassi) che derivano da classi esistenti (superclassi). Le sottoclassi ereditano attributi e metodi dalle superclassi, ma possono anche aggiungere o sovrascrivere comportamenti specifici.
5. **Polimorfismo:** La capacità di utilizzare oggetti di classi diverse attraverso la stessa interfaccia. In pratica, un metodo può funzionare con oggetti di vari tipi, e l'implementazione appropriata viene scelta in base al tipo dell'oggetto.

6. **Astrazione:** La capacità di definire classi astratte che rappresentano concetti generici, lasciando le specifiche implementazioni ai dettagli delle classi concrete derivate. Le classi astratte non possono essere istanziate direttamente ma servono come base per altre classi. Possiamo pensare all'astrazione come parte del polimorfismo, cioè quella capacità di assumere forme diverse in base alla necessità.
-

Capitolo 2: Fondamenti di C#

Variabili e Tipi di Dati

Esempio:

```
int numero = 5;  
string testo = "Ciao";
```

Operatori

- **Operatori aritmetici:** +, -, *, /
- **Operatori logici:** &&, ||

Strutture di controllo

Il C# ha le stesse strutture di controllo e la stessa sintassi del C++

1. if

L'istruzione **if** esegue un blocco di codice se una condizione specificata è vera.

Sintassi:

```
if (condizione)  
{    // Codice da eseguire se la condizione è vera}  
else if (altraCondizione)
```

```
{      // Codice da eseguire se l'altra condizione è vera}

else

{      // Codice da eseguire se nessuna delle condizioni è vera}
```

Esempio:

```
int numero = 10;

if (numero > 0)

{      Console.WriteLine("Numero positivo");

}

else if (numero < 0)

{      Console.WriteLine("Numero negativo");

}

else

{      Console.WriteLine("Numero zero");

}
```

2. switch

L'istruzione **switch** seleziona uno tra molti blocchi di codice da eseguire, in base al valore di un'espressione.

Sintassi:

```
switch (espressione)

{

    case valore1:

        // Codice da eseguire se espressione è uguale a valore1

        break;
```

```
case valore2:

    // Codice da eseguire se espressione è uguale a valore2

    break;

    // Altri case...

default:

    // Codice da eseguire se nessun case corrisponde

    break;

}
```

Esempio:

```
int giorno = 3;

switch (giorno)

{
    case 1:

        Console.WriteLine("Lunedì");

        break;

    case 2:

        Console.WriteLine("Martedì");

        break;

    case 3:

        Console.WriteLine("Mercoledì");

        break;

    case 4:

        Console.WriteLine("Giovedì");

        break;
}
```

```

    case 5:
        Console.WriteLine("Venerdì");
        break;

    case 6:
        Console.WriteLine("Sabato");
        break;

    case 7:
        Console.WriteLine("Domenica");
        break;

    default:
        Console.WriteLine("Numero del giorno non valido");
        break;
    }
}

```

3. for

L'istruzione **for** viene utilizzata per eseguire un blocco di codice un numero specificato di volte.

Sintassi:

```

for (inizializzazione; condizione; incremento)
{
    // Codice da eseguire in ogni iterazione
}

```

Esempio:

```

for (int i = 0; i < 5; i++)
{

```

```
        Console.WriteLine("Valore di i: " + i);  
    }  
  
}
```

4. while

L'istruzione `while` esegue un blocco di codice finché una condizione specificata è vera.

Sintassi:

```
while (condizione)  
{    // Codice da eseguire finché la condizione è vera  
}  
  
}
```

Esempio:

```
int i = 0;  
  
while (i < 5)  
{  
    Console.WriteLine("Valore di i: " + i);  
    i++;  
}  
  
}
```

5. do-while

L'istruzione `do-while` è simile a `while`, ma verifica la condizione dopo l'esecuzione del blocco di codice, garantendo che il blocco venga eseguito almeno una volta.

Sintassi:

```
do  
{
```

```
// Codice da eseguire almeno una volta e poi ripetuto finché la  
condizione è vera  
} while (condizione);
```

Esempio:

```
int i = 0;  
  
do  
  
{  
    Console.WriteLine("Valore di i: " + i);  
  
    i++;  
  
} while (i < 5);
```

Osservazioni

Questi costrutti di controllo del flusso fondamentali per la programmazione in C# sono identici a quelli del linguaggio C++. Permettono di prendere decisioni, eseguire cicli ripetitivi e scegliere tra più opzioni, rendendo il codice flessibile e potente. Utilizzarli correttamente è **essenziale** per scrivere programmi chiari, efficienti e ben strutturati.

6. STAMPA LA TABELLINA IN C++ E IN C#. GESTIONE INPUT/OUTPUT IN C#

Chiedi all'utente di inserire il numero della tabellina richiesta e stampala

6.1. CODICE IN C++

```
#include <iostream>

using namespace std;

int main()
{
    int tabellina;
    do {
        cout << "tabellina del: ";
        cin >> tabellina;
        if (tabellina > 10 || tabellina < 0) {
            cout << "inserisci un numero tra 0 e 10" << endl;
        }
    } while (tabellina > 10 || tabellina < 0);

    for (int i = 1; i <= 10; i++) {
        int ris = tabellina * i;
        cout << tabellina << " * " << i << " = " << ris << endl;
    }
}
```

6.2. CODICE IN C#

Ecco il codice C++ riscritto in C#:

```
using System;

class Program

{

    static void Main()

    {

        int tabellina;

        do

        {

            Console.WriteLine("Tabellina del: ");

            tabellina = Convert.ToInt32(Console.ReadLine());




            if (tabellina > 10 || tabellina < 0)

            {

                Console.WriteLine("Inserisci un numero tra 0 e 10");

            }

        } while (tabellina > 10 || tabellina < 0);




        for (int i = 1; i <= 10; i++)

        {

            int ris = tabellina * i;

            Console.WriteLine($"{tabellina} * {i} = {ris}");

        }

    }

}
```

6.3. Spiegazione delle modifiche:

Abbiamo utilizzato `Console.WriteLine()` e `Console.WriteLine()` per l'output.

Abbiamo utilizzato `Console.ReadLine()` per leggere l'input (ogni input viene letto come stringa).

Abbiamo usato `Convert.ToInt32` per convertire l'input dell'utente da stringa in un intero.

La sintassi per le stringhe usa l'interpolazione (`$(...)`) per una migliore leggibilità .

6.4. LEGGERE UN SINGOLO CARATTERE O UNA STRINGA IN INPUT

`Console.Read` è un metodo della classe `Console` in C#, utilizzato per leggere il valore di un singolo carattere dalla console.

Ecco alcune caratteristiche e dettagli su come funziona:

Caratteristiche di `Console.Read`

6.4.1. **Tipo di ritorno**:

- Restituisce un intero che rappresenta il valore Unicode del carattere letto. Se non ci sono caratteri da leggere (ad esempio, se si raggiunge la fine del flusso), restituisce -1.

6.4.2. **Uso**:

- E' utile quando desideri leggere un singolo carattere, ad esempio per controllare input da tastiera senza premere "Invio" dopo ogni carattere.

- Tipicamente, viene utilizzato per gestire input di tipo carattere, come ad esempio per il controllo di sequenze di tasti.

6.4.3. **Input da tastiera**:

- Quando chiavi `Console.Read`, il programma attende che l'utente prema un tasto. Il carattere corrispondente viene quindi restituito come valore Unicode.

Esempio di utilizzo:

Ecco un semplice esempio che mostra come utilizzare `Console.Read`:

```
using System;

class Program

{
    static void Main()
    {
        Console.WriteLine("Premi un tasto qualsiasi...");
        int key = Console.Read(); // Legge un singolo carattere
        char character = (char)key; // Converte l'intero in char

        Console.WriteLine($"Hai premuto: {character}");
    }
}
```

7. USO DI FOREACH SU UN ARRAY E SU UNA LISTA

7.1 GESTIONE DEGLI ARRAY

Parliamo da un esempio:

Creare un array di 5 elementi in C# e popolarlo con dei valori.

```
### Creazione e popolazione di un array
```

```
#### Esempio di codice
```

```
using System;

class Program

{

    static void Main()

    {

        // Creiamo un array di interi con 5 elementi

        int[] numeri = new int[5];



        // Popoliamo l'array utilizzando un ciclo for

        for (int i = 0; i < numeri.Length; i++)

        {

            numeri[i] = 222; // Assegniamo a ciascun elemento il

            valore 222

        }





        // Stampa i valori dell'array

        Console.WriteLine("I numeri nell'array sono:");

        for (int i = 0; i < numeri.Length; i++)

        {
```

```
        Console.WriteLine(numeri[i]);  
    }  
}  
}
```

Spiegazione

1. **Creazione dell'array**:

```
int[] numeri = new int[5];
```

Qui stiamo creando un array di interi chiamato `numeri` con una dimensione di 5. Gli elementi dell'array sono inizializzati a 0 per default.

2. **Popolamento dell'array**:

Utilizziamo un ciclo `for` per riempire l'array con valori.

```
for (int i = 0; i < numeri.Length; i++)  
{  
    numeri[i] = 222; // Assegniamo a ciascun elemento il valore  
222  
}
```

In questo ciclo, `i` parte da 0 e va fino a 4 (incluso), poiché `numeri.Length` restituisce la lunghezza dell'array. Assegniamo a ciascun elemento il valore `222`, quindi l'array conterrà i numeri 222, 222, 222, 222, 222.

3. **Stampa dei valori**:

Utilizziamo un altro ciclo `for` per stampare i valori contenuti nell'array.

```
Console.WriteLine(numeri[i]);
```

Questo ci permette di vedere i valori che abbiamo appena assegnato all'array.

Riepilogo

In questo esempio abbiamo:

- Creato un array di 5 elementi.
- Popolato l'array .
- Stampato i valori dell'array.

7.2 USO DI FOREACH SU UN ARRAY

Ecco un esempio di utilizzo di `foreach` con un array in C#.

Esempio di codice

```
using System;

class Program

{
    static void Main()
    {
        // Creiamo un array di interi
        int[] numeri = new int[] { 1, 2, 3, 4, 5 };

        // Utilizziamo foreach per iterare attraverso l'array
        foreach (int numero in numeri)
```

```
{  
    Console.WriteLine($"Il numero è: {numero}");  
}  
}  
}
```

Spiegazione

1. **Definizione dell'array**:

Abbiamo creato un array di interi chiamato `numeri`. Utilizziamo la sintassi `new int[]` per inizializzare l'array con alcuni valori (1, 2, 3, 4, 5).

2. **Utilizzo di `foreach`**:

Il ciclo `foreach` permette di iterare attraverso ogni elemento dell'array. In questo caso, `numero` rappresenta ciascun elemento dell'array `numeri` durante ogni iterazione.

3. **Stampa dei valori**:

All'interno del ciclo, utilizziamo `Console.WriteLine` per stampare un messaggio che mostra il valore corrente di `numero`.

Vantaggi di `foreach`

- **Semplicità**: È più semplice e leggibile rispetto a un ciclo `for` tradizionale, in cui dovresti gestire manualmente l'indice.
- **Sicurezza**: Riduce il rischio di errori, come l'accesso a indici non validi, poiché il ciclo gestisce automaticamente la fine dell'array.

Questo esempio dimostra come usare `foreach` con un array in C#.

7.3 USO DI FOREACH SU UNA LISTA

```
### Esempio di codice

using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        // Creiamo una lista di stringhe
        List<string> nomi = new List<string>
        {
            "Alice",
            "Bob",
            "Charlie",
            "Diana"
        };

        // Utilizziamo foreach per iterare attraverso la lista
        foreach (string nome in nomi)
        {
            Console.WriteLine($"Ciao, {nome}!");
        }
    }
}
```

```
    }  
}  
}
```

Spiegazione

1. **Definizione della lista**:

Abbiamo creato una lista di stringhe chiamata `nomi` che contiene alcuni nomi. La lista è una collezione generica fornita da `System.Collections.Generic`.

2. **Utilizzo di `foreach`**:

Il ciclo `foreach` permette di iterare attraverso ogni elemento della lista senza la necessità di gestire un indice. In questo caso, `nome` rappresenta ogni singolo elemento della lista `nomi` durante ogni iterazione.

3. **Stampa dei messaggi**:

All'interno del ciclo, utilizziamo `Console.WriteLine` per stampare un messaggio di saluto per ogni nome nella lista.

Questo esempio illustra come `foreach` possa essere utilizzato per semplificare l'iterazione su collezioni in C#.

7.5 RIDIMENSIONARE UN ARRAY

In C# puoi ridimensionare un array, ma non direttamente perché gli array hanno una dimensione fissa. Tuttavia, puoi usare la classe **Array.Resize** per creare un nuovo array con una dimensione diversa e copiare i dati esistenti.

Ecco un esempio:

```
int[] array = { 1, 2, 3, 4 };

Array.Resize(ref array, 6); // Ridimensiona l'array a una lunghezza
di 6

// Ora array ha due elementi aggiuntivi con valore di default (0)
foreach (int i in array)

{
    Console.WriteLine(i);
}
```

In questo caso, l'array originale viene ridimensionato a 6 elementi, e gli elementi aggiuntivi vengono inizializzati ai valori predefiniti (0 per i tipi numerici).

8. USO DI TRY CATCH PER LA GESTIONE DEGLI ERRORI

Il blocco `try-catch` in C# è utilizzato per gestire le eccezioni, permettendo di catturare errori durante l'esecuzione del codice senza interrompere l'intero programma.

```
### Esempio di codice

using System;

class Program

{

    static void Main()

    {

        try

        {

            // Chiediamo all'utente di inserire un numero

            Console.WriteLine("Inserisci un numero:");

            string input = Console.ReadLine();

            // Proviamo a convertire l'input in un intero

            int numero = Convert.ToInt32(input);

            // Stampiamo il risultato

            Console.WriteLine($"Hai inserito il numero: {numero}");

        }

        catch (FormatException)

        {

            // Questo blocco verrà eseguito se l'input non è un

            numero valido
```

```

        Console.WriteLine("Errore: Inserisci un numero
valido.");
    }

    catch (OverflowException)

    {

        // Questo blocco verrà eseguito se il numero è troppo
        grande o troppo piccolo

        Console.WriteLine("Errore: Il numero inserito è troppo
        grande o troppo piccolo.");

    }

    catch (Exception ex)

    {

        // Cattura qualsiasi altra eccezione non specificata

        Console.WriteLine($"Errore imprevisto: {ex.Message}");

    }

    finally

    {

        // Questo blocco verrà eseguito sempre,
        indipendentemente dal risultato

        Console.WriteLine("Operazione completata.");

    }

}

```

Spiegazione

1. **Blocco `try`**:
try

```

{
    // Codice che può generare eccezioni
}

```

Qui inseriamo il codice che potrebbe generare un'eccezione. In questo esempio, chiediamo all'utente di inserire un numero e tentiamo di convertirlo in un intero.

2. **Blocco `catch`**:

- **`FormatException`**: Questo blocco gestisce l'eccezione che si verifica se l'input non è un numero valido.
- **`OverflowException`**: Questo blocco gestisce l'eccezione che si verifica se il numero inserito è troppo grande o troppo piccolo per essere rappresentato come un intero.
- **`Exception ex`**: Questo blocco cattura qualsiasi altra eccezione non specificata e mostra un messaggio generico.

3. **Blocco `finally`**:

```

finally
{
    // Codice che verrà eseguito sempre
}

```

Questo blocco è opzionale e viene eseguito sempre, sia che si verifichino eccezioni sia che non ce ne siano. È utile per eseguire operazioni di pulizia, come la chiusura di file o la liberazione di risorse.

Riepilogo

In questo esempio abbiamo:

- Utilizzato `try` per tentare di eseguire del codice che può generare eccezioni.
- Gestito specifiche eccezioni con blocchi `catch`.
- Usato un blocco `finally` per eseguire codice finale, indipendentemente dal successo o dal fallimento.

Capitolo 3: Definizione di Classi

Le classi

Una **classe** è come un modello o un progetto che definisce le caratteristiche e i comportamenti di un oggetto. Immagina di voler costruire delle case: una classe sarebbe il progetto architettonico che descrive come devono essere costruite tutte le case (quante stanze hanno, di che materiale sono fatte, ecc.).

Attributi, Costruttori e Metodi

Esempio:

```
public class Auto
{
    public string Modello { get; set; }
    public Auto(string modello)
    {
        Modello = modello;
    }
}
```

Vediamo come il codice fornito può aiutarci a introdurre i concetti di ATTRIBUTO (o proprietà), COSTRUTTORE e METODO

1. **Cos'è un Attributo?**

Un **attributo** è una variabile che è dichiarata all'interno di una classe e rappresenta una caratteristica o proprietà degli oggetti di quella classe. In altre parole, gli attributi sono i dati che descrivono lo stato di un oggetto.

Nell' esempio:

```
public string Modello { get; set; }
```

- **Modello** è un attributo della classe `Auto`. È di tipo `string`, quindi può contenere testo.
- Gli attributi in una classe descrivono le caratteristiche che ogni oggetto della classe avrà. In questo caso, ogni `Auto` avrà un `Modello`.

Prova a fare tu: verifica cosa succede se scrivi **public string Modello=""**; invece di **public string Modello { get; set; }**

2. **Cos'è un Costruttore?**

Un **costruttore** è una funzione speciale in una classe che viene chiamata quando viene creato un nuovo oggetto di quella classe. Il costruttore ha lo stesso nome della classe e viene utilizzato per inizializzare gli attributi di un oggetto, assegnando loro dei valori iniziali.

Prova a fare tu: ripassa i concetti di funzione e procedura visti in C++ l'anno scorso. Tutto quello che abbiamo visto nella teoria della programmazione di C++, continua a valere per C#

Nell' esempio:

```
public Auto(string modello)
```

```
{
```

```
    Modello = modello;
```

```
}
```

- **Auto** è il costruttore della classe `Auto`.

- Questo costruttore accetta un parametro chiamato `modello` di tipo `string`.
- All'interno del costruttore, l'attributo `Modello` dell'oggetto viene inizializzato con il valore passato al costruttore.

Quando crei un nuovo oggetto `Auto` nel Main(), ad esempio:

```
Auto miaAuto = new Auto("Fiat 500");
```

- **new Auto("Fiat 500")**: Questo crea un nuovo oggetto `Auto` e chiama il costruttore `Auto`, passando "Fiat 500" come parametro.

- All'interno del costruttore, l'attributo `Modello` dell'oggetto `miaAuto` viene impostato su "Fiat 500".

3. **Cos'è un Metodo?**

Un **metodo** è una **funzione** definita all'interno di una classe. I metodi sono utilizzati per definire i comportamenti di un oggetto, ovvero ciò che l'oggetto può fare. I metodi possono accedere e modificare gli attributi di un oggetto, oltre a poter eseguire altre operazioni.

Nell'esempio, non c'è un metodo diverso dal costruttore, ma vediamo comunque un esempio per chiarire il concetto:

```
public void StampaModello()
{
    //Console.WriteLine($"Modello dell'auto: {Modello}");
    Console.WriteLine("Modello dell'auto " + Modello);
}
```

- `public void StampaModello()`: Questo è un metodo della classe `Auto`. Non restituisce alcun valore (`void`) e può essere chiamato per eseguire un'azione.
- Quando chiamiamo `miaAuto.StampaModello();`, il metodo stampa il modello dell'auto su cui è stato chiamato.

4. **Esempio Completo**

Mettiamo insieme tutto questo in un esempio completo:

```
public class Auto
{
    // Attributo

    public string Modello { get; set; }

    // Costruttore

    public Auto(string modello)
    {
```

```

    Modello = modello;

}

// Metodo

public void StampaModello()

{
    Console.WriteLine($"Modello dell'auto: {Modello}");

}

```

NEL MAIN:

Come abbiamo visto con C++, anche con la programmazione in C# usando Visual Studio possiamo creare dei programmi di tipo console che hanno come punto di avvio la procedura Main().

```

static void Main(string[] args)

{
    //INSERISCI QUI IL CODICE DI AVVIO DEL PROGRAMMA
}

```

Dunque nel Main possiamo inserire il seguente codice che verrà utilizzato all'avvio del programma:

```

// Creazione di un nuovo oggetto Auto
Auto miaAuto = new Auto("Fiat 500");

// Uso del metodo per stampare il modello
miaAuto.StampaModello(); // Output: Modello dell'auto: Fiat 500

```

5. **Riassumendo**

- **Attributo**: È una variabile all'interno di una classe che rappresenta una proprietà o uno stato di un oggetto. Nell'esempio, `Modello` è un attributo della classe `Auto`.

- **Costruttore**: È un metodo speciale che ha lo stesso nome della classe ed è chiamato automaticamente quando si crea un nuovo oggetto di quella classe. Inizializza gli attributi dell'oggetto. Nell'esempio, `Auto(string modello)` è il costruttore della classe `Auto`.

- **Metodo**: È una funzione definita all'interno di una classe che descrive un comportamento o un'azione che un oggetto della classe può eseguire. Nell'esempio, `StampaModello()` è un metodo della classe `Auto`.

Questi concetti sono fondamentali per comprendere come creare e utilizzare oggetti in programmazione orientata agli oggetti.

Esempio di un Metodo:

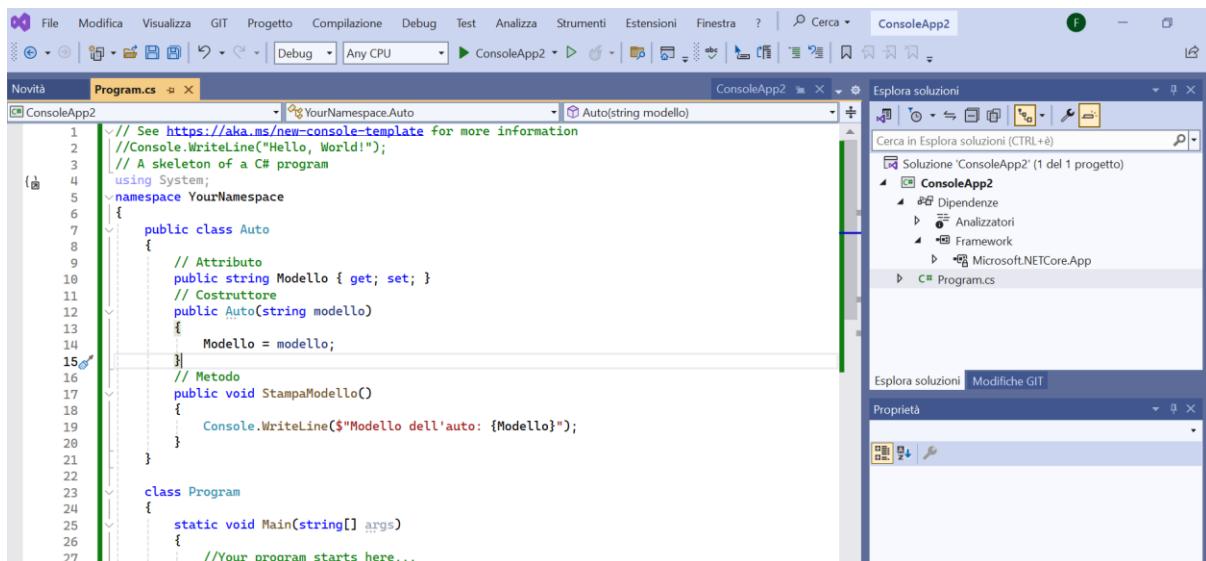
```
public void Accendi()
{
    Console.WriteLine("L'auto è accesa");
}
```

Esempio di un Attributo(Proprietà):

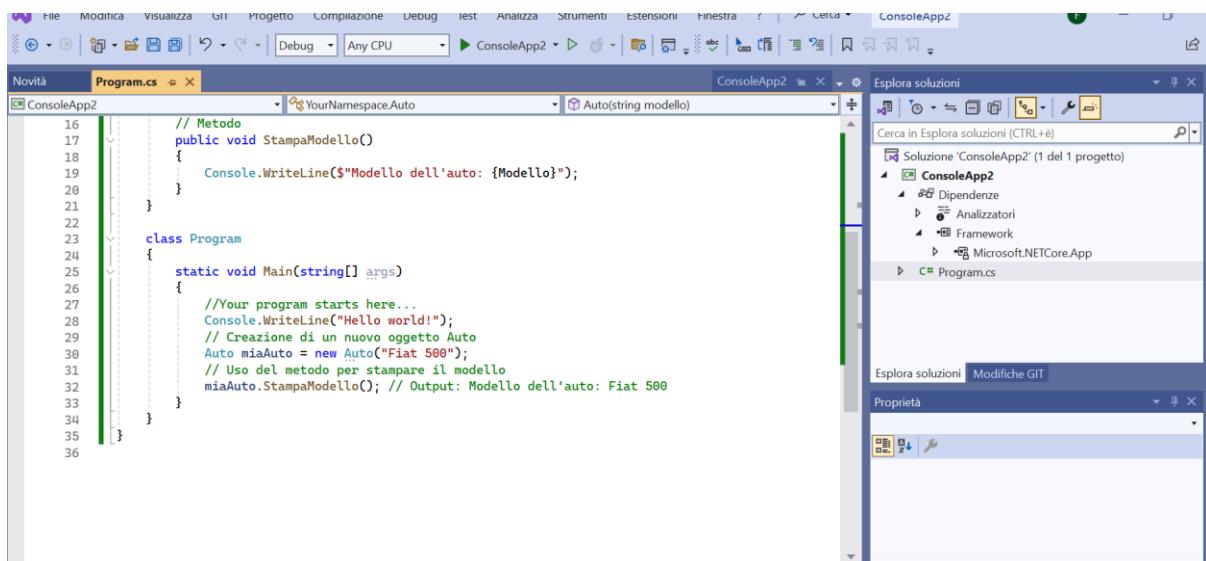
```
public int Velocità { get; set; }
```

Prova a fare tu:

1. crea un programma C# di tipo console in Visual Studio.
2. Crea la classe Auto
3. Inserisci nel Main() il richiamo alla classe creando l'oggetto miaAuto



```
// See https://aka.ms/new-console-template for more information
//Console.WriteLine("Hello, World!");
// A skeleton of a C# program
using System;
namespace YourNamespace
{
    public class Auto
    {
        // Attributo
        public string Modello { get; set; }
        // Costruttore
        public Auto(string modello)
        {
            Modello = modello;
        }
        // Metodo
        public void StampaModello()
        {
            Console.WriteLine($"Modello dell'auto: {Modello}");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            //Your program starts here...
        }
    }
}
```



```
// Metodo
public void StampaModello()
{
    Console.WriteLine($"Modello dell'auto: {Modello}");
}

class Program
{
    static void Main(string[] args)
    {
        //Your program starts here...
        Console.WriteLine("Hello world!");
        // Creazione di un nuovo oggetto Auto
        Auto miaAuto = new Auto("Fiat 500");
        // Uso del metodo per stampare il modello
        miaAuto.StampaModello(); // Output: Modello dell'auto: Fiat 500
    }
}
```

Capitolo 4: Concetti Base della Programmazione ad Oggetti

Classi e Oggetti Una classe è una descrizione astratta di un'entità, mentre un oggetto è un'istanza di una classe. Vediamo ora un'implementazione concreta.

Esempio:

CREO LA CLASSE Persona:

```
public class Persona
{
    public string Nome { get; set; }
    public int Età { get; set; }
}
```

USO LA CLASSE, PER FARLO CREO L'OGGETTO p:

```
Persona p = new Persona();
p.Nome = "Mario";
p.Età = 30;
```

Attraverso questo codice introduciamo i concetti di classe e oggetto.

1. **Cos'è una classe?**

Nel nostro esempio, la classe è chiamata `Persona`. La classe `Persona` definisce due caratteristiche (chiamate anche **proprietà**):

- **Nome**: una proprietà di tipo `string` che rappresenta il nome della persona.
- **Età**: una proprietà di tipo `int` che rappresenta l'età della persona.
- **public**: significa che la classe e le sue proprietà possono essere usate da altre parti del programma.
- **string Nome { get; set; }**: questa linea crea una proprietà `Nome` di tipo `string` (che è usato per rappresentare testo). Le parole `get` e `set` sono modi per ottenere e impostare il valore della proprietà.

- `int Età { get; set; }`: questa linea crea una proprietà `Età` di tipo `int` (un numero intero). Anche qui, `get` e `set` permettono di leggere e modificare il valore dell'età.

N.B. Analizzeremo in dettaglio get e set più avanti affrontando il concetto di encapsulamento. Per adesso non ti preoccupare se la sintassi risulta poco chiara, ma usala in modo intuitivo pensando che serve ad impostare "valori" particolari.

2. **Cos'è un oggetto?**

Un **oggetto** è una "istanza" di una classe, ovvero una realizzazione concreta del modello definito dalla classe. Se la classe è il progetto della casa, l'oggetto è la casa reale costruita seguendo quel progetto.

Nell'esempio, creiamo un oggetto `p` di tipo `Persona`. Questo oggetto rappresenta una persona concreta, con un nome e un'età specifici.

Ecco la parte del codice che crea un oggetto e lo utilizza nel Main():

```
Persona p = new Persona();
```

```
p.Nome = "Mario";
```

```
p.Età = 30;
```

- `Persona p = new Persona();`: Questa linea crea un nuovo oggetto di tipo `Persona`. `new` viene usato per creare una nuova istanza (cioè un nuovo oggetto) della classe `Persona`.

- `p.Nome = "Mario";`: Questa linea assegna il valore `"Mario"` alla proprietà `Nome` dell'oggetto `p`. Quindi, ora `p` ha il nome "Mario".

- `p.Età = 30;`: Questa linea assegna il valore `30` alla proprietà `Età` dell'oggetto `p`. Ora, l'età della persona rappresentata da `p` è 30 anni.

Riassumendo

- **Classe** (`Persona`): è come un modello o una definizione di come deve essere una "persona" nel nostro programma.

- **Oggetto** (`p`): è un'istanza concreta della classe, una "persona" specifica con un nome e un'età assegnati.

Questo codice ci mostra come definire una classe per rappresentare concetti astratti come le persone e come creare oggetti concreti da queste classi per rappresentare individui reali nel programma Main().

Ereditarietà Permette a una classe di ereditare membri da un'altra classe.

Esempio:

CREO LA CLASSE Studente CHE EREDITA DA Persona :

```
public class Studente : Persona
{
    public string Matricola { get; set; }
}
```

USO LA CLASSE, PER FARLO CREO L'OGGETTO stu:

```
Studente stu = new Studente ();
stu.Nome = "Mario";
stu.Età = 10;
stu.Matricola = "s21345";
```

Parliamo del concetto di ****ereditarietà**** usando il codice precedente.

1. ****Cos'è l'ereditarietà?****

****L'ereditarietà**** è un concetto fondamentale nella programmazione orientata agli oggetti (OOP). Consente di creare una nuova classe che "eredita" le proprietà e i metodi di un'altra classe esistente. In altre parole, è un modo per riutilizzare e estendere le funzionalità di una classe già definita.

Immagina l'ereditarietà come una forma di specializzazione. Per esempio, se abbiamo una classe `Persona` che rappresenta le persone in generale, possiamo creare una classe `Studente` che rappresenta una persona che è anche uno studente. Lo studente è una persona, ma ha anche alcune caratteristiche specifiche che le persone normali potrebbero non avere, come una matricola.

2. ****Esempio nel codice****

Nel tuo esempio, abbiamo due classi: `Persona` e `Studente` .

Classe `Persona`

Questa è la classe base o "genitore". Definisce le proprietà comuni a tutte le persone:

- ****Nome**** e ****Età**** sono proprietà che ogni persona avrà.

Classe `Studente`

Questa è la classe derivata o "figlio". `Studente` **eredita** dalla classe `Persona`. Questo significa che la classe `Studente` avrà automaticamente tutte le proprietà della classe `Persona` (cioè `Nome` e `Età`), più qualsiasi altra proprietà o metodo che definiamo in `Studente`.

```
`public class Studente : Persona
{
    public string Matricola { get; set; }
}
```

- **`public class Studente : Persona`**: Il simbolo `:` indica che `Studente` eredita dalla classe `Persona`. Quindi, `Studente` avrà tutto ciò che ha `Persona`.
- **`public string Matricola { get; set; }`**: Questa è una nuova proprietà specifica della classe `Studente`. Ogni oggetto di tipo `Studente` avrà una `Matricola` oltre alle proprietà `Nome` e `Età` che ha ereditato da `Persona`.

3. **Creazione e utilizzo dell'oggetto `Studente`**

Nel Main() creiamo un oggetto di tipo `Studente` e assegniamo valori alle sue proprietà:

- **`Studente stu = new Studente();`**: Questa linea crea un nuovo oggetto `stu` di tipo `Studente`.
- **`stu.Nome = "Mario";`**: Assegna il valore `"Mario"` alla proprietà `Nome` dell'oggetto `stu`. Anche se `Nome` è definito nella classe `Persona`, l'oggetto `stu` di tipo `Studente` può utilizzarlo perché `Studente` eredita da `Persona`.
- **`stu.Età = 10;`**: Assegna il valore `10` alla proprietà `Età` dell'oggetto `stu`.
- **`stu.Matricola = "s21345";`**: Assegna il valore `"s21345"` alla proprietà `Matricola` dell'oggetto `stu`. Questa proprietà è specifica della classe `Studente` e non esiste in `Persona`.

4. **Riassumendo**

- **Ereditarietà**: È il concetto di riutilizzare e estendere una classe esistente. La classe che eredita è chiamata "classe derivata" o "figlio" (in questo caso, `Studente`), e la classe da cui eredita è chiamata "classe base" o "genitore" (in questo caso, `Persona`).
- **Classe base (`Persona`)**: Definisce proprietà comuni come `Nome` e `Età`.
- **Classe derivata (`Studente`)**: Eredita tutte le proprietà di `Persona` e aggiunge altre proprietà specifiche come `Matricola`.

Questo esempio dimostra come l'ereditarietà può semplificare il codice e rendere più facile la gestione di oggetti che condividono proprietà e comportamenti comuni.

Polimorfismo Consente di trattare oggetti di classi derivate come oggetti della classe base.

Vedremo inoltre più avanti come il polimorfismo si applica anche ai **metodi di una classe**.

Esempio:

```
Persona studente = new Studente();
```

Vediamo come introdurre il concetto di **polimorfismo**

1. **Cos'è il Polimorfismo?**

Polimorfismo è un concetto fondamentale in programmazione orientata agli oggetti (OOP) che significa "molte forme". Permette agli oggetti di essere trattati come istanze della loro classe base, anche quando sono in realtà istanze di una classe derivata. Questo consente di scrivere codice più flessibile e generalizzato, che può funzionare con oggetti di diversi tipi **senza conoscere esattamente il tipo di oggetto a tempo di compilazione**.

In altre parole, il polimorfismo permette di utilizzare una classe base per riferirsi a qualsiasi oggetto di una classe derivata. Questo è utile perché possiamo scrivere codice che funziona con il tipo della classe base, e poi passare oggetti di classi derivate senza cambiare il codice.

In altre parole ancora, facendo un esempio, se io creo la classe base animale e poi creo classi derivate come elefante, giraffa, cane etc... posso utilizzare come oggetto generico un'istanza di animale per ospitare (**INCAPSULARE ALL'INTERNO**) un'istanza di elefante o di giraffa o di cane etc..

2. **Esempio di Polimorfismo nel Codice**

Consideriamo il codice **Persona studente = new Studente();** da richiamare nel Main() dopo aver creato le classi Persona (classe base) e Studente (classe derivata)

Vediamo cosa succede:

- **``Persona``**: è la classe base, da cui la classe `Studente` eredita.
- **``studente``**: è una variabile di tipo `Persona` (**istanza della classe o oggetto**).
- **``new Studente()``**: crea un nuovo oggetto di tipo `Studente`.

Con questo codice, stiamo creando un oggetto di tipo `Studente` e lo stiamo assegnando a una variabile di tipo `Persona`. Questo è possibile grazie all'ereditarietà: poiché `Studente` eredita da `Persona`, ogni oggetto di tipo `Studente` è anche un tipo di `Persona`.

3. **Perché è Utile il Polimorfismo?**

Il polimorfismo è utile perché permette di scrivere funzioni o metodi che funzionano con il tipo della classe base (`Persona` in questo caso) ma possono essere utilizzati con oggetti di qualsiasi classe derivata (`Studente` o altre classi che potrebbero derivare da `Persona` in futuro).

Ad esempio, potremmo avere un metodo che accetta una `Persona` come argomento:

```
public void Saluta(Persona p)  
{  
    // Console.WriteLine($"Ciao, {p.Nome}!");  
    Console.WriteLine("Ciao " + p.Nome);  
}
```

Grazie al polimorfismo, possiamo chiamare questo metodo con un oggetto di tipo `Studente`:

```
Studente stu = new Studente();  
stu.Nome = "Mario";  
Saluta(stu); // Funziona perché 'stu' è anche una 'Persona'
```

Il metodo `Saluta` non deve sapere se `p` è una `Persona` o un `Studente` o qualsiasi altra classe che eredita da `Persona`. Questo rende il codice più flessibile e riutilizzabile.

4. **Polimorfismo dei Metodi**

Il **polimorfismo dei metodi** (chiamato anche "polimorfismo di runtime" o "polimorfismo dinamico") è un altro aspetto importante del polimorfismo. Permette a una classe di sovrascrivere un metodo della classe stessa per fornire una specifica implementazione del metodo. Lo vedremo successivamente

5. **Riassumendo**

- **Polimorfismo**: Permette di trattare oggetti di una classe derivata come oggetti della loro classe base.
- **Polimorfismo dei Metodi**: Permette alle classi di sovrascrivere metodi per fornire una specifica implementazione.

Questo concetto è fondamentale per creare codice flessibile, riutilizzabile e facile da estendere, poiché consente di scrivere metodi e funzioni che funzionano con una classe base ma possono essere utilizzati con qualsiasi oggetto derivato da quella classe.

Incapsulamento Nasconde i dettagli dell'implementazione interna e mostra solo ciò che è necessario.

Esempio:

```
private string nome;
public string Nome
{
    get { return nome; }
    set { nome = value; }
}
```

Vediamo come il codice fornito può aiutarci a introdurre il concetto di **incapsulamento** e i metodi **get** e **set** in programmazione orientata agli oggetti.

1. **Cos'è l'Incapsulamento?**

L'incapsulamento è uno dei tre pilastri della programmazione orientata agli oggetti, insieme a ereditarietà e polimorfismo (che comprende l'astrazione). L'incapsulamento consiste nel nascondere i dettagli interni di un oggetto e nel mostrare solo ciò che è necessario. Questo aiuta a proteggere i dati e a prevenire modifiche non autorizzate o errate.

In pratica, l'incapsulamento permette di:

- **Rendere i dati privati** e inaccessibili direttamente dall'esterno della classe.
- **Fornire metodi pubblici** (come `get` e `set`) per leggere e modificare quei dati in modo controllato.

2. **Esempio nel Codice**

Nell' esempio, stiamo utilizzando l'incapsulamento per gestire l'accesso a una proprietà chiamata `nome`. Vediamo il codice passo per passo:

```
private string nome;
```

- **private**: Questa parola chiave rende la variabile `nome` accessibile solo all'interno della classe in cui è definita. Nessun'altra classe può accedere direttamente a `nome`. Questo è l'incapsulamento: proteggiamo `nome` dall'accesso esterno.

Poi, abbiamo:

```
public string Nome
```

```
{
```

```
    get { return nome; }
```

```
    set { nome = value; }
```

```
}
```

- **public**: Questa parola chiave indica che la proprietà `Nome` è accessibile da fuori della classe. È il modo in cui esponiamo `nome` all'esterno.

- **string Nome**: Questa è una proprietà chiamata `Nome` di tipo `string`. A differenza della variabile `nome`, la proprietà `Nome` è pubblica e può essere accessibile dall'esterno.

3. **Metodi `get` e `set`**

I metodi **get** e **set** sono chiamati **accessors** e sono usati per controllare l'accesso alle variabili private. Ecco cosa fanno:

Metodo `get`

```
get { return nome; }
```

- **get**: Questo metodo viene chiamato quando si vuole **leggere** il valore della proprietà `Nome`.

- **return nome;**: Restituisce il valore della variabile privata `nome`.

Quando scriviamo `string valore = oggetto.Nome;`, stiamo usando il metodo `get` per ottenere il valore di `nome`.

Metodo `set`

```
set { nome = value; }
```

- **`set`**: Questo metodo viene chiamato quando si vuole **impostare** un nuovo valore alla proprietà `Nome`.

- **`nome = value;`**: Assegna il valore passato alla variabile privata `nome`.

Quando scriviamo `oggetto.Nome = "Mario";`, stiamo usando il metodo `set` per impostare il valore di `nome` a `"Mario"`.

4. **Perché Usare `get` e `set`?**

Usare i metodi `get` e `set` offre diversi vantaggi:

- **Controllo degli accessi**: Possiamo limitare chi può leggere o modificare i dati. Ad esempio, possiamo avere un `get` pubblico ma un `set` privato se vogliamo che i dati possano essere letti da fuori ma non modificati.

- **Validazione dei dati**: Possiamo aggiungere logica all'interno di `get` e `set` per verificare o trasformare i dati. Ad esempio, possiamo verificare che un nome non sia vuoto prima di assegnarlo.

Ecco un esempio di validazione:

```
private string nome;  
  
public string Nome  
  
{  
    get { return nome; }  
  
    set {  
        if (!string.IsNullOrEmpty(value))  
        {  
            nome = value;  
        }  
    }  
}
```

Qui, stiamo aggiungendo una verifica che impedisce di assegnare a `nome` un valore vuoto o `null`.

5. **Riassumendo**

- **Incapsulamento**: Nasconde i dettagli interni di una classe, esponendo solo ciò che è necessario. Protegge i dati da accessi non autorizzati.
- **Variabile privata (`private`)**: È accessibile solo all'interno della classe.
- **Proprietà pubblica (`public`)**: Fornisce un modo controllato per accedere e modificare la variabile privata.
- **Metodi `get` e `set`**: Sono utilizzati per leggere e impostare il valore della proprietà in modo controllato. Possono includere logica per la validazione o trasformazione dei dati.

Questo concetto di encapsulamento aiuta a creare codice più sicuro, gestibile e meno soggetto a errori.

Capitolo 5: Incapsulamento

Modificatori di accesso

Esempio:

Partiamo da questo esempio per analizzare il concetto di incapsulamento di una classe, che è il fondamento della programmazione ad oggetti.

```
public class Documento
{
    private string testo;
    public string LeggiTesto() { return testo; }
    public void ScriviTesto(string nuovoTesto)
        { testo = nuovoTesto; }
}
```

•

Proprietà automatiche

Esempio:

```
public string Titolo { get; set; }
```

1. **Concetto di Incapsulamento**

L'incapsulamento significa **nascondere i dettagli interni** di un oggetto e **fornire un'interfaccia pubblica** per interagire con quell'oggetto. L'idea è di proteggere i dati all'interno dell'oggetto da modifiche o accessi non controllati.

Nell'esempio, l'incapsulamento è utilizzato per proteggere l'attributo `testo` della classe `Documento`.

2. **Analisi del Codice**

Attributo Privato

```
private string testo;
```

- **private**: Questa parola chiave indica che l'attributo `testo` è **privato**, cioè può essere accessibile **solo** all'interno della classe `Documento`. Nessun'altra classe o codice esterno può accedere direttamente a `testo`. Questa è una forma di **incapsulamento** perché stiamo nascondendo i dettagli interni dell'oggetto.

Metodi Pubblici

```
public string LeggiTesto() { return testo; }

public void ScriviTesto(string nuovoTesto) { testo = nuovoTesto; }
```

- **LeggiTesto()**: È un metodo **pubblico** che restituisce il valore di `testo`. Questo metodo permette di leggere il valore di `testo` senza accedere direttamente all'attributo privato.

- **ScriviTesto(string nuovoTesto)**: È un altro metodo **pubblico** che permette di modificare il valore di `testo`. Anche questo metodo fornisce un modo controllato per modificare il valore di `testo` senza accedere direttamente all'attributo privato.

3. I metodi get e set*

I metodi `LeggiTesto()` e `ScriviTesto()` possono essere sostituiti dai **metodi `get` e `set`** in termini di funzionalità. I metodi `get` e `set` sono spesso usati per **incapsulare** gli attributi di una classe in modo più semplice e idiomatico. Vediamo come:

Supponiamo di voler riscrivere la classe `Documento` utilizzando le proprietà con `get` e `set` invece dei metodi:

```
public class Documento

{
    private string testo;

    public string Testo
    {
        get { return testo; }
        set { testo = value; }
    }
}
```

Differenze tra i due approcci:

1. **Uso di `get` e `set`**:

- **`get`**: È simile al metodo `LeggiTesto()`. È utilizzato per **leggere** il valore dell'attributo `testo`. Quando scriviamo `string testo = documento.Testo;`, stiamo usando il `get` per ottenere il valore.

- **`set`**: È simile al metodo `ScriviTesto(string nuovoTesto)`. È utilizzato per **impostare** il valore dell'attributo `testo`. Quando scriviamo `documento.Testo = "Nuovo contenuto";`, stiamo usando il `set` per assegnare un nuovo valore.

2. **Sintassi più compatta**:

- Utilizzando `get` e `set`, il codice è più **compatto** e **leggibile**. Non c'è bisogno di creare metodi separati per leggere e scrivere il valore di un attributo.

3. **Controllo dei dati**:

- Proprio come i metodi `LeggiTesto()` e `ScriviTesto()`, anche `get` e `set` possono includere logica aggiuntiva. Ad esempio, possiamo aggiungere una condizione per impedire che `testo` venga impostato su un valore vuoto:

```
public string Testo  
{  
    get { return testo; }  
  
    set  
    {  
        if (!string.IsNullOrEmpty(value))  
        {  
            testo = value;  
        }  
    }  
}
```

4. **Riassumendo**

- **Incapsulamento**: È il processo di nascondere i dettagli interni di un oggetto e fornire un'interfaccia pubblica per interagire con esso. Protegge gli attributi di una classe da accessi o modifiche non controllati.
- **'private' Attributi**: Sono nascosti e accessibili solo all'interno della classe. Nel tuo esempio, `testo` è privato.
- **Metodi `get` e `set`**: Sono metodi speciali usati per leggere (`get`) e modificare (`set`) gli attributi privati in modo controllato. Sono una forma di incapsulamento perché permettono di proteggere gli attributi mentre li rendono accessibili in modo controllato.
- **Vantaggio dell'uso di `get` e `set`**: Forniscono un modo semplice e leggibile per accedere e modificare gli attributi, permettendo anche di aggiungere logica per la validazione o altre operazioni.

L'uso di `get` e `set` insieme agli attributi privati è una pratica comune in OOP per mantenere il codice sicuro, modulare e facilmente manutenibile.

Capitolo 6: Ereditarietà e Polimorfismo

Classi base e derivate

Esempio:

```
public class Animale
{
    public void Muovi() { }
}

public class Cane : Animale
{
    public void Abbaia() { }
}
```

Override e Overload dei metodi

Esempio di override di un metodo

L'override si basa sul concetto di ereditarietà.

```
public class Animale
{
    public virtual void Muovi() { Console.WriteLine("L'animale si
muove"); }
}

public class Cane : Animale
{
    public override void Muovi() { Console.WriteLine("Il cane
corre"); }
}
```

Esempio di overload di un metodo

L'overload si basa sul concetto di polimorfismo

```
public class Animale
{
    public void Comunica() { Console.WriteLine("L'animale
comunica"); }
}
```

```

        public void Comunica(string animale) {
Console.WriteLine("L'animale comunica con il seguente animale: " +
animale); }

        public void Comunica(string animale, int numeroDiVolte) {
Console.WriteLine("L'animale comunica con il seguente animale: " +
animale + "per " + numeroDiVolte + "volte"); }

        public void Comunica(int numeroDiVolte) {
for(int i=0; i<numeroDiVolte; i++)
    Console.WriteLine("L'animale comunica");
}

```

Utilizzo di interfacce

Si basa sul concetto di astrazione

Esempio:

```

public interface IMovibile
{
    void Muovi();
}

public class Auto : IMovibile
{
    public void Muovi() { Console.WriteLine("L'auto si muove"); }
}

```

6.1 LA CLASSE CONVERT

In C#, la classe `Convert` è una classe statica che fornisce metodi per convertire i tipi di dati di base in altri tipi. Questa classe è utile per eseguire conversioni tra tipi di dati primitivi come `int`, `double`, `string`, `bool`, ecc. La classe `Convert` gestisce anche le eccezioni che possono verificarsi durante la conversione, come ad esempio tentare di convertire una stringa che non può essere interpretata come un numero.

Esempio di utilizzo della classe `Convert`:

```
using System;

class Program
{
    static void Main()
    {
        // Dichiarazione di variabili di diversi tipi

        string stringNumber = "123";
        string stringDecimal = "45.67";
        string stringBool = "true";

        // Conversione di una stringa in un intero

        int intNumber = Convert.ToInt32(stringNumber);
        Console.WriteLine("String to Int: " + intNumber);

        // Conversione di una stringa in un double

        double doubleNumber = Convert.ToDouble(stringDecimal);
        Console.WriteLine("String to Double: " + doubleNumber);

        // Conversione di una stringa in un booleano

        bool boolValue = Convert.ToBoolean(stringBool);
        Console.WriteLine("String to Boolean: " + boolValue);

        // Conversione di un intero in una stringa

        string convertedString = Convert.ToString(intNumber);
        Console.WriteLine("Int to String: " + convertedString);
```

```
    }  
}  
  
}
```

Spiegazione:

1. `Convert.ToInt32(stringNumber)`: Converte la stringa `"123"` in un intero.
2. `Convert.ToDouble(stringDecimal)`: Converte la stringa `"45.67"` in un numero decimale di tipo `double`.
3. `Convert.ToBoolean(stringBool)`: Converte la stringa `"true"` in un valore booleano.
4. `Convert.ToString(intNumber)`: Converte l'intero `123` di nuovo in una stringa.

Uscita:

...

String to Int: 123

String to Double: 45.67

String to Boolean: True

Int to String: 123

...

La classe `Convert` è comoda perché semplifica le conversioni di tipi di dati in modo sicuro e gestisce automaticamente gli errori, ad esempio lanciando un'eccezione `FormatException` se la conversione fallisce (come nel caso di una stringa non numerica che non può essere convertita in un numero).

6.2 LA CLASSE STRINGBUILDER

La classe `StringBuilder` in C# è una classe utilizzata per la manipolazione efficiente delle stringhe. A differenza della classe `String`, che crea nuove istanze ogni volta che modifichi il contenuto di una stringa (dato che le stringhe sono immutabili), `StringBuilder` permette di modificare il contenuto senza creare nuove istanze, migliorando così le prestazioni, soprattutto quando si devono effettuare molte operazioni sulle stringhe.

Quando usare `StringBuilder`?

- Quando esegui operazioni di concatenazione ripetute su una stringa.
- Quando modifichi frequentemente una stringa esistente.
- Quando lavori con stringhe molto lunghe.

Principali metodi di `StringBuilder`:

1. **Append**: Aggiunge il testo alla fine della stringa.
2. **Insert**: Inserisce del testo in una posizione specificata.
3. **Remove**: Rimuove una porzione di testo.
4. **Replace**: Sostituisce tutte le occorrenze di una stringa con un'altra.
5. **ToString**: Restituisce la stringa finale risultante.

Esempio:

```
using System;
using System.Text;

class Program
{
    static void Main()
    {
        // Creazione di un'istanza di StringBuilder
        StringBuilder sb = new StringBuilder("Ciao");

        // Append: aggiunge testo alla fine
        sb.Append(" mondo!");
    }
}
```

```

Console.WriteLine(sb); // Output: "Ciao mondo!"

// Insert: inserisce testo in una posizione specifica
sb.Insert(5, "bellissimo ");

Console.WriteLine(sb); // Output: "Ciao bellissimo mondo!"

// Replace: sostituisce una parte della stringa
sb.Replace("mondo", "universo");

Console.WriteLine(sb); // Output: "Ciao bellissimo universo!"

// Remove: rimuove una parte della stringa
sb.Remove(5, 11); // Rimuove "bellissimo "

Console.WriteLine(sb); // Output: "Ciao universo"

// ToString: converte il contenuto finale di StringBuilder in una stringa
string result = sb.ToString();

Console.WriteLine(result); // Output: "Ciao universo"

}

}

...

```

Spiegazione:

1. `StringBuilder sb = new StringBuilder("Ciao");`: Crea un'istanza di `StringBuilder` con il testo iniziale "Ciao".
2. `Append(" mondo!");`: Aggiunge " mondo!" alla fine del testo esistente.
3. `Insert(5, "bellissimo ");`: Inserisce "bellissimo " alla posizione indicata (dopo "Ciao ").

4. `Replace("mondo", "universo");`: Sostituisce tutte le occorrenze di "mondo" con "universo".
5. `Remove(5, 11);`: Rimuove 11 caratteri a partire dalla posizione 5 (elimina "bellissimo").
6. `ToString();`: Restituisce la stringa risultante.

Vantaggi di `StringBuilder` rispetto a `String`:

- **Efficienza**: Evita la creazione di nuovi oggetti ogni volta che modifichi una stringa.
- **Performance**: È ideale per operazioni ripetute di modifica, aggiunta o rimozione di stringhe.

Per esempio, se dovessi concatenare molte stringhe in un ciclo, usare `StringBuilder` sarebbe molto più efficiente rispetto alla semplice concatenazione con la classe `String`, perché non si creano nuovi oggetti ad ogni modifica.

6.3 LE CLASSI FILESTREAM, STREAMREADER E STREAMWRITER

In C#, la gestione degli **stream di file** è un'operazione comune quando si lavora con file di input/output (lettura e scrittura di file). Gli **stream** rappresentano una sequenza di byte che può essere letta o scritta. C# offre diverse classi per lavorare con gli stream, la più comune è `FileStream`, che ti permette di leggere da e scrivere su un file a basso livello, ovvero a livello di byte.

Oltre a `FileStream`, ci sono altre classi più astratte e di più alto livello come `StreamReader` e `StreamWriter` (per la lettura e scrittura di file di testo).

Principali classi utilizzate per gestire file stream:

1. **`FileStream`**: Permette di leggere o scrivere su file a livello di byte.
2. **`StreamReader`**: Legge i dati da un file di testo.
3. **`StreamWriter`**: Scrive dati in un file di testo.

1. Utilizzo di `FileStream`

Con `FileStream`, puoi aprire un file, leggere o scrivere byte, e poi chiudere il file dopo l'operazione.

Esempio di lettura e scrittura con `FileStream`:

```
using System;  
using System.IO;  
  
class Program  
{  
    static void Main()  
    {  
        string filePath = "test.txt";
```

```

// Scrittura su file utilizzando FileStream

using (FileStream fs = new FileStream(filePath, FileMode.Create))

{
    byte[] data = System.Text.Encoding.UTF8.GetBytes("Ciao, mondo!");

    fs.Write(data, 0, data.Length);
}

// Lettura da file utilizzando FileStream

using (FileStream fs = new FileStream(filePath, FileMode.Open))

{
    byte[] data = new byte[fs.Length];

    fs.Read(data, 0, data.Length);

    string result = System.Text.Encoding.UTF8.GetString(data);

    Console.WriteLine("Contenuto del file: " + result);
}

}

```

Spiegazione del codice:

1. **Scrittura su file:**
 - Si apre il file `test.txt` in modalità `FileMode.Create`, che crea un nuovo file o sovrascrive quello esistente.
 - Si converte la stringa `"Ciao, mondo!"` in un array di byte con `Encoding.UTF8.GetBytes`.
 - Si scrivono i byte nel file usando `fs.Write`.

- Il blocco `using` assicura che lo stream venga chiuso automaticamente dopo l'uso.

2. **Lettura dal file**:

- Si apre il file in modalità ` FileMode.Open` .
- Si legge tutto il contenuto del file in un array di byte e si convertono i byte in stringa usando ` Encoding.UTF8.GetString` .
- Si stampa il contenuto del file.

2. Utilizzo di `StreamReader` e `StreamWriter`

Se si desidera leggere o scrivere file di testo in modo più semplice, è consigliabile usare ` StreamReader` e ` StreamWriter` , che offrono metodi di più alto livello rispetto a ` FileStream` .

Esempio di utilizzo di `StreamWriter` e `StreamReader` :

```
using System;
using System.IO;

class Program
{
    static void Main()
    {
        string filePath = "test.txt";

        // Scrittura su file di testo usando StreamWriter
        using (StreamWriter writer = new StreamWriter(filePath))
        {
            writer.WriteLine("Ciao, mondo!");
            writer.WriteLine("Questo è un esempio di scrittura su file.");
        }
    }
}
```

```

    }

// Lettura da file di testo usando StreamReader

using (StreamReader reader = new StreamReader(filePath))

{
    string line;

    while ((line = reader.ReadLine()) != null)

    {
        Console.WriteLine(line);

    }

}

}

```

Spiegazione:

1. **Scrittura con `StreamWriter`**:

- `StreamWriter` è utilizzato per scrivere file di testo in modo semplice, usando il metodo `WriteLine` per scrivere linee di testo nel file.
- Il blocco `using` si assicura che lo stream venga chiuso automaticamente.

2. **Lettura con `StreamReader`**:

- `StreamReader` è usato per leggere linee di testo da un file con il metodo `ReadLine`.
- Ogni riga viene letta e stampata fino a quando non si arriva alla fine del file (quando `ReadLine` restituisce `null`).

Modalità di apertura dei file (FileMode):

Quando lavori con `FileStream`, puoi specificare la modalità di apertura del file utilizzando l'enumerazione `FileMode` :

- `FileMode.Create`: Crea un nuovo file. Se il file esiste già, viene sovrascritto.
- `FileMode.Open`: Apre un file esistente. Se il file non esiste, viene generata un'eccezione.
- `FileMode.Append`: Apre il file e posiziona il puntatore alla fine del file per aggiungere nuovi dati.
- `FileMode.Truncate`: Apre il file e ne cancella tutto il contenuto.

Altri concetti utili:

- `FileAccess`: Specifica se il file deve essere aperto in lettura, scrittura o entrambi (es. `FileAccess.Read`, `FileAccess.Write`, `FileAccess.ReadWrite`).
- `FileShare`: Indica il livello di condivisione del file con altri processi (es. `FileShare.Read`).

Conclusione:

La gestione degli stream di file in C# offre molta flessibilità, con `FileStream` che permette di lavorare a livello di byte e `StreamReader/StreamWriter` che forniscono un'interfaccia di più alto livello per la lettura e scrittura di file di testo.

6.4 OVERRIDE DI UN METODO

La classe Veicolo contiene un metodo virtuale Descrizione(), che può essere sovrascritto dalle classi derivate.

La classe Auto eredita da Veicolo e usa override per ridefinire il metodo Descrizione().

Nel Main, creiamo un'istanza di Veicolo e una di Auto per vedere la differenza nel comportamento del metodo.

Codice in C#

```
using System;
```

```
class Veicolo
{
    public virtual void Descrizione()
    {
        Console.WriteLine("Sono un veicolo generico.");
    }
}
```

```
class Auto : Veicolo
{
    public override void Descrizione()
    {
        Console.WriteLine("Sono un'automobile.");
    }
}
```

```
class Program
```

```

{
    static void Main()
    {
        Veicolo mioVeicolo = new Veicolo();
        mioVeicolo.Descrizione(); // Output: Sono un veicolo generico.

        Auto miaAuto = new Auto();
        miaAuto.Descrizione(); // Output: Sono un'automobile.

        Veicolo veicoloPolimorfico = new Auto();
        veicoloPolimorfico.Descrizione(); // Output: Sono un'automobile.
    }
}

```

Cosa succede nel codice?

`mioVeicolo.Descrizione();` → Chiama il metodo della classe `Veicolo`.

`miaAuto.Descrizione();` → Chiama il metodo della classe `Auto`, che ha sovrascritto il metodo di `Veicolo`.

`veicoloPolimorfico.Descrizione();` → Anche se dichiarato come `Veicolo`, il metodo `Descrizione()` viene eseguito dalla classe `Auto`, grazie al polimorfismo.

VIRTUAL VS ABSTRACT

1. virtual

- Definisce un metodo nella classe base che ha un'implementazione predefinita.
- Le classi derivate **possono** sovrascriverlo usando **override**, ma non sono obbligate a farlo.

Esempio con **virtual**

```
using System;

class Veicolo
{
    public virtual void Descrizione()
    {
        Console.WriteLine("Sono un veicolo generico.");
    }
}

class Auto : Veicolo
{
    public override void Descrizione()
    {
        Console.WriteLine("Sono un'automobile.");
    }
}

class Program
{
    static void Main()
    {
        Veicolo mioVeicolo = new Veicolo();
        mioVeicolo.Descrizione(); // Output: Sono un veicolo
        generico.

        Auto miaAuto = new Auto();
        miaAuto.Descrizione(); // Output: Sono un'automobile.
```

```
    }  
}
```

- La classe **Auto** può scegliere se sovrascrivere il metodo **Descrizione()** o usare quello della classe base.
-

2. abstract

- Definisce un metodo **senza implementazione** nella classe base.
- **Obbliga** le classi derivate a fornire un'implementazione.
- La classe che contiene un metodo **abstract** deve essere dichiarata **abstract** e non può essere istanziata.

Esempio con **abstract**

```
using System;  
  
abstract class Veicolo  
{  
    public abstract void Descrizione(); // Metodo senza  
implementazione  
}  
  
class Auto : Veicolo  
{  
    public override void Descrizione()  
    {  
        Console.WriteLine("Sono un'automobile.");  
    }  
}  
  
class Program  
{  
    static void Main()  
    {  
        // Veicolo mioVeicolo = new Veicolo(); // Errore! Una classe  
astratta non può essere istanziata.  
  
        Auto miaAuto = new Auto();  
        miaAuto.Descrizione(); // Output: Sono un'automobile.
```

```
    }  
}
```

- Le classi derivate sono obbligate a implementare **Descrizione()**, altrimenti generano un errore.
-

Riepilogo delle differenze

Caratteristica	virtual	abstract
Ha un'implementazione predefinita?	<input checked="" type="checkbox"/> Sì	<input type="checkbox"/> No
Può essere sovrascritto?	<input checked="" type="checkbox"/> Sì (opzionale)	<input checked="" type="checkbox"/> Sì (obbligatorio)
La classe base può essere istanziata?	<input checked="" type="checkbox"/> Sì	<input type="checkbox"/> No
Deve essere usato con override ?	<input type="checkbox"/> No (ma consigliato)	<input checked="" type="checkbox"/> Sì (obbligatorio)

- ❖ **Quando usare **virtual**?** → Se vuoi fornire un comportamento predefinito, ma permettere alle classi derivate di modificarlo.
- ❖ **Quando usare **abstract**?** → Se vuoi obbligare tutte le classi derivate a definire il metodo in modo personalizzato.
-

Capitolo 7: Introduzione a Windows Forms

Cos'è Windows Forms Una tecnologia per costruire applicazioni desktop basate su GUI (Graphical User Interface) su Windows.

Creazione di un progetto Windows Forms in Visual Studio

- **Passaggi:**
 1. Apri Visual Studio.
 2. Crea un nuovo progetto.
 3. Seleziona "Windows Forms App".

Capitolo 8: Controlli di Base

Bottoni (Button)

Esempio:

```
Button button = new Button();
button.Text = "Cliccami";
button.Click += new EventHandler(Button_Click);
this.Controls.Add(button);
```

Etichette (Label)

Esempio:

```
Label label = new Label();
label.Text = "Benvenuto";
this.Controls.Add(label);
```

•

Caselle di testo (TextBox)

Esempio:

```
TextBox textBox = new TextBox();
this.Controls.Add(textBox);
```

Pannelli (Panel)

Esempio:

```
Panel panel = new Panel();
this.Controls.Add(panel);
```

Capitolo 9: Eventi e Gestione degli Eventi

Cosa sono gli eventi Gli eventi sono azioni come il clic di un pulsante che possono essere gestite tramite codice.

Associazione di eventi ai controlli

Esempio:

```
button.Click += new EventHandler(Button_Click);

private void Button_Click(object sender, EventArgs e)
{
    MessageBox.Show("Pulsante cliccato");
}
```

Capitolo 10: Layout e Design

Organizzazione dei controlli

- **Esempio:** Utilizzare `FlowLayoutPanel` e `TableLayoutPanel` per una disposizione flessibile.

Utilizzo di container come Panel e GroupBox

Esempio:

```
GroupBox groupBox = new GroupBox();  
groupBox.Text = "Informazioni";  
this.Controls.Add(groupBox);
```

Capitolo 11: Esempi Pratici

Calcolatrice Semplice

- **Descrizione:** Una semplice calcolatrice che permette operazioni base come somma, sottrazione, moltiplicazione e divisione.
- **Esempio di codice:** Implementare l'evento di click per eseguire operazioni matematiche.

Rubrica

- **Descrizione:** Un'app che permette di aggiungere, modificare e visualizzare contatti.
- **Esempio di codice:** Utilizzare `ListView` per visualizzare i contatti.

Capitolo 12: Gestione di Dati con Liste e Array

Utilizzo di liste per gestire collezioni di dati

Esempio:

```
List<string> nomi = new List<string>();  
nomi.Add("Mario");  
nomi.Add("Luigi");
```

Visualizzazione di dati con ListView e DataGridView

Esempio:

```
ListView listView = new ListView();  
listView.Items.Add(new ListViewItem("Mario"));  
this.Controls.Add(listView);
```

Capitolo 13: Persistenza dei Dati

Lettura e scrittura su file

Esempio:

```
File.WriteAllText("dati.txt", "Ciao, mondo!");  
string contenuto = File.ReadAllText("dati.txt");
```

Esempio

PARTE CREATA DAL DESIGNER DEL FORM

```
private System.Windows.Forms.ListBox lista_contatti;  
  
this.lista_contatti = new System.Windows.Forms.ListBox();  
  
private void salva_Click(object sender, EventArgs e)  
{  
  
    string file = @"contatti.txt";  
  
  
    using (StreamWriter writer = new StreamWriter(file))  
    {  
  
        foreach (var item in lista_contatti.Items)  
        {  
  
            writer.WriteLine(item);  
  
        }  
  
    }  
  
    MessageBox.Show("Contatti salvati!");  
}
```

Serializzazione

Esempio:

```
BinaryFormatter formatter = new BinaryFormatter();
FileStream stream = new FileStream("data.bin", FileMode.Create);
formatter.Serialize(stream, listaOggetti);
stream.Close();
```

Capitolo 14: Applicazioni Multi-form

Creazione e navigazione tra più form

Esempio:

```
Form form2 = new Form2();
form2.Show();
this.Hide();
```

Passaggio di dati tra form

Esempio:

```
form2.ImpostaDati("Ciao");
```

Capitolo 15: APPROFONDIMENTI

Riepilogo e prospettive future Questo manuale fornisce una base per comprendere C#, la programmazione ad oggetti, e lo sviluppo di applicazioni desktop con Windows Forms.

Esercizi e Progetti finali

- **Esercizio 1:** Creare un'applicazione To-Do List.
 - **Esercizio 2:** Creare una mini applicazione di gestione biblioteca.
-

LINK ESTERNI

LE VARIABILI

<https://www.marcoalbasini.com/2020/12/le-variabili-in-c-sharp/>

Ereditarietà

<https://learn.microsoft.com/it-it/dotnet/csharp/fundamentals/tutorials/inheritance>

IL POLIMORFISMO

<https://www.marcoalbasini.com/2021/01/polimorfismo-in-c-sharp/#:~:text=Il%20polimorfismo%20in%20C%23%20%C3%A8,per%20diversi%20tipi%20di%20oggetti.>

Esempio di Polimorfismo Dinamico con Override dei Metodi

```
using System;

class Animale
{
    public virtual void FaiSuono()
    {
        Console.WriteLine("L'animale fa un suono.");
    }
}

class Cane : Animale
{
    public override void FaiSuono()
    {
        Console.WriteLine("Il cane abbaia.");
    }
}

class Gatto : Animale
{
    public override void FaiSuono()
    {
        Console.WriteLine("Il gatto miagola.");
    }
}

class Programma
{
    static void Main(string[] args)
    {
        Animale mioAnimale = new Animale();
        Animale mioCane = new Cane();
        Animale mioGatto = new Gatto();

        mioAnimale.FaiSuono(); // Output: L'animale fa un
        suono.
        mioCane.FaiSuono(); // Output: Il cane abbaia.
        mioGatto.FaiSuono(); // Output: Il gatto miagola.
    }
}
```

In questo esempio, la classe Animale ha un metodo FaiSuono che è definito come virtual. Le classi Cane e Gatto ereditano da Animale e sovrascrivono il metodo FaiSuono utilizzando la parola chiave override. Nel metodo Main, gli oggetti di tipo Cane e Gatto vengono trattati

come oggetti di tipo Animale, ma chiamano le loro rispettive implementazioni del metodo FaiSuono.

Il polimorfismo permette quindi di scrivere codice più flessibile e riutilizzabile, facilitando la manutenzione e l'estensibilità del software.

LE CLASSI ASTRATTE

Nel polimorfismo in c sharp se dichiariamo un metodo con la parola chiave abstract tale metodo deve essere semplicemente un segnaposto privo di implementazione che indica alle classi derivate che devono obbligatoriamente implementare tale metodo. Abbiamo visto che se un metodo viene dichiarato astratto allora tutta la classe deve essere abstract. Le classi derivate dovranno continuare ad usare la keyword override. Le classi astratte possono contenere metodi di classe static che hanno una loro implementazione, quindi possono essere implementate parzialmente.

```
using System;  
  
namespace Polimorfismo  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Animale i = new Insetto();  
            i.MyMethod();  
            Animale.MyMethod2();  
        }  
    }  
    public abstract class Animale  
    {  
        public abstract void MyMethod();  
    }  
}
```

```
public static void MyMethod2()
{
    Console.WriteLine("Sto eseguendo il metodo statico MyMethod2() della Superclasse.");
}

}

public class Insetto:Animale
{
    public override void MyMethod()
    {
        Console.WriteLine("Sto eseguendo MyMethod() della Sottoclasse.");
    }
}
```

16. ESERCIZI DI TIPO CONSOLE

ESERCIZIO STAMPA LA TABELLINA

Chiedi all'utente di inserire il numero della tabellina richiesta e stampala

1. CODICE IN C++

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

int main()
{
    int tabellina;
    do {
        cout << "tabellina del: ";
        cin >> tabellina;
        if (tabellina > 10 || tabellina < 0) {
            cout << "inserisci un numero tra 0 e 10" << endl;
        }
    } while (tabellina > 10 || tabellina < 0);

    for (int i = 1; i <= 10; i++) {
        int ris = tabellina * i;
        cout << tabellina << " * " << i << " = " << ris << endl;
    }
}
```

2. CODICE IN C#

Ecco il codice C++ riscritto in C#:

```
using System;

class Program
{
    static void Main()
    {
        int tabellina;
        do
        {
            Console.Write("Tabellina del: ");
            tabellina = Convert.ToInt32(Console.ReadLine());

            if (tabellina > 10 || tabellina < 0)
            {
```

```

        Console.WriteLine("Inserisci un numero tra 0 e 10");
    }
} while (tabellina > 10 || tabellina < 0);

for (int i = 1; i <= 10; i++)
{
    int ris = tabellina * i;
    Console.WriteLine($"{tabellina} * {i} = {ris}");
}
}
}

```

3. Spiegazione delle modifiche:

Abbiamo utilizzato `Console.Write()` e `Console.WriteLine()` per l'output.
 Abbiamo utilizzato `Console.ReadLine()` per leggere l'input (ogni input viene letto come stringa).

Abbiamo usato `Convert.ToInt32` per convertire l'input dell'utente da stringa in un intero.

La sintassi per le stringhe usa l'interpolazione (`$(...)`) per una migliore leggibilità.

4. LEGGERE UN SINGOLO CARATTERE O UNA STRINGA IN INPUT

`Console.Read`` è un metodo della classe `Console` in C#, utilizzato per leggere il valore di un singolo carattere dalla console.
 Ecco alcune caratteristiche e dettagli su come funziona:

Caratteristiche di `Console.Read`

1. **Tipo di ritorno**:

- Restituisce un intero che rappresenta il valore Unicode del carattere letto. Se non ci sono caratteri da leggere (ad esempio, se si raggiunge la fine del flusso), restituisce -1.

2. **Uso**:

- È utile quando desideri leggere un singolo carattere, ad esempio per controllare input da tastiera senza premere "Invio" dopo ogni carattere.

- Tipicamente, viene utilizzato per gestire input di tipo carattere, come ad esempio per il controllo di sequenze di tasti.

3. **Input da tastiera**:

- Quando chiavi `Console.Read`, il programma attende che l'utente prema un tasto. Il carattere corrispondente viene quindi restituito come valore Unicode.

Esempio di utilizzo:

Ecco un semplice esempio che mostra come utilizzare `Console.Read`:

```
using System;

class Program
{
    static void Main()
    {
        Console.WriteLine("Premi un tasto qualsiasi...");
        int key = Console.Read(); // Legge un singolo carattere
        char character = (char)key; // Converte l'intero in char

        Console.WriteLine($"Hai premuto: {character}");
    }
}
```

Nota

Se vuoi leggere una stringa o una riga intera di testo (incluso l'input dell'utente fino a "Invio"), dovresti usare `Console.ReadLine()`, che restituisce una stringa e gestisce automaticamente il termine della riga.

18. ESERCIZI CON WINDOWS FORM

18.1 ESERCIZIO GENERATORE PASSWORD

Creare un'applicazione Windows Forms in C# che genera password casuali basate su lunghezza e criteri selezionati

Passaggi:

1. **Crea un nuovo progetto Windows Forms in Visual Studio:**

- Apri Visual Studio.
- Crea un nuovo progetto: `C# -> Windows Forms App (.NET Framework)`.
- Dai un nome al progetto (ad esempio: `PasswordGeneratorApp`).

2. **Progetta l'interfaccia utente:**

Aggiungi i seguenti controlli alla tua form:

- Una **TextBox** per inserire la lunghezza della password.
- **CheckBox** per scegliere se includere numeri, lettere e simboli.
- Un **Button** per generare la password.
- Una **Label** o un'altra **TextBox** per mostrare la password generata.

3. **Codice per la logica di generazione della password:**

Il seguente codice C# esegue la generazione della password in base ai criteri selezionati (numeri, lettere e simboli):

```

### Codice C#

using System;
using System.Text;
using System.Windows.Forms;

namespace PasswordGeneratorApp
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        // Metodo che genera una password

        private string GeneratePassword(int length, bool
includeNumbers, bool includeLetters, bool includeSymbols)
        {
            const string numbers = "0123456789";
            const string letters =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
            const string symbols = "!@#$%^&*()_-=<>?";

            StringBuilder characterPool = new StringBuilder();

            // Aggiunge i set di caratteri selezionati al pool di
caratteri

            if (includeNumbers)

```

```

        characterPool.Append(numbers);

        if (includeLetters)

            characterPool.Append(letters);

        if (includeSymbols)

            characterPool.Append(symbols);

        if (characterPool.Length == 0)

        {

            MessageBox.Show("Seleziona almeno un criterio per
generare la password!");

            return string.Empty;

        }

// Usa un generatore di numeri casuali

Random random = new Random();

StringBuilder password = new StringBuilder();

// Crea la password estraendo casualmente dai caratteri
disponibili

for (int i = 0; i < length; i++)

{

    int index = random.Next(characterPool.Length);

    password.Append(characterPool[index]);

}

return password.ToString();

}

```

```

// Evento click del pulsante "Genera Password"

private void btnGenerate_Click(object sender, EventArgs e)
{
    int length;

    if (!int.TryParse(txtLength.Text, out length) || length <=
0)

    {
        MessageBox.Show("Inserisci una lunghezza valida per la
password.");
        return;
    }

    bool includeNumbers = chkNumbers.Checked;
    bool includeLetters = chkLetters.Checked;
    bool includeSymbols = chkSymbols.Checked;

    // Chiama il metodo per generare la password

    string password = GeneratePassword(length, includeNumbers,
includeLetters, includeSymbols);

    txtPassword.Text = password;
}

}

### Spiegazione del codice:


```

- ****GeneratePassword**:** Questo metodo genera una password casuale basata sui criteri selezionati. Viene creato un pool di caratteri in base a quali opzioni sono state selezionate dall'utente (numeri, lettere, simboli).

- ****Random**:** Utilizziamo la classe `Random` per generare indici casuali all'interno del pool di caratteri.
- ****Evento btnGenerate_Click**:** Questo evento viene attivato quando l'utente fa clic sul pulsante "Genera Password". Verifica i valori inseriti (lunghezza e criteri) e genera la password chiamando il metodo `GeneratePassword`.

Progettazione dell'interfaccia grafica:

1. ****TextBox per la lunghezza**** (`txtLength`): Qui l'utente inserisce la lunghezza desiderata della password.
2. ****CheckBox per i criteri**** (`chkNumbers`, `chkLetters`, `chkSymbols`): Permettono di scegliere se includere numeri, lettere e simboli.
3. ****Button per generare la password**** (`btnGenerate`): Avvia il processo di generazione.
4. ****TextBox per la password generata**** (`txtPassword`): Qui viene mostrata la password generata.

Layout consigliato:

- ****Label:**** "Lunghezza Password"
- ****TextBox:**** `txtLength` (per la lunghezza)
- ****CheckBox:**** `chkNumbers` (per includere numeri)
- ****CheckBox:**** `chkLetters` (per includere lettere)
- ****CheckBox:**** `chkSymbols` (per includere simboli)
- ****Button:**** `btnGenerate` (per generare la password)
- ****Label/TextBox:**** `txtPassword` (per visualizzare la password)

Suggerimenti aggiuntivi:

- Puoi aggiungere funzionalità per copiare automaticamente la password negli appunti.

- Puoi aggiungere un controllo che garantisca la presenza di almeno un numero/lettera/simbolo se sono stati selezionati.

Spero che questa soluzione ti sia utile per creare un generatore di password! Se hai bisogno di ulteriori dettagli, chiedi pure.

18.2 ESERCIZIO GESTIONE PICTURE BOX

```
private void CaricaImmagine(string percorso)
{
    try
    {
        // Verifica se il file esiste
        if (System.IO.File.Exists(percorso))
        {
            // Carica l'immagine dal percorso specificato
            pictureBox1.Image = Image.FromFile(percorso);
            pictureBox1.SizeMode = PictureBoxSizeMode.StretchImage; // Adatta l'immagine alle dimensioni del PictureBox
        }
        else
        {
            MessageBox.Show("Il file non esiste.");
        }
    }
    catch (Exception ex)
    {
        // Gestisce eventuali eccezioni, ad esempio se il file non è un'immagine valida
        MessageBox.Show("Errore durante il caricamento dell'immagine: " + ex.Message);
    }
}
```

18.3 ESERCIZIO GESTIONE TIMER

```
using System;
using System.Windows.Forms;

namespace timer
{
    public partial class Form1 : Form
    {
```

```

private int tempo_passato = 0;
public Form1()
{
    InitializeComponent();
    gestioneTempo.Interval = 5000;
    gestioneTempo.Tick += new EventHandler(eseguiInParallelo);
    gestioneTempo.Start();

}

private void eseguiInParallelo(object sender, EventArgs e)
{
    tempo_passato = tempo_passato + 1;
    orologio.Text = Convert.ToString(tempo_passato);
}

}
}

```

18.4 ESERCIZIO PROGRAMMAZIONE ASINCRONA : BLACKJACK

(creato da W. B.)

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.IO;
using System.Linq;
using System.Reflection.Emit;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Windows.Forms;

//using static System.Net.Mime.MediaTypeNames;

//using static System.Net.Mime.MediaTypeNames;
using static System.Windows.Forms.VisualStyles.VisualStudioElement;
using static System.Windows.Forms.VisualStyles.VisualStudioElement.TextBox;
using static System.Windows.Forms.VisualStyles.VisualStudioElement.ToolBar;

```

```

namespace BlackJack
{
    public partial class Form1 : Form
    {
        public Form1()
        {

            InitializeComponent();

            MischiaArray(mazzo);
            CaricaFishes();
            IniziaGioco();

            // Associa l'evento FormClosing all'handler
            this.FormClosing += new FormClosingEventHandler(Form1_FormClosing);
        }
        //mazzo da 52 carte

        string[] mazzo = { "FA", "F2", "F3", "F4", "F5", "F6", "F7", "F8", "F9", "F0", "FJ", "FR",
        "FK",
            "CA", "C2", "C3", "C4", "C5", "C6", "C7", "C8", "C9", "C0", "CJ", "CR", "CK",
            "QA", "Q2", "Q3", "Q4", "Q5", "Q6", "Q7", "Q8", "Q9", "Q0", "QJ", "QR",
        "QK",
            "PA", "P2", "P3", "P4", "P5", "P6", "P7", "P8", "P9", "P0", "PJ", "PR", "PK"}; 

        string[] mazzoBase = { "FA", "F2", "F3", "F4", "F5", "F6", "F7", "F8", "F9", "F0", "FJ",
        "FR", "FK",
            "CA", "C2", "C3", "C4", "C5", "C6", "C7", "C8", "C9", "C0", "CJ", "CR",
        "CK",
            "QA", "Q2", "Q3", "Q4", "Q5", "Q6", "Q7", "Q8", "Q9", "Q0", "QJ", "QR",
        "QK",
            "PA", "P2", "P3", "P4", "P5", "P6", "P7", "P8", "P9", "P0", "PJ", "PR",
        "PK"}; 

        int cartePlayer = 0;
        int carteBanco = 0;
        int assiPlayer = 0;
        int assiBanco = 0;
        int fishesPlayer = 250;
        int puntata = 0;
        bool blackJack = false;
        string ultimaCartaPlayer = "";
        string ultimaCartaBanco = "";
        string elencoCartePlayer = "";
        string elencoCarteBanco = "";
    }
}

```

```

        string basePath = AppDomain.CurrentDomain.BaseDirectory; // Ottiene il percorso
base
        bool cambiaCarta = false;

        public static void DaiValore(string[] array, int indice, ref int valore, ref int assi)//ref per
passare la variabile per riferimento
    {
        //aggiunge il valore basandosi sul secondo carattere dell'array
        if (array[indice][1] == '1') valore += 1;
        else if (array[indice][1] == '2') valore += 2;
        else if (array[indice][1] == '3') valore += 3;
        else if (array[indice][1] == '4') valore += 4;
        else if (array[indice][1] == '5') valore += 5;
        else if (array[indice][1] == '6') valore += 6;
        else if (array[indice][1] == '7') valore += 7;
        else if (array[indice][1] == '8') valore += 8;
        else if (array[indice][1] == '9') valore += 9;
        else if (array[indice][1] == '0' || array[indice][1] == 'R' || array[indice][1] == 'K' ||
array[indice][1] == 'J') valore += 10;
        else if (array[indice][1] == 'A')
        {
            valore += 11;
            assi += 1;
        }
    }

    private string[] RimuoviCarta(string[] mazzo, int indice) // Restituisce un nuovo array
senza il valore desiderato
    {
        // Creare un nuovo array di dimensione ridotta
        string[] nuovoMazzo = new string[mazzo.Length - 1];

        // Copiare gli elementi nel nuovo array
        int j = 0; // Indice per il nuovo array
        for (int i = 0; i < mazzo.Length; i++)
        {
            // Se l'indice corrente non è quello da rimuovere, copia l'elemento
            if (i != indice) // Cambiato per controllare l'indice invece del valore
            {
                nuovoMazzo[j] = mazzo[i];
                j++;
            }
        }

        // Restituisci il nuovo mazzo
        return nuovoMazzo;
    }

```

```

public static void MischiaArray(string[] array)//mischia l'array delle carte
{
    Random rnd = new Random();

    for (int i = array.Length - 1; i > 0; i--)
    {
        int j = rnd.Next(0, i + 1); // Ottiene un indice casuale tra 0 e i
                                    // Scambia array[i] con array[j]
        string temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
}

private void button1_Click(object sender, EventArgs e)//Player prende la carta
{

    Random rnd = new Random();
    int indice = rnd.Next(0, mazzo.Length);
    DaiValore(mazzo, indice, ref cartePlayer, ref assiPlayer); // ref per passare la
    variabile per riferimento
    sommaCarte.Text = Convert.ToString(cartePlayer);
    ultimaCartaPlayer = mazzo[indice];
    elencoCartePlayer += mazzo[indice] + " ";
    mazzo = RimuoviCarta(mazzo, indice); // Assegna il nuovo mazzo a mazzo

    string imagePath = Path.Combine(basePath, "Carte", $"{ultimaCartaPlayer}.png"); // 
    Combina il percorso con la cartella e il nome dell'immagine
    cartaPlayer.Image = Image.FromFile(imagePath);
    cartaPlayer.SizeMode = PictureBoxSizeMode.StretchImage;
    elCartePl.Text = "Elenco carte: " + elencoCartePlayer.ToString();

    Controlla21();

    if (cartePlayer >= 22)
    {
        if (assiPlayer == 0)
        {
            riparti.Visible = true;
            confermaPuntata.Enabled = true;
            fishesRimanenti.Text = "Fishes rimanenti: " + fishesPlayer;
            boxInsFishes.Text = "";
            vitPerPar.Text = "Hai Perso!";
        }
        else
        {
            cartePlayer -= 10;
            assiPlayer = 0;
        }
    }
}

```

```

        sommaCarte.Text = Convert.ToString(cartePlayer);
    }

}

private void ControllaVincitore() // Metodo per controllare chi ha vinto
{
    if (carteBanco > 21)
    {
        riparti.Visible = true;
        confermaPuntata.Enabled = true;
        fishesPlayer += puntata * 2;
        vitPerPar.Text = "Hai Vinto!";
        chiama.Enabled = false;//disabilita il bottone per chiamare la carta
        stai.Enabled = false;//disabilita il bottone per chiamare la carta
    }
    else if (cartePlayer > carteBanco)
    {
        riparti.Visible = true;
        confermaPuntata.Enabled = true;
        fishesPlayer += puntata * 2;
        vitPerPar.Text = "Hai Vinto!";
        chiama.Enabled = false;//disabilita il bottone per chiamare la carta
        stai.Enabled = false;//disabilita il bottone per chiamare la carta
    }
    else if (cartePlayer < carteBanco)
    {
        riparti.Visible = true;
        confermaPuntata.Enabled = true;
        fishesRimanenti.Text = "Fishes rimanenti: " + fishesPlayer;
        boxInsFishes.Text = "";
        vitPerPar.Text = "Hai Perso!";
        chiama.Enabled = false;//disabilita il bottone per chiamare la carta
        stai.Enabled = false;//disabilita il bottone per chiamare la carta
    }
    else
    {
        riparti.Visible = true;
        confermaPuntata.Enabled = true;
        fishesRimanenti.Text = "Fishes rimanenti: " + fishesPlayer;
        boxInsFishes.Text = "";
        vitPerPar.Text = "Pareggio!";
        fishesPlayer += puntata;
        chiama.Enabled = false;//disabilita il bottone per chiamare la carta
        stai.Enabled = false;//disabilita il bottone per chiamare la carta
    }
}

```

```

        if (fishesPlayer <= 0)
        {
            MessageBox.Show("Hai finito le fishes", "Fine!", MessageBoxButtons.OK,
MessageBoxIcon.Information);
            Application.Restart(); // Riavvia l'applicazione
        }
    }

private void Controlla21()//blackjack
{
    if (carteBanco == 21)
    {
        blackJack = true;
        riparti.Visible = true;
        vitPerPar.Text = "Hai Perso!";
    }
    else if (cartePlayer == 21)
    {
        blackJack = true;
        riparti.Visible = true;
        vitPerPar.Text = "Hai Vinto! BLACKJACK!!!";
        fishesPlayer += puntata * 3;
    }
}

private void stai_Click(object sender, EventArgs e)
{
    chiama.Enabled = false;//disabilita il bottone per chiamare la carta

    if (carteBanco < cartePlayer)
    {

        // Avvia il ciclo delle immagini quando viene premuto un bottone
        StartImageLoop();
    }
}

/*GRAZIE, CHATGPT
 * Esecuzione asincrona e UI non bloccata:
L'uso di async e await con Task.Delay() permette di ritardare il ciclo in modo
da cambiare l'immagine ogni 3 secondi, senza bloccare l'interfaccia utente.
Questo approccio è ideale per applicazioni Windows Forms che devono restare
reattive mentre eseguono operazioni a intervalli regolari.

```

```

        */
private async void StartImageLoop()
{
    while (carteBanco < 17)
    {
        await Task.Delay(3000);

        TurnoBanco();

        /* Aspetta per 3 secondi prima di passare all'immagine successiva
        await Task.Delay(3000): Questo interrompe l'esecuzione del ciclo per 3 secondi
        (3000 millisecondi) tra una iterazione e l'altra, senza bloccare
        l'interfaccia utente, permettendo all'applicazione di rimanere reattiva.
        */
    }

    if (carteBanco > cartePlayer)
    {
        break;
    }
}

Controlla21();

if (!blackJack)
{
    ControllaVincitore();
}
else
{
    chiama.Enabled = false;//disabilita il bottone per chiamare la carta
    stai.Enabled = false;//disabilita il bottone per chiamare la carta
    riparti.Visible = true;
    confermaPuntata.Enabled = true;
    fishesRimanenti.Text = "Fishes rimanenti: " + fishesPlayer;
    boxInsFishes.Text = "";
    confermaPuntata.Enabled = true;
}

private void TurnoBanco()
{
    if (carteBanco <= 16)
    {
        Random rnd = new Random();

```

```

        int indice = rnd.Next(0, mazzo.Length);
        DaiValore(mazzo, indice, ref carteBanco, ref assiBanco); // ref per passare la
variabile per riferimento
        ultimaCartaBanco = mazzo[indice];
        elencoCarteBanco += mazzo[indice] + " ";
        mazzo = RimuoviCarta(mazzo, indice); // Assegna il nuovo mazzo a mazzo
// Combina il percorso con la cartella e il nome dell'immagine
cambiaCarta = false;
string imagePath = Path.Combine(basePath, "Carte", $"{ultimaCartaBanco}.png");

cartaBanco.Image = Image.FromFile(imagePath);
cartaBanco.SizeMode = PictureBoxSizeMode.StretchImage;
nBanco.Text = Convert.ToString(carteBanco);
elCarteBan.Text = "Elenco carte: " + elencoCarteBanco.ToString();

Controlla21();
if (carteBanco >= 22)
{
    if (assiBanco == 0)
    {
        ControllaVincitore();
    }
    else
    {
        carteBanco -= 10;
        assiBanco = 0;
        nBanco.Text = Convert.ToString(carteBanco);
    }
}

}

}

private void IniziaGioco()
{
    riparti.Visible = false;
    cartePlayer = 0;
    carteBanco = 0;
    assiPlayer = 0;
    assiBanco = 0;
    puntata = 0;
    blackJack = false;
    ultimaCartaPlayer = "";
    ultimaCartaBanco = "";
    chiama.Enabled = false;//disabilita il bottone per chiamare la carta
    stai.Enabled = false;//disabilita il bottone per chiamare la carta
}

```

```

vitPerPar.Text = "";
boxInsFishes.Text = "";
elencoCarteBanco = "";
elencoCartePlayer = "";
confermaPuntata.Enabled = true;

// Distribuisci una carta al giocatore
if (mazzo.Length > 0)
{
    Random rnd = new Random();
    int indice = rnd.Next(0, mazzo.Length);
    DaiValore(mazzo, indice, ref cartePlayer, ref assiPlayer);
    ultimaCartaPlayer = mazzo[indice];

    string imagePath = Path.Combine(basePath, "Carte", $"{ultimaCartaPlayer}.png");
    // Combina il percorso con la cartella e il nome dell'immagine
    cartaPlayer.Image = Image.FromFile(imagePath);
    cartaPlayer.SizeMode = PictureBoxSizeMode.StretchImage;
    elencoCartePlayer += mazzo[indice] + " ";
    mazzo = RimuoviCarta(mazzo, indice); // Rimuovi la carta dal mazzo
}

// Distribuisci una carta al banco
if (mazzo.Length > 0)
{
    Random rnd = new Random();
    int indice = rnd.Next(0, mazzo.Length);
    DaiValore(mazzo, indice, ref carteBanco, ref assiBanco);
    ultimaCartaBanco = mazzo[indice];

    string imagePath = Path.Combine(basePath, "Carte", $"{ultimaCartaBanco}.png");
    // Combina il percorso con la cartella e il nome dell'immagine
    cartaBanco.Image = Image.FromFile(imagePath);
    cartaBanco.SizeMode = PictureBoxSizeMode.StretchImage;
    elencoCarteBanco += mazzo[indice] + " ";
    mazzo = RimuoviCarta(mazzo, indice); // Rimuovi la carta dal mazzo
}

// Aggiorna le etichette per mostrare il punteggio iniziale
sommaCarte.Text = Convert.ToString(cartePlayer);
nBanco.Text = Convert.ToString(carteBanco);
fishesRimanenti.Text = "Fishes rimanenti: " + fishesPlayer;
mazzo = mazzoBase;
elCartePl.Text = "Elenco carte: " + elencoCartePlayer.ToString();
elCarteBan.Text = "Elenco carte: " + elencoCarteBanco.ToString();
}

```

```

private void label1_Click_1(object sender, EventArgs e)
{
}

private void Form1_Load(object sender, EventArgs e)
{
}

private void nBanco_TextChanged(object sender, EventArgs e)
{
    nBanco.Text = Convert.ToString(carteBanco);
}

private void sommaCarte_TextChanged(object sender, EventArgs e)
{
    sommaCarte.Text = Convert.ToString(cartePlayer);
}

private void cartaPlayer_Click(object sender, EventArgs e)
{
}

private void cartaBanco_Click(object sender, EventArgs e)
{
}

private void fishesRimanenti_Click(object sender, EventArgs e)
{
    fishesRimanenti.Text = "Fishes rimanenti: " + fishesPlayer;
}

private void boxInsFishes_TextChanged(object sender, EventArgs e)
{
}

private void confermaPuntata_Click(object sender, EventArgs e)
{
    try
    {
        puntata = Convert.ToInt32(boxInsFishes.Text);
        if (puntata <= 0 || puntata > fishesPlayer)
        {

```

```

        boxInsFishes.Text = "Errore!";
    }
    else
    {
        boxInsFishes.Text = "Puntata confermata!";
        confermaPuntata.Enabled = false;
        chiama.Enabled = true;//abilita il bottone per chiamare la carta
        stai.Enabled = true;//abilita il bottone per stare
        fishesPlayer -= puntata;
        fishesRimanenti.Text = "Fishes rimanenti: " + fishesPlayer;
    }

}

catch (Exception ex)
{
    boxInsFishes.Text = "Errore!";
}

}

private void riparti_Click(object sender, EventArgs e)
{
    IniziaGioco();
}

private void vitPerPar_Click(object sender, EventArgs e)
{
    vitPerPar.Text = "";
}

private void cashOut_Click(object sender, EventArgs e)
{
    SalvaFishes();
    Application.Exit(); // Chiude l'applicazione corrente
}

private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
    SalvaFishes(); // Salva le fishes quando l'utente chiude l'applicazione
}

private void SalvaFishes()
{
    string filePath = Path.Combine(basePath, "fishes.txt");
    File.WriteAllText(filePath, fishesPlayer.ToString());
}

private void CaricaFishes()
{

```

```

string filePath = Path.Combine(basePath, "fishes.txt");

if (File.Exists(filePath))
{
    string fishesText = File.ReadAllText(filePath);
    fishesPlayer = int.Parse(fishesText); // Ripristina il valore delle fishes salvate
}
else
{
    fishesPlayer = 250; // Valore di default se non esiste il file
}
}

private void nuovoGiocatore_Click(object sender, EventArgs e)
{
    fishesPlayer = 250;
    SalvaFishes();
    CaricaFishes();

    MessageBox.Show("Fishes resettate", "Nuovo Giocatore!", MessageBoxButtons.OK,
    MessageBoxIcon.Information);
    Application.Restart(); // Riavvia l'applicazione
}

private void elCartePl_Click(object sender, EventArgs e)
{
    elCartePl.Text = "Elenco carte: " + elencoCartePlayer.ToString();
}

private void elCarteBan_Click(object sender, EventArgs e)
{
    elCarteBan.Text = "Elenco carte: " + elencoCarteBanco.ToString();
}
}
}

```

18.5 ESERCIZIO DI GESTIONE DELLE IMMAGINI : TRIS

```

using System;
using System.Drawing;
using System.Windows.Forms;

namespace gioco_del_tris
{

```

```
public partial class Form1 : Form
{
    private char turno;
    public Form1()
    {
        InitializeComponent();
        turno = 'X';

    }

    private void Form1_Load(object sender, EventArgs e)
    {
        // Inizializza il gioco se necessario
    }

    private void button1_Click(object sender, EventArgs e)
    {
        InserisciSimbolo(button1);
    }

    private void button2_Click(object sender, EventArgs e)
    {
        InserisciSimbolo(button2);
    }

    private void button3_Click(object sender, EventArgs e)
    {
        InserisciSimbolo(button3);
    }

    private void button4_Click(object sender, EventArgs e)
    {
        InserisciSimbolo(button4);
    }

    private void button5_Click(object sender, EventArgs e)
    {
        InserisciSimbolo(button5);
    }

    private void button6_Click(object sender, EventArgs e)
    {
        InserisciSimbolo(button6);
    }

    private void button7_Click(object sender, EventArgs e)
    {
```

```

        InserisciSimbolo(button7);
    }

private void button8_Click(object sender, EventArgs e)
{
    InserisciSimbolo(button8);
}

private void button9_Click(object sender, EventArgs e)
{
    InserisciSimbolo(button9);
}

private Image ResizeImage(Image img, int width, int height)
{
    Bitmap resizedImg = new Bitmap(width, height);
    using (Graphics g = Graphics.FromImage(resizedImg))
    {
        g.InterpolationMode =
System.Drawing.Drawing2D.InterpolationMode.HighQualityBicubic;
        g.DrawImage(img, 0, 0, width, height);
    }
    return resizedImg;
}

private void InserisciSimbolo(Button button)
{
//string imagePath = $"C:/Users/Studente/Documents/4C/gioco_del_tris/{turno}.png";
    string imagePath = $"img/{turno}.png";

    Image image = Image.FromFile(imagePath);

    // Ridimensiona l'immagine per adattarla al pulsante
    button.Image = ResizeImage(image, button1.Width, button1.Height);
    button.ImageAlign = ContentAlignment.MiddleCenter;

    //disabilita il bottone
    button.Enabled = false;

    if (turno == 'O')
    {
        turno = 'X';
    }
    else if (turno == 'X')
    {

```

```
    turno = 'O';
}

}

}
```

18.6 ESERCIZIO PER APRIRE E GESTIRE 2 FORM

Per aprire un nuovo FORM nel mio progetto devo:

1. cliccare su progetto: tasto destro>> aggiungi
2. scegliere nuovo elemento
3. scegliere modulo windows form
4. cliccare su aggiungi
5. ricompilare
6. aggiungere il pulsante nel primo form e scrivere il codice seguente nell'evento onclick

```
private void button2_Click(object sender, EventArgs e)
{
    Form2 form2 = new Form2();
    form2.Activate();
    form2.Show();

}
```

18.7 ESERCIZIO PER QUESTIONARIO

Creare un progetto Windows Form per gestire un questionario a scelta multipla con una classe dedicata: ecco una guida passo-passo per realizzare il progetto.

Struttura del progetto

1. **Crea la classe `Domanda`:** rappresenterà ogni domanda con le sue risposte e la risposta corretta.
2. **Usa una lista di oggetti `Domanda`:** per gestire il questionario.
3. **Crea l'interfaccia grafica (Windows Form):** per visualizzare le domande e registrare le risposte dell'utente.

Passaggi dettagliati

1. Creare il progetto

1. Apri Visual Studio.
2. Crea un nuovo progetto Windows Form App (.NET Framework o .NET 6/7 se hai un ambiente aggiornato).
3. Dai un nome al progetto, ad esempio `Questionario`.

2. Creare la classe `Domanda`

Aggiungi una classe al progetto:

1. Tasto destro sul progetto → **Aggiungi** → **Classe**.
2. Dai il nome `Domanda.cs` e definisci la classe come segue:

```
```csharp
public class Domanda
{
 public string Testo { get; set; }
 public List<string> Risposte { get; set; }
 public int IndiceRispostaCorretta { get; set; }

 public Domanda(string testo, List<string> risposte, int indiceRispostaCorretta)
 {
 Testo = testo;
 Risposte = risposte;
 IndiceRispostaCorretta = indiceRispostaCorretta;
 }

 public bool VerificaRisposta(int indice)
 {
 return indice == IndiceRispostaCorretta;
 }
}
```

```
 }
}
...

```

#### #### 3. Creare la lista di domande

In `Form1.cs`, inizializza una lista di domande nel costruttore del form o in un metodo dedicato:

```
```csharp  
List<Domanda> questionario = new List<Domanda>();  
  
private void InizializzaDomande()  
{  
    questionario.Add(new Domanda(  
        "Qual è la capitale d'Italia?",  
        new List<string> { "Roma", "Milano", "Napoli", "Torino" },  
        0  
    ));  
  
    questionario.Add(new Domanda(  
        "Qual è il risultato di 3+5?",  
        new List<string> { "5", "8", "9", "7" },  
        1  
    ));  
  
    questionario.Add(new Domanda(  
        "Chi ha scritto 'Divina Commedia'?",  
        new List<string> { "Dante", "Leopardi", "Manzoni", "Petrarca" },  
        0  
    ));  
}
```

Chiama `InizializzaDomande()` nel costruttore del form.

4. Progettare l'interfaccia utente

Nel designer del form:

1. Aggiungi un **Label** per mostrare la domanda.
2. Aggiungi un **GroupBox** contenente 4 **RadioButton** per le risposte.
3. Aggiungi un pulsante ***Avanti*** per passare alla domanda successiva.

Ad esempio:

- **Label:** `lblDomanda`
- **RadioButtons:** `rbRisposta1`, `rbRisposta2`, `rbRisposta3`, `rbRisposta4`

- **Button:** `btnAvanti`

5. Gestire il flusso delle domande

Aggiungi variabili e logica per gestire il questionario:

```
```csharp
private int indiceCorrente = 0;
private int punteggio = 0;

private void CaricaDomanda()
{
 if (indiceCorrente < questionario.Count)
 {
 var domanda = questionario[indiceCorrente];
 lblDomanda.Text = domanda.Testo;

 rbRisposta1.Text = domanda.Risposte[0];
 rbRisposta2.Text = domanda.Risposte[1];
 rbRisposta3.Text = domanda.Risposte[2];
 rbRisposta4.Text = domanda.Risposte[3];
 }
 else
 {
 MessageBox.Show($"Questionario completato! Punteggio:
{punteggio}/{questionario.Count}");
 btnAvanti.Enabled = false;
 }
}

private void btnAvanti_Click(object sender, EventArgs e)
{
 var domandaCorrente = questionario[indiceCorrente];
 int indiceRispostaSelezionata = -1;

 if (rbRisposta1.Checked) indiceRispostaSelezionata = 0;
 else if (rbRisposta2.Checked) indiceRispostaSelezionata = 1;
 else if (rbRisposta3.Checked) indiceRispostaSelezionata = 2;
 else if (rbRisposta4.Checked) indiceRispostaSelezionata = 3;

 if (indiceRispostaSelezionata == -1)
 {
 MessageBox.Show("Seleziona una risposta!");
 return;
 }

 if (domandaCorrente.VerificaRisposta(indiceRispostaSelezionata))
```

```
{
 punteggio++;
}

indiceCorrente++;
CaricaDomanda();
}
...

```

#### #### 6. Avviare il questionario

Nel costruttore di `Form1` o nel metodo `Load` del form, chiama `CaricaDomanda`:

```
```csharp  
public Form1()  
{  
    InitializeComponent();  
    InizializzaDomande();  
    CaricaDomanda();  
}  
...  
  
---
```

Risultato atteso

Quando avvii il programma:

1. Il form mostrerà la prima domanda con le opzioni a scelta multipla.
2. L'utente seleziona una risposta e clicca ***Avanti*** per passare alla domanda successiva.
3. Alla fine del questionario, verrà mostrato il punteggio totale.

```
---
```

18.8 ESERCIZIO PER VEICOLO

Creare un progetto Windows Form per gestire una concessionaria di veicoli (auto, moto, etc) con classi dedicate: ecco una guida passo-passo per realizzare il progetto.

Questo progetto simula una gestione di veicoli con caratteristiche comuni e specifiche, sfruttando le tecniche di programmazione orientata agli oggetti.

Il codice include anche l'uso di `try-catch` e proprietà (`get` e `set`) per il controllo degli attributi.

Struttura del progetto

1. **Classe base `Veicolo`**

- Dimostra l'incapsulamento con proprietà protette e metodi pubblici.

2. **Classi derivate (`Auto`, `Moto`)**

- Esempio di ereditarietà.

3. **Polimorfismo**

- Sovrascrittura di un metodo (`ToString`) per personalizzare il comportamento.

4. **Interfaccia Windows Form**

- Permette all'utente di aggiungere veicoli e visualizzarli.

Codice del progetto

Classe `Veicolo`

```
public class Veicolo
```

```
{
```

```

public string Modello { get; set; }

public string Marca { get; set; }

private int _anno;

public int Anno

{

    get { return _anno; }

    set

    {

        if (value < 1900 || value > DateTime.Now.Year)

            throw new ArgumentException("L'anno deve essere

compreso tra 1900 e l'anno attuale.");

        _anno = value;

    }

}

public Veicolo(string modello, string marca, int anno)

{

    Modello = modello;

    Marca = marca;

    Anno = anno;

}

public virtual string Descrizione()

{

    return $"Veicolo: {Marca} {Modello}, Anno: {Anno}";

}

```

```

```
Classi derivate

public class Auto : Veicolo

{

 public int NumeroPorte { get; set; }

 public Auto(string modello, string marca, int anno, int
numeroPorte)

 : base(modello, marca, anno)

 {

 NumeroPorte = numeroPorte;

 }

 public override string Descrizione()

 {

 return base.Descrizione() + $" , Porte: {NumeroPorte}";

 }

}

public class Moto : Veicolo

{

 public bool HaBauletto { get; set; }

 public Moto(string modello, string marca, int anno, bool
haBauletto)

 : base(modello, marca, anno)

 {

 }
```

```

 HaBauletto = haBauletto;

 }

 public override string Descrizione()
 {
 return base.Descrizione() + $" , Bauletto: {(HaBauletto ? "Sì" :
"No") }";
 }

}

```
```

```

#### \*\*Interfaccia Windows Form\*\*

Nel file `Form1.cs`:

1. \*\*Inizializza componenti\*\*: Aggiungi controlli per inserire i dettagli dei veicoli e un'area di output.
2. \*\*Lista di veicoli\*\*: Gestisce i veicoli aggiunti dall'utente.

```csharp

```

public partial class Form1 : Form
{
    private List<Veicolo> veicoli = new List<Veicolo>();

    public Form1()
    {
```

```

        InitializeComponent();

        cmbTipoVeicolo.Items.Add("Auto");

        cmbTipoVeicolo.Items.Add("Moto");

    }

private void btnAggiungiVeicolo_Click(object sender, EventArgs e)
{
    try
    {
        string tipo = cmbTipoVeicolo.SelectedItem.ToString();

        string modello = txtModello.Text;

        string marca = txtMarca.Text;

        int anno = int.Parse(txtAnno.Text);

        Veicolo nuovoVeicolo = null;

        if (tipo == "Auto")
        {
            int numeroPorte = int.Parse(txtPorte.Text);

            nuovoVeicolo = new Auto(modello, marca, anno,
numeroPorte);
        }
        else if (tipo == "Moto")
        {
            bool haBauletto = chkBauletto.Checked;

            nuovoVeicolo = new Moto(modello, marca, anno,
haBauletto);
        }
    }
}

```

```

        veicoli.Add(nuovoVeicolo);

        AggiornaListaVeicoli();

    }

    catch (Exception ex)

    {

        MessageBox.Show($"Errore: {ex.Message}", "Errore",
        MessageBoxButtons.OK, MessageBoxIcon.Error);

    }

}

private void AggiornaListaVeicoli()

{

    lstVeicoli.Items.Clear();

    foreach (var veicolo in veicoli)

    {

        lstVeicoli.Items.Add(veicolo.Descrizione());

    }

}

}

```

```

#### \*\*Designer del Form\*\*

#### 1. \*\*Controlli principali\*\*:

- \*\*ComboBox (`cmbTipoVeicolo`)\*\*: Seleziona il tipo di veicolo.

- **TextBox** (`txtModello`, `txtMarca`, `txtAnno`, `txtPorte`)\*\*:  
Inserisce i dettagli del veicolo.
- **CheckBox** (`chkBauletto`)\*\*: Specifica se la moto ha un bauletto.
- **Button** (`btnAggiungiVeicolo`)\*\*: Aggiunge il veicolo.
- **ListBox** (`lstVeicoli`)\*\*: Visualizza i veicoli inseriti.

## 2. Esempio di layout\*\*:

- **Etichette**: "Tipo Veicolo", "Modello", "Marca", "Anno", "Porte", "Bauletto".
- **Bottoni**: "Aggiungi Veicolo".
- **Lista**: Mostra i veicoli.

---

## # ## Esempio di utilizzo\*\*

1. L'utente seleziona "Auto" o "Moto" dal `ComboBox`.
2. Inserisce i dettagli richiesti (modello, marca, anno, ecc.).
3. Clicca su \*\*"Aggiungi Veicolo"\*\*.
4. Il programma aggiunge il veicolo alla lista e lo visualizza nel `ListBox`.
5. Se i dati inseriti sono errati (es. anno non valido), viene mostrato un messaggio di errore tramite `try-catch`.

---

Questo esempio è estensibile: puoi aggiungere altri tipi di veicoli o migliorare l'interfaccia utente per un'esperienza più avanzata!

## 18.9 ESERCIZIO PER CONVERTITORE DA BINARIO A DECIMALE

Creare un progetto Windows Form per convertire da binario a decimale e viceversa (creato da W. B.)

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace ConvertitoreBinDec
{
 public partial class Form1 : Form
 {
 public Form1()
 {
 InitializeComponent();
 }
 int tipoDiConversione = 0;
 private void btn1_Click(object sender, EventArgs e)
 {
 if (Convert.ToInt32(txtboxBin.Text) < 2147483647) txtboxBin.Text += "0";
 }

 private void btn2_Click(object sender, EventArgs e)
 {
 if (Convert.ToInt32(txtboxBin.Text) < 2147483647) txtboxBin.Text += "1";
 }

 private void btnBack_Click(object sender, EventArgs e)
 {
 string ultimo = "";
 for (int i = 0; i < txtboxBin.Text.Length - 1; i++) {
 ultimo += txtboxBin.Text[i];
 }
 txtboxBin.Text = ultimo;
 }

 private void btnConverti_Click(object sender, EventArgs e)
 {
```

```

txtboxDec.Text = "";
try
{
 if (tipoDiConversione == 0)
 {
 string numBin = txtboxBin.Text;
 int numDec = 0;
 int n = 0;
 for (int i = 0; i < numBin.Length; i++)
 {
 if (numBin[numBin.Length - 1 - i] == '0')
 n = 0;
 else
 n = 1;

 numDec += n * (int)Math.Pow(2, i);
 }
 txtboxDec.Text = numDec.ToString();
 }
 else
 {
 string numBin = txtboxBin.Text;
 int numDec = Convert.ToInt32(txtboxBin.Text);
 txtboxDec.Text += Convert.ToString(numDec, 2);
 }
}
catch {
 MessageBox.Show("Errore");
}
}

private void btnReset_Click(object sender, EventArgs e)
{
 txtboxBin.Text = "";
 txtboxDec.Text = "";
}

private void txtboxBin_TextChanged(object sender, EventArgs e)
{

}

private void btnTipConv_Click(object sender, EventArgs e)
{
 if(btnTipConv.Text == "Bin ➔ Dec")
 {

```

```

lblBin.Text = "Numero decimale";
lblDec.Text = "Numero binario";
txtboxBin.ReadOnly = false;
button2.Enabled = true;
button3.Enabled = true;
button4.Enabled = true;
button5.Enabled = true;
button6.Enabled = true;
button7.Enabled = true;
button8.Enabled = true;
button9.Enabled = true;
btnTipConv.Text = "Dec ➔ Bin";
tipoDiConversione = 1;
lblHelp.Visible = true;
}
else
{
 lblBin.Text = "Numero binario";
 lblDec.Text = "Numero decimale";
 txtboxBin.ReadOnly = true;
 button2.Enabled = false;
 button3.Enabled = false;
 button4.Enabled = false;
 button5.Enabled = false;
 button6.Enabled = false;
 button7.Enabled = false;
 button8.Enabled = false;
 button9.Enabled = false;
 btnTipConv.Text = "Bin ➔ Dec";
 tipoDiConversione = 0;
 lblHelp.Visible = false;
}
txtboxBin.Text = "";
txtboxDec.Text = "";
}

private void button2_Click(object sender, EventArgs e)
{
 if (Convert.ToInt32(txtboxBin.Text) < 2147483647) txtboxBin.Text += "2";
}

private void button3_Click(object sender, EventArgs e)
{
 if (Convert.ToInt32(txtboxBin.Text) < 2147483647) txtboxBin.Text += "3";
}

private void button4_Click(object sender, EventArgs e)

```

```
{
 if (Convert.ToInt32(txtboxBin.Text) < 2147483647) txtboxBin.Text += "4";
}

private void button5_Click(object sender, EventArgs e)
{
 if (Convert.ToInt32(txtboxBin.Text) < 2147483647) txtboxBin.Text += "5";
}

private void button6_Click(object sender, EventArgs e)
{
 if (Convert.ToInt32(txtboxBin.Text) < 2147483647) txtboxBin.Text += "6";
}

private void button7_Click(object sender, EventArgs e)
{
 if (Convert.ToInt32(txtboxBin.Text) < 2147483647) txtboxBin.Text += "7";
}

private void button8_Click(object sender, EventArgs e)
{
 if (Convert.ToInt32(txtboxBin.Text) < 2147483647) txtboxBin.Text += "8";
}

private void button9_Click(object sender, EventArgs e)
{
 if (Convert.ToInt32(txtboxBin.Text) < 2147483647) txtboxBin.Text += "9";
}
}
}
```