C# Propedeutico 1

Slide 1 – Introduzione a C#

C# è un linguaggio **moderno, tipizzato e orientato agli oggetti**, nato in casa Microsoft. È pensato per essere chiaro, potente e produttivo, adatto sia a chi inizia sia a chi costruisce applicazioni complesse.

Il suo punto di forza è la **robustezza** del framework .NET, che fornisce librerie, strumenti e sicurezza nella gestione della memoria.

```
// Il classico "Hello World"
using System;

class Program {
    static void Main() {
        Console.WriteLine("Ciao Mondo!");
    }
}
```

Slide 2 – Tipi di Dato

In C# tutto è fortemente tipizzato: ogni variabile ha un **tipo ben definito**, e non si può cambiare senza conversione esplicita.

I tipi più comuni sono: `int`, `double`, `bool`, `string`, e `char`. Esistono anche tipi complessi come classi e strutture.

```
int numero = 10;         // Intero
double prezzo = 9.99;    // Decimale
bool attivo = true;      // Booleano
string nome = "Marius";  // Testo
char iniziale = 'M';     // Singolo carattere
```

Slide 3 – Variabili e Costanti

Le **variabili** possono cambiare valore, le **costanti** no.

Le costanti si dichiarano con `const` e aiutano a rendere il codice più chiaro e meno soggetto a errori.

```
int eta = 31;
eta = 32; // Posso modificarla

const double PI = 3.1416;
// PI = 3.14; // Errore: una costante non può essere cambiata
```

Slide 4 – Operatori

Gli operatori permettono di **manipolare valori**: aritmetici (`+`, `-`, `*`, `/`), logici (`&&`, `||`, `!`) e di confronto (`==`, `!=`, `<`, `>`).

C# supporta anche operatori combinati come `+=` e `++` per scrivere in modo più compatto.

```
int x = 5;
x += 3; // x = 8
bool risultato = (x > 5) && (x < 10); // true
```

Slide 5 – Strutture Condizionali

Il costrutto `if` consente di **eseguire blocchi di codice solo se una condizione è vera**.

C# permette anche `else if` e `else` per gestire più casi in sequenza.

```
int voto = 28;

if (voto >= 30)
    Console.WriteLine("Eccellente!");
else if (voto >= 18)
    Console.WriteLine("Promosso!");
else
    Console.WriteLine("Bocciato!");
```

Slide 6 – Switch

Quando hai più condizioni sullo stesso valore, `switch` rende il codice **più leggibile e pulito** rispetto a tanti `if`.

```
string giorno = "Lunedì";

switch (giorno) {
    case "Lunedì":
        Console.WriteLine("Inizio settimana!");
        break;
    case "Venerdì":
        Console.WriteLine("Quasi weekend!");
        break;
    default:
        Console.WriteLine("Giorno qualsiasi.");
        break;
}
```

Slide 7 – Ciclo For

`for` serve per **ripetere un'azione un numero preciso di volte**.

È ideale quando conosci in anticipo quante iterazioni vuoi fare.

```
for (int i = 1; i <= 5; i++) {
    Console.WriteLine($"Numero: {i}");
}
```

Slide 8 – Ciclo While

`while` continua a eseguire un blocco **finché la condizione è vera**.

È utile quando non sai in anticipo quante volte dovrai ripetere qualcosa.

```
int i = 0;
while (i < 3) {
    Console.WriteLine("Ciao!");
    i++;
}
```

Slide 9 – Foreach

`foreach` è perfetto per **scorrere collezioni** come liste o array, senza dover gestire indici manualmente.

```
string[] nomi = { "Anna", "Luca", "Marius" };

foreach (string nome in nomi) {
    Console.WriteLine($"Ciao {nome}");
}
```

Slide 10 – Array

Un array è una **collezione di elementi dello stesso tipo**.

Gli indici partono da 0 e vanno fino a `Length - 1`.

```
int[] numeri = { 1, 2, 3, 4 };
Console.WriteLine(numeri[0]); // Stampa 1
Console.WriteLine(numeri.Length); // Stampa 4
```

Slide 11 – Liste

A differenza degli array, le `List<T>` possono **crescere dinamicamente**.

Sono la scelta più comune nelle app moderne.

```
using System.Collections.Generic;

List<string> amici = new List<string>() { "Luca", "Anna" };
amici.Add("Marius"); // Aggiunge un elemento
Console.WriteLine(amici.Count); // 3
```

Slide 12 – Dizionari

I `Dictionary<TKey, TValue>` associano **chiavi uniche a valori**, come delle piccole rubriche.

```
var contatti = new Dictionary<string, string>();
contatti["Luca"] = "1234";
```

```
contatti["Anna"] = "5678";

Console.WriteLine(contatti["Anna"]); // 5678
```

Slide 13 – Metodi

Un metodo è un **blocco riutilizzabile di codice** che esegue un'azione.

Può ricevere parametri e restituire valori.

```
int Somma(int a, int b) {
    return a + b;
}

Console.WriteLine(Somma(3, 4)); // 7
```

Slide 14 – Parametri e Return

Puoi **passare parametri** per rendere i metodi più generali.

`return` restituisce un valore al chiamante.

```
string Saluta(string nome) {
    return $"Ciao {nome}!";
}

Console.WriteLine(Saluta("Marius"));
```

Slide 15 – Overload di Metodi

C# permette più metodi con lo **stesso nome ma parametri diversi**, utile per gestire più casi simili.

```
void Stampa(string testo) ⇒ Console.WriteLine(testo);
void Stampa(int numero) ⇒ Console.WriteLine($"Numero: {numero}");

Stampa("Ciao");
Stampa(5);
```

Slide 16 – Scope e Visibilità

Le variabili vivono **solo nel blocco in cui sono dichiarate**.

Capire lo scope evita errori di visibilità e nomi duplicati.

```
int x = 10;
if (true) {
    int y = 5;
    Console.WriteLine(x + y);
}
// Console.WriteLine(y); // Errore: y non è visibile qui
```

Slide 17 – Tipi Valore e Riferimento

I **tipi valore** (come `int`) vengono copiati; i **tipi riferimento** (come `class`) puntano alla stessa istanza in memoria.

Capire la differenza è essenziale per evitare bug.

```
int a = 5;
int b = a;
b = 10;
Console.WriteLine(a); // 5

int[] arr1 = { 1, 2 };
int[] arr2 = arr1;
arr2[0] = 99;
Console.WriteLine(arr1[0]); // 99
```

Slide 18 – Gestione delle Eccezioni

Gli errori si gestiscono con `try`, `catch`, `finally`.

Serve per **evitare crash** e gestire situazioni impreviste in modo controllato.

```
try {
    int x = 10 / 0;
} catch (DivideByZeroException) {
    Console.WriteLine("Errore: divisione per zero!");
} finally {
```

```
    Console.WriteLine("Operazione terminata.");
}
```

Slide 19 – Namespace e Organizzazione

I namespace aiutano a **organizzare il codice in moduli** e prevenire conflitti di nomi tra classi.

```
namespace Scuola {
    class Studente {
        public string Nome;
    }
}
```

Slide 20 – Input da Console

Per rendere un programma interattivo, puoi **leggere input dall'utente** con `Console.ReadLine()`.

```
Console.Write("Come ti chiami? ");
string nome = Console.ReadLine();
Console.WriteLine($"Piacere di conoscerti, {nome}!");
```