# C# - 2

## Slide 21 – Enum

Le **enum** definiscono insiemi di costanti con nomi leggibili.

Servono per gestire valori discreti in modo chiaro e tipizzato.

```
enum Giorno { Lunedì, Martedì, Mercoledì }
Giorno oggi = Giorno.Martedì;
Console.WriteLine(oggi); // Martedì
```

---

## Slide 22 – Struct

Le **struct** sono simili alle classi ma sono **tipi valore**.

Si usano per dati leggeri e immutabili.

```
struct Punto {
    public int X, Y;
    public Punto(int x, int y) { X = x; Y = y; }
}
var p = new Punto(5, 10);
```

---

## Slide 23 – Tuple

Le **tuple** raggruppano più valori eterogenei senza creare una classe.

Possono avere nomi per i campi.

```
var persona = (Nome: "Marius", Età: 31);
Console.WriteLine($"{persona.Nome}, {persona.Età}");
```

---

## Slide 24 – Nullables

I **nullable types** permettono di assegnare `null` ai tipi valore.

Si indicano con `?` dopo il tipo.

```
int? voto = null;
if (voto.HasValue) Console.WriteLine(voto.Value);
else Console.WriteLine("Nessun voto");
```

---

## Slide 25 – Stringhe

Le **stringhe** sono oggetti immutabili.

Si concatenano con `+` o con interpolazione `$"..."`.

```
string nome = "Marius";
Console.WriteLine($"Ciao {nome}!");
```

---

## Slide 26 – Interpolazione e formattazione

L'interpolazione rende il codice più leggibile.

Si possono anche usare **formati numerici e date**.

```
double prezzo = 12.5;
Console.WriteLine($"Prezzo: {prezzo:C}"); // valuta locale
```

---

## Slide 27 – Date e ora

`DateTime` gestisce date e orari.

Si possono ottenere data corrente o creare nuove istanze.

```
DateTime adesso = DateTime.Now;
DateTime domani = adesso.AddDays(1);
Console.WriteLine(domani.ToShortDateString());
```

---

## Slide 28 – Operatore ternario

Permette di scrivere condizioni brevi in una sola riga.

Sintassi: `condizione ? valoreSeVero : valoreSeFalso`.

```
int eta = 20;
string stato = eta >= 18 ? "Adulto" : "Minorenne";
```

---

## Slide 29 – Namespace

I **namespace** organizzano il codice e evitano conflitti di nomi.

Si dichiarano con la parola chiave `namespace`.

```
namespace Scuola {
    class Studente { }
}
```

---

## Slide 30 – Modificatori di accesso

Definiscono la **visibilità** di membri e classi:

`public`, `private`, `protected`, `internal`.

```
class Persona {
    private string nome;
    public void SetNome(string n) ⇒ nome = n;
}
```

---

## Slide 31 – Classi statiche

Le **classi statiche** non possono essere istanziate.

Servono per **metodi di utilità** o costanti globali.

```
static class MathUtils {
    public static double Quadrato(double x) ⇒ x * x;
}
Console.WriteLine(MathUtils.Quadrato(5));
```

---

## Slide 32 – Metodi statici

Un **metodo statico** appartiene alla classe, non all'istanza.

Si invoca direttamente tramite il nome della classe.

```
class Calcolatrice {
    public static int Somma(int a, int b) ⇒ a + b;
}
Console.WriteLine(Calcolatrice.Somma(2, 3));
```

---

## Slide 33 – Costruttori statici

Eseguono logica una sola volta prima della prima istanza o chiamata.

Utili per inizializzare risorse condivise.

```
class Config {
    public static string Versione;
    static Config() { Versione = "1.0.0"; }
}
Console.WriteLine(Config.Versione);
```

---

## Slide 34 – Parametri opzionali e nominati

C# consente di **assegnare valori di default** ai parametri.

Possono essere passati anche per nome.

```
void Saluta(string nome = "Amico") {
    Console.WriteLine($"Ciao, {nome}");
}
Saluta(); Saluta(nome: "Marius");
```

---

## Slide 35 – Overriding

Il **metodo override** sostituisce il comportamento della classe base.

Serve `virtual` nel metodo originale e `override` in quello derivato.

```
class Base { public virtual void Info() => Console.WriteLine("Base"); }
class Derivata : Base { public override void Info() => Console.WriteLine("Der
ivata"); }
```

---

## Slide 36 – Sealed

`sealed` impedisce che una classe venga ereditata.

Serve a proteggere implementazioni specifiche.

```
sealed class Configurazione { }
```

---

## Slide 37 – Interfacce

Le **interfacce** definiscono **contratti** che le classi devono rispettare.

Non contengono implementazioni concrete.

```
interface ISalutabile {
    void Saluta();
}

class Persona : ISalutabile {
    public void Saluta() => Console.WriteLine("Ciao!");
}
```

---

## Slide 38 – Generics

I **generics** permettono di scrivere codice riutilizzabile per tipi diversi.

Evitano cast e aumentano la sicurezza del tipo.

```
class Box<T> {
    public T Valore;
}
Box<int> b = new Box<int> { Valore = 42 };
```

---

## Slide 39 – Collections

Oltre a `List<T>`, esistono molte collezioni utili:

`Dictionary`, `Queue`, `Stack`, `HashSet`.

```
var dizionario = new Dictionary<string, int>();
dizionario["uno"] = 1;
Console.WriteLine(dizionario["uno"]);
```

---

## Slide 40 – File e IO

C# gestisce i file tramite `System.IO`.

Si possono leggere e scrivere testi con pochi comandi.

```
using System.IO;

File.WriteAllText("testo.txt", "Ciao Mondo");
string contenuto = File.ReadAllText("testo.txt");
Console.WriteLine(contenuto);
```