C# Propedeutico 2

Slide 21 – Introduzione alla OOP

La **programmazione orientata agli oggetti** (OOP) è il cuore di C#: un modo di pensare il codice come un insieme di **entità che collaborano**.

Un oggetto rappresenta qualcosa di concreto — uno studente, un libro, un ordine — e ne racchiude **stato (dati)** e **comportamento (metodi)**.

```
class Studente {
    public string Nome;
    public int Eta;

    public void Saluta() {
        Console.WriteLine($"Ciao, sono {Nome} e ho {Eta} anni!");
    }
}
```

Slide 22 – Incapsulamento

L'incapsulamento serve a **proteggere i dati interni** di un oggetto, esponendo solo ciò che serve davvero.

Si ottiene usando modificatori di accesso come `private` e `public`, e proprietà con `get` e `set`.

```
class Studente {
    private int eta;
    public string Nome { get; set; }

    public int Eta {
        get ⇒ eta;
        set {
            if (value < 0) throw new ArgumentException("Età non valida");
            eta = value;
        }
```

```
    }
}
```

Slide 23 – Costruttori

I costruttori servono a **inizializzare correttamente** un oggetto al momento della creazione.

Possono ricevere parametri e preparare lo stato iniziale.

```
class Studente {
    public string Nome { get; }
    public int Eta { get; }

    public Studente(string nome, int eta) {
        Nome = nome;
        Eta = eta;
    }
}
```

Slide 24 – Metodi e Responsabilità

Un metodo rappresenta un'azione dell'oggetto.

È importante che ogni metodo abbia **una singola responsabilità chiara** — principio base del buon design.

```
class Studente {
    public string Nome { get; set; }
    public int Punteggio { get; private set; }

    public void AggiungiPunti(int punti) {
        Punteggio += punti; // Metodo con una sola responsabilità
    }
}
```

Slide 25 – Ereditarietà

L'ereditarietà consente di **riutilizzare codice comune** tra classi simili.

Una classe figlia estende quella base, ereditando proprietà e metodi.

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

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

Slide 26 – Costruttori ed Ereditarietà

Quando una classe eredita, il costruttore della base può essere **richiamato con** `base()` per inizializzare i dati condivisi.

```
class Persona {
    public string Nome { get; }
    public Persona(string nome) ⇒ Nome = nome;
}

class Studente : Persona {
    public int Matricola { get; }
    public Studente(string nome, int matricola) : base(nome) {
        Matricola = matricola;
    }
}
```

Slide 27 – Polimorfismo

Il polimorfismo consente di **usare oggetti diversi attraverso un'unica interfaccia comune**.

Così si può scrivere codice più flessibile e mantenibile.

```
class Persona {
    public virtual void Saluta() ⇒ Console.WriteLine("Ciao, sono una person
a!");
}
```

```
class Studente : Persona {
    public override void Saluta() => Console.WriteLine("Ciao, sono uno stude
nte!");
}

Persona p = new Studente();
p.Saluta(); // Usa la versione di Studente
```

Slide 28 – Classi Astratte

Le classi astratte servono come **base concettuale**, non possono essere istanziate direttamente.

Definiscono metodi comuni e lasciano altri da implementare.

```
abstract class Persona {
    public string Nome { get; set; }
    public abstract void Presentati();
}

class Studente : Persona {
    public override void Presentati() => Console.WriteLine($"Ciao, sono {No
me}!");
}
```

Slide 29 – Interfacce

Un'interfaccia è un **contratto di comportamento**.

Obbliga chi la implementa a fornire determinate funzioni, senza imporre un'implementazione concreta.

```
interface IValutabile {
    void Valuta(int punteggio);
}

class Studente : IValutabile {
    public int Punteggio { get; private set; }
```

```
    public void Valuta(int punteggio) => Punteggio = punteggio;
}
```

Slide 30 – Classi Statiche

Le classi statiche contengono solo **metodi e proprietà condivise**, senza bisogno di creare istanze.

Ideali per utility e funzioni generiche.

```
static class Calcolatrice {
    public static int Somma(int a, int b) => a + b;
}
```

Slide 31 – Tipi Generici

I **generics** permettono di scrivere codice riutilizzabile con tipi parametrizzati.

Così eviti duplicazioni e cast espliciti.

```
class Registro<T> {
    private List<T> elementi = new();
    public void Aggiungi(T elem) => elementi.Add(elem);
}

var registroStudenti = new Registro<Studente>();
```

Slide 32 – Enum

Un `enum` definisce un insieme di **valori simbolici** con nomi chiari, utili per rendere il codice leggibile.

```
enum LivelloCorso { Base, Intermedio, Avanzato }

class Studente {
    public LivelloCorso Livello { get; set; }
}
```

Slide 33 – Delegati

I delegati rappresentano **riferimenti a metodi**.

Sono alla base di eventi e programmazione reattiva.

```
delegate void Messaggio(string testo);

class Program {
    static void Saluta(string nome) ⇒ Console.WriteLine($"Ciao {nome}");
    static void Main() {
        Messaggio m = Saluta;
        m("Marius");
    }
}
```

Slide 34 – Eventi

Gli eventi permettono di **notificare cambiamenti** o azioni a più ascoltatori.

Un concetto centrale per decoupling e IoC.

```
class Studente {
    public event Action<string>? IscrizioneEffettuata;

    public void Iscriviti(string corso) {
        IscrizioneEffettuata?.Invoke(corso);
    }
}
```

Slide 35 – Principio di Responsabilità Singola

Ogni classe dovrebbe fare **una sola cosa e farla bene**.

Questo principio è alla base di un software mantenibile.

```
// Meglio separare le responsabilità:
class Studente { /* gestisce i dati */ }
class GestoreIscrizioni { /* gestisce iscrizioni */ }
```

Slide 36 – Accoppiamento e Coesione

Un buon design mira a **basso accoppiamento** (le classi si conoscono poco) e **alta coesione** (ogni classe fa solo ciò che le compete).

```
class EmailService {
    public void InviaEmail(string testo) { /* ... */ }
}

class Studente {
    private readonly EmailService _emailService;
    public Studente(EmailService service) ⇒ _emailService = service;
}
```

Slide 37 – Astrazione e Implementazione

L'astrazione ti fa pensare **al "cosa" e non al "come"**.

Separare l'interfaccia dall'implementazione rende il codice più flessibile.

```
interface INotifica { void Invia(string messaggio); }

class NotificaEmail : INotifica {
    public void Invia(string messaggio) ⇒ Console.WriteLine($"Email: {mess
aggio}");
}
```

Slide 38 – Pattern DRY e KISS

Due principi fondamentali:

DRY (Don't Repeat Yourself) evita duplicazioni di codice;

KISS (Keep It Simple, Stupid) invita a mantenere soluzioni semplici e dirette.

```
// Non duplicare:
void StampaStudente(Studente s) ⇒ Console.WriteLine(s.Nome);
// Riutilizza sempre metodi già esistenti.
```

Slide 39 – Testabilità e Design

Scrivere classi con **dipendenze iniettate e logica separata** rende il codice testabile.

È un passo verso architetture pulite e integrate con i test unitari.

```
class Studente {
    private readonly INotifica _notifica;
    public Studente(INotifica notifica) ⇒ _notifica = notifica;
}
```

Slide 40 – Riepilogo OOP

Con OOP impari a **modellare il mondo reale** nel codice.

Capendo classi, ereditarietà e interfacce, sei pronto per affrontare i principi **SOLID** e l'inversione di controllo.

```
var s = new Studente("Marius", 30);
s.Presentati(); // OOP in azione
```