C# - 3

Slide 41 – Introduzione a LINQ

LINQ (Language Integrated Query) permette di filtrare, ordinare e trasformare collezioni con sintassi dichiarativa.

Funziona su array, liste e sorgenti di dati.

```
int[] numeri = {1, 2, 3, 4, 5};
var pari = from n in numeri where n % 2 == 0 select n;
foreach (var n in pari) Console.WriteLine(n);
```

Slide 42 – LINQ con metodi

LINQ può essere scritto anche con **metodi di estensione** (`Where`, `Select`, `OrderBy`).

È la forma più comune in codice moderno.

```
var pari = numeri.Where(n ⇒ n % 2 == 0).Select(n ⇒ n * 10);
```

Slide 43 – Delegati

Un **delegato** rappresenta un riferimento a un metodo.

È la base per **eventi**, **callback** e **lambda expressions**.

```
delegate void Saluto(string nome);

void Ciao(string nome) ⇒ Console.WriteLine($"Ciao {nome}!");

Saluto s = Ciao;
s("Marius");
```

Slide 44 – Action e Func

C# include delegati pronti: **Action** (senza ritorno) e **Func** (con ritorno).

Si usano per semplificare codice e passare funzioni come parametri.

```
Action<string> saluta = n ⇒ Console.WriteLine($"Ciao {n}");
Func<int, int> quadrato = x ⇒ x * x;
```

Slide 45 – Lambda Expressions

Le **lambda** sono funzioni anonime.

Si usano spesso con LINQ o delegati inline.

```
var doppi = numeri.Select(n ⇒ n * 2);
```

Slide 46 – Eventi

Gli **eventi** notificano un cambiamento o un'azione.

Si basano sui delegati, seguendo il pattern "publisher/subscriber".

```
class Pulsante {
    public event Action Premuto;
    public void Premi() ⇒ Premuto?.Invoke();
}
```

Slide 47 – Iscriversi a un evento

I metodi possono **sottoscriversi** a un evento con `+=`.

L'evento chiama tutti gli handler quando viene invocato.

```
Pulsante btn = new Pulsante();
btn.Premuto += () ⇒ Console.WriteLine("Bottone premuto!");
btn.Premi();
```

Slide 48 – Delegati multicast

Un **delegato multicast** può contenere più metodi.

Vengono eseguiti in sequenza quando il delegato viene chiamato.

```
Action azione = () => Console.WriteLine("Uno");
azione += () => Console.WriteLine("Due");
azione();
```

Slide 49 – Anonymous methods

Prima delle lambda, si usavano **metodi anonimi** con `delegate`.

Sono ancora compatibili e utili in casi complessi.

```
Action saluta = delegate(string nome) {
    Console.WriteLine($"Ciao {nome}");
};
```

Slide 50 – Async e Await

C# supporta la **programmazione asincrona** per operazioni non bloccanti.

`async` dichiara un metodo asincrono, `await` sospende l'esecuzione.

```
async Task ScaricaDati() {
    await Task.Delay(1000);
    Console.WriteLine("Download completato");
}
```

Slide 51 – Task e Thread

Un **Task** rappresenta un'operazione asincrona.

Può essere avviato manualmente o con metodi async.

```
Task.Run(() => Console.WriteLine("In esecuzione su thread separato"));
```

Slide 52 – Async con ritorno

I metodi async possono restituire valori con `Task<T>`.

Il chiamante li recupera con `await`.

```
async Task<int> SommaAsync(int a, int b) {
    await Task.Delay(100);
    return a + b;
}
int risultato = await SommaAsync(2, 3);
```

Slide 53 – Exception handling in async

Le eccezioni nei metodi `async` vengono propagate nel `Task`.

Si gestiscono con `try-catch` attorno ad `await`.

```
try {
    await ScaricaDati();
} catch (Exception e) {
    Console.WriteLine($"Errore: {e.Message}");
}
```

Slide 54 – Pattern Singleton

Il **Singleton** garantisce un'unica istanza di una classe.

Si usa per configurazioni o servizi condivisi.

```
class Logger {
    private static Logger istanza;
    private Logger() {}
    public static Logger Istanza ⇒ istanza ??= new Logger();
}
```

Slide 55 – Pattern Factory

Il **Factory Pattern** crea oggetti senza esporre la logica di creazione.

Rende il codice più flessibile.

```
class VeicoloFactory {
    public static Veicolo Crea(string tipo) ⇒
```

```
        tipo == "auto" ? new Auto() : new Moto();
}
```

Slide 56 – Pattern Observer

Il **pattern Observer** implementa la comunicazione tra oggetti.

Un oggetto osservato notifica gli osservatori di ogni cambiamento.

```
interface IObserver { void Aggiorna(string msg); }
```

Slide 57 – Pattern Strategy

Il **Strategy Pattern** permette di cambiare comportamento a runtime.

Si definiscono algoritmi intercambiabili tramite interfacce.

```
interface IStrategia { void Esegui(); }
class StrategiaA : IStrategia { public void Esegui() ⇒ Console.WriteLine
("A"); }
```

Slide 58 – Garbage Collector

Il **GC** libera automaticamente la memoria non più usata.

Può essere forzato con `GC.Collect()`, ma è sconsigliato.

```
GC.Collect();
Console.WriteLine("Garbage collector eseguito");
```

Slide 59 – IDisposable e using

Le risorse non gestite (file, connessioni) vanno rilasciate con `IDisposable`.

`using` assicura la chiusura automatica.

```
using (var file = new StreamWriter("log.txt")) {
    file.WriteLine("Salvato");
```

```
}
```

Slide 60 – Conclusione

C# è un linguaggio **versatile e moderno**, capace di coprire ogni contesto — da desktop a web, da mobile a cloud.

Conoscere **OOP, LINQ, async** e i **pattern principali** è la base per scrivere codice **pulito, robusto e mantenibile**.

```
Console.WriteLine("Hai completato i Fondamenti + Avanzato di C#!");
```