Approfondimenti #2

Approfondimenti #2 – Costruttori, Nullables e Generics in C#

Slide 1 – Introduzione

In questo secondo approfondimento esploreremo tre pilastri del linguaggio C# che diventano fondamentali appena si supera la fase propedeutica:

- **I Costruttori**, che controllano la creazione e l'inizializzazione degli oggetti.
- **I tipi Nullable**, che permettono di gestire i valori null in modo sicuro.
- **I Generics**, che consentono di scrivere codice riutilizzabile e fortemente tipizzato.

Ogni sezione include teoria, best practice ed esempi di codice commentato.

SEZIONE 1 — COSTRUTTORI

Slide 2 – Cos'è un Costruttore

Un **costruttore** è un metodo speciale che viene chiamato automaticamente quando si crea un'istanza di una classe.

Serve a inizializzare gli attributi dell'oggetto o eseguire logica di setup.

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

    // Costruttore: stesso nome della classe, nessun tipo di ritorno
    public Persona(string nome, int eta)
    {
        Nome = nome;
        Eta = eta;
        Console.WriteLine("Oggetto Persona creato!");
    }
```

```
}

var mario = new Persona("Mario", 30); // Invoca automaticamente il costrut
tore
```

Spiegazione:

Il costruttore definisce i valori iniziali di `Nome` ed `Eta`. Quando viene creato l'oggetto `mario`, il messaggio viene stampato.

Slide 3 – Overloading dei Costruttori

Come per i metodi, i costruttori possono essere **sovraccaricati** per fornire diverse modalità di creazione.

```
class Rettangolo
{
   public int Larghezza, Altezza;

   // Costruttore base
   public Rettangolo(int lato)
   {
      Larghezza = lato;
      Altezza = lato;
   }

   // Costruttore alternativo
   public Rettangolo(int larghezza, int altezza)
   {
      Larghezza = larghezza;
      Altezza = altezza;
   }
}

// Possiamo creare un quadrato o un rettangolo
var quadrato = new Rettangolo(5);
var rettangolo = new Rettangolo(4, 8);
```

Spiegazione:

Il compilatore sceglie automaticamente il costruttore in base ai parametri forniti.

Slide 4 – Chiamata tra Costruttori (`this()`)

C# consente di **chiamare un costruttore da un altro** all'interno della stessa classe usando `this()`.

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

    // Costruttore principale
    public Studente(string nome, int eta, string corso)
    {
        Nome = nome;
        Eta = eta;
        Corso = corso;
    }

    // Costruttore che riusa quello principale
    public Studente(string nome) : this(nome, 18, "Informatica")
    {
        Console.WriteLine("Costruttore semplificato chiamato!");
    }
}

var s1 = new Studente("Lucia");
var s2 = new Studente("Marco", 22, "Fisica");
```

Spiegazione:

Il costruttore con un solo parametro riutilizza la logica del principale, riducendo la duplicazione di codice.

Slide 5 – Costruttore di Classe Base (`base()`)

Quando una classe eredita da un'altra, il costruttore della base viene eseguito **prima** di quello derivato.

```
class Persona
{
    public string Nome;
    public Persona(string nome)
    {
        Nome = nome;
        Console.WriteLine("Costruttore Persona");
    }
}

class Studente : Persona
{
    public string Corso;
    public Studente(string nome, string corso) : base(nome)
    {
        Corso = corso;
        Console.WriteLine("Costruttore Studente");
    }
}

var s = new Studente("Luca", "C#");
```

Spiegazione:

Il costruttore base viene richiamato con `base(nome)` per inizializzare i membri ereditati.

Slide 6 – Costruttore Statico

Un **costruttore statico** viene eseguito **una sola volta** per classe, non per istanza.

Serve per inizializzare membri statici o risorse condivise.

```
class Config
{
    public static string ConnectionString;

    // Costruttore statico: senza parametri, chiamato automaticamente
```

```
    static Config()
    {
        ConnectionString = "Server=localhost;DB=app;";
        Console.WriteLine("Config inizializzato.");
    }
}

// La prima volta che accedo a Config, il costruttore statico viene eseguito
Console.WriteLine(Config.ConnectionString);
```

Nota:

Non è possibile chiamare manualmente un costruttore statico o passargli parametri.

SEZIONE 2 — NULLABLE TYPES

Slide 7 – Perché servono i Nullable

I tipi valore (come `int`, `double`, `bool`) **non possono essere null**.

Per rappresentare l'assenza di un valore, C# fornisce la versione *nullable* di ogni tipo valore, indicata con `?`.

```
int? eta = null; // 'eta' può essere null o avere un valore intero
eta = 25;
Console.WriteLine(eta.HasValue); // True
Console.WriteLine(eta.Value);    // 25
```

Spiegazione:

`int?` è un `Nullable<int>`, che contiene o un valore o null.

È molto utile per dati provenienti da database o input opzionali.

Slide 8 – Operatore Null-Coalescing `??`

Serve a fornire un valore di fallback nel caso una variabile sia null.

```
string? nome = null;
string messaggio = nome ?? "Anonimo";
```

```
Console.WriteLine(messaggio); // Output: Anonimo
```

Spiegazione:

L'espressione `A ?? B` restituisce `A` se non è null, altrimenti `B`.

Slide 9 – Operatore Null-Conditional `?.` e `??=`

Due operatori che semplificano il controllo dei null.

```
class Persona
{
    public string? Nome { get; set; }
    public Indirizzo? DatiIndirizzo { get; set; }
}

class Indirizzo
{
    public string Citta { get; set; } = "Roma";
}

var p = new Persona();

// Uso di ?. → evita NullReferenceException
Console.WriteLine(p.DatiIndirizzo?.Citta ?? "Città non disponibile");

// Uso di ??= → assegna un valore solo se null
p.Nome ??= "Sconosciuto";
Console.WriteLine(p.Nome);
```

Spiegazione:

Questi operatori rendono il codice più sicuro e leggibile, eliminando i controlli `if (x != null)` ripetuti.

SEZIONE 3 — GENERICS

Slide 10 – Perché usare i Generics

I **Generics** consentono di scrivere classi, metodi e interfacce che lavorano con **tipi generici** invece di uno specifico.

Questo permette di creare codice **riutilizzabile e sicuro a livello di tipo**.

```
class Box<T>
{
    public T Contenuto;

    public void Mostra()
    {
        Console.WriteLine($"Contiene: {Contenuto}");
    }
}

var boxInt = new Box<int> { Contenuto = 42 };
var boxString = new Box<string> { Contenuto = "Ciao" };

boxInt.Mostra();    // Contiene: 42
boxString.Mostra(); // Contiene: Ciao
```

Spiegazione:

`T` è un segnaposto per un tipo qualsiasi.

Quando creiamo un `Box<int>`, `T` diventa `int`. Con `Box<string>`, diventa `string`.

Slide 11 – Metodi Generici

Anche i metodi possono essere generici, indipendentemente dal tipo della classe.

```
class Util
{
    public static void Scambia<T>(ref T a, ref T b)
    {
        T temp = a;
        a = b;
        b = temp;
    }
}
```

```
int x = 10, y = 20;
Util.Scambia(ref x, ref y);
Console.WriteLine($"{x}, {y}"); // Output: 20, 10
```

Spiegazione:

Il metodo `Scambia<T>` funziona con qualsiasi tipo (`int`, `string`, `DateTime`, ecc.) senza duplicare codice.

Slide 12 – Vincoli sui Generics (`where`)

Per limitare i tipi accettati da un generico si usano i **vincoli**.

```
class Repository<T> where T : IEntity, new()
{
    public T Crea()
    {
        return new T(); // possibile perché abbiamo 'new()'
    }
}

interface IEntity { int Id { get; set; } }

class Utente : IEntity { public int Id { get; set; } }

var repo = new Repository<Utente>();
Utente u = repo.Crea();
```

Spiegazione:

`where T : IEntity, new()` significa:

- `T` deve implementare `IEntity`;
- deve avere un costruttore pubblico senza parametri.

Slide 13 – Vantaggi dei Generics

1. **Riutilizzabilità** – un solo codice per tutti i tipi.
2. **Performance** – evita il *boxing/unboxing* tipico di `object`.

3. **Sicurezza di tipo** – errori di cast intercettati dal compilatore.
4. **Migliore leggibilità** – esplicita il tipo con cui si lavora.

```
List<int> numeri = new List<int> { 1, 2, 3 };
// Non serve il cast e non si può inserire un tipo sbagliato.
```

Slide 14 – Conclusione

Abbiamo esplorato tre concetti chiave:

- **Costruttori**: per inizializzare correttamente gli oggetti.
- **Nullables**: per gestire i valori mancanti in sicurezza.
- **Generics**: per creare codice riutilizzabile e tipizzato.

Questi elementi sono le fondamenta del C# intermedio e tornano in ogni architettura più complessa (Repository Pattern, DI, TDD, ecc.).