.NET Conf 2022

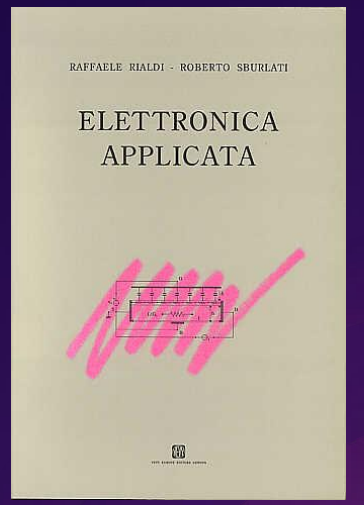
DotNet Liguria

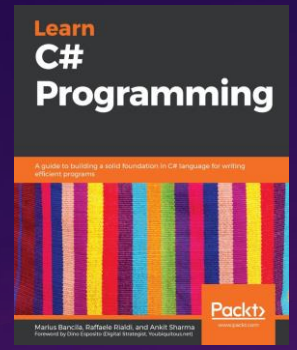
Le novità di .NET 7 e C# 11

Ing. Raffaele Rialdi, Senior Software Architect

Chi sono

- Laurea magistrale in Ingegneria Elettronica
- Insegnante all'Università di Ingegneria Informatica a Genova
- Senior Software Architect
- Consulenze di architettura e sviluppo software
  - Manufacturing, racing, healthcare, financial, …
- Speaker, Trainer e Autore di libri (fisica elettronica e C#)
  - Italy, Romania, Bulgaria, Russia, USA, …
- Orgoglioso membro della grande famiglia dei Microsoft MVP dal 2003





C# 11

CR e LF nelle stringhe interpolate

- Le stringhe interpolate sono espressioni come:

```
var str = $"Pi={Math.PI}";
```

- È ora possibile andare "a capo" dentro le graffe { ... }

```
var upperCaseInLoremIpsum = $"{string.Join(", ",
    loremIpsum
    .Where(c => Char.IsUpper(c)))}";
```

Raw String Literals

- Da C#11 è possibile dichiarare stringhe il cui contenuto ha graffe e 'a capo', senza però rinunciare all'interpolazione
  - Il numero di **"** minimo è 3
  - Visual Studio mostra una linea verticale per mostrare il margine sx
  - Basta mettere un numero superiore di **"** a quelle da visualizzare
  - Il numero di **$** specifica le parentesi usate per interpolare

```
string raw1 = """
        a
        b
        c
        """;
```

```
Console.WriteLine(
    """
        string raw1 = """
            a
            b
            c
            """;
    """);
```

```
var raw2 = $$"""Coordinates: {{{x}}, {{y}}}""";
```

UTF-8 String Literals

- È ora possibile dichiarare stringhe UTF-8
  - Attenzione che si parla solo dei literal, non di un nuovo tipo stringa

```
ReadOnlySpan<byte> s1 = "hello"u8;
```

- La dimensione della sequenza di byte dipende dai caratteri!

```
var multi1 = "€"u8;
Debug.Assert(multi1.Length == 3);
Debug.Assert(multi1.SequenceEqual(new byte[] { 0xe2, 0x82, 0xac }));
```

- In memoria le stringhe UTF-8 vengono *null-terminated* per poterle usare nella interoperabilità

List patterns

- I pattern sono stati estesi alle liste

```
var arr1 = new int[] { 1, 2, 3, 4, 5 }
new int[] { 1, 2, 3, 4, 5 } is [1, 2, 3, 4, 5]  => True
new int[] { 1, 2, 3, 4, 5 } is [1, 2, 3, 4]     => False
new int[] { 1, 2, 3, 4, 5 } is [1, _, 3, 4, _]  => True
new int[] { 1, 2, 3, 4, 5 } is [1, ..]          => True
new int[] { 1, 2, 3, 4, 5 } is [.., 5]          => True
new int[] { 1, 2, 3, 4, 5 } is [1, .., 5]       => True
new int[] { 1, 2, 3, 4, 5 } is [_, >= 2, ..]    => True
new int[] { } is [..]                           => True
if(arr1 is [1, 2, .. var rest]) Dump(rest); ➔ [3, 4, 5]
```

Oggetti: inizializzazione e '`required`'

- La fase di inizializzazione di un oggetto era già stata estesa

```
public class Point
{
    public double X { get; init; }
    public double Y { get; init; }
}
```

```
var p2 = new Point { X = 1 };
```

- Ma ora possiamo aggiungere

```
public required double Y { get; init; }
```

- L'attributo `[SetRequiredMembers]` sui costruttori indica che tutte le validazioni di '`required`' sono state soddisfatte

```
[SetsRequiredMembers]
public Point(string _) { /* ... */ }
```

Visibilità dei tipi ridotta ad un singolo file

- È possibile specificare che un Type sia visibile solo all'interno del file in cui è dichiarato

```
file class SomeType
{
    public string Name { get; set; } = string.Empty;
}
```

- Il Type SomeType non è visibile al di fuori di quel file

Generic Math

- Generic Math consiste nella possibilità di aggiungere gli operatori matematici nei contratti delle interfacce
  - Storicamente in C# gli operatori sono statici e non si può cambiare
- C# ha introdotto i membri statici virtuali per poter supportare questi casi

```
T Add<T>(T x, T y) where T : INumber<T> => x + y;
```

- Il lavoro in Generic Math è monumentale perché è servito a definire una lunghissima lista di interfacce per catalogare le operazioni eseguibili sui numeri

Che altro offre C#11?

- Auto-default structs
- Cache Delegates For Method Groups
- Checked User Defined Operators
- Extended nameof scope
- File Local Types
- Generic Attributes
- Generic Math
- Interface Static Virtual Members
- List patterns
- Newlines in string interpolation expressions
- Numeric IntPtr
- Object Initialization
- Raw string literals
- ref fields and scoped ref
- Relaxed Shift Operators
- Required members
- Span Char Patterrn Matching
- Struct Initialization
- Unsigned Right-Shift Operator
- UTF-8 string literals

- Argomenti trattati oggi
- Argomenti inerenti le performance
- Argomenti necessari a Generic Math
- Altro

.NET 7

Performance ancora migliori

- Il totale del lavoro è "riassunto" in 232 pagine di PDF
  - https://devblogs.microsoft.com/dotnet/performance_improvements_in_net_7/
- Molto degni di nota:
  - Garbage Collector: Regions (~4MB ciascuna) al posto dei Segments
  - Native AOT, OSR e PGO
  - MethodInfo.Invoke drasticamente più veloce
  - ThreadPool (ottimizzazioni varie)
  - Parse dei tipi base
  - Ricerca in stringhe con Span e Regular Expressions
  - Max e Min in Linq drasticamente più veloce (~30x)
  - Molto altro!.

Qualcuno ricorda la compilazione "Tiered"?

- Fin da .NET Core 3.0 la compilazione tiered è ON di default
  - Tier 0: codice compilato veloce ma poco ottimizzato
    - Se AOT ReadyToRun è ON, il Tier 0 carica il codice AOT
  - Tier 1: codice compilato lento ma molto ottimizzato
    - Se una "call" viene eseguita più volte (30) è considerata "hot" e ricompilata
  - Problema: i loop eseguono spesso del codice ma non sono call separate e sono "hot". Lo stesso vale per il codice "inlined"

OSR e PGO in .NET 7

- OSR (On Stack Replacement)
  - Permette la ricompilazione del Tier 1 durante l'esecuzione, sostituendo lo stack
  - OSR è attiva di default

- Dynamic PGO (Profile Guided Optimization)
  - Esegue la profilazione a runtime dell'esecuzione.
  - Le informazioni raccolte servono ad ottimizzare la successiva ricompilazione del codice (usando OSR e la Tiered compilation)
  - PGO si abilita con <TieredPGO>true</TieredPGO>

AOT: Ahead Of Time Compilation

- Consiste nella compilazione di tutto il codice (CLR incluso) in assembly nativo
  - L'eseguibile prodotto è un file PE (Windows) o ELF (Linux) nativo
  - Il compilatore JIT non entra più in gioco
  - Non ci sono più istruzioni IL dentro il file binario
  - Zero dipendenze esterne (a parte il runtime nativo libc)
  - È l'evoluzione di .NET Native
  - Le performance di esecuzione sono sempre uguali e molto veloci
- Durante la compilazione il codice IL inutilizzato viene rimosso ("IL Trimming")
  - Questo riduce in modo significativo la dimensione totale del file eseguibile
- Problemi di AOT
  - Non è possibile eseguire caricamento dinamico di Assembly
  - Non è possibile fare generazione dinamica d codice (Expression Trees incluse)

AOT all'opera

- Come si abilita?
  - File csproj: <PublishAot>true</PublishAot>
  - Deployment: dotnet publish -r win-x64 -c Release

- Come fare senza generazione dinamica di codice?
- Ecco perché Microsoft stia puntando sui C# Code Generator
  - Generazione di codice durante la compilazione
  - Non risolvono tutti i casi ma certamente molti

C# Code Generators all'opera

- I C# Code Generators non sono nuovi di .NET 7
  - Come esempio, fate riferimento al generatore di codice di INotifyPropertyChanged che ho pubblicato su nuget
  - https://github.com/raffaeler/SpeedyGenerators

- In .NET 7 I generatori nella BCL sono già presenti:
  - Codice di serializzazione / deserializzazione JSON System.Text.Json
  - Codice di Interoperabilità PInvoke
  - Regular Expressions
  - Interoperabilità con Javascript (Blazor e WebAssembly).

Domande al termine della giornata

Oppure sul canale Discord