.NET Conf 2025 - Genova

Le novità di C# 14 e .NET 10

Raffaele Rialdi

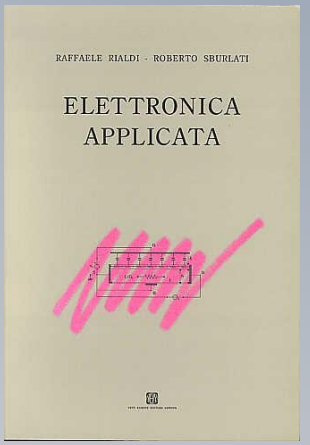
Senior Software Architect - Consultant

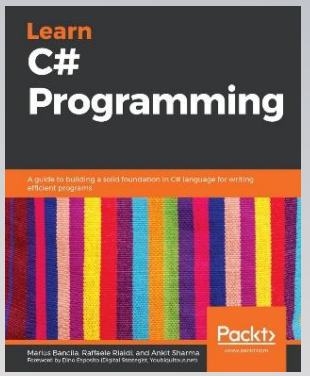
@raffaeler - raffaeler@vevy.com

https://www.linkedin.com/in/raffaelerialdi/

Chi sono

- Laurea Master in Ingegneria Elettronica (Unige)
  - Insegna saltuariamente a Ingegneria Informatica (Unige)
  - Membro della commissione ICT dell'Ordine degli Ingegneri di Genova
- Libero professionista, Software Architect Consultant in diverse aree:
  - Financial, Manufacturing, Healthcare, F1 racing, ...
- Speaker in conferenze nel mondo (più di 200 interventi in 20 anni)
  - Europa, Asia, USA
- Co-Autore di "Elettronica Applicata" e "C# Programming"
- Presidente di DotNetLiguria, community attiva a Genova e in Liguria
- Microsoft Most Valuable Professional per 21 anni consecutivi





.NET 10 Runtime

Estensione al supporto ufficiale delle STS di .NET

- Long Term Support (LTS): sono le versioni pari come .NET 10
  - Sono supportate per 3 anni
- Standard Term Support (STS): sono le versioni dispari come .NET 9
  - A partire dalla versione 9 sono supportate per 2 anni
  - Rende possibile adottarle senza differenze nella durata del supporto

Versione	Supporto	Data fine supporto	Note
.NET 6	LTS	12 Novembre 2024	Fuori supporto
.NET 7	STS	14 Maggio 2024	Fuori supporto, durata inferiore
.NET 8	LTS	10 Novembre 2026	Attuale
.NET 9	STS	10 Novembre 2026	Attuale
.NET 10	LTS	14 Novembre 2028	Attuale
Windows Server 2025	—	14 Novembre 2034	Contiene .NET Framework 3.5

Un ripasso veloce sul Garbage Collector

- **Workstation GC** (default in .NET Core per le app non-hosted)
  - Gira nello stesso thread/priorità che ha scatenato il GC
- **Server GC** (default per le app come ASP.NET Core)
  - Gira su più thread concorrenti alla massima priorità
  - GCSettings.IsServerGC

```
<PropertyGroup>
  <ServerGarbageCollection>true
  </ServerGarbageCollection>
</PropertyGroup>
```

```
DOTNET_gcServer=1
```

Dynamic Adaptation To Application Sizes (DATAS)

- È stato già introdotto in .NET 9 (STS) per default
- È solo disponibile nel Server GC
  - Si (dis)abilita con la variabile DOTNET_GCDynamicAdaptationMode
- Adatta dinamicamente la dimensione dell'Heap a seconda delle allocazioni eseguite dall'applicazione
  - In passato la dimensione era tendenzialmente quasi sempre in crescita
- Scenari dove DATAS è utile in applicazioni/container che:
  1. consumano poca memoria ma hanno picchi improvvisi
  2. tollerano un piccolo rallentamento usato per disallocare
  3. necessitano di liberare memoria a vantaggio di altri (thread o processi).

Perché DATAS è rilevante in .NET 10?

- Scenario 👍 ASP.NET Core in .NET 10 dopo un picco di utilizzo di RAM
  1. Dopo un picco di utenti, Kestrel Web Server proattivamente rilascia la memoria
  2. DATAS adatta la dimensione dello heap, liberando al più presto la memoria
     - Il working set può scendere anche del 93%
  3. Il container dell'applicazione occupa meno memoria
     - Molto importante per evitare il *Linux Out-of-Memory (OOM) che termina i processi*
  4. La VM del container rende disponibile la memoria all'Host
     - Va configurato il **Memory Ballooning**. Disponibile in KVM/Proxmox, VMWare e in Cloud
     - È una feature per cui la VM può rilasciare memoria a vantaggio di altre VM
- Scenario 👎 ASP.NET Core .NET 10 con preferenza alla performance
  - Disabilitare DATAS prediligendo la performance all'occupazione di RAM
- Emblematico di come le nuove architetture cadano nel primo scenario.

Breaking Change: Niente più SIGTERM di default

- Breaking Change: Il CLR non fornisce più gli handler di default per:
  - CTRL_SHUTDOWN_EVENT e CTRL_CLOSE_EVENT su Windows
  - SIGTERM e SIGHUP nei sistemi *nix
- Conseguenza
  - Gli eventi di AppDomain.ProcessExit e AssemblyLoadContext.Unloading non sono più chiamati
- Motivi
  - Insufficienti/inadeguati per applicazioni console, container e servizi Windows
- Rimedio
  - Utilizzare HostingHostBuilderExtensions.UseConsoleLifetime (default in ASP.NET)
  - Handler manuali:
    https://learn.microsoft.com/en-us/dotnet/core/compatibility/core-libraries/10.0/sigterm-signal-handler

Altre novità interessanti

- Nuovo formato SLNX per le Solution
  - È il default per "dotnet new sln"
- Crittografia
  - Post-quantum crypto APIs
  - Disponibilità di TLS 1.3
- Nuove conversioni da numero Hex
  - Convert.FromHexString
- Nuovo WebSocketStream
  - Abilita l'uso di stream via WebSocket
- Integrato il progetto community
  - System.Linq.AsyncEnumerable

- Nuovo ordinamento numerico

```
StringComparer sc =
    StringComparer.Create(
        CultureInfo.CurrentCulture,
        CompareOptions.NumericOrdering);
Console.WriteLine(sc.Equals("02", "2"));
// Output: True


var versions =
     new[] { "V 1.3", "V 1.10", "V 2.0" }
    .Order(sc);
foreach (string ver in versions)
{
    Console.WriteLine(ver);
}
// Output:
// V 1.3
// V 1.10
// V 2.0
```

Performance

JIT: ottimizzazioni sull'Enumeratore

```
foreach (var n in numbers)
{
    sum += n;
}
```

Method	Runtime	Mean	Ratio	Code Size	Allocated	Alloc Ratio
PerfArray	.NET 10.0	41.12 ns	0.86	34 B	-	NA
PerfArray	.NET 9.0	39.07 ns	0.81	34 B	-	NA
PerfArray	.NET Framework 4.8	48.17 ns	1.00	52 B	-	NA
PerfList	.NET 10.0	152.00 ns	0.19	269 B	-	0.00
PerfList	.NET 9.0	256.60 ns	0.32	411 B	40 B	1.00
PerfList	.NET Framework 4.8	796.63 ns	1.00	212 B	40 B	1.00
PerfQueue	.NET 10.0	154.82 ns	0.15	302 B	-	0.00
PerfQueue	.NET 9.0	410.96 ns	0.40	549 B	40 B	1.00
PerfQueue	.NET Framework 4.8	1,034.38 ns	1.00	212 B	40 B	1.00
PerfStack	.NET 10.0	72.05 ns	0.07	256 B	-	0.00
PerfStack	.NET 9.0	378.90 ns	0.39	639 B	40 B	1.00
PerfStack	.NET Framework 4.8	983.96 ns	1.00	212 B	40 B	1.00
PerfDictionary	.NET 10.0	118.35 ns	0.12	323 B	-	0.00
PerfDictionary	.NET 9.0	340.94 ns	0.35	449 B	40 B	1.00
PerfDictionary	.NET Framework 4.8	972.96 ns	1.00	212 B	40 B	1.00

JIT: allocazioni eseguite sullo stack invece dell'heap

Method	Runtime	Mean	Ratio	Code Size	Allocated	Alloc Ratio
IntArray	.NET 10.0	6.8047 ns	0.12	95 B	-	0.00
IntArray	.NET 9.0	15.6022 ns	0.28	468 B	40 B	0.56
IntArray	.NET Framework 4.8	56.2670 ns	1.00	316 B	72 B	1.00
StopWatchObj	.NET 10.0	0.0227 ns	0.001	3 B	-	0.00
StopWatchObj	.NET 9.0	18.3711 ns	0.975	345 B	40 B	1.00
StopWatchObj	.NET Framework 4.8	18.9289 ns	1.004	346 B	40 B	1.00
PointClass	.NET 10.0	1.0283 ns	0.273	6 B	-	0.00
PointClass	.NET 9.0	0.0196 ns	0.005	6 B	-	0.00
PointClass	.NET Framework 4.8	3.7884 ns	1.006	41 B	24 B	1.00
Persons	.NET 10.0	0.0026 ns	0.000	6 B	-	0.00
Persons	.NET 9.0	0.0363 ns	0.002	6 B	-	0.00
Persons	.NET Framework 4.8	18.0991 ns	1.000	152 B	64 B	1.00
ForEachPersons	.NET 10.0	20.4099 ns	0.60	222 B	64 B	0.62
ForEachPersons	.NET 9.0	31.2960 ns	0.91	184 B	104 B	1.00
ForEachPersons	.NET Framework 4.8	34.2549 ns	1.00	221 B	104 B	1.00
PersonsList	.NET 10.0	41.1831 ns	1.15	514 B	136 B	0.94
PersonsList	.NET 9.0	46.9043 ns	1.30	563 B	136 B	0.94
PersonsList	.NET Framework 4.8	37.3750 ns	1.04	288 B	144 B	1.00
IntArraySum	.NET 10.0	4.2722 ns	0.50	111 B	-	0.00
IntArraySum	.NET 9.0	12.9106 ns	1.51	82 B	48 B	1.00
IntArraySum	.NET Framework 4.8	8.6489 ns	1.01	88 B	48 B	1.00
StringArraySumLength	.NET 10.0	2.9564 ns	0.16	122 B	-	0.00
StringArraySumLength	.NET 9.0	12.6118 ns	0.69	100 B	48 B	1.00
StringArraySumLength	.NET Framework 4.8	18.9330 ns	1.03	135 B	48 B	1.00

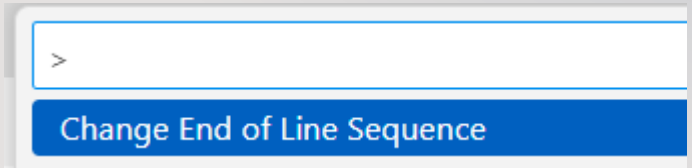
File-based apps

Usare C# come linguaggio di scripting

- È sufficiente scrivere un file .cs ed avviarlo con la CLI di dotnet
- Il compilatore C# supporta la direttiva **#:** che **ignora** ciò che segue
  - La CLI di dotnet riconosce i parametri e li usa per la compilazione

```
#:package Colorful.Console@1.2.15
#:property TargetFramework=net10.0
#:property Nullable=enable
```

- La CLI dotnet tratta il file .cs come fosse un progetto
  - dotnet publish helloworld.cs -o . -r win-x64 -p:PublishSingleFile=true
- Su Linux è possibile usare C# come bash script
  1. Prima riga: #!/usr/bin/env dotnet
  2. Usare LF come per gli a-capo dei file
  3. chmod +x helloworld.cs



VSCode "F1"

Change End of Line Sequence

Le novità di C# 14

Extension members

C#14 va oltre gli "Extension Methods" includendo tutti i membri

```
public static class PointExtensionsOld
{
    public static string ToStringEx(this Point p) => $"Point(X: {p.X}, Y: {p.Y})";
}

public static class PointExtensionsNew
{
    extension(Point p)
    {
        public string ToStringEx() => $"Point(X: {p.X}, Y: {p.Y})";
        public Point Neg => new Point(-p.X, -p.Y);
        public static Point operator +(Point left, Point right)
            => new Point(left.X + right.X, left.Y + right.Y);
    }
}
```

Field-backed properties

```
public class Person
{
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    public string Name
    {
        get => field;
        set => field = value ?? throw new ArgumentNullException(nameof(value));
    }

    public int Age
    {
        get;
        set => field = value;
    }
}
```

Nuove conversioni implicite per Span<T>

```
string[] strings = { "1", "2", "3", "4", "5" };

//                              C# 13                   C# 14
X.M(strings);                 // IEnumerable<T>          ReadOnlySpan<T>
X.M(strings.AsSpan());        // Span<T>                 Span<T>
X.M(strings.AsReadOnly());    // IEnumerable<T>          IEnumerable<T>


static class X
{
    public static void M<T>(IEnumerable<T> s) => Console.WriteLine("IEnumerable<T>");

    public static void M<T>(Span<T> s) => Console.WriteLine("Span<T>");

    // Ignorato prima di C# 14
    public static void M<T>(ReadOnlySpan<T> s)=>Console.WriteLine("ReadOnlySpan<T>");
}
```

Utilizzo di **nameof** sui tipi *unbounded*

- I tipi unbounded sono quelli privi dell'argomento generico
  - List<>
  - Dictionary<,>
  - …
- Da C# 14 è possibile utilizzare `nameof(List<>)`
- Tutti i **nameof(…)** che seguono, restituiscono la stringa "List"

```
// Nuovo a partire da C# 14
Console.WriteLine($"nameof(List<>)       => {nameof(List<>)}");

// Funzionavano già prima
Console.WriteLine($"nameof(List<string>) => {nameof(List<string>)}");
Console.WriteLine($"nameof(List<int>)    => {nameof(List<int>)}");
```

Assegnazione condizionale dei tipi nullabili

```
Person? person = null;
UpdatePerson(person, 30);

public void UpdatePerson(Person? person, int age)
{
    // A partire da C# 14
    person?.Age = age;

    // Prima di C# 14
    if (person != null) person.Age = age;
}
```

Dichiarazione parziale dei membri

- Tipi e metodi parziali erano già disponibili da molte versioni
- C# 13 ha esteso a proprietà e indexers
- C# 14 completa la lista con costruttori ed eventi parziali

```
public partial class Alarm : BaseAlarm
{
    // dichiarazione del ctor parziale
    public partial Alarm(
       string msg, string location);
}

public partial class Alarm
{
    private string _message;

    // implementazione del ctor parziale
    public partial Alarm(string msg, string location)
        : base(location)  { _message = msg; }
}
```

```
public class BaseAlarm
{
    protected string _location;
    public BaseAlarm(string location)
    {
        _location = location;
    }
}
```

Ridefinire gli operatori composti

- Gli operatori composti sono +=, *=, etc.
- Prima di C# 14 non era possibile ridefinirli
  - Ridefinendo solo `operator +` viene creata ogni volta una nuova istanza

```
public class Age
{
    public Age(int years) => Years = years;
    public int Years { get; set; }

    // Questo operatore muta lo stato dell'istanza (nuovo in C# 14)
    public void operator +=(int years) => Years += years;

    // Questo operatore crea sempre una nuova istanza
    public static Age operator +(Age left, Age right)
        => new Age(left.Years + right.Years);
}
```

Cosa potremmo vedere in .NET 11 e C# 15?

- Discriminated Union Types
  - Già presenti in Typescript
  - Si sposano perfettamente con il pattern matching di C#
- Evoluzione del pattern matching
- Uso degli iteratori dentro le Lambda
- Nuova implementazione dei async/await (già disponibile come experimental in .NET 10)
  - Oggi il compilatore genera una macchina a stati
  - Domani potrebbe essere gestita dal runtime
  - https://github.com/dotnet/runtime/issues/109632
- Ne parleremo a DotNetConf 2026!

Domande?