




# Da 7 a 13: l'evoluzione di C# negli ultimi 8 anni

Nicola Paro

# Sponsor



# Tiobe Index 2024

Dec 2024	Dec 2023	Change	Programming Language		Ratings
1	1			Python	23.84%
2	3	^		C++	10.82%
3	4	^		Java	9.72%
4	2	v		C	9.10%
5	5			C#	4.87%
6	6			JavaScript	4.61%
7	13	^^		Go	2.17%
8	9	^		SQL	1.99%
9	8	v		Visual Basic	1.96%
10	12	^		Fortran	1.79%





# 2014 – Satya Nadella diventa CEO di Microsoft

- È **diventato CEO della società il 4 febbraio 2014**, succedendo a Steve Ballmer.
- La sua nomina ha segnato una svolta significativa per Microsoft, che all'epoca stava affrontando difficoltà legate alla **transizione verso** un mondo sempre più dominato dal **cloud computing e dai dispositivi mobili**.
- Principali modifiche e contributi di Nadella:
  - Focus sul Cloud Computing
  - Cultura Aziendale - **Apertura e Collaborazione**: cambiamento della cultura aziendale, promuovendo l'empatia, la **collaborazione e l'inclusività**. "Growth Mindset": Ha introdotto una **mentalità orientata alla crescita**, incoraggiando i dipendenti a sperimentare e innovare.
  - Riorientamento delle Priorità: Mobilità e Cross-Platform: **Dal modello Windows-centrico verso un approccio multiplatforma**.



# 2016 – .NET Core

- Lanciato nel 2016 come runtime multiplatforma.
- Motivazioni principali:
  - Adattarsi alle esigenze del cloud computing e dello sviluppo multiplatforma.
  - Promuovere una piattaforma open-source con supporto dalla community.
  - Offrire scalabilità e performance migliori.





# C# 7

## Rilasci

- Versione 7.0
  - Marzo 2017
- Versione 7.1
  - Agosto 2017
- Versione 7.2
  - Novembre 2017
- Versione 7.3
  - Maggio 2018

# C# 7.0 – Value Tuples

Consente di raggruppare più valori insieme, come una coppia di stringhe (nome e cognome), senza la necessità di creare una classe o una struttura.

```
(string firstName, string lastName) person = ("John", "Doe");  
Console.WriteLine($"First Name: {person.firstName}, Last Name: {person.lastName}");
```

```
var person = (firstName: "John", lastName: "Doe");  
Console.WriteLine($"First Name: {person.firstName}, Last Name: {person.lastName}");
```

```
var person = ("John", "Doe");  
Console.WriteLine($"First Name: {person.Item1}, Last Name: {person.Item2}");
```

# C# 7.0 – Value Tuples

Attenzione alla sintassi! queste due istruzioni fanno cose diverse:

```
(string name, string surname) person = ("John", "Doe");
```

!=

```
(string name, string surname) = ("John", "Doe");
```

Possiamo anche usare le tuple per fare lo swap di due valori.

```
var a = 15;  
var b = 18;  
(a, b) = (b, a);
```

# C# 7.0 – Deconstructors

Il deconstructor è un metodo che permette di "estrarre" i valori di un oggetto in variabili separate, generalmente per facilitare l'assegnazione dei valori all'interno di una tupla o per una gestione semplificata dei dati.

```
public class Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y) { X = x; Y = y; }
    public void Deconstruct(out int x, out int y) { x = X; y = Y; }
}
```

# C# 7.0 – Deconstructors

Utilizzo di un deconstructor.

Il metodo Deconstruct è invocato implicitamente

```
Point point = new Point(10, 20);  
  
// Uso del deconstructor per decomporre l'oggetto in variabili.  
  
var (x, y) = point;  
  
Console.WriteLine($"X: {x}, Y: {y}");
```

# C# 7.0 – Out Variables

Le variabili **out** possono essere dichiarate direttamente nell'invocazione del metodo, semplificando il codice.

```
// old way
int result;
if (int.TryParse("123", out result))
{
    Console.WriteLine($"Parsed number: {result}");
}
```

```
// new way
if (int.TryParse("123", out int result))
{
    Console.WriteLine($"Parsed number: {result}");
}

// or

if (int.TryParse("123", out var result))
{
    Console.WriteLine($"Parsed number: {result}");
}
```



# C# 7.0 – Pattern Matching

```
public class MyIgnorantClass
{
    public override bool Equals(object obj)
    {
        return true;
    }

    public static operator ==(MyIgnorantClass a, MyIgnorantClass b)
    {
        return true;
    }
}

var a = new MyIgnorantClass();
Console.WriteLine(a == null); // prints True
Console.WriteLine(a.Equals(null)); // prints True
Console.WriteLine(a is null); // prints False
```

SUS.



# C# 7.0 – Pattern Matching

- Rispetto a **Equals** e **==**, **is** è più sicuro specialmente quando si lavora con oggetti di riferimento o quando si ha a che fare con valori nulli.
- **is** è focalizzato sul controllo del tipo e sul pattern matching, mentre gli altri sono utilizzati per confrontare l'uguaglianza o l'identità degli oggetti.
- Quando si utilizza **is**, non viene ricercato alcun overload di Equals per effettuare il match.

# C# 7.0 – Local Functions

Local Functions: Le funzioni locali sono metodi definiti all'interno di un altro metodo, migliorando l'incapsulamento e la leggibilità.

```
public void Bar()
{
    void Bar1()
    {
        void Bar2()
        {
            void Bar3()
            {
                Console.WriteLine("Hello, World!");
            }
            Bar3();
        }
        Bar2();
    }
    Bar1();
}
```



# C# 7.0 – Ref Returns e Ref Locals

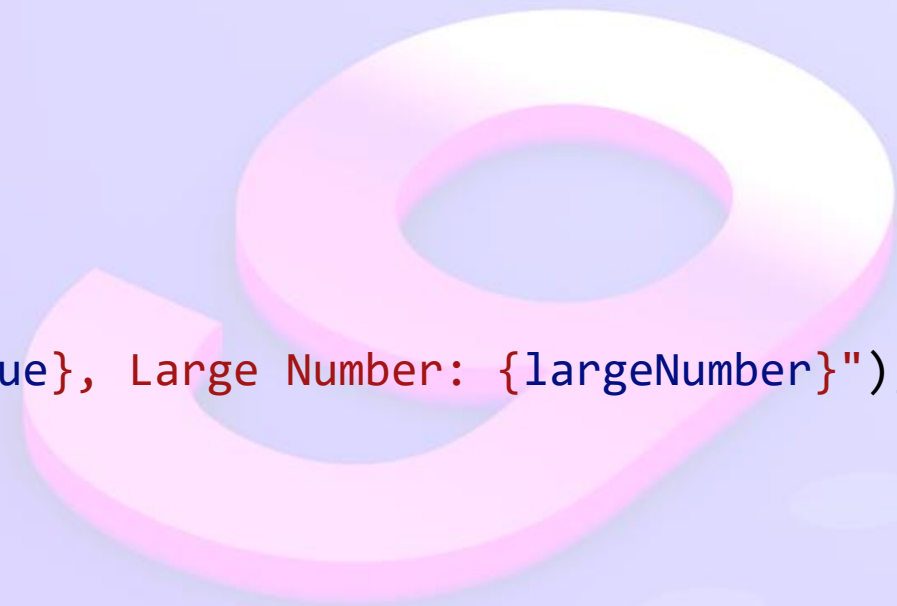
I metodi possono restituire riferimenti a variabili locali, migliorando le prestazioni.

```
ref int FindElement(int[] array, int index) => ref array[index];  
int[] numbers = { 10, 20, 30 };  
ref int element = ref FindElement(numbers, 1);  
element = 50;  
Console.WriteLine(numbers[1]); // Outputs: 50
```

# C# 7.0 – Literal Improvements

I numeri possono essere rappresentati in modo più leggibile usando separatori di cifre e numeri binari.

```
int binaryValue = 0b1010_1010;  
int largeNumber = 1_000_000;  
Console.WriteLine($"Binary: {binaryValue}, Large Number: {largeNumber}");
```



# C# 7.1

## Async Main method

- Il punto di ingresso per un'applicazione può includere il modificatore async.

## Default literal expressions

- È possibile utilizzare espressioni letterali predefinite (default) come valori predefiniti quando il tipo di destinazione può essere dedotto.

## Inferred tuple element names

- I nomi degli elementi delle tuple possono essere dedotti automaticamente durante l'inizializzazione della tupla in molti casi.

## Pattern matching on generic type parameters

- È possibile utilizzare espressioni di pattern matching su variabili il cui tipo è un parametro generico.



# C# 7.2

## Initializers on stackalloc arrays

- Permettono di inizializzare gli array allocati con stackalloc direttamente al momento della creazione.

## Use fixed statements with any type that supports a pattern

- Le istruzioni fixed possono essere utilizzate con qualsiasi tipo che supporta i pattern, consentendo l'accesso diretto alla memoria non gestita.

## Access fixed fields without pinning

- È possibile accedere ai campi fixed senza la necessità di utilizzare l'istruzione fixed per fissare la variabile.

## Reassign ref local variables

- Le variabili locali di tipo ref possono essere riassegnate a nuovi riferimenti.

## Declare readonly struct types:

- Indica che una struttura è immutabile e deve essere passata come parametro in ai suoi metodi membri.

## Add the in modifier on parameters:

- Specifica che un argomento viene passato per riferimento ma non modificato dal metodo chiamato.

## Use the ref readonly modifier on method returns:

- Indica che un metodo restituisce il suo valore per riferimento ma non consente scritture su quell'oggetto.

## Declare ref struct types

- Indica che un tipo di struttura accede direttamente alla memoria gestita e deve sempre essere allocato sullo stack.

## Use more generic constraints

- Sono disponibili vincoli generici più flessibili ed espressivi.

## Non-trailing named arguments

- Gli argomenti posizionali possono seguire gli argomenti denominati.

## Leading underscores in numeric literals

- I letterali numerici possono ora avere underscore iniziali prima di qualsiasi cifra stampata.

## private protected access modifier

- Consente l'accesso alle classi derivate nello stesso assembly.

## Conditional ref expressions

- Il risultato di un'espressione condizionale (?:) può ora essere un riferimento.

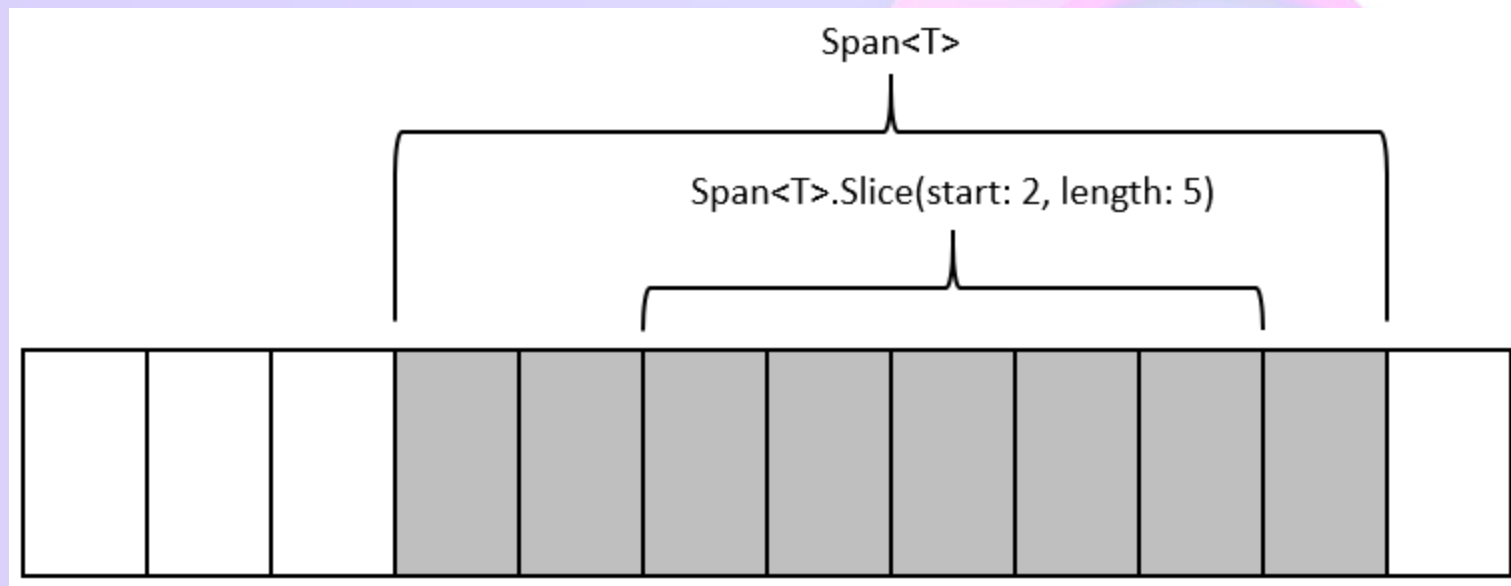
# C# 7.2 – Span<T>


Rappresenta una "finestra" su un array, un buffer o una porzione di memoria contigua senza copiare i dati.

Consente operazioni di slicing efficienti su stringhe, array, stackalloc e puntatori.

Può essere utilizzato solo nello stack, quindi non può essere memorizzato nei campi delle classi.

Ottimale per operazioni ad alte prestazioni in contesti come parsing, elaborazione di buffer e manipolazione di stringhe.





# C# 8



Rilasciato a Settembre  
2019

# C# 8 – Nullable Reference Types

I tipi di riferimento possono essere dichiarati come nullable, riducendo gli errori legati ai riferimenti nulli.

Argomento nullable reference type.  
Warning se invoco un metodo perché il valore potrebbe essere null.

```
public class Foo
{
    public void Bar(string? name)
    {
        Console.WriteLine(name.ToUpper());
    }
}
```

Argomento Non-nullable reference type. Nessun problema nell'invocazione di un metodo

```
public class Foo
{
    public void Bar(string name)
    {
        Console.WriteLine(name.ToUpper());
    }
}
```

Argomento nullable reference type.  
Nessun problema nell'invocazione di un metodo perché controllo prima se la variabile è null o meno

```
public void Bar(string? name)
{
    if (name is not null)
    {
        Console.WriteLine(name.ToUpper());
    }
}
```

# C# 8 – Nullable Reference Types

I tipi di riferimento possono essere dichiarati come nullable, riducendo gli errori legati ai riferimenti nulli.

Variabile non-nullable reference type. Warning perché sto assegnando il valore null.

```
public void Bar()
{
    string value = null;

    Console.WriteLine(value);
}
```

Variabile nullable reference type. E' corretto poter assegnare null a questa variabile.

```
public void Bar()
{
    string? value = null;

    Console.WriteLine(value);
}
```

Variabile non-nullable reference type. Posso comunque forzare il null per quella variabile utilizzando il !

```
public void Bar()
{
    string value = null!;

    Console.WriteLine(value);
}
```

# C# 8 – Nullable Reference Types

- Nullable Reference Type è un'impostazione del progetto.

```
<PropertyGroup>
  <TargetFramework>net8.0</TargetFramework>
  <ImplicitUsings>enable</ImplicitUsings>
  <Nullable>enable</Nullable>
</PropertyGroup>
```

- I warning per non-nullable reference type non garantiscono che una variabile possa avere solamente valore non-null.
- Anche cambiando ad error la severity di queste segnalazioni, non è garantito comunque che non possano arrivare valori null.
- I metodi public devono validare che i parametri siano effettivamente non null anche se sono dichiarati non null.



# C# 8 – Async Enumerable

- Introduzione di `IAsyncEnumerable<T>` per iterare asincronamente su flussi di dati.
- Permette una gestione efficiente di flussi di dati asincroni, come la lettura da stream di rete.

```
async IAsyncEnumerable<int> GenerateSequence()  
{  
    for (int i = 0; i < 10; i++)  
    {  
        await Task.Delay(1000);  
        yield return i;  
    }  
}  
  
await foreach (var number in GenerateSequence())  
{  
    Console.WriteLine(number);  
}
```

# C# 8 – Async Enumerable

```
public class AsyncExample
{
    public async IAsyncEnumerable<int> GetNumbersAsync()
    {
        for (int i = 1; i <= 5; i++)
        {
            // Simulating async work
            await Task.Delay(1000);
            yield return i;
        }
    }
}
```

```
public class GetNumbersAsyncStateMachine : IAsyncEnumerable<int>, IAsyncEnumerator<int>
{
    ...

    public IAsyncEnumerator<int> GetAsyncEnumerator(CancellationTokens cancellationToken = default)
    {...}

    public int Current => _current;

    public async ValueTask<bool> MoveNextAsync()
    {
        switch (_state)
        {
            case 0:
                _counter = 1;
                _state = 1;
                goto case 1;
            case 1:
                if (_counter <= 5)
                {
                    await Task.Delay(1000);
                    _current = _counter++;
                    return true;
                }
                _state = -1;
                break;
        }
        return false;
    }

    public ValueTask DisposeAsync()
    {...}
}
```

# C# 8 – Index e Range

Nuovi operatori per accedere a elementi e sottosequenze di array e liste.

```
var array = new[] { 1, 2, 3, 4, 5 };  
var ultimoElemento = array[^1]; // 5  
var sottoArray = array[1..3]; // { 2, 3, 4 }
```

# C# 8 – Null Coalescing Assignment

Introduzione dell'operatore `??=`, che consente di assegnare un valore a una variabile solo se questa è attualmente null.

```
string? nome = null;  
nome ??= "Nome Predefinito";
```

# C# 8 – Stackalloc (senza unsafe)

Estensione dell'uso di **stackalloc** permettendone l'utilizzo in più contesti, migliorando le prestazioni per determinate allocazioni.

```
Span<int> numeri = stackalloc[] { 1, 2, 3, 4, 5 };
```

# C# 8 – Using Declarations

Nuova sintassi per gestire automaticamente il rilascio delle risorse non gestite.

```
using var stream = new FileStream("file.txt", FileMode.Open);  
// Use the stream  
// Automatically disposed at the end of the scope
```

```
void MyMethod()  
{  
    var stream = new FileStream(...);  
    try  
    {  
        // Use the stream  
    }  
    finally  
    {  
        stream.Dispose();  
    }  
}
```

```
void MyMethod()  
{  
    using (var stream = new FileStream(...))  
    {  
        // Use the stream  
    }  
}
```

```
void MyMethod()  
{  
    using var stream = new FileStream(...);  
    // Use the stream  
}
```



OH GOD WHY



# C# 8 – Default Interface Methods



- Possibilità di definire metodi con implementazioni predefinite nelle interfacce.
- Facilita l'evoluzione delle interfacce senza interrompere le implementazioni esistenti.

```
public interface ILogger
{
    void Log(string message)
    {
        Console.WriteLine(message);
    }
}
```

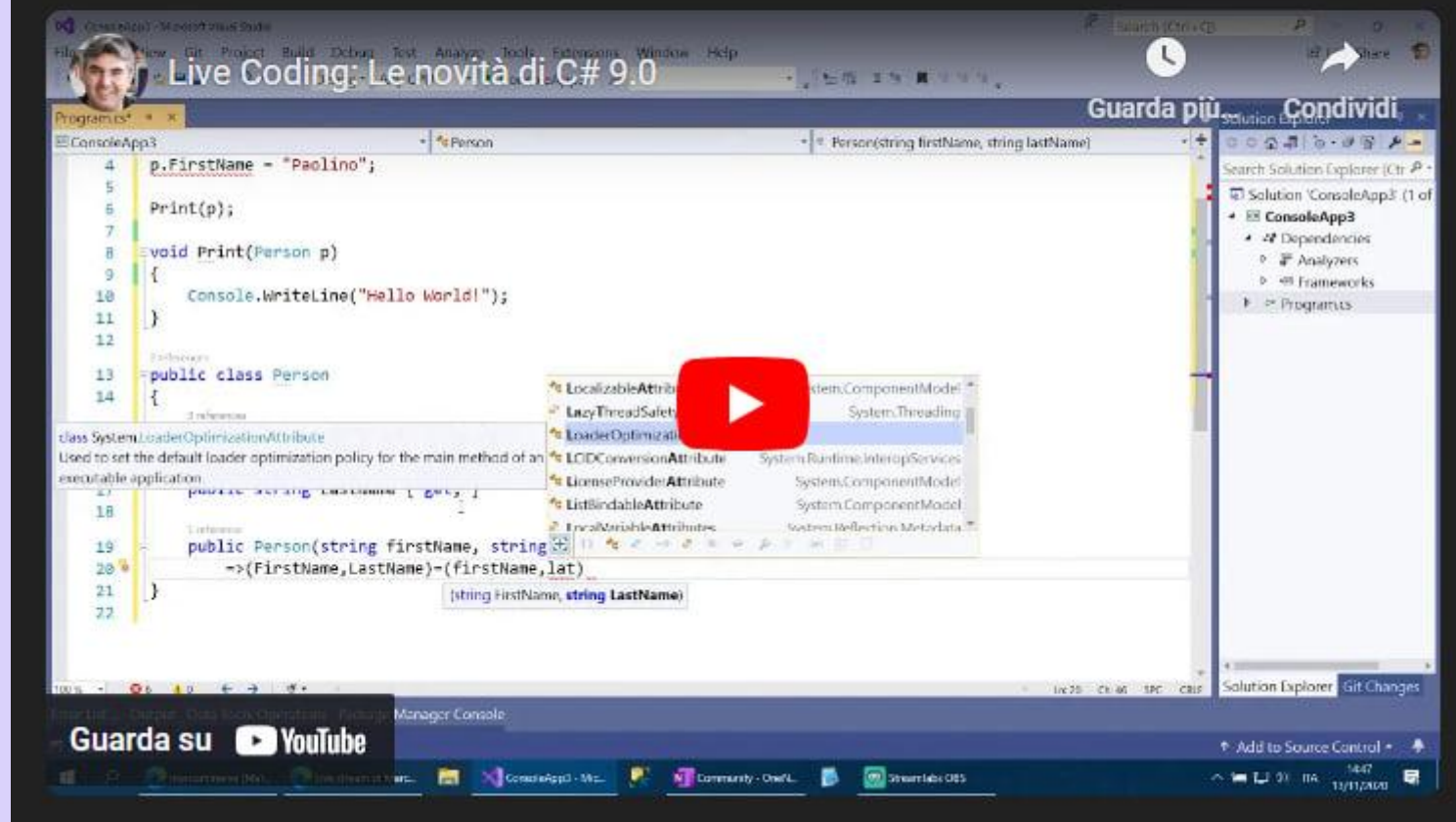
Rilasciato a Novembre 2020  
insieme a .NET 5

C# 9



Queste funzionalità rendono C# 9.0 un linguaggio più potente e flessibile, migliorando la produttività degli sviluppatori e la qualità del codice prodotto.

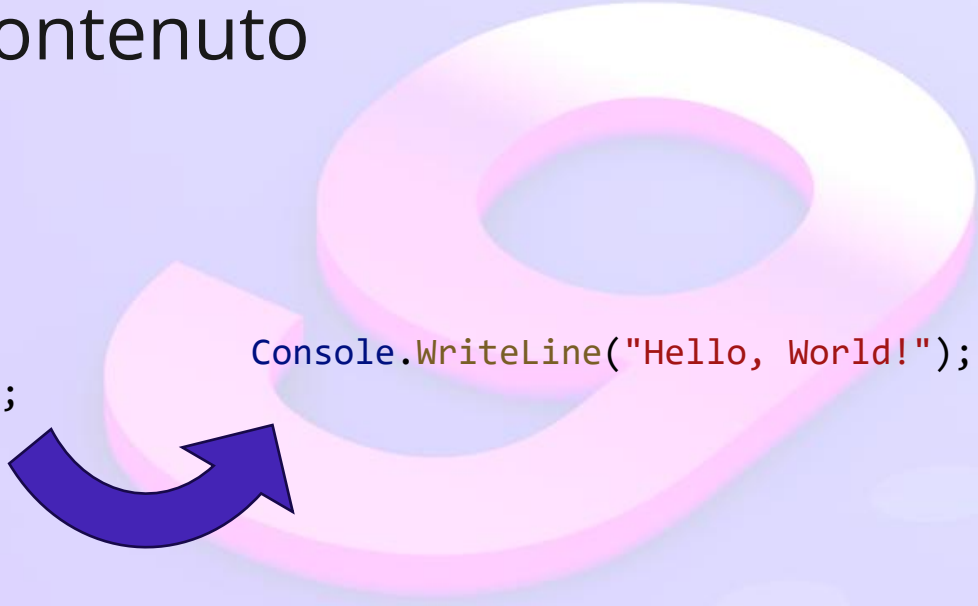
Per approfondire ulteriormente le novità di C# 9.0, puoi consultare il seguente video:



# C# 9 – Top level statements

- Riduzione del boilerplate code
- Non è più necessario Main(), basta creare un file Program.cs e scrivere lì dentro tutto il contenuto

```
public class Program
{
    public static void Main(string[] args)
    {
        Console.WriteLine("Hello, World!");
    }
}
```



```
Console.WriteLine("Hello, World!");
```

# C# 9 – Record Types

- Il nuovo tipo di riferimento che fornisce supporto integrato per dati immutabili, semplificando la creazione di classi orientate ai dati.

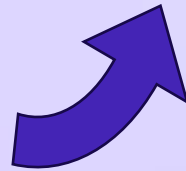
```
public class Person
{
    public string FirstName { get; }
    public string LastName { get; }
    public int Age { get; }

    public Person(string firstName, string lastName, int age)
    {
        FirstName = firstName;
        LastName = lastName;
        Age = age;
    }
    protected Person(Person original)
    {
        FirstName = original.FirstName;
        LastName = original.LastName;
        Age = original.Age;
    }

    // equals, gethashcode, tostring, etc.
    public override string ToString() => $"FirstName={FirstName} LastName={LastName} Age={Age}";
    public int GetHashCode() => FirstName.GetHashCode() ^ LastName.GetHashCode() ^ Age.GetHashCode();
    public override bool Equals(object obj)
    {
        if (obj is Person person)
        {
            return person.FirstName == FirstName && person.LastName == LastName && person.Age == Age;
        }
        return false;
    }

    public static bool operator ==(Person left, Person right) => left.Equals(right);
    public static bool operator !=(Person left, Person right) => !left.Equals(right);
}
```

```
public record Person(string FirstName, string LastName, int Age);
```



# C# 9 – init;

- Le proprietà con accessor init permettono l'assegnazione durante l'inizializzazione dell'oggetto, impedendo modifiche successive.

```
public class Libro
{
    public string Titolo { get; init; }
    public string Autore { get; init; }
}
```



# C# 9 – init;

```
var book = new Book
{
    Title = "The Hobbit",
    Author = "J.R.R. Tolkien"
};

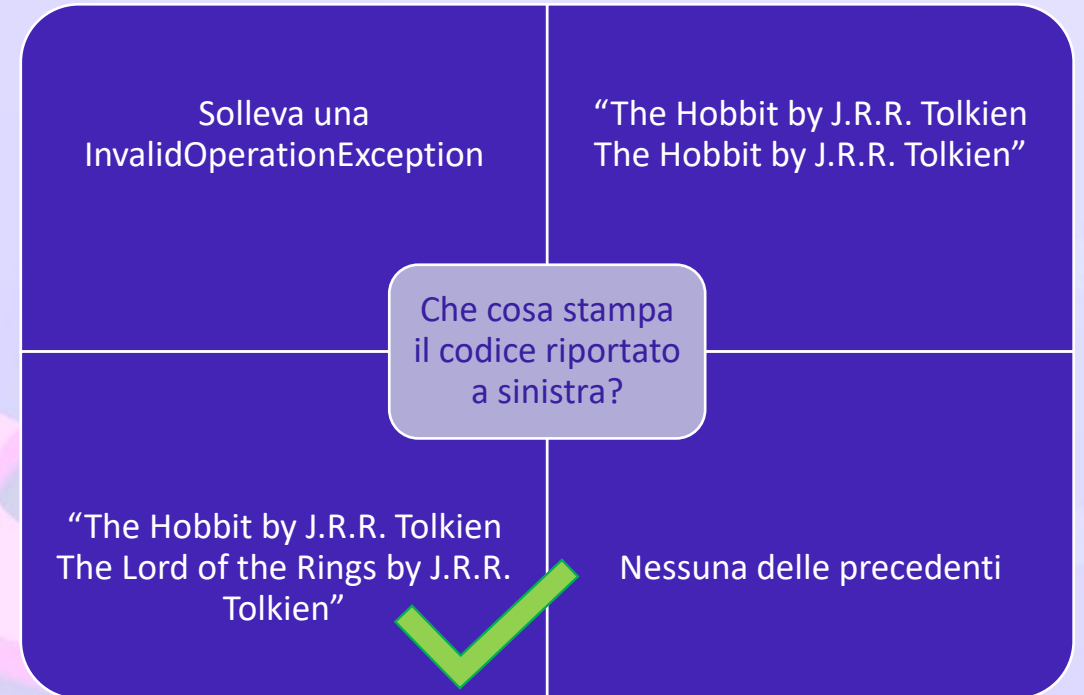
Console.WriteLine(book);

var titleProperty = typeof(Book)
    .GetProperty(nameof(Book.Title));
titleProperty.SetValue(book, "The Lord of the Rings");

Console.WriteLine(book);

public class Book
{
    public string Title { get; init; }
    public string Author { get; init; }

    public override string ToString()
        => $"{Title} by {Author}";
}
```





# C# 9 – with

Espressioni **with** per i Record permettono di creare una nuova istanza di un record modificando alcune proprietà, mantenendo l'immutabilità.

```
public record Persona(string Nome, string Cognome);  
  
var originale = new Persona("Mario", "Rossi");  
var aggiornato = originale with { Cognome = "Bianchi" };
```

# C# 9 – Pattern Matching

Introduzione di nuovi pattern come **or**, **and** e **not** per semplificare logiche condizionali complesse.

## Pattern Matching

In C# 7



In C# 9



```
public static void StampaRisultato(object obj)
{
    if (obj is int or string)
    {
        Console.WriteLine(
            $"L'oggetto è un int o una stringa: {obj}");
    }

    if (obj is int and > 10)
    {
        Console.WriteLine($"Intero maggiore di 10: {obj}");
    }
}
```

# C# 9 – Module Initializers

Un metodo può essere designato come iniziatore di modulo decorandolo con l'attributo **[ModuleInitializer]**. Il metodo deve essere statico, senza parametri, restituire void, non essere generico e deve essere accessibile dal modulo contenente.

```
using System.Runtime.CompilerServices;
class C
{
    [ModuleInitializer]
    internal static void M1()
    {
        // ...
    }
}
```

# C# 9 – Partial Methods

- Ora supportano modificatori di accesso
- Possono avere tipi di ritorno non void

```
public partial class MyClass  
{  
    public partial int Compute();  
}
```

# C# 9 – Extension GetEnumerator() for foreach loops

```
foreach (var n in 1..10)
{
    Console.WriteLine(n);
}
```

```
public static class RangeExtensions
{
    private static IEnumerable<int> Enumerate(this Range range)
    {
        for (var i = range.Start.Value; i <= range.End.Value; i++)
            yield return i;
    }

    public static IEnumerator<int> GetEnumerator(this Range range)
    {
        return range.Enumerate().GetEnumerator();
    }
}
```



# C# 10



Rilasciato a Novembre 2021  
insieme a .NET 6

# C# 10 – Global Usings

La direttiva `global using` consente di dichiarare direttive `using` che sono valide per l'intero progetto, senza doverle ripetere in ogni file.

```
global using System;  
global using System.Collections.Generic;  
global using System.Linq;
```



# C# 10 – Implicit Usings

Abilitando gli using impliciti nel file .csproj, vengono aggiunte automaticamente direttive using comuni in base al tipo di progetto, semplificando ulteriormente il codice.

```
<PropertyGroup>  
  <ImplicitUsings>enable</ImplicitUsings>  
</PropertyGroup>
```

SDK	Default namespaces
Microsoft.NET.Sdk	<a href="#">System</a> <a href="#">System.Collections.Generic</a> <a href="#">System.IO</a> <a href="#">System.Linq</a> <a href="#">System.Net.Http</a> <a href="#">System.Threading</a> <a href="#">System.Threading.Tasks</a>
Microsoft.NET.Sdk.Web	Microsoft.NET.Sdk namespaces <a href="#">System.Net.Http.Json</a> <a href="#">Microsoft.AspNetCore.Builder</a> <a href="#">Microsoft.AspNetCore.Hosting</a> <a href="#">Microsoft.AspNetCore.Http</a> <a href="#">Microsoft.AspNetCore.Routing</a> <a href="#">Microsoft.Extensions.Configuration</a> <a href="#">Microsoft.Extensions.DependencyInjection</a> <a href="#">Microsoft.Extensions.Hosting</a> <a href="#">Microsoft.Extensions.Logging</a>
Microsoft.NET.Sdk.Worker	Microsoft.NET.Sdk namespaces <a href="#">Microsoft.Extensions.Configuration</a> <a href="#">Microsoft.Extensions.DependencyInjection</a> <a href="#">Microsoft.Extensions.Hosting</a> <a href="#">Microsoft.Extensions.Logging</a>
Microsoft.NET.Sdk.WindowsDesktop (Windows Forms)	Microsoft.NET.Sdk namespaces <a href="#">System.Drawing</a> <a href="#">System.Windows.Forms</a>
Microsoft.NET.Sdk.WindowsDesktop (WPF)	Microsoft.NET.Sdk namespaces Removed <a href="#">System.IO</a> Removed <a href="#">System.Net.Http</a>



# C# 10 – Structs

Le struct possono ora avere costruttori senza parametri definiti dall'utente e inizializzatori di campo, offrendo maggiore flessibilità.

```
public struct Indirizzo
{
    public string Città { get; init; } = "<sconosciuta>";
}
```

Introdotti i record struct, che combinano i vantaggi dei record con le struct.

```
public record struct Persona(string Nome, string Cognome);
```

# C# 10 – \$

L'interpolazione di stringhe fornisce una sintassi più leggibile e conveniente per formattare le stringhe. È più facile da leggere rispetto alla formattazione composita delle stringhe.

```
var name = "Mark";  
var date = DateTime.Now;
```

```
// Formattazione composita:
```

```
Console.WriteLine("Hello, {0}! Today is {1}, it's {2:HH:mm} now.", name,  
date.DayOfWeek, date);
```

```
// Interpolazione di stringhe:
```

```
Console.WriteLine($"Hello, {name}! Today is {date.DayOfWeek}, it's {date:HH:mm}  
now.");
```

# C# 10 – \$ FormattableString

Classe in C# che rappresenta una stringa interpolata.

```
double speedOfLight = 299792.458;
FormattableString message = $"The speed of light is {speedOfLight:N3} km/s.";

var specificCulture = CultureInfo.GetCultureInfo("en-IN");
string messageInSpecificCulture = message.ToString(specificCulture);
Console.WriteLine(messageInSpecificCulture);
// Output: The speed of light is 2,99,792.458 km/s.

string messageInInvariantCulture = FormattableString.Invariant(message);
Console.WriteLine(messageInInvariantCulture);
// Output: The speed of light is 299,792.458 km/s.
```

# C# 10 – File-scoped namespace declaration

```
namespace SampleNamespace  
{  
    class SampleClass { }  
}
```



```
namespace SampleNamespace;  
class SampleClass { }
```



```
package samplenamespace;  
class SampleClass { }
```

# C# 10 – Lambdas

- Le espressioni lambda possono ora avere un tipo "naturale", permettendo al compilatore di inferire il tipo senza una conversione esplicita.

```
var parse = (string s) => int.Parse(s); // Il tipo è inferito come Func<string, int>
```

- I metodi con un solo overload possono avere un tipo naturale, facilitando l'assegnazione a variabili tipizzate.

```
var read = Console.Read; // Inferito come Func<int>
```

- Tipi di ritorno espliciti per le lambda

```
var choose = object (bool b) => b ? 1 : "due"; // Func<bool, object>
```

- Attributi sulle lambda

```
Func<string, int> parse = [Esempio(1)] (s) => int.Parse(s);
```

# C# 11

Rilasciato a Novembre 2022  
insieme a .NET 7

# C# 11 – Generic Math

**Generic Math** è una funzionalità che consente di definire operazioni matematiche su tipi generici. Estende le capacità dei generics permettendo operazioni numeriche senza dover specificare un tipo concreto come int, double, ecc. Questo è reso possibile grazie ai **static abstract methods** nelle interfacce, che consentono di imporre vincoli sui tipi generici per includere operazioni matematiche.



# C# 11 – Static virtual e static abstract

E' possibile definire membri static abstract o static virtual nelle interfacce, che possono includere overload operators, metodi statici e proprietà statiche.

```
public struct MyNumber : IAdditionOperators<MyNumber, MyNumber, MyNumber>
{
    public static MyNumber operator +(MyNumber left, MyNumber right) =>
        new MyNumber { Value = left.Value + right.Value };

    public int Value { get; set; }
}
```

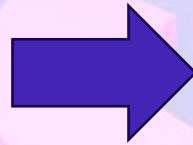


# C# 11 – Generic Attributes

È possibile dichiarare una classe generica la cui classe di base è `System.Attribute`. Questa funzionalità offre una sintassi più pratica per gli attributi che richiedono un parametro `System.Type`. Nelle versioni precedenti è necessario creare un attributo che accetta un `Type` come parametro del relativo costruttore:

```
public class TypeAttribute : Attribute
{
    public TypeAttribute(Type t) => ParamType = t;

    public Type ParamType { get; }
}
```

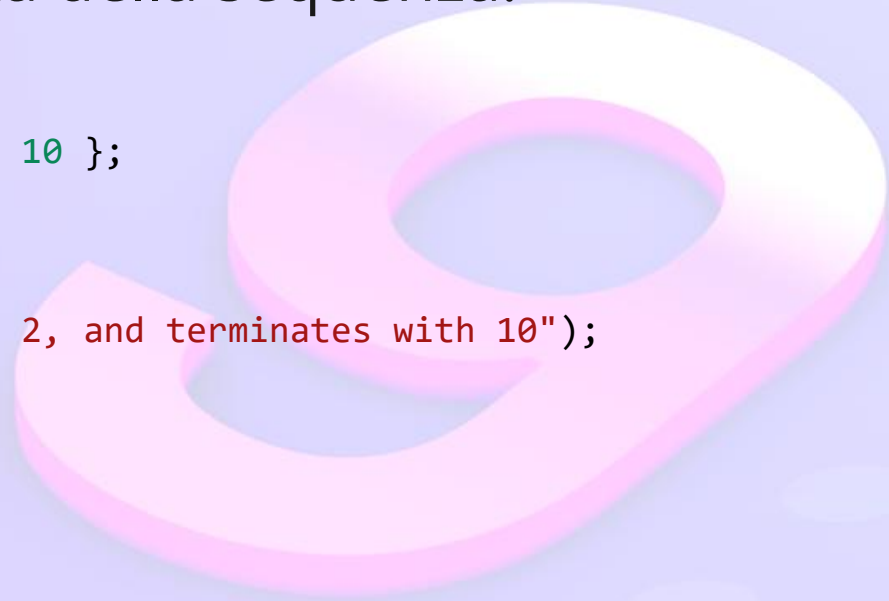


```
public class GenericAttribute<T> : Attribute
{
    public Type ParamType { get; } = typeof(T)
}
```

# C# 11 – List Pattern

Estensione del pattern matching per consentire il confronto di sequenze di elementi in array o liste. È possibile verificare la presenza di elementi specifici, indipendentemente dalla lunghezza della sequenza.

```
var numbers = new [] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
  
if(numbers is [1,2,..,10])  
{  
    Console.WriteLine("The list starts with 1 and 2, and terminates with 10");  
}
```



# C# 11 – Required

È possibile contrassegnare proprietà e campi con il modificatore required, imponendo l'inizializzazione di questi membri durante la creazione dell'istanza dell'oggetto.

Il costruttore di una classe che imposta i membri required deve essere annotato con **[SetsRequiredMembers]**

```
var person = new Person
{
    Name = "John",
    Surname = "Doe"
};

public class Person
{
    public required string Name { get; set; }
    public required string Surname { get; set; }
}
```

# C# 11 – Required

```
var person = new Person("John", "Doe");

public class Person
{
    public required string Name { get; set; }
    public required string Surname { get; set; }

    [SetsRequiredMembers]
    public Person(string name, string surname)
    {
        Name = name;
    }
}
```

Da errore in compilazione,  
perché Surname non è  
inizializzato nel costruttore

Crea un'istanza di person  
con Name = John e  
Surname = Doe

Il codice a  
sinistra...

Solleva ArgumentException  
al runtime perché Surname  
non è inizializzato nel  
costruttore

Nessuna delle precedenti

# C# 11 – UTF-8 Strings

Tradizionalmente, le stringhe in C# utilizzano la codifica UTF-16. Con C# 11, è possibile specificare che una stringa utilizzi la codifica UTF-8 aggiungendo il suffisso u8 al letterale della stringa.

```
var utf8String = "Questa è una stringa in UTF-8!"u8;
```

# C# 11 – Raw string literals

Permettono di definire stringhe multi-linea senza la necessità di sequenze di escape, facilitando l'inclusione di testo con caratteri speciali, codice JSON, XML o espressioni regolari. Un raw string literal inizia e termina con tre o più virgolette doppie (""")

```
var json = """
{
    "nome": "Mario",
    "età": 30,
    "linguaggi": ["C#", "Python", "JavaScript"]
}
""";
```

# C# 12

Rilasciato a Novembre 2023  
insieme a .NET 8

# C# 12 – Primary Constructors

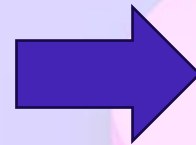
Estendono la possibilità di definire costruttori direttamente nella dichiarazione di una classe o struct, funzionalità precedentemente limitata ai record.

I parametri del costruttore primario sono accessibili in tutto il corpo della classe o struct, semplificando l'inizializzazione dei membri.

```
public class Punto
{
    public int X { get; }
    public int Y { get; }

    public Punto(int x, int y)
    {
        X = x;
        Y = y;
    }

    public double Distanza => Math.Sqrt(X * X + Y * Y);
}
```



```
public class Punto(int x, int y)
{
    public int X { get; } = x;
    public int Y { get; } = y;

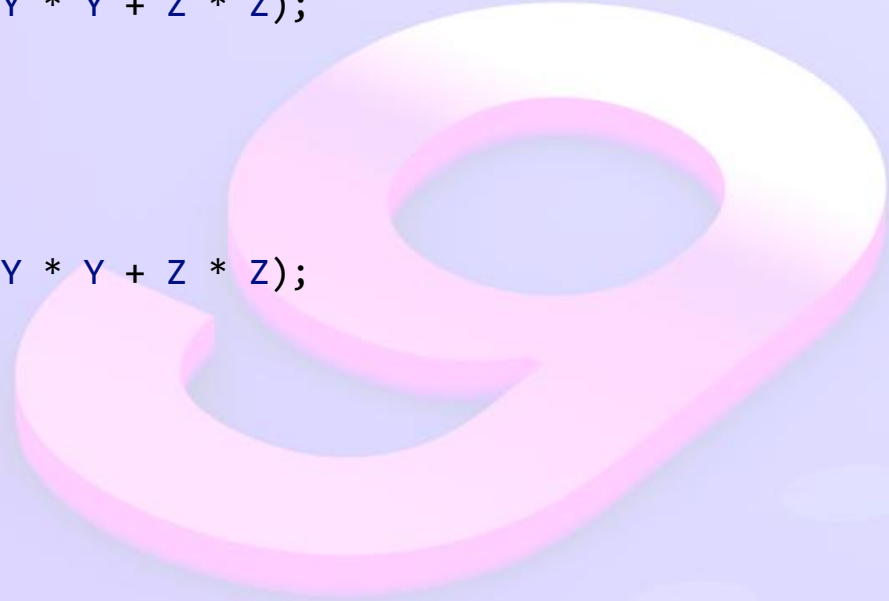
    public double Distanza => Math.Sqrt(x * x + y * y);
}
```



# C# 12 – Primary Constructors

```
public class VectorClass(int X, int Y, int Z)
{
    public double Magnitude => Math.Sqrt(X * X + Y * Y + Z * Z);
}
```

```
public record VectorRecord(int X, int Y, int Z)
{
    public double Magnitude => Math.Sqrt(X * X + Y * Y + Z * Z);
}
```



# C# 12 – Collection Expressions

Introducono una nuova sintassi terse per creare valori di collections comuni. Inoltre, l'operatore di diffusione (..) consente di includere elementi da altre collezioni

```
// Creazione di un array
int[] numeri = [1, 2, 3, 4, 5];

// Creazione di una lista
List<string> parole = ["uno", "due", "tre"];

// Creazione di uno span
Span<char> lettere = ['a', 'b', 'c', 'd'];
```

```
List<int> parte1 = [1, 2, 3];
ReadOnlySpan<int> parte2 = [4, 5, 6];
int[] tutti = [..parte1, ..parte2, 7, 8, 9];
// tutti contiene [1, 2, 3, 4, 5, 6, 7, 8, 9]
```



# C# 13



Rilasciato a Novembre 2024  
insieme a .NET 9

# C# 13 – Params Collections

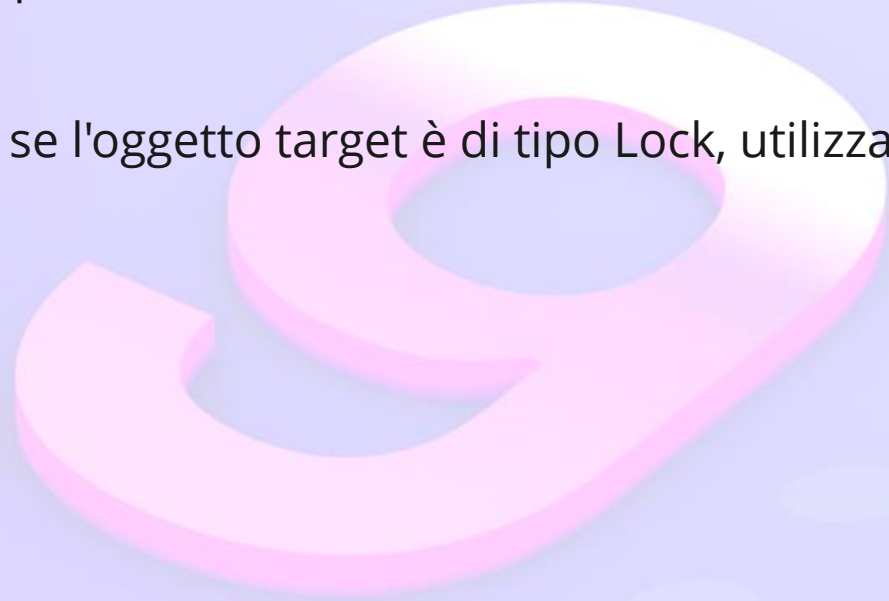
- Il modificatore **params** non è più limitato ai tipi di array.
- Ora può essere utilizzato con qualsiasi tipo di raccolta riconosciuto, inclusi `System.Span<T>`, `System.ReadOnlySpan<T>`, e tipi che implementano `System.Collections.Generic.IEnumerable<T>` e possiedono un metodo `Add`.

```
public void Concat<T>(params ReadOnlySpan<T> items)
{
    foreach (var item in items)
    {
        Console.Write(item);
        Console.Write(" ");
    }
    Console.WriteLine();
}
```



# C# 13 – Lock

- Introdotto il tipo `System.Threading.Lock` per una migliore sincronizzazione dei thread.
- Il metodo `Lock.EnterScope()` entra in uno scope esclusivo, restituendo un ref struct che supporta il pattern `Dispose()` per uscire dallo scope.
- Il costrutto `lock` riconosce automaticamente se l'oggetto `target` è di tipo `Lock`, utilizzando l'API aggiornata invece di `System.Threading.Monitor`.



# C# 13 – Allows ref struct

I tipi generici possono ora dichiarare un anti-vincolo allows ref struct, permettendo l'uso di tipi ref struct come argomenti di tipo.

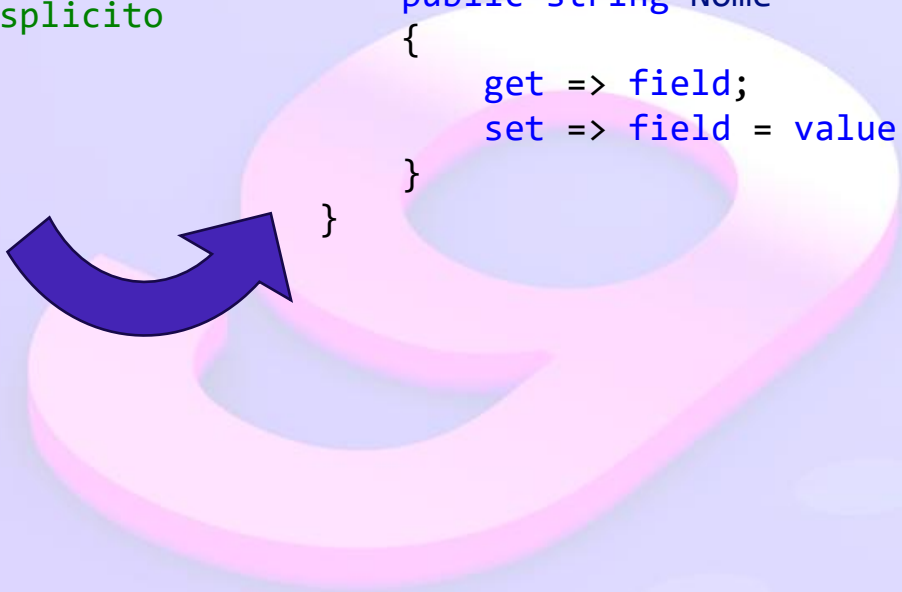
```
public class C<T> where T : allows ref struct
{
    public void M(scoped T p)
    {
        // Utilizzo di T come ref struct
    }
}
```

# C# 13 – field

```
public class Persona
{
    private string _nome; // Campo di supporto esplicito

    public string Nome
    {
        get => _nome;
        set => _nome = value.Trim();
    }
}
```

```
public class Persona
{
    public string Nome
    {
        get => field;
        set => field = value.Trim();
    }
}
```



# La direzione del linguaggio

Multipiattaforma

Open Source

Developer  
Productivity


Alte prestazioni

Multiparadigma



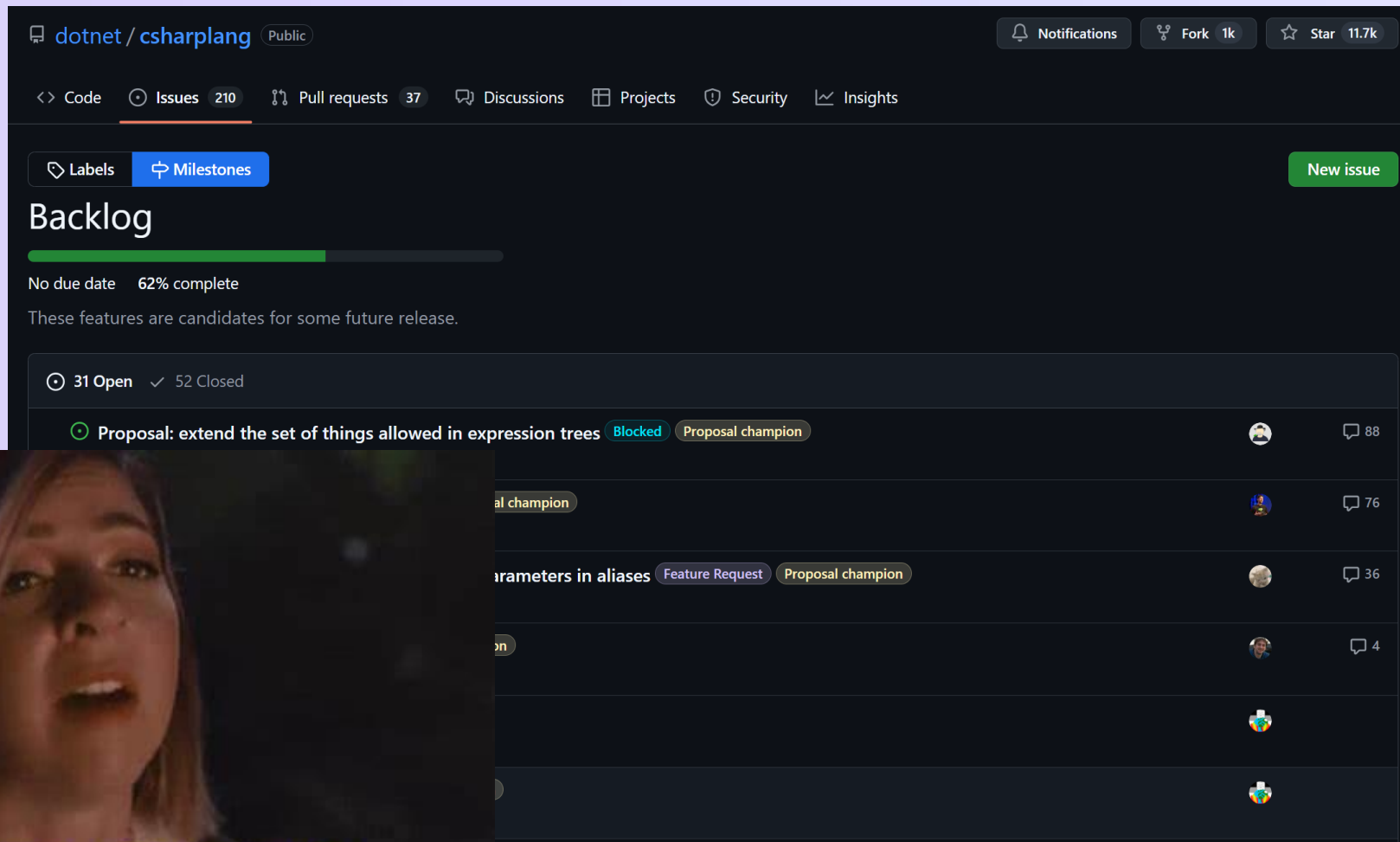


# C# 14



Sarà rilasciato a Novembre  
2025 insieme a .NET 10

# Il backlog è carico



dotnet / csharp-lang Public

Notifications Fork 1k Star 11.7k

Code Issues 210 Pull requests 37 Discussions Projects Security Insights

Labels Milestones New issue

## Backlog

No due date 62% complete

These features are candidates for some future release.

31 Open 52 Closed

- Proposal: extend the set of things allowed in expression trees (Blocked, Proposal champion) 88
- al champion 76
- parameters in aliases (Feature Request, Proposal champion) 36
- on 4



# Grazie!



## Nicola Paro

Solution Architect – beanTech



[linktr.ee/nicolaparo](https://linktr.ee/nicolaparo)