aspitalia.com

C# 12
Cosa c'è di nuovo e interessante

UGIdotNET

Nicola Iarocci
Microsoft MVP
info@nicolaiarocci.com | @nicolaiarocci

.NET Conference
Italia 2023

.NET

What’s new in C# 12

- **Primary constructors**
- **Collection expressions**
- Alias any type
- Default lambda parameters
- ref readonly parameters
- Inline arrays
- Experimental attribute
- Interceptors (“Preview”)

Primary constructors

You can now create primary constructors in any class and struct.
Primary constructors are no longer restricted to record types.

Demo

Primary constructors

.NET Conference
Italia 2023

.NET

Primary constructors

- You can now add parameters to a struct or class declaration
- Parameters are in scope throughout the class definition
- Parameters are not stored if they aren't needed
- When needed the compiler creates hidden fields to represent each parameter
- Parameters aren't members of the class (`this.parameter` won’t work)
- Parameters don't become properties, except in record types (we can create properties)
- Secondary or parameterless constructors must invoke the primary constructor
- Validation can be added when assigning the corresponding properties
- Derived type can have a PC (it must invoke the base class primary constructor)
- Derived type can avoid a PC (a regular constructor must invoke the base primary)
- In derived types, watch out for ‘nested captures’ of primary parameters values
- VS and VSCode offer built-in support for primary constructors (refactorings, etc.)
- Trivia: the original implementation goes back to C# 6 (2015)

Collection expressions

Collection expressions introduce a new terse syntax to create common collection values

Demo

Collection expressions

.NET Conference
Italia 2023

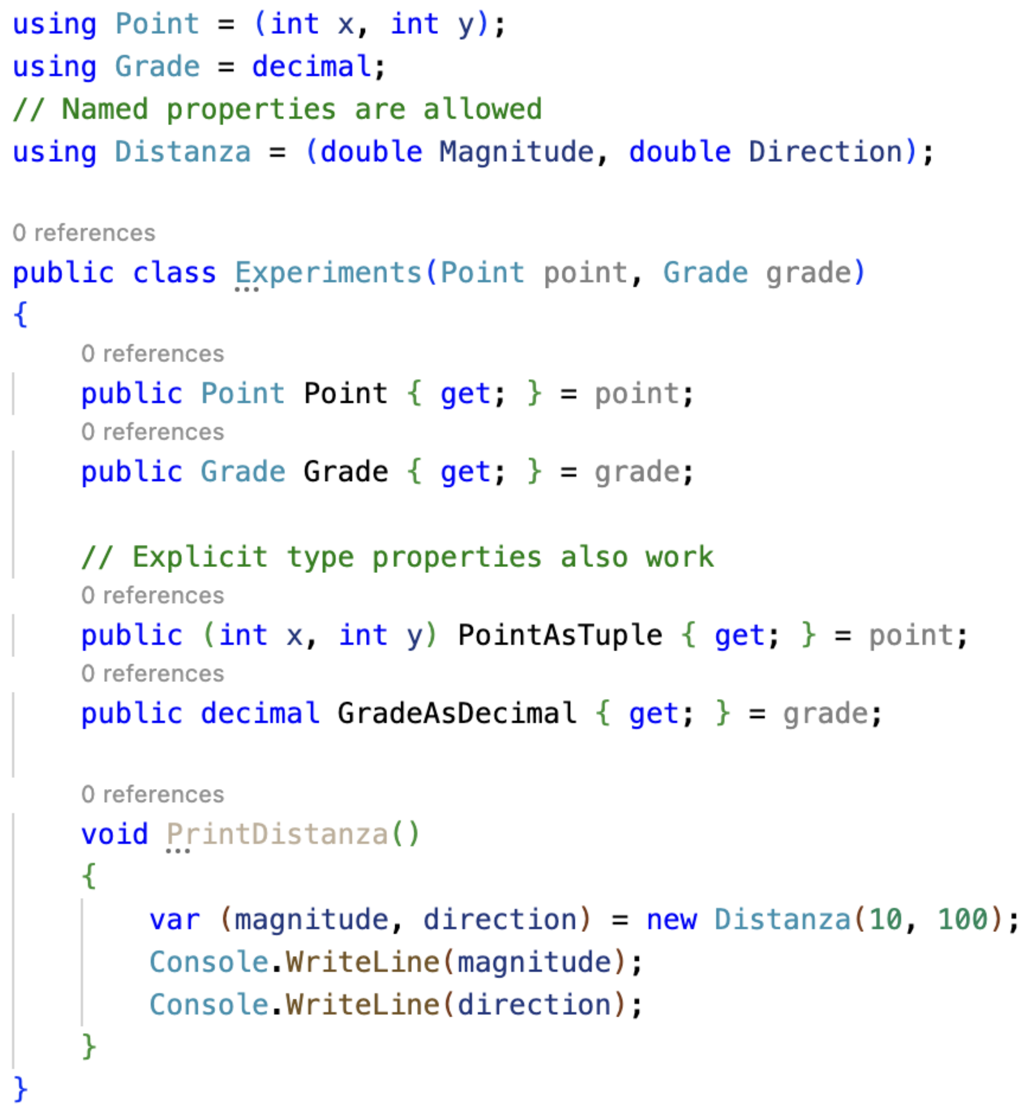
.NET

Collection expressions

- Collection expressions offer a terse, unified syntax
- In most cases, they also offer superior performance
- Support a large number of collection types and variants
- They avoid refactoring when the underlying type changes
- Accept both constant and variable values
- Support inclusion of other collections via spread operator
- Can be supported even in custom types (library authors!)
- Syntax symmetricity with pattern matching and/or slicing
- VS and VS Code offer full collection expressions support
- Trivia: Dictionary expressions? Maybe in the future

Alias any type

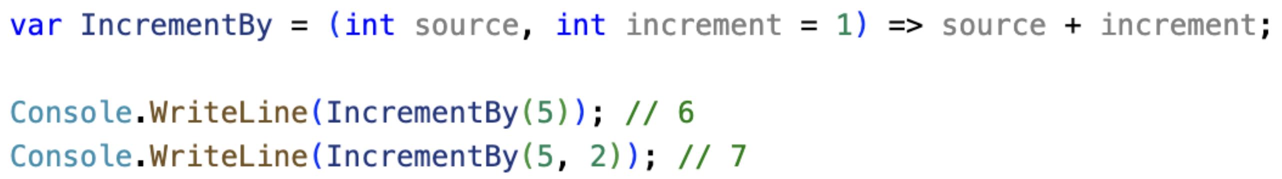
You can use the using alias directive to alias any type, not just named types.



```
using Point = (int x, int y);
using Grade = decimal;
// Named properties are allowed
using Distanza = (double Magnitude, double Direction);

0 references
public class Experiments(Point point, Grade grade)
{
    0 references
    public Point Point { get; } = point;
    0 references
    public Grade Grade { get; } = grade;

    // Explicit type properties also work
    0 references
    public (int x, int y) PointAsTuple { get; } = point;
    0 references
    public decimal GradeAsDecimal { get; } = grade;

    0 references
    void PrintDistanza()
    {
        var (magnitude, direction) = new Distanza(10, 100);
        Console.WriteLine(magnitude);
        Console.WriteLine(direction);
    }
}
```

Default lambda parameters

Beginning with C# 12, you can provide default values for parameters on lambda expressions.



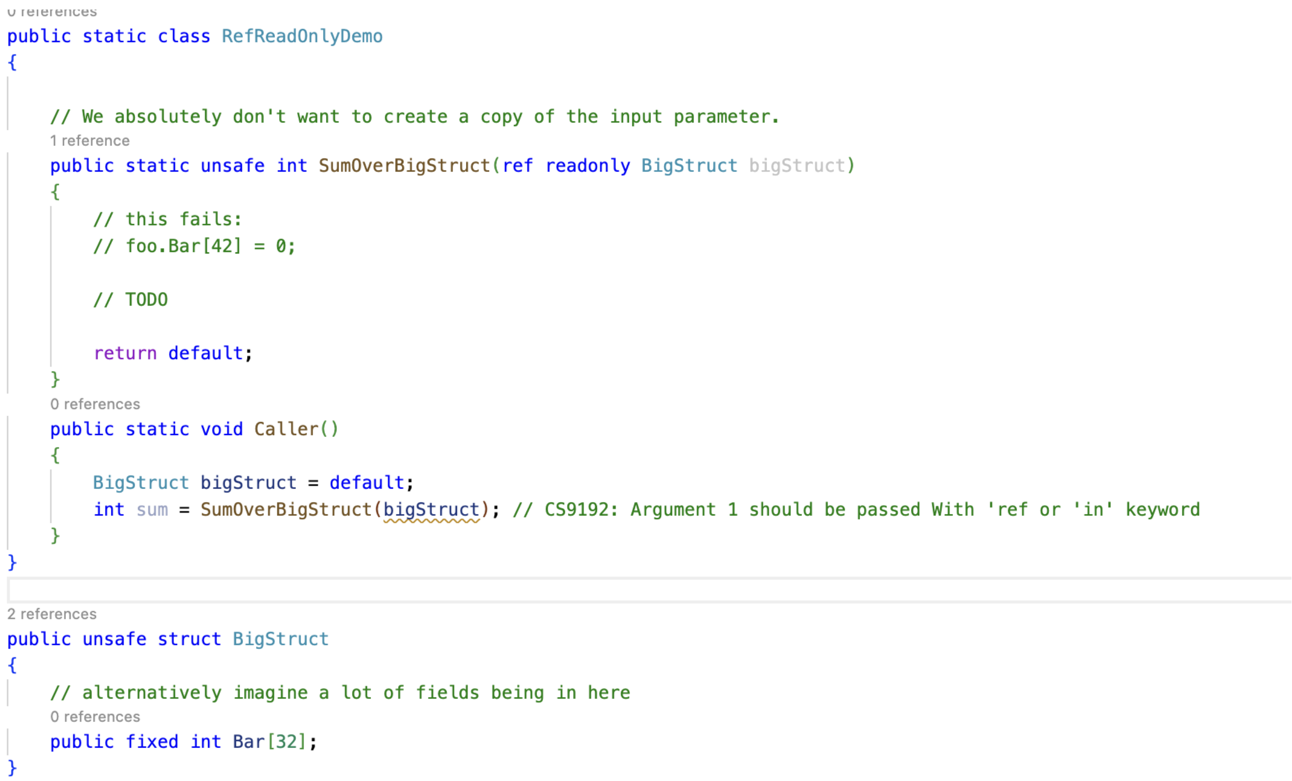
```
var IncrementBy = (int source, int increment = 1) => source + increment;

Console.WriteLine(IncrementBy(5)); // 6
Console.WriteLine(IncrementBy(5, 2)); // 7
```

ref readonly

The addition of ref readonly parameters provides the final combination of passing parameters by reference or by value.

Assume you have a fairly large struct that you absolutely don't want to copy around: `def readonly` triggers a warning on the caller side unless he/she uses ‘ref’ or ‘in’.

Mostly used by the runtime team and library authors. Performance and clarity, again.



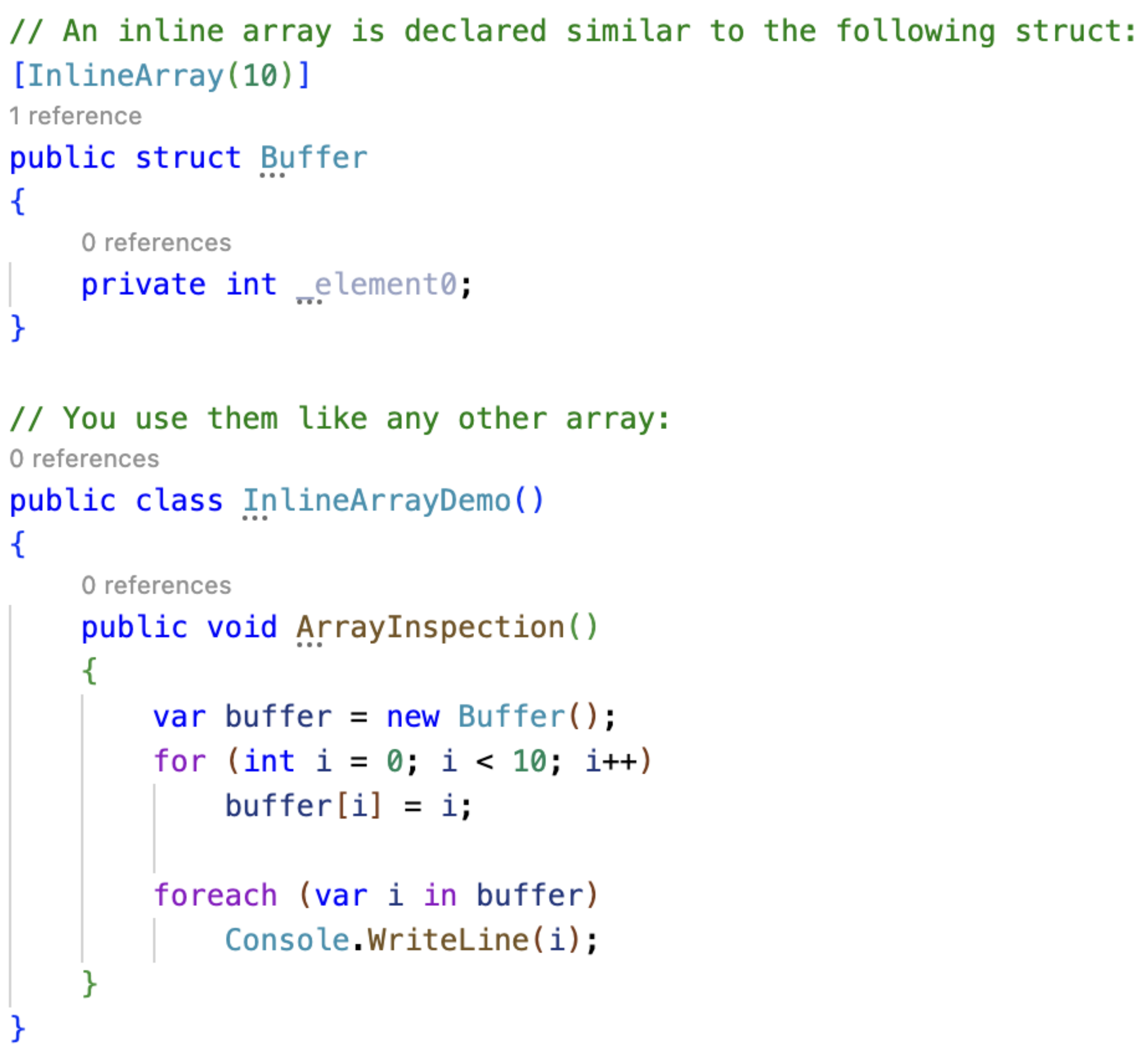
```
0 references
public static class RefReadOnlyDemo
{

    // We absolutely don't want to create a copy of the input parameter.
    1 reference
    public static unsafe int SumOverBigStruct(ref readonly BigStruct bigStruct)
    {
        // this fails:
        // foo.Bar[42] = 0;

        // TODO

        return default;
    }
    0 references
    public static void Caller()
    {
        BigStruct bigStruct = default;
        int sum = SumOverBigStruct(bigStruct); // CS9192: Argument 1 should be passed With 'ref or 'in' keyword
    }
}

2 references
public unsafe struct BigStruct
{
    // alternatively imagine a lot of fields being in here
    0 references
    public fixed int Bar[32];
}
```

Inline arrays

Inline arrays enable a developer to create an array of fixed size in a struct type.

They are used mainly by the runtime team and other library authors for improved performance *with* safety. Inline arrays perform similar to unsafe fixed size buffers.

You likely won't declare your own inline arrays, but you use them transparently when they're exposed as `System.Span<T>` or `System.ReadOnlySpan<T>` objects from runtime APIs.



```
// An inline array is declared similar to the following struct:
[InlineArray(10)]
1 reference
public struct Buffer
{
    0 references
    private int _element0;
}

// You use them like any other array:
0 references
public class InlineArrayDemo()
{
    0 references
    public void ArrayInspection()
    {
        var buffer = new Buffer();
        for (int i = 0; i < 10; i++)
            buffer[i] = i;

        foreach (var i in buffer)
            Console.WriteLine(i);
    }
}
```

@nicolaiarocci
nicolaiarocci.com

aspitalia.com

UGIdotNET

Slide e materiale su
https://www.dotnetconference.it/

.NET Conference
Italia 2023

.NET