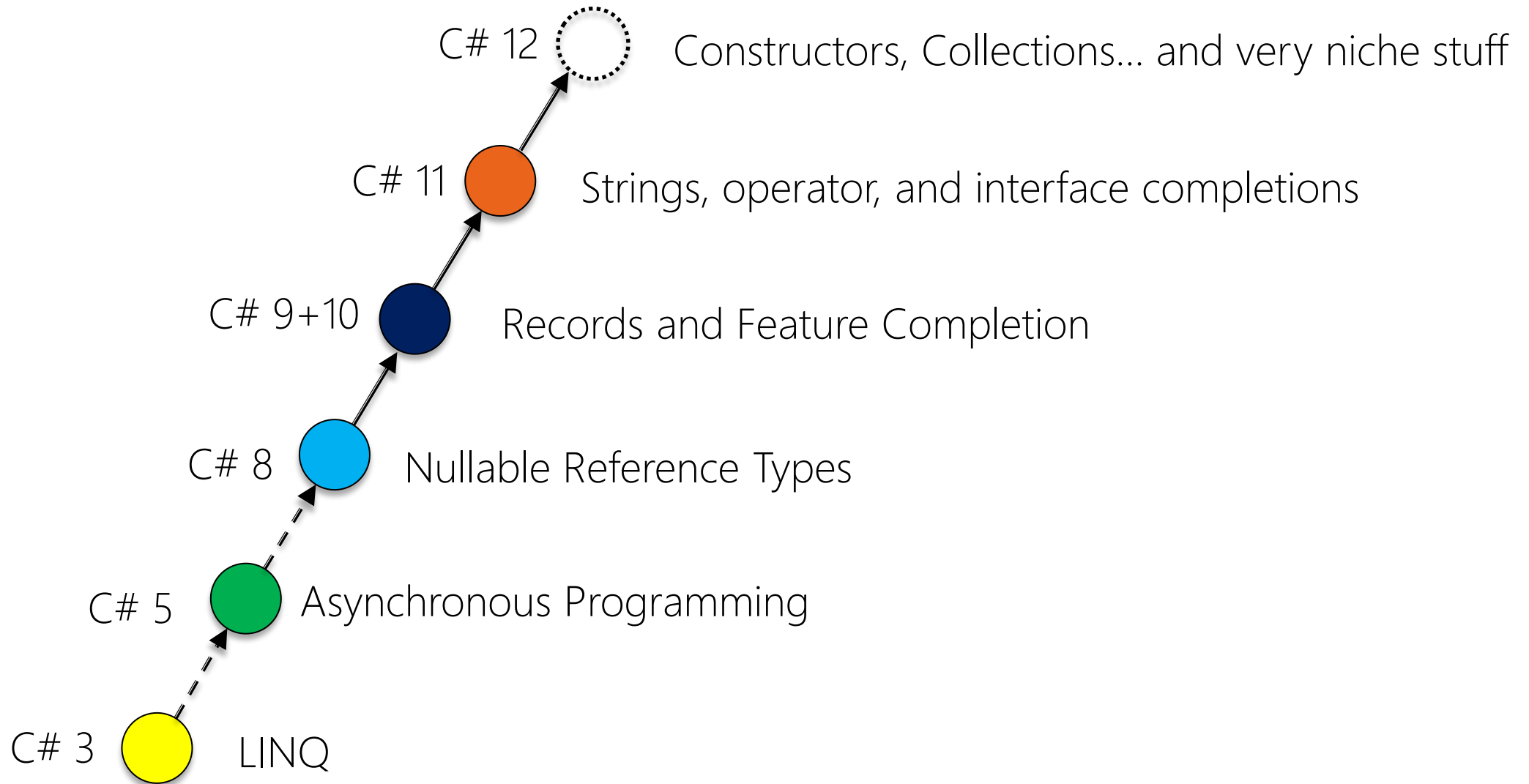


Module 06:

"Newest Features in C# 12"



Major Evolutions of C#



Agenda

- ▶ Introduction
- ▶ Object-Oriented Improvements
- ▶ Collection Improvements
- ▶ Method Improvements
- ▶ Namespaces and Usings
- ▶ Unsafe Improvements
- ▶ Summary



Introducing Primary Constructors

- ▶ Classes can now have *primary constructors*

```
class BankAccount(decimal initialBalance)
{
    public decimal Balance { get; private set; } = initialBalance;

    public void Deposit(decimal amount) => Balance += amount;
}
```

- ▶ Looks like the primary constructors for records...
 - ...but not identical!
- ▶ Note: Constructor parameters available throughout entire type

Parameter Capturing

- ▶ Primary constructor parameters can be captured lambda-style

```
class BankAccount(decimal initialBalance)
{
    ...
    public void Deposit(decimal amount)
    {
        Balance += amount;
        WriteLine( $"Balance is now {Balance:c} (initially: {initialBalance})" );
    }
}
```

- ▶ *Potentially* in scope for the *entire* lifespan of the type
- ▶ Note:
 - Not readonly...!
 - Initialization vs. Computation
 - Can "uncapture" if desired

Constructor Chaining

- ▶ Primary constructor must be at the top of the constructor chain

```
class BankAccount(decimal initialBalance)
{
    public decimal Balance { get; private set; } = initialBalance;

    public BankAccount() : this(0)
    {
    }
}
```

- ▶ All other usual rules regarding constructor chaining apply
 - E.g. for inheritance



Use Cases for Primary Constructors

- ▶ Many excellent use cases for constructors
- ▶ "Use Primary Constructor"
- ▶ "Use Primary Constructor (And Remove Fields)"
- ▶ Primary constructors still work for Dependency Injection
 - But required dependencies are slightly less explicit



Primary Constructors for Structs

- ▶ Also available for structs

```
struct Money(int euro, int cents)
{
    public int Euro { get; init; } = euro;
    public int Cents { get; init; } = cents;

    public override readonly string ToString() => $"EUR {Euro}:{Cents:d2}";
}
```

- ▶ Works in a manner similar to classes, except
 - For classes the default constructor is not created when primary constructor
 - For structs the default constructor is created regardless



Agenda

- ▶ Introduction
- ▶ Object-Oriented Improvements
- ▶ **Collection Improvements**
- ▶ Method Improvements
- ▶ Namespaces and Usings
- ▶ Unsafe Improvements
- ▶ Summary



Collection Expressions

- ▶ Unified collection syntax across a multitude of collection types

```
class LookupTable(List<string> elements, Func<string, string> mapping)
{
    public LookupTable() : this([], s => s) {}

    public string Get(Index index) => mapping(elements[index]);
}
```

```
List<string> elements = ["Hello", "World", "Booyah"];
```

- ▶ Essentially the construction syntax corresponding to the matching syntax of C# 11



Supported Collection Types

- ▶ Arrays
- ▶ `Span<T>` and `ReadOnlySpan<T>`
- ▶ Types with collection initializer, such as `List<T>` and `Dictionary<K, V>`
- ▶ (and actually more such as `ImmutableArray<T>` and custom types)

```
int[] array = [1, 2, 3, 4, 5, 6, 7, 8];  
List<string> list = ["one", "two", "three"];  
Span<char> span = ['a', 'b', 'c', 'd', 'e', 'f', 'h', 'i'];  
int[][] array2d = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];  
  
// Create an enumerable? (WTF?!)  
IEnumerable<int> enumerable = [1, 2, 3];
```

Spread Operator

- ▶ The *spread operator* replaces its argument with the elements from that collection

```
int[] row0 = [1, 2, 3];  
int[] row1 = [4, 5, 6];  
int[] row2 = [7, 8, 9];  
  
int[] all = [...row0, ...row1, ...row2];  
  
foreach (var element in all)  
{  
    Console.WriteLine(element);  
}
```



Argument must be an enumerable expression

Frozen Collections

- ▶ .NET 8 introduces a new set of *Frozen Collections*
 - `FrozenSet<T>`
 - `FrozenDictionary<K, V>`

```
using System.Collections.Frozen;

List<int> list = [11, 22, 33];
FrozenSet<int> frozen = list.ToFrozenSet(); // Now read-only
if(frozen.TryGetValue(22, out int actualValue))
{
    Console.WriteLine($"Got {actualValue}");
}
```

- ▶ "But why"? Performance..! ☺
 - There is no `FrozenList<T>`



Agenda

- ▶ Introduction
- ▶ Object-Oriented Improvements
- ▶ Collection Improvements
- ▶ **Method Improvements**
- ▶ Namespaces and Usings
- ▶ Unsafe Improvements
- ▶ Summary



Default and **params** Parameters in Lambdas

- ▶ Lambda expressions are now allowed default parameters like regular methods

```
var add = (int x, int y = 100) => x + y;
```

```
Console.WriteLine(add(42));
```

- ▶ Similarly, **params** is now allowed

```
var total = (params int[] elements) => elements.Sum();
```

```
Console.WriteLine(total(11, 22, 33));
```



ref readonly Parameters

- ▶ As a fine-graining of the **in** modifier, the **ref readonly** modifier is now allowed:

```
double CalculateDistance(ref readonly Point3D first, in Point3D second = default)
{
    double xDiff = first.X - second.X;
    double yDiff = first.Y - second.Y;
    double zDiff = first.Z - second.Z;

    return Sqrt(xDiff * xDiff + yDiff * yDiff + zDiff * zDiff);
}
```

- ▶ Can be used to force by-reference instead of the potential copying of **in**



Agenda

- ▶ Introduction
- ▶ Object-Oriented Improvements
- ▶ Collection Improvements
- ▶ Method Improvements
- ▶ **Namespaces and Usings**
- ▶ Unsafe Improvements
- ▶ Summary



Alias Any Type

- ▶ Now also *unnamed* types can be aliased with the **using** keyword

```
using Vector3D = (double x, double y, double z);  
  
var v1 = (1, 2, 3);  
var v2 = (4, 5, 6);  
Console.WriteLine(AddVectors(v1, v2));  
  
static Vector3D AddVectors(Vector3D first, Vector3D second) =>  
    (first.x + second.x, first.y + second.y, first.z + second.z);
```

- ▶ Great for tuple types and pointer types
- ▶ Remember global usings? ☺

Note: Cannot be nullable reference types at top-level

Agenda

- ▶ Introduction
- ▶ Object-Oriented Improvements
- ▶ Collection Improvements
- ▶ Method Improvements
- ▶ Namespaces and Usings
- ▶ **Unsafe Improvements**
- ▶ Summary



Inline Array Types

- ▶ Inlined and laid out sequentially

```
Buffer buffer = new();
```

```
buffer[0] = "Hello";
```

```
buffer[1] = "Inline";
```

```
Console.WriteLine(buffer[0]);
```

```
[System.Runtime.CompilerServices.InlineArray(10)]
```

```
public struct Buffer
```

```
{
```

```
    private object _element0;
```

```
}
```



Summary

- ▶ Introduction
- ▶ Primary Constructors
- ▶ Collection Improvements
- ▶ Method Improvements
- ▶ Namespaces and Usings
- ▶ Unsafe Improvements



