"What's New in C# 11 and 12?"

Danske Bank
December 5th 2023

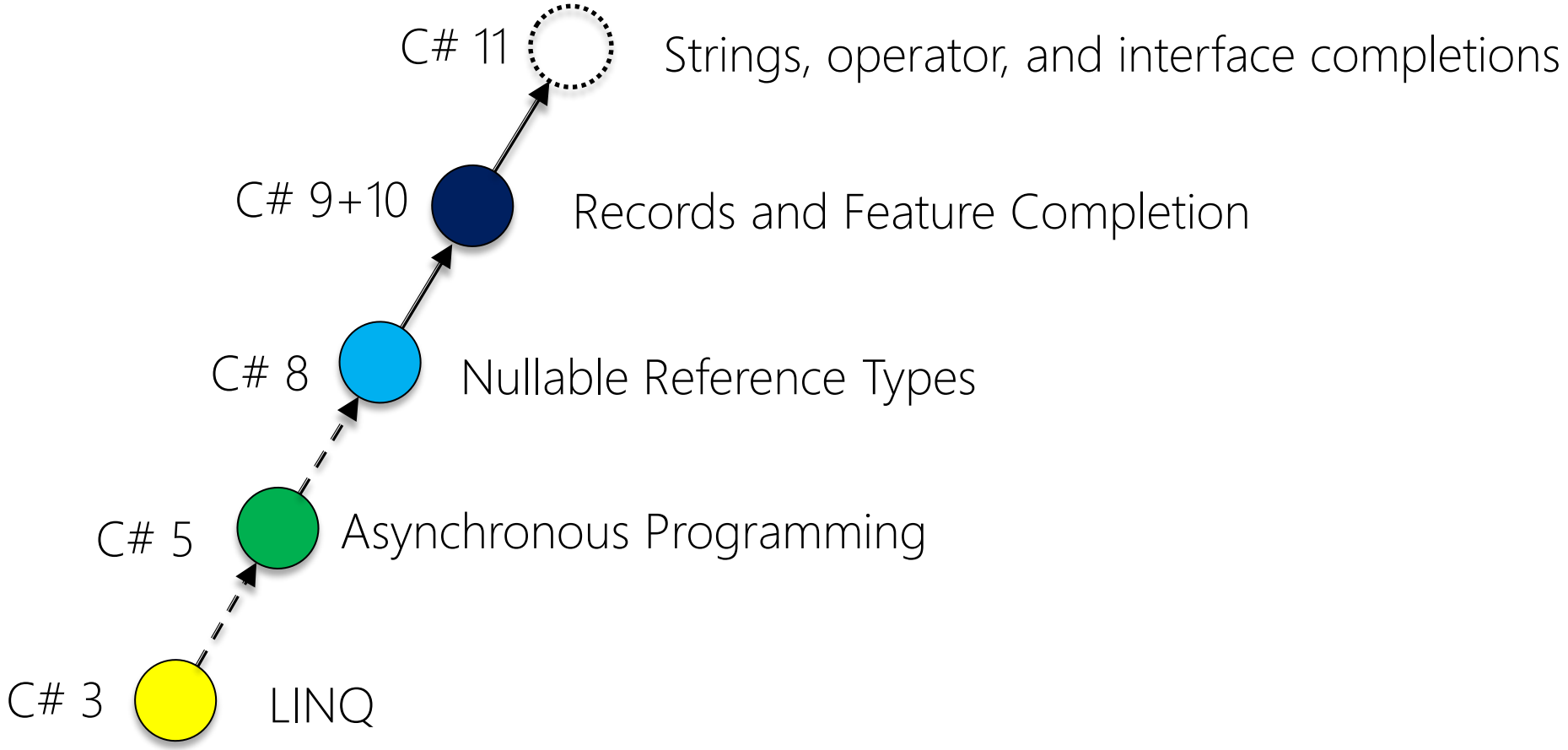
Jesper Gulmann Henriksen

Agenda

- What's New in C# 11?
- Newest Features in C# 12

- *Questions and Discussion*

Major Evolutions of C#



Agenda

- What's New in C# 11?
  - Introduction
  - String Improvements
  - Expression Improvements
  - Object-Oriented Improvements
  - Math and Operators
- Newest Features in C# 12

- *Questions and Discussion*

Raw String Literals

- Strings now support multi-line string literals using """

```
string s = """
    Hello,
    "World"
    """;

Console.WriteLine(s);
```

- Excellent for e.g. JSON or XML string literals
- Blocks of *n* "'s in strings can be escaped using *n*+1 "'s in begin and end

- Indentions can also be controlled by ending white-space before """

What about String Interpolation?

- String interpolation proceeds as usual, but might need **$$** and **{{}}** (or more ☺)

```
string firstName = "Jesper";
string lastName = "Gulmann";
string company = "Wincubate ApS";

string s = $$"""
    {
      "firstName": "{{firstName}}",
      "lastName": "{{lastName}}",
      "company": "{{company}}"
    }
    """;
```

- Note: Line breaks are now allowed within string interpolation expressions!

UTF-8 String Literals

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

ReadOnlySpan<byte> s2 = """
    Hello,
    "World"
    """u8;
```

- Note:
  - Not strings exactly, but strings already encoded as bytes.
  - Not compile-time constants, because `ReadOnlySpan<byte>` cannot be const

```
var moreBytes = "Hello, "u8 + "World"u8 + "!!"u8;
byte[] moreBytesArray = moreBytes.ToArray();
```

Agenda

- What's New in C# 11?
  - Introduction
  - String Improvements
  - Expression Improvements
  - Object-Oriented Improvements
  - Math and Operators
- Newest Features in C# 12

- *Questions and Discussion*

Pattern-matching Enhancements

- C# 7, 8, 9, and 10 introduced a total of 13 patterns and enhancements

- C# 11 introduces 3 additional list and string patterns or enhancements:
  - List patterns *[a,b,c]* e.g. `[11,22,33]`
  - Slice (or range) patterns .. e.g. `[11, ..]`
  - Spans of chars for constant string "ABC" e.g. `"ABC"`

List Patterns

- Can now match sequences against specific element patterns

```
var elements = new int[] { 11, 22, 33 };

Console.WriteLine(elements is [11, 22, 33]);
Console.WriteLine(elements is [11, 22, 33, 44]);
Console.WriteLine(elements is [>10, <100, 33 or 44]);
```

- Works for types which are *countable* and *indexable*

- Discard pattern _ can be used to match single elements in list patterns

```
Console.WriteLine(elements is [11, _, 33]);
Console.WriteLine(elements is [11, _, _, _]);
```

Slice Patterns

- The Slice (a.k.a. Range) Pattern `..` can be used *at most once* within a list pattern

```
var elements = new int[] { 11, 22, 33 };

Console.WriteLine(elements is [11, ..]);
Console.WriteLine(elements is [.., 33, 44]);
Console.WriteLine(elements is [11, ..] or [.., 44]);
```

- Works for types which are *countable* and *sliceable*

- Slice elements can also be extracted

```
if( elements is [11, ..var sub, _])
{
    // Print sub here
}
```

Agenda

- What's New in C# 11?
  - Introduction
  - String Improvements
  - Expression Improvements
  - Object-Oriented Improvements
  - Math and Operators
- Newest Features in C# 12

- *Questions and Discussion*

Required Members

- Express that a member must be initialized during construction
  - *Not* required to be initialized to a valid nullable state at the end of the constructor

```
class Person
{
    public required string FirstName { get; init; }
    public string? MiddleName { get; init; }
    public required string LastName { get; init; }
}
```

- Defer the check to the site of object construction

- Help address the shortcoming of nullability checks for reference types of C# 8
- But are actually completely orthogonal to non-nullable reference types
  - Also work for nullable types etc.

[SetsRequiredMembers]

- Asserts that a specific constructor initializes all required members

```
class Person
{
    ...
    [SetsRequiredMembers]
    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }
}
```

- Essentially this is the "**!**" of required members at the constructor level
- Note: Static analysis does *not* check whether correct!

File Accessibility Modifier

- New access modifier on type definitions only
  - Restricts visibility to defining *file*

```
file class C
{
    public static void M()
    {
        Console.WriteLine("Hello from File1");
    }
}
```

- No accessibility modifiers can be used in combination with **`file`**
- Some restrictions apply

Static Abstract Members in Interfaces

- You can add static abstract members in interfaces

```
interface ICanBeEmpty<T>
{
    static abstract T Empty { get; }
}
```

- Can define static abstract properties, methods, events, and operators
  - We will make crucial use of this in the "Math and Operators" section later!

```
class Person : ICanBeEmpty<Person>
{
    public static Person Empty => new Person { ... };

    ...
}
```

Static Virtual Members in Interfaces

- Similarly, static virtual members are now allowed in interfaces

```
interface ICanCreateDefault<T> where T : ICanCreateDefault<T>, new()
{
    static virtual T CreateDefault() => new();
}
```

- Enables polymorphism where the method called depends on the compile-time type rather than the runtime instance type

- Static members are also allowed to be **`sealed`**

Generic Attributes

- C# 11 finally allows custom generic attributes

```
[AttributeUsage(AttributeTargets.All)]
public class DeveloperAttribute<T> : Attribute
{
    public T Info { get; init; }

    public DeveloperAttribute(T info)
    {
        Info = info;
    }
}
```

Agenda

- **What's New in C# 11?**
  - Introduction
  - String Improvements
  - Expression Improvements
  - Object-Oriented Improvements
  - **Math and Operators**
- Newest Features in C# 12

- *Questions and Discussion*

Generic Math Support

- Goal: Use mathematical operators in generic types

- `static abstract` / `virtual` members in interfaces
- checked user defined operators
- relaxed shift operators
- unsigned right-shift operator

INumber<T>

- Math operators are now generic

```
T MultiplySequence<T>( IEnumerable<T> sequence ) where T : INumber<T>
{
    T total = T.One;
    foreach (T i in sequence)
    {
        total *= i;
    }
    return total;
}
```

Unsigned Right Shift Operator

- Before C# 11: to force an unsigned right-shift, you would need to
  - cast any signed integer type to an unsigned type
  - perform the shift
  - cast the result back to a signed type

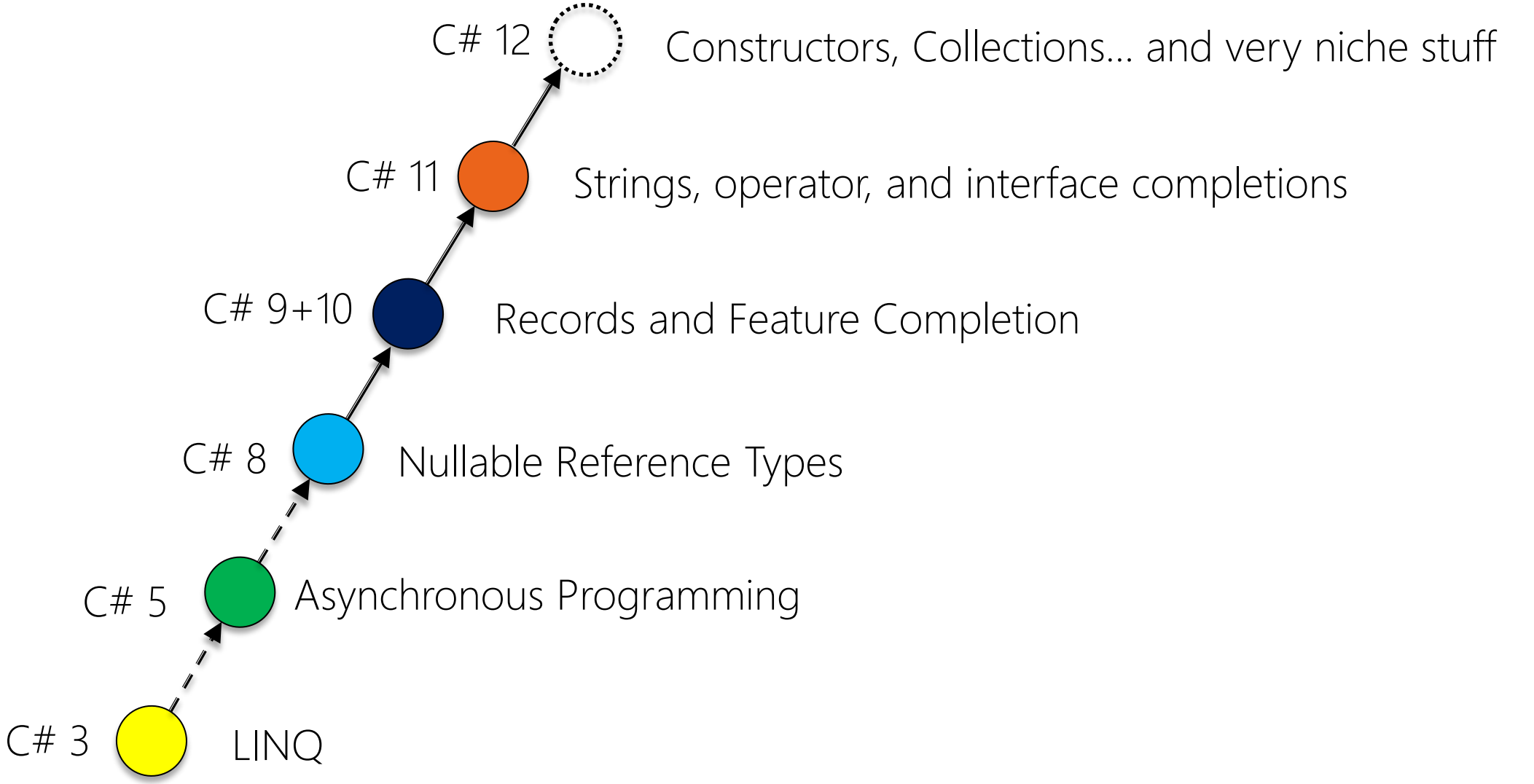
- C# 11 introduces the new **>>>** called *unsigned right shift operator*

```
int x = -8;
int y = x >> 2;
int z = x >>> 2;
```

Agenda

- What's New in C# 11?
- **Newest Features in C# 12**
  - **Introduction**
  - Object-Oriented Improvements
  - Collection Improvements
  - Method Improvements
  - Namespaces and Usings
- *Questions and Discussion*

Major Evolutions of C#



Agenda

- What's New in C# 11?
- **Newest Features in C# 12**
  - Introduction
  - **Object-Oriented Improvements**
  - Collection Improvements
  - Method Improvements
  - Namespaces and Usings
- Questions and Discussion

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

- What's New in C# 11?
- **Newest Features in C# 12**
  - Introduction
  - Object-Oriented Improvements
  - **Collection Improvements**
  - Method Improvements
  - Namespaces and Usings
- *Questions and Discussion*

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

- What's New in C# 11?
- **Newest Features in C# 12**
  - Introduction
  - Object-Oriented Improvements
  - Collection Improvements
  - **Method Improvements**
  - Namespaces and Usings
- *Questions and Discussion*

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

- What's New in C# 11?
- **Newest Features in C# 12**
  - Introduction
  - Object-Oriented Improvements
  - Collection Improvements
  - Method Improvements
  - **Namespaces and Usings**

- *Questions and Discussion*

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

Summary

- What's New in C# 11?
- Newest Features in C# 12

- *Questions and Discussion*

- Slides and examples:

  https://github.com/wincubate/danske-bank-cs11-12


WINCUBATE
Jesper Gulmann Henriksen
PhD, MCT, MCSD, MCPD
Phone : +45 22 12 36 31
Email : jgh@wincubate.net
WWW : http://www.wincubate.net
Ringgårdsvej 4A
8270 Højbjerg
Denmark