The Hidden Gold of C#

Jesper Gulmann Henriksen
jgh@dotnet.coach

linkedin.com/in/jespergulmann/

youtube.dotnet.coach

Agenda

- Introduction
- Exceptions
- Pattern Matchings
- Collections
- LINQ
- Extensions
- Read-only Features
- Records *(if time permits)*
- Diagnostics and Debugging *(if time permits)*
- Summary

Exception Filters

- Exception filters facilitates the handling of exceptions matching a specific type and/or predicate

```
try
{
    Bank.TransferFunds(from, 200, to);
}
catch (InsufficientFundsException e) when (e.Account.IsVIP)
{
    // Handle VIP account
}
```

- Distinct clauses can match same exception type but with different conditions

Rethrowing Exceptions


```
try { ... }
catch (DivideByZeroException exception)
{
    throw exception;
}
```

or

```
try { ... }
catch (DivideByZeroException)
{
    throw;
}
```

.NET 6 Shorthands

- Keep an eye out for convenience methods being added as .NET evolves ☺

```
public class EmployeeRepository : IEmployeeRepository
{
    private readonly IList<Employee> _employees;
    ...
    public void Add(Employee employee)
    {
        ArgumentNullException.ThrowIfNull(employee);

        _employees.Add(employee);
    }
}
```

Throw Expressions

- In C# 6 one could not easily just throw an exception in an expression-bodied member
- C# 7 allows **`throw`** expressions as subexpressions
  - Also outside of expression-bodied members..!

```
public class EmployeeRepository : IEmployeeRepository
{
    private readonly IList<Employee> _employees;
    ...
    public void Add( Employee employee ) =>
        _employees.Add(employee ??
            throw new ArgumentNullException(nameof(employee)));
}
```

- Note that a **`throw`** expression does not have an expression type as such…

Agenda

- Introduction
- Exceptions
- Pattern Matchings
- Collections
- LINQ
- Extensions
- Read-only Features
- Records *(if time permits)*
- Diagnostics and Debugging *(if time permits)*
- Summary

Pattern-matching Additions

- C# 7 and 8 introduced a total of 6 patterns

- C# 9 introduces 6 additional patterns or enhancements:

• Type patterns	*Type*	e.g. **`int`**
• Negation patterns	not *P1*	e.g. **`not null`**
• Parenthesized patterns	(*P*)	e.g. **`(string)`**
• Conjunctive patterns	*P1 and P2*	e.g. **`A and (not B)`**
• Disjunctive patterns	*P1 or P2*	e.g. **`int or string`**
• Relational patterns	*P1* < *P2*	e.g. **`< 87`**
	P1 <= *P2*	e.g. **`<= 87`**
	P1 > *P2*	e.g. **`> 87`**
	P1 >= *P2*	e.g. **`>= 87`**

To Be or Not To Be Null

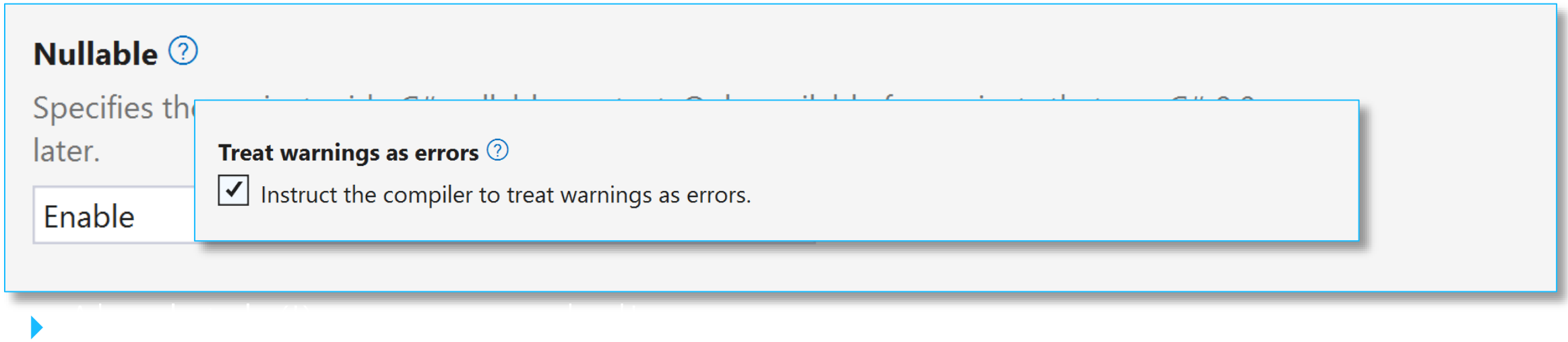
- At last(!) we are allowed both positive and negative pattern assertions

```
Person p = new() { ...};

if (p is null)
{
    Console.WriteLine("p is null");
}
else
{
    Console.WriteLine("p is not null");
}
```

- This is in fact superior to ==

Nullability Checks + Treat Warnings As Errors



Positional Patterns

- Positional patterns use deconstructors for matching

```
Album album = new Album(
    "Depeche Mode",
    "Violator",
    new DateTime(1990, 3, 19)
);

string description = album switch
{
    Album(_, string s, int age) when age >= 25 => $"{s} is vintage <3",
    Album(_, string s, int age) when age >= 10 => $"{s} is seasoned",
    Album(_, string s, _) => $"{s} is for youngsters only! ;-)"
};
```

- Can be simplified using `var`

The Edge of Pattern Matching?

- What to do if pattern matching is clumsy and essentially “fails”?

```
List<object> mixOfObjects = new() { true, 87, "Hello World", 176.0 };

ObjectHandler handler = new ObjectHandler();

foreach (object o in mixOfObjects)
{
    // Perform distinct handling depending upon the runtime type of o
}
```

- Answer: **`dynamic`** to the rescue…! ☺

Agenda

- Introduction
- Exceptions
- Pattern Matchings
- Collections
- LINQ
- Extensions
- Read-only Features
- Records *(if time permits)*
- Diagnostics and Debugging *(if time permits)*
- Summary

Indices

- The ^ operator describes the end of the sequence

```
string[] elements = new string[]
{
   "Hello", "World", "Booyah!", "Foobar"
};

Console.WriteLine(elements[^1]);
Console.WriteLine(elements[^0]); // ^0 == elements.length
Index i = ^2;
Console.WriteLine(elements[i]);
```

- Indices are captured by a new **System.Index** type
  - Can be manipulated using variables etc. as any other type

Ranges

- The .. operator specifies (sub)ranges using indices *i* and *j*
  - `i..j` Full sequence (start is inclusive, end is exclusive)
  - `i..` Half-open sequence (start is inclusive)
  - `..i` Half-open sequence (end is exclusive)
  - `..` Entire sequence (equivalent to `0..^0`)

```
foreach (var s in elements[0..^2])
{
    Console.WriteLine( s );
}

Range range = 1..;
```

- Ranges are captured by a new `System.Range` type
  - Can be manipulated using variables etc. as any other type

Supported Types

`string`	Indices	Ranges
`Array`	Indices	Ranges
`List<T>`	Indices	
`Span<T>`	Indices	Ranges
`ReadOnlySpan<T>`	Indices	Ranges

- Any type that provides an indexer with a `System.Index` or `System.Range` parameter (respectively) explicitly supports indices or ranges

- Compiler will implement some implicit support for indices and ranges

PriorityQueue in System.Collection.Generic

- "New Collection on the Block" in .NET 6

```
PriorityQueue<Connection, TimeOnly> pq = new();
pq.Enqueue(new Connection("vm-dev-1"), new TimeOnly(15, 57, 46, 231));

while (pq.TryDequeue(out Connection? conn, out TimeOnly time))
{
    Console.WriteLine($"{conn} last connected at {time.ToLongTimeString()}");
}
```

- Implements the classical "Min-Heap" priority queue
  - Dequeues highest priority element
  - Can supply own priority and comparer if needed

Agenda

- Introduction
- Exceptions
- Pattern Matchings
- Collections
- LINQ
- Extensions
- Read-only Features
- Records *(if time permits)*
- Diagnostics and Debugging *(if time permits)*
- Summary

LINQ Additions in .NET 6 Overview

- `ElementAt<T>` and `ElementAtOrDefault<T>`
  - New support for `Index`
- `Take<T>`
  - New support for `Range`

- `XxxOrDefault<T>`
  - New support for supplying default
- `Zip<T>`
  - New support for three enumerables

- New `Chunk<T>` method
- New `DistinctBy<T>`, `MinBy<T>` and `MaxBy<T>` methods
- New `UnionBy<T>`, `IntersectBy<T>`, and `ExceptBy<T>`
- New `TryGetNonEnumeratedCount<T>`

LINQ Additions in .NET 6 for Movie Lovers ☺


```
IEnumerable<Movie> movies = new List<Movie>
{
    new("Total Recall", 2012, 6.2f),
    new("Evil Dead", 1981, 7.5f),
    new("The Matrix", 1999, 8.7f),
    new("Cannonball Run", 1981, 6.3f),
    new("Star Wars: Episode IV - A New Hope", 1977, 8.6f),
    new("Don't Look Up", 2021, 7.3f),
    new("Evil Dead", 2013, 6.5f),
    new("Who Am I", 2014, 7.5f),
    new("Total Recall", 1990, 7.5f),
    new("The Interview", 2014, 6.5f)
};
```

"List the 20th to 30th Fibonacci Number which 4 divides"

- `Fib(1) = 1`
- `Fib(2) = 1`
- `Fib(i) = Fib(i-2) + Fib(i-1)` for `i > 2`

- Everybody loves LINQ, but…

Agenda

- Introduction
- Exceptions
- Pattern Matchings
- Collections
- LINQ
- Extensions
- Read-only Features
- Records *(if time permits)*
- Diagnostics and Debugging *(if time permits)*
- Summary

C# 3: "Good old" Extension Methods

- C# 3 introduced extension methods with LINQ

```
static class DateTimeExtensions
{
    public static string ToMyTimestamp(this DateTime dt) =>
        dt.ToString("yyyy-MM-dd HH:mm:ss.fff");
}
```

- Compiler allows syntactic sugar to "preserve illusion" that method is invoked on the extended type itself

```
DateTime dt = DateTime.Now;
Console.WriteLine(dt.ToMyTimestamp());
```

Extending Interfaces is Very Powerful!

- LINQ works by extending **`IEnumerable<T>`**

```
static class EnumerableExtensions
{
    public static IEnumerable<T> Sample<T>(
        this IEnumerable<T> sequence,
        int frequency
    )
    {
        ...
    }
}
```

- Why sequence now has this capability...
- "Roll you own LINQ"

C# 6: Collection Initializer Extensions

- Collection initializers work if
  - Type implements `IEnumerable<T>`
  - Type has an `Add()` method
- But what about `Queue<T>`, `Stack<T>`, …, custom types from libraries?
- C# 6 answered our prayers by *extension collection initializers*..!

C# 9 Extension Enumerables

- In C# 9 it is possible to create an "extension implementation" of **IEnumerable<T>** for a third-party type
  - **foreach** now respects extension **GetEnumerator<T>** methods

```
static class SequenceExtensions
{
    public static IEnumerator<T> GetEnumerator<T>( this Sequence<T> t )
    {
        SequenceElement<T>? current = t.Head;
        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }
}
```

C# 7 Extension Deconstructors

- A perfect companion to pattern matching third-party type:

```
static class MyPersonExtensions
{
    public static void Deconstruct(this Person person,
        out int length, out string fullName)
    {
        fullName = $"{person.FirstName} {person.LastName}";
        length = fullName.Length;
    }
}
```

- Excellent for constructing custom, reusable query pattern matches

Supported Types

`string`	Indices	Ranges
`Array`	Indices	Ranges
`List<T>`	Indices	
`Span<T>`	Indices	Ranges
`ReadOnlySpan<T>`	Indices	Ranges

- Any type that provides an indexer with a `System.Index` or `System.Range` parameter (respectively) explicitly supports indices or ranges
  - Not possible via extension methods, however

- Compiler will implement some implicit support for indices and ranges

Example: Custom Data Structure

- `SequencePacker<T>` stores sequences of elements of type T in a compressed form. More precisely, the sequence

  42 87 87 87 87 11 22 22 87 99

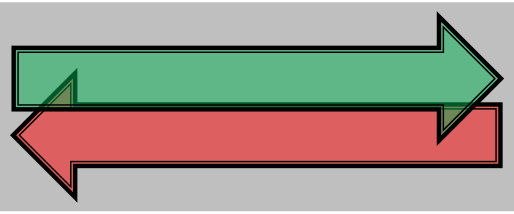
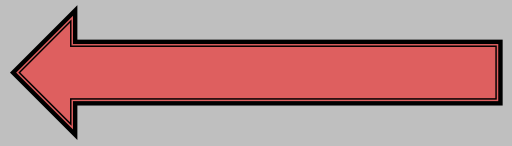
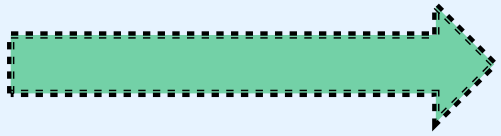
- is stored internally as a list of `Node<T>` elements as follows:

  (**42**,1) (**87**,4) (**11**,1) (**22**,2) (**87**,1) (**99**,1)

Agenda

- Introduction
- Exceptions
- Pattern Matchings
- Collections
- LINQ
- Extensions
- Read-only Features
- Records *(if time permits)*
- Diagnostics and Debugging *(if time permits)*
- Summary

`in` Parameter Modifier

Modifier	Effect	Description
		Copies argument to formal parameter
`ref`		Formal parameters are synonymous with actual parameters. Call site must also specify **`ref`**
`out`		Parameter cannot be read. Parameter must be assigned. Call site must also specify **`out`**
`in`		Parameter is "copied". Parameter cannot be modified! Call site can optionally specify **`in`**. ~ "**`readonly ref`**"

`in` Parameter Modifier

- It can be passed as a reference by the runtime system for performance reasons

```
double CalculateDistance( in 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);
}
```

- The call site does not need to specify `in`
- Can call with constant literal -> Compiler will create variable

```
Point3D p1 = new Point3D { X = -1, Y = 0, Z = -1 };
Point3D p2 = new Point3D { X = 1, Y = 2, Z = 3 };
double d = CalculateDistance(p1, p2));
```

Readonly Structs

- Define immutable structs for performance reasons

```
readonly struct Point3D
{
    public double X { get; }
    public double Y { get; }
    public double Z { get; }

    public Point3D( double x, double y, double z ) { ... }

    public override string ToString() => $"({X},{Y},{Z})";
}
```

- Compiler generates more optimized code for these values
- Also applies to **`record struct`**

Read-only Members for Structs

- C# 7.2 allowed the **`readonly`** modifier on structs
- C# 8 makes this more fine-grained

```
struct Point3D
{
    public Point3D(double x, double y, double z) { ... }
    public readonly override string ToString() =>
        $"({X},{Y},{Z}) at distance {CalculateDistance()} from (0,0,0)";
    public readonly double CalculateDistance(in Point3D other = default)
    {
        double xDiff = X - other.X;
        double yDiff = Y - other.Y;
        double zDiff = Z - other.Z;
        return Sqrt(xDiff * xDiff + yDiff * yDiff + zDiff * zDiff);
     }
}
```

Beware!

- Marking a non-readonly method with `readonly` forces compiler to make a copy!

`private protected` Access Modifier

- **`private protected`**
  - Is visible to containing types
  - Is visible to derived classes in the same assembly

```
public class ClassInOtherAssembly
{
    private protected int X { get; set; }

    public void Print() => Console.WriteLine(X);
}
```

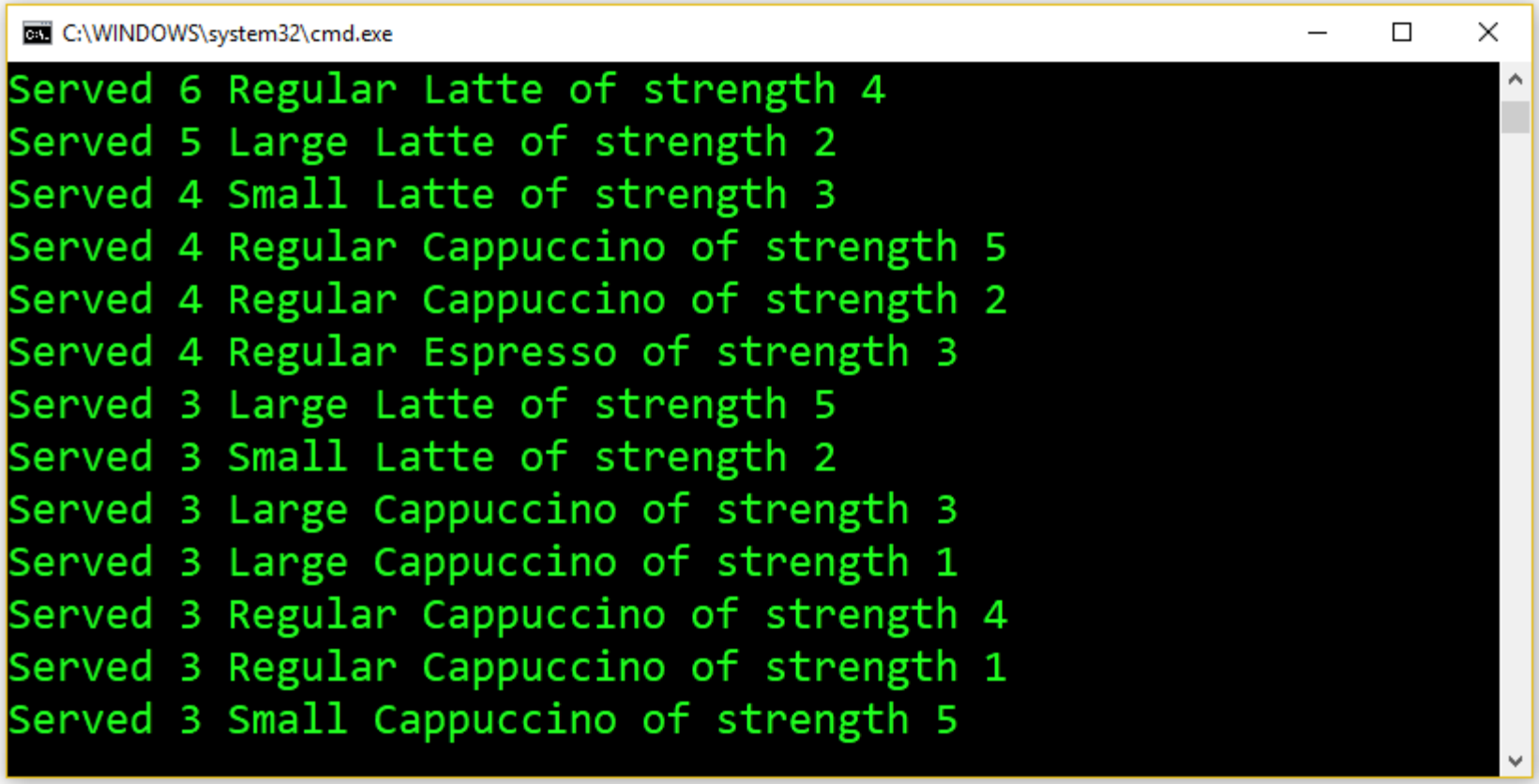
- **`protected internal`**
  - Is visible to types in same assembly
  - Is visible to derived classes (in same or other assemblies)

Agenda

- Introduction
- Exceptions
- Pattern Matchings
- Collections
- LINQ
- Extensions
- Read-only Features
- Records *(if time permits)*
- Diagnostics and Debugging *(if time permits)*
- Summary

Dictionary Problems?

- Do a `PrintSummary()` method which provides a summary like:



Solution: Records as Keys

- Define a `PrintSummary()` method outputting a number of strings to the console as illustrated, i.e.:
  - Sort *first* by the count of specific coffee combinations served (from high to low)
  - Sort *secondly* by kind (from first to last)
  - Sort *thirdly* by size within that kind (from largest to smallest)
  - Use the strength as the *final* sort criterion (from strongest to weakest).

Agenda

- Introduction
- Exceptions
- Pattern Matchings
- Collections
- LINQ
- Extensions
- Read-only Features
- Records *(if time permits)*
- Diagnostics and Debugging *(if time permits)*
- Summary

Caller Info Attributes Revisited

- C# 5.0 introduced three types of caller info attributes
  - [CallerMemberName]
  - [CallerFilePath]
  - [CallerLineNumber]

```
void Log(
     [CallerMemberName] string? callerName = null,
     [CallerFilePath] string? callerFilePath = null,
     [CallerLineNumber] int callerLine = -1
)
{ ... }
```

- Applicable to default parameters
  - Compiler replaces values at compilation time

Caller Argument Expressions

- C# 10 adds a `CallerArgumentExpression` attribute

```
void Validate( bool condition,
    [CallerArgumentExpression("condition")] string? message = null)
{
    if (!condition)
    {
        throw new InvalidOperationException(
            $"Argument failed validation: {message}"
        );
    }
}
```

- Excellent for developer-centric logs etc.

Controlling Appearance in Debugger

- Many interesting way of visualizing data

- Easy-to-use and very simple are:
  - Overriding **`ToString()`**
  - `[DebuggerDisplay]`

Conditional Attributes on Methods

- Attributes can now be placed on both local (static!) and member methods and lambdas

```
void Main(string[] args)
{
    [Conditional("DEBUG")]
    static void PrintInfo(string s)
    {
        Console.WriteLine($"Debug: {s}");
    }

    PrintInfo("Start");
}
```

Revisiting Partial Methods


```
partial class Customer
{
    private string _name;
    public string Name
    {
        get { return _name; }
        set
        {
            OnNameChanging(value);
            _name = value;
            OnNameChanged();
        }
    }
    partial void OnNameChanging(string newName);
    partial void OnNameChanged();
}
```

New Features for Partial Methods

- Traditionally, partial methods suffered some restrictions:
  - ~~Cannot define access modifiers (implicitly private)~~
  - ~~Must have `void` return type~~
  - ~~Parameters cannot have the `out` modifier~~
- These restrictions are removed if
  - Can be annotated with explicit accessibility modifier (if consistent)
  - When explicit modifier, it must have a matching implementation

```
partial class Customer
{
    ...
    private partial bool OnNameChanging(string newName);
    public partial void OnNameChanged(out string oldName);
}
```

Summary

- Introduction
- Exceptions
- Pattern Matchings
- Collections
- LINQ
- Extensions
- Read-only Features
- Records *(if time permits)*
- Diagnostics and Debugging *(if time permits)*


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