

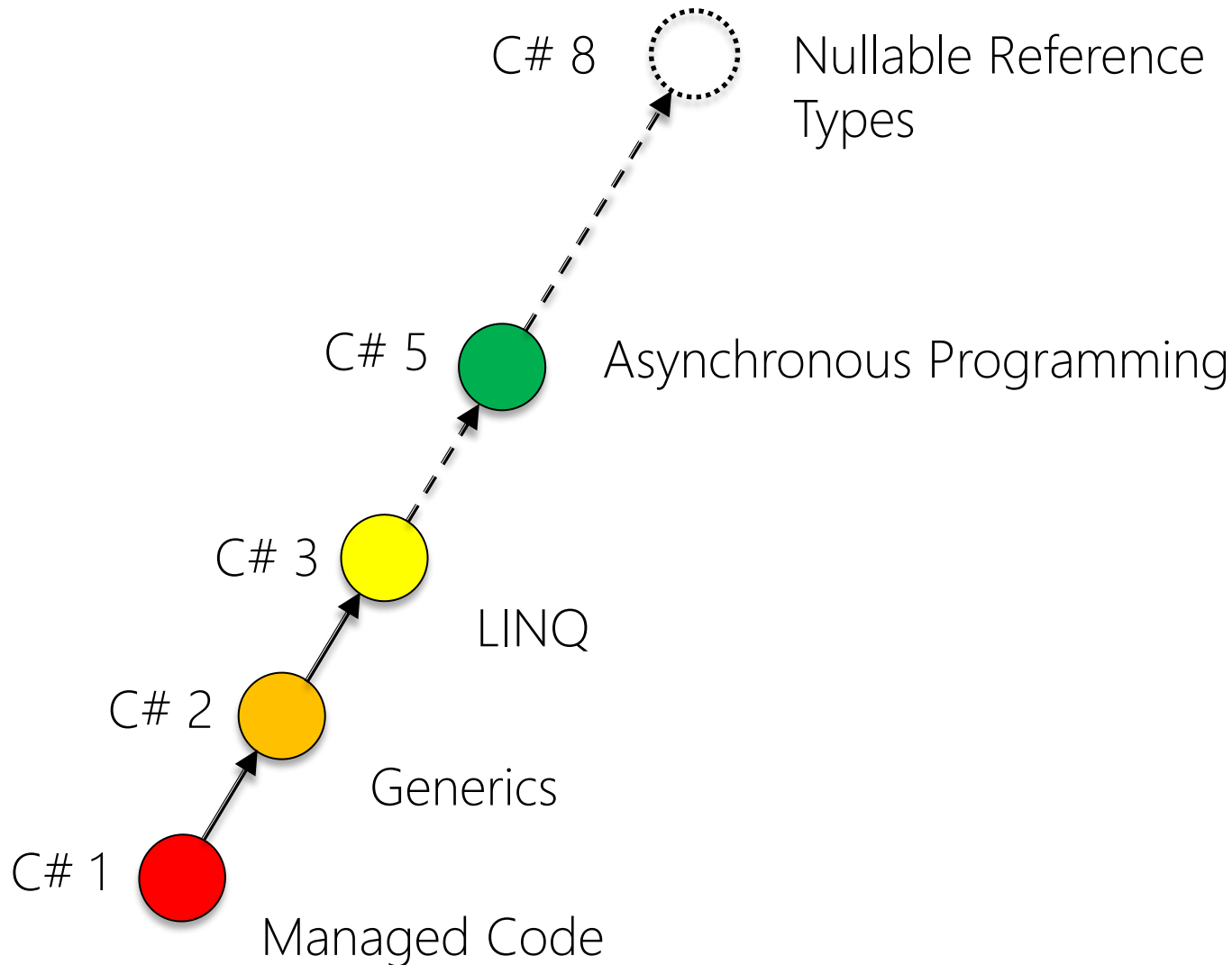
# Module 02:

## "An Introduction to C# 8"



**TEKNOLOGISK**  
**INSTITUT**

# Major Evolutions of C#



# Agenda

- ▶ Introduction
- ▶ **Nullable Reference Types**
- ▶ More Pattern Matching
- ▶ New Expressions
- ▶ Default Interface Implementation
- ▶ Asynchronous Streams and Disposables
- ▶ Statement Improvements
- ▶ Method Improvements
- ▶ Class and Struct Improvements
- ▶ Summary

# Null References:

## "The Billion-dollar Mistake"

*"I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."*

– Tony Hoare 2009

# Introducing Nullable Reference Types

- ▶ C# 8 allows declaring intent of reference types
  - *Nonnullable Reference Types*
    - A reference is not supposed to be null
  - *Nullable Reference Types*
    - A reference is allowed to be null

```
class Person
{
    public string  FirstName { get; } // Non-nullable string
    public string? MiddleName { get; } // Nullable string
    public string  LastName { get; }  // Non-nullable string

    ...
}
```

- ▶ Traditionally, C# reference types do not make this distinction!

# Static Analysis

- ▶ Produces compile-time static analysis warning when
  - Setting a **nonnullable** to null
  - Dereferencing a **nullable** reference

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

    public Person( string firstName ) => FirstName = firstName;

    int GetLengthOfMiddleName( Person p ) => p.MiddleName.Length;
}
```


# Null-forgiving Operator

- ▶ You can assert to the compiler that a reference is not null using the Null-forgiving Operator !

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

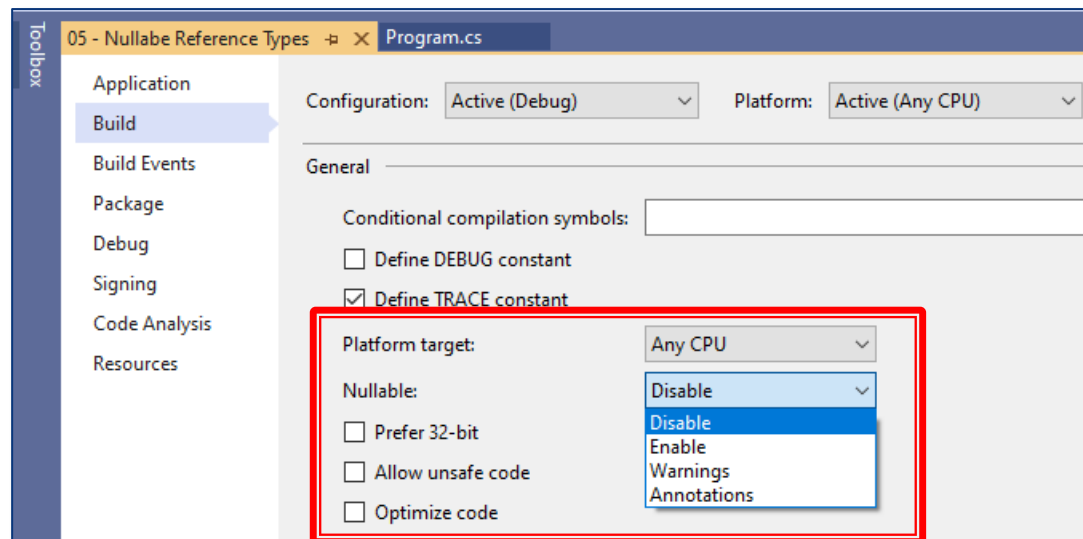
    public Person( string firstName ) => FirstName = firstName;

    int GetLengthOfMiddleName( Person p ) => p.MiddleName!.Length;
}
```



# Wait a Minute...!?

- ▶ Not Backwards Compatible with C# 7.x!
- ▶ Behavior can be controlled in Project Properties



- ▶ Nullable Contexts
  - Annotations
  - Warnings



# Annotations + Warning Contexts

- ▶ Can also be enabled/disabled locally by means of compiler directive **#nullable**
  - **enable / disable / restore**
  - **warnings / annotations**

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

    #nullable disable
    public Person( string firstName ) => FirstName = firstName;
    #nullable restore
}
```

# Agenda

- ▶ Introduction
- ▶ Nullable Reference Types
- ▶ **More Pattern Matching**
- ▶ New Expressions
- ▶ Default Interface Implementation
- ▶ Asynchronous Streams and Disposables
- ▶ Statement Improvements
- ▶ Method Improvements
- ▶ Class and Struct Improvements
- ▶ Summary

# Switch Expressions

- ▶ A new functionally-inspired **switch** expression

```
string? Choose( Employee employee ) =>  
    employee switch  
    {  
        SoftwareArchitect sa => $"Hello, Mr. Architect {sa.LastName}",  
        SoftwareEngineer se => "Please code!",  
        StudentProgrammer sp => $"Please get coffee, {sp.FirstName}",  
        _ => "Have a nice day... :-)"  
    }  
}
```

- ▶ Produces a value, so
  - no fallthrough!
  - **case** and **:** elements are replaced with **=>**
  - **default** case is replaced with a **\_**
  - bodies can only be expressions (not statements!)

# New Patterns for Matching

- ▶ C# 7 introduced three patterns for matching
  - Constant patterns      *c*      e.g. **null**
  - Type patterns      *T x*      e.g. **int x**
  - Var patterns      **var x**
  
- ▶ C# 8 introduces three additional patterns for matching
  - Property patterns      *Type{ p1: v1, ... , pn: vn }*      e.g. **{IsValid: false}**
  - Tuple patterns      *( x1, ... , xn )*      e.g. **(42, 87)**
  - Positional patterns      *Type( x1, ... , xn )*      e.g. **Album(s, age)**
  
- ▶ Moreover, in C# 8 patterns can now be “recursive”!

# Property Patterns

- ▶ Property patterns match member properties to values

```
string? Evaluate( SoftwareEngineer se ) =>
    se switch
    {
        { Level: Level.Lead } => $"{se.FullName} does great work",
        { Level: Level.Chief } => $"You da boss, {se.FullName}",
        null => "You're not even a software engineer, dude!",
        _ => $"Well done coding SOLID, {se.Level}... :-)"
    }
}
```

- ▶ Also works for multiple, simultaneous name-value pairs

# Property Patterns Variations

- ▶ Can in fact simultaneously match the type as well...

```
string? Evaluate( Employee employee ) =>  
    employee switch  
    {  
        SoftwareEngineer { Level: Level.Lead } => $"...",  
        SoftwareArchitect { Level: Level.Chief } => $"...",  
        _ => $"Well done making the company thrive... :-)"  
    }  
}
```

- ▶ Not tied to **switch** expressions: Also works for **is** etc.

# Tuple Patterns

- ▶ Tuple patterns use two or more values for matching

```
Hand left = GetRandomMember<Hand>();  
Hand right = GetRandomMember<Hand>();  
  
Outcome winner = (left, right) switch  
{  
    (Hand.Paper, Hand.Rock) => Outcome.Left,  
    (Hand.Paper, Hand.Scissors) => Outcome.Right,  
    (Hand.Rock, Hand.Paper) => Outcome.Right,  
    (Hand.Rock, Hand.Scissors) => Outcome.Left,  
    (Hand.Scissors, Hand.Paper) => Outcome.Left,  
    (Hand.Scissors, Hand.Rock) => Outcome.Right,  
    (_, _) => Outcome.Tie  
};
```

# 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**



# Agenda

- ▶ Introduction
- ▶ Nullable Reference Types
- ▶ More Pattern Matching
- ▶ **New Expressions**
- ▶ Default Interface Implementation
- ▶ Asynchronous Streams and Disposables
- ▶ Statement Improvements
- ▶ Method Improvements
- ▶ Class and Struct Improvements
- ▶ 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

- |                                      |         |        |
|--------------------------------------|---------|--------|
| ▶ <code>string</code>                | Indices | Ranges |
| ▶ <code>Array</code>                 | Indices | Ranges |
| ▶ <code>List&lt;T&gt;</code>         | Indices |        |
| ▶ <code>Span&lt;T&gt;</code>         | Indices | Ranges |
| ▶ <code>ReadOnlySpan&lt;T&gt;</code> | 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

# Null-Coalescing Assignment

- ▶ A new `??=` operator completing the question mark feature landscape ☺
- ▶ Assigns `j` to `i` if `i` is `null`.

```
int? i = null;  
int? j = 42;  
int? k = 87;  
  
i ??= j; // i = i ?? j;  
i ??= k; // i = i ?? k;  
  
Console.WriteLine( i );
```

- ▶ Also works for reference types of course

# Interpolated Verbatim Strings

- ▶ C# 6 demanded that the string interpolation token **\$** preceded the verbatim token **@**, i.e. **\$@** when both were present
- ▶ C# 8 allows the token to appear in any order, i.e. **\$@** or **@\$** (with same semantics)

```
string directoryName = "Tmp";  
string fullPath1 = @$"C:\{directoryName}\readme.txt"; // Allowed in C# 6  
string fullPath2 = @$"C:\{directoryName}\readme.txt"; // Allowed in C# 8  
  
// fullPath1 == fullPath2
```

# Agenda

- ▶ Introduction
- ▶ Nullable Reference Types
- ▶ More Pattern Matching
- ▶ New Expressions
- ▶ **Default Interface Implementation**
- ▶ Asynchronous Streams and Disposables
- ▶ Statement Improvements
- ▶ Method Improvements
- ▶ Class and Struct Improvements
- ▶ Summary

# Default Interface Members

- ▶ Allow better backwards compatibility in interfaces

```
interface ILogger
{
    void Log(LogLevel level, string message);
    void Log(Exception ex) => Log(LogLevel.Error, ex.ToString());
}
```

```
class FileLogger : ILogger
{
    public void Log(LogLevel level, string message) { ... }
}
```

```
class ConsoleLogger : ILogger
{
    public void Log(LogLevel level, string message) { ... }
    public void Log(Exception ex) { ... }
}
```



# Static Members in Interfaces

- Somewhat controversial...

```
interface ILogger
{
    static string ProduceExceptionLog(Exception exception) =>
        $"Exception occurred: {exception.Message}. " +
        $"Call stack size: {exception.StackTrace?.Length ?? 0}";

    void Log(LogLevel level, string message);
    void Log(Exception ex) =>
        Log(LogLevel.Error, ProduceExceptionLog(ex));
}
```

```
class ConsoleLogger : ILogger
{
    ...
    public void Log(Exception ex) =>
        ... ILogger.ProduceExceptionLog(exception) ...
    ...
}
```

# C# 8 Interfaces vs. Classes

- ▶ Default interface members cannot be invoked on concrete classes – only through the interface!
  - Bears resemblance to explicit interface implementation
- ▶
- ▶ But...
- ▶ Static members can have access modifiers in interfaces..!
  - Default access modifier on interface members: **public**
  - Default access modifier on class members: **private**

# Agenda

- ▶ Introduction
- ▶ Nullable Reference Types
- ▶ More Pattern Matching
- ▶ New Expressions
- ▶ Default Interface Implementation
- ▶ **Asynchronous Streams and Disposables**
- ▶ Statement Improvements
- ▶ Method Improvements
- ▶ Class and Struct Improvements
- ▶ Summary

# New C# 8.0 Async Features

- ▶ Use new types only in .NET Core 3.x (and .NET 5 later)
- ▶ Async Enumerables a.k.a. "Async Streams"
- ▶ **await foreach** keyword
- ▶ Async Disposables
- ▶ **await using** keyword

# IEnumerable<T>

- ▶ The traditional **IEnumerable<T>** designates a sequence for use with **foreach** or LINQ.

```
namespace System.Collections.Generic
{
    interface IEnumerable<out T> : IEnumerable
    {
        IEnumerator<T> GetEnumerator();
    }
}
```

```
interface IEnumerator<T>
{
    T Current { get; }
    bool MoveNext();
    void Reset();
}
```

# IAsyncEnumerable<T>

- ▶ `IAsyncEnumerable<T>` designates an asynchronous sequence for use with **await foreach**

```
namespace System.Collections.Generic
{
    interface IAsyncEnumerable<out T>
    {
        IAsyncEnumerator<T> GetEnumerator(CancellationToken cts = default);
    }
}
```

```
interface IAsyncEnumerator<T>
{
    T Current { get; }
    ValueTask<bool> MoveNextAsync();
}
```

# Example of Async Stream

```
async IEnumerable<string> GetWordsAsync(string[] urls)
{
    foreach (var url in urls)
    {
        WebClient wc = new WebClient();
        string result = await wc.DownloadStringTaskAsync(url);
        yield return result.Substring(0, 256);
    }
}
```

```
string[] urls = new string[] { ... };
await foreach (string s in GetWordsAsync(urls))
{
    Console.WriteLine(s);
}
```

# IDisposable

- ▶ Traditionally, .NET has **IDisposable** interface built-in for implementing Dispose Pattern

```
public interface IDisposable
{
    void Dispose();
}
```

- ▶ The **using** keyword can be applied to ensure **Dispose()** is always invoked.



# IAsyncDisposable

- ▶ Now, for asynchronous disposal .NET Core 3.x has **IDisposableAsync** interface built-in for implementing Dispose Pattern

```
public interface IAsyncDisposable
{
    ValueTask DisposeAsync();
}
```

- ▶ The **await using** keyword can be applied to ensure **DisposeAsync()** is always invoked.

# Example of Async Disposables

```
class Connection : IAsyncDisposable
{
    public async ValueTask DisposeAsync()
    {
        ...
        await DisconnectAsync();
        ...
    }
}
```

```
await using (var connection = new Connection())
{
    await connection.ConnectAsync();
    // Do stuff...
}
```

# Agenda

- ▶ Introduction
- ▶ Nullable Reference Types
- ▶ More Pattern Matching
- ▶ New Expressions
- ▶ Default Interface Implementation
- ▶ Asynchronous Streams and Disposables
- ▶ **Statement Improvements**
- ▶ Method Improvements
- ▶ Class and Struct Improvements
- ▶ Summary

# Using Declarations

- ▶ Instruct compiler to dispose at the end of the scope

```
using FileStream inStream = File.OpenRead(sourceFilePath);
using FileStream outStream = File.Create(destinationFilePath);
using DeflateStream compress = new DeflateStream(
    outStream, CompressionMode.Compress );

for (int i = 0; i < inStream.Length; i++)
{
    compress.WriteByte((byte)inStream.ReadByte());
}

// inStream, outStream, compress are disposed here at the end of the scope!
```

- ▶ Also works for the new async disposables **await using**!

# Agenda

- ▶ Introduction
- ▶ Nullable Reference Types
- ▶ More Pattern Matching
- ▶ New Expressions
- ▶ Default Interface Implementation
- ▶ Asynchronous Streams and Disposables
- ▶ Statement Improvements
- ▶ **Method Improvements**
- ▶ Class and Struct Improvements
- ▶ Summary

# Static Local Functions

- ▶ Local functions introduced in C# 7 can in C# 8 be marked as **static** to prevent capturing of variables

```
class A
{
    public int Counter { get; set; }

    public void DisplayStatus()
    {
        static string FormatStatus( int c ) => $"Counter is {c}";

        Console.WriteLine( FormatStatus( Counter ) );
    }
}
```

# Agenda

- ▶ Introduction
- ▶ Nullable Reference Types
- ▶ More Pattern Matching
- ▶ New Expressions
- ▶ Default Interface Implementation
- ▶ Asynchronous Streams and Disposables
- ▶ Statement Improvements
- ▶ Method Improvements
- ▶ **Class and Struct Improvements**
- ▶ Summary

# 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 {DistanceFrom()} from (0,0,0)";
    public readonly double DistanceFrom(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);
    }
}
```



# Disposable Ref Structs

- ▶ Ref structs were introduced in C# 7.2 as never-heap allocated structs
  - But could not implement interfaces

```
ref struct Point3D
{
    public Point3D(double x, double y, double z) { ... }
    ...
    public void Dispose() { ... }
}
```

```
using Point3D point = new Point3D(1, 0, 1);
Console.WriteLine(point.ToString());

// point.Dispose() is invoked here!
```

- ▶ C# 8 compiler checks explicitly for **Dispose()** method as special-case treatment workaround for **IDisposable**

# Unmanaged Constructed Types

- ▶ In C# 8 a constructed value type is deemed *unmanaged* if it contains fields of unmanaged types only

```
public struct Position<T>
{
    public T X;
    public T Y;
}
```

```
unsafe static void HandleUnmanaged<T>( T t ) where T : unmanaged
{
    T* p = &t;
    // ...
}

HandleUnmanaged( new Position<int> ); // Compiles!
```

# Stackalloc in Nested Expressions

- ▶ In C# 7 **stackalloc** was allowed in declaring expressions only
- ▶ In C# 8 you can use **stackalloc** in nested expressions
  - But only if it is of type **Span<T>** or **ReadOnlySpan<T>**

```
Span<int> numbers = stackalloc[] { 11, 22, 33, 42, 44, 87, 88 };  
int index = numbers.IndexOfAny(stackalloc[] { 42, 87 });  
  
Console.WriteLine(index);
```

- ▶ This is a slight performance enhancement for computations

# Summary

- ▶ Introduction
- ▶ Nullable Reference Types
- ▶ More Pattern Matching
- ▶ New Expressions
- ▶ Default Interface Implementation
- ▶ Asynchronous Streams and Disposables
- ▶ Statement Improvements
- ▶ Method Improvements
- ▶ Class and Struct Improvements



WINCUBATE

Jesper Gulmann Henriksen

PhD, MCT, MCSD, MCPD

Phone : +45 22 12 36 31

Email : [jgh@wincubate.net](mailto:jgh@wincubate.net)

WWW : <http://www.wincubate.net>

Ringgårdsvej 4A

8270 Højbjerg

Denmark