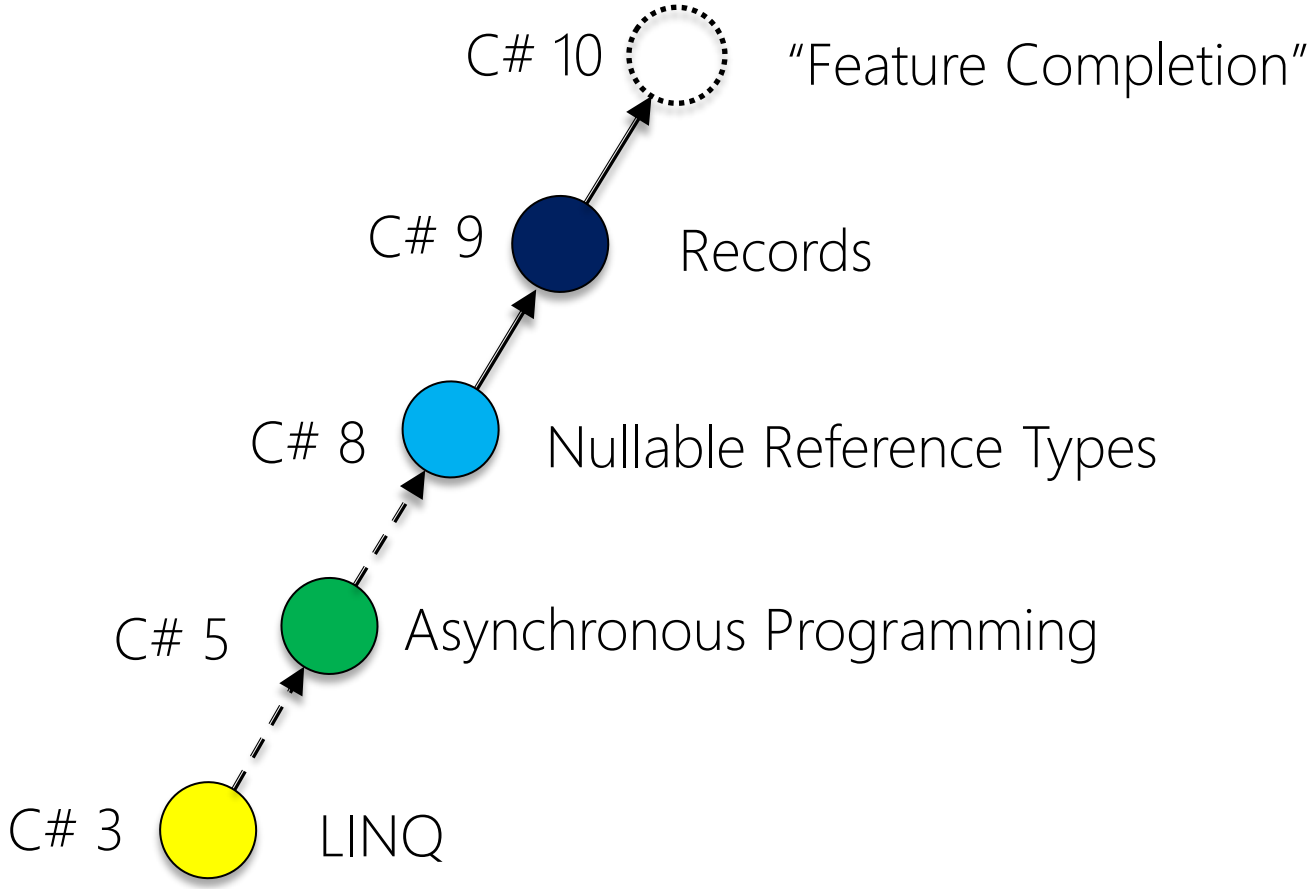
Module 04:

"An Introduction to C# 10"

Major Evolutions of C#



Agenda

- Introduction
- **Namespace Improvements**
- Object-Oriented Improvements
- Expression Improvements
- Statement Improvements
- Other Additions
- Not Really Interesting for Mortals
- Summary

File-scoped Namespace Declarations

- In spirit of the `using` directive, the `namespace` declarations have been "horizontally optimized" similarly

```
namespace Wincubate.CS10.Shapes;

interface IShape
{
    double Area { get; }
}

class Rectangle : IShape
{
    ...
}
```

- How often do you have multiple namespaces in same file?

Global Using Directives

- Project-wide **`using`** directives are now supported

```
global using System.Text.Json;

namespace Wincubate.CS10.Shapes;

record Rectangle(double Width, double Height) : IShape
{
    public double Area => Width * Height;

    public string Serialize() => JsonSerializer.Serialize(this);
}
```

- Also works for **`using static`**

Implicit Usings

- Implicit `usings` are enabled in project file for **new** projects

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <RootNamespace>Wincubate.CS10.A</RootNamespace>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

- .NET libraries supply default implicit `usings`

Custom Implicit Usings

- You can configure your custom implicit **`usings`**

```
<Project Sdk="Microsoft.NET.Sdk">
  ...
 <ItemGroup Condition="'$(ImplicitUsings)' == 'enable'">
    <Using Remove="System.Threading" />
    <Using Include="System.Text.Json" />
    <Using Include="Wincubate.CS10.Shapes" />
    <Using Static="true" Include="System.Console"/>
  </ItemGroup>

</Project>
```

- An alternative to **`GlobalUsings.cs`** or similar

Agenda

- Introduction
- Namespace Improvements
- **Object-Oriented Improvements**
- Expression Improvements
- Statement Improvements
- Other Additions
- Not Really Interesting for Mortals
- Summary

C# 9 Object-oriented Topology

- Value Types:
- `struct`

- Reference Types:
- `class`
  - `record`

- Anonymous Types

C# 10 Object-oriented Topology

- Value Types:
- `struct`
  - `record struct`

- Reference Types:
- `class`
  - `record class`
- Anonymous Types

Record Structs and Record Classes

- Use **`record struct`** for "value-type records"

```
Money m1 = new(87, 25);
Money m2 = new(87, 25);

Console.WriteLine(m1 == m2);

record struct Money( int Euro, int Cents)
{
    public int TotalCents => Euro * 100 + Cents;
}
```

- Use **`record`** or **`record class`** for "reference-type records"

Comments on Record Structs

- **`record class`**
  - Immutable for primary constructor parameters
  - Mutable for other properties
- **`record struct`**
  - Mutable for primary constructor parameters
  - Mutable for other properties

- However, thinking back to C# 7.x:

- **`readonly record struct`**
  - Immutable for primary constructor parameters
  - Other properties are not allowed to be mutable!

C# 10 Additions to Structs

- Default constructors and initializers are allowed in C# 10

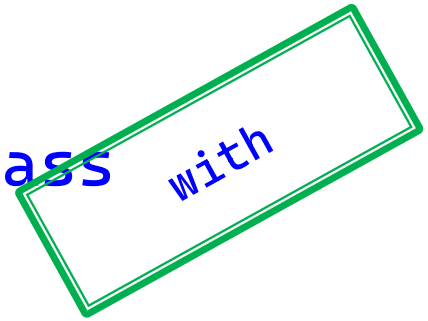
```
struct Money
{
    public int Euro { get; set; } = 99;
    public int Cents { get; set; } = 99;

    public Money()
    {
        Euro = 1;
        Cents = 0;
    }
}
```

- Note: Beware of relation to **`default`** keyword and/or non-default constructors

C# 9 Non-destructive Mutation

- Value Types:
- `struct`

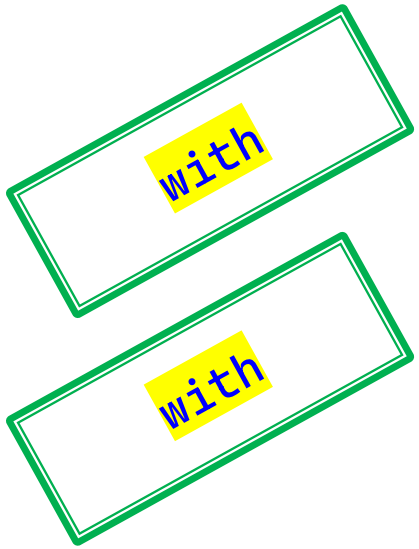
- Reference Types:
- `class`
- `record class`



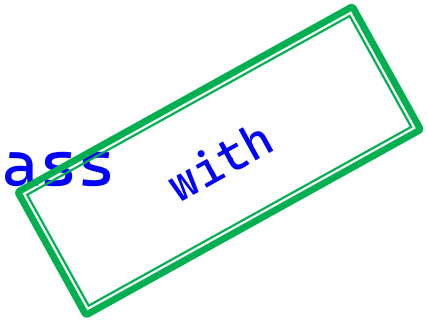
- Anonymous Types

C# 10 Non-destructive Mutation

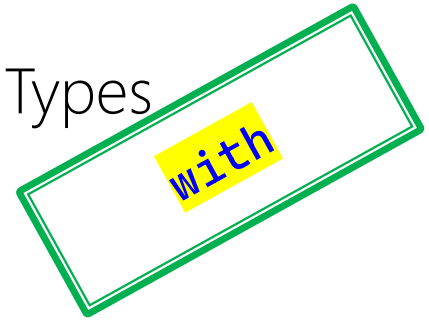
- Value Types:
  - `struct`
  - `record struct`



- Reference Types:
  - `class`
  - `record class`



- Anonymous Types



Non-destructive Mutation Extended

- C# 9 **with** expressions allowed for all structs in C# 10

```
struct Money
{
    public int Euro { get; set; };
    public int Cents { get; set; };
}
```

```
Money m1 = new(87, 25);
Money m2 = m1 with { Cents = 87 };
```

- C# 9 **with** expressions also allowed for anonymous types

```
var p1 = new { FirstName = "Bruce", LastName = "Wayne" };
var p2 = p1 with { LastName = "Campbell" };
```

Fixing the C# 9 Record Bug ☺

- The bug with **`ToString()`** and inheritance from C# 9?
- "Fixed" in C# 10 by marking synthesized method as sealed

```
abstract record Record(
    string Artist, string Name,
    DateTime? ReleaseDate = default)
{
    sealed public override string ToString() =>
        $"{Artist}: \"{Name}\" [{ReleaseDate}]";
}
```

```
record Album(
    string Artist, string Name,
    DateTime? ReleaseDate, int NumberOfDiscs = 1)
    : Record(Artist, Name, ReleaseDate)
{
    // New ToString() is *not* synthesized here
}
```

Agenda

- Introduction
- Namespace Improvements
- Object-Oriented Improvements
- **Expression Improvements**
- Statement Improvements
- Other Additions
- Not Really Interesting for Mortals
- Summary

Pattern-matching Enhancements

- C# 7, 8, and 9 introduced a total of 12 patterns and enhancements

- C# 10 introduces just one: Extended Property Pattern
  - *Type{ p1.p2: v }*

```
record class Company(string Name, Company? OwnedBy = default);
```

```
var query = companies
    .Where(c => c is { OwnedBy.OwnedBy.Name: "Sharp10" })
    ;
```

Lambda Natural Type Inference

- Lambda expressions are now given a natural type when inferred

```
var parseToInt = (string s) => int.Parse(s);
int i = parseToInt("87");

var writeInt = (int i) => Console.WriteLine(i);
writeInt(176);
```

- Type needs to conform to either
  - a generic `Func` , or
  - a generic or non-generic `Action`
- What do you think will happen if it does not…?

- Are there any other quirky cases…?

Lambda Explicit Return Type

- Lambda expressions may declare a return type when the compiler can't infer it

```
var choose = object (bool b) => b ? 1 : "two";
Console.WriteLine(choose(false));
```

- Makes lambda expressions as similar to methods and local functions as possible
- Easier to use lambda expressions without declaring a variable of a delegate type
- Work seamlessly with the new ASP.NET Core Minimal APIs.

Attributes on Lambda Expressions

- In C# 10 attributes can be added to
  - a lambda expression
  - its parameters
  - Its return value

```
// Attribute on lambda expression itself
Func<string, int> parse = [Developer("JGH")] (s) => int.Parse(s);

// Attribute on parameter
Action<string?> info =([Developer("JGH")] s) =>      Console.WriteLine(s);

// Attribute on return type
Func<int, int> compute = [return: Developer("JGH")] (i) => i + 87;
```

Constant Interpolated Strings

- Strings interpolated from constants can now be const

```
static class DeveloperInfoConstants
{
    public const string Name = "JGH";
    public const string Company = "Wincubate ApS";

    public const string Message = $"{Name} / {Company}";
}
```

- Finally! Now interpolated strings can be used in
  - Attributes
  - API Routes
  - …

String Interpolation Optimized

- In C# 10 string interpolation has been optimized at a number of places, e.g.
  - `Debug.Assert()`

```
int i = 0;
while( i < 100 )
{
    // Does not compute string if not necessary
    Debug.Assert(i >= 0, $"{DateTime.Now} - {GetLogMessage()}");
    i++;
}
```

- API is open for "everyone" to build their own special interpolation handlers
  - If You Must: See Lab 04.4 for tutorial ☺

Agenda

- Introduction
- Namespace Improvements
- Object-Oriented Improvements
- Expression Improvements
- **Statement Improvements**
- Other Additions
- Not Really Interesting for Mortals
- Summary

Mixed Deconstruction

- In C# 9 deconstructions we could either
  - Declare and initialize *all* new variables, or
  - Assign to *all* existing variables

- In C# 10 this restriction is relaxed

```
record Person(string FirstName, string LastName)
{
    public string FullName => $"{FirstName} {LastName}";
}
```

```
string f = string.Empty;
(f, string l) = person;
```

Agenda

- Introduction
- Namespace Improvements
- Object-Oriented Improvements
- Expression Improvements
- Statement Improvements
- **Other Additions**
- Not Really Interesting for Mortals
- 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}"
        );
    }
}
```

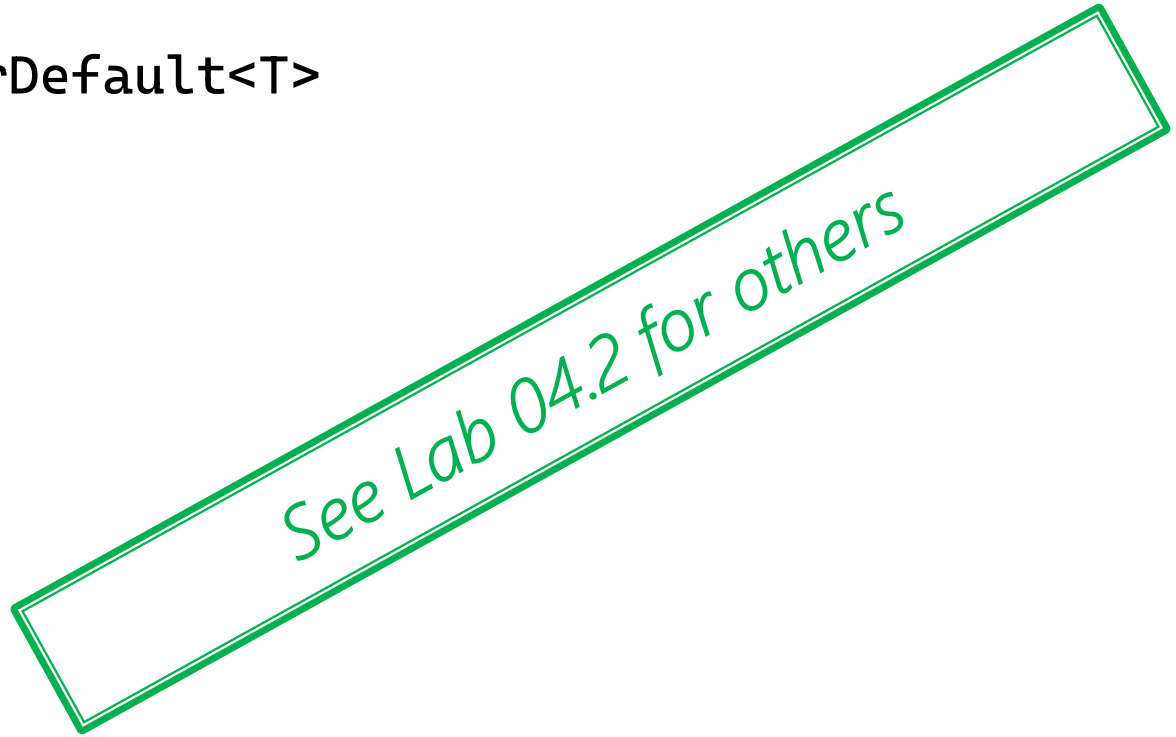
- Excellent for developer-centric logs etc.

LINQ Additions in .NET 6

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

- ***`Xxx`*`OrDefault<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>`**



Agenda

- Introduction
- Namespace Improvements
- Object-Oriented Improvements
- Expression Improvements
- Statement Improvements
- Other Additions
- **Not Really Interesting for Mortals**
- Summary

Improved Definite Assignment

- More accurate warnings for definite assignment and null-state analysis

```
if ((c != null && c.GetValue(out object obj1)) == true)
{
    representation = obj1.ToString(); // Undesired error
}
if (c?.GetValue(out object obj2) == true)
{
    representation = obj2.ToString(); // Undesired error
}
if (c?.GetValue(out object obj3) ?? false)
{
    representation = obj3.ToString(); // Undesired error
}
```

Enhanced Line Pragmas

- C# 10 supports a new format for the **`#line`** pragma
- You likely won't use the new format, but you'll see its effects in e.g. Razor

```
#line (1, 1) - (5, 60) 10 "partial-class.g.cs"
/*34567*/int b = 0;
```

- For the "uninitiated": ☺ ☺ ☺
  - *The **`#line`** directive might be used in an automated, intermediate step in the build process. For example, if lines were removed from the original source code file, but you still wanted the compiler to generate output based on the original line numbering in the file, you could remove lines and then simulate the original line numbering with **`#line`**.*

More Granular AsyncMethodBuilder Attribute

- Since C# 7: Add **`AsyncMethodBuilder`** attribute to a type that can be an async return type
  - Available in System.Runtime.CompilerServices
  - specifies the type that builds the async method implementation when the specified type is returned from an async method

- In C# 10: **`AsyncMethodBuilder`** allowed on individual async methods

```
[AsyncMethodBuilder(typeof(MyAsyncTaskMethodBuilder<>))]
public async Task<R> ComputeAsync()
{
    await Task.Delay(1000);
    return new R("Yay!");
}
```

Summary

- Introduction
- Namespace Improvements
- Object-Oriented Improvements
- Expression Improvements
- Statement Improvements
- Other Additions
- Not Really Interesting for Mortals


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