Module 01:

"A Very Quick Recap of C# 8, 9, and 10"

WINCUBATE

Major Evolutions of C# – The Story So Far
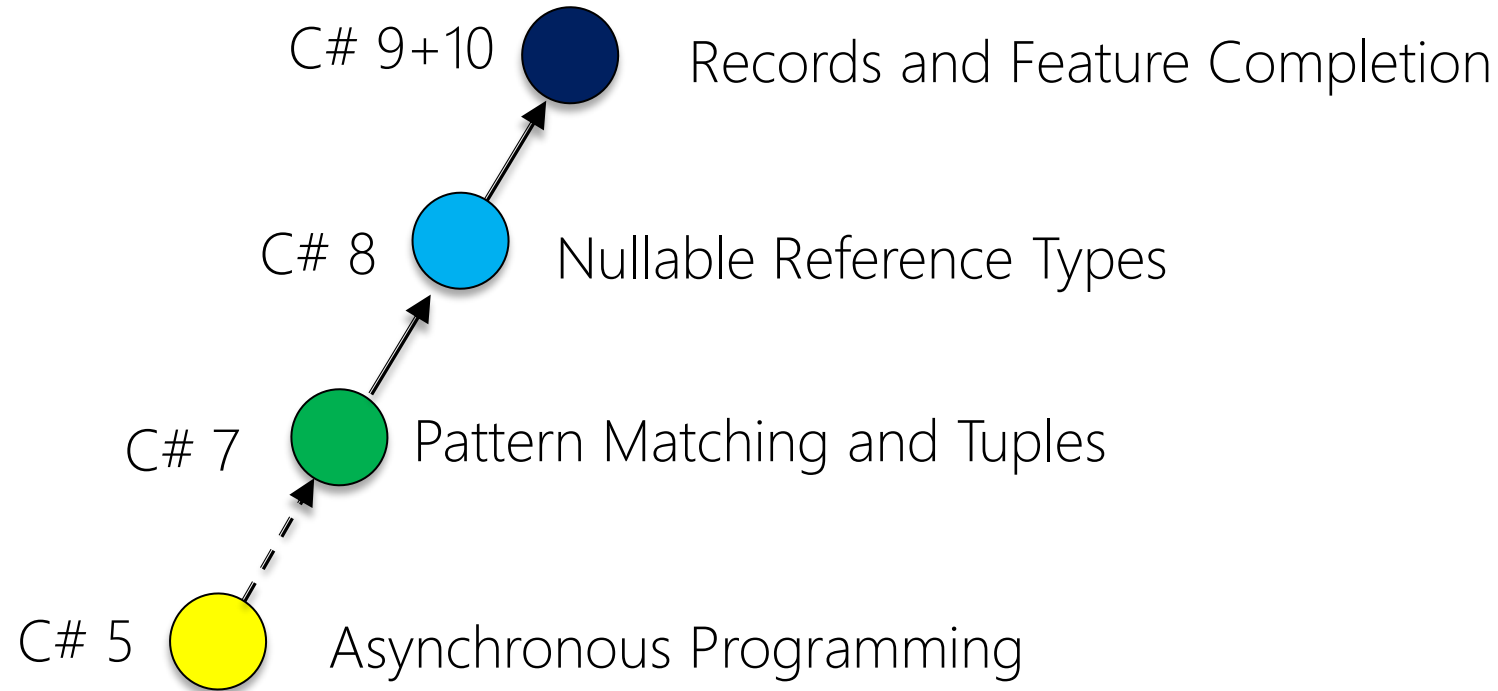
C# 9+10 — Records and Feature Completion

C# 8 — Nullable Reference Types

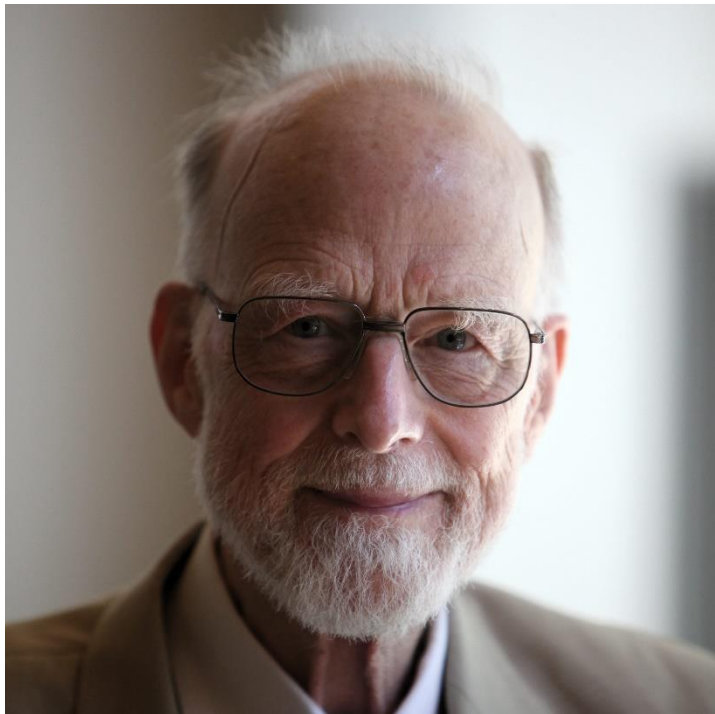C# 7 — Pattern Matching and Tuples

C# 5 — Asynchronous Programming

Agenda

- Introduction
- **Nullable Reference Types**
- Pattern Matching and Switch Expressions
- Record Classes and Structs
- Indices and Ranges
- Readonly Features
- 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 <u>intent</u> of reference types
- *Nonnullable Reference Types*
  - A reference is not supposed to be null
- *Nullable Reference Types*
  - A reference is allowed to be null

```csharp
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

```csharp
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* **!**

```csharp
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;
}
```
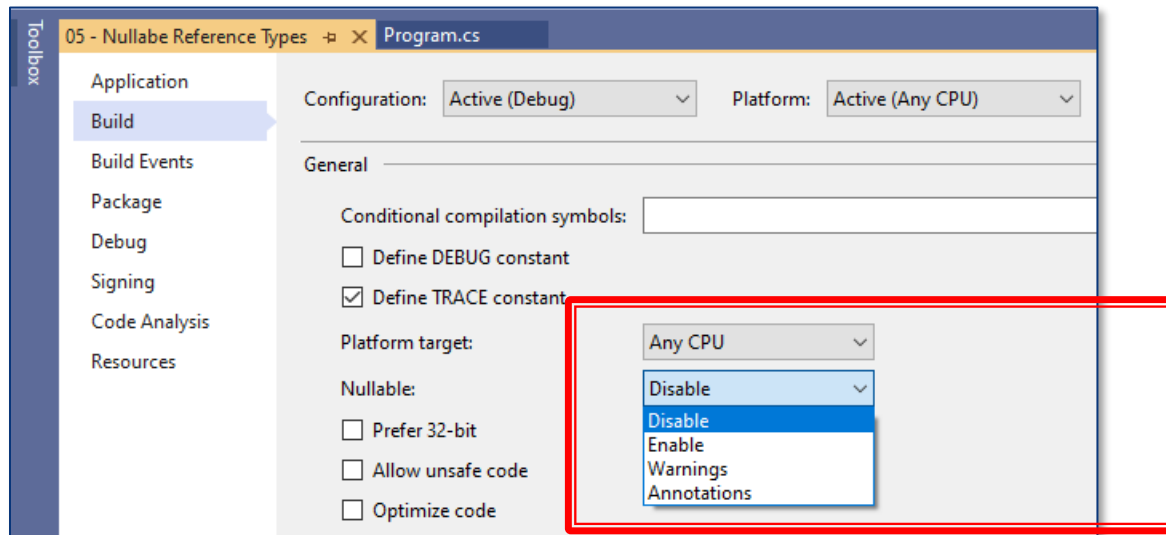
Wait a Minute…!?

- **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**

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

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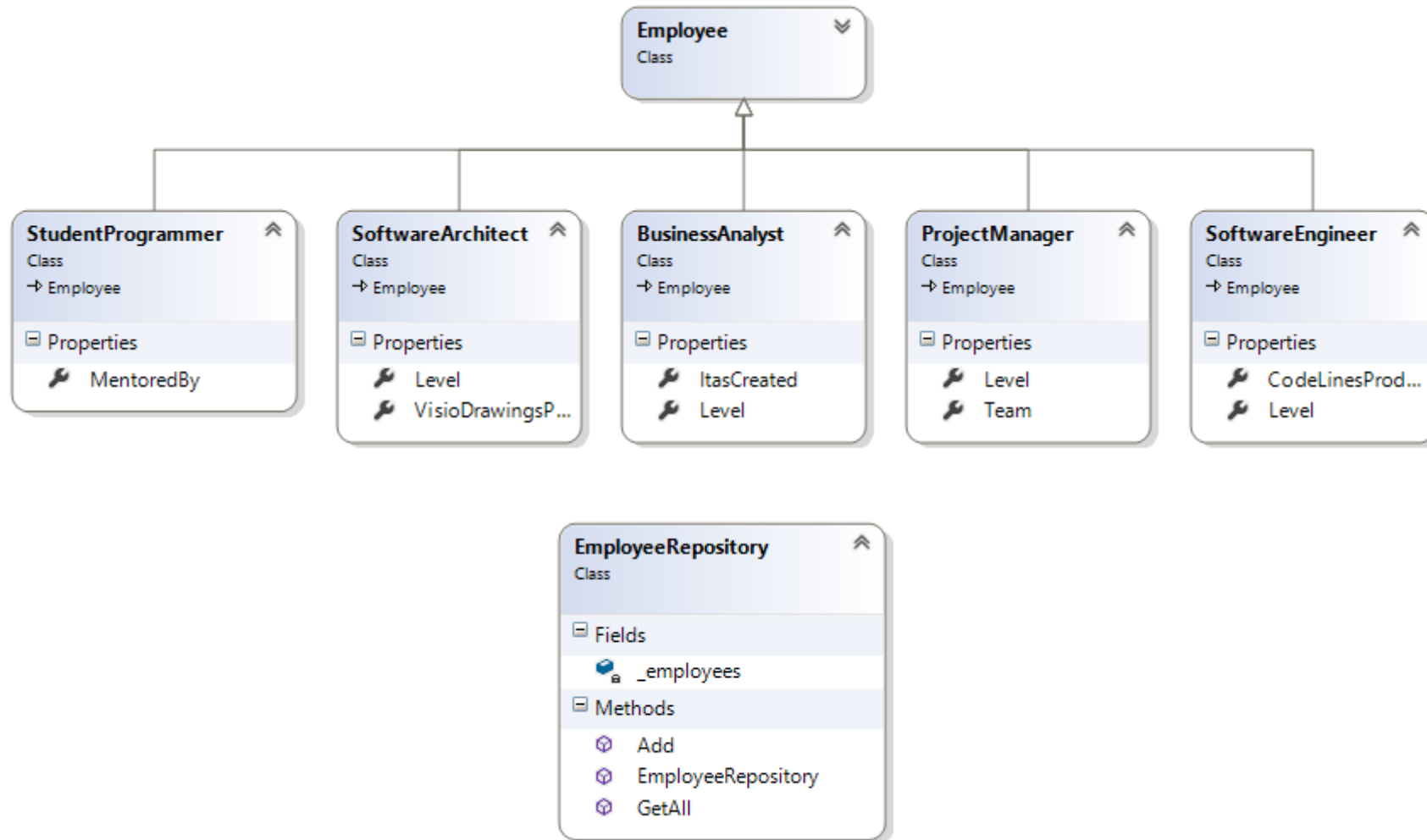
Agenda

- Introduction
- Nullable Reference Types
- **Pattern Matching and Switch Expressions**
- Record Classes and Structs
- Indices and Ranges
- Readonly Features
- Summary

Example: Employee

Pattern Matching with **is**

- Three types of patterns for matching in C# 7
  - Constant patterns   *c*   e.g.   **null**
  - Type patterns   *T x*   e.g.   **int** x
  - Var patterns   **var** *x*
- Matches and/or captures to identifiers to nearest surrounding scope
- More patterns are introduced in later C# versions

```csharp
foreach (Employee e in all)
{
    if (e is SoftwareEngineer se)
    {
        WriteLine($"{se.FullName} has produced {se.CodeLinesProduced} lines of C#");
    }
}
```

- The **is** keyword is now compatible with patterns

Type Switch with Pattern Matching

▸ Can switch on <u>any </u>type
  • Case clauses can make use of patterns and new **when** conditions

```csharp
Employee e = ...;
switch (e)
{
    case SoftwareArchitect sa:
        WriteLine($"{sa.FullName} plays with Visio");
        break;
    case SoftwareEngineer se when se.SoftwareEngineerLevel == SoftwareEngineerLevel.Lead:
        WriteLine($"{se.FullName} is a lead software engineer");
        break;
    case null:
    default:
        break;
}
```

Cases are no longer disjoint – evaluated sequentially!

Switch Expressions

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

```csharp
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!)

C# 8 Pattern Matching Enhancements

▸ 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 are now be "**compositional**"!

Property Patterns

▸ Property patterns match member properties to values

```csharp
string Evaluate( SoftwareEngineer se ) =>
    se switch
    {
        { Level: SoftwareEngineerLevel.Lead } => $"{se.FullName} does great work",
        { Level: SoftwareEngineerLevel.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...

```csharp
string Evaluate( Employee employee ) =>
    employee switch
    {
        SoftwareEngineer { Level: SoftwareEngineerLevel.Lead } => $"...",
        SoftwareArchitect { Level: SoftwareArchitectLevel.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

```csharp
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`

C# 9 Pattern Matching Enhancements

▸ 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`

Type Patterns

▸ This is more or less only a compiler-theoretic enhancement
  • But now it "mixes better" with the new or compound patterns

```csharp
object o1 = 87;
object o2 = "Yeah!";


var t = (o1, o2);


if (t is (int, string))
{
    Console.WriteLine("o1 is an int and o2 is a string");
}
```

Negation Patterns

▸ At last(!) we are allowed negative pattern assertions

```csharp
public void DoStuff(object o)
{
    if( o is not null )
    {
        Console.WriteLine(o);
    }
}
```

Parenthesized Patterns

▸ This is simply a means to disambiguate parsing

- Carries not semantic meaning in itself

```
public string WhatIsIt(object o) =>
    o switch
    {
        (((string))) => "string",
        (((int))) => "int",
        _ => "Something else :-)",
    };
```

▸ But has tremendous significance for the other patterns following shortly...

Conjunctive Patterns

▸ Conjunctive patterns specify an **and** between patterns

```
string evalution = employee switch
{
    (not ProjectManager) and (not StudentProgrammer) =>
        "Codes a little",
    _ => "Probably codes a bit more..."
};


Console.WriteLine($"{employee.FullName}: {evalution}");
```

Disjunctive Patterns

▸ Disjunctive patterns specify an **or** between patterns

```csharp
IEnumerable<object> elements = new List<object>
{
    42, "Yay", 87.0, "Nay", 12.7m
};


foreach (var o in elements)
{
    Console.WriteLine(o switch {
        int or double or decimal => $"{o} is a number",
        _ => "Not a number..."
    });
}
```

Relational Patterns

▸ The relational patterns are all the "usual" comparisons

- **<** , **<=** , **>** , **>=**

```csharp
int temperature = int.Parse(Console.ReadLine());
string forecast = temperature switch
{
    <= 0 => "Freezing...",
    < 12 => "Autumn-like",
    <= 19 => "Spring-ish",
    <= 40 => "Summer!",
    _ => "Death Valley?"
};
```

▸ Note that there is no **=>**

C# 10 Pattern Matching Enhancements

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

▸ C# 10 introduces just one: <mark>Extended</mark> 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" })
    ;
```

Agenda

- Introduction
- Nullable Reference Types
- Pattern Matching and Switch Expressions
- **Record Classes and Structs**
- Indices and Ranges
- Readonly Features
- Summary

Init-only Setters

▸ Restricted setter only allowing initialization:

```csharp
public sealed class Album
{
    public string Artist { get; init; }
    public string AlbumName { get; init; }
    public DateTime ReleaseDate { get; init; }


    ...
}
```

▸ Can have usual visibility on init-only setter
▸ Can not have both **init** and **set**...!

Records

▸ Records are simpler, immutable classes

```
record Person(string FirstName, string LastName);
```

▸ Defines init-only properties with "Primary Constructors"

▸ Can have additional properties + methods, of course

```
record Album(string Artist, string AlbumName, DateTime ReleaseDate)
{
    public int Age
    {
        get { ... }
    }
}
```

Built-in Features of Records

▸ Overrides
  - **ToString()**
  - **Equals()** (Implements **IEquatable\<T\>**)
  - **GetHashCode()**
  - **==** and **!=**

▸ What about **ReferenceEquals**?

▸ Supplies built-in deconstructors

Mutation-free Copying

▸ Additional keyword: Create copies using **with**

```csharp
Album album = new Album("Prince",
                        "Purple Rain",
                        new DateTime(1984, 11, 02));


Album renamed = album with
{
    Artist = "The Artist Formerly Known as..."
};
```

▸ Does not mutate source record
   • Copies and replaces

Records and Inheritance

▸ Almost all OO aspects are identical to classes
- Visibility, parameters, etc.
- But Records and Classes cannot mix inheritance!

▸ Can override and change built-in method overrides, if needed

C# 9 Object-oriented Topology

▸ <u>Value Types:</u>

 ▸ `struct`

▸ <u>Reference Types:</u>

 ▸ `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`**
  - <mark>Mutable</mark> for primary constructor parameters
  - Mutable for other properties

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

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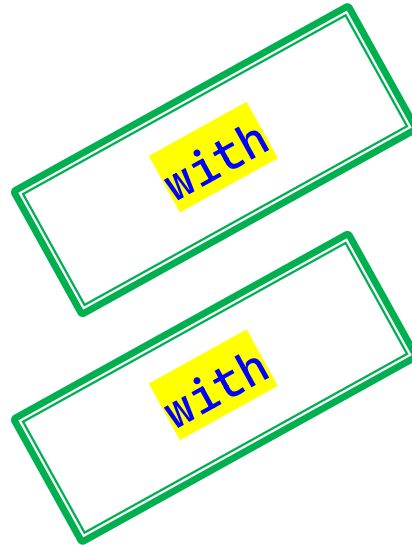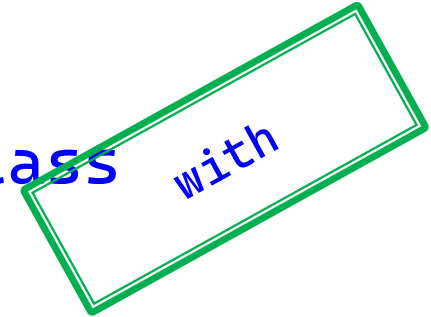 - Other properties are not allowed to be mutable!

C# 10 Non-destructive Mutation

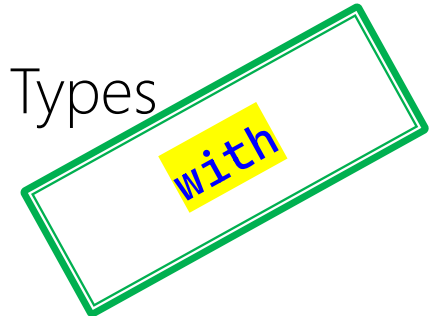**Value Types:**

- `struct`

- `record struct`

`with`

`with`

Reference Types:

- `class`

- `record class`   `with`

- Anonymous Types

`with`

Agenda

- Introduction
- Nullable Reference Types
- Pattern Matching and Switch Expressions
- Record Classes and Structs
- **Indices and Ranges**
- Readonly Features
- 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`)

```csharp
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

Agenda

▸ Introduction

▸ Nullable Reference Types

▸ Pattern Matching and Switch Expressions

▸ Record Classes and Structs

▸ Indices and Ranges

▸ **Readonly Features**

▸ 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

```csharp
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})";
}
```

▸ Can always be passed as **in**
▸ 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

```csharp
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!

- Calling non-readonly methods within `readonly` methods can force compiler to make a copy of locals etc.!

Summary

- Introduction
- Nullable Reference Types
- Pattern Matching and Switch Expressions
- Record Classes and Structs
- Indices and Ranges
- Readonly Features