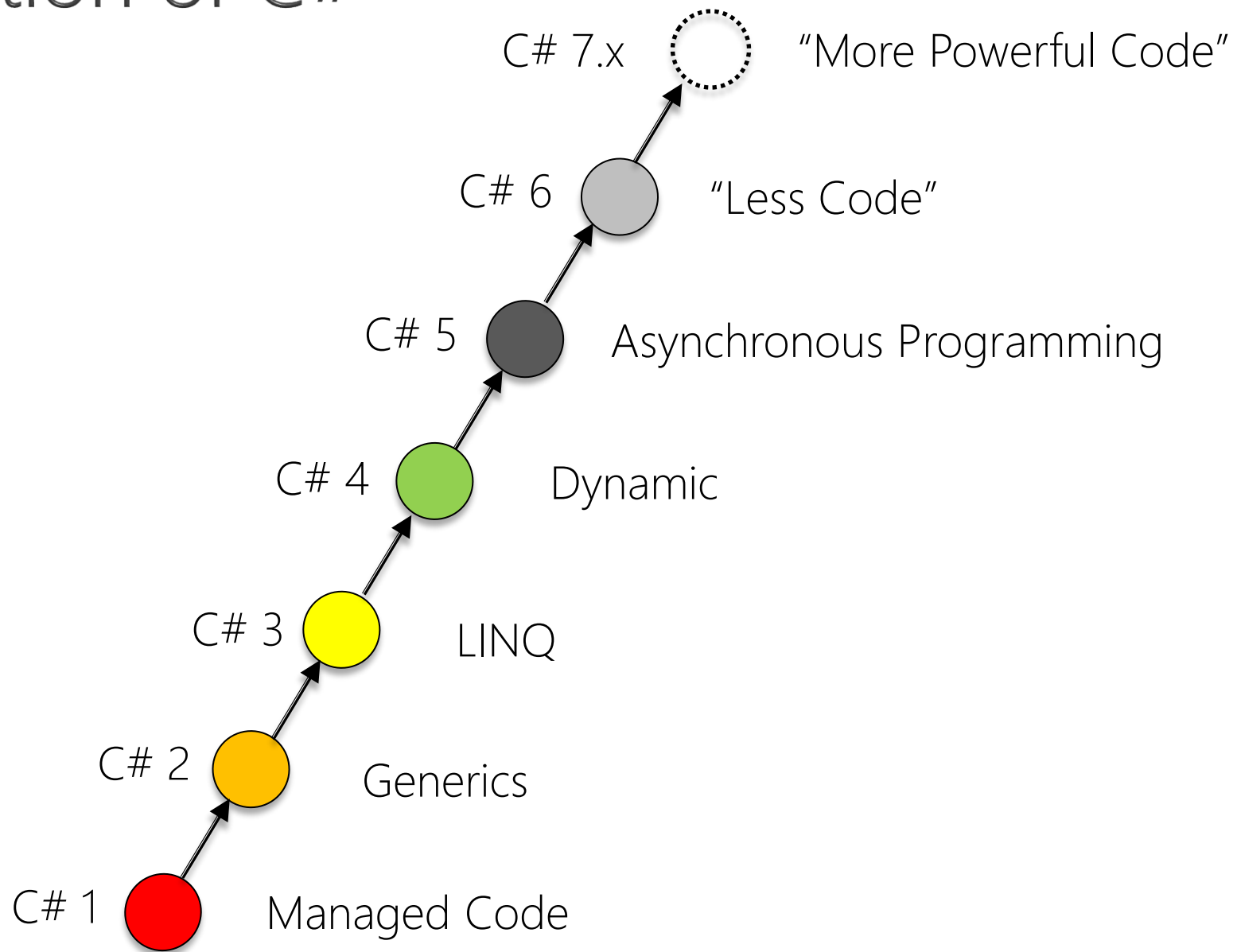


# Module 01:

"Recapping  
C# 7.0, 7.1, 7.2, and 7.3"



# Evolution of C#



# Agenda

- ▶ Introduction
- ▶ **Value Tuples and Syntax**
- ▶ Pattern Matching
- ▶ Method Improvements
- ▶ Expression Improvements
- ▶ Other C# 7.1 Additions
- ▶ Other C# 7.2 Additions
- ▶ Other C# 7.3 Additions

# Introducing Tuples

- ▶ Not the `Tuple<T1,T2>` type already in .NET 4.0
  - Instead it is a value type with dedicated syntax

```
(int, int) FindVowels( string s )  
{  
    int v = 0;  
    int c = 0;  
    foreach (char letter in s)  
    {  
        ...  
    }  
    return (v, c);  
}
```

```
string input = ReadLine();
```

```
var t = FindVowels(input);
```

```
WriteLine($"There are {t.Item1} vowels and  
{t.Item2} consonants in \"{input}\"");
```

# Syntax, Literals, and Conversions

- ▶ Can be easily converted / deconstructed to other names

```
var (vowels, cons) = FindVowels(input);  
(int vowels, int cons) = FindVowels(input);
```

```
WriteLine($"There are {vowels} vowels and {cons} consonants in ... ");
```

```
(int vowels, int cons) FindVowels( string s )  
{  
    var tuple = (v: 0, c: 0);  
    ...  
    return tuple;  
}
```

- ▶ Tuples can be supplied with descriptive names
- ▶ Mutable and directly addressable
- ▶ Built-in: `ToString()` + `Equals()` + `GetHashCode()` (but not `==` until C# 7.3)

# Inferred Tuple Names

## (aka. Tuple Projection Initializers 😊)

- ▶ Tuple names are redundant when they can be inferred from the context
  - Similar to what the anonymous types of C# 3.0

```
struct Equipment
{
    public string Console { get; set; }
    public int Controllers { get; set; }
    public bool IsVREnabled { get; set; }
}
```

```
Equipment e = new Equipment { ... };
var tuple = (e.Console, e.Controllers);

Console.WriteLine( tuple.Console );
```

- ▶ Compiles in C# 7.1, but not in C# 7.0

# Custom Tuple Deconstruction

- ▶ Can be easily deconstructed to individual parts

```
(int vowels, int cons) = FindVowels(input);
```

- ▶ Custom types can also be supplied with a *destructor* with out parameters

```
public class Employee
{
    ...
    public void Deconstruct( out string firstName, out string lastName )
    {
        firstName = FirstName;
        lastName = LastName;
    }
}
```

```
Employee elJefe = new Employee { ... };
var (first, last) = elJefe;
WriteLine(first);
```

- ▶ Works for two or more deconstruction parts
  - Deconstructors can be overloaded

# Extension Deconstructors

- ▶ A powerful feature is that deconstructors can be extension methods

```
static class AlbumExtensions
{
    public static void Deconstruct(this Album album,
        out string summary, out int age)
    {
        summary = $"{album.AlbumName}\" by {album.Artist}";
        DateTime today = DateTime.Now;
        age = today.Year - album.ReleaseDate.Year -
            (album.ReleaseDate.DayOfYear < today.DayOfYear ? 0 : 1);
    }
}
```

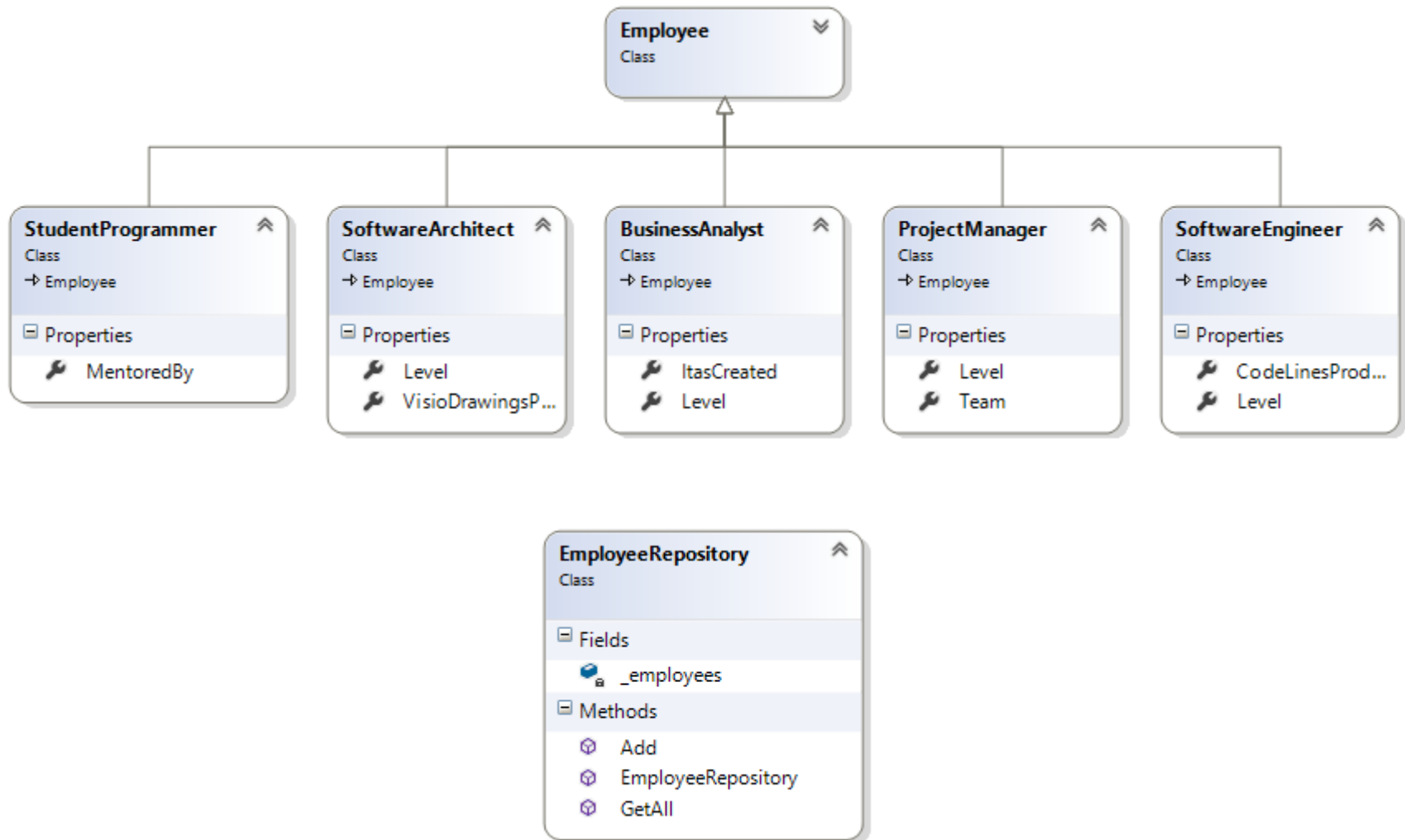
- ▶ See Lab 01.2



# Agenda

- ▶ Introduction
- ▶ Value Tuples and Syntax
- ▶ **Pattern Matching**
- ▶ Method Improvements
- ▶ Expression Improvements
- ▶ Other C# 7.1 Additions
- ▶ Other C# 7.2 Additions
- ▶ Other C# 7.3 Additions

# 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

```
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 any type
  - Case clauses can make use of patterns and new **when** conditions

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

- ▶ Cases are no longer disjoint – evaluated sequentially!

# Agenda

- ▶ Introduction
- ▶ Value Tuples and Syntax
- ▶ Pattern Matching
- ▶ **Method Improvements**
- ▶ Expression Improvements
- ▶ Other C# 7.1 Additions
- ▶ Other C# 7.2 Additions
- ▶ Other C# 7.3 Additions

# Local Functions

- ▶ Methods within methods can now be defined

```
(int vowels, int cons) FindVowels( string s )  
{  
    ...  
    foreach (char letter in s)  
    {  
        bool IsVowel( char letter )  
        {  
            ...  
        }  
        ...  
    }  
    return tuple;  
}
```

- ▶ Has some advantages
  - Captures local variables
  - Avoids allocations

# Ref Locals

- ▶ Can now create references in the style of C++
  - Similar to the **ref** modifier for parameters

```
int x = 42;  
ref int y = ref x;
```

```
x = 87;  
WriteLine(y);
```

# Ref Locals Reassignment

- ▶ C# 7.0 added references in the style of C++
- ▶ C# 7.3 completes ref locals by allowing them to be reassigned

```
int x = 42;  
int y = 87;  
ref int z = ref x; // Declaration and Initialization of z;  
  
x = 112;  
WriteLine($"z = {z}");  
  
z = ref y; // Reassignment of z;  
WriteLine($"z = {z}");
```



# Ref Conditionals

- ▶ C# 7.2 allows the well-known selection operator `?:` for refs

```
int x = 42;  
int y = 87;  
bool b = ...;  
  
ref int z = ref (b ? ref x : ref y);  
  
z = 112;  
  
Console.WriteLine( $"x={x}, y={y}, z={z}");
```

# Ref Returns

- ▶ Methods can now also return references

```
ref int FindMax( int[] numbers )
{
    int indexOfMax = 0;
    for (int i = 1; i < numbers.Length; i++)
    {
        if (numbers[i] > numbers[indexOfMax])
        {
            indexOfMax = i;
        }
    };

    return ref numbers[indexOfMax];
}
```

- ▶ Can only return references to heap-based values – not locals

# Ref Readonly

- ▶ Ref Returns can be enforced read-only by the compiler

```
ref readonly int FindMax( int[] numbers )  
{  
    int indexOfMax = 0;  
    ...  
    return ref numbers[indexOfMax];  
}
```

```
ref readonly int max = ref FindMax(numbers);  
WriteLine($"{nameof(max)} is now {max}");
```

```
max = 1000; // Not allowed!
```

- ▶ Must manually create a copy to make it modifiable later

```
int maxCopy = FindMax(numbers); // Copy  
maxCopy = 999999;
```

# Agenda

- ▶ Introduction
- ▶ Value Tuples and Syntax
- ▶ Pattern Matching
- ▶ Method Improvements
- ▶ **Expression Improvements**
- ▶ Other C# 7.1 Additions
- ▶ Other C# 7.2 Additions
- ▶ Other C# 7.3 Additions

# More Expression-bodied Members

- ▶ Earlier only getters and methods could be expression-bodied

```
public class Person
{
    ...
    public Person( string name ) => Names.Add(_id, name);

    ~Person() => Names.Remove(_id);

    public string Name
    {
        get => Names[_id];
        set => Names[_id] = value;
    }
}
```

- ▶ New in C# 7.0
  - Constructors
  - Destructors
  - Setters

# 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...

# Declaration Expressions: **out var**

- ▶ Introduces local variable in nearest surrounding scope

```
string s = ReadLine();  
int result;  
if (int.TryParse(s, out result))  
{  
    WriteLine(result);  
}
```

- ▶ Visual Studio has a handy refactoring for this



```
string s = ReadLine();  
if (int.TryParse(s, out int result))  
{  
    WriteLine(result);  
}
```

# Discards

- ▶ Temporary, dummy variables which are intentionally unused in application code

```
Employee elJefe = new Employee { ... };  
var (first, _) = elJefe;  
WriteLine(first);
```

```
if (int.TryParse(s, out _))  
{  
    // s is a legal int  
}
```

- ▶ Supported scenarios
  - Tuples and object deconstruction
  - Pattern matching
  - Calls to methods with **out** parameters
  - A standalone `_` (when no `_` is in scope)



# Binary Literals and Digit Separators

```
enum FileAttributes
{
    ReadOnly =          0b00_00_00_00_00_00_01, // 0x0001
    Hidden =            0b00_00_00_00_00_00_10, // 0x0002
    System =             0b00_00_00_00_00_01_00, // 0x0004
    Directory =          0b00_00_00_00_00_10_00, // 0x0008
    Archive =            0b00_00_00_00_01_00_00, // 0x0010
    Device =             0b00_00_00_00_10_00_00, // 0x0020
    Normal =             0b00_00_00_01_00_00_00, // 0x0040
    Temporary =          0b00_00_00_10_00_00_00, // 0x0080
    SparseFile =         0b00_00_01_00_00_00_00, // 0x0100
    ReparsePoint =       0b00_00_10_00_00_00_00, // 0x0200
    Compressed =         0b00_01_00_00_00_00_00, // 0x0400
    Offline =            0b00_10_00_00_00_00_00, // 0x0800
    NotContentIndexed =  0b01_00_00_00_00_00_00, // 0x1000
    Encrypted =          0b10_00_00_00_00_00_00 // 0x2000
}
```

# Leading Underscores in Numeric Literals

- ▶ Starting from C# 7.2 the numeric literals of C# 7.0 are allowed to start with an underscore

```
int i = 0b00_00_00_00_00_00_01; // Allowed in C# 7.0
int j = 0b_00_00_00_00_00_00_01; // Allowed in C# 7.2
int k = 0x_ffffff;                // Allowed in C# 7.2
int m = 8_7;                      // Allowed in C# 7.0
int n = _8_7;                     // Not allowed
```

- ▶ Note:
  - Only allowed for hexadecimal and binary literals
  - Not decimals...!

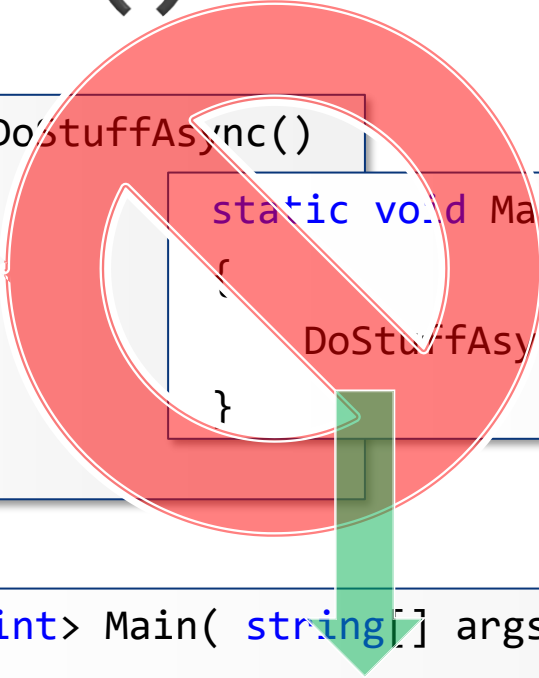
# Agenda

- ▶ Introduction
- ▶ Value Tuples and Syntax
- ▶ Pattern Matching
- ▶ Method Improvements
- ▶ Expression Improvements
- ▶ **Other C# 7.1 Additions**
- ▶ Other C# 7.2 Additions
- ▶ Other C# 7.3 Additions

# Evolution of C# 7.1



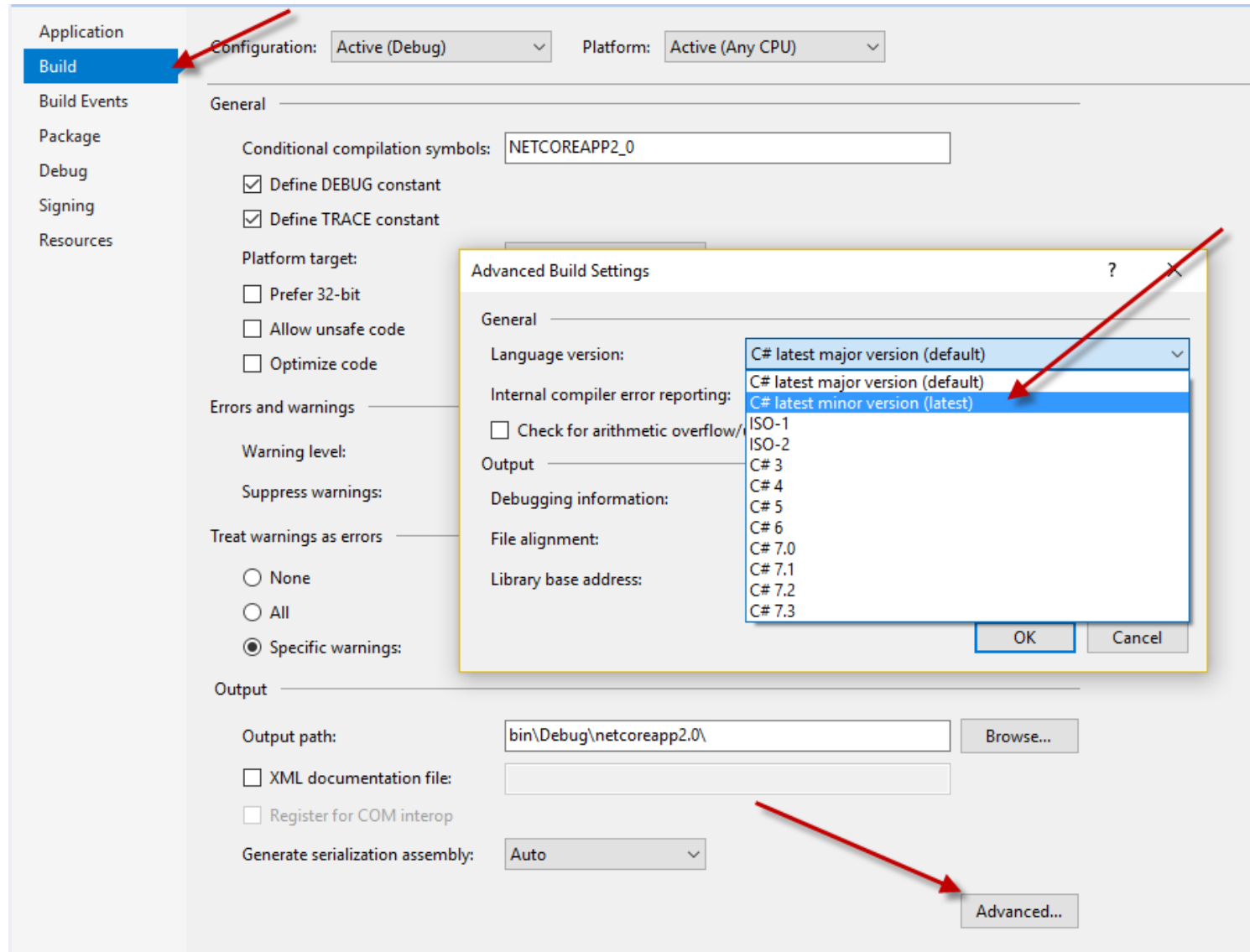
# Async Main()



```
static async Task DoStuffAsync()  
{  
    ... await ...  
    ... await ...  
    ... await ...  
}  
  
static void Main(string[] args)  
{  
    DoStuffAsync().GetAwaiter().GetResult();  
}
```

```
static async Task<int> Main( string[] args )  
{  
    ... await ...  
}  
  
int $GeneratedMain( string[] args )  
{  
    return Main(args).GetAwaiter().GetResult();  
}
```

# Enabling C# 7.x in Visual Studio 2017



# Visual Studio 2019+2022

## Default C# Versions

Target framework	version	C# language version default
.NET	6.x	C# 10.0
.NET	5.x	C# 9.0
.NET Core	3.x	C# 8.0
.NET Core	2.x	C# 7.3
.NET Standard	2.1	C# 8.0
.NET Standard	2.0	C# 7.3
.NET Standard	1.x	C# 7.3
.NET Framework	all	C# 7.3

- ▶ Visual Studio 2017 introduced **LangVersion** in project file
- ▶ Visual Studio 2019 + 2022 attempts to use defaults

# Pattern Matching Open Types

- ▶ Patterns now play well with (sub-)type constraints for generic types

```
static void Promote<T>( T employee )
{
    switch (employee)
    {
        case SoftwareArchitect sa:
            sa.Level = SoftwareArchitectLevel.Lead;
            break;
        case SoftwareEngineer se:
            se.Level = SoftwareEngineerLevel.Chief;
            break;
    }
}
```


- ▶ Compiles in C# 7.1, but not in C# 7.0



# Default Literal

- ▶ C# 7.1 now allows to omit the type in the default operator
  - When the type can be deferred from the context

```
bool flag = false;  
int i = flag ? 87 : default(int);  
WriteLine(i);
```



```
bool flag = false;  
int i = flag ? 87 : default;  
WriteLine(i);
```

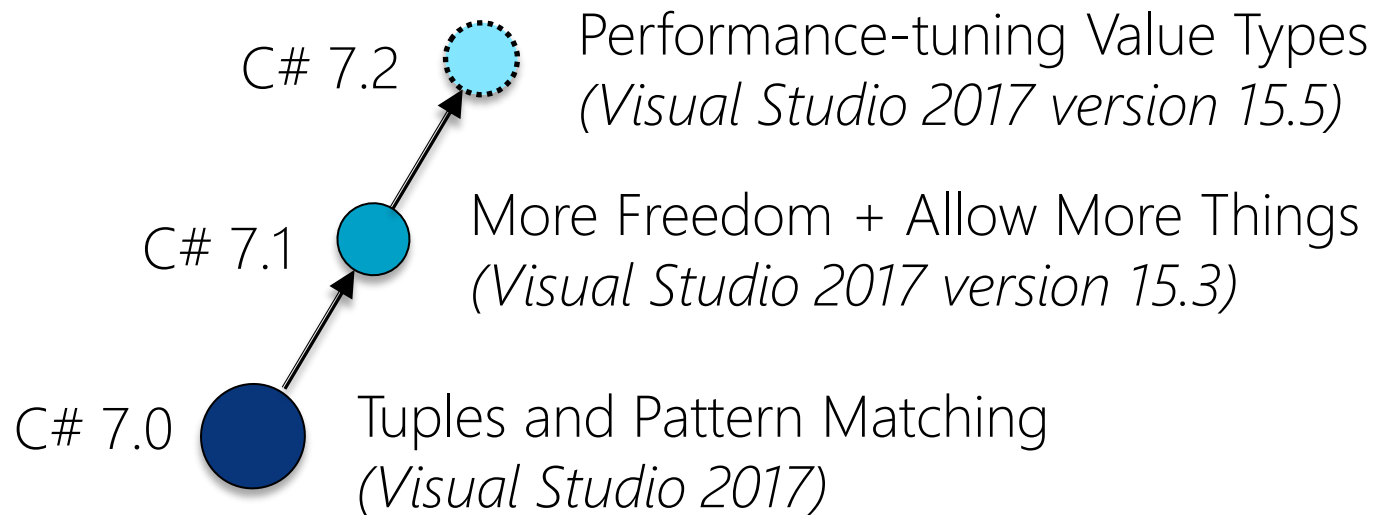
- ▶ Compiles in C# 7.1, but not in C# 7.0
- ▶ Has a number of nice and simple uses such as

```
void DoStuff( int x, int y = default, bool z = default )  
{  
    WriteLine($"x={x}\ty={y}\tz={z}");  
}
```


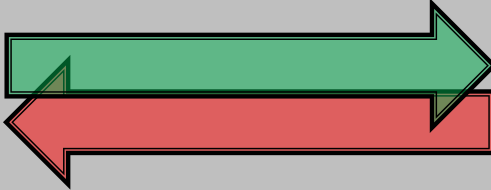


# Agenda

- ▶ Introduction
- ▶ Value Tuples and Syntax
- ▶ Pattern Matching
- ▶ Method Improvements
- ▶ Expression Improvements
- ▶ Other C# 7.1 Additions
- ▶ **Other C# 7.2 Additions**
- ▶ Other C# 7.3 Additions

# Evolution of C# 7.2



# **in** Parameter Modifier

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

- ▶ Can always be passed as **in**
- ▶ Can always be **readonly ref** returned
- ▶ Compiler generates more optimized code for these values

# Ref Structs

- ▶ Structs can be enforced as “always stack allocated” using **ref struct**

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

- ▶ These values can never be allocated on the heap
  - Cannot be boxed
  - Cannot be declared members of a class or (non-ref) struct
  - Cannot be local variables in async methods
  - Cannot be declared local variables in iterators
  - Cannot be captured in lambda expressions or local functions

# Span<T> and ReadOnlySpan<T>

- ▶ Ref-like types to avoid allocations on the heap
  - Don't have own memory but points to someone else's
  - Essentially: "ref for sequence of variables"

```
int[] array = new int[10];  
...  
Span<int> span = array.AsSpan();  
Span<int> slice = span.Slice(2, 5);  
foreach (int i in slice)  
{  
    Console.WriteLine( i );  
}
```

```
string s = "Hello, World";  
ReadOnlySpan<char> span = s.AsSpan();  
ReadOnlySpan<char> slice =  
    span.Slice(7, 5);  
foreach (char c in slice)  
{  
    Console.Write(c);  
}
```

- ▶ Note:
  - Located in System.Memory prerelease nuget package



# Non-trailing Named Arguments

- ▶ As of C# 7.2 named arguments can now be followed by positional arguments...
  - ... but only if named argument is used in the **correct** position

```
void M( int x, int y = 87, bool z = default )  
{  
    Console.WriteLine($"x = {x}, y = {y}, z = {z}");  
}
```

```
M(1, 2, true);           // Allowed in C# 4.0  
M(x: 1, 2, z: true);    // Allowed in C# 7.2 (but not C# 7.1)  
M(z: true, 1);          // Not allowed!
```

# 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);
}
```

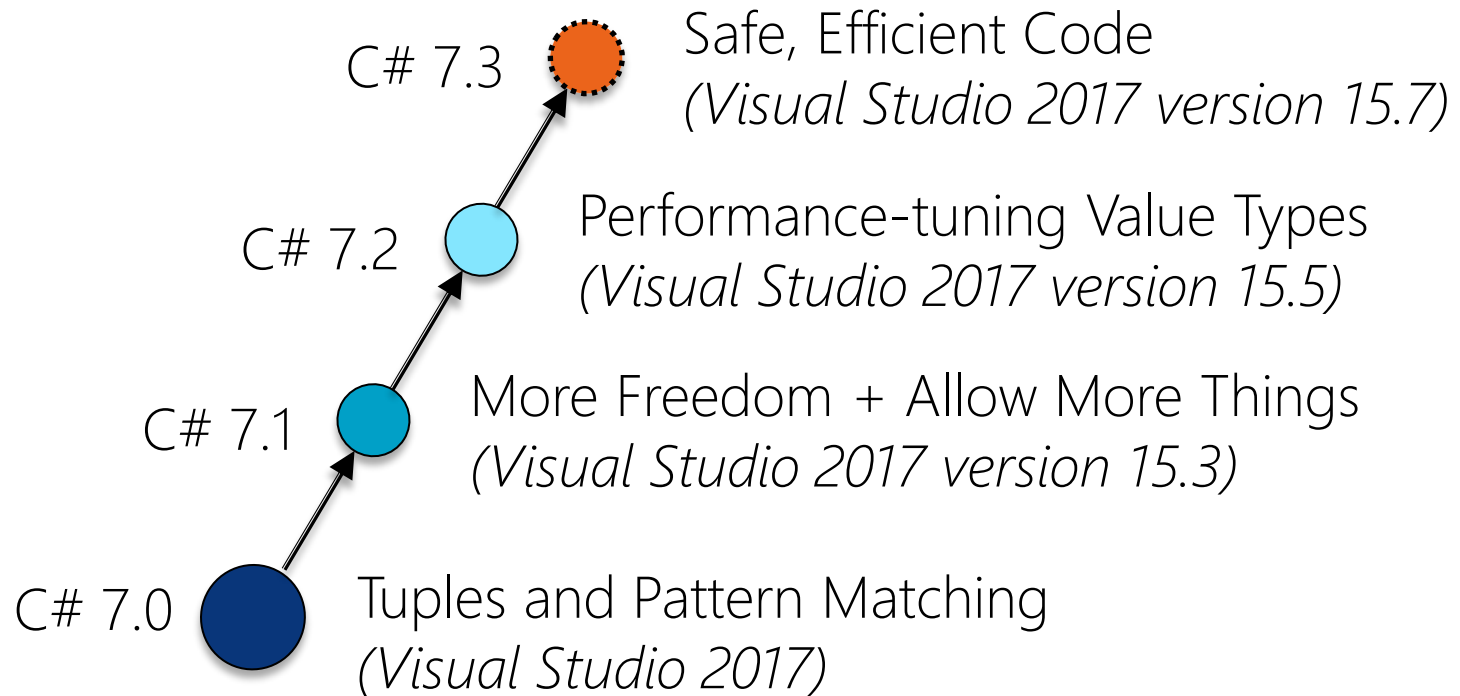
## ▶ protected internal

- Is visible to types in same assembly
- Is visible to derived classes (in same or other assemblies)

# Summary

- ▶ Introduction
- ▶ Value Tuples and Syntax
- ▶ Pattern Matching
- ▶ Method Improvements
- ▶ Expression Improvements
- ▶ Other C# 7.1 Additions
- ▶ Other C# 7.2 Additions
- ▶ **Other C# 7.3 Additions**

# Evolution of C# 7.3



# Expression Variables in Initializers

- ▶ More flexible initialization was introduced in C# 7.0
- ▶ C# 7.3 extends out var and pattern variables to initializers

```
class Base
{
    public int Coordinate { get; } =
        int.TryParse("hello", out int x) ? x : default;

    public Base( int coordinate = default ) => Coordinate = coordinate;
}
```

```
class Derived : Base
{
    public Derived( object o ) : base(o is Point p ? p.X : default)
    {
    }
}
```

# Attributes on Backing Fields

- ▶ C# 7.3 allows attributes targeting the backing fields for auto-properties

```
[Serializable]
class ShoppingCartItem
{
    public int ProductId { get; }
    public decimal Price { get; }
    public int Quantity { get; }
    [field:NonSerialized]
    public decimal Total { get; }

    public ShoppingCartItem( int productID, decimal price, int quantity )
    {
        ProductId = productID;
        Price = price;
        Quantity = quantity;
        Total = price * quantity;
    }
}
```

# More Generic Constraints

Generic Constraint	Description
<code>where T : struct</code>	T must ultimately derive from <b>System.ValueType</b>
<code>where T : class</code>	T must be a reference type
<code>where T : new()</code>	T must have a default constructor
<code>where T : BaseClass</code>	T must derive from the class <i>BaseClass</i>  T can now be <b>System.Enum</b> T can now be <b>System.Delegate</b>
<code>where T : Interface</code>	T must implement the interface <i>Interface</i>
<code>where T : unmanaged</code>	T must be "unmanaged", i.e. can take unmanaged pointer to T

# Enum Constraints

- ▶ Finally(!) we can now do proper enum constraints on generic types

```
public static T GetRandomMember<T>() where T : struct, Enum
{
    Random random = new Random();
    T[] ts = Enum.GetValues(typeof(T))
        .OfType<T>()
        .ToArray()
        ;

    return ts[random.Next(ts.Length)];
}
```

- ▶ *See Lab 01.7*



# Misc. Unmanaged Interop

- ▶ Now **stackalloc** expressions can have initializers

```
Span<int> span = stackalloc int[] { 11, 22, 33 };
```

- ▶ Indexing movable fixed buffers (without pinning)

```
unsafe struct S  
{  
    public fixed int FixedField[10];  
}
```

```
static S s;  
...  
// No fixed required  
int i = s.FixedField[5];
```

- ▶ Custom fixed statement

```
byte[] byteArray = new byte[10];  
fixed (byte* ptr = byteArray)  
{  
    // byteArray is protected from being moved/collected by the GC  
    // for the duration of this block  
}
```

# Summary

- ▶ Introduction
- ▶ Value Tuples and Syntax
- ▶ Pattern Matching
- ▶ Method Improvements
- ▶ Expression Improvements
- ▶ Other C# 7.1 Additions
- ▶ Other C# 7.2 Additions
- ▶ Other C# 7.3 Additions

