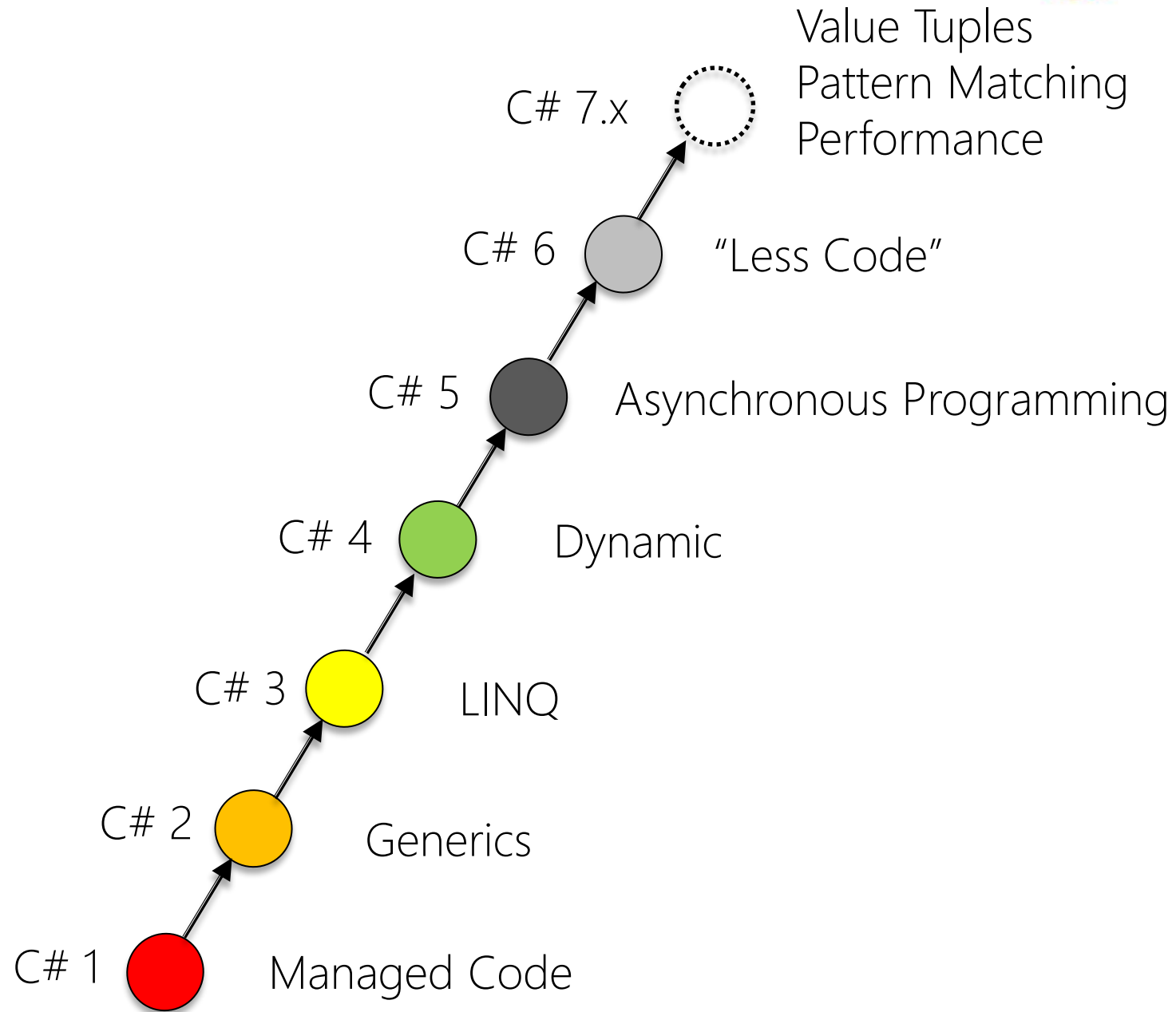


Module 01:

"Very Quick Recap of C# 7.x"



# Evolution of C#



# Agenda

- ▶ Introduction
- ▶ **Value Tuples and Syntax**
- ▶ Pattern Matching
- ▶ Method Improvements
- ▶ Other C# 7.x 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 ... ");
```

- ▶ Tuples can be supplied with descriptive names
- ▶ Mutable and directly addressable
- ▶ Tuples can be supplied with descriptive names
- ▶ Mutable and directly addressable
- ▶ Built-in: **To**String() + **E**quals() + **G**etHashCode() (but not == until C# 7.3)

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

# 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

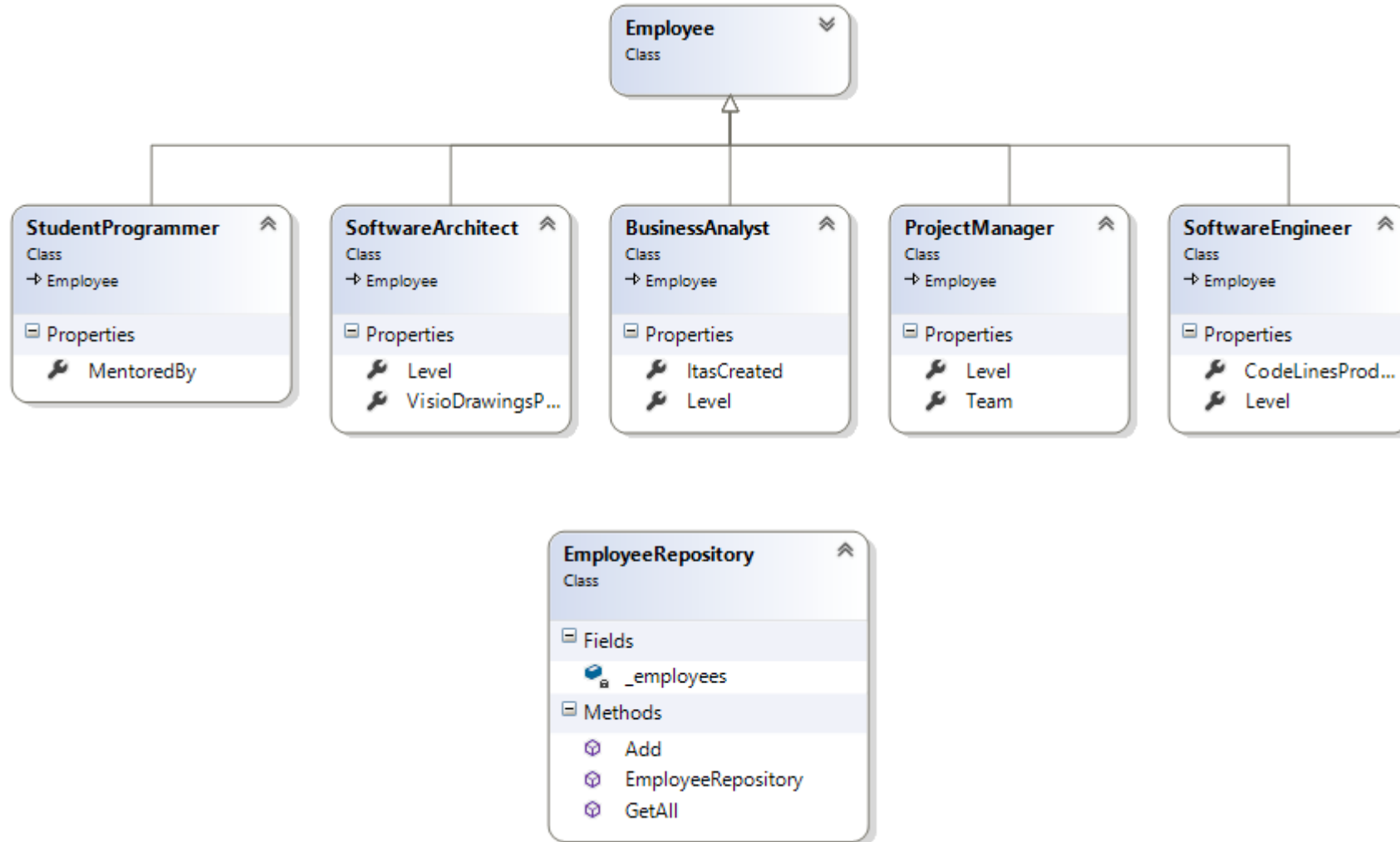
- Deconstructors can be overloaded

# Agenda

- ▶ Introduction
- ▶ Value Tuples and Syntax
- ▶ **Pattern Matching**
- ▶ Method Improvements
- ▶ Other C# 7.x 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**
- ▶ Other C# 7.x 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 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;
```


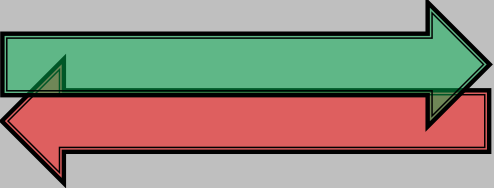
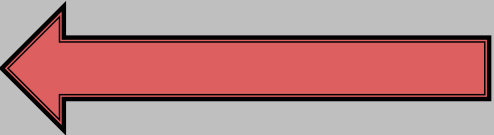
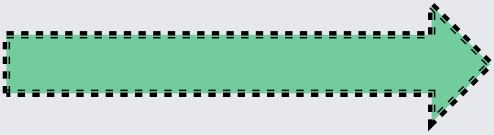
# Agenda

- ▶ Introduction
- ▶ Value Tuples and Syntax
- ▶ Pattern Matching
- ▶ Method Improvements
- ▶ **Other C# 7.x Additions**





# in Parameter Modifier

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

# Summary

- ▶ Introduction
- ▶ Value Tuples and Syntax
- ▶ Pattern Matching
- ▶ Method Improvements
- ▶ Other C# 7.x Additions



