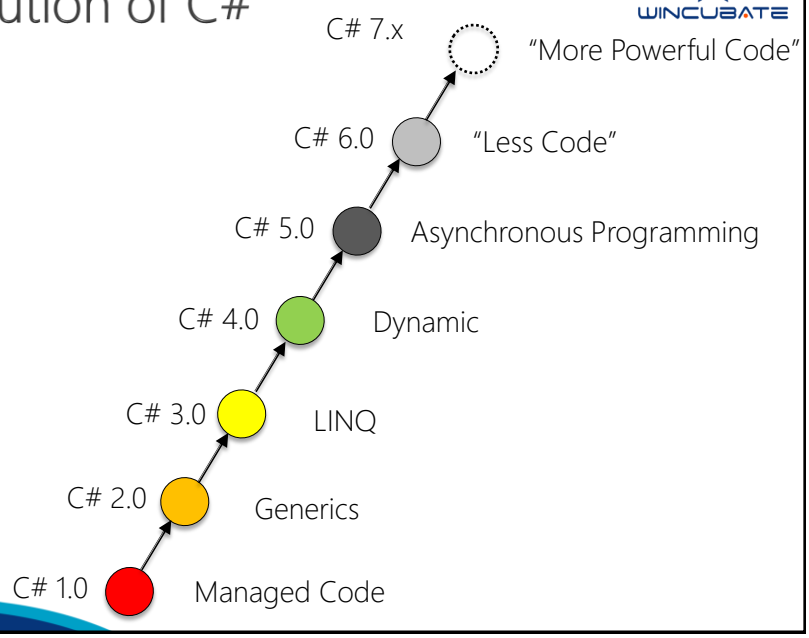
Module 09

"What's New in C# 7.x"

Evolution of C#



Agenda

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

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 = Console.ReadLine();

var t = FindVowels( input );

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

- Note: In VS 2017 (RC?) you must manually add reference to `System.ValueTuple` NuGet package

Tuple 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 \"{input}\"");
```

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

- Mutable and directly addressable
- Some built-in implicit tuple conversions
  - `ToString()` + `Equals()` + `GetHashCode()` (but not ==)

Custom Tuple Deconstruction

- Can be easily deconstructed to individual parts

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

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

```
class Employee
{
    ...

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

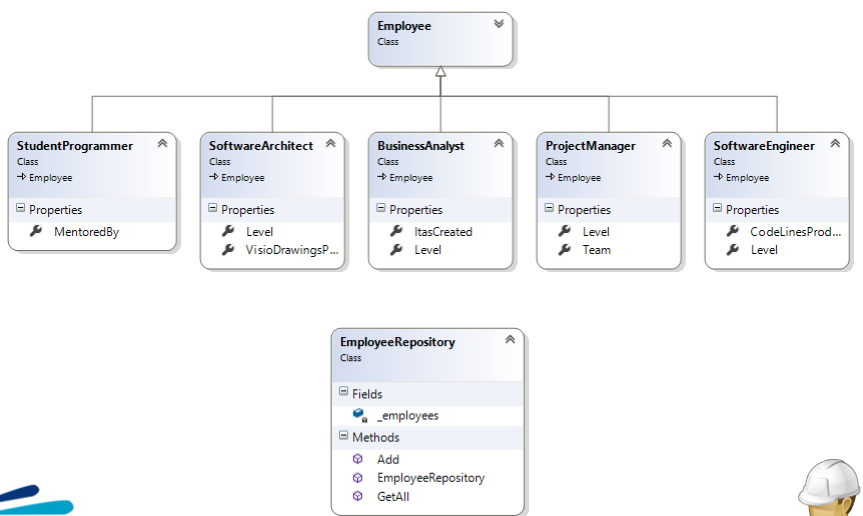
```
Employee employee = new Employee { ... };

var (first,last) = employee;
WriteLine( first );
```

- Works for two or more deconstruction parts

Agenda

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

Example: Employee



Pattern Matching with is

- Three types of patterns for matching in C# 7.0
  - 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 to come in C# 8.0…

```
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

- Enhanced switch statement
  - Can switch on <u>any</u> type
  - Case clauses can make use of patterns
  - Case clauses can make use of `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;
}
```

- <u>Beware</u>: Cases are no longer disjoint – evaluated sequentially!

Agenda

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

Local Functions

- Methods within methods can now be defined

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

        if( IsVowel() ) { ... }
    }
    ...
}
```

- Has a few advantages
  - Captures local variables
  - Avoids allocations

Ref Locals and Ref Returns

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

    return ref numbers[ index ];
}
```

- References in the style of C++
  - Local variables
  - Return values

Agenda

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

More Expression-bodied Members

- Earlier only getters and methods could be expression-bodied

```
class Person
{
    ...

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

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

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

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

Throw Expressions

- In C# 6.0 one could not easily just throw an exception in an expression-bodied member
- C# 7.0 allows `throw` expressions as subexpressions
  - Also outside of expression-bodied members..!

```
public class EmployeeRepository
{
    ...

    private List<Employee> _employees;

    public void Add( Employee e ) =>
        _employees.Add( e ?? throw new ArgumentNullException(nameof(e)) );
}
```

- Note that a `throw` expression does not have an expression type as such…

Declaration Expressions: `out var`

- Introduces local variable in nearest surrounding scope
  - Limitation of general declaration expressions which were scrapped for C# 6.0

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

- VS 2017 has a handy refactoring for this

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

- Note: `return var` is still not in C# 7.0 ☺

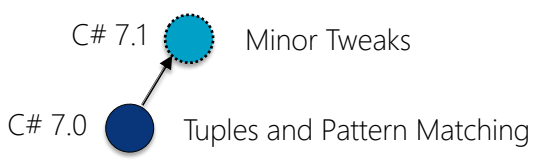
Binary Literals and Digit Separators

```
public 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
}
```

Agenda

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

Evolution of C# 7.1



Async **Main()**

```
static async Task DoStuffAsync()
{
    ... await ...
    ... await ...
    ... await ...
}
```

```
static void Main()
{
    DoStuffAsync().GetAwaiter().GetResult();
}
```

```
static async Task<int> Main( string[] args )
{
   ...
}

int $GeneratedMain( string[] args )
{
    return Main( args ).GetAwaiter().GetResult();
}
```

Generics Types with Patterns

- Patterns now plays 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 );
```

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

- Has a number of nice and simple uses such as

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; }
}
```

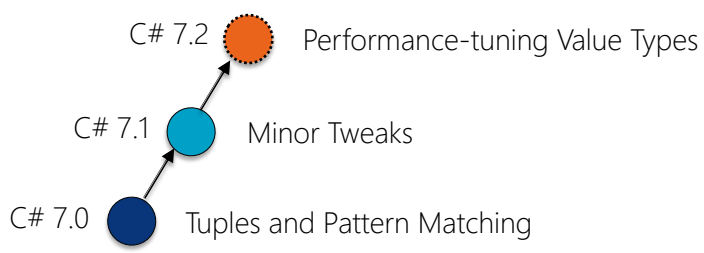
```
Equipment e = new Equipment { ... };
var tuple = (e.Console, e.Controllers);

WriteLine( tuple.Console );
```

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

Agenda

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

Evolution of C# 7.2



in Parameter Modifier

- As a supplement for **ref** and **out**: **in** parameter modifier
- Indicates that the method does not modify the value (i.e. it is read-only!)
  - Hence 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 Math.Sqrt(xDiff * xDiff + yDiff * yDiff + zDiff * zDiff);
}
```

- Note: The call site does not need to specify **in**

```
Point3D p1 = new Point3D { X = -1, Y = 0, Z = -1 };
Point3D p2 = new Point3D { X = 1, Y = 2, Z = 3 };
double distance = CalculateDistance(p1, p2);
```

Ref Readonly Returns

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

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

```
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;
```

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 )
    {
        ...
    }
}
```

- 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 <u>never</u> (accidentally) 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

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!
```

Leading Underscores in Numeric Literals

- Starting from C# 7.2 the numerics 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_ffff;                 // 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…!

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
- C# 7.1 Additions
- C# 7.2 Additions