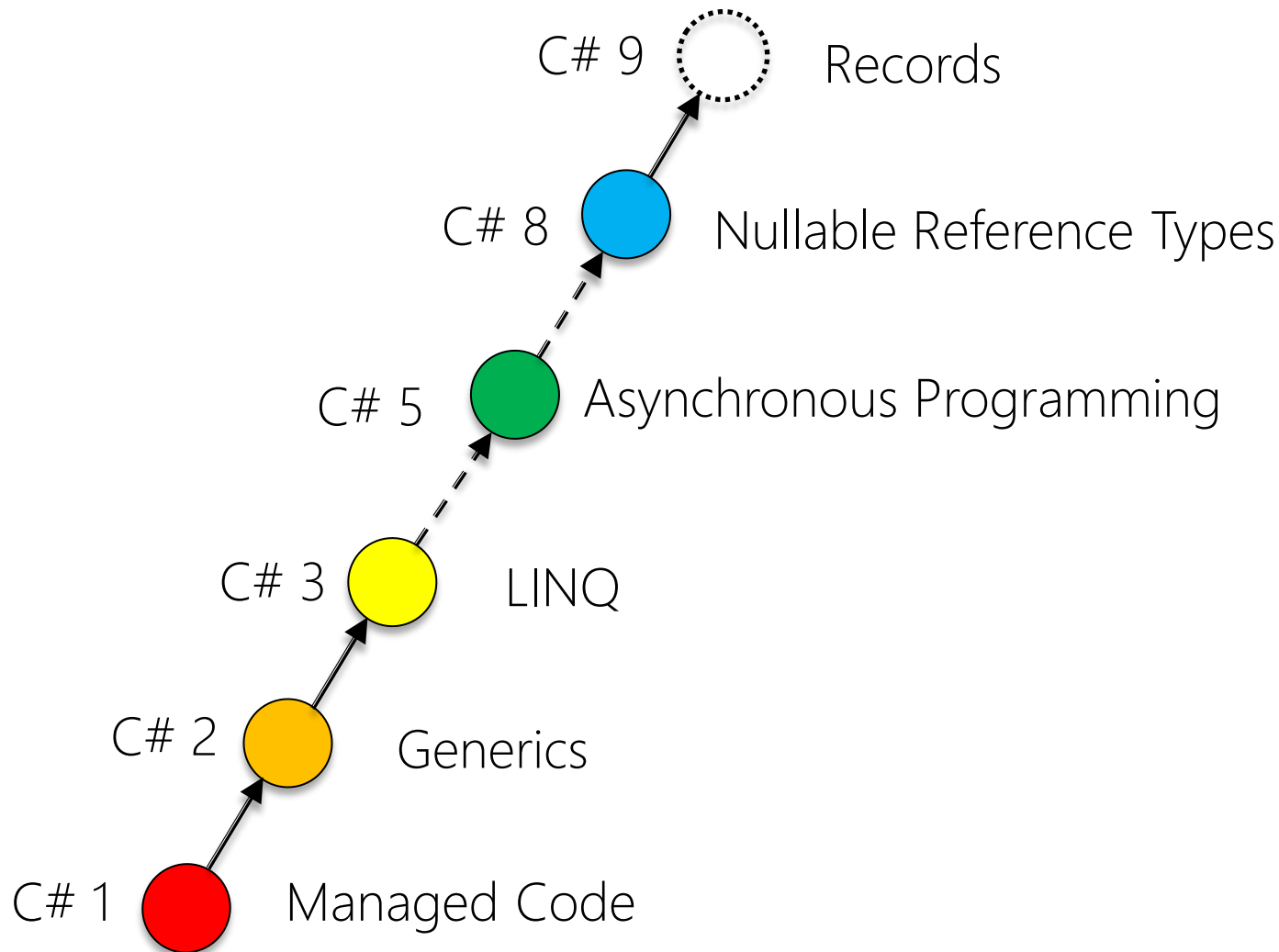


Module 02:

"An Introduction to C# 9"

Major Evolutions of C#



Agenda

- ▶ Introduction
- ▶ **Object-Oriented Improvements**
- ▶ Statement Improvements
- ▶ Expression Improvements
- ▶ Method Improvements
- ▶ Performance, Interop, and Code Generation
- ▶ Summary

Init-only Setters

- ▶ Restricted setter only allowing initialization:

```
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
- ▶ Additional keyword: Create copies using **`with`**

Mutation-free Copying

- ▶ Additional keyword: Create copies using **with**

```
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

A Few Words of Warnings

- ▶ Could we “break” immutability manually?
- ▶ What if we override **ToString()** on **Record**?
- ▶ What happens if we add “**Albums**” as a property?

Agenda

- ▶ Introduction
- ▶ Object-Oriented Improvements
- ▶ **Statement Improvements**
- ▶ Expression Improvements
- ▶ Method Improvements
- ▶ Performance, Interop, and Code Generation
- ▶ Summary

Top-level Statements

- ▶ A fundamental rule of C# has now been relaxed:

```
using System;  
namespace Wincubate.CS9.B  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Console.WriteLine("Hello World!");  
        }  
    }  
}
```

- ▶ But what about the arguments then?

Extension Enumerables

- ▶ It is possible to create an “extension implementation” of **IEnumerable<T>** for a third-party type
 - **foreach** now respects extension **GetEnumerator<T>** methods

```
static class SequenceExtensions
{
    public static IEnumerator<T> GetEnumerator<T>( this Sequence<T> t )
    {
        SequenceElement<T>? current = t.Head;
        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }
}
```

Agenda

- ▶ Introduction
- ▶ Object-Oriented Improvements
- ▶ Statement Improvements
- ▶ **Expression Improvements**
- ▶ Method Improvements
- ▶ Performance, Interop, and Code Generation
- ▶ Summary

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

```
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

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

- ▶ (See lab on relation to `!= null` 😊)

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

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

Disjunctive Patterns

- ▶ Disjunctive patterns specify an **or** between patterns

```
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
 - `<` , `<=` , `>` , `>=`

```
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 `=>`

Target-typed New

- ▶ Target-typed new expressions are essentially the “counterpart” of **var**

```
Dictionary<string, List<int>> field = new()  
{  
    { "item1", new() { 1, 2, 3 } }  
};
```

- ▶ There are a number of disallowed scenarios:
 - Interfaces
 - Enums
 - Dynamic types
 - Tuples
 - ...

Target-Typed Conditionals

- ▶ New implicit conditional expression conversion:

```
bool b = true;  
int i = 87;  
  
Console.WriteLine($"Result: {( b ? i : null)}");
```

- ▶ There are some slightly strange implications, but works well in practice 😊

Agenda

- ▶ Introduction
- ▶ Object-Oriented Improvements
- ▶ Statement Improvements
- ▶ Expression Improvements
- ▶ **Method Improvements**
- ▶ Performance, Interop, and Code Generation
- ▶ Summary

Covariant Return Types

- ▶ Return types for methods are now relaxed to covariance

```
public class ConfigProvider
{
    public virtual Config GetConfig() { ... }
}

public class AppleConfigProvider : ConfigProvider
{
    public override AppleConfig GetConfig() { ... }
}
```

- ▶ Must exist an implicit conversion
- ▶ What about interface, strings, int, doubles, ... ?

Lambda Discard Parameters

- ▶ Previous versions of C# introduced discards in various guises – C# 9 extends to lambda expressions

```
using System;  
  
Func<int, int, int?> add = (x, y) => x + y;  
Func<int, int, int?> nullFunction = (_, _) => 0;  
  
Console.WriteLine(add(42, 87));  
Console.WriteLine(nullFunction(42, 87));
```

- ▶ Beware of edge-cases for single params or existing vars

Static Anonymous Functions

- ▶ C# 7.0 introduced local function
- ▶ C# 8.0 introduced static local functions
- ▶ C# 9.0 introduces status anonymous (and lambda) functions

```
foreach (char letter in s)
{
    static bool IsVowel(char letter) => char.ToLower(letter) switch
    {
        'a' or 'e' or 'i' or 'o' or 'u' or 'y' or 'æ' or 'ø' or 'å'
            => true,
        _ => false,
    }
    ...
};
```

- ▶ No capture of locals, members and use of **this**, **base**.

Attributes on Local Methods

- ▶ Attributes can now be placed on local functions and lambdas
 - Levelling the playing field for methods variations

```
void Main(string[] args)
{
    [Conditional("DEBUG")]
    static void PrintInfo(string s) =>
    {
        Console.WriteLine($"Debug: {s}");
    }

    PrintInfo("Start");
}
```

- ▶ Method must be static!
- ▶ Note: Local methods can now also be marked **extern**

Revisiting Partial Methods

```
partial class Customer
{
    private string _name;
    public string Name
    {
        get { return _name; }
        set
        {
            OnNameChanging(value);
            _name = value;
            OnNameChanged();
        }
    }
    partial void OnNameChanging(string newName);
    partial void OnNameChanged();
}
```

New Features for Partial Methods

- ▶ Traditionally, partial methods suffered some restrictions:
 - ~~Cannot define access modifiers (implicitly private)~~
 - ~~Must have `void` return type~~
 - ~~Parameters cannot have the `out` modifier~~
- ▶ These restrictions are removed if
 - Can be annotated with explicit accessibility modifier (if consistent)
 - When explicit modifier, it must have a matching implementation

```
partial class Customer
{
    ...
    private partial bool OnNameChanging(string newName);
    public partial void OnNameChanged(out string oldName);
}
```

Agenda

- ▶ Introduction
- ▶ Object-Oriented Improvements
- ▶ Statement Improvements
- ▶ Expression Improvements
- ▶ Method Improvements
- ▶ **Performance, Interop, and Code Generation**
- ▶ Summary

Native-Sized Integers

- ▶ For interop scenarios and low-level libraries:
 - `nint` ~ `System.IntPtr`
 - `nuint` ~ `System.UIntPtr`

```
unsafe
{
    nint y = 87;
    Console.WriteLine(y);
    Console.WriteLine(sizeof(nint));
}
```

- ▶ Extended with a lot of conversions and operations

Function Pointers

- ▶ Pretty much only for emitting highly optimized unsafe code

```
unsafe class Example
{
    public void Compute(Action<int> a, delegate*<int, void> f)
    {
        a(42);
        f(42);
    }
}
```

- ▶ Not for “ordinary” people... 😊

Suppress Localsinit

- ▶ Optimization technique for really advanced low-level library authors:

```
[SkipLocalsInit]
```

```
unsafe class Entity
```

```
{
```

```
    public void DoStuff(out int a)
```

```
    {
```

```
        // a will contain "non-zero'ed" memory if read here
```

```
        a = 87;
```

```
    }
```

```
}
```

- ▶ Avoids the penalty of zero-whitewashing memory

Module Initializers

- ▶ A module initializer runs whenever the module is **loaded!**
 - The method must be static.
 - The method must be parameterless.
 - The method must return **void**.

```
class C
{
    [ModuleInitializer]
    internal static void InitializeLogFile()
    {
        File.CreateText(_fileName);
    }
}
```

- ▶ There can be several module initializers

Summary

- ▶ Introduction
- ▶ Object-Oriented Improvements
- ▶ Statement Improvements
- ▶ Expression Improvements
- ▶ Method Improvements
- ▶ Performance, Interop, and Code Generation

