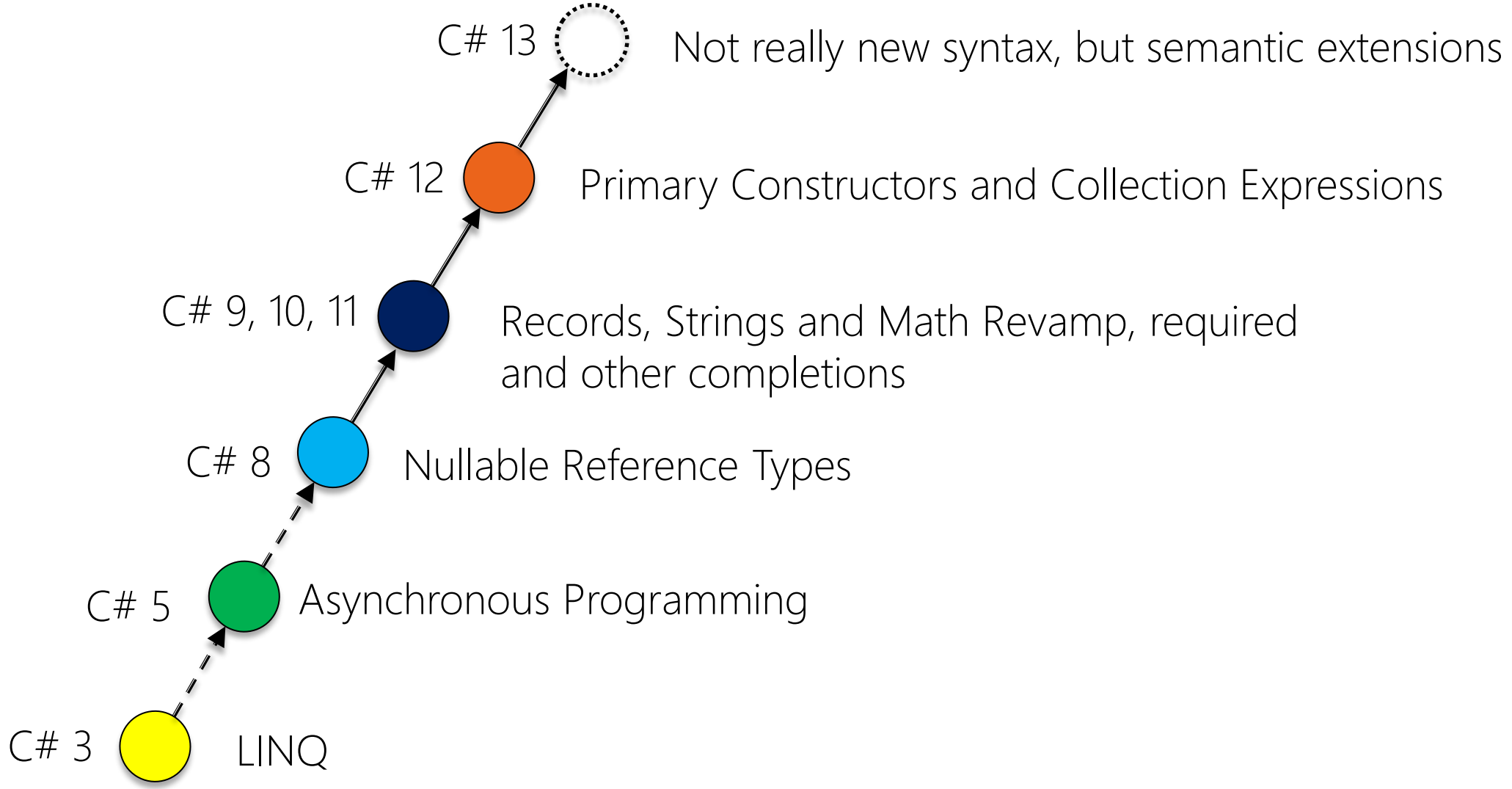
Module 07:

"The Brand New C# 13"

Major Evolutions of C#



Agenda

- Introduction
- **Method Improvements**
- Statement Improvements
- Expression Improvements
- Object-Oriented Improvements
- Summary

Param Collections

- The **params** modifier can now be used with any collection type or interface

```
record class Pizza
{
    ...
    public Pizza(params IEnumerable<Topping> toppings)
    {
        _toppings = toppings;
    }
}
```

```
Pizza meatLover = new(Topping.Beef, Topping.Kebab, Topping.Bacon);
```

- Including concrete types `List<T>`, `Span<T>` `ReadOnlySpan<T>`, …
- Types implementing interfaces `IEnumerable<T>`, `IReadOnlyCollection<T>`, `IReadOnlyList<T>`, `ICollection<T>`, `IList<T>` …

Extended to Custom Collection Types

- Also works for any custom type
  - Implementing **IEnumerable<T>**
  - Public **Add()** method

```
public class SequencePacker<T> : IEnumerable<T>
{
    ...

    public void Add(T element)
    {
        ...
    }
}
```

Overload Resolution

- C# 12 in general improved overload resolution to chose the best overload available
- In C# 13 library authors can manually prioritize to avoid ambiguities using the `OverloadResolutionPriority` attribute

```
class ObjectHandler
{
    public static void Handle<T>(ImmutableArray<T> elements) {  ...  }

    [OverloadResolutionPriority(1)]
    public static void Handle<T>(params ReadOnlySpan<T> elements) {  ...  }
}
```

Agenda

- Introduction
- Method Improvements
- **Statement Improvements**
- Expression Improvements
- Object-Oriented Improvements
- Summary

New Lock Type

- .NET 9 introduces a new **System.Threading.Lock** type

```
public sealed class Lock
{
    public void Enter() { ... }
    public void Exit() { ... }
    public Scope EnterScope() { ... }

    public ref struct Scope
    {
        public void Dispose() { ... }
    }
}
```

```
Lock l = new();

using (var scope = l.EnterScope())
{
    //
    // Critical region ...
    //
}
```

- Decidated solely to being a lock – Better than Monitor-based approach!

The `lock` keyword vs. "Good" Old `Monitor`?

- The lock keyword is integrated into the C# language, but uses the ancient `Monitor` type

```
object l = new();

lock (l)
{
    //
    // Critical region ...
    //
}
```

```
bool lockTaken = false;
try
{
    Monitor.Enter(l, ref lockTaken);
}
finally
{
    if (lockTaken) { Monitor.Exit(l); }
}
```

- Fortunately, C# 13 solves this with
  - Type-dependent compilation
  - Proper warnings for unintended use

Agenda

- Introduction
- Method Improvements
- Statement Improvements
- **Expression Improvements**
- Object-Oriented Improvements
- Summary

Implicit Index Access in Object Initializers

- The "index from end expression" operator **^** from C# 8 is now allowed in object initializer expressions, e.g.
  - Collection initializers

```
Thingy thingy = new()
{
    Elements =
    {
        [^2] = "Hello",
        [^1] = "World",
        [0] = "Booyah!",
        [1] = "Foobar"
    }
};
```

Character Literal Escape Sequence

- You can now use **\e** as as a character literal escape sequence for unicode ESCAPE character U+001B.

```
Console.WriteLine("\eBooyah!! Nicely Escaped!");
```

- Much better alternative than using **\x1b** or **u001b**

Agenda

- Introduction
- Method Improvements
- Statement Improvements
- Expression Improvements
- **Object-Oriented Improvements**
- Summary

Partial Properties and Indexers

- Partial classes and methods were introduced in C# 2 and improved in C# 9
- In C# 13 both properties and indexers are now allowed to be partial
  - But must have an implementation part!

```
partial class Customer
{
    ...
    public partial string FavoriteFood
    {
        get;
        set;
    }
}
```

```
partial class Customer
{
    public partial string FavoriteFood
    {
        get ⇒ _favoriteFood;
        set
        {
            ...
            _favoriteFood = value;
        }
    }
}
```

Revisting 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

Ref Structs and Interfaces

- Up to C# 12 ref structs were never allowed to implement interfaces
  - Problem: Implementing interfaces allows converting to them!
  - Special "duck-typing" allowed `IDisposable` to be "implemented" (remember `Scope` in `Lock`?)

- In C# 13 ref structs can implement interfaces, but with restrictions
  - Cannot be converted to the interface it implements
  - Must implement all interface member – also those with default implementations

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

    public readonly void Display() { Console.WriteLine(ToString()); }
}
```

allows ref struct

- C# 13 introduces a new "anti-constraint" for ref structs and generics

```
static class DisplayHelper
{
    public static void Display<T>(this d)
        where T : IDisplayable, allows ref struct
    {
        d.Display();
    }
}
```

Summary

- Introduction
- Method Improvements
- Statement Improvements
- Expression Improvements
- Object-Oriented Improvements


WINCUBATE
Jesper Gulmann Henriksen
PhD, MCT, MCSD, MCPD
Phone : +45 22 12 36 31
Email : jgh@wincubate.net
WWW : http://www.wincubate.net
Ringgårdsvej 4A
8270 Højbjerg
Denmark