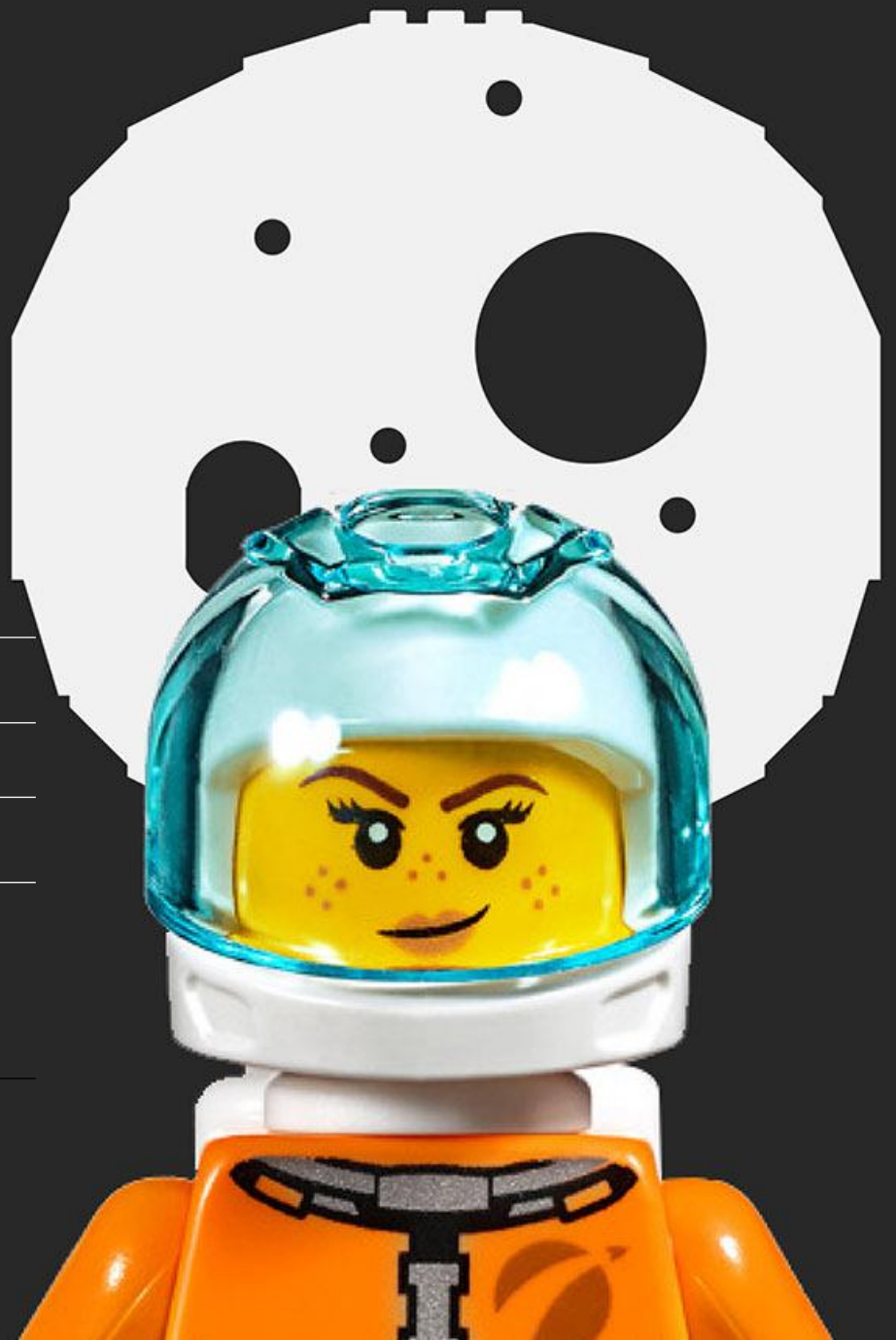


Modern C# For Python Developers

Session 3

September 24, 2025

Jesper Gulmann Henriksen

Lead IT Engineer
BrickLink Technology DK
jesper.henriksen@LEGO.com

Agenda for Session 3

Incoming

- Lookups for Dictionary<K,V>?
- .exe or .dll for assemblies?

3.1 Records

- Syntax and Purpose
- Mutation-free Copying

3.2 Lambdas

- Delegate Types
- Action and Func
- Lambda Expressions
- Expression-bodied Methods
- Throw Expressions

3.3 LINQ

- Introducing LINQ
- Query Keywords
- Deferred Execution
- Query Methods

3.4 Patterns

- Pattern Matching with is
- Switch and Expressions
- Type Patterns
- Property Patterns
- Positional Patterns
- Boolean and Relational Patterns
- List and Slice Patterns

3.5 Object Lifetime

- Garbage Collection
- Destructors
- Dispose and Using

3.6 Tasks

- Introduction
- Async and Await
- Cancellation

Incoming Questions

Q: Lookups for Dictionary<K,V>?

Question:

- *"During the presentation of Session 2, the question regarding reference types and dictionary lookups arose. How does this work again..?"*

Answer:

- Type must override `Equals()` and `GetHashCode()` to provide equality lookup correctly
- Alternatively; Implement `IEquatable<T>` instead of overriding `Equals()`
- Best solution is to use record types... ☺

Q: Why is There a .dll File and Not Just an .exe?

Question:

- *"We mentioned earlier that assemblies are either .exe or .dll files.*

  Then why was there a necessary Application.dll file last time?"

Answer:

- This just means that the **entry point** for an assembly is either

  .exe ~ executable application
  .dll ~ class library

- In newer versions of .NET the .exe file is just a bootstrapper for the application code in an associated .dll file

Module 3.1
Records

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
        {
            return ...;
        }
    }
}
```

Built-in Features of Records

Overrides

- `ToString()`
- `Equals()` (Implements `IEquatable<T>`)
- `GetHashCode()`
- `==` and `!=`

What about `ReferenceEquals`?

Supplies built-in deconstructors

Mutation-free Copying

Additional keyword: Create copies using `with`

Does not mutate source record – Copies and replaces!

```
Album album = new("Prince",
                  "Purple Rain",
                   new DateTime(1984, 11, 02));

Album renamed = album with
{
   Artist = "The Artist Formerly Known as..."
};
```

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

Module 3.2
Lambdas

Introducing Delegates

Like in Python methods in C# are first-class objects

Delegates are references to "method objects".

Deletage types can be defined using `delegate` keyword

```
LookupDelegate del = TransformString;
Console.WriteLine(del("hello!"));

static string TransformString(string s)
{
    return s.ToUpper();
}

delegate string LookupDelegate(string s);
```

Delegates are Multicast

Delegates can point to multiple method objects
- has internal "invocation list"

Can add and remove using += and −=

Note: This is not recommended!

```
MathOperation m = SimpleMath.Multiply;
m += SimpleMath.Add;
m(5, 7);

delegate void MathOperation(int i, int j);
```

Generic Delegates

Delegates can of course be generic

```
GenericDelegate<int> del1 = IntTarget;
GenericDelegate<string> del2 = StringTarget;
...

public delegate void GenericDelegate<T>(T arg);

static void StringTarget(string arg)
{
    Console.WriteLine("arg in uppercase is: {0}", arg.ToUpper());
}
static void IntTarget(int arg)
{
    Console.WriteLine("++arg is: {0}", ++arg);
}
```

Func and Action

The solution lies in built-in, predefined generic versions of `Func` and `Action` types

```
public delegate TResult Func<out TResult>();
public delegate TResult Func<in T, out TResult>(T arg);
public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
...

public delegate void Action();
public delegate void Action<in T>(T obj);
public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);
...
```

Anonymous Methods

Anonymous methods are a way to directly "inline" method syntax for delegates

```
Func<int, DateTime, bool> func = delegate(int i, DateTime dt)
{
    bool isEven = i % 2 == 0;
    Console.WriteLine($"i={i}. Is even at {dt}: {isEven}");

    return isEven;
};

func(87, DateTime.Now);
```

Capturing Local Variables

Note: Local variables are silently captured..!

```
int i = 87;

Action<DateTime> action = delegate(DateTime dt)
{
    bool isEven = i % 2 == 0;
    Console.WriteLine($"i={i}. Is even at {dt}: {isEven}");

    i++;
};

action(DateTime.Now);
```

Lambda Expressions

Anonymous methods have a very nice and compact syntax alternative: "Lambda Expressions"

```
( Type1 arg1, ..., Typen argn ) => code to process arguments
```

Usually types can be inferred by the compiler from the context so they can be left out

```
Action<DateTime> func = dt =>
{
    bool isEven = i % 2 == 0;
    Console.WriteLine($"i={i}. Is even at {dt}: {isEven}");

    i++;
};

Func<int, bool> isEven = i => i % 2 == 0;
```

Lambda Variations

Many syntactic variations are available depending on the context

```
Func<double> vat = () => 25.0;
Console.WriteLine($"Denmark's VAT is {vat()}%");

Func<int,string,bool> alwaysTrue = (_, _) => true;

var choose = object (bool b) => b ? 1 : "two";
Console.WriteLine(choose(false));
```

Expression-bodied Members

All functionality-based members can have expression bodies

- Methods
- Properties

```
public class Person
{
    public required string FirstName { get; set; }
    public required string LastName { get; set; }
    public int Age { get; set; }

    public string FullName => $"{FirstName} {LastName}";

    public override string ToString() =>
        $"{FullName} is {Age} year{(Age == 1 ? "" : "s")} old";
}
```

More Expression-bodied Members

- For properties `get`, `set` accessors can be individually expression-bodied
- Constructors are also supported

```
public class Student
{
    private static IDictionary<Guid, string> Names = new Dictionary<Guid, string>();
    private readonly Guid _id = Guid.NewGuid();

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

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

Throw Expressions

C# allows `throw` expressions as subexpressions

Also outside of expression-bodied members..!

```
public class PersonRepository
{
    private readonly List<Person> _persons;
    ...
    public IEnumerable<Person> GetAll() => [.. _persons];

    public void Add(Person? person) =>
        _persons.Add(person ?? throw new ArgumentNullException(nameof(person)));
}
```

Module 3.3
LINQ

LINQ Keywords

C# initially introduced “Language Integrated Query” = LINQ as a set of SQL-like keywords

Processes any collection class of type `IEnumerable<T>`

```
record LEGOSet
{
    public required int Number { get; init; }
    public required string Title { get; init; }
    public required int Pieces { get; init; }
}


List<LEGOSet> sets = [ ... ];

var results = from set in sets
              where set.Pieces >= 1000
              select set;
```

Don't Use the LINQ Keywords

LINQ was introduced with a number of nice keywords...

- `from`
- `where`
- `select`
- `orderby`
- `group`
- `join`
- `let`

... But **don't** use these! ☺

LINQ Query Method Resolution

The LINQ keywords are resolved to LINQ query methods behind the scenes

- These are essentially extension methods on `IEnumerable<T>`

Always use the LINQ query methods..!

```
var results = sets
    .Where(set => set.Pieces >= 1000)
    .Select(set => set.Title.ToUpper())
    ;
```

Deferred Execution

Note: LINQ queries are not evaluated until the sequence is enumerated!

```
int[] numbers = [ 10, 20, 30, 40, 0, 1, 2, 3, 8 ];
var query = numbers
    .Where( i => i < 10)
    .Select( i =>  87 / i )
    ;

// No exception yet!

foreach (var i in query)
{
    Console.WriteLine(i);
}
```

Immediate Execution

Enumeration can be forced by sequence operations:

- `ToArray()`
- `ToList()`
- `ToDictionary()`

Enumeration can also be forced by aggregations:

- `Count()`
- `Min()` + `Max()` + `Average()` + `Sum()` + …

```
int[] numbers = [ 10, 20, 30, 40, 0, 1, 2, 3, 8 ];
var query = numbers
    .Where( i => i < 10)
    .Select( i =>  87 / i )
    .ToList() // <-- Exception
    ;
```

LINQ Query Method Examples

Boolean

- `Where()`
- `Any()`
- `All()`

Sequences

- `Select()`
- `SelectMany()`

Set operations

- `Union()`
- `Intersect()`
- `Except()`

Singleton operations

- `First()`
- `Last()`
- `Single()`
- `FirstOrDefault()`
- `LastOrDefault()`
- `SingleOrDefault()`

Complex operations

- `Group()`
- `Aggregate()`
- `Zip()`

+ many more..!

Variations of LINQ

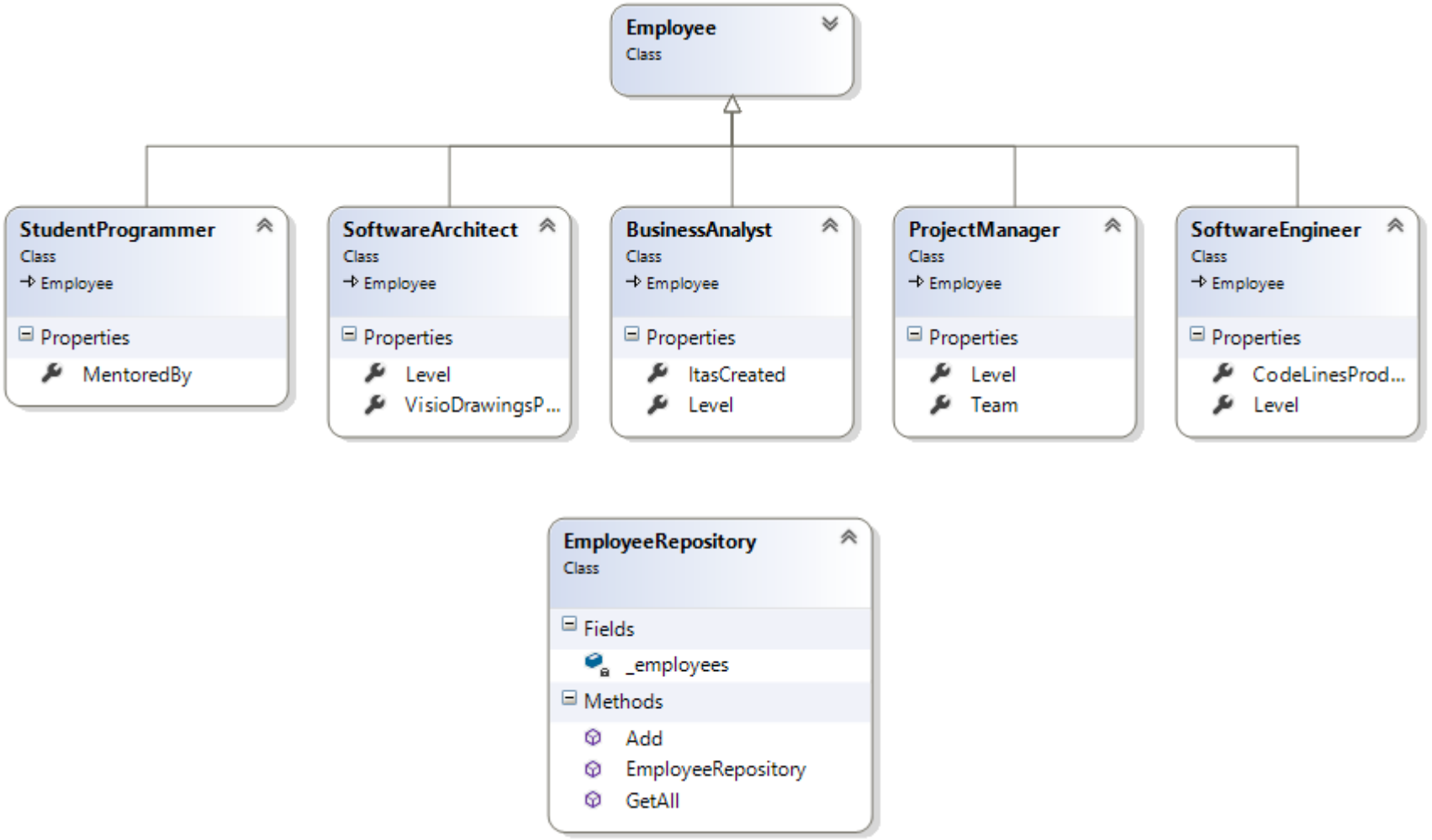
- LINQ to Objects
- LINQ to XML
- LINQ to JSON

- Entity Frameworks a.k.a. LINQ to Entities

- …

Module 3.4 Patterns

Example: Employee



Pattern Matching with `is`

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

The `is` keyword is now compatible with patterns

```
foreach (Employee e in all)
{
    if (e is SoftwareEngineer se)
    {
        Console.WriteLine(
            $"{se.FullName} has produced {se.CodeLinesProduced} lines of C#"
        );
    }
}
```

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:
        Console.WriteLine($"{sa.FullName} plays with Miro");
        break;
    case SoftwareEngineer se:
        Console.WriteLine($"{se.FullName} has a fun job coding all day");
        break;
    case null:
    default:
        break;
}
```

Switch Expressions

A new functionally-inspired `switch` expression

```
string Choose( Employee employee ) =>
    employee switch
    {
        SoftwareArchitect sa => $"Hello, Mr. Architect {sa.LastName}",
        SoftwareEngineer se => "Please code!",
        StudentProgrammer sp => $"Please get coffee, {sp.FirstName}",
        _ => "Have a nice day... :-)"
    }
}
```

- no fallthrough!
- `case` and `:` elements are replaced with `=>`
- `default` case is replaced with a `_`
- bodies can only be expressions (not statements!)

Additional Patterns for Matching

Property patterns	*Type*{ *p1: v1*, … , *pn: vn* }	e.g.	`{IsValid: false}`
Tuple patterns	(*x1*, … , *xn*)	e.g.	`(42, 87)`
Positional patterns	*Type*(*x1*, … , *xn*)	e.g.	`Album(s, age)`

Pattern are “**compositional**”!

Property Patterns

Property patterns match member properties to values

Also works for multiple, simultaneous name-value pairs

```
string Evaluate(SoftwareEngineer? se) =>
    se switch
    {
        { Level: Level.Lead } => $"{se.FullName} does great work",
        { Level: Level.Chief } => $"You da boss, {se.FullName}",
        null => "You're not even a software engineer, dude!",
        _ => $"Well done coding SOLID, {se.Level}... :-)"
    };
```

Property Pattern Variations

Can in fact simultaneously match the type as well…

Not tied to `switch` expressions: Also works for `is` etc.

```
string Evaluate( Employee employee ) =>
    employee switch
    {
        SoftwareEngineer { Level: SoftwareEngineerLevel.Lead } => $"...",
        SoftwareArchitect { Level: SoftwareArchitectLevel.Chief } => $"...",
        _ => $"Well done making the company thrive... :-)"
     }
}
```

Tuple Patterns

Tuple patterns use two or more values for matching

```
Hand left = GetRandomMember<Hand>();
Hand right = GetRandomMember<Hand>();

Outcome winner = (left, right) switch
{
    (Hand.Paper, Hand.Rock) => Outcome.Left,
    (Hand.Paper, Hand.Scissors) => Outcome.Right,
    (Hand.Rock, Hand.Paper) => Outcome.Right,
    (Hand.Rock, Hand.Scissors) => Outcome.Left,
    (Hand.Scissors, Hand.Paper) => Outcome.Left,
    (Hand.Scissors, Hand.Rock) => Outcome.Right,
    (_,_) => Outcome.Tie
};
```

Positional Patterns

Positional patterns use deconstructors for matching

Can be simplified using `var`

```
string description = album switch
{
    Album(string summary, int age) when age >= 25 => $"{summary} is vintage <3",
    Album(string summary, int age) when age >= 10 => $"{summary} is seasoned",
    Album(string summary, _) => $"{summary} is for youngsters only! ;-)"
};
```

Boolean and Relational Patterns for Matching

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);
    }
}
```

Parenthesized Patterns

This is simply a means to disambiguate parsing

Carries no semantic meaning in itself

```
string WhatIsIt(object o) =>
    o switch
    {
        (((string))) => "string",
        (((int))) => "int",
        _ => "Something else :-)",
     };
```

Conjunctive Patterns

Conjunctive patterns specify an and between patterns

```
string evalution = employee switch
{
    (not ProjectManager) and (not StudentProgrammer) =>
        "Codes somewhat",
    _ => "Probably codes a little less..."
};

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

Disjunctive Patterns

Disjunctive patterns specify an `or` between patterns

```
IEnumerable<object> elements = [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?"
};
```

Sequence Patterns for Matching

There are additional list patterns or enhancements:

List patterns	[*a*,*b*,*c*]	e.g. `[11,22,33]`
Slice (or range) patterns	..	e.g. `[11, ..]`

List Patterns

Can now match sequences against specific element patterns

```
List<int> elements = [ 11, 22, 33 ];

Console.WriteLine(elements is [11, 22, 33]);
Console.WriteLine(elements is [11, 22, 33, 44]);
Console.WriteLine(elements is [>10, <100, 33 or 44]);
```

Works for types which are *countable* and *indexable*

Discard pattern `_` can be used to match single elements in list patterns

```
Console.WriteLine(elements is [11, _, 33]);
Console.WriteLine(elements is [11, _, _, _]);
```

Slice Patterns

The Slice (a.k.a. Range) Pattern `..` can be used *at most once* within a list pattern

```
List<int> elements = [ 11, 22, 33 ];

Console.WriteLine(elements is [11, 22, ..]);
Console.WriteLine(elements is [.., 33, 44]);
Console.WriteLine(elements is [11, ..] or [.., 44]);
```

Works for types which are *countable* and *sliceable*

Slice elements can also be extracted

```
if( elements is [11, ..var sub, _])
{
    // Print sub here
}
```

Module 3.5
Object Lifetime

Deallocating Objects

There is no construct in C# to explicitly destroy objects

- Like in Python there is a garbage collector reclaiming memory of unused objects

The garbage collector *finalizes* the objects back into unused memory

The cleanup logic for objects is performed in the `Finalize()` method

- This virtual method cannot be overridden or called directly
- Implement a *class destructor* to override `Finalize()`

If present, the garbage collector will invoke destructor just before turning object back into unused memory

Destructors in C#

Similar to constructors, destructors have no return type

- No access modifier is allowed
- Just a single destructor (with no parameters!) is allowed

```
class DataWriter
{
    private FileStream _fs;
    ...
    ~DataWriter()
    {
        _fs.Close();
    }
}
```

Be Careful Out There!

The finalization process takes place after “ordinary” garbage collection

If your class has only managed resources, you should never use a destructor!

Avoid destructors whenever possible
- Costs time
- Hard to debug
- Prolongs object life and memory usage
- Makes your program multithreaded!

Cannot know exactly when finalization takes place…!

Deterministic Clean Up

Implement `IDisposable` for deterministic clean up

Invoking `Dispose()` deinitializes object (but does not reclaim the memory!)

```
namespace System;

public interface IDisposable
{
    void Dispose();
}
```

The Using Statement

Need to make sure that the `Dispose()` is always invoked when done with object

The using statement ensure this

- Can be used both with and without an explicit scope

```
using (DataWriter dataWriter = new())
{
    Console.WriteLine("Writing byte...");
    dataWriter.Write();
} // <-- Dispose()
```

Module 3.6 Tasks

Tasks

Tasks and asynchronous programming are very much like what you know from Python

In fact: `async` and `await` in Python were modelled after the same constructs in C#

`Task`	captures a task
`Task<T>`	captures a task returning a result of type T

```
Task task1 = Task.Run(() => { ... });

Task task2 = new(() => { ... });
task2.Start();

Task<DateTime> t = Task.Run(() => DateTime.Now);
Console.WriteLine(t.Result); // <-- Blocking!
```

Async and Await

Never use
- `.Result`
- `.Wait()`

unless absolutely sure task is stopped, because these are blocking!

They throw `AggregateException` instead of “regular” exception

Instead: Use `await` and `async` as for Python!

```
async Task<string> GetTextAsync(string url)
{
    HttpClient client = new();
    string result = await client.GetStringAsync(url);
    return result[..256];
}
```

CancellationToken

A `CancellationToken` is small token signalling cancellation

- Propagated to tasks and asynchronous operations through (last) method parameter

Controlled by a `CancellationTokenSource`

Note: ASP.NET Core operations accept `CancellationToken`

```
async Task<string> GetTextAsync(string url, CancellationToken cancellationToken)
{
    HttpClient client = new();
    string result = await client.GetStringAsync(url, cancellationToken);

    return result[..256];
}
```

Summary

00 Incoming Questions
01 Records
02 Lambdas
03 LINQ
04 Patterns
05 Object Lifetime
06 Tasks

Thank you