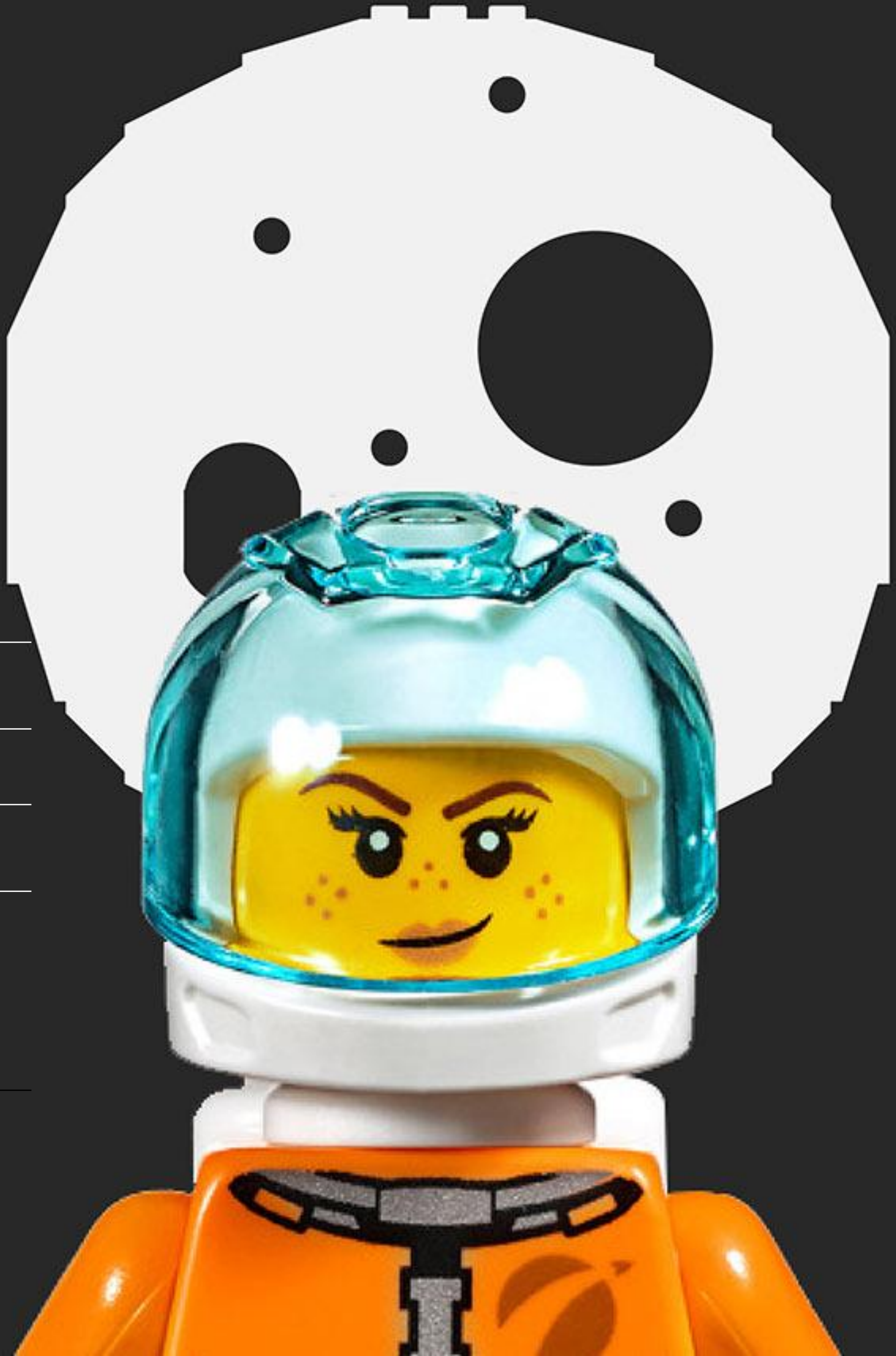
Modern C# For Python Developers

Session 2

September 10, 2025

Jesper Gulmann Henriksen

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



Agenda for Session 2

Incoming

- Solutions, Files, and Namespaces

2.1 Inheritance

- Subclasses
- Base and Protected
- Overriding Members
- Controlling Inheritance

2.2 Exceptions

- Exceptions
- Built-in and User-defined
- Try-Catch-Finally
- Throw
- Inner Exceptions
- Exception Filters

2.3 Structs

- Structs vs. Classes
- Readonly

2.4 Generics

- Object, ToString(), and Boxing
- Syntax
- Generic Methods
- Generic Types
- Constraints

2.5 Interfaces

- Implementation
- IEnumerable as an Example
- Default Interface Implementation
- Static Members
- Classes vs. Interfaces

2.6 Collections

- Built-in
- Collection Initializer Syntax
- Index and Range Expressions
- Spans
- Collection Expressions
- Spread Operator

Incoming Questions

Q: Solutions, Files, and Namespaces

Question:

- *“In Python everything is built upon the file structure, where an include is a reference to a module which is a file.*
- *In C# the location of the files are not that important, but the namespaces are as they are related to the solution.*
- *However, you must manually make references between projects – even for projects in the same solution!*
- *It seems that C# is very dependent upon the IDE having a Solution Explorer?*
- *Can we cover this in Session 2?”*

A: Solutions, Files, and Namespaces

- In C# files and namespaces are completely independent
  - Files can contain multiple namespaces
  - Namespaces can be split in multiple files
    - Even classes can be “partial” and split into multiple files, in fact

- A single unit of compilation is always a project compiling to an “assembly” (.exe or .dll)
- If a project is to call another, yes, an **explicit reference** must be made in the project file

```
<ItemGroup>
    <ProjectReference Include="..\Library\Library.csproj" />
</ItemGroup>
```

- Solutions don’t really provide anything for the compilation between projects
  - It is just a logical grouping for developer “convenience”
  - They could actually be eliminated: Projects compile fine without solutions ☺
  - VS Code does not have a Solution Explorer (but can be installed through “C# Dev Extension”)

Module 2.1 Inheritance

Inheritance

Inheritance is specified explicitly as with a ':'

C# allows only single inheritance

- Can implement multiple interfaces (Later in Session 2)

```
class SoftwareEngineer : Employee
{
    public int CodeLinesProduced { get; set; }
}
```

Base and Protected

Additional access modifier

- `protected` Visible inside class itself and subclasses `_method_()`

`base` is somewhat equivalent to `super` and `__super__()`

```
class SoftwareEngineer : Employee
{
    protected int CodeLinesProduced { get; set; }

    [SetsRequiredMembers]
    public SoftwareEngineer(string firstName, string lastName, int codeLineProduced = 0)
        : base(firstName, lastName)
    {
        CodeLinesProduced = codeLineProduced;
    }
}
```

Overriding Members

Unlike Python we must **explicitly** declare the ability to override methods

- `virtual` ~ subclasses can override
- `abstract` ~ subclasses must override
- `override`
- `sealed` ~ subclasses cannot override further, i.e. “virtual” stops here

```
class Employee
{
    ...
    public override string ToString()
    {
        return $"{FirstName} {LastName}";
    }
}
```

Controlling Inheritance

Similar keywords also to control inheritance

- `abstract` ~ must derive class
- `sealed` ~ cannot derive class

```
Employee employee = new Employee("John", "Doe"); // <-- Does not compile!

abstract class Employee
{
    ...
    public Employee(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }
}
```

Module 2.2 Exceptions

Exception Hierarchy

Exceptions in C# are objects derived from the built-in `System.Exception` type

Arranged in OO hierarchy inheriting members and properties from base:

- `Exception`
  - `SystemException`
    - `ArithmeticException`
      - `DivideByZeroException`
      - `...`
    - `FormatException`
    - `...`
  - ~~`ApplicationException`~~

Custom Exceptions

Define custom exception by deriving from “best” existing exception

```
class InsufficientFundsException(
   BankAccount account,
   string? message = null,
   Exception? inner = null
) : Exception(message, inner)
{
    public BankAccount Account { get; } = account;
}
```

Try-Catch-Finally

Very close in spirit to try-except-finally

“Empty” (or “generic”) exception match allowed in `catch`

```
try
{
    Bank.TransferFunds(from, 200, to);
}
catch (InsufficientFundsException exception)
{
    Console.WriteLine($"Only {exception.Account.Balance} in account");
}
finally
{
    Console.WriteLine("Done processing transaction...");
}
```

Throw

Corresponds to raise

- But with a slight word of warning…

```
try
{
    Bank.TransferFunds(from, 50, to);
    Console.WriteLine("Successfully transferred funds");
}
catch (InsufficientFundsException exception)
{
    Console.WriteLine($"Only {exception.Account.Balance} in account");
    throw;
}
```

Inner Exceptions

Good practice to include inner exception when “changing” exception at extension points

```
try
{
    from.Withdraw(amount);
    to.Deposit(amount);
}
catch (InsufficientFundsException exception)
{
    throw;
}
catch (Exception exception)
{
    throw new BankException("Could not complete transfer", exception);
}
```

Exception Filters

Good practice to only catch exceptions that can in fact be handled

```
try
{
    from.Withdraw(amount);
    to.Deposit(amount);
}
catch (InsufficientFundsException exception) when (exception.Account.IsVIP)
{
    Console.WriteLine("Don't worry, rich kid. We've got you covered!");

    // Handle VIP account...
}
```

Module 2.3
Structs

Structs

Structs are like classes – but are value types!

Structs (and methods in structs!) can be **readonly** (unlike classes).

Structs ~ capture values
Classes ~ capture objects (with identity)

```
readonly struct Money
{
    public int Euro { get; init; }
    public int Cents { get; init; }

    public override string ToString()
    {
        return $"EUR {Euro}.{Cents:d2}";
    }
}
```

Module 2.4 Generics

Boxing

All objects ultimately derive from object

- All values can be “boxed” as an object (by copying)
- Boxed values can be “unboxed” back to its original value (only!)

Boxed objects can only perform general object methods

```
int i = 87;
object o1 = i; // Boxing
Console.WriteLine(o1.ToString());

int j = (int) o1; // Unboxing
Console.WriteLine(++j);
```

Stack

Collection classes can be built on the `object` type.

But you can insert **anything** into the collection.

```
Stack stack = new();
stack.Push(new Person { FirstName = "John", Age = 42 });
stack.Push(new Person { FirstName = "Jane", Age = 87 });

...

Person? top = stack.Peek() as Person;
Person? removed = stack.Pop() as Person;
foreach (Person person in stack)
{
    Console.WriteLine(person.FirstName);
}
```

Stack<T>

Wouldn't it be great if we

- … we only needed to construct each type once?
- … and it had no (un)boxing performance hit?
- … and everything was still type-safe?

```
Stack<Person> stack = new();
stack.Push(new Person { FirstName = "John", Age = 42 });
stack.Push(new Person { FirstName = "Jane", Age = 87 });

Person top = stack.Peek();
Person removed = stack.Pop();
foreach (Person person in stack)
{
    Console.WriteLine(person.FirstName);
}
```

Generic Methods

We can easily create generic methods

Compiler will try to infer actual type at call site

```
void Swap<T>(ref T a, ref T b)
{
    (a, b) = (b, a);
}
```

Generic Types

Generic types can be defined similarly

```
Point<int> pt1 = new(42, 87);
Point<double> pt2 = new(11.2, 8.7);

readonly struct Point<T>
{
    public T X { get; init; }
    public T Y { get; init; }

    public Point(T x, T y)
    {
        X = x;
        Y = y;
    }
}
```

Default

The `default` keyword indicates the default value of the generic type

```
readonly struct Point<T>
{
    ...
    public static Point<T> Zero
    {
        get
        {
            return new(default, default);
        }
    }
}
```

Struct and Class Generic Constraints

The struct and class constraints indicates kind of generic type

```
readonly struct Point<T> where T : struct
{
    ...
    public static Point<T> Zero
    {
        get
        {
            return new(default, default);
        }
    }
}
```

Other Generic Constraints

Generic Constraint	Description
`where T : struct`	T must ultimately derive from `System.ValueType`
`where T : class`	T must be a reference type
`where T : new()`	T must have a default constructor
`where T :` *BaseClass*	T must derive from the class specified by *BaseClass*
`where T :` *Interface*	T must implement the interface specified by *Interface*
`where T : notnull`	T is a non-nullable type

- Multiple constraints can be separated by commas
- There can be only one *BaseClass*, but many *Interfaces*

- https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/constraints-on-type-parameters

Module 2.5 Interfaces

Introducing Interfaces

C# defines interfaces which are essentially “code-free contracts” or “obligations”

- A type can only derive from a **single class**, but implement **multiple interfaces**

Note: Python doesn’t need interfaces, because of multiple inheritance + duck typing.

```
interface ICanBeCleared
{
    void Clear();
}

struct Point<T>(T x, T y) : ICanBeCleared
    where T : INumber<T> { ... }

class Person(string firstName, int? age = null) : ICanBeCleared { ... }
```

IEnumerable<T> and Foreach

Interfaces provided “extended” functionality for types and keywords throughout C# and .NET

Example: The foreach keyword can iterate over anything implementing `IEnumerable<T>`

```
interface IEnumerable<out T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}

interface IEnumerator<T>
{
    T Current { get; }
    bool MoveNext();
    void Reset();
}
```

Implementing IEnumerable<T>

There is special syntax for implementing `IEnumerable<T>`

```
class Family : IEnumerable<Person>
{
    ...

    public IEnumerator<Person> GetEnumerator()
    {
        yield return _persons[0];
        yield return _persons[1];
        yield return _persons[2];
    }
}
```

Static Abstract Interface Members

These make better sense and are cleaner

```
interface ICanBeEmpty<T>
{
    static abstract T Empty { get; }
}

class Person : ICanBeEmpty<Person>
{
    public static Person Empty
    {
        return new Person { ... };
    }
    ...
 }
```

Module 2.6 Collections

Collection Classes

`System.Collection.Generic` contains a number of built-in collection classes and interfaces

- https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic?view=net-8.0

Most prominent all implement `IEnumerable<T>` and include

```
List<T>                   IList<T>
Dictionary<K,V>           IDictionary<K,V>
HashSet<T>                ISet<T>

Queue<T>
Stack<T>
```

Example: Dictionary<K,V>

Note: .NET uses `object.GetHashCode()` for key storage and lookup!

```
Dictionary<int, string> dict = new();
dict.Add(75192, "Millennium Falcon");
dict.Add(21318, "Tree House");
dict.Add(51515, "Robot Inventor");
Console.WriteLine($"Number 51515 is \"{dict[51515]}\""+ Environment.NewLine);

foreach (KeyValuePair<int, string> kv in dict)
{
    Console.WriteLine($"Product {kv.Key} is \"{kv.Value}\"");
}
```

Index Initializer Syntax

Indexed collection such as dictionaries have a specific initialization syntax available

```
Dictionary<int, string> dict = new()
{
    [75192] = "Millennium Falcon",
    [21318] = "Tree House",
    [51515] = "Robot Inventor"
};
```

Collection Initializer Syntax

Any collection equipped with an `Add()` method can make use of collection initializer syntax

```
List<int> list = new()
{
   0, 1, 2, 3, 4, 5, 6, 7, 8,
};

HashSet<int> set = new()
{
    42, 87, 42, 112, 176, 176, 176, 87
};

Dictionary<int, string> dict = new()
{
    { 21318, "Tree House" },
    { 51515, "Robot Inventor" }
};
```

Index

Very inspired by Python: The ^ operator describes the end of the sequence (like e.g. −2 in Python)

Indices are captured by a new `Index` type

```
string[] elements = new string[]
{
   "Hello", "World", "Booyah!", "Foobar"
};

Console.WriteLine(elements[^1]);
Console.WriteLine(elements[^0]); // ^0 == elements.length

Index i = ^2;
Console.WriteLine(elements[i]);
```

Range

"Inspired" by Python: The `..` operator specifies (sub)ranges (like e.g. `0:2` in Python)

Ranges are captured by a new `Range` type

`i..j`	Full sequence	(start is inclusive, end is exclusive)
`i..`	Half-open sequence	(start is inclusive)
`..i`	Half-open sequence	(end is exclusive)
`..`	Entire sequence	(equivalent to `0..^0`)

```
foreach (var s in elements[0..^2])
{
    Console.WriteLine( s );
}

Range range = 1..;
```

Spans

Ref-like types to avoid allocations on the heap ~ `memoryview`

```
int[] array = new int[10] { ... };
Span<int> span = array.AsSpan();
Span<int> slice = span.Slice(2, 5);
foreach (int i in slice) { ... }

string s = "Hello, World";
ReadOnlySpan<char> span = s.AsSpan();
ReadOnlySpan<char> slice = span.Slice(7, 5);
foreach (char c in slice) { ... }
```

Supported Collection Types for Index and Range

Supported by any type that provides an indexer with an `Index` or `Range` parameter

`string`	Indices	Ranges
`Array`	Indices	Ranges
`List<T>`	Indices	
`Span<T>`	Indices	Ranges
`ReadOnlySpan<T>`	Indices	Ranges

Collection Expressions

New, unified collection syntax across a multitude of collection types

```
List<string> elements = ["Hello", "World", "Booyah"];

class LookupTable(List<string> elements)
{
    public string Get(Index index)
    {
        return elements[index].ToUpper();
    }

    public LookupTable() : this([])
    {
    }
}
```

Supported Collection Types for Expressions

- Arrays
- `Span<T>` and `ReadOnlySpan<T>`
- Types with collection initializer, such as `List<T>` and `Dictionary<K,V>`

(and actually more such as `ImmutableArray<T>` and custom types)

```
int[] array = [1, 2, 3, 4, 5, 6, 7, 8];

List<string> list = ["one", "two", "three"];

Span<char> span = ['a', 'b', 'c', 'd', 'e', 'f', 'h', 'i'];

int[][] array2d = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];

// Create an enumerable? (WTF?!)
IEnumerable<int> enumerable = [1, 2, 3];
```

Spread Operator

The **spread operator** replaces its collection argument with the individuals elements from that collection

```
int[] row0 = [1, 2, 3];
List<int> row1 = [4, 5, 6];
IEnumerable<decimal> row2 = [7.1m, 8.2m, 9.3m];

decimal[] all = [.. row0, .. row1, .. row2];

foreach (var element in all)
{
    Console.WriteLine(element);
}
```

Summary

00 Incoming Questions
01 Inheritance
02 Exceptions
03 Structs
04 Generics
05 Interfaces
06 Collections

Thank you