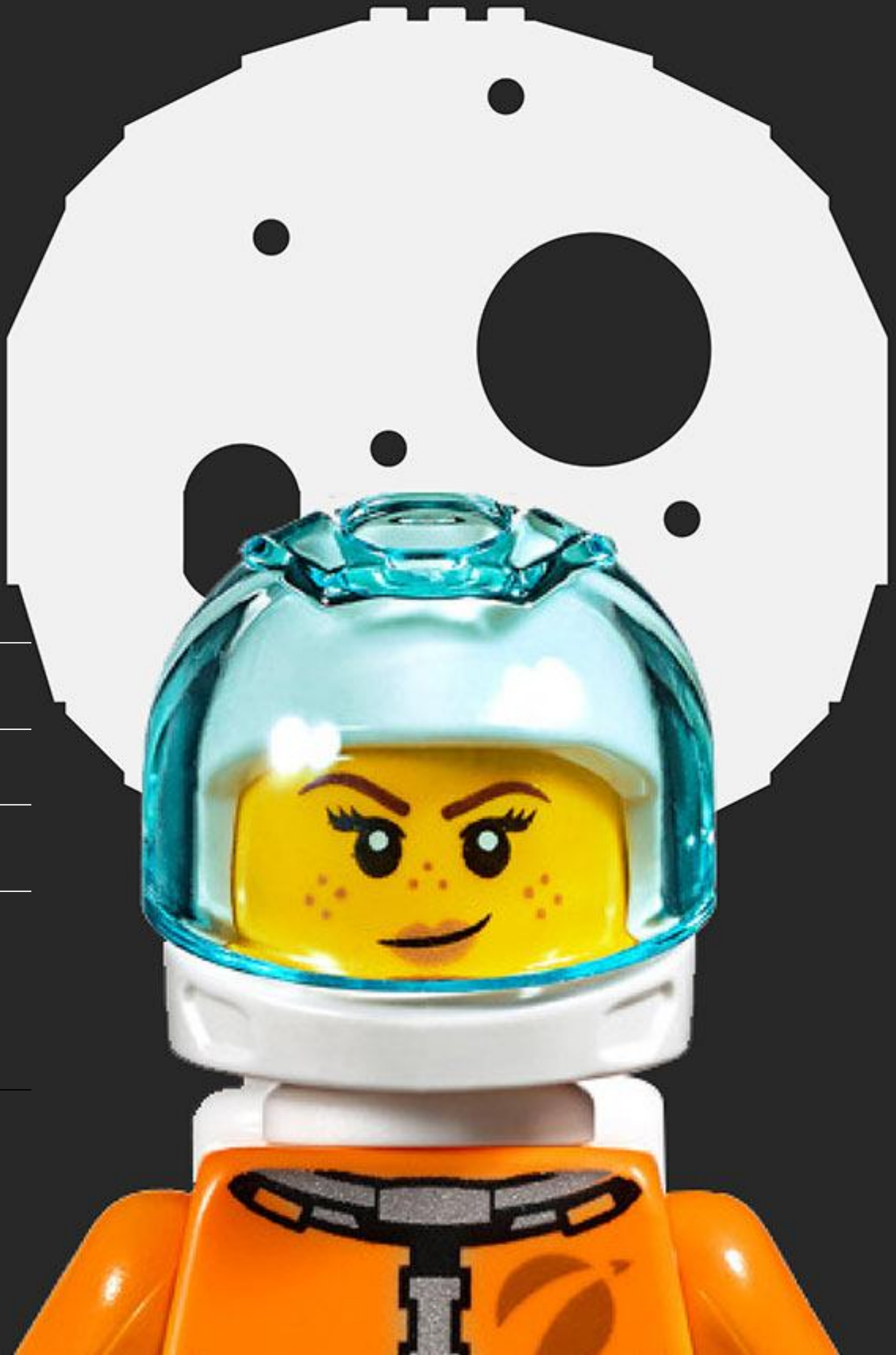
Modern C# For Python Developers

Session 1: Object-Oriented C#

September 3, 2025

Jesper Gulmann Henriksen

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



Agenda for Session 1: Object-Oriented C#

1.1 Introduction

- What is Essentially Different?
- C# vs. .NET
- IDEs for .NET

1.2 Hello, World

- White-space
- Casing
- Block scopes
- Namespaces
- Top-level Statements

1.3 Types

- Value vs. Reference Types
- Variables, scopes, and typing
- Nullable Value Types
- Nullable Reference Types

1.4 Strings

- Formatting
- Interpolations
- Raw String Literals
- Strings are Strange

1.5 Methods

- Parameter Modifiers
- Local functions
- Method Overloading
- Optional and Named parameters

1.6 Classes

- Classes and Properties
- Access Modifiers
- Constructors
- Object Initializer Syntax
- Required
- Deconstructors
- Static Members
- Inheritance
- Overriding

1.7 Exceptions

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

1.8 Structs

- Structs vs. Classes
- Readonly

Module 1.1 Introduction

Similarities Between C# and Python

- Object-oriented
- Cross-platform
- Garbage Collection
- Strongly typed
- Async and Await
- Pattern matching `match ~ switch`
- Statement keywords `if`, `else`, `while`, …

Differences between C# and Python

- Indentation vs. tokens
- Static Typing
- Nullable Types
- LINQ

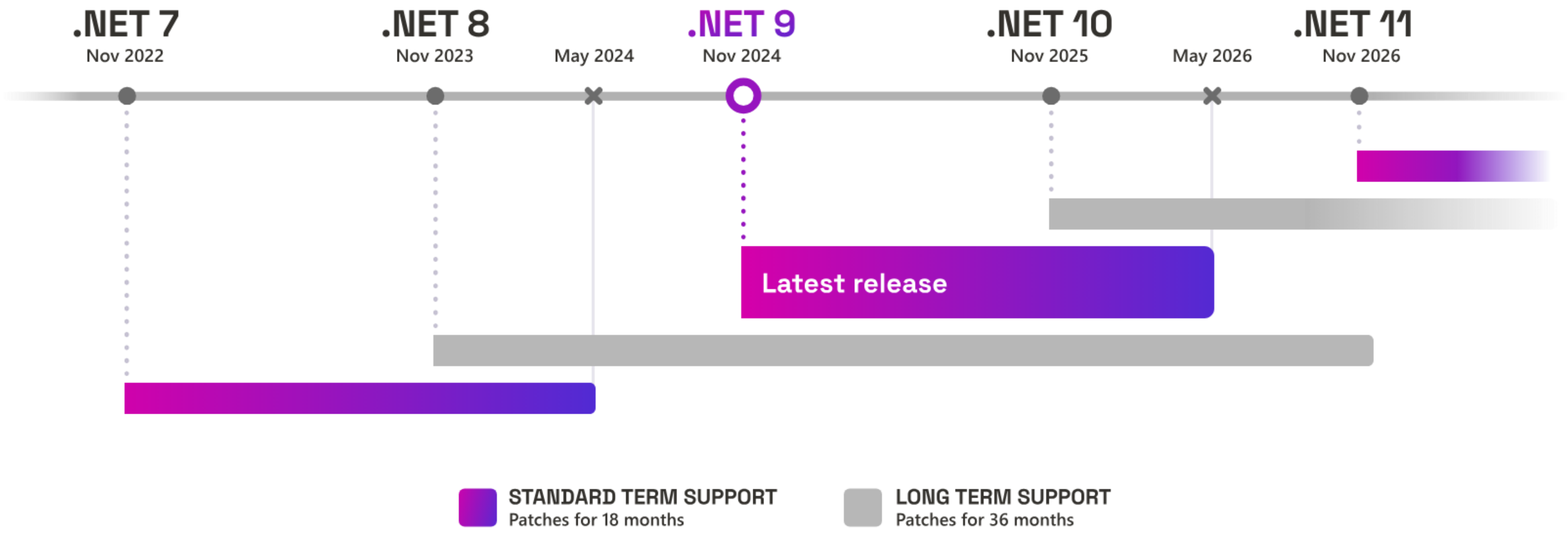
itertools
more-itertools
pylinq

- Generics

Missing from C#

- Structural typing (i.e. “duck” typing)
- REPL
- Significant whitespace

.NET Release Cadence



C# and .NET Versions

Target	.NET Version	C# Version
.NET	10.x	C# 14
.NET	9.x	C# 13
.NET	8.x	C# 12
.NET	7.x	C# 11
.NET	6.x	C# 10
.NET	5.x	C# 9.0
.NET Core	3.x	C# 8.0
.NET Core	2.x	C# 7.3
.NET Standard	2.1	C# 8.0
.NET Standard	2.0	C# 7.3
.NET Standard	1.x	C# 7.3
.NET Framework	All	C# 7.3

IDEs for .NET (Core) Development

- MacOS
  - VS Code
  - Rider
  - Visual Studio for Mac

- PC
  - VS Code
  - Visual Studio 2022
  - Rider

- Linux
  - VS Code
  - Rider

Requires paid license

Being discontinued

Module 1.2
Hello, World

Hello, World

C# has traditionally had the worst Hello, World ever:

```
using System;
using System.Collections.Generic;
using System.Text;
...
namespace ModernCS
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}
```

Worth Noting

- Case-sensitive, but white-space insensitive
- Indention is ignored, so block scopes are created using `{` and `}`

- All types are located in a `namespace` (or global)
- `using` imports names into scope
- Namespaces are unrelated to “import” of packages

- Methods must exist within types

Solution	~ collection of projects
Project	~ “assembly”, e.g. .exe or .dll

File-scoped Namespaces

The `namespace` declarations have been “horizontally optimized”

```
using System;

namespace ModernCS;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello, World!");
    }
}
```

Global Usings

The `using` can be global within project unit

```csharp
global using System;

namespace ModernCS;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello, World!");
    }
}
```

Implicit Usings

The implicit `using` are enabled in IDE or .csproj file

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
    <RootNamespace>ModernCS.Session1</RootNamespace>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

Top-level Statements

Top-level Statements automatically compiles code as part of `Program.Main()`

Namespace is optional (and not needed here!)

Finally… Meet the “modern” C# Hello, World ☺

```
Console.WriteLine("Hello, World!");
```

Module 1.3
Types

Value Types

int

bool

double

DateTime

enum

struct

Reference Types

object

string

Exception

delegate

record

class

Strictness and Scope of Variables

- Must be declared of a specified type
- Must be initialized before read

- Variable scope is confined to the defining block

```
bool isSet = true;

if (isSet)
{
    int a = 87;
    Console.WriteLine(a);
}

Console.WriteLine(a); // <-- Will not compile!
```

Use Keyword for Types

Can use both fully qualified type name for all types

But always better to use keyword when available..!

```
string s = "Hello";
bool b = false;

DateTime dt = DateTime.Now;
```

Compiler-inferred Variable Types

Compiler can infer variable type from object type using `var`

```
var isSet = true;

if (isSet)
{
    var a = 87;
    Console.WriteLine(a);
}
```

Dynamically Typed Variables

Discouraged in C# but local dynamic typing can be enabled via `dynamic` keyword

Use only for

- Interoperability with a dynamic language
- To overcome C# type system being too strict

```
dynamic d = 87;
d = "Does this compile? Yes!";
d.ThrowsRuntimeError();
```

Nullable Value Types

Any value type has a nullable version by using `?` in the type definition

Essentially extends set of values for type with `null`

- Does not make it a reference type!

`??` is the null-coalescing operator *“get value if exists; else provide default”*

```
int? i = 87;
int? j = null;

if (i.HasValue)
{
    int k = i.Value + j ?? 42;
    Console.WriteLine(k);
}
```

Nullable Reference Types

Perhaps surprisingly there are nullable **reference** types too!

`?.` is the null conditional operator
`!` is the null-forgiving operator

```
string firstName = "Bruce";
string? middleName = null;
string lastName = "Campbell";

string fullName = $"{firstName} {middleName} {lastName}";
Console.WriteLine(fullName);

Console.WriteLine(middleName?.Length ?? 0);
Console.WriteLine(middleName!.Length);
```

Module 1.4
Strings

Strings

- Can be
  - concatenated,
  - formatted,
  - escaped, and
  - interpolated corresponds to f".."

```
string firstName = "Bruce";
string lastName = "Campbell";

string name1 = firstName + " " + lastName;
string name2 = string.Format("{0} {1}", firstName, lastName);
string name3 = $"{firstName} {lastName}";

string escaped = "This is a \t \\tab\\ with newline\r\n";
string verbatim = @"This is a \t \\tab\\ with newline\r\n";
```

Raw String Literals

Strings now support multi-line string literals using """

```
string s = """
    Hello,
    "World"
    """;

Console.WriteLine(s);
```

- Excellent for e.g. JSON or XML string literals
- Blocks of *n* "'s in strings can be escaped using *n*+1 "'s in begin and end

- Indentions can also be controlled by ending white-space before """

What about String Interpolation?

String interpolation proceeds as usual, but might need $$ and {{}} (or more ☺)

```
string firstName = "Jesper";
string lastName = "Gulmann Henriksen";
string company = "LEGO Group";

string s = $$"""
    {
      "firstName": "{{firstName}}",
      "lastName": "{{lastName}}",
      "company": "{{company}}"
    }
    """;

Console.WriteLine(s);
```

Strings are “Strange”

- There are some pitfalls to strings!

- Immutable
- Reference type

- Equality

- Use `StringBuilder` for gradually building large strings

Module 1.5 Methods

Methods

Methods in C# are slightly different than in methods in Python

- Methods can be local, but not global
- There is no “self” argument for class methods

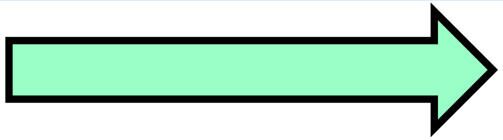
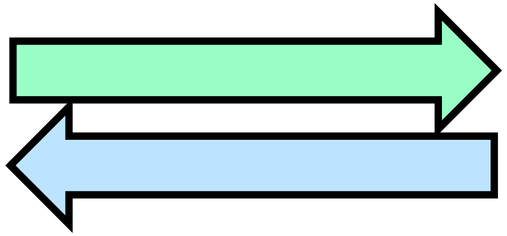
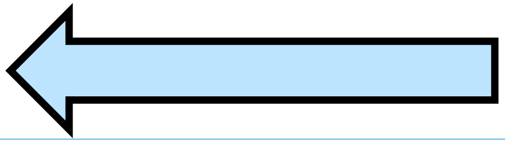
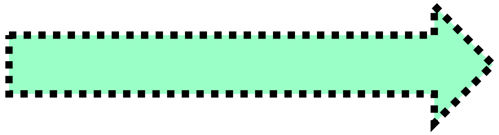
- All parameters are by default passed as “by value”, i.e. copied

```
int x = 42;
Twice(x);
Console.WriteLine($"x={x}");

static void Twice(int x)
{
    x = 2 * x;
}
```

Parameter Modifiers

Default behavior of method passing can be changed using parameter modifers

Modifier	Effect	Description
		Copies argument to formal parameter
`ref`		Formal parameters are synonymous with actual parameters. Call site must also specify `ref`
`out`		Parameter cannot be read and must be assigned. Call site must also specify `out`
`in`		Parameter is "copied" and cannot be modified! Call site can optionally specify `in`

Params Modifier

Parameter lists of varying length can be passed by using the `params` modifier

```
Console.WriteLine(Sum(42, 87));

int Sum(params int[] values)
{
    int total = 0;
    foreach (int i in values)
    {
        total += i;
    }
    return total;
}
```

Method Overloading

```
class Calculator
{
    public static int Add(int x, int y)
    {
        return x + y;
    }
    public static int Add(int x, int y, int z)
    {
        return x + y + z;
    }
    public static double Add(double a, double b)
    {
        return a + b;
    }
}
```

Optional and Named Parameters

Parameters can be passed as named

```
FavoriteTeam("Liverpool", ConsoleColor.Red, "Premier League");
FavoriteTeam("Tottenham", league: "Premier League");
FavoriteTeam("AGF");

static void FavoriteTeam(
    string team,
    ConsoleColor color = ConsoleColor.White,
    string league = "Superliga"
)
{
    Console.ForegroundColor = color;
    Console.WriteLine($"I support {team} playing in the {league}");
}
```

Local Methods

Methods can be local, but not global

```
static void FavoriteTeam(string team, ConsoleColor color = ConsoleColor.White,
    string league = "Superliga")
{
    void Print()
    {
        Console.ForegroundColor = color;
        Console.WriteLine($"I support {team} playing in the {league}");
    }

    ConsoleColor old = Console.ForegroundColor;
    Print();
    Console.ForegroundColor = old;
}
```

Module 1.6
Classes

Classes and Properties

Classes are defined using properties and fields explicitly declared

Properties have accessors (methods with a special syntax)

- get
- set / init

Target-typed new is a convenient shorthand to avoid stating type twice

```
class Employee
{
    public string FirstName
    {
        get{ return _firstName; }
        set{ _firstName = value; }
    }
    private string _firstName;
}
```

Automatic Properties

95% of all properties have automatically generated `get` and `set` / `init`

```
class Employee
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

Access Modifiers

Any member property or method has an access modifier

`public`	Globally visible
`private`	Visible inside class
`internal`	Visible inside assembly
`...`	

`method()`
`__method__()`

Default is `private` for members and `internal` for types

```
public class Employee
{
    public string FirstName { get; private set; }
    public string LastName { get; private set; }

    private string Password { get; set; }
}
```

Constructors

Construction method named after type

- corresponds to `__init__()`
- `self` is implicit and not passed but accessed by `this` keyword if needed

```
Employee employee = new("John", "Doe");

class Employee
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

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

Primary Constructors

Recent addition to C# is the Primary Constructors which IDEs seem to love

Other constructors should call primary constructor using `this`

```
Employee employee = new("John", "Doe");

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

Object-initializer Syntax

Allows to create new object by setting properties explicitly

Properties with `init` can also be set in the object-initializer syntax

```
Employee employee = new()
{
    FirstName = "John",
    LastName = "Doe"
};

class Employee
{
    public string FirstName { get; init; }
    public string LastName { get; init; }
}
```

Required

There is a problem with non-nullability of members which we have ignored so far

`required` fixes the problem with non-nullability and object-initializer syntax

```
Employee employee = new()
{
    FirstName = "John",
    LastName = "Doe"
};

class Employee
{
    public required string FirstName { get; set; }
    public required string LastName { get; set; }
}
```

Setting Required Members in Constructors

Might have to employ `[SetsRequiredMembers]` on constructor to satisfy compiler

```
class Employee
{
    public required string FirstName { get; set; }
    public required string LastName { get; set; }

    [SetsRequiredMembers] // <-- C# "attribute" - not to be mistaken with Python attributes
    public Employee(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }
}
```

A Word of Warning on “Attributes”

C# “attributes” and Python “attributes” are used for distinct things…!

In C# attributes are metadata info about types, methods, variables etc. ~ @property

In Python attributes are properties associated with objects, i.e. variables or methods defined within a class or class instance.

Deconstructors

Reserved “duck-typed” feature to break objects into tuples

```
(string firstName, string lastName) = employee;
Console.WriteLine(firstName);

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

Static Members

Keyword `static` captures class-level members ~ *“shared”*

- In Python corresponds to variable declared outside of `__init__()` or instance method

```
class Employee
{
    private static int _nextEmployeeNumber = 100_000;
    public int Number { get; }
    ...

    public Employee()
    {
        Number = _nextEmployeeNumber++;
    }
}
```

Static Classes and Extension Methods

Classes can be `static` too

- Only allowed to contain static members (no instance members!)

Usually used to enable Extension Methods via `static-static-this`

```
int i = 87;
Console.WriteLine(i.IsEven());
Console.WriteLine(IntExtensions.IsEven(i));

static class IntExtensions
{
    public static bool IsEven(this int i)
    {
        return i % 2 == 0;
    }
}
```

Inheritance

Inheritance is specified explicitly as with a ':'

C# allows only single inheritance

- Can implement multiple interfaces (See Session 2 next time)

```
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 1.7 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 1.8
Structs

Structs

Structs are like classes – but are value types!

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}";
    }
}
```

Summary

01 Introduction
02 Hello, World
03 Types
04 Strings
05 Methods
06 Classes
07 Exceptions
08 Structs

Thank you