C# Overview

Lecture 2 - Enterprise Application Development

C# language

- C# (pronounced "See Sharp") is a modern, object-oriented, and type-safe programming language
- It has roots in the C family of languages and will be immediately familiar to C, C++, Java, and JavaScript programmers
- C# support both value and reference types, as well as generics, which provide increased type safety and performance

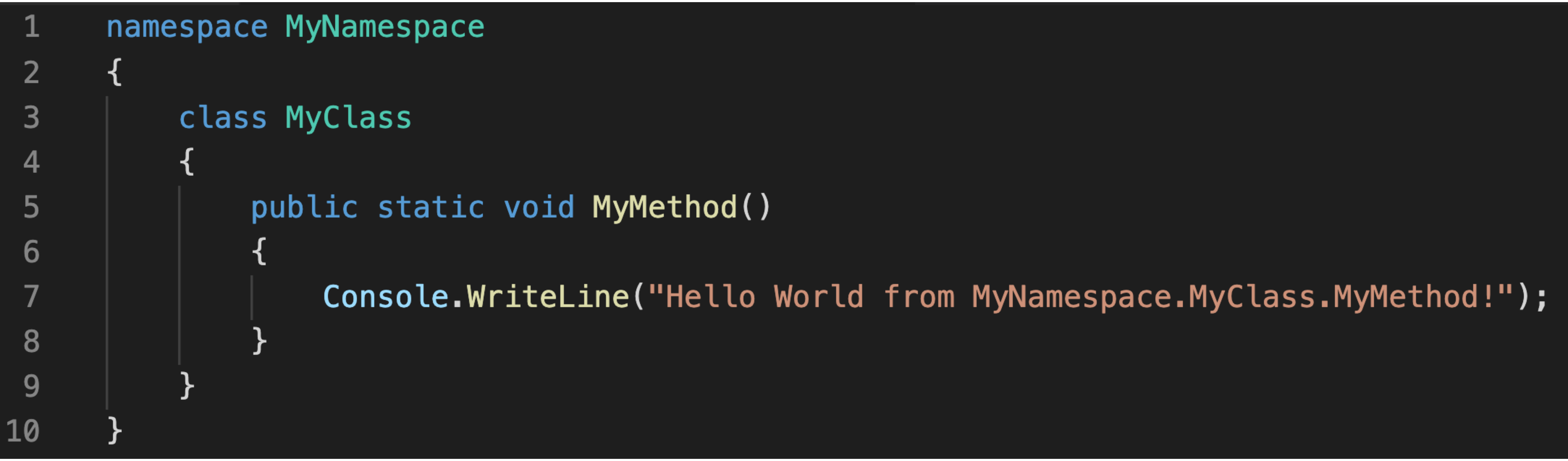
C# features

- **Garbage collection:** automatically reclaims memory occupied by unreachable unused object
- **Nullable types:** guard against variables that don't refer to allocated objects
- **Exception handling:** provides a structured and extensible approach to error detection and recovery
- **Lambda expressions:** support functional programming techniques

C# features

- **Language Integrated Query (LINQ):** syntax creates a common pattern for working with data from any source
- **Asynchronous operations:** provides syntax for building distributed systems
- **Unified type system:** meaning every variable and constant has a type, as does every expression that evaluates to a value.

Hello World in C#



```
namespace MyNamespace
{
    class MyClass
    {
        public static void MyMethod()
        {
            Console.WriteLine("Hello World from MyNamespace.MyClass.MyMethod!");
        }
    }
}
```

- Namespace is optional but highly recommended, as it allows us to group classes and prevents class name collisions when using external packages

Naming conventions

- **PascalCase:** when naming a **class**, **struct, (record), method**, **member vars**
- **camelCase:** when naming **parameters, local variables,** and for **private** or internal **fields**, (also prefix these with **_**)
- C# naming conventions:
  - https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/identifier-names
  - https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions

Namespaces

- This line prints a message to the console:

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

- **System** is a namespace and **Console** is a class in that namespace.

- The **using** keyword can be used so that the complete name isn't required, as in the following example:

```
using System;
Console.WriteLine("Hello World!");
```

C# type system

- Value types
  - Simple (primitive) types
    - Int, char, float, double, decimal and bool
  - Enum
  - Struct
  - Nullable value types
  - Tuple

- Reference types
  - Class
  - Interface
  - Array
  - Delegate

- Some implicit typing, eg.
  - for (**var** i = 0; i < 10; i++)
  - **var** greeting = "Hey"; (local)

Properties

- Properties are a form of smart fields in a class or object
- From outside the object, they appear like fields in the object
- Unlike fields, properties are implemented with accessors that define the statements executed when a property is accessed or assigned
- Properties are typically used for validation and controlling accessibility

Properties

- This is an example of **auto property** syntax. The compiler generates the storage location for the field that backs up the property. The compiler also implements the body of the get and set accessors:

```
public string FirstName { get; set; }
```

- You can also define the storage (field) yourself:

```
public string FirstName {
        get { return firstName; }
        set { firstName = value; }
}
private string firstName;
```

Classes

- A type that is defined as a class is a reference type

- This is example of class declaration

```
//[access modifier] - [class] - [identifier]
public class Customer {
      // Fields, properties, methods and events go here...
}
```

- A class and an object are different things although they are sometimes used interchangeably. A class defines a type of object, but it is not an object itself

Classes

- An object is a concrete entity based on a class and is sometimes referred to as an instance of a class

- Objects can be created by using the new keyword followed by the name of the class that the object will be based on, like this:

```
Customer object1 = new Customer();
```

- When a variable is declared, the value is set to **null** until a new instance is created

Enumeration (enum)

- An enumeration type (or enum type) is a value type defined by a set of named constants of the underlying int type

- You use an enumeration type to represent a choice from a set of mutually exclusive values or a combination of choices

- Example

```
enum Season { Spring, Summer, Autumn, Winter }
```

- The associated constant values of enum members are of type int; they start with zero and increase by one following the definition text order

Comments

- Single line

```
// This is a single line comment
```

- Multi line

```
/*
    This is a multiline
    comment
*/
```

- Documentation

```
/// <summary>
/// This is a documentation comment
///</summary>
```

  - Documentation generators can be used to produce XML from these comments

Methods

- A method is a code block that contains a series of statements
- It can take no, one or multiple parameters
- It can have a return type, or it can be declared as **void** (no return type)
- Methods can be either instance or static

Static Methods

```
public class Example {
    public static void Main() {
            // Call with an int variable.
            int num = 4;
            int productA = Square(num);

            // Call with an integer literal.
            int productB = Square(12);
            // Call with an expression that evaluates to int.
            int productC = Square(productA * 3);
    }
    static int Square(int i) {
            // Store input argument in a local variable.
            int input = i;
            return input * input;
    }
}
```

Instance Methods

```
TestMotorcycle slowMotocycle = new TestMotorcycle();
slowMotocycle.StartEngine();
slowMotocycle.AddGas(15);
slowMotocycle.Drive(5, 20);
double speed = slowMotocycle.GetTopSpeed();
Console.WriteLine("My top speed is {0}", speed); }

TestMotorcycle fastMotocycle = new TestMotorcycle();
fastMotocycle.StartEngine();
fastMotocycle.AddGas(80);
fastMotocycle.Drive(50, 200);
double speed = fastMotocycle.GetTopSpeed();
Console.WriteLine("My top speed is {0}", speed); }
```

Access Modifiers

- The accessibility level controls whether type members can be used from other code in your assembly or other assemblies
- **public:** The type or member can be accessed by any other code in the same assembly or another assembly that references it
- **private:** The type or member can be accessed only by code in the same class or struct
- **protected:** The type or member can be accessed only by code in the same class, or in a class that is derived from that class

Access Modifiers

- **internal:** The type or member can be accessed by any code in the same assembly, but not from another assembly

- We also have **protected internal** and **private protected** which are the combination of two access modifiers

- Default access modifier for classes and structs is **internal** and for their members is **private**

Pass value types by reference

- In C# value types are passed by value by default

- This means the value can be changed in the method, but the changed value will not be retained when control passes back to the calling procedure.

- We can also pass them by reference

- For that we need to use **in**, **ref** or **out**

Pass value types by reference

- **in:** specifies that this parameter is passed by reference but is only read by the called method, it cannot be modified

```
int readonlyArgument = 44;
InArgExample(readonlyArgument);
Console.WriteLine(readonlyArgument);  // value is still 44 void

InArgExample(in int number)
{
    // Uncomment the following line to see error CS8331
    //number = 19;
}
```

Pass value types by reference

- **ref:** specifies that this parameter is passed by reference and may be read or written by the called method

```
int number = 1;
Method(ref number);
Console.WriteLine(number); // Output: 45

void Method(ref int refArgument)
{
    refArgument = refArgument + 44;
}
```

Pass value types by reference

- **out:** specifies that this parameter is passed by reference and is written by the called method. It’s similar to **ref**, the difference is that **ref** requires the variable to be initialized

```
int initializeInMethod;
OutArgExample(out initializeInMethod);
Console.WriteLine(initializeInMethod); // value is now 44 void

OutArgExample(out int number)
{
    number = 44;
}
```

Generic classes and methods

- Generics introduces the concept of type parameters to .NET, which make it possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code

- Use generic types to maximize code reuse, type safety, and performance

- By using a generic type parameter **T**, you can write a single class that other client code can reuse by specifying a type

Generic classes and methods

- Example:

```
static void Swap<T>(ref T lhs, ref T rhs)
{
    T temp;
    temp = lhs;
    lhs = rhs;
    rhs = temp;
}
```

Generic collections

- Most popular generic collection is **List**

```
List<Employee> employees = new List<Employee>();
List<int> ids = new List<int>();
List<float> prices = new List<float>();
```

- There are other generic collections too
  - Stack
  - Queue
  - Dictionary
  - HashSet

Generic collections

- There are other generic collections too
  - Stack
  - Queue
  - Dictionary
  - HashSet

- Generic collections implement IEnumerable<T> interface
  - This allows us to reuse methods that operate with generic collections. For example, we can create a method that prints values in a collection regardless which collection we pass it as a parameter

Generic collections

```
using System;
using System.Collections.Generic;
class Program
{
    static void Main()
    {
        Display(new List<bool> { true, false, true });
    }

    static void Display(IEnumerable<bool> argument)
    {
        foreach (bool value in argument)
        {
            Console.WriteLine(value);
        }
    }
}
```

- When working with generics we need to use System.Collections.Generic namespace
- Display method can take any collection of bool and display its values

LINQ

- LINQ stands for Language-Integrated Query

- A query is an expression that retrieves data from a data source

- All LINQ query operations consist of three distinct actions:
    - Obtain the data source.
    - Create the query.
    - Execute the query.

- There are two ways to write LINQ queries
    - Query syntax
    - Method syntax

LINQ

```
class QueryVsMethodSyntax {
        static void Main() {
                int[] numbers = { 5, 10, 8, 3, 6, 12};

                //Query syntax:
                IEnumerable<int> numQuery1 =
                        from num in numbers
                        where num % 2 == 0
                        orderby num
                        select num;

                //Method syntax:
                IEnumerable<int> numQuery2 = numbers.Where(num => num % 2 == 0).OrderBy(n => n);
        }
}
```

Resources

- Documentation
  - https://docs.microsoft.com/en-us/dotnet/csharp/
- Online courses
  - Linkedin Learning: https://www.linkedin.com/learning/learning-c-sharp-8581491
- Books
  - Programming C# 8.0: Build Windows, Web, and Desktop Applications, Ian Griffiths
  - C# in Depth, Jon Skeet