01
.NET Fundamentals

Giovanni Ciatto
giovanni.ciatto@unibo.it

Dipartimento di Informatica – Scienza e Ingegneria
Alma Mater Studiorum—Università di Bologna, Cesena

a.a. 2022/2023

Outline

What is .NET

Definition

.NET is a free, general-purpose, open-source, and multi-platform *programming ecosystem*

programming ecosystem — as it comprehends several languages, compilers, tools, libraries, etc.

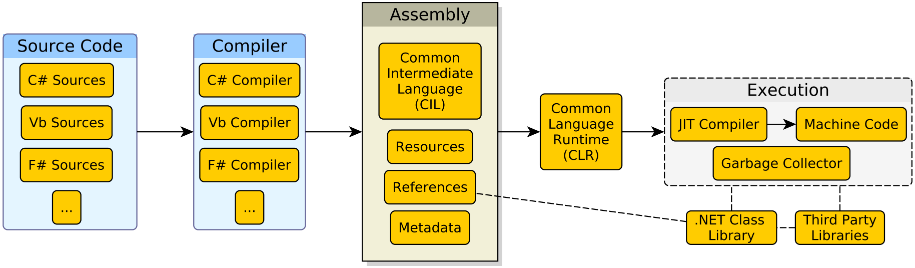
multi-platform — as it can be used on several OS and architectures (e.g. Win, Linux, MacOs, Android, etc)

open-source — as its source code is publicly available and openly licensed

general-purpose — as it supports several sorts of applications (e.g. desktop, mobile, web, videogames, databases, etc)

free — as it is can be exploited with no additional costs

.NET in a Nutshell I



.NET in a Nutshell II

1. Sources written using disparate languages (e.g. C#, Vb, F#, etc.)
2. can be compiled, via as many compilers
3. into assemblies containing

common intermediate language (CIL)	— a language- and platform-agnostic, compiled version of the sources
references	— dependencies declarations for the assembly
resources	— non-code files (e.g. internationalization strings, icons, default configurations, etc.)
metadata	— for identifying the specific *version* of the assembly

4. which can then be executed by the common language runtime (CLR)
   - essentially, an *interpreter* for the CIL
5. on any platform, via a just in time (JIT) conversion into machine code.

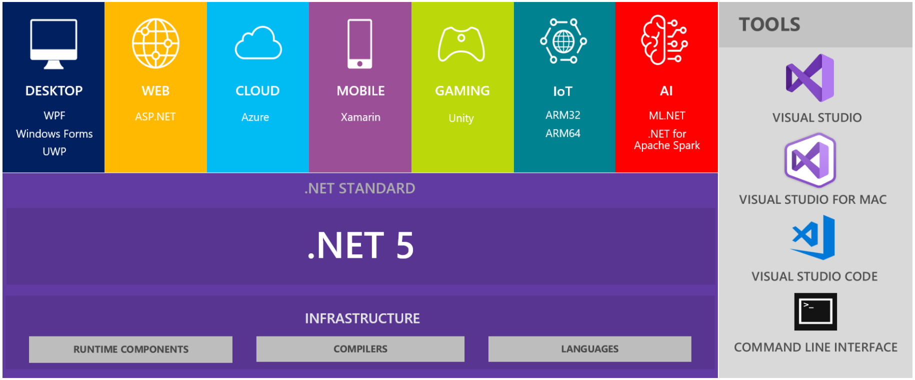
.NET in a Nutshell III

6. Execution of .NET code is *managed* by default, meaning that:
    - developers must not take care of allocating/freeing memory
    - as a garbage collector is in charge of dynamically taking care of that

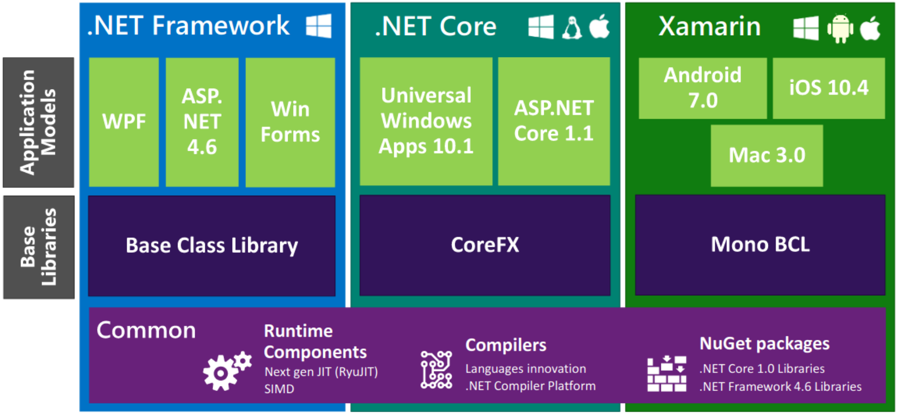
7. .NET programs may reference (a.k.a. depend upon) other assemblies, such as
    - the .NET class library, containing the standard SDK
    - third party libraries, either locally or remotely available

    and therefore exploit any class therein contained.

.NET Platform – The Present



.NET Platform – The Past



.NET Platform – Present vs. Past

- Before .NET 5 there used to be three major implementations of the *class library*:

.NET Framework	— Windows-specific, full-featured, targetting desktop and web applications
.NET Core	— multi-platform (Win, Mac, Linux), less-featured, targetting desktop and web applications
Xamarin	— mobile-oriented (Android, iOS, Mac OS)

- Since .NET 5, implementations are aligned

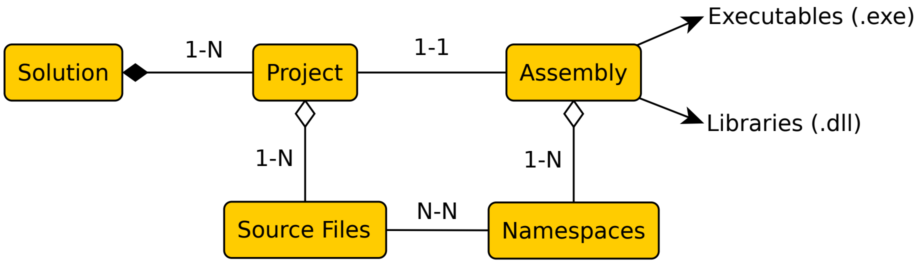
In this course

We stick to .NET Core 6.0, to maximise interoperability and to avoid compatibility issues

Outline

Next In Line...

Overview about Code Base Organization I



1. A .NET code base is called solution
2. Each solution is a set of one or more related projects
   - each project can contain sources written using a single .NET language
   - different projects may target different .NET languages
   - each project esplicitly targets one *application model*
     - eg Class Library, Console/WinForms/WPF/Web Application, etc.

Overview about Code Base Organization II

3. Each project is compiled into an assembly
    - assemblies can either be executable or not—i.e. they can be *libraries*
    - executable assemblies have the `.exe` extension
    - library assemblies have the `.dll` extension
4. Each project is a container of several source files
    - containing several classes, structures, interfaces, or delegates definitions
    - possibly organised into a number of namespaces
5. Therefore, each assembly may *expose* a number of namespaces, along with their definitions

Takeway

Assemblies (and therefore projects) are *deployment* and *execution* units

Code Base Organization Enforcement

- Tools (such as IDEs) *enforce* such code base organization
- You can expect all .NET-enabled tools to stick to this organization
  - eg Visual Studio (VS) or JetBrain Rider
- Analogies exists with other IDEs:

	VS/Rider	**Eclipse**	**Idea**
Code Base	Solution	Workspace	Project
Deployment Unit	Project	Project	Module

Next In Line. . .

Canonical Directory Structure of a .NET Solution I

```
SolutionName/                          solution root dir
├── Project1Name/                      Project1Name root dir
│   ├── Project1Name.csproj            Project1Name metadata (it's a C# project!)
│   ├── bin/                           destination dir for assemblies
│   ├── obj/                           destination dir for temp compilation files
│   ├── SubNamespace1/
│   │   └── SubSubNamespace1/
│   │       └── SomeClassDefinition1.cs nested definition
│   └── Program.cs                     entry point of Project1Name
├── Project2Name/                      Project2Name root dir
│   ├── Project2Name.vbproj            Project2Name metadata (it's a Vb project!)
│   ├── bin/
│   ├── obj/
│   ├── SubNamespace2/
│   │   └── SubSubNamespace2/
│   │       └── SomeClassDefinition2.vb nested definition
│   ├── SomeOtherDefinition.vb         root-level definition
│   └── Program.vb                     entry point of Project2Name
├── ...                                other projects
└── SolutionName.sln                   metadata of the whole solution
```

Canonical Directory Structure of a .NET Solution II

- Each solution *S* has its own directory – named *S* – containing:
  - a single *S*`.sln` file
  - a sub-directory for each project

- Each project *P* has its own directory – named *P* –, within the solution directory, containing:
  - a single *P*`.csproj` file (or *P*`.vbproj` for Vb projects)
  - a directory for each namespace
    - possibly containing other sub-namespaces (and their directories)
    - containing `.cs` source files (or `.vb` for Vb projects)
  - two directories, namely `bin/` and `obj/`, automatically generated
  - some root-level `.cs` source files (or `.vb` for Vb projects)

- Conventionally, *executable* projects contain a root-level `Program.cs` (or `.vb`) file
  - containing a `Main` method which is the entry proint of the program

Example of `.sln` File I

```
Microsoft Visual Studio Solution File, Format Version 12.00
Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") = "GreetingsLib", "GreetingsLib\
     GreetingsLib.csproj", "{A2010235-65D9-475D-A870-5D7AE6FB722F}"
EndProject
Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") = "GreetingCLI", "GreetingCLI\
     GreetingCLI.csproj", "{D9218FDA-9CBB-44ED-925D-4690EAD32428}"
EndProject
Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") = "ExampleProject", "ExampleProject\
     ExampleProject.csproj", "{8B2DA644-0577-45D0-B768-E1AB9F8FACBC}"
EndProject
Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") = "CodingConventionsExample", "
     CodingConventionsExample\CodingConventionsExample.csproj", "{1140EED2-83B9-4F06-A59D
     -4C36E6781A93}"
EndProject
Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") = "Snippets", "Snippets\Snippets.csproj
     ", "{117EC99C-8522-4F20-B72A-BBBD22C33CB7}"
EndProject
Global
  GlobalSection(SolutionConfigurationPlatforms) = preSolution
    Debug|Any CPU = Debug|Any CPU
    Release|Any CPU = Release|Any CPU
  EndGlobalSection
  GlobalSection(ProjectConfigurationPlatforms) = postSolution
    {A2010235-65D9-475D-A870-5D7AE6FB722F}.Debug|Any CPU.ActiveCfg = Debug|Any CPU
    {A2010235-65D9-475D-A870-5D7AE6FB722F}.Debug|Any CPU.Build.0 = Debug|Any CPU
    {A2010235-65D9-475D-A870-5D7AE6FB722F}.Release|Any CPU.ActiveCfg = Release|Any CPU
    {A2010235-65D9-475D-A870-5D7AE6FB722F}.Release|Any CPU.Build.0 = Release|Any CPU
    {D9218FDA-9CBB-44ED-925D-4690EAD32428}.Debug|Any CPU.ActiveCfg = Debug|Any CPU
    {D9218FDA-9CBB-44ED-925D-4690EAD32428}.Debug|Any CPU.Build.0 = Debug|Any CPU
```

Example of `.sln` File II

```
    {D9218FDA-9CBB-44ED-925D-4690EAD32428}.Release|Any CPU.ActiveCfg = Release|Any CPU
    {D9218FDA-9CBB-44ED-925D-4690EAD32428}.Release|Any CPU.Build.0 = Release|Any CPU
    {8B2DA644-0577-45D0-B768-E1AB9F8FACBC}.Debug|Any CPU.ActiveCfg = Debug|Any CPU
    {8B2DA644-0577-45D0-B768-E1AB9F8FACBC}.Debug|Any CPU.Build.0 = Debug|Any CPU
    {8B2DA644-0577-45D0-B768-E1AB9F8FACBC}.Release|Any CPU.ActiveCfg = Release|Any CPU
    {8B2DA644-0577-45D0-B768-E1AB9F8FACBC}.Release|Any CPU.Build.0 = Release|Any CPU
    {1140EED2-83B9-4F06-A59D-4C36E6781A93}.Debug|Any CPU.ActiveCfg = Debug|Any CPU
    {1140EED2-83B9-4F06-A59D-4C36E6781A93}.Debug|Any CPU.Build.0 = Debug|Any CPU
    {1140EED2-83B9-4F06-A59D-4C36E6781A93}.Release|Any CPU.ActiveCfg = Release|Any CPU
    {1140EED2-83B9-4F06-A59D-4C36E6781A93}.Release|Any CPU.Build.0 = Release|Any CPU
    {117EC99C-8522-4F20-B72A-BBBD22C33CB7}.Debug|Any CPU.ActiveCfg = Debug|Any CPU
    {117EC99C-8522-4F20-B72A-BBBD22C33CB7}.Debug|Any CPU.Build.0 = Debug|Any CPU
    {117EC99C-8522-4F20-B72A-BBBD22C33CB7}.Release|Any CPU.ActiveCfg = Release|Any CPU
    {117EC99C-8522-4F20-B72A-BBBD22C33CB7}.Release|Any CPU.Build.0 = Release|Any CPU
  EndGlobalSection
EndGlobal
```

! This is not somithing a developer may manually write!

Example of `.csproj` File

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

    <PropertyGroup>
        <OutputType>Exe</OutputType>
        <TargetFramework>net6.0</TargetFramework>
    </PropertyGroup>

    <ItemGroup>
      <ProjectReference Include="..\GreetingsLib\GreetingsLib.csproj" />
    </ItemGroup>

</Project>
```

! This is not something a developer may comfortably manipulate!

Next In Line. . .

Handling a Solution via the Command Line I

The `dotnet` tool

- All modern .NET installations comprehend a command-line tool named `dotnet`
- It is the simpler way to handle a solution *without* an IDE
- Among the many functionalities, it allows developers to:
  1. create a solution
  2. add projects to a solution
  3. compile a project into an assembly
  4. execute an executable assembly
  5. etc.

Handling a Solution via the Command Line II

How to use the dotnet tool

- General syntax (square brackets denote optionality):

```
$ dotnet [sdk-options] [command] [command-options] [arguments]
```

- How to learn how to use `dotnet`:
  - Run `dotnet [command] --help`
  - See https://docs.microsoft.com/dotnet/core/tools/dotnet

Handling a Solution via the Command Line III

How to create & manage a a solution with via dotnet

1. Create a directory for the solution, say `MySolution`
2. Open a shell into that directory
3. Create an empty `.sln` file named after the current directory (i.e. `MySolution`):

   ```
   $ dotnet new sln
   ```

4. Create a C# console app. project named `MyConsoleProject`:

   ```
   $ dotnet new console -n MyConsoleProject -o MyConsoleProject
   ```

   (where `-n` indicates the project name, and `-o` its relative path)
5. Register `MyConsoleProject` into `MySolution.sln`:

   ```
   $ dotnet sln add MyConsoleProject/MyConsoleProject.csproj
   ```

   (recall to use '`\`' instead of '`/`' on Windows systems)
6. Run `MyConsoleProject` (re-compilation is implicit):

   ```
   $ dotnet run --project
     MyConsoleProject/MyConsoleProject.csproj
   ```

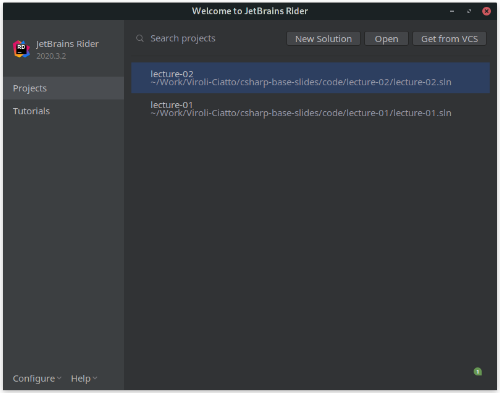
Handling a Solution via the IDE I

Disclaimer

- Using the command line may be hard, but it is free
- IDEs (e.g. Rider or VS) may be used instead, but they require a license
- We provide instructions for Rider, as it is multi-platform
- Apart for the appearance, VS is functionally very similar to Rider
    - as they rely on the same abstractions (solution, project, assembly, etc.)

Handling a Solution via the IDE II

1. Open your IDE



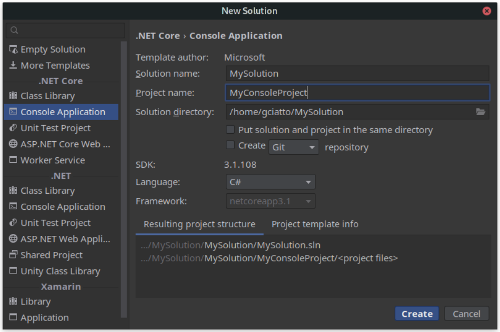
lecture-02
~/Work/Viroli-Ciatto/csharp-base-slides/code/lecture-02/lecture-02.sln

lecture-01
~/Work/Viroli-Ciatto/csharp-base-slides/code/lecture-01/lecture-01.sln

(if you cannot see the welcome dialog above, then click on 'File' → 'New. . . ' to proceed)

Handling a Solution via the IDE III

2. Click on ‘New Solution’



New Solution

.NET Core › Console Application

Template author: Microsoft
Solution name: MySolution
Project name: MyConsoleProject
Solution directory: /home/gciatto/MySolution
☐ Put solution and project in the same directory
☐ Create Git repository
SDK: 3.1.108
Language: C#
Framework: netcoreapp3.1

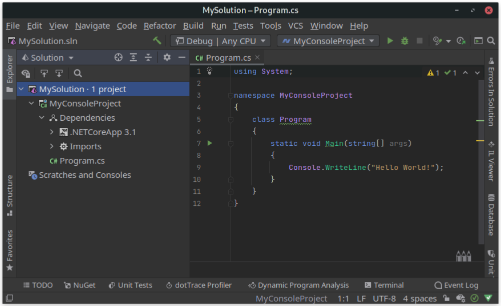
Resulting project structure | Project template info

```
.../MySolution/MySolution/MySolution.sln
.../MySolution/MySolution/MyConsoleProject/<project files>
```

2.1 Choose the *‘Console Application’* template from the *‘.NET Core’* group
2.2 Set the *‘Solution Name’* to ‘`MySolution`’
2.3 Set the *‘Project Name’* to ‘`MyConsoleProject`’
2.4 Ensure the *‘Solution Directory’* ends with ‘`MySolution`’
2.5 Press the ‘Create’ button

Handling a Solution via the IDE IV

3. The IDE should now appear like this:

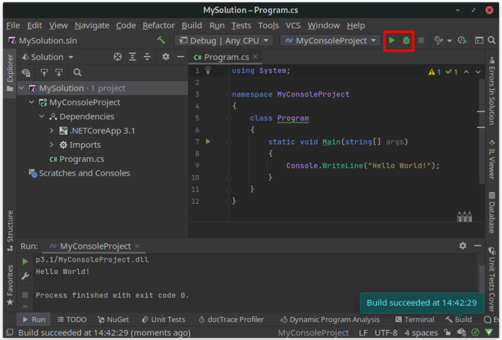


```
using System;

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

- The *Solution Explorer* view (here on the left) shows all the projects, along with their source files, dependencies (a.k.a. references), and resource files
- You can use that to browse the code base or you can Ctrl+Click any symbol of any source file to jump to its definition

Handling a Solution via the IDE V

4. You may use the 'Play' or 'Bug' button to run a project



```
using System;

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

```
p3.1/MyConsoleProject.dll
Hello World!

Process finished with exit code 0.
```

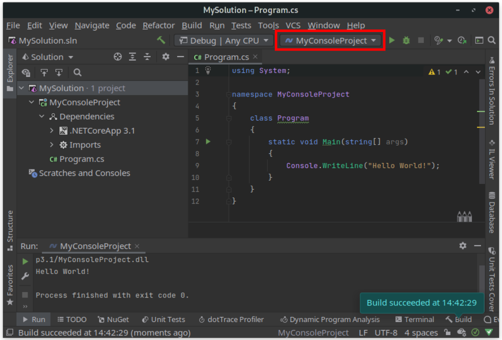
Build succeeded at 14:42:29

Build succeeded at 14:42:29 (moments ago)

- The output of the program appears either below (Rider) or into a new window (VS)
- The 'Play' appears close to the `Main` method as well

Handling a Solution via the IDE VI

5. Which project is actually run when you press ‘Play’ depends on which project is currently selected:



```
using System;

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

```
p3.1/MyConsoleProject.dll
Hello World!

Process finished with exit code 0.
```

Build succeeded at 14:42:29

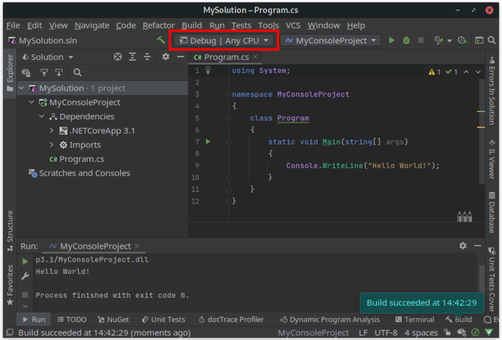
Build succeeded at 14:42:29 (moments ago)

- Only *executable* projects can be selected on that menu
- Recall that each executable project has a single *entry point*

    ie its `static void Main` method

Handling a Solution via the IDE VII

6. Assemblies may be compiled either in 'Debug' or in 'Release' mode:


MySolution – Program.cs
File Edit View Navigate Code Refactor Build Run Tests Tools VCS Window Help
MySolution.sln
Debug | Any CPU
MyConsoleProject
Solution
MySolution · 1 project
MyConsoleProject
Dependencies
.NETCoreApp 3.1
Imports
Program.cs
Scratches and Consoles
Run: MyConsoleProject
Build succeeded at 14:42:29
Run TODO NuGet Unit Tests dotTrace Profiler Dynamic Program Analysis Terminal Build
Build succeeded at 14:42:29 (moments ago)
MyConsoleProject LF UTF-8 4 spaces

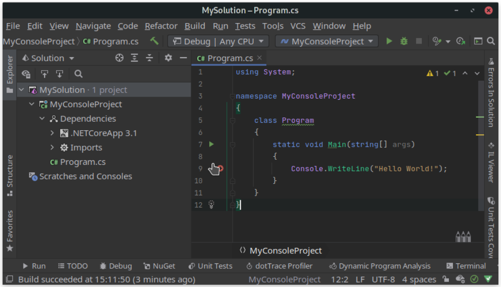
```
using System;

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

```
p3.1/MyConsoleProject.dll
Hello World!

Process finished with exit code 0.
```

- When compiled in 'Release' mode, optimisation are performed, which makes step-by-step debugging hard as some instructions may be pruned
- When compiled in 'Debug' mode, no optiomisation is performed

Handling a Solution via the IDE VIII

7. You may click close to a code line to set up a breakpoint on that line



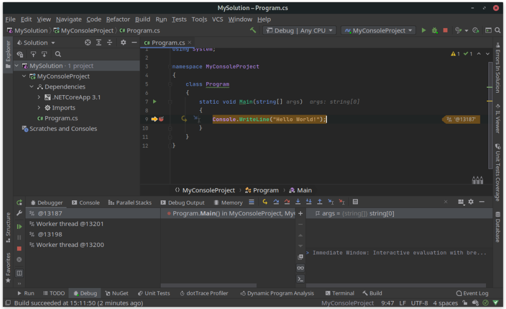
```
using System;

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

Build succeeded at 15:11:50 (3 minutes ago)

- breakpoints are *ignored* when launching the program with ‘Play’
- breakpoints may suspend a program execution, if it has been launched via the ‘Bug’ button—i.e. in debug mode

Handling a Solution via the IDE IX

8. While in debug mode, the program execution is suspended whenever the program reaches a break point



```
using System;

namespace MyConsoleProject
{
    class Program
    {
        static void Main(string[] args)  args: string[0]
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

```
@13187
Worker thread @13201
@13198
Worker thread @13200

Program.Main() in MyConsoleProject, My

args = {string[]} string[0]

> Immediate Window: Interactive evaluation with bre...
```

Build succeeded at 15:11:50 (2 minutes ago)

- while suspended, you may inspect the current status of a program execution

  eg variables values, content of objects, call stacks, etc.

- you may also make the program proceed step-by-step

Visual Studio Code

- Microsoft's official .NET Extension pack

Outline

About Coding Style Conventions I

Definition

- Stylistic rules about how to write good-quality code
  - eg fields/properties/methods/types names, usage of white spaces, empty lines, comments, etc.
- Usually followed by all developers in a given community / project
- Possibily enforced by *automatic* tools
- Aimed at easing:
  - interoperability among developers,
  - readability of the source code, and
  - the maintainability of the code base

About Coding Style Conventions II

Coding Style Conventions are **Important**

- Sticking to some shared coding style is *fundamental*
  - especially, when working in teams
- Even if you are working alone, people may eventually join the project, forming a team
  - → always act like you are in a team, even when coding alone
- Coding conventions are not a matter of taste
  - do not ignore some convention just because it appears ugly to you
  - conventions are never ugly/beautiful, nor right/wrong
  - conventions are important as they are shared
- One everybody gets used to conventions, they easy developers' understanding of the code
  - eg naming conventions help understanding what a symbol is without need to see its definition
  - → violating a convention is harmful: it misleads code readers

About Coding Style Conventions III

C# Coding Conventions

We stick to the conventions enumerated here:

`https://github.com/dotnet/runtime/blob/main/docs/coding-guidelines/coding-style.md`

→ we provide an overview of most relevant conventions in the next slides

C# Coding Style I

Concatenated Words Styles around the World

`camelCase`	— `https://en.wikipedia.org/wiki/Camel_case`
`PascalCase`	— like `camelCase` but the first letter is uppercase
`snake_case`	— `https://en.wikipedia.org/wiki/Snake_case`
`kebab-case`	— `https://it.wikipedia.org/wiki/Kebab_case`

! .NET conventions mostly rely on `camelCase` and `PascalCase`

C# Coding Style II

Suggested Naming Conventions for C#

namespaces names are in `PascalCase`

type names (classes, interfaces, structures, delegates) are in `PascalCase`

eg `String`, `List`, `Int32`, `Action` etc.

inteface names start with a '`I`' and are in `PascalCase`

eg `IList`, `ISet`, `IDictionary` etc.

abstract class names start with '`Abstract`' and are in `PascalCase`

field names start with a '`_`' and are in `camelCase`

method names are in `PascalCase`

property names are in `PascalCase`

local variables and **methods parameters** names are in `camelCase`

! all names are in English

C# Coding Style III

Suggested Bracing Conventions for C#

C# bracing style is Allman's one[a]

- braces are always mandatory, except in single-line `if/else` bodies
- open and closed braces always lay their own line
- indentation levels of open/closed braces is the same of the clause they belong to
- statements within braces are subject to 4-spaces indentation

! this style is different from Java's and JavaScript's ones

```
if (/* ... */)
{
    // something
}
```

[a] (cf. https://en.wikipedia.org/wiki/Indentation_style#Allman_style)

C# Coding Style IV

Suggested White Space Conventions for C#

- Indentation exploits 4 spaces instead of tabulations
  - you may need to enable white characters visualization to spot the difference
- A space is mandatory before and after each infix operator

  ie arithmetic, boolean, bitwise, comparison operators, etc.

  eg `‘a + b’` is ok, `‘a!=b’` is not ok
- Commas require no space before and a single space after

  eg `‘a, b’` is ok, `‘a , b’` or `‘a,b’` are not ok
- Semicolons require no space before
- Constructs require a single space within name and round parenthesis opening

  eg `‘if (...’`, `‘while (...’` or `‘for (...’` are ok

C# Coding Style V

Other Suggested Conventions for C#

- Define variables with `var` only if the type is obvious and evident in that context
- Use keywords instead of full names for built-in types (e.g., `int` instead of `System.Int32`)
- Define `readonly` variables/fields/properties whenever possible (cf. `final` in Java)
- Always specify visibility modifiers explicitly
- Order classes members as follows (top-down):
  1. fields
  2. constructors
  3. properties (public first)
  4. methods (public first)
  5. static members

C# Coding Style VI

Suggested File/Directory Organization for .NET

- One type definition (class, struct, enum, delegate, ...) per file
  - so that developers may easily locate definitions
- Name the file after the type definition it carries

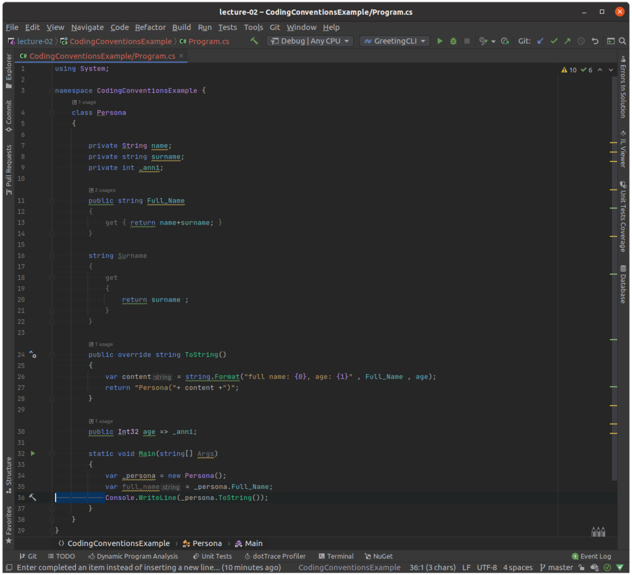
  eg `class Person` defined into `Person.cs` (or `Person.vb`)
- Each project exposes a root namespace named after it

  eg the root namespace of project `P` is named `P`
- All source files from a project root directory contain type definitions laying within that project root namespace

  eg the file `P/Person.cs` defines class `Person` within namespace `P`
- Sub-directories reflect sub-namespaces

  eg the file `P/People/Person.cs` defines class `Person` within namespace `P.People`

C# Coding Style – Can you spot all the problems?

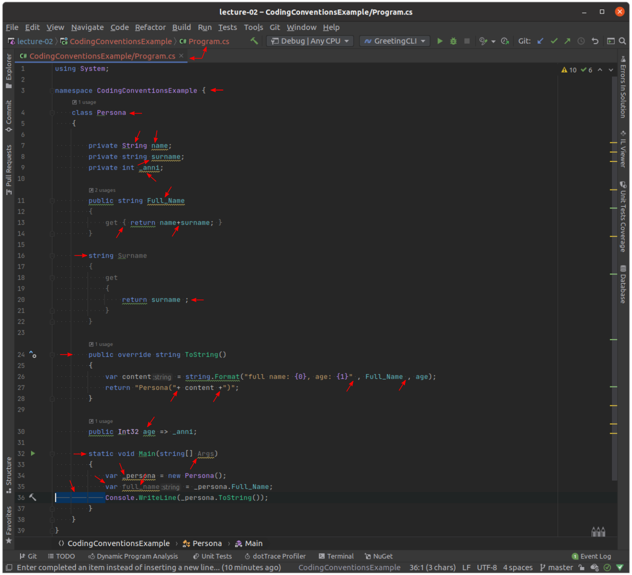


lecture-02 – CodingConventionsExample/Program.cs

```csharp
using System;

namespace CodingConventionsExample {
    class Persona
    {

        private String name;
        private string surname;
        private int _anni;

        public string Full_Name
        {
            get { return name+surname; }
        }

        string Surname
        {
            get
            {
                return surname ;
            }
        }

        public override string ToString()
        {
            var content = string.Format("full name: {0}, age: {1}" , Full_Name , age);
            return "Persona("+ content +")";
        }

        public Int32 age => _anni;

        static void Main(string[] Args)
        {
            var _persona = new Persona();
            var full_name = _persona.Full_Name;
            Console.WriteLine(_persona.ToString());
        }
    }
}
```

Enter completed an item instead of inserting a new line... (10 minutes ago)   CodingConventionsExample   36:1 (3 chars)   LF   UTF-8   4 spaces   master

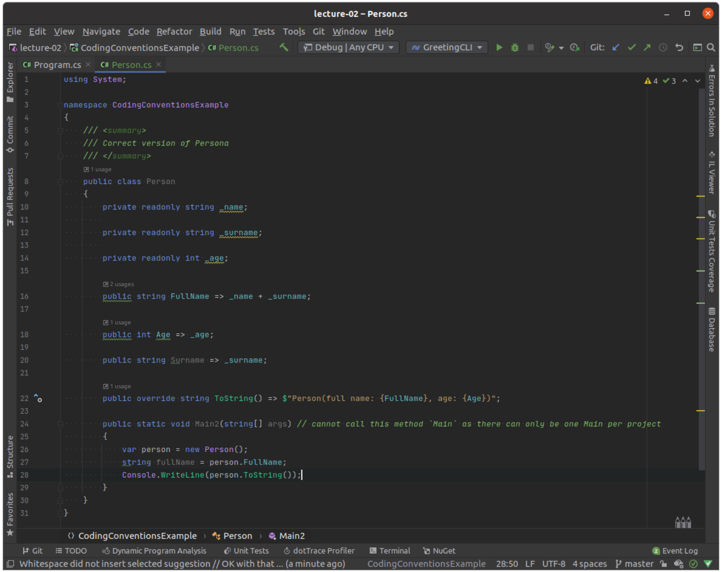
C# Coding Style – Can you name all the problems?



```
using System;

namespace CodingConventionsExample {
    class Persona
    {

        private String name;
        private string surname;
        private int _anni;

        public string Full_Name
        {
            get { return name+surname; }
        }

        string Surname
        {
            get
            {
                return surname ;
            }
        }

        public override string ToString()
        {
            var content = string.Format("full name: {0}, age: {1}" , Full_Name , age);
            return "Persona("+ content +")";
        }

        public Int32 age => _anni;

        static void Main(string[] Args)
        {
            var _persona = new Persona();
            var full_name = _persona.Full_Name;
            Console.WriteLine(_persona.ToString());
        }
    }
}
```

Enter completed an item instead of inserting a new line... (10 minutes ago)

C# Coding Style – Correct Version


lecture-02 – Person.cs
File Edit View Navigate Code Refactor Build Run Tests Tools Git Window Help
lecture-02 〉 CodingConventionsExample 〉 Person.cs
Debug | Any CPU
GreetingCLI
Git:
Program.cs
Person.cs
Explorer
Commit
Pull Requests
Structure
Favorites
Errors In Solution
IL Viewer
Unit Tests Coverage
Database
CodingConventionsExample 〉 Person 〉 Main2
Git TODO Dynamic Program Analysis Unit Tests dotTrace Profiler Terminal NuGet Event Log

```csharp
using System;

namespace CodingConventionsExample
{
    /// <summary>
    /// Correct version of Persona
    /// </summary>
    public class Person
    {
        private readonly string _name;

        private readonly string _surname;

        private readonly int _age;

        public string FullName => _name + _surname;

        public int Age => _age;

        public string Surname => _surname;

        public override string ToString() => $"Person(full name: {FullName}, age: {Age})";

        public static void Main2(string[] args) // cannot call this method `Main` as there can only be one Main per project
        {
            var person = new Person();
            string fullName = person.FullName;
            Console.WriteLine(person.ToString());
        }
    }
}
```

Whitespace did not insert selected suggestion // OK with that ... (a minute ago)   CodingConventionsExample   28:50   LF   UTF-8   4 spaces   master